TU WIEN Informatics

NETCONF for TSN End Stations

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Engineering

by

Nora Hartner
Registration Number 01327206

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Assistance: Dipl.-Ing.(FH) Dieter Etz, MBA
Projektass. Dipl.-Ing. Dr.techn. Thomas Frühwirth, BSc

Vienna, March 6, 2024

Nora Hartner	Wolfgang Kastner

Erklärung zur Verfassung der Arbeit

Nora Hartner

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 6. März 2024

Nora Hartner

Acknowledgements

A huge thank you to Thomas Frühwirth and Dieter Etz for their continuous support throughout writing this thesis. Thank you for all of your time and the amazing guidance. I would also like to thank Johanna Hartner for proofreading the whole thing. You are so much better than my spellchecker. And one last thank you goes to Aaron Duxler, for being a great rubber duck.

Abstract

The requirements for flexibility in industrial networks are higher today than ever before. This leads to a shift in how the technologies already in existence need to come together. While the network management protocol NETCONF has been around for a while, the idea of using it in combination with Time-Sensitive Networking (TSN) has brought new attention to it. And although various network devices often support NETCONF, end devices rarely do. This thesis explores an idea on how to bridge this gap between the NETCONF network and an end device. In order to do so, a daemon is implemented, observing the configuration data of a NETCONF server and passing any incoming changes along to an end device. It is shown that the concept has potential, however, future work is needed to ensure that the high safety requirements of industrial automation systems are met.

Contents

CHAPTER 1

Introduction

Industrial automation has seen multiple revolutions within the past 300 years. The first industrial revolution introduced steam- and water power and made the switch from manual labor to machines possible [XDK+18]. The second industrial revolution was marked by the widespread availability of electrical power and introduced production lines. Finally, during the third industrial revolution we switched from analogue electronic technologies to digital electronics. Today we are right in the middle of the so-called fourth industrial revolution, which focuses on interconnectivity between machines, devices, sensors and humans. In order to achieve this interconnectivity, open standards that allow for the interoperability of machines from different vendors are needed. Being able to dynamically configure all machines and devices involved in production is now becoming a necessity to ensure smooth, efficient and safe operations, even with frequent changes within the industrial environment.

Industry 4.0 generates new requirements for industrial automation systems. The integration of mobile devices and a more flexible reconfiguration of participating devices during continuous operation becomes a necessity [SPFA19]. Industrial networks are more dynamic and have higher flexibility requirements. One approach to solve these new demands is Time-Sensitive Networking (TSN). TSN extends current IEEE 802 standards like 802.11 wireless local area network or 802.3 wired Ethernet with the capabilities to provide real-time functionality. For industrial automation, a more specific standard has been created. TSN for industrial automation, as defined in IEC/IEEE 60802, bundles some of the TSN standards. By defining a standardized network infrastructure, TSN for industrial automation bridges the gap between operational technologies (OT) and information technology (IT).

TSN acts as an Industry 4.0 enabler by defining ways of communication between all participants, whether they work with time-sensitive information or are satisfied with best-effort networking. However, before communication can happen at all, another issue needs to be addressed. With Industry 4.0, the new challenge of adding and removing

participants during continuous operations arises. To provide this flexibility that was not required in earlier systems, some form of network management to configure new devices in order to properly fit into the existing setup is required. While the network management protocol NETCONF exists since 2006, the idea of using it in combination with TSN has brought new attention to it in the past few years [EBBS11]. Various network participants like switches with NETCONF support have been available for a while now, but with the newly developed need to reconfigure even end stations during continuous operations, it becomes more attractive to use NETCONF for those as well. In combination with the data modeling language Yet Another Next Generation (YANG), the NETCONF protocol can be used to configure a wide range of devices within a network [Bjö10]. Vendors are able to create YANG models for their devices, creating a well-defined interface to configure those devices using NETCONF.

1.1 Motivation

While modern switches mostly support NETCONF, the majority of end stations such as industrial robots or machines do not. In order to establish communication and receive configuration from a NETCONF client, a device needs a NETCONF server. Switches do have this server already built-in, making configuration easy. End stations on the other hand do not possess these built-in servers. To be able to still make them configurable by the client, a stand-alone server can be added on the end station, that enables configuration of its Network Interface Card (NIC) via the NETCONF client. Figure 1.1 illustrates the communication between a client and various network participants. A Network Management System (NMS) with a NETCONF client is connected to a TSN switch. The switch itself can be configured via NETCONF. Additionally, it acts as a bridge to the other network participants, in this example a robot and two machines. Those are the end stations that typically can not be configure directly via NETCONF, but need an extra NETCONF server.

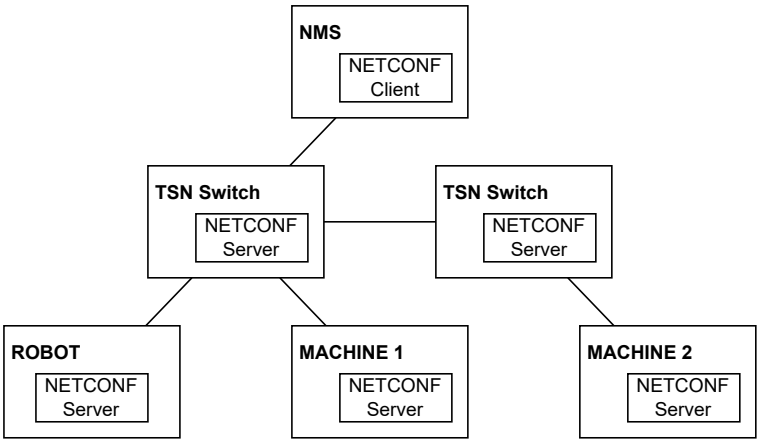

Figure 1.1: TSN Device Configuration with NETCONF

This thesis focuses on the first steps of making industrial systems more flexible, the dynamic discovery and subsequent configuration of devices. This is done by creating a server that is able to detect when devices become available. Newly found devices will be sent a configuration that enables them to participate and cooperate with the already established network of devices, machines and humans.

1.2 Structure of the Work

The structure of this thesis is as follows:

- **Chapter 2 - State of the Art:** This chapter serves as an introduction for the reader to the technologies used in the further thesis.
- **Chapter 3 - Design and Limitations:** This chapter gives an overview on available tools and describes design choices and which limitations were set.
- **Chapter 4 - Implementation:** This chapter describes the individual components of the implementations, gives further insights into their workings, and explains how to put all components together into a working solution.
- **Chapter 5 - Evaluation:** This chapter documents the results of testing the implementation against different scenarios.
- **Chapter 6 - Conclusion:** This final chapter gives a final overview of the project and discusses what could be further improved in future versions.

