Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor's Thesis

Design and Implementation of a Measurement Framework for Time-Sensitive Networks

Valentin Simon Nepomuk König

Course of Study:	B.Sc. Informatik
Examiner:	Prof. Dr. rer. nat. Kurt Rothermel
Supervisor:	David Hellmanns, M.Sc.
Commenced:	October 7, 2020
Completed:	April 7, 2021

Abstract

Networks in industrial control and automation applications must be capable of providing highly predictable delivery, both in terms of latency and latency variation. Thus, a network must guarantee deterministic behavior for time-sensitive traffic. Currently, dedicated and highly engineered networking solutions such as field buses are commonly used for time-critical traffic in industrial applications. Current trends in network convergence combine mixed-criticality traffic flows into the same real-time-capable networks. Currently, the IEEE Time-Sensitive Networking (TSN) Task Group are augmenting the widely supported Ethernet standard to provide time-critical flows alongside best-effort traffic. For its capabilities in high bandwidth, mixed-application and real-time-support, TSN is expected to be widely adapted in industries. In TSN, many of the guarantees on deterministic behavior have their basis in a traffic scheduling mechanism called Time-Aware Shaper (TAS). The development and integration of TSN-capable networks require increasingly powerful diagnosis and measurement tools. However, no such framework is publicly available at this point. The focus of this thesis is the development of a framework fulfilling requirements for performance analysis of physical networks with real-time constraints. This framework enables high-performance measurement tasks for TSN, which makes measurable the inaccuracies of physical network devices. As a first result of this insight, we show that many of the assumptions researchers often make about TSN networks are simplified and do not properly model the real world. Moreover, scheduling algorithms for the TAS often disregard the same inaccuracies in the synthesis of the traffic schedule. This could render invalid the guarantees on determinism for out-of-the-box scheduling. We propose a mechanism to improve the determinism of low-quality senders and the overall network bandwidth utilization, proving its effectiveness in a proof-of-concept setup.

Contents

Acronyms

API application programming interface. 34

AVB Audio Video Bridging. 18

BC Boundary Clock. 20

CBS Credit Based Shaper. 18

CM configuration manager. 37

CRC cyclic redundancy check. 12

DEI drop eligible identifier. 12

EBNF extended Backus-Naur form. 83

ETF earliest transmit time first. 39

FCS Frame Check Sequence. 12

FPGA Field Programmable Gate Array. 15

GCL Gate Control List. 10

HAL Hardware Abstraction Layer. 34

IFG Inter-Frame Gap. 12

IIoT Industrial Internet of Things. 9

IT Information Technology. 9

LAN Local Area Network. 11

LDP Link Discovery Protocol. 46, 47

LLC logical link control. 11

MAC medium access control. 11, 12

NIC network interface card. 20

OM output manager. 40

OSI Open Systems Interconnection. 11

OT Operational Technology. 9

PCP Priority Code Point. 12

PPS Pulse-Per-Second. 20

PTP Precision Time Protocol. 20

QJC Queue Jitter Containment. 74

SFD Start Frame Delimiter. 11

SFP small form-factor pluggable. 16

SOF Start of Frame. 12

TAS Time-Aware Shaper. 3, 10

TC Transparent Clock. 20

TCI tag control information. 12

TDMA Time-division Multiple Access. 19

TPID tag protocal identifier. 12

TSA transmission selection algorithm. 18

TSN Time-Sensitive Networking. 3, 9

TT Time-Triggered. 10

VID VLAN identifier. 12

VLAN Virtual Local Area Network. 12

1 Introduction

Beginning in the 1970s, control and monitoring systems of the industrial automation technologies started to move towards a more centralized approach, revolutionizing the way industries supervise and control their automation technologies [8]. Now, with the rise of Industry 4.0 and the Industrial Internet of Things (IIoT), the world undergoes a new industrial revolution in the information age. However, this requires powerful real-time capable networking solutions, to fulfill the industries' requirements. In industrial automation, for instance, there are hard time deadlines that have to be met, a network has to be able to provide guarantees on real-time deterministic delivery. Due to this rapidly growing importance of real-time communication, a variety of vendor-specific solutions were developed to satisfy these requirements. This caused industries to deploy a wide range of different networking solutions that are mutually incompatible. Architects of large industrial networks were often even required to roll out multiple different systems in parallel [23]. Customers of such vendor-specific solutions are locked into a product portfolio, further reducing flexibility and interoperability [11, 30].

As an alternative to many of these vendor-specific protocols, the IEEE are currently working on extending the well-established Ethernet standard to create an openly standardized solution called Time-Sensitive Networking (TSN). Open standardization of TSN allows component builders, system integrators and hardware or software vendors to work together, creating an inter-operable network design [11]. This interoperability holds a great advantage of TSN over traditional state-of-the-art networking systems. Hence, Ethernet using TSN already has a very good chance to unify industrial networking solutions [23]. Another advantage over problem-specific networking solutions is TSN's capability to deliver time-sensitive communication along with best-effort services [7, 17, 23]. This would allow industries to deploy a single physical TSN-based network for many communicating systems, each with different requirements. Most importantly, it would allow the convergence of Operational Technologies (OT) and Information Technologies (IT). OT are used to control and monitor physical processes, these systems require deterministic networks with real time-capable end systems [8]. Currently, OT networks are in most cases realized using vendor-specific field buses. IT, on the other hand, are used in cloud computing or other high-level applications and often require high bandwidth networks, rather than real-time communication. Despite the very different communication requirements, TSN would be capable of transporting both alongside each other over the same network [21, 23]. Hence, TSN is enabling the IT/OT convergence, which is widely considered to be a major step towards IIoT and Industry 4.0 [30].

However, the convergence of IT and OT further accelerates network growth and interconnectedness, increasing complexity significantly. With rising network complexity comes the need for more powerful diagnosis tools. Specifically the capability for real-time analysis is vital for the development and adaptation of TSN. As the TSN family of standards is relatively new, no such diagnosis tools are currently widely available. Researchers in particular often had to rely on network simulators for their individual testing scenarios [6, 21, 26, 28, 35]. While a simulation can give very detailed insight into a network, it is difficult to verify functional aspects of the real world. The simulations often only

consider simplified scenarios that do not accurately reflect the real world. Now, more TSN-capable hardware is becoming available and the industry's interest in transitioning to TSN is increasing [3]. This makes it necessary to validate the simulated results in the real world. Though, performing accurate measurements in real-time networks is a non-trivial task; both the measurement hardware and software need to operate with high enough accuracy and precision to evaluate the network functionality. To the best of our knowledge, no publicly available diagnosis framework currently exists that is capable of high precision measurements in real-time networks. This comprises a significant research gap in the field of real-time communication. In this thesis, we fill this research gap by providing a measurement framework for real-time-capable Time-Triggered (TT) networks. Using this framework, we perform first validation measurements, comparing simulated results to real world tests.

With the new TSN standards as a main motivation, this thesis is concerned with IEEE 802.3 Ethernet networks. For TSN, many of the real-time capabilities and determinism guarantees have their basis on the Time-Aware Shaper (TAS). This is a TT traffic shaper, working on a pre-calculated schedule, the so-called Gate Control List (GCL). Such a schedule provides time slots for each important time-sensitive communication. We provide more detailed technical background in Chapter 2. Many researchers have already shown in simulation that TSN can guarantee in-time delivery, if these time slots are chosen correctly [21, 28, 35]. This related work is presented in Chapter 3. Chapter 4 defines the aforementioned research gap and specifies requirements that need to be met by our developed framework. First, the framework should be able to emulate transmitting end-point behavior, acting as an high-precision network stimulus. These stimuli are required to allow evaluation and verification of the timing behavior of the system. Secondly, the developed framework should be capable of simultaneously capturing and time stamping traffic on multiple measuring points in the network. Lastly, the collected data has to be aggregated and processed, allowing for interpretation of the results. In Chapter 5 we present our system model. We define a topology model and a delay model to describe the timing of traffic in realistic networks. We use these models in the design and evaluation parts of this thesis. We design the framework's architecture in Chapter 6, based on our requirements. The implementation details are specified in Chapter 7. In Chapter 8, we verify the fulfillment of the requirements and use our framework to present difficulties that can arise in real world networks. We verify the general functionality of the TAS, and take a closer look at pitfalls and problems that can be hidden in simulation but become important in a physical network setup. Using this knowledge, we give brief insight into a solution that can achieve improved determinism guarantees for high-jitter transmitters. This could allow scheduling algorithms to fulfill very low jitter requirements in the real world.

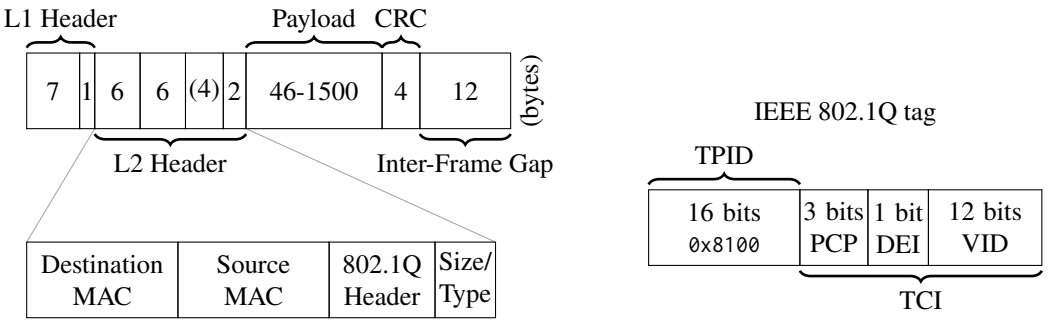
2 Technical Background

Developing a measurement framework for real-time capable networks requires knowledge about the network type, its behavior, and the used measurement hardware. We require precise knowledge about the network to correctly interpret the timing behavior of the stations in the network. Furthermore, knowing and understanding the measurement hardware is important. We are only able to interpret the results correctly, if we understand how they were measured. There are a wide variety of different types of networking solutions available. Due to the fact that this thesis is motivated by TSN, we limit the scope of this thesis to IEEE 802.3 Ethernet based networks. Hence, we first discuss the specific stations in Ethernet networks and how two systems can communicate with each other. Knowing this, we go on with the expected time behavior, explaining the network's delay model used for our measurements. To complete the understanding of all network components over time, we discuss the hardware and connector delays last.

2.1 Ethernet

The IEEE 802.3 standard [14] defines Ethernet as a means of communication in local wired networks, so called Local Area Networks (LANs). It provides services on the Physical (Layer 1) and the Data Link Layer (Layer 2) of the Open Systems Interconnection (OSI) reference model. For the scope of this thesis, we only consider communication up to Layer 2. This reduces the number of active components (stations) in the network to two types. There are *switches* (or bridges), providing the network functionality itself and *end stations* as senders and receivers of data. Connections between stations are called links. Links can either be unidirectional (simplex) or bidirectional (duplex). Two end stations can communicate over the network by sending frames. In the following, we describe the contents of such an Ethernet frame.

If not stated otherwise, we refer to the IEEE 802.3 standard [14]. Figure 2.1a shows the structure of an Ethernet frame. The first seven octets are the preamble field. This field is used to announce an incoming frame and allows the receiver to synchronize to the frame's transmit timing. Next, the Start Frame Delimiter (SFD) marks the begin of the frame. These two fields together are the header of the Physical Layer. The Ethernet specification splits the Layer 2 of the OSI reference model into two sublayers. First, the medium access control (MAC) sublayer is responsible for medium-independent access control to the network or medium. Secondly, the logical link control (LLC) sublayer is situated above the MAC layer, as an access-independent layer used to identifiy and encapsulate Network Layer (Layer 3) protocols and perform error checking. In the frame, the MAC header begins after the SFD mentioned above. A MAC header contains two 6 B long address fields, with the first specifying the destination(s), i. e., the receiver(s) of the frame. The second field contains the address of the sender of the frame.



(a) The structure of an Ethernet frame. The Layer 1 header consists of the preamble and the SOF byte. This is followed by the Layer 2 header with destination and source MAC addresses, the optional IEEE 802.1Q tag, and the EtherType or length field. At the end of the frame, we have the FCS in the form of a CRC, followed by the IFG.

(b) The four bytes of the IEEE 802.1Q tag are split into TPID and TCI. The latter is further divided into PCP, DEI and VID.

Figure 2.1: The structure of an ethernet frame and a more detailed look at the optional IEEE 802.1Q header.

The addresses are optionally followed by the IEEE 802.1Q tag, which is used for Virtual Local Area Network (VLAN) identification according to IEEE 802.1Q [15]. A frame containing this tag is called a Q-tagged or VLAN-tagged frame. Figure 2.1b visualizes the structure of the IEEE 802.1Q tag. This header will be important because TSN uses the Q-tag for example for traffic prioritization. In a Q-tagged frame, the two octets after the source MAC are set to a value of `0x8100`, called the tag protocal identifier (TPID). This value makes it possible to differentiate between tagged and untagged frames. The TPID is followed by an additional two octets called the tag control information (TCI). The first three bits of the TCI are the Priority Code Point (PCP) field, which is used to indicate one of eight levels of priority. This is followed by the one-bit-long drop eligible identifier (DEI). The last twelve bits of the TCI are used as the VLAN identifier (VID), to assign the frame to a VLAN. In tagged and untagged frames, the next field is the EtherType or length field. It is transmitted after the source MAC in the case of an untagged frame or after the Q-tag if it exists. If the value of the field is less than 1,500 (i. e., the maximum payload size), the content represents the payload length. Otherwise, the field contains information about the payload's network protocol called the EtherType. Note that in the case of a tagged frame the TPID is in the place of the EtherType field of an untagged frame, allowing for a differentiation.

After the Layer 2 header, we have the frame's payload of variable size. The overall frame size of a Q-tagged frame is allowed to be between 64 and 1,522 B (including the Layer 2 header). The payload size can thus be in the range of 44 and 1,500 B. After the payload, the Frame Check Sequence (FCS) terminates the frame. For Ethernet frames, a cyclic redundancy check (CRC) is used to check for errors in the frame. On the wire, the Inter-Frame Gap (IFG) is the minimal distance between two frames.

2.2 Network Delay Model

On its path through the network, a frame encounters multiple stages introducing different kinds of delay. Every switch, all end stations and even every link delays the overall transmission of each frame. Understanding and modeling these delays is key for analyzing a network. We present in the following the delay model from Hellmanns et al. [12]. If not declared otherwise, we refer to this paper. To understand the different delays, we follow a frame through a switched network. We omit the time that is required to create the frame and to move it to the sending hardware, but assume that the frame is already ready for transmission. We begin with a sender, serializing the frame on a cable. On the path to the first *hop* (the next switch or end station) we already encounter the first delay.

Definition 2.2.1 (Propagation Delay)
The Propagation Delay *is the time a signal travels from an egress port to the next ingress port. Hence, it is calculated using the cable length and medium-specific propagation speed*

$$d_{\text{pg}} = \frac{\text{cable length}}{\text{propagation speed}}.$$

The propagation delay is the time between the sender beginning to transmit the frame and the next connected station receiving the first bit. The propagation delay can not be avoided because it depends on the physical length and properties of the cable connecting two stations. In our test setup used later on, we use fiber optical links, which results in a propagation delay of about 5 ns per meter of fiber length.

The time it takes to capture the entire frame is exactly the time it takes the sender to serialize the frame onto the wire.

Definition 2.2.2 (Transmission Delay)
The Transmission Delay *is the time required to serialize a frame, i. e., the time from begin to end of transmission. It depends on link speed and frame size,*

$$d_{\text{tr}} = \frac{\text{frame size}}{\text{link speed}}.$$

That means, the next station in our network sees the first bit of the frame after the propagation delay has passed. After that, the transmission delay passes until the station received the entire frame. The frame is now at the next station in the network. We assume that this next station is a network switch. Here, the switch has to determine over which port the incoming frame is forwarded. Hence, the next delay is introduced by the switching logic of the switch. The processing delay models the time, a switch needs to determine the output queue for the received frame.

Definition 2.2.3 (Processing Delay)
The Processing Delay *is the time between end of reception of a frame and the time where this frame is queued for forwarding. In formulas or figures, it will be used as* d_{pc}.

The processing delay is independent of the frame size, but depends on the processing architecture of the switch. It is important to keep in mind that the processing delay of every switch is not a constant value. It can vary significantly from frame to frame; we call this the *Processing Jitter*. Our frame is now in the correct output queue of the switch. Now, there are two possibilities: First, the

forwarding to the next hop can begin immediately, without any additional delay. In the second case, the frame has to wait, i. e., stay queued. This could be due to interference with other frames that are currently in transmission, or have higher priority. The time a frame spends in an output queue is the next delay of our model.

Definition 2.2.4 (Queuing Delay)
The Queuing Delay *is the time, a frame that is ready to be transmitted, must wait until its transmission starts. In formulas or figures it will be used as* d_{qu}.

The queuing delay can be avoided using special techniques that e. g., prevent cross traffic. Figure 2.2 visualizes each of the four delays shown so far, on a network consisting of three switches forwarding a frame A. This frame needs to wait for some cross traffic at the link $(S2, S3)$, thus introducing queuing delay.

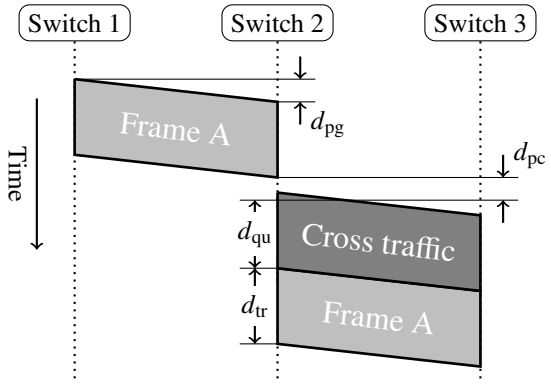


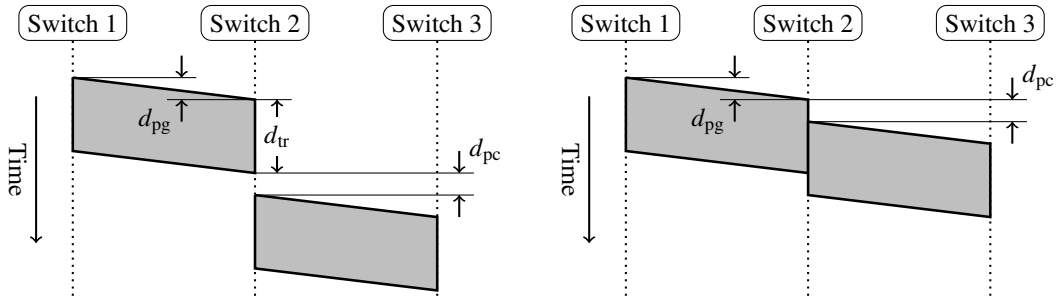
Figure 2.2: Visualization of the delays a frame encounters while being forwarded through a switched network. The frame A is forwarded through a switched network. At Switch 2, some cross traffic is ready for transmission before the processing of frame A is finished. This prevents the immediate forwarding of the frame, causing a queuing delay.

2.2.1 Forwarding Mechanisms

For the delays on the switch, we assumed that the switch has to capture the entire frame before it can begin processing it. However, this is only one possibility for the forwarding behavior. The previously used forwarding behavior is called *Store-and-Forward*. As the name implies, a frame is stored in its entirety before processing and ultimately forwarding begins (cf. Figure 2.3a) [2]. This causes the overall forwarding time of the frame to be dependent on the frame's size. The switch not only determines the egress queue, but also checks the FCS (for Ethernet that is the CRC) and other values, discarding the frame if an error is found. Discarding the frame if it contains an error is part of the IEEE 802.1Q standard [15], with the intention of quickly removing corrupted frames from the network.

In high-performance use-cases, requiring very low latencies between the end stations, the time to capture the entire frame (i. e., the transmission delay) can be saved. A switch can theoretically determine the egress queue as early as it received the first 14 B of the frame, i. e., the destination

MAC address[1]. Hence, a second forwarding mechanism, called *Cut-Through* forwarding, can begin processing the frame while it still being received [2]. Because of this, the delay introduced by the switch can be constant (cf. Figure 2.3b). However, the benefits of this method only come into effect if the egress port is free for transmission. Otherwise the switch is forced to queue the frame.



(a) With the Store-and-Forward mechanism, the entire frame has to be received before processing. Hence, the transmission delay is part of the time it takes to forward the frame.

(b) With Cut-Through forwarding, the switch is able to forward the frame even before it was fully received. The delay introduced by the switch can be independent of the frame size.

Figure 2.3: Visualization of the behavior over time when using the two forwarding mechanisms Store-and-Forward or Cut-Through.

2.3 Hardware and Connector Delays

Up until now, we were only concerned with the delays a frame experiences while in the network. However, for the purposes of measuring the frame's overall delay, we are interested in the time difference from transmission to capturing on two end systems. A problem arises here because we usually do not have access to the exact time at which the network hardware receives or sends a frame. Only using special hardware, we can gain access to accurate transmit or capture time stamps.

On the sender side, we use a feature called *Time Stamp Injection*. High-end network adapters support this feature in hardware, increasing the time stamp accuracy. A network adapter with this feature enabled writes a time stamp in the frame payload itself which indicates the time the frame was actually sent. Due to hardware limitations however, it is not possible to time stamp the exact moment the frame leaves the sender with this injected time stamp. However, on adapters supporting time stamp injection, it is common that deterministic hardware such as Field Programmable Gate Arrays (FPGAs) are used. This way, the inevitable delay between taking the time stamp and begin of transmission can be deterministic and thus, it becomes measurable by the network adapter [25, called "TX Path Delay"].

Definition 2.3.1 (TX Hardware Delay)
We define the time between the network adapter injecting the time stamp in the frame payload and begin of transmission as the TX Hardware Delay*, or* $d_{\text{hw,TX}}$.

[1] This is a theoretical minimum.

For the capture time stamp, a similar argument has to be made. Using a high resolution clock, almost any network adapter can provide a high resolution capture time stamp for a frame. However, this is not enough to actually achieve high precision measurements. Due to the same limitations as on the sender, there is a delay between the first bit arriving and the adapter taking the corresponding capture time stamp. Again, using special capture hardware, this delay can be be deterministic and measurable [25, called "RX Path Delay"].

Definition 2.3.2 (RX Hardware Delay)
On the receiver side, the delay between capturing the frame to taking the corresponding capture time stamp is called the RX Hardware Delay*, or* $d_{\text{hw,RX}}$.

Figure 2.4 shows both the TX and RX hardware delays. Knowing the exact hardware delays is critical to performing precise latency measurements.

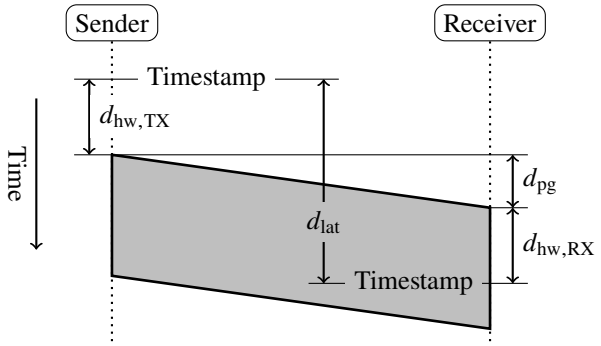


Figure 2.4: The inevitable delays included in the latency measurements, visualized on two directly communicating systems. First, we have the TX hardware delay as the time between injecting the time stamp to transmitting the frame. Secondly, there is the RX hardware delay, as the time between receiving the frame and taking the corresponding time stamp.

We also need to take into account the possible delay introduced by the connection to the physical link medium. We discussed the propagation delay as the delay of a message on the medium. For this, we assumed that a signal can be moved from the hardware to the medium *instantly*. However, this is not always true.

2.3.1 Small Form-Factor Pluggables (SFPs)

The connections between hardware and physical medium of a link can have a significant impact on real-time performance. In high-bandwidth or enterprise-grade networking devices, small form-factor pluggables (SFPs) are most commonly used as transceivers on either side of a link [27]. The popularity has its roots in the medium-independence of the SFP specification. Networking hardware with SFP ports can use a variety of different transceivers, all using the same end station or switch hardware. The SFP transceivers are modular and support several physical mediums e. g., fiber optic or copper twisted pair (BASE-T) connections, at different link speeds [1, 27]. Only the module on the cable determines the physical medium of the link.

This implies that the transceiver has to perform a conversion between a medium-independent signal on the hardware and the medium-specific signal on the link. This conversion can increase the link latency or even introduce additional jitter. Hence, a low-latency and jitter-free medium and SFP combination is required for conducting precise measurements. For our measurements, we used fiber optic transceivers as they do not introduce any latency or jitter because they do not require additional conversion circuitry.