CHAPTER 2

State of the Art

2.1 NETCONF - Network Configuration Protocol

The NETCONF protocol was developed by the Internet Engineering Taskforce (IETF) as a network management protocol to install, manipulate and delete network devices [EBBS11]. Configuration data and protocol messages are formatted using Extensible Markup Language (XML). The operations are realized as remote procedure calls (RPCs). A strength of NETCONF lies in its ability to mimic the native functionality of the devices implementing it, thus reducing the time and cost needed to implement new features. NETCONF uses sessions between clients and servers. The basic communication schema between server and client is depicted in Figure 2.1. At the start of the session, client and server exchange a `<hello>` element, containing a list of their capabilities. Once the connection is established, the client is able to send various operations reading and manipulating data, to which the server reacts by sending a `<rpc-reply>`. The exchange is terminated with a `<close-session>` by the client, followed by an `<ok>` from the server.

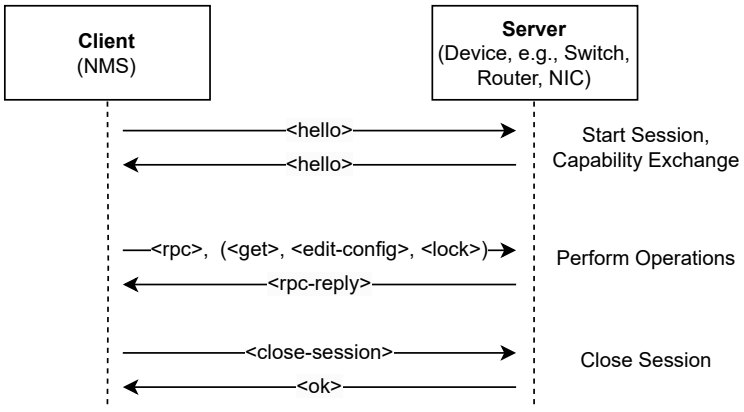

Figure 2.1: Basic NETCONF Communication

Clients are part of the NMS, servers are network devices. Servers need to support at least one NETCONF session, but often support more. NETCONF supports so-called capabilities, that allow for the basic NETCONF functionalities to be extended. Capabilities define, for example, additional operations. Clients are to discover which capabilities are used by a server and are able to use them subsequently.

2.1.1 Protocol Layers

NETCONF is separated into four layers, as can be seen in Figure 2.2 [EBBS11]. The **content layer** includes configuration data and notification data. A common language used to model this layer is YANG (see Section 2.1.3) The **operations layer** defines a set of XML-encoded operations shown in Table 2.1, that is used to manage and retrieve device configurations. The **messages layer** provides a mechanism to encode remote procedure calls, and finally the **secure transport layer** provides the requirements to send secure messages between server and client devices. Further specific requirements for each layer can be found in [EBBS11].

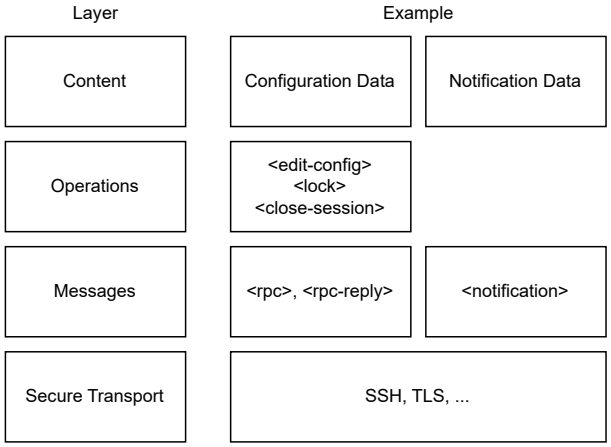

Figure 2.2: NETCONF Protocol Layers, Adapted from [EBBS11]

<get>	retrieves configuration and device state information
<get-config>	retrieves only the configuration from the datastore
<edit-config>	loads the given config to a given datastore
<copy-config>	copies the config from one datastore to another
<delete-config>	deletes a configuration datastore
<lock>	locks an entire datastore
<unlock>	releases the lock from the <lock> operation
<close-session>	gracefully terminates the NETCONF session
<kill-session>	forcefully terminates the NETCONF session

Table 2.1: NETCONF Operations

2.1.2 Datastores

Datastores are where the configuration specified by the YANG data models are stored. In datastores the configuration can be implemented using flash memories, databases, files or combinations of those [BSS+18]. Datastores are used to bind network management data models to network management protocols. There are two architectural frameworks. An original simple model (Figure 2.3) and an updated version (Figure 2.4), in which problems with the simple model were addressed. The original model consists of a candidate, a startup and a running configuration store. The extended version adds an intended configuration datastore and replaces the operational state of the simple version with an operational state datastore. The original model is currently used by NETCONF [EBBS11]. Different implementations may not require all datastores included in either model and may omit one or more of them, based on their needs. The only datastore that is always required, is the running configuration datastore. The following sections give an overview of the different datastores available.

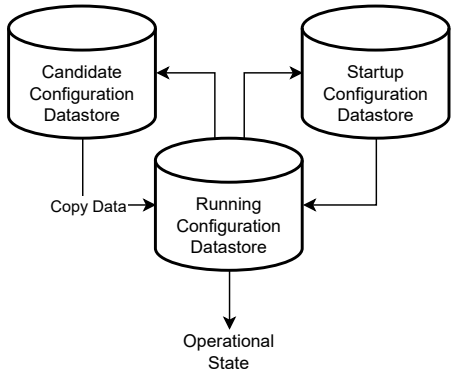

Figure 2.3: Datastores, Simple Version, Adapted from [BSS+18]

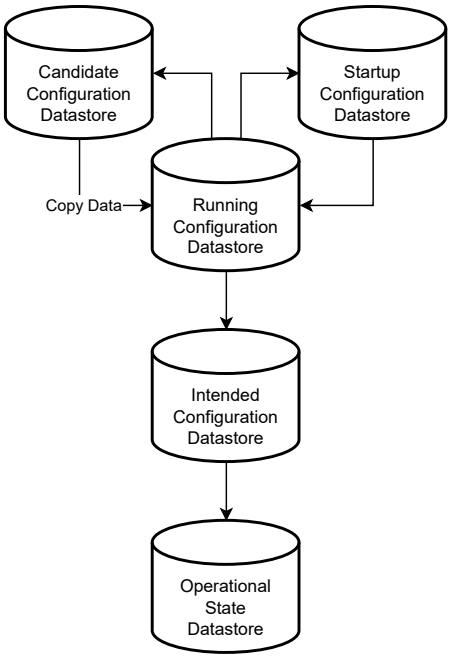

Figure 2.4: Datastores, Extended Version, Adapted from [BSS+18]

Running Configuration Datastore

The running datastore is where the current operational configuration is stored. It is required that the configuration in running is valid at all times. The running datastore may not be omitted. If the startup datastore is not in use, the running datastore is typically implemented using non-volatile storage, allowing it to persist across reboots.