2.3.2 Network Taps

To monitor network traffic, without introducing delay on either some end-station or a bridge, we use breakout network taps. A breakout tap is a device capable of duplicating network traffic between two stations, without any bandwidth limitations [9]. It is added as hardware along the connection and has four connection ports (cf. Figure 2.5). Two of those are full-duplex ports as pass-through for the original communication. The other two ports are simplex monitoring connections, each carrying a duplicate of one communication direction between the full-duplex ports. One monitoring port for each direction is required to carry the full bandwidth. Taps that combine both traffic directions into a single monitoring port exist as well, they are called aggregation taps [10]. However, they were not used in this work.

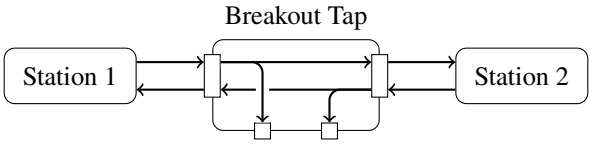


Figure 2.5: A breakout tap with its four connection ports. Two ports are full-duplex, carrying the data between Station 1 and Station 2. The other two ports are simplex monitoring ports, carrying a duplicate of either communication direction between the stations.

Theoretically, software tapping is also possible and would allow a far greater number of monitoring points in the system, e. g., on every device and used port. However, software taps introduce computation overhead on the communicating devices and therefore influence the network devices, possibly changing the network behavior during measurement. This would negatively impact the time-accuracy of the measurement setup. Furthermore, a hardware network tap creates an exact duplicate of the traffic and includes the regular traffic as well as possible network errors that might be omitted by a software tap [9].

For the measurements conducted here, we used only fiber optic connections and passive fiber optic network taps. This type of network tap is optimal for high-precision measurement purposes, as it does not introduce any measurable latency on the connection between two communicating devices while producing an exact and time-accurate duplicate of the communication [9]. This is done by splitting the optical signal using a form of a beam splitter. In most cases, a beam splitter consists of two triangular prisms, arranged such that part of the light beam is reflected while the rest can pass through the splitter [32]. Hence, we are measuring passively, the network devices themselves are unchanged.

2.4 Features of the Time-Sensitive Networking (TSN) Standards

As we discussed earlier, one goal of this thesis is to use the developed framework on networks that deploy the TSN standards. TSN is a family of standards of which we want to give a brief overview of the background before explicitly presenting features deployed on test networks later on. The TSN standards have their roots in the Audio Video Bridging (AVB) Task Group, started by the IEEE 802.1 in 2007. At first, the main goal of this task group was to provide audio-video production studios with an Ethernet based, real-time capable solution for audio and video data transmission [23]. For these purposes, the AVB Task Group developed standards for time synchronization along with capabilities for bridges to guarantee bounded worst-case end-to-end delays and zero congestion loss in Layer 2 networks [7, 29]. With demand from the industrial automation and automotive industries for an Ethernet based alternative to the state-of-the-art field bus technologies, the AVB standards became the basis for a new task group called TSN, replacing AVB in 2009 [7]. TSN has a broader focus than AVB, since it is concerned with the standardization of real-time and mission-critical communication. This task group is, to this date, working on extending the bridged Ethernet standard IEEE 802.1Q. However, there are already many AVB or TSN features ready for deployment in Ethernet networks. In the amendment IEEE 802.1Qav [18], the AVB task group defined the Credit Based Shaper (CBS). With this, they made first steps towards bounded latency and bounded jitter guarantees, mostly for time- or loss-sensitive applications [23]. The TAS developed by the TSN task group extends these guarantees by deploying a traffic schedule on the network. It was defined in the IEEE 802.1Qbv [17]. Both amendments are now part of the current IEEE 802.1Q standard [15]. As part of this thesis, we will take a closer look at this finalized feature and analyze TSN-aware network devices with respect to their TAS performance. Hence, we give detailed information about the TAS in the next section. After that, we present protocols that are commonly used to achieve a common time reference in TSN networks. Finally, we give a brief overview on Frame Preemption which is often researched in combination with the TAS.

2.4.1 Time-Aware Shaper (TAS)

Communication in TSN networks takes place in the form of *streams* or *flows* that are defined by one sender and at least one receiver [3]. For these streams, the previously discussed Q-tag for the Ethernet header comes into effect. Specifically, each stream can have a priority, which is set using the 3 bit PCP field. TSN networks support a traffic scheduling mechanism, called the TAS, which was originally defined in the amendment IEEE 802.1Qbv [17]. We refer to the standard unless mentioned otherwise. Conceptually, the TAS uses a pre-calculated schedule for extended TT traffic control. In a switch, there is one TAS instance running per TSN-aware egress port. Such an instance differentiates between the priority values of the PCP field, and thus holds up to eight egress queues per port. Each queue is first controlled by a transmission selection algorithm (TSA) and after that, by a gate. If we disregard the gates, the structure is the same as that of any other traffic shaper. The TAS can extend any already existing priority-based shaper to the extent of TT capabilities. Figure 2.6 shows the functional structure of the TAS for one egress port. The gates are opened and closed according to a pre-calculated schedule, which is encoded in the GCL. Buffered frames in any queue can only be forwarded to the egress port if the gate is open (i. e., if the GCL entry is 1). If two queues are viable for transmission that means the TSA allows it and the gate is open, the transmission selection chooses the frame with highest priority.

If a coordinated schedule across the entire network exists, the TAS allows for temporal isolation between streams [28] in the form of an access pattern similar to Time-division Multiple Access (TDMA) [12, 26]. Furthermore, a network with such a coordinated schedule can guarantee bounded latencies because it can enforce uninterrupted transmission for high-priority streams [24]. For this to work, the switch has to guarantee that each transmission is finished before the gate closes. The standard encourages the switches to prevent this so-called *Transmission Overrun*. This ensures that other queues can transmit immediately once their gate opens. To achieve this, frame forwarding can be stopped before the gate actually closes. For example, forwarding could be interrupted as long as the transmission time of one frame of maximum size before a gate closes. This is called the *Guard Band*. This is a safe and easy to implement approach, however, it reduces available bandwidth due to possibly long idle times. A second possible implementation is for the bridge to check if the next frame in the queue can be completely transmitted in the remaining time before the gate-close event. Thus, the closer the gate closing event comes, the more the possible frame size shrinks. This method is called *Length-Aware Scheduling*.

A problem with the TAS is computing a network-wide coordinated schedule i. e., the GCL for every switch that fulfills all real-time guarantees. Finding such a schedule is proven to be NP-complete, the runtime of algorithms calculating such schedules is increasing drastically with growing network size or stream count [12, 23]. In many cases only probabilistic or heuristic-based approximations are made to speed up calculation times. Hence, a comparison between different algorithms can produce very different results regarding implementability or network requirements.

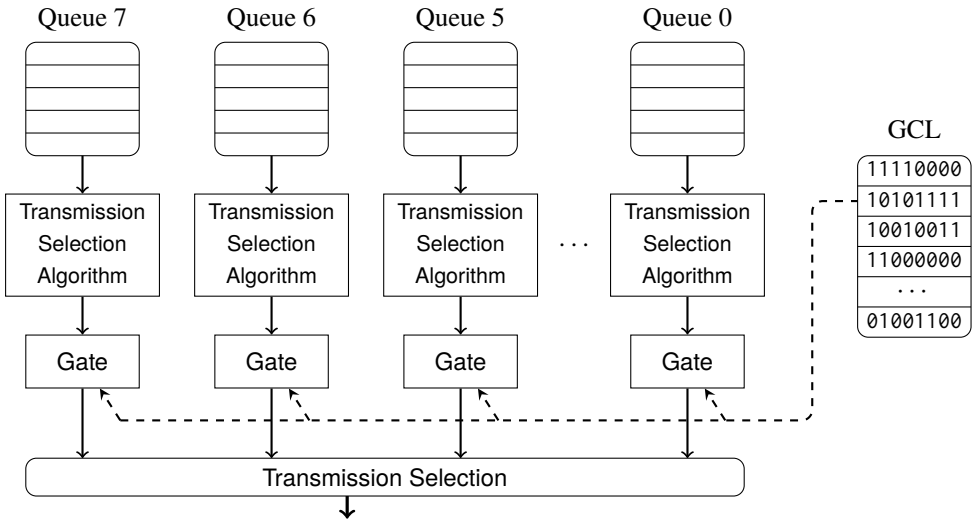


Figure 2.6: Functional structure of the Time-Aware Shaper (TAS) for one egress port. The TAS has up to eight egress queues that are controlled by a TSA, followed by a gate. The TSA, together with the gate, control whether a frame of the corresponding queue is available to the transmission selection. All (up to eight) paths are merged into the egress transmission selection that performs the final decision.

2.4.2 Time Synchronization

When talking about TT communication of real-time capable networks, time synchronization between the participating stations is crucial. TSN or TT networks in general depend on a common time reference for their functionality. Furthermore, the quality of synchronization has great influence on the quality of the possible communication guarantees. Thus, we require a good method for synchronizing the network devices to a common time reference. The features provided by TSN are independent of a specific synchronization protocol. However, Precision Time Protocol (PTP), defined by the IEEE 1588 standard [19] is currently often used [7]. For this reason, our measurement setups use PTP for synchronizing the switch clocks to the network interface cards (NICs).

We refer to the IEEE 1588 standard [19] unless mentioned otherwise. For each PTP domain, i. e., connected devices that synchronize to each other using PTP, one device is chosen as the *Grandmaster* from which all devices derive their local time. Outgoing from this grandmaster, the network establishes a master-slave hierarchy. All devices receive their time signal over their master port (except the grandmaster) and pass on a time signal over the slave port(s). Ports that do not connect to a PTP-capable device are inactive for the PTP operation. There exist two modes of PTP, namely Boundary Clock (BC) and Transparent Clock (TC). BC PTP instances process the incoming time signal to synchronize their own local time. After that, they use this local time to send new PTP messages over the slave ports. TC PTP instances only forward the master's signal to other devices, compensating the delay that is accumulated while the signal was processed by the TC instance. Generally, sub-100 ns synchronization between two devices is possible with PTP when using hardware time stamping [5]. Even a time synchronization accuracy with errors below approximately 20 ns per hop is often achievable [34].

For the synchronization between the sending and capturing NICs we used the NT-TS protocol which is specific to Napatech hardware. The NT-TS signal enables absolute time synchronization between Napatech equipment. We refer to the Napatech documentation unless mentioned otherwise [25]. The transferred signal contains the NT-TS protocol, carries a Pulse-Per-Second (PPS) signal and embedded time information. It uses high frequency sampling on the NIC's FPGAs to achieve high synchronization accuracies. The master sends synchronization information with 50 kHz which allows the slave to sample the external time every 20 µs. The convergence time to synchronization is below 30 seconds. The main reason we used this protocol in our setup was the high synchronization accuracy between two network adapters. The reported clock offset between master and slave are within ±1 ns. This allows us to use the time references of both NICs as a ground truth for the measurements.

2.4.3 Frame Preemption

Normally, even highest priority frames have to stay queued if a switch is already transmitting any other frame on the egress port. In settings that require very low latencies, this queuing time could result in a high priority frame being late. The IEEE 802.1Qbu Frame Preemption amendment [16] allows network devices to interrupt the transmission of (non-time-critical) transmissions, possibly circumventing long queuing times. High-priority frames do not have to wait for lower-priority frames to transmit fully, thus resulting in shorter latencies for the high-priority traffic. Hence, frame preemption can reduce the interference of best effort traffic on TSN traffic.

The standard allows to designate the eight PCP values as either *preemptable* or *express* [16]. As the name implies, the transmission of non-time-critical frames can be interrupted by express (i. e., time-critical) frames. The interrupted frame is split into multiple *fragments* that must be reassembled at the next hop. Each fragment is at least 64 B long (i. e., the minimum frame length for Ethernet). Non-terminal fragments are assigned a new 4 B FCS, and contain lower-level (i. e., Layer 1) information for the receiver, to determine that the original frame was preempted. The FCS for the last fragment is equal to the original checksum. Because of the size limitations, frames shorter than 124 B cannot be preempted. Therefore, the worst case preemption latency equals the transmission delay of a 123 B frame [35]. As soon as the time-critical frame(s) have been transmitted, transmission of the interrupted frame is resumed.

3 Related Work

In the following section, we introduce related research regarding the TSN standards and evaluation of TSN-specific real-time capable features. We group them into four categories, first, research providing a general overview and discussing the importance of TSN for various use-cases. Secondly, research conducting evaluations of TSN-specific features, such as the TAS in simulated networks. Thirdly, papers on the evaluation of these features on real world time-sensitive test networks. Lastly, research on the scheduling problems for the TAS.

The papers [7, 23, 24] provide an overview of the key TSN standards and discuss essential features of TSN that set it apart from AVB or best-effort services. Other papers, such as [30, 33], are concerned with future use-cases for TSN and analyze the application of TSN in more specific domains. Furthermore, Nsaibi et al. [26] research one possible transition from state-of-the-art field bus networks to TSN. Specifically, they show the advantages of using new TSN features in combination with the networking solution Sercos III. All mentioned papers uniformly consider TSN to be of great importance for future networking.

There is a multitude of papers that evaluate TSN features, most importantly the TAS. To date, most research is done using simulations; extending popular network simulators, e. g., OMNeT++, to time-based TSN features. The papers [6, 21, 28, 35] extend such simulators with TSN capabilities. Simon et al. [35] study the effects of TAS in combination with IEEE 802.1Qbu frame preemption in an OMNeT++ simulation. They consider the TAS as necessary for truly deterministic guarantees that can then be improved by using the TAS in combination with frame preemption. The authors show the effects of frame preemption on latency and determinism, which we omit in this work. However, they only illustrate the core effect of IEEE 802.1Qbu and TAS in an optimized scenario. They do not model clock synchronization inaccuracies and assume switching delays to be constant. However, we have found in our evaluation that exactly these variations are the difficulty in real world TSN networks that can, when neglected, even cause negative side-effects. Jiang et al. [21] simulate the TAS and conclude that it can guarantee low-latency high-priority traffic, even if best effort traffic would completely saturate the link. While they include time-synchronization errors in their simulation, they also do not include the second possible source of problems, the switching delay jitter. Similar to these two papers, Pahlevan and Obermaisser [28] use OPNET to simulate the TAS and analyze its delay characteristics. In their simulation, the TAS could significantly reduce both the end-to-end latency and frame jitter of the high-priority traffic. The achieved low latency and jitter were shown to be independent of best-effort load on the network. Moreover, they can show that multiple high-priority streams are not affected by each other, as long as their transmission windows are not overlapping. The authors omitted the transmission inaccuracies of the senders and the switches which can negatively affect two streams that are in theory temporally isolated. To fill this gap, our evaluations naturally contains all possible inaccuracies because it is based on real world measurements. Thus, our evaluation takes into account all aforementioned inaccuracies.

Jiang et al. [22] extend their previous work [21], further developing their simulation model and comparing the results to an identical real world setup. We consider this to be the first public research paper that compares simulations to a real world TSN setup. The authors tested a two-switch setup, where four best effort senders and one TT traffic sender are connected to the first switch. That switch is connected to the second, which in turn distributes the traffic to four best effort receivers along with one receiver for TT traffic. For their real world evaluation, they used active in-line network taps, capable of traffic capture and time stamping (ProfiShark 1G+). While their time stamping resolution was high (8 ns [31]), they influenced the network behavior, to the extent of added frame jitter caused by the network taps themselves. To prevent such effects, we use passive measurement techniques, such as fiber optic network taps. Furthermore, their setup did not allow to fully saturate the link with high best effort data rates, due to lower link speeds between best-effort talkers and the TSN-aware switch.

To create the schedules required for the TAS, many scheduling algorithms were developed and published. While the schedulers themselves are not strictly related to this work, the synthesized schedules are. Currently, most schedulers ignore the physical circumstances of networking hardware. They assume "perfect" networks, with jitter-free end stations and network devices, calculating the schedules for such optimized scenarios, e. g., [4, 11, 12]. However, our evaluation shows that this is not true, it is fair to assume almost any component to introduce jitter. Many of the papers about scheduling algorithms evaluate their calculation runtime for the synthesis of the schedule, but often disregard the implementability of the solution on non-optimal networks. Thus, determining how existing schedulers must be adapted to provide schedulability on physical networks should be addressed with the framework developed in this work.

Thus, the present work is considered to be the first that performs extensive high-precision testing on real world TSN-capable networks. This setup naturally includes all possible inaccuracies of the switches, senders and the time synchronization. To achieve our tests, we fill the current research gap and develop a measurement framework that fulfills high-performance real-time requirements.

4 Problem Statement

Many researchers working on real-time networking solutions, such as TSN, find themselves in need of a testbed. Up until now, researchers mostly relied on simulations for their evaluations. The primary reasons for this were the lack of available TSN-capable hardware and the missing evaluation capabilities. Now, researches are no longer limited by the availability, with more such hardware becoming available to the public. Rather, a framework that is capable of performing such measurements in the real world is missing. With this thesis, we aim to develop a measurement framework capable of performing measurements on TSN-aware networks. We first outline the general problem aspects and derive requirements for our solution afterwards.

4.1 Problem Definition and Use-Cases

Making measurement in real-time networks is a challenge because these measurements require a high time resolution and must be transparent to the system. In this thesis, we focus on a measurement framework to diagnose TSN networks. For low-latency low-jitter requirements, e. g., in the industrial automation domains, the two TSN features IEEE 802.1Qbv (TAS) and IEEE 802.1Qbu Frame Preemption are most important. Many researchers have already simulated these two features, often under optimized circumstances, proving their effectiveness in that regard. As a main purpose of the framework and this work, we aim to verify the results of the simulators and highlight possible difficulties that could arise in a non-optimal real world scenario. Therefore, we want to be able to measure the effects of TSN-specific features on the communications in a network, under different representative scenarios. Furthermore, we assume that highly specific network configurations are very easily influenced by inaccuracies in the network. Thus, we intend to analyze possible safe-guarding methods with which we can optimize the effectiveness of TSN features in a non-optimal network.

With this in mind, we need to be able to analyze single devices in an isolated setup, using the gathered information in more complex tests on larger networks. For example, we should to be able to measure the device-specific latency of a single switch, to use this value in multi-hop switched networks. With accurate knowledge over the network components, larger compound networks and their overall behavior can be analyzed. We limit the scope of this work's evaluation to that concerning the TAS. This has two main reasons: First, the effects of frame preemption are mostly simulated in combination with TAS because the TAS is considered to be more beneficial to low-latency and low-jitter requirements. Frame preemption on the other hand is most commonly used *in addition*, to further improve the performance. Secondly, hardware support for frame preemption, especially on analysis NICs, is still very limited. Thus, the framework must be able to measure and visualize the behavior of a switch with enabled TAS and compare that to the configured GCL. We must be able to verify the network's adherence to a set of constraints, using the framework as a means of gathering data. With a multitude of different schedulers for the GCLs available, we want to

be able to compare their performance in real world applications rather than in simulations. The primary focus of this work is to highlight the possible difficulties that are commonly omitted in scheduling algorithms. We leave an evaluation and comparison of schedulers with focus on the implementability and required synchronization accuracy for future work.

4.2 Requirement Analysis

With this problem definition, we can derive three key features we want to address. First, the device-specific evaluation, to analyze and understand single devices in a network. Secondly, the analysis of one or more devices in a network, using the TAS for real-time network functionality. Lastly, highlight the imperfections of a real world system in contrast to simulations, with the problems that can arise from them. The primary functional requirements are therefore measuring possible delays in a network.

Req. A.1 The framework must be able to measure propagation delays.

Req. A.2 The framework must be able to measure processing delays.

Req. A.3 The framework must be able to measure queuing delays.

Req. A.4 The framework must be able to show the effect of the TAS in a network.

To measure delays, we require time information about the traffic before and after the device or network under test. Therefore, for such time-critical measurements, we require at least two features of the framework: First, accurate and precise data capturing mechanisms, and second, a stimulus emulating end host behavior. The framework-controlled transmitter is necessary to have specific and time-accurate control over outgoing messages. To achieve meaningful measurements, we have to rely on the transmitters to be accurate and behave in a deterministic way. The transmission points must transparently emulate network behavior, with as little error as possible.

Req. B The framework must control transmitters, emulating end point behavior by injecting traffic into the network.

Req. B.1 The injected traffic must be precisely time-triggered, with precision only limited by the hardware and clock synchronization quality.

We limit the bandwidth requirements on the framework to 1 Gbit/s because most real-time networks were operating on 100 Mbit/s, and are only now moving towards higher bandwidths, with 1 Gbit/s currently being the upper limit of what is commonly deployed.

Req. B.2 The framework must be capable of generating full line rate stimuli, at least 1 Gbit/s.

Req. B.3 The framework must be capable of outputting stimuli on all hardware ports simultaneously, without compromising the timing accuracy.

The second core functionality of the framework is traffic capture. The receivers may not drop any received frames and we have to rely on the accuracy of the capture time stamps.

Req. C The framework must be capable of capturing traffic over receiving hardware ports.

Req. C.1 The captured traffic must be time stamped with an accurate arrival time stamp.

Req. C.2 The framework must be capable of lossless capture at full line rate (at least 1 Gbit/s).

Req. C.3 Capture must be possible on all hardware ports simultaneously.

Req. D The measurements should be performed transparently and passively, to not influence the network.

The framework should provide the stimulus and capture functionality independently of the physical networking hardware. The software would be of little use if it only operated in combination with specific network hardware vendors. With many hardware solutions available, we want the framework to be able to handle this hardware to the full extent.

Req. E The framework must be modular and hardware-independent.

Req. F The number of possible transmitters and receivers must only be limited by the hardware, not by the framework itself.

Capturing data is only the first step for the evaluation, the data needs to be aggregated to extract information about the network state.

Req. G.1 Important frame data has to be aggregated and saved for further analysis.

Req. G.2 The aggregated data must be presented in an inter-operable format.

Req. H.1 The data must be processed to analyze the network functionality.

Req. H.2 The evaluation must be extensible.

The evaluation may be performed after the measurement, the framework does not need to support on-the-fly evaluation.

Req. I The operational life cycle is limited to short-term measurements, rather than monitoring purposes.

With focus on high accuracy measurements, we limit the functional environment of the framework. This implies that we have a global view over the network under test. Furthermore, the networks can be purpose-built for the measurement goal.

Req. J.1 The functional environment should be primarily small-scale test networks.

Req. J.2 The test network should be static, i. e., the topology does not change while measuring.

Req. K The framework is to be deployed on a single machine.

Req. L The framework should be easy to configure and use.

5 System Model

For our research problems, we are concerned with test networks over which we have total control, i. e., we ensure a static network topology and we have a global view on our test setups. Furthermore, we can control the traffic flows in the network. For our evaluations, we need information about the physical layout of the topology but we also need to consider the network's behavior over time. Thus, we define two models that describe these two aspects. We represent the physical layout of the network with the topology model. This model builds the basis for the framework and the measurements conducted therewith. With our second model, the delay model, we can describe the timing behavior of the network traffic. It is the basis for our evaluations. The following section defines the mathematical model for the topology information, along with a graphical representation. After the topology model, we present our delay model.

5.1 Topology Model

Because TSN is a Layer 2 technology, we consider switched, multi-hop Layer 2 networks. Such networks consist of stations and links connecting the stations. To model these networks, we use directed graphs $G = (V, E)$, where the set of vertices (V) is the set of stations in the network, the edges $E \subseteq V \times V$ represent directional communication links between two stations. Thus, a simplex link between two stations v_1, v_2 is described by the pair (v_1, v_2). A full duplex link exists, if $(v_1, v_2) \in E$ and $(v_2, v_1) \in E$. On the hardware devices, we differentiate between ingress and egress ports. An egress port either has no connection at all, or is connected to an ingress port. This also holds in our mathematical model, the link (x, y) represents an egress port on device x and an ingress port on the device y. To model the network taps in our mathematical representation, we use a tapping function $t : E \mapsto V$, with

$$t(x, y) = \begin{cases} m & \text{if } (x, y) \text{ is tapped,} \\ \text{undefined} & \text{else.} \end{cases}$$

Where m is the station that receives a copy of the traffic on (x, y). Thus, connections are clearly identifiable and the model represents the real world in a simplified way. Figure 5.1 shows the elements of our graphical topology representation. The graphical representations consist of stations that are interconnected with simplex or full-duplex links. These links terminate on ports, which can be enumerated as a visual aid. Furthermore, we introduce a representation of network taps to clearly visualize the monitoring connections inside the network. Based on this topology model, we define our second model: the delay model. We use this model to describe the timing of a frame through the network.

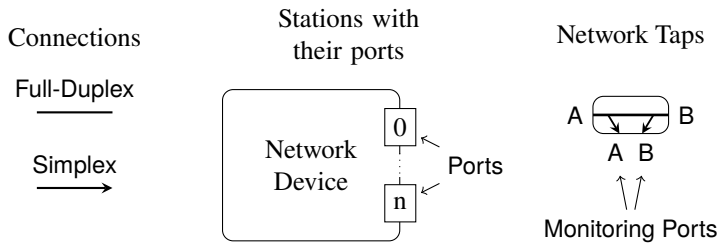


Figure 5.1: Elements of the graphical topology model. Port numbers of stations can be omitted. The network tap's monitoring port A receives the outgoing traffic from from A; analogously for B.