Candidate Configuration Datastore

The candidate datastore is where configuration may be manipulated before copying it to the running datastore. Candidate does not have to exist in every configuration. Usually, it is implemented using volatile storage, meaning it is not saved across reboots, but set to a copy of running at the time of startup.

Startup Configuration Datastore

The startup datastore holds the configuration that is loaded by a device during booting. With NETCONF, the only way to change the configuration stored in the startup datastore is to copy the running datastore to startup. Other protocols, like RESTCONF, do not set this limitation. During the boot process, the configuration saved in startup will be copied to startup. Due to its nature, the startup datastore is usually stored in non-volatile storage. Like candidate, the startup datastore does not need to be included in every implementation.

Intended Configuration Datastore

The intended datastore is a read-only configuration datastore that holds the configuration of running after all transformations, like template extensions or removal of inactive configuration, are applied. Whenever configuration is written to the running datastore, intended must be updated and validated immediately after. Like running, the intended datastore must always consist of valid configuration. The contents of the intended datastore do not have to persist across reboots, as it can be taken from running after a reboot.

Operational State Datastore

The operational state datastore contains the whole system state. Additionally to the configuration data specified in the intended datastore, it also includes all the „config false“ nodes. It is a read-only datastore. The original model does not implement the operational state as a datastore, containing „config false“ nodes.

2.1.3 YANG

Yet Another Next Generation (YANG) is a data modeling language developed and maintained by the NETMOD working group of the Internet Engineering Task Force (IETF). While originally developed to be used to model data for NETCONF, it has also been used with other protocols like RESTCONF or the Constrained Application Protocol (CoAP) [Bjö16]. Using YANG, it is possible to further specify the layers 1 and 2 of NETCONF. YANG defines its data in modules. Each module consists of a header and a number of thematically related nodes. The header provides the YANG version used, a description, includes of other modules and revisions. Data is modeled as a tree with unique nodes that contain either values or sub-nodes. There are mainly four different types of nodes: leaf nodes, leaf-list nodes, container nodes and list nodes [Bjö10]. A brief explanation of each of those nodes can be found alongside the following Listings.

Leaf Nodes

Leaf nodes contain a singular value and no further child nodes. Listing 2.1 depicts an exemplary YANG model of a leaf node defining a IP address for a device using the string type. Listing 2.2 the corresponding NETCONF entry with an IP address.

```
leaf address {
   type string;
   description "IP address for this device";
}
```

Listing 2.1: Leaf Node YANG

```
<address>192.168.0.1</address>
```

Listing 2.2: Leaf Node NETCONF

Leaf-List Nodes

Leaf-list nodes are a sequence of leaf nodes, containing a specific value type. The elements in a leaf-list node have to be unique. Listing 2.3 depicts an exemplary YANG model of a leaf-list node, defining the user leaf-list as a string type. Listing 2.4 shows the corresponding NETCONF entry, in which multiple users exist, all defined by using the user tag.

```
leaf-list user {
   type string;
   description "user accounts";
}
```

Listing 2.3: Leaf List Node YANG

```
<user>Alice</user>
<user>Bob</user>
<user>James</user>
```

Listing 2.4: Leaf List Node NETCONF

Container Nodes

Container nodes group related nodes together in one sub-tree. They contain only child nodes and no values. Listing 2.5 depicts an exemplary YANG model of a container node, in which a container containing a leaf-list is nested into another container. Listing 2.6 depicts the corresponding NETCONF entry.

```
container system {
   container virtual-interfaces {
      leaf-list vlan {
         type int;
         description "vlan tag";
      }
   }
}
```

Listing 2.5: Container Node YANG

```
<system>
   <virtual-interfaces>
      <vlan>7</vlan>
      <vlan>42</vlan>
   </virtual-interfaces>
</system>
```

Listing 2.6: Container Node NETCONF

List Nodes

List nodes specify list entries. They contain a unique key and sub-nodes of any type, but no values. Listing 2.7 depicts an exemplary YANG model of a list node. An interface is uniquely defined by the name leaf and additionally holds information about the interfaces address and subnet-mask. Listing 2.8 depicts the corresponding NETCONF entry in which such an interface is defined.

```
list interface {
   key "name";
   leaf name {
      type string;
      description "Interface name"
   }
   leaf address {
      type string;
      description "Interface IP address"
   }
   leaf subnet-mask{
      type string;
      description "Interface subnet mask"
   }
}
```

Listing 2.7: List Node YANG

```
<interface>
   <name>GigabitEthernet 0/0/1</name>
   <address>192.168.1.1</address>
   <subnet-mask>255.255.255.0</subnet-mask>
</interface>
```

Listing 2.8: List Node NETCONF

YANG already provides a number of built-in types, but it is also possible to define additional types that can restrict input to more specific rules. YANG can differentiate between configuration and state data. Configuration data include values that can be read and written (e.g., the configured IP-address for a device), while state data is read-only data (e.g., which ports of a switch are currently up). For each node, a config parameter can be set, defining whether it contains configuration data („config true“) or state data („config false“). This goes hand in hand with NETCONF's differentiation of <get> and <get-config> operations. Using <get-config> will only return nodes with the config parameter set to true, while <get> will return all nodes. YANG is also able to define remote procedure calls and notifications. Together with the configuration and state data, this allows for all traffic between NETCONF servers and clients to be described by YANG.

2.2 TSN - Time-Sensitive Networking

For a long time, industries like automobile control, audio and video production or industrial control all used special-purpose systems for their data traffic. At that point Ethernet was neither reliable nor cheap enough to be used in those industries. With the huge demand that Ethernet is in nowadays, Ethernet would be significantly cheaper, which encourages the switch to it [Fin18]. Ethernet, as it is, is not real-time capable, due to its nondeterministic behavior. To improve the reliability of Ethernet, the IEEE 802.1 Working Group has released a set of standards called TSN. These standards aim to provide the features that Ethernet is currently missing, so that it can be used for time-sensitive networks. Those requirements are, among others, time synchronization, guaranteed packet transport or bounded latency. The following sections will discuss these requirements in more detail.

2.2.1 Time Synchronization

An important requirement of real-time applications is time synchronization. Devices within a network need to have the same understanding of time, with fixed precision. The length of this precision depends on the area of use, but typical duration ranges from 10ns to 1μs [Fin18]. A fixed precision is necessary to ensure that packages will be delivered in time. Typically, the Precision Time Protocol defined in IEEE 1588 is used to synchronize the clocks of all network devices, but other algorithms may be used as well if desired. Building on IEEE 1588, additional specifications to further regulate the time-synchronization requirements needed for time-sensitive applications have been defined in IEEE 802.1AS-2020 [91220].

2.2.2 Scheduling and Traffic Shaping

To be able to give guarantees that important packages will arrive at their destination in time, scheduling and traffic shaping is necessary. One standard describing such measures is IEEE 802.1Qbv. A simple visualization of the explanation below can be found in Figure 2.5 to aid with understanding.

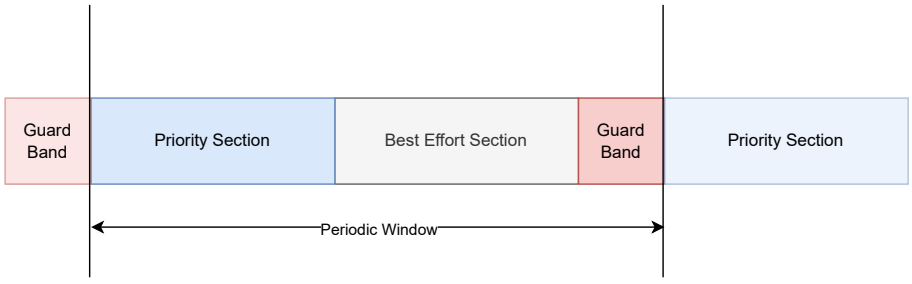

Figure 2.5: Simple Visualization of Sections with Guard-Band IEEE 802.1Qbv

First, access to the communication channel gets divided into multiple time slots. Certain traffic is assigned to each time slot. This separates the traffic into priority sections, reserved for time-sensitive data and best-effort sections, reserved for non-critical data [Mes18]. The purpose of this separation is to ensure that time-critical packets have a guaranteed sending slot where they can't be overwritten by less important packets. Next, there are guard-bands added before the sections of time-sensitive data. If a transmission would start at the very end of a non-time-sensitive section, and if the transmission takes longer than the remainder of that best-effort section, that packet will continue sending during the time-sensitive section, taking time away from more important packets. Adding a guard-band ensures that all packets are finished by the time the section changes, and that no new packets are started when there is not enough time left to finish. To ensure that no packet that started in the best-effort section is still not fully transmitted by the time the priority section starts, the duration of the guard-band needs to be the length of the longest possible packet. This method of course has the drawback of a lot of idle time during the guard-band. To prevent this inefficiency, TSN defines Preemption in the standards IEEE 802.1QBU and IEEE 802.3BR. Preemption allows for the transmission of packets to be suspended and to be picked up at a later time. More precisely, a longer packet that is transmitted at the end of the best-effort section can be interrupted and finished once the next best-effort section comes around, allowing some priority packets to be sent in between. A visualization of this can be found in Figure 2.6. Using preemption allows for the guard-band to be significantly smaller, resulting in a more efficient schedule due to less idle time.

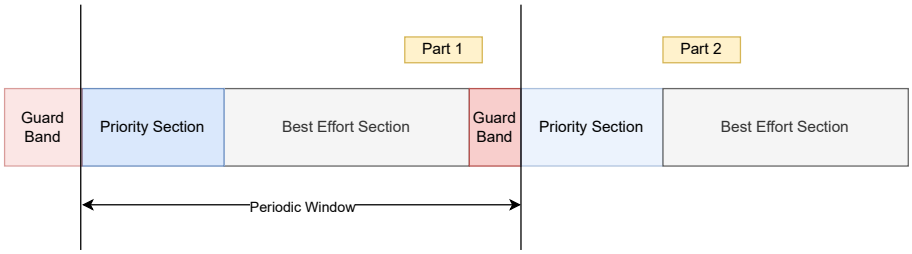

Figure 2.6: Visualization of Preemption

2.2.3 TSN for Industrial Automation

Some of the TSN standards, like IEEE 802.1AS-rev—Timing and Synchronization for Time-Sensitive Applications and IEEE 802.1Q-2018—Bridges and Bridged Networks and more form the basis for TSN for industrial automation, defined in IEC/IEEE 60802. TSN for industrial automation further restricts TSN and creates a guideline for developers, vendors and users of time-sensitive applications within industrial automation networks [LBS19].

CHAPTER 3

Design and Limitations

This thesis asks the question of how to configure end stations in a network using NETCONF, despite the device not understanding the NETCONF protocol. The following chapter starts by taking a look at the various tools and libraries available for NETCONF. It then gives a short overview on how to configure a NIC on Linux systems and finally describes the design of the implementation and explains which limitations were set for this work.

3.1 Tools and Libraries

While there is a wide variety of tools for NETCONF, a lot of them are outdated. The following tables (Table 3.1-3.3) list some of the tools and libraries available. Two comprehensive, although slightly outdated lists of tools can be found on Netconf Central [Neta] and the IETF Community Wiki [IET].

Software	**Version**	**Last updated**	**Interface**	**Availability**
ConfD	8.0.8	2023	command line	free basic version
NETCONFc	-	2023	GUI	commercial
MG-SOFT NETCONF Browser	11	2023	GUI	commercial
enSuite	3	2013	GUI	open source
netopeer2-cli	-	2023	command line	open source
YENCA	-	2014	GUI	open source

Table 3.1: NETCONF Server and Client Tools

Software	Version	Last updated	Availability
netopeer2	2.1.71	2023	open source
libyang	2.1.111	2023	open source
libnetconf2	2.1.37	2023	open source
sysrepo	2.2.105	2023	open source
netconf4j	0.0.9	2015	open source
netconf4android	1.0	2012	open source

Table 3.2: NETCONF Libraries

Software	Version	Last updated	Interface	Availability
pyang	2.5.3	2023	cmd line	open source
yangbuilder	1.3.0	2017	cmd line	open source
yang-explorer	-	2020	GUI	open source
yangsuite	1.17.15	2023	GUI	open source

Table 3.3: YANG Tools

In this thesis, the netopeer2 library ([Netb]) was used. Netopeer2 builds on the libyang ([libb]), sysrepo ([Sysc]) and libnetconf2 ([liba]) libraries. The sysrepo library provides datastore functionality according to RFC 8342 [BSS+18]. Libyang provides functions to parse YANG models. The libnetconf2 library consists of functions that help in building NETCONF servers and clients. Finally, netopeer2 makes use of the other three libraries and provides a ready-to-use NETCONF server and a basic client to go alongside the server.