5.2 Mathematical Delay Model

For our evaluations, we have to consider the timing behavior of the network under test, e. g., Requirements **A.1** to **A.4**. Therefore, we require a model with which we can represent the measured data, and in a second step transform it to evaluate the network behavior. Therefore, we extend the network delay model from Section 2.2. We define the *Ingress* or *Egress Location Time* of a frame for each station $x \in V$. The ingress location time $x.\mathsf{t}_{\mathsf{loc,in}}(i)$ for a station $x \in V$ models the time, the last bit of frame i is received. Similarly, the egress location time $x.\mathsf{t}_{\mathsf{loc,out}}(i)$ is the point in time when x transmits the last bit of frame i. Figure 5.2 visualizes the location time definition on a small network. With these two functions, we can model the expected time of a frame at the ingress or egress ports of any station in the network. According to our network delay model, we use the location times as follows: Given two stations $x, y \in V$, and a link $(x, y) \in E$ between the two, we calculate the ingress location time based on the egress location time

$$y.\mathsf{t}_{\mathsf{loc,in}}(i) = x.\mathsf{t}_{\mathsf{loc,out}}(i) + d_{\mathrm{pg}}(x, y). \tag{5.1}$$

Where the mapping d_{pg} maps a link to its propagation delay. Furthermore, we model the time a frame i spends inside a station x with the transmission, processing and queuing delay of the station

$$\begin{aligned} x.\mathsf{t}_{\mathsf{loc,out}}(i) &= x.\mathsf{t}_{\mathsf{loc,in}}(i) + (x.d_{\mathrm{pc}}(i) + x.d_{\mathrm{qu}}(i) + x.d_{\mathrm{tr}}(i)) && (5.2)\\ &= x.\mathsf{t}_{\mathsf{loc,in}}(i) + x.\mathsf{d}_{\mathsf{res}}(i). && (5.3) \end{aligned}$$

We define this delay in a station as the *Residence Time* $x.\mathsf{d}_{\mathsf{res}}(i)$ at station x. With this value, two very important evaluations are possible. First, we can measure a station's processing delay (Requirement **A.2**), if we know that no queuing is occurring. Secondly, we can measure the queuing delay (Requirement **A.3**), given an approximation for the processing delay. The transmission delay $x.d_{\mathrm{tr}}(i)$ models the idle time of station x, until processing of the frame i can begin. It does not necessarily model the time it takes to transmit the frame (i. e., serialization). Recall the forwarding mechanisms of a station (cf. Subsection 2.2.1): A station supporting Cut-Through would have a frame-size-independent $x.d_{\mathrm{tr}}(i)$ although the time for serialization of the frame depends on the frame size.

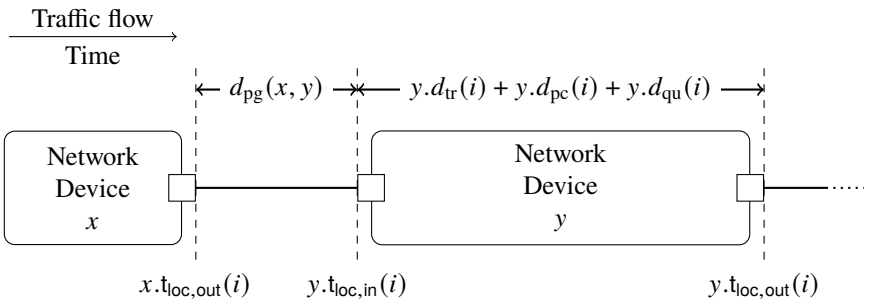


Figure 5.2: Location times at different points in the topology and the relations between them.

With this mathematical delay model, we define frequently used metrics.

Definition 5.2.1 (Latency)
According to IEEE 802.1Q [15], the Latency *is defined as the overall delay of a frame from one station in the network to another. The relevant time stamps for both stations are taken at the same reference point in the frame. In our case, this reference point is the end of the frame. Hence, we calculate the latency between the stations $x, y \in V$ as the delay between x transmits the last bit to y receives the last bit. In our traffic model that is*

$$(x \to y).\mathsf{d}_{\mathsf{lat}}(i) := y.\mathsf{t}_{\mathsf{loc,in}}(i) - x.\mathsf{t}_{\mathsf{loc,out}}(i).$$

This way, the delay of the two stations x and y is not part of the latency. To achieve good real-time performance, a predictable latency, i. e., only small deviation for a given frame size, is required [20]. Therefore, we define the range in which the latency typically lies, as a performance measure for TT networking.

Definition 5.2.2 (Frame Jitter)
The Frame Jitter *is the difference between the minimum and maximum latencies of a transmission of a frame*

$$(x \to y).\delta_{\mathsf{lat}} := \max_i \big((x \to y).\mathsf{d}_{\mathsf{lat}}(i)\big) - \min_i \big((x \to y).\mathsf{d}_{\mathsf{lat}}(i)\big).$$

In TT communication such as TSN, it should be possible to reduce the frame jitter to the precision of the used time synchronization protocol [23]. The frame jitter is closely related to the quality of the synchronization protocol and is influenced by the clock jitter. To describe the TT traffic behavior, we define two distance metrics, Figure 5.3 visualizes the next two definitions.

Definition 5.2.3 (Inter-Arrival Time)
The Inter-Arrival Time *of a frame to its predecessor is calculated as the difference of the two ingress location times.*

$$x.\mathsf{d}_{\mathsf{ia}}(i) := x.\mathsf{t}_{\mathsf{loc,in}}(i) - x.\mathsf{t}_{\mathsf{loc,in}}(j),$$

where frame j is the frame that arrived before frame i. The inter-arrival time is not defined for the first captured frame.

Note that the inter-arrival time only describes the distance between the ends of the two frames. The frame size does not necessarily influence the inter-arrival time. The next distance metric takes the frame size into account and allows for a more intuitive traffic description in some cases.

Definition 5.2.4 (Frame Distance)
The Frame Distance *is the link idle time between two frames that arrive at station x.*

$$x.\mathsf{d}_{\mathsf{fd}}(i) := x.\mathsf{d}_{\mathsf{ia}}(i) - \frac{\ell_i}{\text{link speed}}.$$

If the frame distance is zero, the two frames were sent *back-to-back*. The inter-arrival time is then equal to the frame's serialization time.

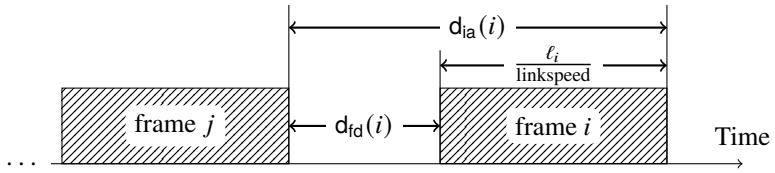


Figure 5.3: The inter-arrival time, the frame distance and the relation between the two.

6 Design and Architecture

On the basis of our requirements and our system model, we derive the architecture of our framework. Therefore, we begin with the architectural analysis, using the requirements to highlight key features and difficulties that have to be taken into consideration when developing the framework. We then derive core philosophies and design guidelines that we use during the design phase. Next, we design the architecture, starting with the general problem and the requirements on the framework, to our final solution.

6.1 Architectural Analysis and Core Philosophies

We analyze the requirements for their high-level quality specifications. The core architecture design is driven by these qualities. The requirements **B**, **C** and **H.1** describe the core functionality aspects of the framework, the transmission and capture along with the evaluation of the gathered data. First of all, we separate the measurement from the evaluation. This has two main reasons: There exists only one data flow between the two concerns, namely the flow of measurement data from the receivers to the evaluating components (Requirement **G.1**). Other than that, no control or data flow is required between the two. Furthermore, the operational life cycle of the framework allows the separation. The framework is required to run for short-term measurements with after-the-fact evaluation (Requirement **I**). This makes the separation even more compelling. With the two components separated, we could even use completely different applications, allowing for easier extensibility in the evaluation chains (Requirement **H.2**), without influencing or complicating the measurement. Architecturally the most demanding is the combination of required flexibility and the performance requirements for the measurement components. The requirements **B.1** and **C.1** describe the needed accuracy of the transmission and capture functionalities. This alone has immediate effects on the architecture, it forces us to provide high-performance and highly optimized implementations for the core functions. This is amplified by the requirements for high data rates without compromise on accuracy and precision (Requirements **B.2** and **C.2**). Furthermore, we are required to modularize the main components, such that many instances can run simultaneously, without interference (Requirements **B.3**, **C.3** and **F**). Fulfilling these performance requirements, while at the same time providing a high level of flexibility, e. g., in the underlying hardware (Requirement **E**), is difficult and requires careful planning.

We developed three quality-driven philosophies for the design phase. These philosophies have their basis in the required performance, extensibility and maintainability. First, we design the architecture with a *high level of modularity* and *flexibility* in mind. Secondly, a *high abstraction level* between the components allows optimizing the internals of each module, while still providing high flexibility and extensibility. Finally, we *separate between problems* and design specific problem-solving components.

6.2 Architecture Synthesis

The general workflow of a measurement comprises three main tasks: First, we configure the test setup and our measurement equipment for the task. Secondly, we execute the measurement and gather raw data. In a final step, we evaluate the raw data and compare the processed results to our expectations. In our case, we want to measure delays in real-time networks such as TSN. Recall our problem statement (cf. Chapter 4), where we highlighted the key problems of such highly time-critical measurements. For our measurements, we require information about the traffic at multiple locations in the network. For the measurement purposes, we use traffic analysis NICs, capable of accurately time stamping the incoming traffic. Using these cards, we must capture the traffic, extracting and storing important information. Furthermore, we must stimulate the network under test, emulating end-point behavior. This way, we can experiment with different traffic load scenarios or analyze the network behavior on specifically timed messages.

In the following sections, we take a closer look at our framework components, specifically, how the modularity aspects were considered in the architecture. We begin with an overview of the hardware and software components involved in our real-time measurement testbed: First of all, we have the network under test itself. We can model its topology with a graph data structure, recall our system model (cf. Chapter 5). In this topology, there are among others, the end-station ports that should later on be executing the framework-controlled measurement. The measurement process then comprises the execution of stimulating and capturing functions on the hardware ports of our analysis NICs (Requirements **C** and **B**). These NICs are installed in the host machine that runs the framework (Requirement **K**). The NIC vendors supply an application programming interface (API) with which we gain access to the hardware and the transmission and capture capabilities of every hardware port. An API represents a Hardware Abstraction Layer (HAL) but it is still hardware-dependent because it is commonly vendor-specific.

However, we can not limit the hardware support of our framework. For our purposes because we want to achieve hardware-independence in the framework control structures (Requirement **E**), we require a more abstract interface to the hardware functions. With such an abstraction interface, we can implement higher-level measurement processes independently of the underlying hardware. This separates two different types of control: First, there is the *Hardware Control* that we have to implement specifically for every hardware solution. This is e. g., responsible for hardware initialization. Secondly, there is the *Framework Control*, responsible for the overarching measurement process. But these measurement processes are more complex than just the transmit or receive operations. For example, we have to think about the traffic generation or data output functionalities as well. Furthermore, our measurement process includes a pre-measurement protocol that validates the test setup and the configuration. Therefore, it is only reasonable to strictly separate the framework control structures from network hardware-dependent implementations. Hence, we require an abstraction layer, generalizing hardware-specific API instructions, i. e., the *Hardware Control* functions, so that *Framework Control* functions can use them independently of the underlying hardware.

To achieve this control separation, we introduce a layer model to categorize the framework components. Figure 6.1 shows the layers of the framework and their main goals. We begin with the hardware layer, representing the hardware itself. Next, the HAL, i. e., the vendor-specific API, provides access to the hardware functions. The framework's first layer is the *Abstraction Layer*, it is situated above the HAL and contains all functionality that is used for hardware control. The abstraction layer ensures interchangeability and flexibility in the measurement hardware by

implementing low-level API interactions (Requirement **E**). Above that, we use the *Control Layer*, implementing the framework control functionality. With these layers, we separate not only the two control structures but achieve modularity in the underlying hardware as well, supporting Requirement **E**. Above the control layer, we introduce the *Presentation Layer*, responsible for the data output during a measurement. The presentation layer aggregates all measurement data (Requirement **G.1**) and pre-processes it into a format applicable for further processing (Requirement **G.2**). Finally, the *Evaluation Components* process the previously gathered data (Requirement **H.1**).

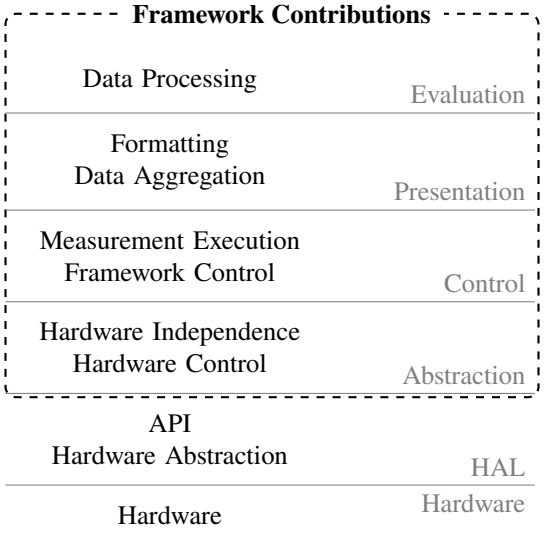


Figure 6.1: We split the framework functionality into four layers that are situated above the HALs of the vendor-specific hardware. Shown are the layers with their main goals and purposes.

Figure 6.2 shows the components we introduce in the following paragraphs. We start with the network representation of our test setup. The topology data structure models the stations in the network along with the links between these stations. The topology itself is not intended for any kind of measurement control, but is only there to hold the network layout, so that other components can use it. This way, we adhere to a strict separation between the controlling logic (i. e., the components of the control layer) and the common data structures. Internally, each end-station port and network device in the physical topology corresponds to a node in the framework's topology data structure. We extend the representation of each node in the topology with framework control structures. Hence, the following components are situated in the *Control Layer*: Every node in the topology has a *Type*. Furthermore, every type can fulfill several *Roles*. The types control the roles and the roles implement the framework-controlled functionality aspects of the measurement. Moreover, the type components extend the topology, to strictly separate control logic from the topology data structure, improving maintainability and re-usability. The two most important possibilities for roles are the transmission and reception tasks for the measurement (Requirements **B** and **C**). We implement the `TX_role` for the transmission side of the measurement procedure (traffic selection, frame creation, etc.). The `RX_role` implements the capture counterpart. The roles and the types implement the main framework control functions, require hardware control access. For this hardware access, we add API-specific subclasses in the *Abstraction Layer*. These subclasses correspond to one API, therefore we group one set of API-specific type and role implementations into a *Hardware Abstraction Module*.

Thus, adding support for new hardware solutions requires the implementation of the corresponding hardware abstraction module. Inside the module, we implement three components for each hardware solution, providing generic interfaces to upper layers. We begin with the *API-Specific Type* as a type subclass. We use it for the API initialization and overall hardware configuration tasks. Subclasses of a type implement hardware-specific initialization tasks and provide access to hardware information, e. g., the port's MAC address. These basic setup tasks are independent of a type's role in the measurement process, therefore they are separated from the roles. The *API-Specific TX* is a subclass of the `TX_role`, it uses the already opened API connection and configures the hardware for transmission tasks. It implements a generic transmission interface that the `TX_role` uses. Similarly for the *API-Specific RX*, this component configures the hardware for capture functionalities and provides a generic frame-level capture interface. Thus, each API-specific role implementation only needs to provide a generic transmission or capture interface required by their superclass (recall our high-abstraction level philosophy). With this feature, the framework can accommodate different low-level hardware-specific implementations. Thus, the types and roles together represent the extendability and flexibility required by Requirement **E**.

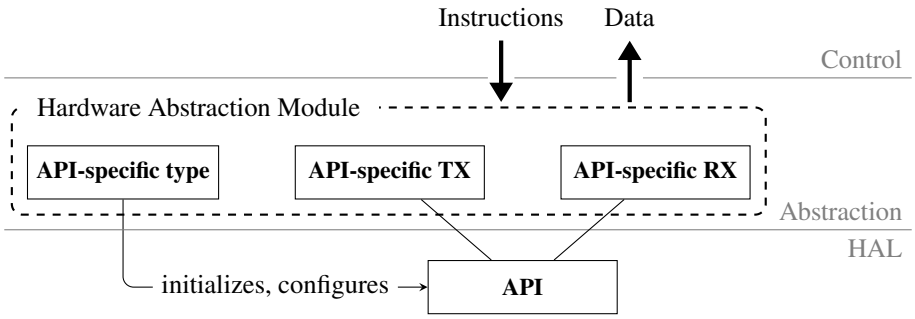


Figure 6.2: The Hardware Abstraction Module comprising the API-specific type and role implementations. They act on instructions from the control layer and provide access to information and network data.

To achieve high bandwidth transmission or capture capabilities (Requirements **B.2**, **C.2**), even on multiple ports simultaneously (Requirements **B.3**, **C.3**), we must modularize these functions for every hardware port. We must be able to support all possible hardware transmission or capture configurations, in the worst case transmitting and receiving on all hardware ports simultaneously (Requirement **F**). It would therefore be optimal to run the tasks for each hardware port in a separate thread, to optimally utilize the available processing cores of the host computer. This way, the number of parallel running tasks of the framework is not limited by the framework itself, but only the hardware it runs on. Thus, we group all framework components that involved in the transmission task during the measurement into a *Transmission Group* for each hardware port. Similarly, we group all components concerned with the capturing measurement tasks into the *Capture Group* for each port. We assign one worker thread to each group. Thus, each framework task runs independently of other tasks. To achieve multi-threading in an elegant way, we include the superclass `role`, of which `TX_role` and `RX_role` are subclasses. The `role` superclass provides the multi-threading support for both roles at the same time, again minimizing code duplication. Moreover, the `role` class represents a *Start/Stop* interface to the `type`. It allows the type instance to start and stop all its roles, without any knowledge over the implemented functionality. Furthermore, this simplifies the integration of new roles, if they are required in the future, improving the framework's maintainability.

6.2.1 Configuration manager (CM)

Every measurement starts with the configuration of the testbed and the measurement equipment. In the case of our measurement framework, we pass most of the configuration tasks on the the framework, and only supply the configuration parameters for the setup specification. For the configuration of a measurement, the user wants to have a user-friendly and reproducible way of configuring the framework (Requirement **L**), hence we need a powerful interface between the user and the framework. We introduce the configuration manager (CM) as a configuration interpreter that provides the setup information to all framework components. The CM represents the main interaction point between the user and the framework, increasing ease-of-use. As input, we use a human-readable configuration file. We chose a configuration file over e. g., a pure command line interface, to allow for easy reproduction of measurements. Furthermore, this allows to store the setup instructions along with the resulting measurement data. This inherently documents the output, reducing the possibility of undocumented measurement results. The input configuration contains at least three pieces of information: First, it should contain the topology of the physical setup (cf. Chapter 5), including the locations of transmitters and capture points. Secondly it must contain the setup parameters for the underlying hardware (e. g., the port numbers on the adapter). Thirdly, it must specify the traffic, i. e., the stimuli, the senders should produce (Requirement **B**). Lastly, the framework outputs the measurement data, hence we require an output format and a storage location for this resulting raw data (Requirement **G.1**). The CM is now responsible for the interpretation of the input configuration, and setup of the framework components. Generally, the CM creates elements of the abstraction, control and presentation layers and interacts with the control layer. The control layer then gives instructions to the underlying abstraction layer. In the setup phase, the CM first initializes the internal topology data structure. The user specifies the required hardware abstraction module for each station. Hence, in a second step, the CM instantiates a type instance for every station, using this information. Furthermore, it selects the correct abstraction layer, in the form of a hardware abstraction module underneath. The type then performs the hardware initialization in a second step, using the setup instructions provided by the CM. The type instances open API connections, which enables general hardware access for the port, e. g., we can retrieve the MAC address from the hardware. The configuration furthermore specifies, whether the station is a transmitter or a receiver, i. e., the configuration specifies the roles of each station. The running type instance for every port receives the role information from the CM and starts all roles, with the correct hardware abstraction modules underneath. The type instances delegate all role-specific setup tasks to these role instances and ultimately to the hardware abstraction module. Types that are configured as senders start a `TX_role`, to enable the transmission capabilities on the corresponding port. Analogously, the receiving types start their `TX_role` which enables the capturing hardware capabilities. The hardware abstraction module components then wait for further instructions from the control layer. The configuration file also contains information on the stimuli each sender should produce during the measurement phase. Thus, the CM extracts this traffic information and provides it to the transmission role. This concludes the general setup phase and the active lifetime of the CM. Next, we look specifically at the transmission and capture control logic of the measurement. After that, we present the data aggregation functionality in the form of output managers. Finally, we outline the architectural decisions for the evaluations.

6.2.2 Transmission Capabilities

Measurements in real-time networks require highly accurate TT stimuli. These stimuli allow for an analysis of the network's behavior over time. For example, for measurement concerning the TAS, we must be able to inject these stimuli precisely aligned with the gate operations of the TAS (Requirement **A.4**). Thus, the traffic injection task of the framework plays an important role and we must meet several requirements to achieve meaningful measurements. Mainly, these requirements are of a performance-oriented nature, we must ensure that we create the TT traffic frames early enough, so that the NIC is able to transmit them on time (Requirement **B.1**). Moreover, the framework must fulfill this, even at high data rates (Requirement **B.2**) and on all hardware ports at the same time (Requirement **B.3**). The supported hardware ports must not be vendor-specific (Requirement **E**). Thus, we find ourselves with multiple problems, we have to solve. Specifically, these are the TT transmission of frames, the performance requirements on this task and hardware-independence we must achieve at the same time. In the previous section, we have already discussed the way our architecture inherently supports hardware-independence. Figure 6.3 shows the relations we describe in this section for one transmitting port of a NIC. The previously mentioned *Transmission Group* is the collection of all framework components contributing to the traffic injection (Requirement **B**). The API-specific TX implementation is a part of a hardware abstraction module and uses the API, thus is able to transmit frames on a specific hardware solution. Furthermore, the API-specific TX provides a generic transmission interface we can use in more abstract components. Most importantly, the `TX_role` uses the API-specific TX implementation to transmit frames as part of the measurement procedure. Hence, we separate the overall measurement procedure from the transmission capabilities. This separation allows for hardware-independent measurement execution, fulfilling Requirement **E**. Furthermore because we have one transmission group for every transmitting hardware port, we achieve independence between multiple ports. Therefore, we meet the multi-port requirements Requirement **B.3**, if we fulfill the performance requirements for the TT data rate on a single port.

Because of this, we focus on the fulfillment of Requirement **B.2** in the following paragraphs. Because a NIC typically requires all frames to be enqueued in order of transmission, we must determine the absolute order of frames of each port ahead of time. Moreover, we have to determine this order quickly and efficiently to provide the required high data rates. As a quick overview, we have to select and prepare the next frame for transmission all in the time the NIC transmits the last one. In the worst-case scenario, the last frame was of minimum size, leaving us only just over 670 ns (at 1 Gbit/s) to prepare the next frame for the NIC. Thus, we require a fast transmission selection and frame creation method, to achieve the required high stimulus data rates.

Generally, there were two possible solution approaches. First, for each transmitter, we could fill a queue with frames that are ready for transmission and in order of transmission. This has two main advantages, first, we have no time constraints on determining the order of the frames and the frame creation process, as it would be done before the measurement starts. Secondly, the transmission preparation of a frame during the measurement can not be done any faster, as it is only a copy operation from the queue to the NIC. The disadvantage of this approach is the high memory pressure of the queues, especially for longer measurements. However, this contradicts the Requirements **B.3** and **F**, as a large number of senders can very quickly use up the available memory on the host machine.

For this reason, we opted for the second possible solution approach. This comprises transmission selection and frame creation in an on-demand solution during the measurement process. This requires two things: First, a model for the stimuli, with which we can describe the traffic in the configuration. Secondly, a method that uses this description to determine the correct frame order during the measurement. By separating the traffic description from the transmission selection methods, we achieve a clear separation between the data structures and the controlling methods, simplifying the larger problem. We solve the traffic description with a data structure, specific to this purpose. We define *Streams* that represent a communication direction from one sender to at least one receiver. In the configuration, we assign a periodically scheduled time slot to each stream. Inside this time slot, the stream's messages can be sent. Thus, the CM creates a set of streams for each transmitting end-station port, based on the given configuration. This, along with the previously discussed tasks concludes the CM's setup for transmitting stations. During the measurement the `TX_role`, as the main transmission controller then needs the second component, the transmission selection. Here, we must optimize the selection methods, to fulfill our performance requirements. Due to these requirements, we separate the transmission selection methods from the overall measurement control logic in the `TX_role`. Therefore, we introduce a new class, the *Scheduler*, implementing an earliest transmit time first (ETF) selection algorithm. The `TX_role` can use this scheduler to determine the next frame to send. Thus, in the `TX_role`, we can concentrate on the overall measurement execution and can ignore details such as the transmission selection algorithm. This separates the two control mechanisms and allows for better maintainability and optimization.