3.2 End Station Configuration

For this thesis, a TSN NIC was chosen as the end station to be configured via NETCONF. The NIC configuration was done by using the Linux traffic control system (`tc`). `tc` offers various modes to define the queuing of packets, so-called queuing disciplines (qdiscs). One of those is the TAPRIO (Time Aware Priority Shaper) qdisc, allowing for traffic shaping according to IEEE 802.1Qbv. Listing 3.1 shows an example configuration in which the NIC with the name enp4s0 is configured with a taprio qdisc. `num_tc` sets the number of traffic classes to 4. The `map` parameter maps Linux network priorities to the different traffic classes. Those traffic classes are also mapped to hardware queues with the `queues` parameter. `base-time` defines the start time of the schedule in nanoseconds. `flags 0x2` enables the full-offload feature in which the gate control list is handled by the NIC. Finally the `sched-entry s <gatemask> <interval>` parameters set which which traffic classes are active in which interval. In this example the traffic class 0 is open for 20ms, followed by traffic classes 1 and 2 for 10ms. A more detailed description of the individual parameters can be found at `man tc-taprio`.

```
tc qdisc add dev enp4s0 parent root handle 100 taprio \
   num_tc 4 \
   map 0 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 \
   queues 1@0 1@1 1@2 1@3 \
   base-time 5000 \
   flags 0x2 \
   sched-entry S 01 20000000 \
   sched-entry S 06 10000000
```

Listing 3.1: Example Configuration

3.3 Design

This work takes the existing netopeer2 project as a basis and adds a daemon that reads data from sysrepo and configures a device. Figure 3.1 illustrates how the different components work together. The upper half of the diagram is covered by netopeer2. A server and client are started in order to communicate with each other. The resulting data are written to the datastores provided by sysrepo. Sysrepo offers multiple ways to access the data stored in its datastores. One can choose between directly calling the sysrepo function, or taking the indirect approach by either developing a plugin or a daemon [sysd]. Sysrepo provides a small daemon called sysrepo-plugind that groups all plugins loaded to sysrepo into a single process. For this thesis, the approach of developing a daemon was chosen, simplifying the writing and testing of code, since fewer steps had to be executed in order to test changes. Plugins are favored if multiple daemons are needed, this was not the case in this implementation though. The daemon is a stand-alone program that subscribes to changes made in sysrepo, creates TSN configurations based on those values and sends that configuration to the TSN-capable device.

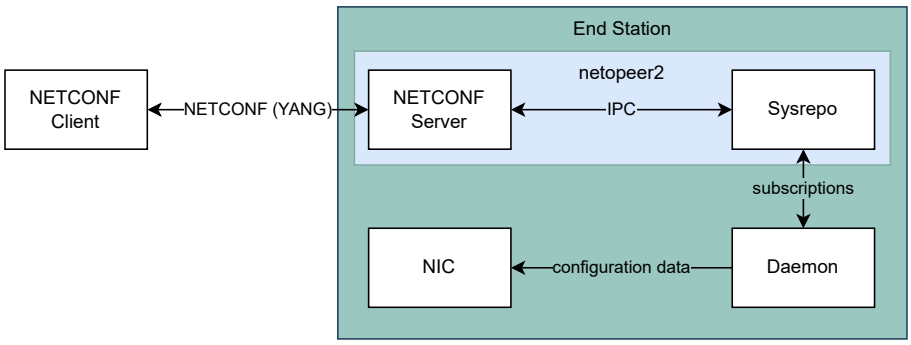

Figure 3.1: Interactions Between Components

3.4 Limitations

In this thesis, the following limitations were set and are open to improvement for future works.

- The YANG files used in this implementation (as described in Section 4.1) build up a specific hierarchy to be used. This leads to the parameters used to create a TSN schedule to be child nodes of the bridge interface. As this thesis concerns itself with the configuration of an end station, a bridge would not be necessary. A rather unusual configuration file is used to comply with the provided structure, where a bridge with a single interface is wrapped around the actual configuration.
- In this thesis the NETCONF server subscribes to changes in the running datastore. A configuration based on those parameters is then sent to the Ethernet card. Reading the actual configuration from the card and writing it back to the operational datastore is not implemented. In an industrial setting, it might be desirable not just to write but also to read the actual configuration of a device, in order to keep a better overview on the current state of devices.
- While there are ways to replace a running schedule on a network interface card with a new one by using different parameters and handles for the tc command that is configuring the card, a simpler approach is used in this thesis. Instead of modifying the reconfiguration command based on the current configuration, the active schedule is simply deleted prior to sending a new configuration. Without any configuration, the card just accepts all traffic until the new schedule starts.
- Error handling is kept to a minimum. The YANG files implementing IEEE 802.1Qbv already provide a first check of the parameters, but it is still possible to create an invalid configuration. Once a configuration change is detected, the daemon will try to reconfigure the card. Whether the sent parameters are actually valid for the card or even if the card is reachable at the moment of configuration will not be sent back to the NETCONF client. The daemon, however, will provide an appropriate error message. Together with the former limitation of not filling the operational datastore, this results in the user not knowing the result of a reconfiguration without looking a the output of the daemon.

CHAPTER 4

Implementation

For the actual implementation of the project described in the design chapter, two components were needed. A NETCONF client and server and a daemon for sysrepo. This chapter starts by describing how to correctly set up the netopeer2 server in order to use TSN configuration. This is followed by detailed description of the daemon developed for this thesis.

4.1 Setting up the Netopeer2 Server

The netopeer2 project was set up on a local machine according to its compilation and installation guide, found at [Netb]. After that, the following YANG modules were loaded in sysrepo using the sysrepoctl command in order to enable server and client to use TSN configuration parameters.

- iana-if-type@2023-01-26
- ieee802-dot1q-bridge@2023-10-26
- ieee802-dot1q-sched-bridge@2023-10-26
- ieee802-dot1q-sched@2023-10-22 with the scheduled-traffic feature enabled
- ieee802-dot1q-types@2023-10-26
- ieee802-types@2023-10-22
- ietf-interfaces@2018-02-20

4.2 The Daemon

With netopeer2 providing a NETCONF client, server and the needed datastores via sysrepo, all that is missing is the daemon detecting changes in the configuration and passing them along to the NIC. Sysrepo offers a variety of functions to connect to a datastore and extract the needed information. The following sections give an overview of the implementation of the daemon with a focus on which functions were used to satisfy the requirements needed for this task. The full code for the application can be found at GitHub at [cod]. A resource worth mentioning is the sysrepo documentation located at [sysa].

4.2.1 General Structure of a Sysrepo Daemon

There are two initial steps to run a sysrepo daemon. Firstly, a connection to the desired datastore is needed, secondly, a subscription to the modules that one is interested in needs to be established. After those are in place, changes in the datastore alert the program and can be acquired using the appropriate functions. What is done with that data afterwards depends on the requirements of the daemon. Writing a plugin instead of a daemon is a very similar progress, but instead of connecting to the datastores, a callback function called `sr_plugin_init_cb()` needs to be implemented in which the subscriptions to the modules are established. Sysrepo provides an example plugin, which can be found at [sysb], that explains the basic functionalities.

4.2.2 Interfacing with the Datastore

A connection to the desired datastore can be established using the two functions shown in Listing 4.1. `sr_connect` will create the connection to sysrepo and, in a second step, `sr_session_start` connects to the specified datastore. This daemon is only interested in configuration changes made in the running datastore, though there are functions provided by sysrepo, if switching datastores is necessary.

```
int sr_connect(
    const sr_conn_options_t opts,
    sr_conn_ctx_t** conn
)
int sr_session_start(
    sr_conn_ctx_t* conn,
    const sr_datastore_t datastore,
    sr_session_ctx_t** session
)
```

Listing 4.1: Connecting to the datastore

The next step in acquiring data is to set a subscription on the modules of interest. Listing 4.2 shows the function to do this. One of the notable parameters is the callback parameter, defining which function to call if changes occur. A second parameter worth mentioning is the opts parameter. In this daemon it is set to not only call the callback function at datastore changes but also once at startup of the daemon.

```
int sr_module_change_subscribe(
    sr_session_ctx_t* session,
    const char* module_name,
    const char* xpath,
    sr_module_change_cb callback,
    void* private_data,
    uint32_t priority,
    sr_subscr_options_t opts,
    sr_subscription_ctx_t** subscription
)
```

Listing 4.2: Setting up a Subscription

With those settings in place, the specified callback function will be called every time there is a change of configuration in the observed values. There are multiple ways to acquire those changes, like iterating through all changes or reading and parsing whole trees of data. In this daemon, for the sake of simplicity, the approach to extract the data was to just query for each value that was needed to generate a valid tc command, whether the value actually changed or not. Listing 4.3 shows the function used to get the individual values.

```
int sr_get_item(
    sr_session_ctx_t* session,
    const char* path,
    uint32_t timeout_ms,
    sr_val_t** value
)
```

Listing 4.3: Getting Data

4.2.3 Filtering for Data Using XPath

Most functions shown above include a parameter called either a path or xpath. XPath is an expression language designed to query for specific elements within XML documents. Adding this path allows to extract exactly the configuration data one is interested in. The YANG modules used for TSN create a somewhat deep XML tree, leading to a rather long xpath string for all modules. Listing 4.4 shows the line used in the daemon to retrieve the values to be used as sched-entries in the tc command.

```
sr_get_items(
    session,
    "/ietf-interfaces:interfaces/interface/ieee802-dot1q-
        bridge:bridge-port/ieee802-dot1q-sched-bridge:gate-
        parameter-table/admin-control-list/gate-control-entry
        /*",
    0, 0, &val, &val_cnt
);
```

Listing 4.4: Acquireing the Values for TCs sched-entry

4.2.4 Interfacing with the NIC

For this project the Intel Ethernet-Controller I225 was chosen to act as the TSN NIC to be configured using NETCONF. It supports various TSN features, including IEEE 802.1Qbu, 802.3br, 802.1Qbv, 802.1AS-REV, 802.1p/Q, and 802.1Qav. The device can be configured using the `tc` command as already explained in Section 3.2. The daemon combines the parameters that are fixed by the specifications of the card, like the number of available traffic classes and the parameters that are extracted from the datastore as explained above and crafts a fitting tc command that is then sent to the card. Listing 4.5 shows the code directly interfacing with the card. This is done by using the `system` command that executes the char* given to it as a shell command. Firstly `system` is called with the deletion command as the parameter. Included in that command is the string `> /dev/null 2>&1` which redirects any resulting output to `/dev/null`, since it is not of interest. Deleting an old configuration fails for example, when there simply is none to delete. As mentioned in the Section 3.4, this could be improved in future versions. Secondly, `system` is called again, this time with the command for the new configuration, which was crafted earlier in the code through string manipulation. This time the return value is checked in an if-else statement, as it signals whether the configuration worked or not.

```
//delete old config
system("tc qdisc delete dev enp4s0 parent root > /dev/null
   2>&1");

//send new config
if(system(cmd)==0){
    printf("Configuration successfull\n");
} else{
    printf("Error writing to card\n");
}
```

Listing 4.5: Sending the Command to the NIC

CHAPTER 5

Evaluation

The daemon was tested in a few different scenarios. For the test setup, a NETCONF client was started on one device, the NETCONF server on different device. The daemon was then started on the same device as the server. After connecting the client to the server using the ssh setting, different configurations where sent to the server. After recognizing a change in the server's running sysrepo, the daemon copied the new configuration to the NIC. The success of that could be observed using the `tc show` command to monitor the settings of the NIC. Additionally, Wireshark was used to observe the actual cycle of the NIC. The following sections show the results of a few different valid and invalid configurations.

5.1 Test Results

Four different scenarios, both with valid and invalid configurations, were tested. The following sections document the different responses of the NETCONF server, the daemon and the NIC.

5.1.1 Valid Configuration

For the first scenario a valid configuration was sent to the NETCONF server and properly copied by the daemon to the NIC. The configuration cycles between opening the channel for packets belonging to traffic class 0 for 20ms and opening for all other traffic classes for 10ms. To test this scenario, traffic classified as belonging to traffic class 0 was sent to the NIC and measured using Wireshark. Figure 5.1 displays the behavior of the NIC.