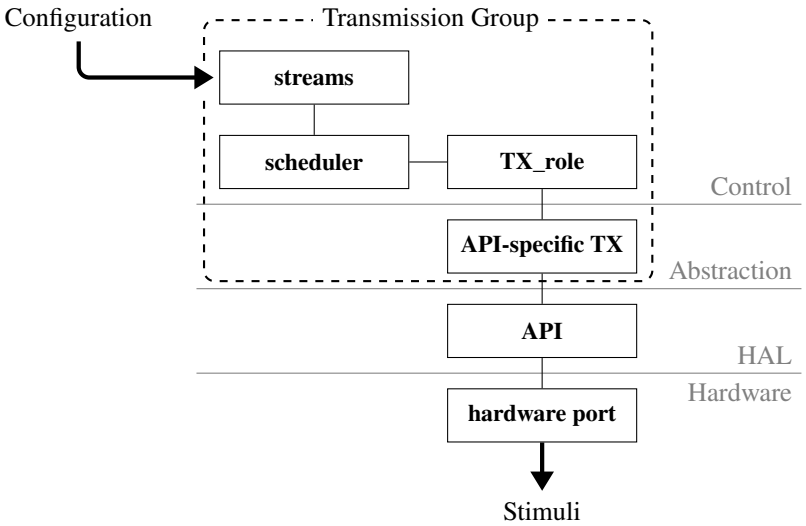


Figure 6.3: The components of the transmission group.

6.2.3 Traffic Capture

For our measurements, we require information about the traffic in the network. Specifically, we are interested in the time the frames arrive at a station in the network. Therefore, we use capturing end-stations throughout the network, time stamping and storing the incoming traffic. With the gathered data, we can evaluate the network behavior in a next step. For these capturing capabilities, we must support full line rate capture (Requirement **C.2**) on theoretically all hardware ports simultaneously (Requirement **C.3**). We have to aggregate the data (Requirement **G.1**) and output it in a format applicable for the evaluation (Requirement **G.2**).

Figure 6.4 shows the relations we describe in this section for one hardware port. Similar to the transmitting side of the measurement, we use a *Capture Group* for every capturing hardware port of our NICs. Because we design these capture groups to act independently, i. e., the capture functionality of every port runs on its own, we fulfill Requirement **C.3**. The capture group contains three main components: The first is our hardware abstraction component (API-specific RX) that provides hardware-independent frame capturing capabilities. Second, and similar to the transmission side, we use a `RX_role`, responsible for the the overarching execution of the measurement process. In the case of the `RX_role` this comprises administration of the measurement start or end, while providing the capture and store loop during the measurement. To store the frame data, the `RX_role` uses the presentation layer. We deliberately move the data aggregation and storage tasks outside of the `RX_role`, to reduce the complexity. To store the captured frames to an output format usable by the evaluation components, the `RX_role` uses an output manager (OM) (Requirement **G.1**). Thus, the fulfillment of our performance requirements (**C.2**) hinges on the performance of the OM. To reduce the load on the OM and the final storage impact, we perform data filtering in the `RX_role` where possible. This way, unnecessary data can be discarded quickly, reducing the required disk space and I/O bandwidth. Not the entire frame data should be stored to the output, but only data relevant to the measurement. While it is, of course, possible to store the entire frame in small measurements, it quickly becomes less feasible, when measuring over longer periods of time or when writing to a slow medium. Moreover, to prevent frame drop on the receiver side, even when writing to a slow output format, a fast OM is to be implemented that ensures it is always capable of receiving data. The architectural details of the OM are discussed in the next section.

6.2.4 Data Aggregation

For the overall evaluation of the network behavior, our previously measured time stamps and other frame data must be aggregated and stored. The data we gather originates from a capturing end point, i. e., a capture role. The capture roles use the hardware to time stamp the incoming traffic and, as previously discussed, move the frame data to the presentation layer, specifically the OM. Thus, the OM fulfills the aggregation purposes (Requirement **G.1**). However, we must consider two requirements for this data aggregation process: First, the data output process must support high data rates (Requirement **C.2**), but at the same time, we must support an inter-operable, or in the best case, an interchangeable data output format (Requirement **G.2**). Figure 6.5 visualizes the relations between the components of our solution. We once again simplify the problem by solving one problem at a time, creating two collaborating components: The first is the OM as interface to the capture group. It must fulfill the high data rate requirements. To achieve this, we implement a data buffer in the OM, allowing for high data rate storage, independent of the output format. It

allows buffering larger chunks of data to memory that can then be moved to persistent storage in batches or after the measurement. This increases the framework's memory footprint and, in the event of a system failure, could result in loss of data. However, the risk of losing data on a system crash is far less severe, compared to the possibility of regular frame drops on the receiving NIC due to an overflowing receive buffer. To mitigate the increased memory pressure of this buffering, the write interval (i. e., the batch size to write to persistent storage) is customizable. This way, depending on the medium write speed, the batch size can be adjusted for smaller memory impact on faster mediums, while slower ones can benefit from a larger memory buffer. Thus, we ensure that the OM can always take data away from the capturing components, even if we are writing to an extremely slow medium or format. Thus, we fulfill Requirement **C.2**. The second component is the *Format Writer*, responsible for writing the buffered data into a format that is usable by the evaluation. Furthermore, we want the format writer to be interchangeable, allowing for different output formats. Because of this interchangeability, we ensure flexibility in the output format (Requirement **G.2**). In the next section, we give an overview of the evaluation components. They use this formatted data and process it into meaningful outputs.

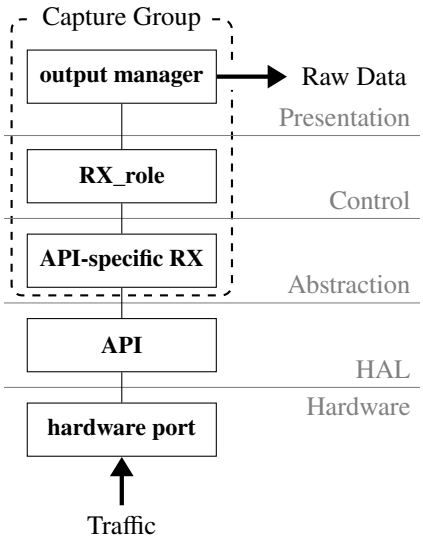


Figure 6.4: The components of the capture group.

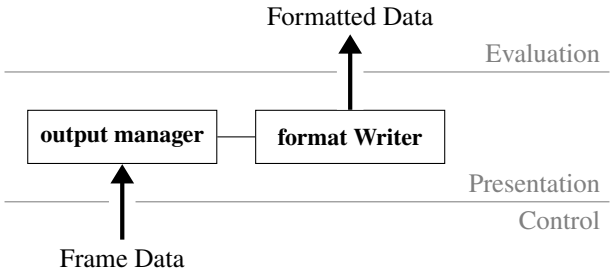


Figure 6.5: The output components. They receive data from the underlying control layer and ultimately output formatted data for the evaluation.

6.2.5 Evaluation Functions

The goal of every measurement is the analysis of the network behavior in comparison to a hypothesis. For such comparisons, we must process and visualize the raw data into a form, understandable and meaningful to the user (Requirement **H.1**). This is the final step in the framework workflow. We already know how the data is gathered, filtered and finally stored. Therefore, we now create an environment, in which the analysis of this raw data is as simple as possible. This implies the import of data, the pre-processing and finally the visualization process. We are mainly concerned about the evaluation to be extensible and customizable by the user of the framework (Requirement **H.2**). Because the measurement goal may change frequently, the evaluation must be very flexible and should allow for user-defined evaluation pipelines. The most important step for this flexibility is the strict separation between measurement and evaluation. The measurement components are used for the actual measurement and data acquisition purposes, whereas the evaluation components come into effect after the measurement is finished, processing the gathered raw data. This enables separation of concerns and allows for a better extensibility in the evaluations. We achieve highly flexible evaluation chains because we limit the interaction point between measurement and evaluation to the required data flow only. Thus, we provide general data acquisition methods that import the measurement's gathered data. More specifically, we develop a `CapturePoint` data interface that presents the data in a preprocessed form, grouped by the position in the topology, at which is was acquired. To fulfill that, we build an interface to the output format of the measurement data. In the following chapter, we present implementation challenges we had to solve and details about the solutions implemented in the framework.

7 Implementation Details

In the scope of this chapter, we will present some of the design decisions we took to meet our requirements. We will in most cases only highlight the used concepts and ideas and will not go into such detail to show language-specific implementation details. For information even more detailed than presented here, we refer to the codebase of the framework. For the implementation, we chose C++ for the configuration and measurement components. C++ was mainly chosen for the structuring and object oriented programming capabilities, while still providing very high performance thanks to highly optimized compilers. Because we saw a great benefit in object orientation, we chose C++ over C, even though C *could* theoretically have resulted in a framework capable of higher performance. Thanks to object orientation though, we achieved a more flexible, extensible and maintainable framework. The evaluations are implemented in Python, with SQLite3 databases holding the captured data.

In the following sections, we will first introduce the common data structures, used throughout the framework. After that, we follow the data and control flow in the framework, first discussing the measurement initialization. Secondly, we present the implementation details in the measurement mechanics. Finally, we discuss the details of the data output and processing mechanisms. The data processing mechanisms also hold more specific information about the data acquisition and evaluation methods we use later on in the evaluation chapter.

7.1 Data Structures

The following three sections describe the data structures we use throughout the implementation. We show the internal topology representation, the contents of transmitted frames and a data structure for scheduled communications in the form of streams.

7.1.1 Topology Representation

The framework holds the relevant topology information in a data structure, for later reference. This data structure is very similar to any directed graph implementation and our system model (cf. Chapter 5). We model stations, each with their own MAC address and a station delay. On initialization, the `type` instances read the MAC address from the hardware and write it to the topology data structure. Our transmitters then use this information to write the destination MAC addresses into the transmitted frames. We use the station delay as an approximation of the station's processing delay, for an overall latency estimation in the setup phase of the measurement. Furthermore, each station stores a bit mask, indicating functionality of this station. With this we can e. g., mark switches and network taps appropriately. We store links between two stations as pairs of the two station pointers, together with their `linkParams_t` that holds the link speed and propagation delay information.

7.1.2 Contents of a Measurement Frame

The framework needs to transport several pieces of information with each transmitted frame. For example, the framework has to be able to unambiguously differentiate between two captured frames, thus requires a sequence number in the frame payload. Because we use time stamp injection, we require a field in the frame for this data. To transport all the required information bundled together with the frame header itself, the framework uses a custom frame layout that extends the Layer 2 header (cf. Figure 7.1). The physical and data link layers, i. e., the Layers 1 and 2 are unchanged by this.

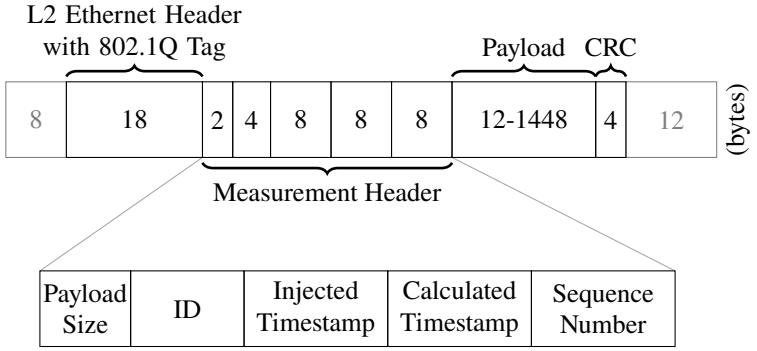


Figure 7.1: The measurement framework prepends the Layer 2 payload with a measurement header, containing information about the transmit timing along with a sequence number and a payload length tag.

First, a measurement frame always contains the data fields of the Q-tag. The Q-tagging functionality can be disabled by overriding the TPID, thus making the frame appear untagged. We require the Q-tag's PCP field, for measurements on the TAS (Requirement **A.4**). After the Q-tag, a payload length field is included. We use this, to transport the desired payload length of the transmitted frame. After that, a 4 B measurement identifier, which is constant to one measurement is situated. We use this identifier, to quickly differentiate between cross-traffic and traffic injected by the framework itself. After this, the timing information follows. We use transmit time stamp injection to have information about the transmission time of a frame, which is required for delay measurements (Requirements **A.1** to **A.4**). Hence, the first 8 B unsigned integer field is the place where the senders inject the transmit time stamp. Some NICs (for example older Napatech cards) are only capable of injecting a time stamp at a frame offset that is a multiple of 8. For this reason, the place for the injected time stamp was chosen here, at offset 24 from the start of the frame. It was not strictly necessary for the transmitting card used later on, but seeing as it might avoid refactoring work for other network cards, the data fields were positioned with this restriction in mind. The next 8 B field is used for the calculated time stamp. This value is the internally calculated transmit time for this frame. Thus, the value corresponds to the expected transmit time of the frame, and can be compared to the injected time stamp. This way we can verify the sender's transmission behavior relative to its own clock, giving a first indication about the transmission accuracy (Requirement **B.1**). We use this for example in our framework evaluation in Section 8.2. Last in the measurement header is the frame's sequence number. We use this as identification of the frames in our measurements. After this, the regular layer 2 payload follows. Due to the length of the mentioned fields though, the possible payload length is reduced to between 12 B and 1,448 B. All fields along with the layer 2

header and a Q-tag are represented in the `measurehdr` structure, used throughout the framework implementation. The header also defines useful macros for the interaction with the contents of the `measurehdr`. For example, when using these macros, the network and host byte orders are automatically taken care of.

7.1.3 Stream Data Structure

The measurement frames we inject into the network are always part of a stream. Each stream represents periodic traffic from one sender to a set of receiver addresses. For this, each stream is assigned a *Transmission Slot*, in which all frames must be sent. We configure the transmission slots with three variables, `slotPeriod`, `slotLength` and `slotOffset` (cf. Figure 7.2). The `slotPeriod` (t_p) determines the distance between the start times of two slots. The `slotLength` (t_ℓ) describes the length of the slot, i. e., the duration in which the stream may schedule frames. Lastly, the `slotOffset` (t_Δ) describes the phase of the stream's cycle, as an offset from zero. We can calculate the transmission time of the first frame in slot number $i \in \mathbb{N}_0$ as

$$t_{i,0} = t_\Delta + i \cdot t_\mathrm{p}.$$

The transmission times of frames other than the first depend on the stream's transmission mode. In *Regular Transmission Mode*, we spread the frames out over the transmission slot. So for N frames in the slot, we calculate the scheduled transmission time of frame j in slot i as

$$\begin{aligned} t_{i,j} &= t_{i,0} + \frac{j \cdot t_\ell}{N} \\ &= t_\Delta + i \cdot t_\mathrm{p} + \frac{j \cdot t_\ell}{N}, \quad i, j \in \mathbb{N}_0. \end{aligned}$$

Otherwise, in *Packed Transmission Mode*, we schedule the frames back-to-back inside the slot.

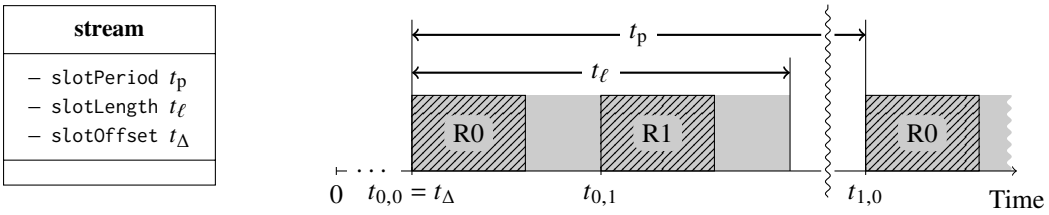


Figure 7.2: Given a transmission slot defined by the three parameters `slotPeriod`, `slotLength` and `slotOffset`, a stream can independently calculate transmission times for each receiver.

Transmission outside of the slot boundaries is not allowed. Therefore, the user should ensure that the slot is large enough to hold all possible frames without talking over other slots. To give the user feedback about the slot boundaries, we include a simple check for possible slot size violations. The stream implementation assumes the maximum possible size for each frame in the slot and checks whether all frames can be transmitted inside the configured slot length. If a possible error is found, a warning is raised to the console output, however the measurement execution stays unchanged. The frame size can be randomly generated, given a payload size range in the configuration. We can even configure a step size for the payload sizes, allowing, e. g., for multiples of 8 B in frame

size. Note that the payload size does not include the `measurehdr` length of 48 B. Due to the frame size limitations of Ethernet, the effective payload sizes are restricted to between 12 and 1,448 B. Values between 0 and 12 are allowed, but do not result in smaller frames. If the range $(12, 20)$ with step size 8 is given, the framework would transmit two different frame sizes, both with the same probability of $\frac{1}{2}$ (64 B and 72 B).

7.2 Measurement Initialization

This section describes the components employed during the initialization phase of the framework. We first discuss the technicalities of the modularization efforts we implemented in the framework. After that, we further elaborate on the algorithmic specifics of our pre-measurement Link Discovery Protocol (LDP). Lastly, we detail difficulties in developement of the the Napatech hardware abstraction module we used for our measurement network cards.

7.2.1 Configuration Setup Hooks

As we have discussed, we use a CM as main interface between the user and the framework. The user supplies a configuration file, which the CM reads and interprets. Using the given configuration, it initializes the components required for the measurement. The details of the configuration file syntax and structure are shown in Appendix A. Moreover, the topology initialization or the stream information import is just a matter of translating the input file to a data structure and are therefore not discussed in detail. We concentrate on the setup of the `types` along with their `roles` in this section.

The user supplies information about the hardware abstraction modules for each station. We associate each hardware abstraction module (i. e., a child of `type` with its `roles`) with an identifier in the configuration file. If the user defines a station with a known identifier, the corresponding *Setup Hook* is started. A setup hook realizes a constructor call to a `type` instance that furthermore extracts all necessary arguments from the configuration. Hence, we support custom constructor calls that can be satisfied with argument lists in the configuration file. Thus, the configuration file is inherently modular. To satisfy hardware independence (Requirement **E**), a user needs to provide the module and setup hook, and can then directly use the configuration file for control. This way, no separate configuration method is required, even if the framework is extended with custom modules, providing easy configuration and usability (Requirement **L**). With the topology initialized and all modules running, the CM reads the stream information. This configures the transmission behavior of the senders and instructs the receivers to prepare for capture. This preparation results in a pre-measurement protocol, which we discuss in the next section.

7.2.2 Link Discovery Protocol (LDP)

For the measurement purposes of the framework, the transmission and capture roles run a protocol, prior to starting the measurement loops. The protocol has two main purposes. First, it gives a sanity check to the user and confirms that the physical topology corresponds to the intended configuration of the current measurement. It thus helps with ease-of-use of the framework (Requirement **L**).

This allows for quick error discovery, in the case that e. g., a cable is plugged in the wrong port. The second purpose of Link Discovery Protocol (LDP) is that it allows the listeners to prepare for incoming traffic. It is sometimes extended for transmitter calibration as well (we use the LDP frames as transmit timing calibration for our transmit NICs, cf. Subsection 7.2.3). Moreover, we can use LDP to propagate specific pieces of information of the stream, for example the number of frames to expect.

With the topology and all transmit and capture points initialized, the role controllers start the LDP. For each stimulus, LDP uses the topology information to calculate all listeners i. e., the actual receivers *and* other monitoring stations that are tapped into the signal path. The groundwork for LDP is done by the CM, which uses the topology information to calculate the expected streams for each receiving station. The algorithm used for this purpose is executed once for each stream and functions in two parts. First, the forward pass, calculates the shortest paths from the sender of the stream to every other station, using the *Bellman Ford* algorithm with the expected delay as a metric. The expected delay is calculated with the propagation delay of each link and the processing delay of each network station, if specified in the configuration. This way, the expected latency of the path from sender to receiver is already known at this stage, disregarding any queuing or traffic shaping in the switches. Secondly, the *Previous Hop* for each station is stored. This is always the next station on the path from receiver to the sender. This way, a spanning tree is calculated alongside the shortest paths. In the second phase of the algorithm, we traverse this tree from the receivers (a subset of the leaves) to the sender (the root) and mark all nodes as *Path Relevant* if they are seen on the way. Thus, all stations that are not relevant to the transmission of this stream stay unmarked. Using these markings, we iterate the set of links again and find all taps that are path relevant. If the tap has a monitor attached in the correct direction, the monitoring station becomes path relevant itself and is thus a listener for the traffic of this stream. All listening stations (i. e., path relevant stations that are receivers as well) will have this stream added to their list of expected streams. This concludes the preparation phase. Afterwards the transmitters send one message for each sender-receiver pair of every stream they know. These frames contain the regular header information and all time stamps. The frame is always of minimum size (64 B). The value in the sequence number field indicates the total number of frames to expect in exactly this sender-receiver configuration. Hence, the receiver can store this information and use the list of expected streams for the running topology configuration as comparison. Marking off the seen streams and remembering streams that are not in the expected list allows the user to quickly know if something is not working as intended and either the configuration is not correct or the actual network topology is wrong. Warnings are raised if either an unexpected stream is captured or a stream was expected, but no frame arrived.

The measurement procedure begins after LDP is finished. The user can specify a cycle length in the configuration. This cycle length of the configuration can for example be used to synchronize the transmitter to the GCL timing of a connected switch. If this length is configured, our framework will ensure that the measurement starts with a new cycle. Hence, we can align our transmission cycles to other devices in the network, for example for TAS evaluation (Requirement **A.4**).

7.2.3 Napatech Hardware Abstraction Module

As we have seen in the architecture description, each station is of some `type`, these types can then take several roles in the topology. For each hardware solution, we require specific type and role implementations. The set of hardware-specific `type` and `role` implementations ist grouped as the hardware abstraction module. The CM then selects, configures and initializes the components of the hardware abstraction module. During the course of this thesis we implemented a hardware abstraction module for the Napatech hardware we used. We present the implementation details of that module in this section. Figure 7.3 displays the hardware abstraction module in a class diagram with the relations to the control layer.

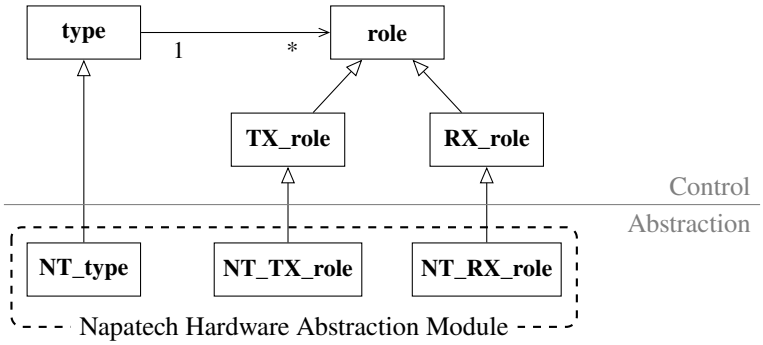


Figure 7.3: A class diagram describing the relations between the control layer and the abstraction layer, specifically the Napatech hardware abstraction module.

We used two different Napatech NICs as main transmission and capture cards in our measurements. For these cards, we implemented the `NT_type` and both the transmission and reception roles `NT_TX_role` and `NT_RX_role` using the Napatech API. Both of our NICs use the same API and are theoretically both capable of transmit and capture. However, due to their hardware differences they do not support all functionalities to the same degree. For example, the first adapter does not support hardware transmit time stamp injection, but has a higher time stamp resolution while capturing frames. Both cards use FPGAs in their architecture, for transmission and capture. Thus, by changing the FPGA image, the available feature set for in-hardware functions can be adjusted (of course, only with limitations). We chose to use the hardware time stamp injection feature on the second NIC, to have the transmit time information for all frames originating from ports on this NIC. Therefore, to know the transmission time of a frame, we did not require another capturing port as monitor of the sender. However, this decision was not without sacrifices because this FPGA image disabled other features, we initially wanted to use as well, forcing us to build workarounds for such missing features.