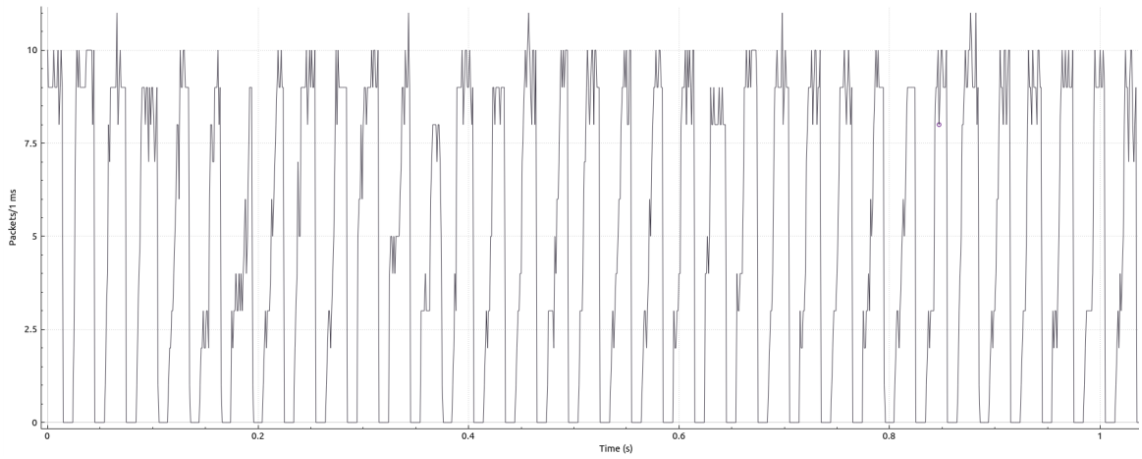

Figure 5.1: Wireshark - Channel Cycling Between Being Open for 20ms, then Closed for 10ms

5.1.2 Violating YANG Restrictions

For the second scenario an invalid configuration was used. The implemented YANG models set limitations for most values, providing a guideline on how values have to be formatted, how big or small they can get in relation to other values, etc. In this scenario a cycle-time bigger than the highest supported cycle-time was chosen. Writing the invalid configuration to the candidate datastore works since candidate does note require valid configuration. Once it is copied over to the running datastore however, the configuration fails the validity check by the server, where sysrepo validates it and replies with an error message. This prompts the NETCONF client to reply with the error message shown in Figure 5.2. In this case no configuration is saved in the server's sysrepo and, therefore, the daemon does not have to copy anything to the NIC.

```
> edit-config --target candidate --config=config.xml --defop replace
OK
> copy-config --target running --source candidate
ERROR
        type:     application
        tag:      operation-failed
        severity: error
        message:  admin-cycle-time must not be greater than supported-cycle-max (Data
location "/ietf-interfaces:interfaces/interface[name='enp4s0']/ieee802-dot1q-bridge:br
idge-port/ieee802-dot1q-sched-bridge:gate-parameter-table/admin-cycle-time".)

        type:     application
        tag:      operation-failed
        severity: error
        message:  User callback failed.
```

Figure 5.2: YANG Error Message when Sending an Invalid Configuration to the NETCONF Server

5.1.3 Card Failure

The third scenario presents the problem that arises when a configuration that is valid according to the corresponding YANG module is sent to the server, but is still failing to be written to the card. This occurs when either the card is simply not available at the time of writing, or when the configuration exceeds the physical limitations of the card. Form client side, no error is visible, since the server accepts the configuration without any problems (depicted in Figure 5.3). Only when the daemon tries to copy the new configuration to the card, the problem occurs.

```
> edit-config --target candidate --config=config.xml --defop replace
OK
> copy-config --target running --source candidate
OK
```

Figure 5.3: The Client does not Detect an Error when the NIC is Offline

Figure 5.4 shows the error message displayed by the daemon in the case of the NIC not being available at the time of reconfiguration. Future work could improve this scenario by adding a way inform the client of the failure.

```
Cannot find device "enp4s0"
Error writing to card
```

Figure 5.4: Error Message from the Daemon when NIC is Offline

5.1.4 Testing the Card Boundaries

In this last scenario, a valid configuration was sent to server and correctly applied to the NIC. While the NIC will accept any valid configuration, a configuration close to the NIC's limits might lead to errors. In this test case, a configuration was sent to the card alternating between opening the channel for packets traffic class 0 for 20ms and all other traffic classes for 50ms. After successfully applying this configuration, traffic associated with traffic class 0 was sent to the NIC. The traffic was captured using Wireshark. Figure 5.5 shows that while the NIC accepts the configuration, it can not quite handle it. This can be observed in the windows where traffic class 0 is closed, as there are still regular spikes where a single packet gets through. This behavior could not be observed in intervals shorter than 15ms.

5.2 Execution Time

Next, the execution times of different commands and components were evaluated. To measure the execution time of the individual commands between the NETCONF server and client, Wireshark was used. The time measured is the time that elapsed between the first packet sent to the server after entering a command and the last packet received

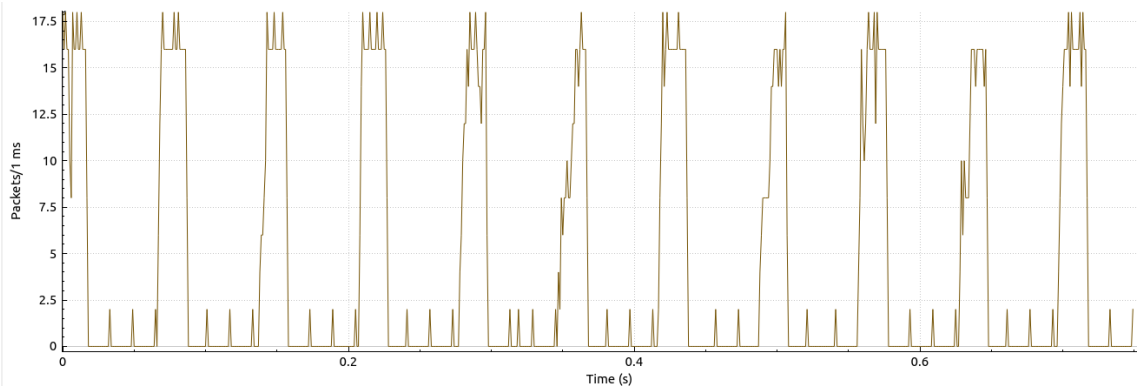

Figure 5.5: Wireshark - Channel Cycling Between Being Open for 20ms, then Closed for 50ms

in this burst of communication. These measurements are meant to provide a vague understanding of how long different actions take, but are to be taken with a grain of salt, since actual timings might vary from setup to setup, depending on type of connections or configuration choices made in setting up server and client. Figure 5.6 displays the durations measured.