The TT transmission with absolute time stamps is an example for such a feature. To truly achieve TT transmission emulation, we must be able to transmit accurately to the absolute time reference. That means, we have to be able to start the transmission at specific points in time, coordinated with other devices. We require transmission capabilities on the absolute time scale, for example for TAS evaluations (Requirement **A.4**). Yet, the FPGA image for time stamp injection did not support this *Transmit On Time Stamp* feature, we initially wanted to use for this purpose. With this feature it would have been possible to directly set an absolute transmit start time. However, due to the FPGA image selection and the hardware limitations resulting from it, we had to perform the

synchronization of the NIC's transmission clock to the absolute timescale in software. However, thanks to time stamp injection and a feedback loop (based on LDP) in the framework, we did not suffer a loss in timing precision after all.

The Napatech NICs require a transmit time stamp for each frame, and can then schedule the TT traffic relative to the first transmitted frame. The transmission of the first frame sets the transmission clock equal to the time stamp in the frame [25]. After that, they transmit every other frame relative to the first one. For example, if the first frame time stamp is 0 and the second is 1,000, the NIC transmits the second frame 1,000 ns after the first. The main limitation of this over the aforementioned transmit on time stamp feature is that the transmission time of the first frame, relative to absolute time can not be controlled. Hence, all transmission times have no relation to an absolute time reference. To achieve synchronization to the absolute time, the framework implementation includes a mechanism with which the offset between transmission clock and absolute time can be compensated *after* the first frame was sent, circumventing these limitations. We extended the LDP to include this absolute timing feedback for the senders. The LDP frame is sent with some arbitrary time stamp (in our case it is the time of frame creation, but the value does not matter to functionality). A receiver of this frame calculates the offset between the frame time stamp to the injected transmission time stamp (i. e., the absolute time) and passes this offset to the sender. The frame time stamp is the value of the transmission clock at the time the injected time stamp was written into the frame. The injected time stamp is always an absolute time stamp. Hence, the transmission clock is behind the absolute time by the calculated offset. Thus, subtracting this offset from a frame's expected transmission time stamp results in transmission on the absolute time scale. Our testing showed that with this method, transmission on the absolute time scale can be achieved, accurate to the measurement uncertainty of the injected time stamp. Though, this solution is not without limitations. First, some framework-controlled receiver must capture the first frame during LDP and secondly, the frame must be a part of an expected stream to this receiver. The receivers can only propagate timing feedback to the senders of expected streams.

7.3 Measurement Details

This section describes the details of the framework control mechanics during the measurement execution. We first discuss the transmission side, which includes the frame creation and transmission selection functions, followed by the overall transmission control methods. After that, we move to the capturing side of the measurement, discussing the capture control methods.

7.3.1 Frame Creation and Transmission Selection

As we have discussed previously, we use an on-demand transmission selection and frame creation approach for our transmitters. This was mainly due to the high memory requirements of pre-calculated traffic queues for every sender. This could limit the number of supported transmitters in framework, thus we decided against it. For our on-demand solution, we opted for the algorithmically simplest and most deterministic solution in the form of a single-threaded selection, creation and transmission approach. The single-threaded solution avoids synchronization between a frame creation thread and the transmission thread. However, we have to optimize the selection and creation operations for the required high data rates (Requirement **B.2**) and in-time frame creation (Requirement **B.1**).

We introduce a simplification of our scheduling problem, to reduce the cost of the transmission selection operation. Recall that transmission scheduling in the framework takes place in the form of streams. Each stream has a transmission time slot, in which the stream's frames can be sent. No frames are allowed to be transmitted outside the bounds their stream's time slot. The schedule timings of the transmission slots is non-trivial, e. g., they might be configured with different scheduling periods, causing the scheduling patterns of multiple streams to interleave. However, while the periods may very well be overlapping, we assume that the transmission slots themselves are non-overlapping. That means, at every sender, at most one transmit slot can be active at one point in time. This allows us to simplify the transmit scheduling, by applying a two-step scheduling approach. We split the transmission selection into a stream-based scheduling algorithm followed by a per-stream frame-based scheduling solution. This increases the transmission selection performance, by simply reducing the number of possible choices every time. We call this scheduling method *Stream-Based Transmission Selection*. We can assume that the stream slots are not overlapping without loss of generality because a valid schedule synthesized by a scheduling algorithm also shows this property. Moreover, there is no limitation to the number of streams for one sender; By creating many streams, each only responsible for one frame, we essentially disable the stream functionality and achieve traffic scheduling on a by-frame basis. As another simplification, the number of frames per stream slot is constant and known at configuration time. We then send one frame per configured receiver of the stream and keep the order of sender-receiver pairs constant over all transmission slots of one stream. The stream itself is responsible for initially defining the order of the individual frames in one slot, if there are more than one. This allows us to speed up the frame creation operation because the headers of the frames in one slot can be partly reused, even in the same order. Another advantage of this is that we do not need any particular scheduling algorithm on a per-frame basis. The scheduler for each sender only has to select streams by their slot description and can then schedule all frames in the next slot. The transmitters move the frames of one transmit slot at a time to the NIC's memory, as the frames inside the slot are already in the correct transmission order.

The remaining problem is the stream slot selection algorithm. The problem with that is that we can not assume equal periods, with which the individual stream slots are rescheduled. This means that we can not directly infer the relative order of the transmission slots. Specifically, stream A's slot period might be four times as long as the period time of stream B, which would result in four slots of B during the same time, A transmits only one slot. While there theoretically exists a *Hyper Cycle* that contains all smaller periods, this is, in general, an extremely large value for arbitrary period lengths (one trivial solution of this would be the least common multiple of all stream slot periods). Thus, to not limit the framework to special cycle times, where the hyper cycle is trivially calculable (and small), the hyper cycle and the slot order inside it is never calculated. Instead, we use the assumption that a scheduler ensures non-overlapping transmit slots, and use a fast data structure that extracts the *next* stream according to the start time of its next slot.

Specifically, we used an AVL tree which holds the streams and the start time stamps of the next transmission slot. This allows to extract the next stream slot in $O(\log n)$, where n is the number of streams. After the transmission is done, we reinsert the same stream, again in $O(\log n)$. Furthermore, we optimize our AVL tree implementation for extraction and re-insertion tasks, by reusing the previously extracted node. Thus, the stream selection and with that the frame selection is complete. We used an efficient way of selecting the next transmit slot, ensuring in-time frame selection and creation (Requirement **B.1**) for high possible transmission data rates (Requirement **B.2**). We validate the performance in our evaluations (cf. Section 8.2).

7.3.2 Transmission Control

We now know that we can use the `scheduler` component to quickly and efficiently select the next frames for transmission. Thus, in our transmission controller we can concentrate on the overall measurement execution. We implement the transmission control functionality in the `TX_role`. In the framework, the transmission group's runner thread executes the transmission control functionality (cf. Section 6.2). To achieve the tasks of the measurement, the `TX_role` uses two interfaces: First, it uses the *scheduling* interface to the transmission scheduler. Secondly, it uses the *transmission* interface to the abstraction layer, for hardware control functions and frame transmission. Figure 7.4 shows these interfaces and their relations to the transmitting components in a class diagram.

We use the *scheduling* interface to interact with the transmission selection and frame creation methods. It comprises three methods: First is the `initialize()` method, used to create the first set of transmit slots. During the transmit loop, we use `get_next_headers()` to retrieve the set of frames (i. e., `measurehdr` structures) corresponding to the next stream's transmission slot. Finally, the `reschedule_slot()` method creates the next slot's contents, reusing large parts of the previous slot. Furthermore, the scheduler reinserts the stream and its new slot into the scheduling data structure. With this, the framework only holds the next slot for each stream in local memory, for reduced memory pressure. The second interface is the *transmission* interface. It is kept very generic, to specifically account for a broad variety in measurement hardware, each with their corresponding implementation (Requirement **E**). The interface comprises three methods, two of these are the initializer (`pre_transmit()`) and de-initializer (`post_transmit()`) of the transmit functionality of the API. The last method, `transmit()`, implements the transmission of a single frame.

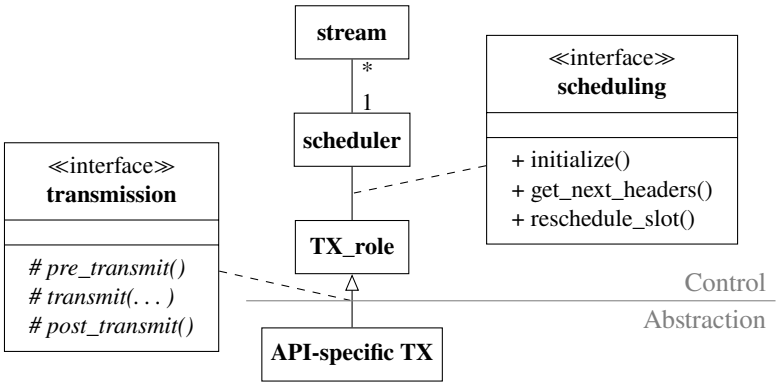


Figure 7.4: A brief overview over the classes partaking in frame transmission and stimulus injection.

Algorithm 7.1 shows the `TX_role`'s main transmission control method we describe in the following paragraph. The transmission control function starts by initializing the scheduler. After that it calls the `pre_transmit()` method, instructing the hardware to prepare for transmission. The abstraction layer implementation of this method determines, what this function call implies specifically. After that, all transmission and capture roles are synchronized with a barrier, to allow every role to fully initialize. Next, the previously mentioned LDP is executed (cf. Subsection 7.2.2). The end of LDP is again synchronized with a barrier. Now the transmission loop of the `TX_role` starts. It uses the stream scheduler to get the next slot, transmits all frames in it and then reschedules the slot. The transmission of one frame is done by calling `transmit()` on the underlying hardware-specific

abstraction layer. Once all frames were moved to the NIC, the transmission controller terminates, calling the `post_transmit()` method to close down the API or perform other clean-up tasks. The scheduler automatically deletes all frames that are in memory and terminates as well. After this, the role's runner thread is deleted.

Algorithm 7.1: The TX_role::run(), executed by the transmission thread.

```
streamScheduler.initialize();
pre_transmit();

preLDPBarrier.wait();
TX_discovery();                                          // Run the TX side of LDP
postLDPBarrier.wait();

/* Transmission ends if requested by the user or if the desired number of frames was
   transmitted.                                                                    */
while keep_transmitting do
    slot_t* slot = streamScheduler.get_next_slot();          // Stream TX slot selection
    for measurehdr* header in slot do
        transmit(header);                                   // single frame transmission
        ++txPacketCounter;
    end for
    streamScheduler.reschedule_slot();
end while

post_transmit();
```

7.3.3 Traffic Capture

The receivers use a control method similar to that of the transmitters. The main capture control method is implemented in the `RX_role` class. It uses two interfaces to interact with the surrounding layers: First, it uses the *capture* interface to receive frames from the abstraction layer. Secondly, it outputs the captured data to presentation layer, using the *output* interface. We discuss the capture interface in this section and move the explanation of the output interface to the next section, where we discuss the details of the data aggregation implementation. Figure 7.5 shows the capture interface and the relations between the receiving components in a class diagram. The *capture* interface consists of four methods: First, we have the `pre_receive()` method that configures the hardware for capturing tasks. Secondly, the `receive()` method returns a single frame that was previously captured on the hardware. This method should guarantee non-starvation and include a timeout, if no frame is received. The third method `release_packet()` declares that all processing of the frame is finished. This allows the NIC to free the memory that was allocated for the frame. After a call to this method, the previously received frame pointer should no longer be used as this may cause undefined behavior. Finally, we use the `post_receive()` method to close down the capture interface of the API or perform any other clean-up tasks in the hardware-specific implementation. Hence, the capture interface represents a generic high-abstraction level reception interface to the underlying hardware-specific implementation. This allows the framework to operate with a multitude of different capture solutions (Requirement **E**).

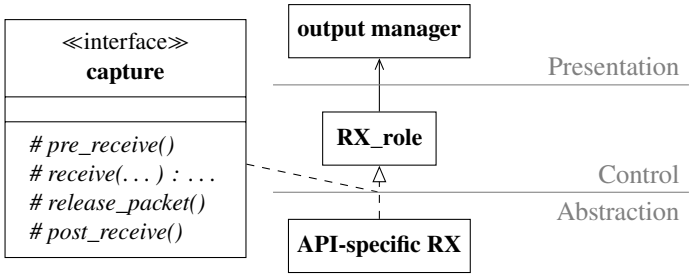


Figure 7.5: The main classes involved with traffic capture and the high-level capture interface used for traffic capture. The interface to the output manager (OM) is omitted

Algorithm 7.2 outlines the capture control method implemented in the `RX_role`. During startup, the `RX_role` instance calls `pre_receive()` to configure the underlying hardware ports for capture. After the configuration is done, we execute LDP similarly to the transmission side of the framework. Before and after LDP are barriers, synchronizing the `TX_role` and `RX_role` threads, ensuring that every station in the network is ready. Because of LDP, the receivers know the streams and number of frames to expect (`expectedFrames` in the algorithm). In each iteration of the capture loop, the capture runner thread calls `receive()` to get a captured frame from the hardware-specific implementation. With the captured frame, we decide whether the frame belongs to the measurement or not. This decision is based on the `ID` field in the `measurehdr` of the captured frame (cf. Subsection 7.1.2). If the frame belongs to the measurement, we passed it to the OM. If the ID does not match the expected value, i. e., the frame does not belong to this measurement, the frame can not be expected to have the `measurehdr` data layout. Hence, we extract the EtherType of the frame and store it along with capture time and other Layer 2 header data. The EtherType specifically is stored, to know what protocol caused the cross-traffic. After we moved all data to the OM, the controller calls `release_packet()` to free the memory of the frame. Once all frames from each discovered stream are captured, we call the `post_receive()` method, deleting all needed variables and closing the API connection. If the user sends a `SIGINT` (i. e., `Ctrl+C`) before that time, the framework forces the end of all ongoing transmission and capture operations. This way, open API connections of the individual role implementations are closed, terminating the program safely. The capture controller moves the data it gathers with every frame to our output methods. The next section is concerned with the interface we use for that purpose and other implementation details of the data output.

Algorithm 7.2: The RX_role::run() executed the receiver thread.

```
pre_receive();
preLDPBarrier.wait();
RX_discovery();                                          // Run the RX side of LDP
postLDPBarrier.wait();
/* Capture will end either if requested by the user or if the expected number of
   frames was captured.                                                          */
while keep_receiving do
    // A capture object contains the header along with the receive timestamp
    capture_pkt_object_t* captureObj = receive();
    --expectedFrames[captureObj.header.vid];  // Count the frames per discovered stream
    outputHandler.write(captureObj);
    release_packet();
end while
outputHandler.write_through();
post_receive();
```

7.4 Output Implementation Details

In this section, we present the details of the data output mechanics deployed in the framework. We start with general data aggregation at the OMs, used by the capturing stations. After that, we highlight some details of the SQLite3 format output implementation we used as the main output format and interface to the evaluations. We finish this chapter with a brief overview of the developed evaluation methods already included in the framework. This includes details about the processing we perform with the gathered raw data, as part of our evaluations.

7.4.1 Memory Buffer of the Output Manager

The data output functionality consists of two classes working together. Figure 7.6 is a class diagram that shows the classes involved in the data aggregation and formatting tasks, along with the interfaces between them. The first class, the OM, provides the interface between the traffic capture control layer and the presentation layer. This class has two purposes: First, it ensures the the high data rate output requirements for capturing (Requirement **C.2**). For this, it implements a memory buffer for the frame data. Secondly the OM provides a unified output interface to the capture controller, i. e., independent of the final output format. The second class, the *format writer*, allows for flexibility in the output formats. Therefore, we differentiate between the format-independent *output* interface as an abstraction for different output formats and the format-specific *format output* interface, to accommodate the flexibility. By splitting the interfaces, we get a clear separation of format-specific code to the unified output interface used in the control layer.

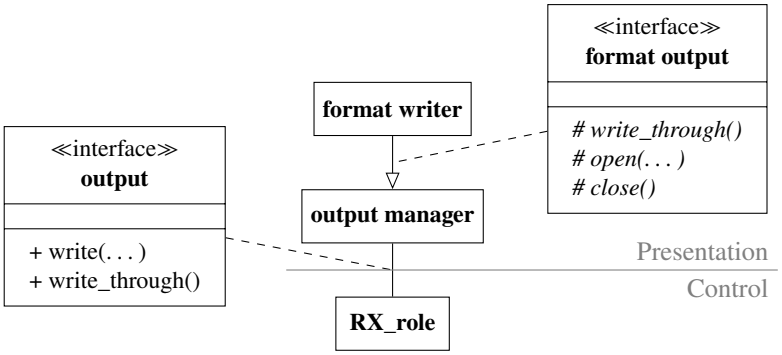


Figure 7.6: The classes responsible for the data output tasks of the framework and the interfaces between them.

The OM's *output* interface, used by the capturing components implements only two methods. Thus, we can keep the interface very simple. The first method is `write()`, which passes the given measurement data over to the OM. This keeps the data in the OM's memory. We use the second method, `write_through()` to clear the current memory buffer and move its data to the final output format. For this final write operation, the OM uses the *format output* interface implemented by a *format writer*. This interface comprises three methods: It implements basic methods for opening and closing a handle to the output file with the methods `open()` and `close()`. Furthermore, it implements a medium or format-specific implementation of the `write_through()` method. This must be called after `open()` and before `close()`, and writes the data into the final format.

The OM must fulfill the required high output data rates, even when writing to a very slow medium or format. The OMs must support this even when multiple receiver threads try to write to the same output file (Requirement **C.3**). In this case locking can occur which can then quickly cause frame drop on the receivers. To circumvent this and achieve zero-loss capture capabilities (Requirement **C.2**), every receiver uses its own OM instance, which holds a dynamically resizeable memory buffer for the measurement data. With `write()`, the frame data is written to the buffer, while `write_through()` flushes the current buffer contents to the final output format. We implemented the buffer to dynamically resize, if the current capacity is reached. For the resize, we multiply the current capacity with a constant value. The contents of the old buffer are then moved to the new buffer. The resize multiplicand can be passed to the constructor of the OM. The standard value for this parameter is the golden ratio (≈ 1.62), for a good trade-off between memory footprint and required number of resize operations. With this multiplicative buffer increase, we achieve amortized constant insertion time on the `write()` method. Thus, we implemented a fast output method between the receiver and the first stage of data storage. The initial buffer size can be controlled via a second constructor parameter as well (the standard here is arbitrarily chosen to 10 frames). We expect the format-specific output implementations to override the standard values, if needed.

7.4.2 SQLite3 Data Output

As main data output method and thus as interface for the evaluation components, we chose SQLite3 databases. The main reasons for this decision were the portability, ease of use and the powerful data access interface. Using the database functionality for relational pre-processing of data allows the evaluation components to be more light-weight. The `DatabaseWriter` class extends the OM with SQLite3 format-specific code. It implements the *format output* interface. With `open()`, the instance opens a thread-safe connection to the database. This is a feature supported by SQLite3 for parallel database operations. The `write_through()` method performs a bulk insert into the database, using prepared statements. All insertions of one `write_through()` are encapsulated in a transaction. This way, the database does not write to persistent storage line-by-line, increasing the possible data rate. Calling `close()` closes the database connection. The `DatabaseWriter` destructor uses an instance counter, so that only the last `DatabaseWriter` shuts down the SQLite3 API, as the API shutdown is not a per-thread operation. With the SQLite3 format, we chose an industry-standard data storage method and provided a loosely coupled, inter-operable data output format (Requirement **G.2**). Thanks to the modular output architecture, other formats, such as the PCAP format, e. g., used by *tcpdump* [36], can be integrated in a simple way just by providing a format writer.

7.4.3 Evaluation Components

The evaluation components consist of multiple reusable functions and most importantly, a counterpart to the data output handler of the measurement components. Here, we implemented a database reader which can import the framework's output data into the evaluation chains. This is part of the `SQLite3DBInterface` that mainly implements two interaction functions, `getCols()` and `getRows()`. Both take a SQL command as parameter and return the database's output either as columns or as rows. This interface is used by the `CapturePoint` class that corresponds to one capturing station in the measurement's topology. This can already output some preprocessed information about the captured data, e. g., the per-frame latency, the arrival time stamps, the inter-arrival times, etc. Furthermore, we already implemented a set of plotting functions that can generate commonly used analysis plots, just by passing a `CapturePoint` instance. For example the latency or inter-arrival times over the course of the measurement, or plots directly visualizing the queuing time near a gate operation of the TAS are already implemented and can be reused.

As we have seen before, the header used for our measurements contains a set of values we use for our evaluation (cf. Subsection 7.1.2). A frame and the corresponding set of values is identified by three pieces of information, the capture point x, on which it was received, the stream's VID and the sequence number. The output handler of some capture point stores the following frame data for each received frame i. First, we have the capture time stamp $x.t_{\text{cap}}(i)$, representing the time the end of the frame was received on station x. Secondly, it stores the calculated transmit time stamp $t_{\text{calc}}(i)$ that is used to schedule the transmission of this frame. Next, the injected time stamp $t_{\text{inj}}(i)$ represents the actual transmission time (only if the sender hardware supports it). Lastly, we store the frame length ℓ_i (at Layer 2) in bytes. With these data points, we use our delay model (cf. Chapter 5) as a basis of our data processing tasks. Using our capturing ports in the topology, we can measure the ingress location times for these end stations. Similarly, we can even measure the

egress location times for our time stamp injection-enabled transmitters. Therefore, we can use the measured data to get information in our mathematical delay model.

$$x.\mathsf{t}_{\mathsf{loc,in}}(i) = x.t_{\mathrm{cap}}(i) - x.d_{\mathrm{hw,RX}} \quad \text{if } x \text{ is a receiver, and} \tag{7.1}$$

$$x.\mathsf{t}_{\mathsf{loc,out}}(i) = t_{\mathrm{inj}}(i) + x.d_{\mathrm{hw,TX}} \quad \text{if } x \text{ is a transmitter.} \tag{7.2}$$

Where $d_{\mathrm{hw,RX}}(x)$ is the RX hardware delay (cf. Section 2.3), i. e., the time the station x requires between capturing the frame and setting the corresponding time stamp. The delay $d_{\mathrm{hw,TX}}(x)$ is the time that passes between time stamp injection and the begin of transmission. We assume both of these delays to be measurable by the networking hardware. Note that the time synchronization between multiple NICs may be inaccurate, thus these measured data underly the inaccuracies of time synchronization. In our case, we ensure a highly accurate time synchronization between the transmitting NIC and the receiving one. The location time is, in general, not observable for *any* station $x \in V$ in the topology. Wherever we position a capture point, we gain information about the location times at that point. Using the measured data as ground truth, we can formulate approximations for the expected location times. Outgoing from the points, where we can measure the location time, we can propagate assumptions about the location times at other points in the topology. For this we use the Equations 5.1 and 5.2. We presented these rules in our system model (cf. Chapter 5) With these we can approximate an ingress location time at station y out of the egress location time at station x and the propagation delay of the link (x, y). Furthermore, we can approximate the ingress location time of a station x based on the egress location time of the same station and an approximation of the residence time $x.\mathsf{d}_{\mathsf{res}}$. The *real* values deviate from these approximations by the inaccuracies or the jitter of the network stations. The accuracy of these approximations is limited by the accuracy of the measured data. We use these rules and ground truth values to evaluate the framework and our test networks in the following chapter.

8 Evaluation

The contribution of this thesis is twofold: We implement a measurement framework and we use this framework to evaluate the behavior of real world networks. Therefore, we split our evaluation in two parts: First, we evaluate if our framework fulfills the requirements (cf. Chapter 4) and, secondly, we use the framework to measure the characteristics of real world TSN networks. We begin with our measurement environment, and a short reiteration of our measured data. After that, we present our framework analysis, evaluating the fulfillment of our requirements, Next, we begin our second evaluation part and analyze real world TSN-aware networks. With that, we provide detailed insight into the possible inaccuracies of network devices or networks as a whole. Specifically, we analyze real world TAS implementations, both for the effectiveness of the TAS and the possible inaccuracies. Finally, we present a method with which high ingress jitter can be contained, regaining bandwidth in the network at the cost of delaying one stream.

8.1 Measurement Environment

In our test setups, we use two Napatech Link™ NICs. First, we used a NT40A01-4x1 with four 1 Gbit/s SFP+ ports. This network card is capable of time stamping incoming traffic in hardware, with a resolution of 1 ns and, therefore, was used as the main capture card in our measurements (Requirement **C.1**). Furthermore, the 4 GB of on-board memory allowed for packet buffering, reducing the risk of packet loss on the receiver side. Secondly we used a NT40E3-4-PTP with four 1/10 Gbit/s SFP+ ports mainly for traffic generation. The FPGA image used for our testing enables the use of hardware time stamp injection, with a resolution of 10 ns on the outgoing traffic. For the TT traffic generation, we used the traffic scheduling capabilities of this adapter. The adapter buffers the outgoing packets in its on-board memory, while the set transmission time is still in the future. The transmit time stamps have a 10 ns resolution and are floored down to the next multiple of 10. We verified this accuracy in a verification measurement (cf. Subsection 8.2.1). The two Napatech cards support time synchronization using the NT-TS protocol to an accuracy of one nanosecond through a separate connection between the two devices [25]. This is a critical point for all delay measurements, as these measurements are only as accurate as this time synchronization. We configured a constant time synchronization offset of 15 ns, to compensate the signal delay between the two devices. This includes 8 ns for the output circuit delay, 1 ns for approximately 20 cm of cable and 6 ns for the input circuit delay [25].