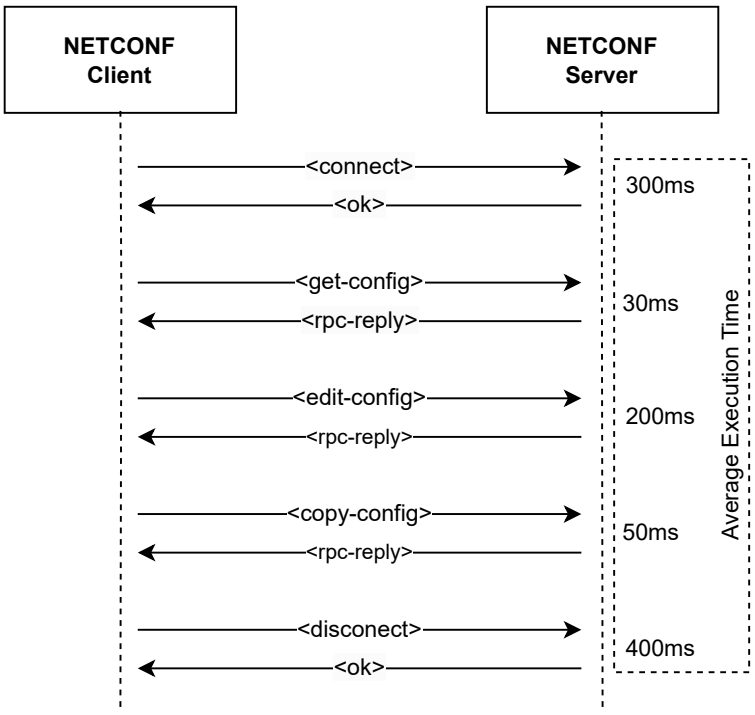

Figure 5.6: Average Duration of NETCONF Operations

As a second measurement, the time of the daemon between recognizing a change in sysrepo and finishing the reconfiguration of the NIC was taken, which is roughly 2ms. This time represents only how long it takes for the NIC to be aware of the new configuration. The actual time at which the NIC starts executing the new configuration depends on the base-time set in the configuration file.

CHAPTER 6

Conclusion

This thesis presented the idea of using a NETCONF server observed by a sysrepo daemon to bridge the gap between a network using NETCONF and a TSN NIC not capable to do that. A first draft of a daemon was presented and tested against various scenarios. The implementation shows that the concept of creating a daemon to act as the bridge between a server and an end station could be a viable way of integrating those end stations into any network. As discussed in Section 3.4 - Limitations, there are a few open topics left though, that need addressing first. The daemon presented keeps error handling to a minimum. Some of these errors are hardware-specific, but depending on the end station used, this could be added in the implementation with an acceptable amount of effort. A limitation that is presenting a larger workload is implementing a way to read data from the TSN NIC. Without the knowledge of what the actual status of configuration in the TSN NIC is, critical errors might go unnoticed, which is certainly not an acceptable event in an industrial setting. The questions arises, if simply reading the active configuration and writing it back to the operational datastore so that it could be accessed using the <get-config> command of a NETCONF client would be enough, or if the client needs more notifications which could be realized by creating a custom YANG module, augmenting those used in this prototype, to add more functionality based on the end station in use. But that question will be left as future work at this point in this thesis.

List of Figures

List of Tables

Acronyms

IETF Internet Engineering Taskforce. 5

IT information technology. 1

NETCONF Network Configuration Protocol. vii, 2, 5–11, 15, 16, 18–20, 22–26, 29, 31, 33

NIC Network Interface Card. 2, 15, 16, 20, 22–25, 27, 29, 31

NMS Network Management System. 2, 6

OT operational technologies. 1

qdiscs queuing disciplines. 16

RPC remote procedure call. 5

TSN Time-Sensitive Networking. vii, 1, 2, 12, 13, 17–19, 21, 22, 29, 31

XML Extensible Markup Language. 5, 6, 21

YANG Yet Another Next Generation. 2, 6, 7, 9–11, 16, 18, 19, 21, 24, 25, 29, 31, 33

Bibliography

[91220] Ieee standard for local and metropolitan area networks–timing and synchronization for time-sensitive applications. *IEEE Std 802.1AS-2020 (Revision of IEEE Std 802.1AS-2011)*, pages 1–421, 2020.

[Bjö10] Martin Björklund. YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). RFC 6020, October 2010.

[Bjö16] Martin Björklund. The YANG 1.1 Data Modeling Language. RFC 7950, August 2016.

[BSS+18] Martin Björklund, Jürgen Schönwälder, Philip A. Shafer, Kent Watsen, and Robert Wilton. Network Management Datastore Architecture (NMDA). RFC 8342, March 2018.

[cod] Code github repoistory. `https://github.com/hartnern/netconf4tsn`. Accessed: 05-02-2024.

[EBBS11] Rob Enns, Martin Björklund, Andy Bierman, and Jürgen Schönwälder. Network Configuration Protocol (NETCONF). RFC 6241, June 2011.

[Fin18] Norman Finn. Introduction to time-sensitive networking. *IEEE Communications Standards Magazine*, 2(2):22–28, 2018.

[IET] Ietf community wiki. `https://wiki.ietf.org/group/netconf`. Accessed: 06-10-2023.

[LBS19] Lucia Lo Bello and Wilfried Steiner. A perspective on ieee time-sensitive networking for industrial communication and automation systems. *Proceedings of the IEEE*, 107(6):1094–1120, 2019.

[liba] libnetconf2 github repoistory. `https://github.com/CESNET/libnetconf2`. Accessed: 16-01-2024.

[libb] libyang github repoistory. `https://github.com/CESNET/libyang`. Accessed: 16-01-2024.

[Mes18] John L. Messenger. Time-sensitive networking: An introduction. *IEEE Communications Standards Magazine*, 2(2):29–33, 2018.

[Neta] Netconf central. `https://www.netconfcentral.org/tools-list`. Accessed: 06-10-2023.

[Netb] Netopeer2 github repoistory. `https://github.com/CESNET/netopeer2?tab=readme-ov-file#compilation-and-installation`. Accessed: 16-01-2024.

[SPFA19] Luis Silva, Paulo Pedreiras, Pedro Fonseca, and Luis Almeida. On the adequacy of sdn and tsn for industry 4.0. In *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 43–51, 2019.

[sysa] Sysrepo documentation. `https://netopeer.liberouter.org/doc/sysrepo/master/html/modules.html`. Accessed: 05-02-2024.

[sysb] Sysrepo documentation. `https://netopeer.liberouter.org/doc/sysrepo/libyang1/html/example.html`. Accessed: 05-02-2024.

[Sysc] Sysrepo github repoistory. `https://github.com/sysrepo/sysrepo`. Accessed: 16-01-2024.

[sysd] Sysrepo plugin vs daemon. `https://netopeer.liberouter.org/doc/sysrepo/master/html/index.html#about`. Accessed: 15-02-2024.

[XDK+18] Min Xu, Jeanne M David, Suk Hi Kim, et al. The fourth industrial revolution: Opportunities and challenges. *International journal of financial research*, 9(2):90–95, 2018.