For networking hardware, we used different switches that were all TSN-capable. Primarily, we used two Hirschmann™ RSPE35 switches, each implementing the TAS on three ports, each at 1 Gbit/s. On these, we used the firmware version "`HiOS-3S-TSN-08.4.00-A_BR_I2`". Secondly, we used a Hirschmann™ BRS40 with 12 TSN-capable 1 Gbit/s ports. On the BRS40, we used a TSN-capable

beta firmware version. Between all networking devices, we used SFP+ fiber optic transceivers at link speeds of 1 Gbit/s. Only optical signals can be replicated, without changing their timing. Therefore, we use only passive fiber-optic taps, adding no jitter or latency and satisfying Requirement **D**.

Figure 8.1 shows our time synchronization network between our NICs and the switches. We synchronized all our network switches using IEEE 1588 PTP [19] in BC mode, via a dedicated set of links. Thereby, the data links are free of PTP traffic that might impact the measurement results. Moreover, synchronization can still be done even if the data links are under full load. We selected the transmitting Napatech NIC as the PTP grandmaster of the network and the NT-TS master. The receiving Napatech NIC is synchronized to the sender via the NT-TS protocol, ensuring high precision time synchronization in the measurement hardware. Because the number of hops between two stations can affect the time synchronization accuracy, our expected time synchronization errors are not the same for all switches. While this can make it difficult to compare two switches, it is also an unavoidable issue in real world networks. In our setup, the transmitting NIC is required to be the grandmaster. This made it difficult to measure the precise PTP offsets during each measurement. We had no capabilities of measuring the exact time synchronization accuracy of each device, e. g., with the use of a PPS signal. Therefore, we can only use worst-case error approximations for the time synchronization quality. Each switch is able to report the maximum time offset between itself and its master (i. e., the next hop on the way to the grandmaster). The reported maximum absolute values are between 70 ns and 160 ns per hop, with the actual deviations being at or below 60 ns most of the time.

The host computer executing the framework code and containing all measurement NICs was running Ubuntu 20.04.1 LTS with kernel version 5.4.0 on two Intel(R) Xeon(R) E5-2687W v3 3.10 GHz (40 virtual cores, 20 physical) and is equipped with 128 GB main memory. We used `gcc` version 9.3.0 to compile our framework for data acquisition.

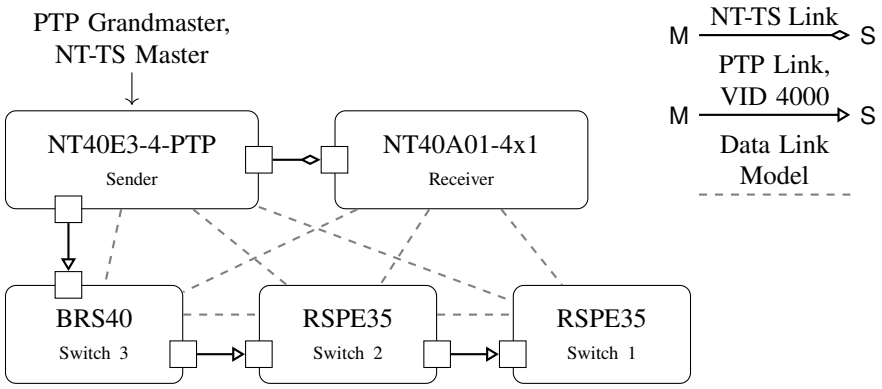


Figure 8.1: The time synchronization links between all used devices. The PTP links are completely separated from the data links used for the measurements.

8.1.1 Measured Values

For our measurement purposes, we collect several pieces of data for every frame. We already discussed the layout of the data inside our measurement frames in Subsection 7.1.2. All this data is called the *Measured Data*. Measured data values will be shown in regular math font, e. g., "d". Data that is derived form measured data using our models is called *Calculated Data*. They are the result of some evaluation and pieces of calculated data will be marked in sans-serif font, e. g., "d". Such calculations are based on our delay model (cf. Chapter 5). We discussed the way we use the measured data in our mathematical delay model in Subsection 7.4.3. To present a coherent view here, we quickly reiterate the measured data we have at our disposal. Let x be the station that captured the frame i. We store $x.t_{\text{cap}}(i)$ as the time at which the end of the frame was captured on station x. Secondly, we have $t_{\text{calc}}(i)$, which corresponds to the expected transmission time of this frame. Next, $t_{\text{inj}}(i)$ is the frame's injected transmit time stamp. Finally, we store the frame's length ℓ_i. We use these data and our system model as basis for the evaluations in the following sections. We begin with the framework evaluation, analyzing the fulfillment of our requirements.

8.2 Framework Evaluation

In this part of the evaluation, we evaluate the framework functionality and its performance characteristics, based on our requirements (cf. Chapter 4). First, we analyze the transmitter and receiver accuracies (Requirements **B.1** and **C.1**), by performing a propagation delay measurement (Requirement **A.1**). Next, we show the performance capabilities of our scheduling mechanisms (Requirement **B.2**). This evaluation shows the performance of our capturing and data output operations as well (Requirement **C.2**). We use these findings as basis for the evaluation of the multi-port framework capabilities (Requirements **B.3** and **C.3**). This concludes the framework functionality evaluation. We present the measurement capabilities for processing and queuing delays (Requirements **A.2**, **A.3**) in the second part of the evaluation. This second part also contains measurements on the TAS functionality (Requirement **A.4**).

8.2.1 Latency Measurement Verification

Since all TT traffic measurements rely on the senders to transmit accurately, we first of all conducted a measurement, comparing the measured results to our expectations. We did this on a constant-latency network, i. e., we connected the sender (s) directly to the receiver (r). Figure 8.2 shows our measurement setup for these evaluations. This way, no third device could introduce latency or jitter. Furthermore, we can exactly calculate the expected latency, as it is the propagation delay of the cable connecting the two devices.

$$(s \rightarrow r).\mathsf{d}_{\mathsf{lat}}(i) = r.\mathsf{t}_{\mathsf{loc,in}}(i) - s.\mathsf{t}_{\mathsf{loc,out}}(i) \overset{(5.1)}{=} d_{\text{pg}}(s, r)$$

With 10 m of fiber between the sender and receiver, the expected latency is 50 ns. The goal of this measurement was to visualize the errors in our measurement path, i. e., show the deviation of the measured value to the expected, calibrating the latency measurements. We calculate the

measured latency between the two stations as the difference between capture time and injected time, compensating the constant hardware delays

$$(s \to r).\mathsf{d}_{\mathsf{lat}}(i) \overset{(7.1,7.2)}{=} \left(r.t_{\mathrm{cap}}(i) - r.d_{\mathrm{hw,RX}}\right) - \left(t_{\mathrm{inj}}(i) + s.d_{\mathrm{hw,TX}}\right).$$

The capture time stamp has a resolution of 1 ns, the injected time stamp has a resolution of 10 ns, however we do not know the internal behavior of the transmitting network card. Furthermore, we do not know the accuracy of the hardware delays and the jitter of the two clocks. Because we synchronized the sender and receiver with NT-TS, we expect very little error in the time synchronization between sender and receiver. Larger deviations from the expected value are most likely due to transmission or capture clock jitter inaccuracies.

Figure 8.2 shows our test network topology for this measurement. We configured the transmitter to send 1×10^6 minimum size frames at approximately 67 % line rate. Figure 8.3 shows the results of our latency measurements. Overall, the measured frame latencies range from 74 ns to 108 ns, which results in a frame jitter of $(s \to r).\delta_{\mathsf{lat}} = 34$ ns. We see it as unlikely that the time synchronization is the cause for these errors. Moreover, we observe that even latency values are around twice as probable as odd latencies. We can not definitely explain this effect but assume that it is due to hardware clock jitter on the receiver side, where even time stamp values are more probable. Furthermore, the measured latencies are larger than expected. We would expect the clock of our measurement hardware to jitter symmetrically around the real value. This means, we would expect a symmetric latency distribution with its mean at 50 ns, where the clocks jitter in both directions symmetrically. While this is counter-intuitive to the physical lower bound of 50 ns latency, we would not expect the true value to be very improbable, if the measured values jitter only in the positive direction. With this hypothesis, we would see an absolute offset of approximately 41 ns from the expected value. Our hardware then jitters around that offset by about ±17 ns. Part of the jitter around the offset can be explained by the inaccurate injected time stamp of which we do not know the internal rounding and transmission behavior. The offset on the other hand is most likely introduced by other transmission or capture delays in the NICs which are not compensated by the hardware delays. For very high-precision measurements, these results are not ideal. Though, if we keep in mind that the PTP time synchronization quality from the NIC to other network devices is most likely in the sub-100 ns area, the measured inaccuracies do not negatively impact the measurement significance.

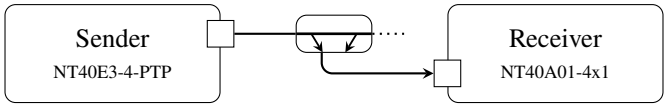


Figure 8.2: A basic setup of one receiving port directly monitoring the transmitting NIC. This setup is part of a larger network, hence the network tap.

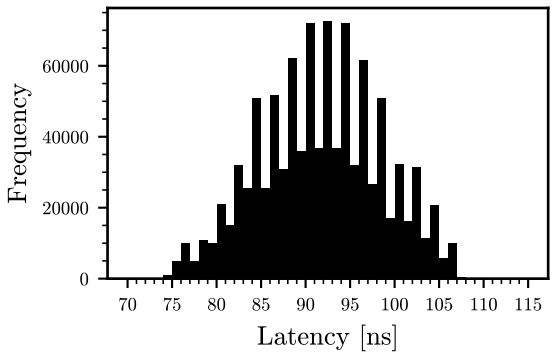


Figure 8.3: Latency histogram of one million frames through 10 m of fiber (propagation delay $d_{\text{pg}} = 50\,\text{ns}$).

8.2.2 Transmit Timing Behavior

To analyze the behavior of our transmitting ports we first analyze the traffic scheduling on the senders. As we have said, the TT traffic generation uses a calculated time stamp as the expected transmit time stamp. In this section, we compare the expected transmit time stamp to the measured values, evaluating the sender's accuracy and our framework's performance.

Definition 8.2.1 (Transmit Timing Error)
The Transmit Timing Error *is a metric for evaluating the sender's accuracy of TT traffic injection.*

$$\delta_{\text{tx}}(i) := t_{\text{inj}}(i) - t_{\text{calc}}(i) \tag{8.1}$$

If $\delta_{\text{tx},i} > 0$, the NIC transmitted the frame too late, if the value is negative, it transmitted the frame early.

The significance and accuracy of the transmit timing error is limited by the precision of the injected time stamp. Most importantly, late transmission can be caused by a too slow transmission selection and frame creation implementation. Hence, large positive errors could indicate that our framework implementation is not fulfilling its performance requirements (Requirement **C.2**). First, we use this metric to evaluate the accuracy of our transmitters under low load on the framework. Hence, we get a baseline on the expected accuracy and compare these results to high load circumstances in the next subsection.

We use the same network topology as above, and configured the transmitter to send 1×10^6 frames of minimum size, every 1,001 ns. The smallest step for the transmit scheduling on the Napatech NIC is 10 ns, however we give the expected transmit time stamp at a 1 ns resolution, which makes

rounding necessary in almost all cases. This way, we forced a transmit timing error with almost every frame. To ensure that the presented effects are unrelated to frame size or data rate, we repeated this measurement with different data rates and frame sizes. Figure 8.4 shows the results of our error analysis. For any calculated time stamp, the error is always within ±10 ns, while both rounding up and down is possible. The `NT_TX_role` implementation floors the calculated time stamp to a multiple of 10, before passing it to the hardware. Therefore the observed negative transmit timing errors are the expected behavior. However, for 25 % of all frames, a positive transmit timing error up to 10 ns can occur (i. e., the NIC transmits the frame up to 10 ns late). We have no information on the rounding behavior of the transmitting NIC. With that in mind, the observed results are likely due to hardware clock jitter on the sender side and are within the expected error of ±10 ns. For the exact cause of the error, we have analyzed the error in greater detail, finding that of 8 frames, 2 show the positive rounding error in a repeating pattern. However, an attempt to compensate these effects in software was unsuccessful because the underlying hardware behavior is unknown. That said, we can fulfill the required transmission accuracy of Requirement **B.1** because the transmission error is within the measurement inaccuracy of 10 ns. We now confirmed that the transmitting hardware is able to schedule frames for transmission with errors smaller than the measurement resolution. In the following section, we analyze the behavior under higher loads, with primary focus on the transmission selection and frame creation performance of the framework.

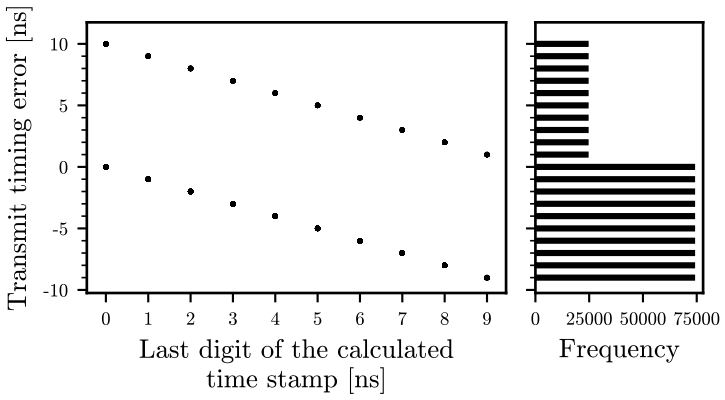


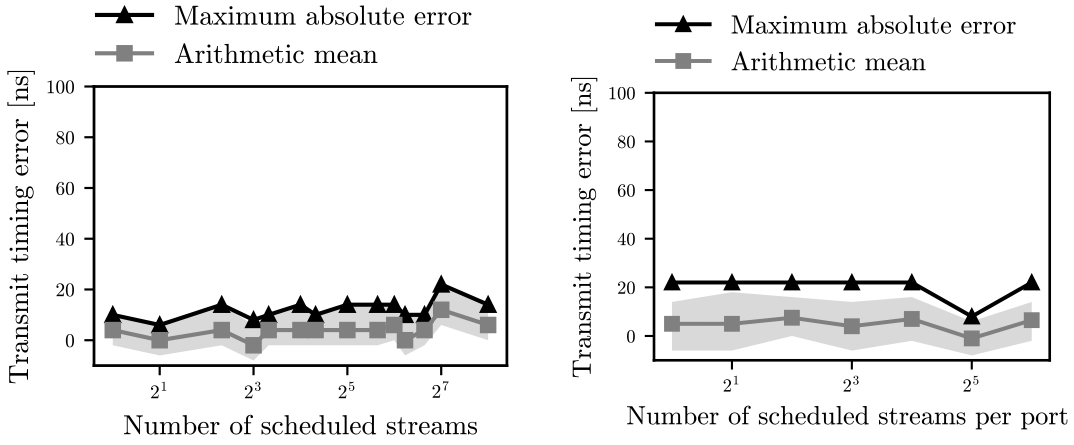
Figure 8.4: Analysis of transmit timing error on the Napatech NIC in relation to the expected transmit time. The injected time stamp and the TT transmission timing has a resolution of 10 ns. The expected transmit time stamp is given at a 1 ns resolution, hence rounding is necessary. The y-axis shows the error introduced in this process.

8.2.3 Framework Performance Capabilities

To verify the fulfillment of our performance requirements for high bandwidth transmission and capture capabilities (Requirements **B.2** and **C.2**), we measure the transmit timing error at maximum bandwidth (1 Gbit/s) with increasing number of scheduled streams. We have two different setups to force high loads on the scheduler and the host computer: First, we use up to 256 streams on a single transmission port, with one corresponding receiver. In a second measurement, we split these streams over all four hardware ports of the sender, scheduling up to 64 streams per port (Requirements **B.3** and **C.3**). Hence we measure the behavior under concentrated high loads on a single port and when

distributing the same high loads over multiple ports. The resulting traffic is captured by one port on the receiving NIC each. We transmit 1,000 frames per stream. This measurement shows multiple effects: First, it shows the transmit timing error at the data rate limit of every port. Secondly, we can measure the performance of our scheduling mechanisms. Lastly, we verify that our capture and output functions are capable of full line rate capture and data export. We created the worst-case scenario for the scheduler, where we schedule one minimum-size frame per stream slot. Thus, the transmitter invokes the transmission selection for every transmitted frame. Recall that multiple frames per stream require no scheduling because their order is fixed at configuration time.

Figure 8.5a shows the results of our evaluation for one transmitting port. The transmit timing error stays below a maximum absolute transmit timing error of 20 ns, up to at least 256 scheduled streams on one transmitter. This error is slightly higher than the behavior seen before. Because we transmit back-to-back now, the previously seen rounding error can add up. When splitting the same load over multiple ports, we expected results similar to the single-port scenario because our host computer can provide the additional 6 processing cores for the transmission and capture threads. Figure 8.5b shows our measurement results for this case. As expected, we see similar performance, albeit with higher absolute errors just over 20 ns compared to a single port. That said, the 90 % confidence interval is almost unchanged to the previous test. If we take into account that scheduling problems commonly result in less than 10 TT streams per transmitter (e. g., [11]), each with relatively low data rates, these results prove that our transmission capabilities exceed the bandwidth requirements. Furthermore, no frame was dropped in any of the measurements presented here, proving the effectiveness of our OM in fulfilling Requirement **C.2**.



(a) one transmitter, one receiver

(b) four transmitters, four receivers

Figure 8.5: The overall transmit timing error over increasing number of scheduled streams. The shaded area shows the 90 % confidence interval.

8.3 Real World Network Analysis

To show the capabilities of our measurement framework in high-precision measurements, we analyze the behavior of our network switches under different circumstances. First, we measure the processing delays of our switches (Requirement **A.2**) and prove that while the processing delay is not a constant value, it is frame size-independent. Next, we test the TAS implementations of our network switches, first showing the conceptual effect of the shaper, and secondly, highlighting inaccuracies in the implementations that must be considered in real world switches and networks. The measurements in this section have the goal of understanding each switch's behavior in an isolated scenario. Hence, we used a simple line topology comprising our sender, the switch under test and finally the receiving NIC (cf. Figure 8.6).

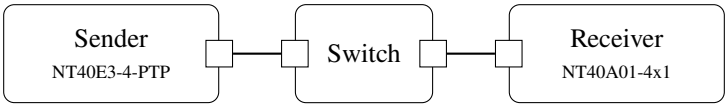


Figure 8.6: The network topology used for our network device analysis.

8.3.1 Processing Delay of TSN-Aware Network Switches

We begin with a processing delay analysis, to learn about the behavior of a switch, independently of any TSN features. Moreover, we use the gathered information for further analysis. With an estimation for the processing delay, we can measure the queuing delay and therefore the behavior of the TAS with higher accuracy. We used the RSPE35's high-speed ports 1 and 2, with TAS enabled but all gates open at all times. We discovered that disabling the TAS or using non-TSN-aware ports can result in completely different processing delays on this switch. We include such a comparison in our TAS evaluation in Subsection 8.3.3. On the Hirschmann BRS40, we disabled TSN entirely to disable the gates.

Let t be the transmitter, s the switch and r the receiver. To calculate the processing delay of each switch, we calculated the residence time

$$
\begin{aligned}
s.\mathsf{d}_{\mathsf{res}}(i) &\overset{(5.3)}{=} s.\mathsf{t}_{\mathsf{loc,out}}(i) - s.\mathsf{t}_{\mathsf{loc,in}}(i). \\
&\overset{(5.1)}{=} \big(r.\mathsf{t}_{\mathsf{loc,in}}(i) - d_{\mathrm{pg}}(s,r)\big) - \big(t.\mathsf{t}_{\mathsf{loc,out}}(i) + d_{\mathrm{pg}}(t,s)\big) \\
&\overset{(7.1,7.2)}{=} \big(r.t_{\mathrm{cap}}(i) - r.d_{\mathrm{hw,RX}} - d_{\mathrm{pg}}(s,r)\big) - \big(t_{\mathrm{inj}}(i) + t.d_{\mathrm{hw,TX}} + d_{\mathrm{pg}}(t,s)\big) \\
s.\mathsf{d}_{\mathsf{res}}(i) &= r.t_{\mathrm{cap}}(i) - t_{\mathrm{inj}}(i) - r.d_{\mathrm{hw,RX}} - t.d_{\mathrm{hw,TX}} - d_{\mathrm{pg}}(s,r) - d_{\mathrm{pg}}(t,s).
\end{aligned}
$$

Or short, we can subtract the propagation delays from the latency $(t \to r).\mathsf{d}_{\mathsf{lat}}(i)$ to get the residence time. The residence time contains the processing delay, queuing delay and the transmission delay

$$
s.\mathsf{d}_{\mathsf{res}}(i) \overset{(5.2)}{=} s.d_{\mathrm{pc}}(i) + s.d_{\mathrm{qu}}(i) + s.d_{\mathrm{tr}}(i).
$$

To analyze the processing delay, we ensured zero queuing delay, by sending with low enough bandwidth. We configured the transmitter to send 2.5×10^6 frames of any wire length between minimum and maximum frame size, with a constant inter-arrival time of 26 µs. A frame of maximum length takes just over 12 µs to transmit, therefore the link is always less than 50 % loaded. Thus, the

egress port is always free for transmission, whenever a new frame arrives. Because both switches use Store-and-Forward, the residence time contains the transmission delay for the entire frame length. Therefore, we expect the residence time to increase linearly with the frame size. However, we expect the processing delay to be frame size-independent. Therefore, we calculate the processing delay as follows

$$\begin{aligned} s.\mathsf{d}_{\mathrm{pc}}(\mathsf{i}) &= s.\mathsf{d}_{\mathsf{res}}(i) - s.d_{\mathrm{tr}}(i) - s.d_{\mathrm{qu}}(i) \\ &= s.\mathsf{d}_{\mathsf{res}}(i) - \frac{\ell_i}{1\,\text{Gbit/s}} - 0. \end{aligned}$$

Figure 8.7 is a scatter plot of these calculated processing delay values subject to the corresponding frame size. Our results confirm the frame size-independence of the processing delay for both switches. However, the two switches behave very differently. Figure 8.8 shows a histogram of the processing delays for each switch. We see a more deterministic and generally faster switching behavior on the RSPE35 compared to the BRS40. Most notably, the maximum measured processing delay for the RSPE35 is 1,039 ns, the processing delay of the BRS40 on the other hand is at least 1,419 ns long. For the RSPE35, the average measured processing time is around 1 µs. Our measured values correspond to the user manual of the switch [13]. In the manual, the processing delay is specified between 924 to 954 ns. Subtracting our measurement error (approximately 40 ns, cf. Subsection 8.2.1), results in a measured processing delay close to the specified range. However, the range in which we measured the processing delay is wider, we measured around 82 ns, or about 2.7 times wider than the reference. This is because our measurement hardware itself jitters with around 30 ns. On the BRS40, our measurement error does not have such a significant impact because the measured processing delay jitter is over 10 times larger than our measurement jitter (363 ns). This proves that the processing jitter can be of significant size, thus can have influence on the timing behavior of TT traffic. Our evaluation shows that different switches can show very different behavior, not only in their overall processing delay, but also the jitter they introduce on the traffic.

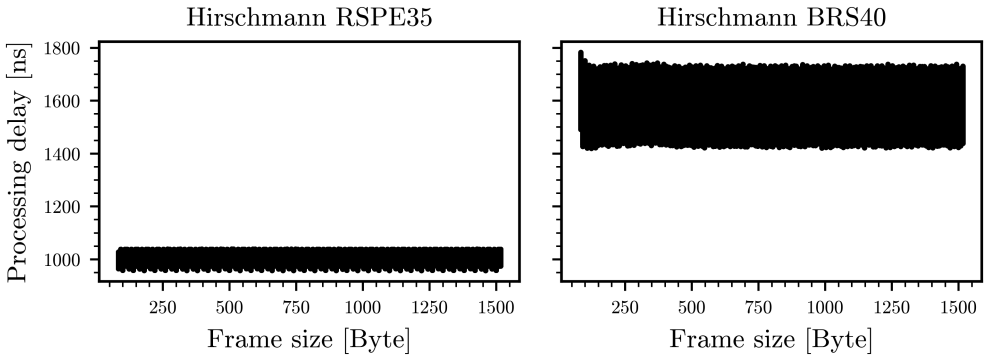


Figure 8.7: Calculated processing delay over frame size on the two switches.

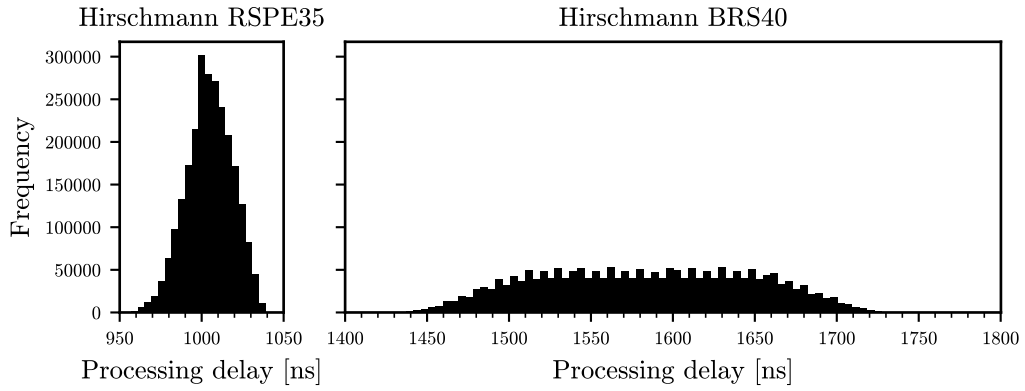


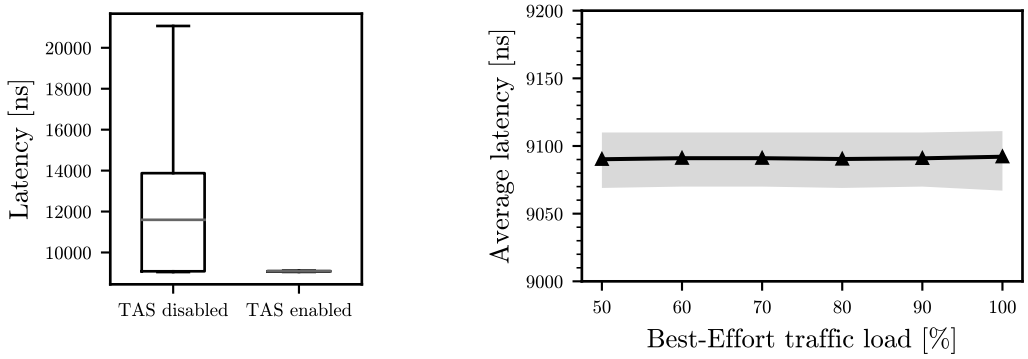
Figure 8.8: Histogram of the calculated processing delay of each switch.

8.3.2 Verification Measurements on the Effect of the Time-Aware Shaper (TAS)

In this section, we show the effect of the TAS on high-priority traffic under varying best-effort cross-traffic. For our specific measurement setup, we took the simulations from [28] and [21] as a basis. We show that the TAS can isolate high-priority traffic from other network traffic, achieving deterministic latencies. We come to the same conclusions as the related work, verifying their simulated results with our real world test. Our test setup comprises two transmitters on two different ingress ports of the RSPE35 switch. One sender is for high-priority traffic generation and the other for best-effort load on the system. The switch is connected to our receiver, thus both traffic classes are merged together onto one link. We adapted the GCL configuration from [21]. The GCL contains three intervals, the first is the *Protected Window*, where only gate 7 is open. Next is the *Unprotected Window*, here the gates 0 through 6 are open. In the third interval, the *Guard Band*, all gates are closed. We reproduced the measurement with the same high-priority frame size of 1,030 B. However, we increased the protected window size from 8.24 µs to 10 µs, to account for inaccuracies in the gate operation. This is the first difference between the jitter-free simulation and the real world. Reducing the gate-open time to *exactly* the transmission time of one frame results in congestion. This is due to internal delays that have to be considered in the gate open interval length [13]. The guard band duration was untouched at 9.9 µs, but we decreased the size of the unprotected window to 980.1 µs, to account for the larger protected window. We scheduled the high-priority traffic every 1 ms, i. e., one frame per cycle.

As a baseline, we measured the latency at the worst case, 100 % best-effort load, with and without the TAS enabled (cf. Figure 8.9a). When the TAS is disabled, we see significant interference between the two traffic classes in the network. This is visible in the form of a relatively high latency variation, caused by queuing. Note that in our setup, no more than one best-effort frame can be in the queue before a high-priority frame. This corresponds to a maximum queuing delay of the transmission delay of one best-effort frame of maximum size (12.1 µs). The latency of the high-priority frames always contains the transmission delay (8.2 µs) and the switch's processing delay (approximately 1 µs). We observe a maximum latency of up to 21.3 µs when the TAS is disabled, this corresponds exactly to the sum of all the mentioned delays.

However, with the TAS enabled, the best-effort traffic no longer has any effect on the high-priority latency. We then see an end-to-end latency that consists only of the transmission delay and processing delay. Figure 8.9b shows that the latency and jitter of the high-priority traffic stay unchanged under changing best-effort traffic load on the link. This even holds, if the the link between switch and receiver would be completely saturated by best-effort traffic. Our results verify that the TAS can guarantee low latency deterministic traffic compared to standard Ethernet, and can even fulfill these guarantees independently of lower priority loads, given the correct GCL configuration. These results correspond to the simulated network evaluations performed by the authors of [28]. However, the evaluations here were designed so that possible effects of the switch's inaccuracies have no effect. In the following section, we take a closer look at these possible errors in the TAS implementations.



(a) Comparison of the same traffic flows, with and without TAS at 100 % best-effort load

(b) Effect of best-effort load on high-priority traffic, the shaded area is the 90 % confidence interval of the latency.

Figure 8.9: Effect of the TAS on high-priority traffic under different best-effort loads.

8.3.3 Timing Error of the Time-Aware Shaper (TAS) Gate Operations

Previously, we have shown the possible effect of the TAS on small-scale networks with high best-effort load. However, we did this with a large margin for error, which allowed a conceptual evaluation of the effectiveness. In this section, we analyze the gate operations in more detail, with focus on the accuracy and precision of the switch, rather than the general functionality of the TAS. Specifically, we investigate the gate-opening operation of the high-priority gate. We highlight inaccuracies that are present on network switches but are commonly disregarded in network simulations or scheduling algorithms. We argue that these inaccuracies should be considered in the network's traffic schedule.

The network topology is unchanged from the previous measurements. The sender, switch and receiver form a line in that order. For our GCL configuration, we chose the cycle length at 1 ms. However, we change the GCL configuration from the previous measurements. This is due to a bug in the hardware or software of the Hirschmann BRS40 that was discovered by our measurements. We found that with the GCL configuration of the previous measurements (protected window, unprotected window, guard band), the high-priority gate opens over 7 µs too early. This was clearly an error in the switch and not due to inaccuracies in our measurements. Over a larger analysis of

this error, we found that changing the GCL preset results in more accurate results (guard band, protected window, unprotected window). However, on this second preset, we found that the guard band size had a significant influence on the accuracy of the high-priority gate-opening operation. For larger guard band lengths, the gate opened too soon, whereas a smaller guard band caused the gate to open too late. Due to these errors, our results on the BRS40 may no longer reflect the switch's behavior after the bug-fix. For the measurements presented in the following, we chose this GCL configuration on all switches: First, we set a 100 µs guard band (at this size we saw comparable results between BRS40 and RSPE35). Secondly, the high-priority gate is open for a 200 µs protected window. The remaining 700 µs are the unprotected window for best effort traffic (priorities 0 through 6). We configured our senders to send one high-priority frame (222 B size) per cycle, to prevent queuing. The frame's arrival time within the cycle length changes with every cycle, which allows an evaluation of the latency depending on the frame's arrival time. We sent 200,000, distributed over an analysis window of width 100 µs around the expected gate open time (100 µs). Thus, we have 20 samples for each arrival time stamp shown below. Additional to the previously evaluated switch configurations, we add a second RSPE35 configuration: We changed the ingress port of our traffic to the non-high-speed port 3. We do this to highlight internal differences in a switch as well as differences between multiple switches.

Our goal is to gain knowledge about the gate operations: First, we analyze the gate accuracy, i. e., the offset with which the switch executes the gate operation too early or too late. Secondly, we analyze the gate event precision, i. e., the the gate jitter or the noise on the gate operation time around that offset. The state of the gate influences the queuing behavior of the switch. Therefore, we compare the time at which the switch starts the transmission of a frame relative to the frame's arrival time at the switch. We know that the switch processes a frame, before it enqueues it and the gate can come into effect. Thus, we compensate the processing delay to show the departure time of the frame relative to the time at which the frame was ready for departure. We calculate the *Ready-For-Departure Time* of each frame as the ingress location time plus a processing delay compensation $s.\mathsf{d}'_{\mathrm{pc}}$. Keep in mind that the ingress location time is an end-of-frame time stamp, the switch already received the entire frame and we do not have to compensate the transmission delay for Store-and-Forward. Let t be the transmitter, s the switch and r the receiver, we then get

$$\begin{aligned} s.\mathsf{t}_{\mathsf{rfd}}(i) &= s.\mathsf{t}_{\mathsf{loc,in}}(i) + s.\mathsf{d}'_{\mathrm{pc}} \\ &\overset{(5.1,7.2)}{=} t_{\mathrm{inj}}(i) + t.d_{\mathrm{hw,TX}} + d_{\mathrm{pg}}(t,s) + s.\mathsf{d}'_{\mathrm{pc}}. \end{aligned}$$

It is important to keep in mind the variance of the processing delay, as it has an effect on this calculated ready-for-departure time. Because we subtract a constant approximation for the processing delay, the variance is still part of the result. Therefore, the resulting $s.\mathsf{t}_{\mathsf{rfd}}$ values are only as accurate as the processing delay approximation. Furthermore, the accuracy of our time synchronization quality between the NICs and the switch has an influence on our measurements. In our plots, we display the ready-for-departure time relative to the cycle time, i. e., we calculated $s.\mathsf{t}_{\mathsf{rfd}}(i)$ modulo the cycle length.

For the *Departure Time*, we model the time at which the switch started the transmission of frame i. Hence, we compensate the frame length on the egress location time at the switch

$$s.\mathsf{t_d}(i) = s.\mathsf{t_{loc,out}}(i) - \frac{\ell_i}{1\,\text{Gbit/s}}.$$

Using the capture time stamp of our receiving end station results in

$$\begin{aligned} s.\mathsf{t_d}(i) &\overset{(5.1)}{=} r.\mathsf{t_{loc,in}}(i) - d_{\text{pg}}(s,r) - \frac{\ell_i}{1\,\text{Gbit/s}} \\ &\overset{(7.1)}{=} r.t_{\text{cap}}(i) - r.d_{\text{hw,RX}} - d_{\text{pg}}(s,r) - \frac{\ell_i}{1\,\text{Gbit/s}}. \end{aligned}$$

In our figures, we show this value modulo the cycle time as well. For our evaluations, we used the departure time to compensate the processing delay. If the gate is open, we expect the departure time to be equal to the ready-for-departure time. Hence, we set $s.\mathsf{d}'_{\text{pc}}$ such that the mean difference between ready-for-departure time and departure time is 0, if the gate is open. The subtracted approximations are 1,111 ns for the RSPE35 using the high-speed ports (in 1, out 2). On the second RSPE35 configuration with the non-high-speed port 3, this resulted in 5,246 ns (in 3, out 2). On the BRS40 we compensated 1,658 ns. These values roughly correspond to the previously measured processing delays from Subsection 8.3.1. Note that changing the ingress port on the RSPE35 increased the average processing delay by a factor of almost 5.

If the gate is closed, we expect the departure time to be equal to the next gate-open time. Figure 8.10 shows the departure time of frames with a ready-for-departure time in an interval around the configured gate-opening operation. On all three configurations and if the gate is open, the departure time is, apart from processing delay jitter, equal to the ready-for-departure time – we compensated the processing delay to achieve this. The departure time is, as a lower bound, clamped to the true gate-open time. However, on all three configurations, we observe a gate-open time that is on average over 200 ns earlier than we would expect it. The expected gate open time is shown with the gray dotted line at 100 µs. Every point in Figure 8.10 corresponds to one entire GCL cycle. Hence, this error in gate accuracy seems too deterministic to be completely explained by time synchronization errors. Furthermore, we see the same error on both RSPE35 configurations, even though those are two separate devices. It seems likely that the observed error is a hardware constant, rather than a time synchronization error. In every case, the gate accuracy is not perfect. If it is in fact a hardware constant, it could be compensated by a scheduling algorithm. Moreover, the gate operations are often assumed to have perfect precision. Yet, our evaluation shows that this is not the case, we see the gate-open time jitters (gate jitter), even after considering the 30 ns jitter of our measurement hardware.

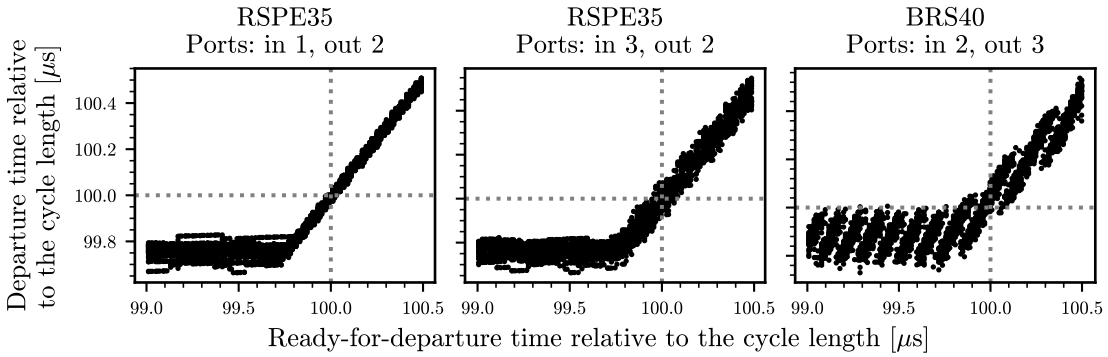


Figure 8.10: Departure time of frames with a ready-for-departure time near the configured gate-open operation (gray lines).

To quantify these gate inaccuracies and imprecisions, we use a different visualization. From the ready-for-departure time, we can calculate whether the gate is open at that time. The gate state gives us information about the time at which we expect the switch to start the transmission, i. e., the expected departure time. Comparing the expected departure time to the measured departure time visualizes the gate error and overall jitter. Figure 8.11 shows the difference between measured departure time and expected departure time (*Departure Time Error*). We differentiate between frames that were ready while the gate was open and frames that were ready while the gate was closed. This differentiation allows for specific analysis of the processing jitter in comparison to the gate jitter.

The latency of a frame that passes through the switch while the gate is open (black in the figure) underlies the processing delay jitter plus the previously accumulated jitter. Note that the mean of the black distribution is zero because we configured it that way. The jitter we observe if the gate is open corresponds to the previously seen processing jitter of the devices (cf. Subsection 8.3.1). Remarkably, the RSPE35 shows very different processing jitter when using the ports 3/2 compared to 1/2. The standard deviation is almost 3 times larger when using the (non-high-speed) port 3 as the ingress port (40 ns compared to 14 ns). The standard deviation on the BRS40 is even larger, at 4.6 times the standard deviation of the RSPE35 on ports 1/2 (at 64 ns).

On the other hand, the departure time of a frame that is at the front of the queue while the gate is closed does not depend on the frame's arrival time[1]. Its departure time only depends on the next gate-open time. Thus, for frames with a ready-for-departure time while the gate is closed, the departure time error shows the error of the gate operation (gray in the figure). In this case, we clearly see the gate operation inaccuracies, both in accuracy and precision. The mean errors of all three configurations show the gate to open too early by between 200 (on the BRS40) to 240 ns (on both RSPE35). Furthermore, our evaluation shows that the gate-open time jitters around this mean error. This error is approximately the same for both RSPE35 configurations (standard deviations 37 and 34 ns). Interestingly, the gate error is almost the same on the two RSPE35 configurations, while the processing error showed an almost threefold difference in precision. This can give an indication about the internal architecture of the switch, we conclude that the internal architecture previous to the queue changed to a less deterministic solution. The BRS40 shows an even less

[1] Apart from the fact that the frame must have arrived while the gate is closed.

deterministic behavior with an error standard deviation of 67 ns. We see that different hardware can behave in a very different way. Furthermore, the RSPE35 with the ports 3/2 proves that even the same switch can behave differently, depending on the hardware ports we use. These two facts make it even more difficult to account for the inaccuracies of the switch hardware in a real world network. If we wanted to take these inaccuracies into account in the synthesis of a schedule, we require knowledge about the switch's behavior on the specific port configuration we intend to use. Moreover, while a constant offset in the gate-open time could be compensated, we can only assume upper jitter bounds to compensate the imprecisions around that offset. However, reserving time for such imprecisions in the GCL, especially on larger networks, can result in other undesired effects, such as a significant impact on the available bandwidth of the network. In the next section, we detail this problem and its effects. Furthermore, we propose a solution to this problem in the form of a jitter-reduction mechanism.

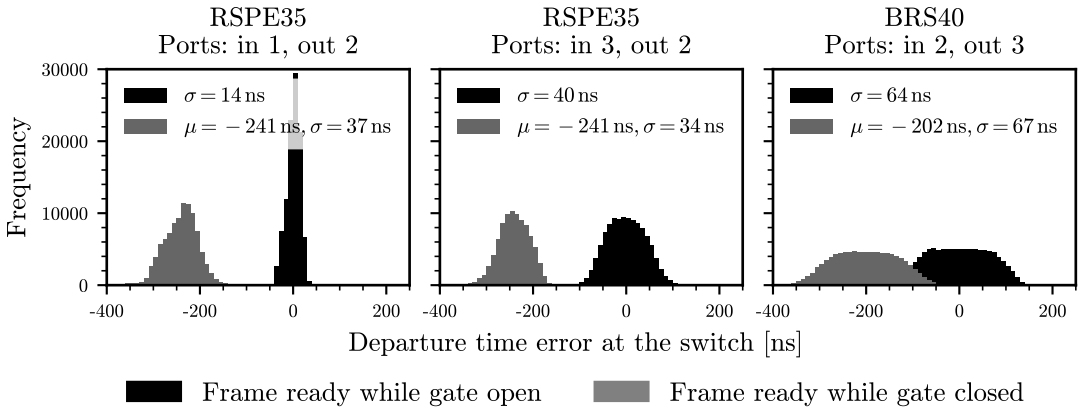


Figure 8.11: Transmission start time error. Negative values correspond to early transmission, positive values to late transmission.

8.3.4 The Effect of Queuing on Determinism

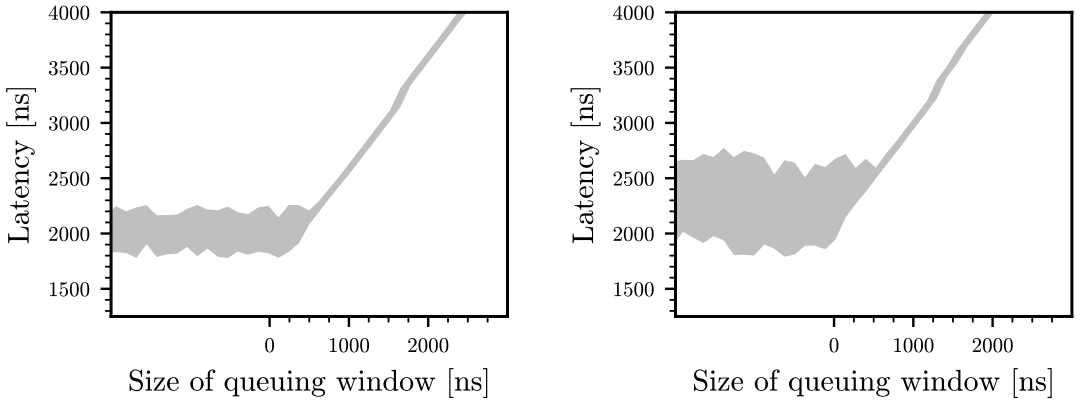
A problem with high-jitter devices, especially high-jitter transmitters arises in the synthesis of the traffic schedule for the network. An uncertain arrival time at a TAS-enabled switch requires consideration in the schedule. If a high-priority frame experiences maximum transmission jitter and, as a result of that, misses the gate-open interval, the frame would stay queued for an entire cycle, violating the latency bounds. We could solve this problem with two strategies: First, we can increase the gate-open interval to account for maximum jitter in the previous hop. However, this has one major drawback: Any jitter that was accumulated previous to queuing will carry over to the output, if the TAS gate is open. This can for example be the TT transmission jitter of a sender or the processing jitter of the switch itself. We then need to configure an even longer gate-open interval on the next switch in the transmission path because the current switch is not jitter-free as well. The same process applies for all following switches in the transmission path. Thus, we would need to increase the size of the gate-open intervals with every hop. This is undesirable behavior as it reduces the available bandwidth and could even render schedulability in larger networks impossible.

A second possibility for solving this problem is a compromise between latency, bandwidth and jitter. If a frame has to stay queued due to a closed gate, the previously accumulated jitter is removed, at the cost of increased delay. The dequeuing time (i. e., the transmission time) is then no longer dependent on previous jitter, but only the time at which the egress port becomes ready for transmission. After queuing, the frame jitter is equal to the gate jitter and the transmission jitter of the switch. In theory, we can configure the switch such that incoming high-jitter traffic always hits the closed gate and gets enqueued. Only after a *Queuing Window* time has passed, we open the gate to release the traffic. A switch with low output jitter and a large enough queuing window should be able to reduce the accumulated frame jitter on TT traffic. This way, we can contain the jitter of the incoming traffic with the help of the queue, introducing only the transmission and gate jitter of the switch on the next hop, in contrast to the switch's jitter plus the jitter of the previous hop. We have performed *Queue Jitter Containment (QJC)*. However, even for high-jitter senders, this queuing window mechanism is not without drawbacks. By doing so we have to reserve more bandwidth for the stream at this station because we have to isolate this traffic class for the queuing window in addition to the gate-open time. That said, we can regain otherwise lost bandwidth on the following hops because the stream's arrival time at the next switch has gotten more deterministic, thus the required gate-open length can be reduced. Moreover, during the queuing time of time high-priority stream, other traffic classes can be forwarded because they use a different queue. However, the latency of the jitter-reduced stream can increase significantly for two reasons: The first argument is trivial, we deliberately introduce a queuing delay which naturally increases the end-to-end latency. Secondly, we concentrate the arrival time of the stream at the next hops tightly around a later point in time. The jitter-reduced stream has, due to queuing, a higher average latency. Thus, the introduced jitter of following stations has a greater effect on the mean latency of the jitter-reduced stream. To reiterate, we add the same jitter in the following stations but on average at a later point in time, thus the effect of the jitter on the average latency increases. The QJC mechanism is a trade-off between worse latency for one stream at the benefit of improved determinism for this jitter-reduced stream and improved bandwidth utilization in the overall signal path.

Disregarding the errors of gate-opening inaccuracies and time synchronization, we would expect the queuing window to reach maximum effectiveness if the latest possible frame is still captured within it. Hence, in an error-free setup we would expect maximum effectiveness if the window has the length of the size of the maximum jitter. However, for the implementation of QJC in real world

networks, we require detailed knowledge about network's components and their behavior, as they have an influence on the effectiveness in multiple ways: First, we must know the ingress jitter on the switch, e. g., the transmitter's jitter. Secondly, we have to know the gate jitter of the switch on which we want to use QJC. We can only perform jitter reduction if the gate jitter is smaller than the ingress jitter. Thirdly, we should at least be able to estimate the gate inaccuracy. If the gate opens earlier than configured, the window must begin earlier by that error. Finally, we require an upper bound estimation on the time synchronization quality between the devices. The time synchronization error prolongs the queuing window by the maximum deviation in time synchronization. Because both these values can increase the required window size, they are important for the trade-off assessment of QJC. The loss of bandwidth on the following hops must justify the increase in latency on the jitter-reduced stream.

The queuing window can be implemented in two ways, either the transmitter schedules the frames earlier or we must push back the gate-open operation in the GCL configuration by the size of queuing window. To show a proof-of-concept, we change the transmitter timing in the following evaluations. Figure 8.12 shows this mechanism on simulated ingress jitter on minimum-size frames. Our evaluation shows that this method can reduce the jitter significantly: Any jitter size can be reduced to a constant switch-specific value. Figure 8.12a shows the effect of QJC for 500 ns of simulated ingress jitter. Here, we see the maximum effectiveness in jitter reduction at a window size of at least 750 ns, the RSPE35 reduces the frame jitter down to around 200 ns. The second example in Figure 8.12b shows that even a jitter as high es 1,000 ns can be contained down to the same value. In our measurement, we required a queuing window size of at least 1,000 ns to achieve that. However, we did not compensate the time synchronization or gate-open inaccuracies in both cases. We see that the lower bound of the latency increases first. The switch forwards frames experiencing minimum jitter later in time. With increasing window size, the lower bound increases further until it meets the upper bound. At this point, the queuing window has reached maximum effectiveness. Further increasing the window size only introduces larger queuing delays without a benefit in jitter reduction.



(a) Up to 500 ns of simulated jitter

(b) Up to 1,000 ns of simulated jitter

Figure 8.12: Effect of the queuing window on a the latency and jitter of high-priority traffic with varying ingress jitter. Shown is the 90 % confidence interval of the latency. We did not compensate the time synchronization error.

9 Conclusion and Outlook

In this thesis, we developed a measurement framework that enables research on physical TSN networks. This extends researchers insight into real-time networks from simulators to real world test setups. Via a user-friendly and descriptive configuration method, the framework automatically executes the measurement procedures and aggregates data. To achieve this on multiple hardware platforms, we developed a layer model for light-weight hardware integration, providing generic interfaces to hardware functionality. Furthermore, the framework supports extensions in the form of new hardware-independent measurement procedures, guaranteeing easy maintenance in the future. The implemented framework is able to generate TT traffic stimuli of over 1 Gbit/s, which matches the current state-of-the-art in industrial applications and fulfills our requirements. Furthermore it is capable of lossless capture and pre-processing data in varying output formats. We presented our framework's high-precision measurement capabilities in a variety of measurements, giving new insight into real world real-time networks. For example, we evaluated the effectiveness of the TAS, verifying other researcher's simulations regarding high-priority latency guarantees under any best-effort load. Additionally, we used the developed framework to identify two key problems with physical setups that are often disregarded in network simulation and schedule synthesis: First, we measure the processing jitter that is introduced due to varying processing delays of a switch. Secondly, we showed that the TAS gate event times can be inaccurate and themselves be introducing considerable jitter. These inaccuracies and imprecisions must be considered in the synthesis of traffic schedules. Disregarding jitter in networks of greater size or lower quality devices could result in a significant loss in the available bandwidth or even violation of guarantees. We presented a queuing maneuver (QJC) with which high jitter can be reduced, increasing overall network bandwidth at the cost of higher latency for the high-jitter stream.

The greatest limitation to our measurement setup lies in the inability to measure the time synchronization quality between the network devices. This is only due to hardware constraints; because of our software architecture, a seamless and light-weight integration is already prepared, only missing a hardware abstraction layer. Therefore, a crucial addition to the framework's capabilities would be the integration of such hardware. Furthermore, hardware support of IEEE 802.1Qbu Frame Preemption can be investigated and integrated if available. Moreover, the framework's flexibility could be used to extend configuration tasks to the whole network under test. For example, the existing hardware configuration layers allow easy integration of the configuration tasks in network switches (e. g., the GCL). With this, the framework could unify the entire network configuration and measurement task into one application and configuration file, completely removing the risk of flawed measurements due to configuration inconsistencies.

Acknowledgments

I wish to thank *Professor Kurt Rothermel* for his inspiring lecture *Rechnernetze* which initially piqued my interest in the field of study. In addition, Professor Rothermel welcomed me into his institute and enabled a productive research environment for this thesis.
I would like to express my gratitude to my research supervisor, *David Hellmanns*, for his continued support, wise input and helpful feedback throughout the course of this thesis.
I would like to thank all members of the institute, in particular *Frank Dürr* for his valuable insight in our meetings.
I thank my friends and family for their input and moral support.

Bibliography

[1] Cisco Systems Inc. “Cisco MGE SFP Modules Cisco Small Business Network Accessories Data Sheet”. In: (Oct. 2018), pp. 1–5. URL: https://www.cisco.com/c/en/us/products/collateral/interfaces-modules/small-business-network-accessories/datasheet-c78-741408.pdf (cit. on p. 16).

[2] Cisco Systems Inc. “Cut-Through and Store-and-Forward Ethernet Switching for Low-Latency Environments”. In: (Apr. 2008), pp. 1–13 (cit. on pp. 14, 15).

[3] S. S. Craciunas, R. S. Oliver, M. Chmelík, W. Steiner. “Scheduling Real-Time Communication in IEEE 802.1Qbv Time Sensitive Networks”. In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. RTNS ’16. Brest, France: Association for Computing Machinery, 2016, 183–192. ISBN: 9781450347877. DOI: 10.1145/2997465.2997470. URL: https://doi.org/10.1145/2997465.2997470 (cit. on pp. 10, 18).

[4] F. Dürr, N. G. Nayak. “No-Wait Packet Scheduling for IEEE Time-Sensitive Networks (TSN)”. In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. RTNS ’16. Brest, France: Association for Computing Machinery, 2016, 203–212. ISBN: 9781450347877. DOI: 10.1145/2997465.2997494. URL: https://doi.org/10.1145/2997465.2997494 (cit. on p. 24).

[5] End Run Technologies. *White Paper: Precision Time Protocol (PTP/IEEE-1588)*. URL: https://endruntechnologies.com/pdf/PTP-1588.pdf (cit. on p. 20).

[6] J. Falk, D. Hellmanns, B. Carabelli, N. Nayak, F. Dürr, S. Kehrer, K. Rothermel. “NeSTiNg: Simulating IEEE Time-sensitive Networking (TSN) in OMNeT++”. In: *2019 International Conference on Networked Systems (NetSys)*. 2019, pp. 1–8. DOI: 10.1109/NetSys.2019.8854500 (cit. on pp. 9, 23).

[7] N. Finn. “Introduction to Time-Sensitive Networking”. In: *IEEE Communications Standards Magazine* 2.2 (2018), pp. 22–28 (cit. on pp. 9, 18, 20, 23).

[8] P. Gaj, J. Jasperneite, M. Felser. “Computer Communication Within Industrial Distributed Environment—a Survey”. In: *IEEE Transactions on Industrial Informatics* 9.1 (2013), pp. 182–189 (cit. on p. 9).

[9] Garland Technology. *Garland Technology Website: Breakout Network TAPs*. URL: https://www.garlandtechnology.com/breakout-passive-fiber-copper-network-tap (cit. on p. 17).

[10] Garland Technology. *Garland Technology Website: Network Test Access Point (TAP)*. URL: https://www.garlandtechnology.com/network-tap (cit. on p. 17).

[11] V. Gavriluţ, P. Pop. “Scheduling in time sensitive networks (TSN) for mixed-criticality industrial applications”. In: *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*. 2018, pp. 1–4. DOI: 10.1109/WFCS.2018.8402374 (cit. on pp. 9, 24, 65).

[12] D. Hellmanns, A. Glavackij, J. Falk, R. Hummen, S. Kehrer, F. Dürr. "Scaling TSN Scheduling for Factory Automation Networks". In: Mar. 2020. DOI: 10.1109/WFCS47810.2020.9114415 (cit. on pp. 13, 19, 24).

[13] Hirschmann Automation and Control GmbH. *User Manual Rail Switch Power Enhanced HiOS-2A Rel. 08600*. URL: https://www.doc.hirschmann.com/pdf/ManualCollection_RSPE_HiOS-2A-08600_en.pdf (cit. on pp. 67, 68).

[14] "IEEE Standard for Ethernet". In: *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)* (2018), pp. 1–5600 (cit. on p. 11).

[15] "IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks". In: *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014)* (2018), pp. 1–1993 (cit. on pp. 12, 14, 18, 31).

[16] "IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks – Amendment 26: Frame Preemption". In: *IEEE Std 802.1Qbu-2016 (Amendment to IEEE Std 802.1Q-2014)* (2016), pp. 1–52 (cit. on pp. 20, 21).

[17] "IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic". In: *IEEE Std 802.1Qbv-2015 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q-2014/Cor 1-2015)* (2016), pp. 1–57 (cit. on pp. 9, 18).

[18] "IEEE Standard for Local and metropolitan area networks– Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams". In: *IEEE Std 802.1Qav-2009 (Amendment to IEEE Std 802.1Q-2005)* (2010), pp. 1–72. DOI: 10.1109/IEEESTD.2010.8684664 (cit. on p. 18).

[19] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems". In: *IEEE Std 1588-2019 (Revision of IEEE Std 1588-2008)* (2020), pp. 1–499 (cit. on pp. 20, 60).

[20] J. Jasperneite, J. Imtiaz, M. Schumacher, K. Weber. "A Proposal for a Generic Real-Time Ethernet System". In: *IEEE Transactions on Industrial Informatics* 5.2 (2009), pp. 75–85 (cit. on p. 31).

[21] J. Jiang, Y. Li, S. H. Hong, A. Xu, K. Wang. "A Time-sensitive Networking (TSN) Simulation Model Based on OMNET++". In: *2018 IEEE International Conference on Mechatronics and Automation (ICMA)*. 2018, pp. 643–648. DOI: 10.1109/ICMA.2018.8484302 (cit. on pp. 9, 10, 23, 24, 68).

[22] J. Jiang, Y. Li, S. H. Hong, M. Yu, A. Xu, M. Wei. "A Simulation Model for Time-sensitive Networking (TSN) with Experimental Validation". In: *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2019, pp. 153–160. DOI: 10.1109/ETFA.2019.8869206 (cit. on p. 24).

[23] L. Lo Bello, W. Steiner. "A Perspective on IEEE Time-Sensitive Networking for Industrial Communication and Automation Systems". In: *Proceedings of the IEEE* 107.6 (2019), pp. 1094–1120 (cit. on pp. 9, 18, 19, 23, 31).

[24] J. L. Messenger. "Time-Sensitive Networking: An Introduction". In: *IEEE Communications Standards Magazine* 2.2 (2018), pp. 29–33. DOI: 10.1109/MCOMSTD.2018.1700047 (cit. on pp. 19, 23).

[25] Napatech A/S. *Reference Documentation Capture 12.5*. URL: https://docs.napatech.com/search/all?query=Reference+Documentation (cit. on pp. 15, 16, 20, 49, 59).

[26] S. Nsaibi, L. Leurs, H. D. Schotten. "Formal and simulation-based timing analysis of Industrial-Ethernet sercos III over TSN". In: *2017 IEEE/ACM 21st International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. 2017, pp. 1–8. DOI: 10.1109/DISTRA.2017.8167670 (cit. on pp. 9, 19, 23).

[27] PC Magazine Encyclopedia. *SFP Definition*. Mar. 6, 2021. URL: https://www.pcmag.com/encyclopedia/term/sfp (cit. on p. 16).

[28] M. Pahlevan, R. Obermaisser. "Evaluation of Time-Triggered Traffic in Time-Sensitive Networks Using the OPNET Simulation Framework". In: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 2018, pp. 283–287. DOI: 10.1109/PDP2018.2018.00048 (cit. on pp. 9, 10, 19, 23, 68, 69).

[29] P. Pop, M. L. Raagaard, S. S. Craciunas, W. Steiner. "Design optimisation of cyber-physical distributed systems using IEEE time-sensitive networks". In: *IET Cyber-Physical Systems: Theory Applications* 1.1 (2016), pp. 86–94 (cit. on p. 18).

[30] P. Pop, M. L. Raagaard, M. Gutierrez, W. Steiner. "Enabling Fog Computing for Industrial Automation Through Time-Sensitive Networking (TSN)". In: *IEEE Communications Standards Magazine* 2.2 (2018), pp. 55–61 (cit. on pp. 9, 23).

[31] Profitap HQ B.V. *ProfiShark 1G+ Datasheet*. 2018. URL: https://www.profitap.com/wp-content/uploads/ProfiShark-1G-Plus-Datasheet.pdf (cit. on p. 24).

[32] RP Photonics Encyclopedia. *Beam Splitters*. URL: https://www.rp-photonics.com/beam_splitters.html (cit. on p. 17).

[33] S. Samii, H. Zinner. "Level 5 by Layer 2: Time-Sensitive Networking for Autonomous Vehicles". In: *IEEE Communications Standards Magazine* 2.2 (2018), pp. 62–68. DOI: 10.1109/MCOMSTD.2018.1700079 (cit. on p. 23).

[34] S. Schriegel, A. Biendarra, O. Ronen, H. Flatt, G. Leßmann, J. Jasperneite. "Automatic determination of synchronization path quality using PTP bridges with integrated inaccuracy estimation for system configuration and monitoring". In: *2015 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS)*. 2015, pp. 52–57. DOI: 10.1109/ISPCS.2015.7324683 (cit. on p. 20).

[35] C. Simon, M. Maliosz, M. Mate. "Design Aspects of Low-Latency Services with Time-Sensitive Networking". In: *IEEE Communications Standards Magazine* 2.2 (2018), pp. 48–54. DOI: 10.1109/MCOMSTD.2018.1700081 (cit. on pp. 9, 10, 21, 23).

[36] The Tcpdump Group. *Official web site of tcpdump and libpcap*. URL: https://www.tcpdump.org (cit. on p. 56).

All links were last followed on April 6, 2021.

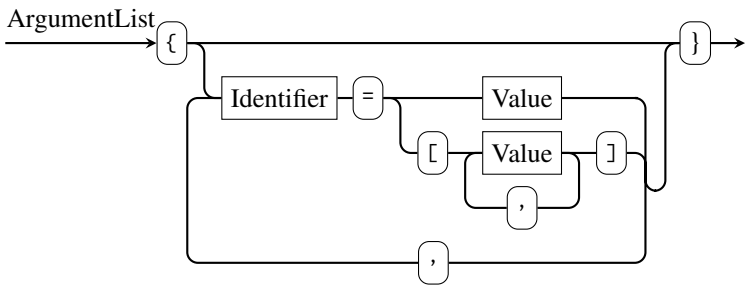
A Configuration Syntax for the Framework

Our framework was developed with the goal of a high level of flexibility and ease-of-use. Therefore, the framework uses a configuration file with custom format to integrate the different flexibilities in a simple way. In the following we state the syntax of this configuration file, highlighting key aspects of the format. We use extended Backus-Naur form (EBNF) rules to define the syntax of the file. The basics for these definitions are the rules for digits, characters, identifiers and values. Note that all identifiers and values will have their leading and trailing white space removed.

```
Digit      ::= '0'|...|'9';
Char       ::= 'a'|...|'z'|'A'|...|'Z';
Identifier ::= { Digit|Char|'_'|'-'|'␣' }+[1];
Value      ::= { Digit|Char|'_'|'-'|'␣'|':'|'.' }+;
```

For all other rules, we use syntax diagrams, where rectangles represent variables and boxes with rounded corners represent terminals.

The configuration file uses *Argument Lists* containing key-value pairs in multiple occasions. We use these to pass parameters to all configurable parts of the framework. The key-value pairs are grouped with curly braces and separated with commas. Using square brackets, values can be of a list type.



A configuration consists of four sections, prefaced by their section identifiers `[general]`, `[stations]`, `[links]` and `[streams]`. We detail these sections in the following paragraphs.

General Parameters All parameters affecting the entire setup of a measurement are declared in the general section. For example, the number of frames to send as part of any stream is specified in this section of the configuration file. Thus, the general section comprises an argument list, holding all these parameters. Table A.1 lists possible keys for the general section.

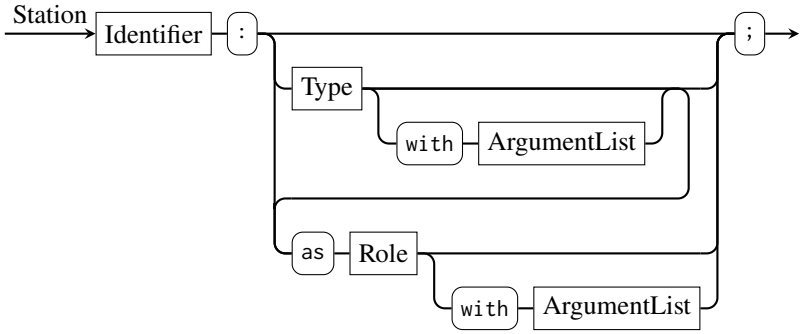
[1] Note that we use "`{}+`" as "at least once".

Table A.1: Possible key value pairs for the general parameter section.

Key	Description	Value type
`packets`	number of packets to be sent by every stream	Integer
`database`	path to the output database	String, no quotes needed
`hypercycle length`	The cycle time, to align the transmission to	Integer

Definition of a Network Station With the common parameters defined, we start the topology definition. The first part of this is the definition of the network stations. Every station defined in the `[stations]` section represents a node in the topology graph. According to our architecture, every station has a *type* and a *role*, which the user can specify in the configuration using an identifier. The type identifier resolves to a setup hook, initializing a hardware abstraction module. Thus. the framework uses types to differentiate between specific end station implementations. The default type is `none`, meaning that the framework does not control this station and it does not select a hardware abstraction module. The available roles of a station depend on the type of the station. For example, the `none` type can have the role `tap`, which enables monitoring connections over this station. We will come into more detail on this, later on. The default role is, once again, `none`. For the configuration, it is only important to remember that active components under framework control have a type other than `none`.

Both type and role can be parametrized with arguments, for example any type can receive the argument `mac`, which then manually sets this station's MAC address. This could, for example, be useful if a receiver is not under the framework's control but the framework sends messages to this station. The general syntax of one station definition is as follows.



Both type and role are identifiers, but only specific values are interpreted. For example, the measurement framework defines the type *NT* to start up a type instance of the Napatech hardware abstraction module. This NT type can take the roles `capture`, `transmit`, a combination of the two or `none`. The `capture` role is used to capture all incoming traffic on the adapter's port, whereas `transmit` enables transmitting end point emulation. The NT type requires the arguments `adapter` and `port`, which correspond to the adapter and port number, according to the Napatech driver configuration. Furthermore, the NT type automatically sets the MAC address of the corresponding station to that port's true hardware address. The Tables A.2 and A.3 give a list of the implemented type and role arguments.

As an example, a station definition for a NT implementation capturing on port 5 of adapter 1, can be as follows.

```
example: NT with {adapter=1, port=5} as capture with {to=tablename};
```

In the case of our database output format, the captured data would be written to a table called "tablename" in the database specified in the general section.

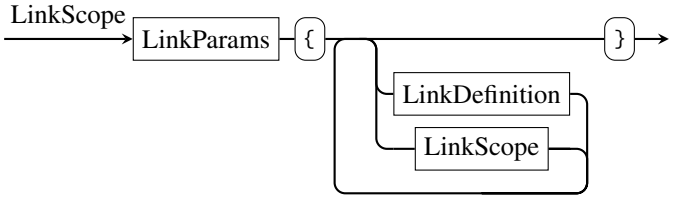
Table A.2: Possible key value pairs for the type arguments of a station.

Key	Description	Value type
`mac`	mac address of this station	Six values of `0x00` to `0xFF` separated by a colon
`adapter`	adapter number	Integer
`port`	port number on the adapter	Integer

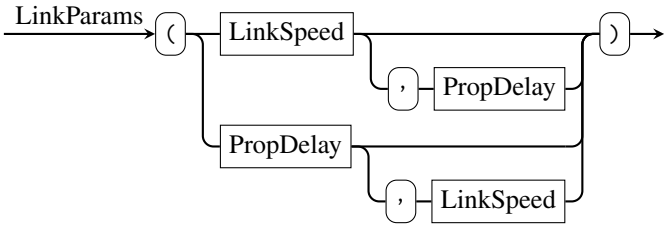
Table A.3: Possible key value pairs for the role arguments of a station.

Key	Description	Value type
`to`	output specification, in the case of a database, this is the table name	String, no quotes needed

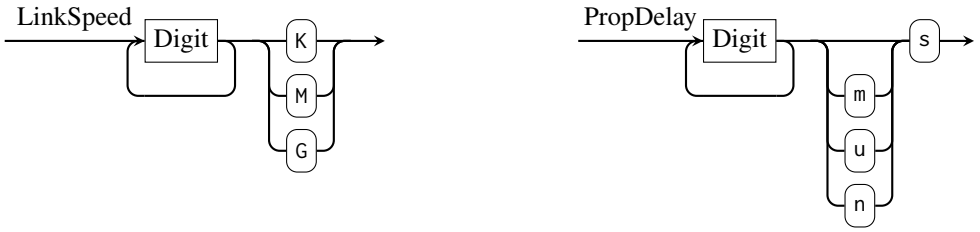
Link Definitions Links represent either a simplex or a full-duplex connection between two previously defined stations. All link definitions have to be placed in the `[links]` section of the configuration file. To minimize the amount of text needed for the link definitions, we group links according to their link parameters into recursively nesting scopes.



So far, the framework requires the two parameters link speed and propagation delay for each link. A parameter definition `LinkParams` can either hold both values in any order, or only one of the two. The latter is for example useful, if the user wants to define many links with different propagation delays but the same link speed. The framework allows recursive scope nesting and applies the inner-most parameter set to each link definition. We hence define the link parameter rule as follows.

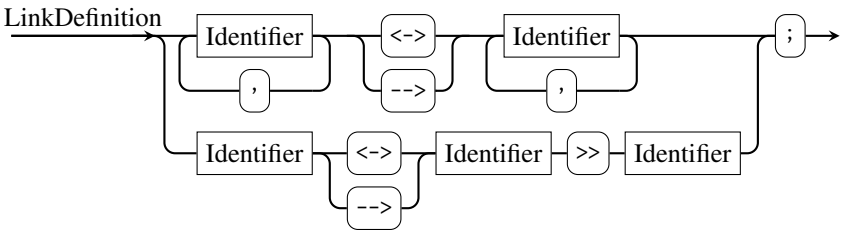


The definitions for propagation delay and link speed are as follows. Here, the link speed is given in either kbit/s, Mbit/s or Gbit/s. Similarly, the propagation delay is defined in seconds, with the optional SI prefixes for milli-, micro- or nanoseconds.



For example, the link parameters `(100M, 50ns)` would result in a scope of links with 100 Mbit/s link speed and 50 ns propagation delay.

The rule for a link definition looks as follows.

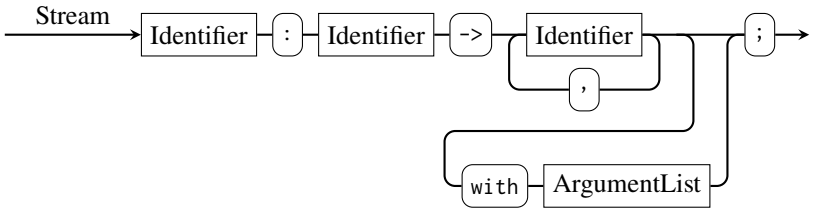


The link definitions themselves always consist of at least two stations with a link type identifier (i. e., simplex `-->` or full-duplex `<->` connection) between them. It is also possible, to provide a list of stations on either side of the link type identifier. Given those lists, the framework adds a link for each pair in the Cartesian product of the left-hand side and right-hand side lists to the topology. For example, given four stations `s1`, `s2`, `r1`, `r2`, the definition scope

```
1 (100M, 50ns) {
2   s1, s2 --> r1, r2;
3 }
```

would create four simplex links (`s1`, `r1`), (`s1`, `r2`), (`s2`, `r1`), (`s2`, `r2`), all at the same link speed of 100 Mbit/s with 50 ns propagation delay. Another feature of the link definitions is the monitoring identifier ">>". It is used to add a monitoring connection to the defined link. If using the monitoring identifier, only one station for sender and receiver is allowed. Furthermore, one of the stations must be of role `tap`. The link definition `s <-> t >> m;` creates a full-duplex link between `s` and `t` and adds a monitoring connection to `m` that receives a copy of the traffic from `s` to `t`.

Definition of a Stream A stream defines the transmit schedule of a station in the topology. Each stream has one sender and at least one receiver, with an optional argument list to pass parameters to the stream.



All sender or receiver identifiers have to be previously defined stations. Table A.4 lists the possible argument keys and descriptions.

As an example, the following line creates a stream with sender `s` and the two receivers `r1` and `r2`. The transmit slot is 500 ns long and is rescheduled every 1,000 ns.

```
1 exampleStream: s -> r1, r2 with {cycle=1000, offset=0, length=500};
```

Table A.4: Possible key value pairs for the arguments of a stream.

Key	Description	Value type
`length`	the length of the transmit slot in nanoseconds	Integer
`offset`	the offset of the transmit slot scheduling in nanoseconds	Integer
`cycle`	the cycle length of the transmit slot scheduling in nanoseconds	Integer
`packed TX`	enables back-to-back transmission	A boolean-equivalent
`payload sizes`	Specifies the values for random payload generation.	List of three integers: `[min, max, step]`. Payload sizes between 0 and 1,448 B are possible
`Q tagged`	Activates Q-tagging by setting the TPID	A boolean-equivalent
`vid` or `vlan`	VID of the frame, only has an effect when Q-tagged	Integer value between 1 and 4094
`pcp` or `prio`	priority of the frame, only has an effect when Q-tagged	Integer value between 0 and 7
`stepCorrVal`	correction offset that is added to the next slot time after `stepCorrNum` scheduled slots (cyclic)	Integer
`stepCorrNum`	number of slots after which the correction value is applied to the slot start time	Integer

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature