Universitat
de les Illes Balears

MASTER'S THESIS

DEVELOPMENT OF A CENTRALIZED NETWORK CONTROLLER FOR NETWORKS BASED ON THE TIME-SENSITIVE NETWORKING (TSN) ETHERNET STANDARD

Andreu Marc Servera Lauder

Master's Degree in Intelligent Systems (MUSI)

Specialisation: Internet of Things

Centre for Postgraduate Studies

Academic Year 2020-2021

Development of a Centralized Network Controller for Networks based on the Time-Sensitive Networking (TSN) Ethernet Standards

Andreu Marc Servera Lauder
Tutor: Inés Álvarez Vadillo, Julián Proenza Arenas
Trabajo de fin de Máster Universitario en Sistemas Inteligentes (MUSI)
Universitat de les Illes Balears
07122 Palma, Illes Balears, Espanya
andreumarc12@gmail.com

***Abstract*—The constant evolution of the Industry 4.0 requires industrial network communications to continuously improve to meet the requirements of the existing industrial operations. Cyber-physical systems enable new ways of production, value creation and real-time optimization, providing means for the customization and personalization of products. To make this change possible, industrial network communications have to provide deterministic real-time properties and the ability to adapt to the constant changes in the requirements of the product line.**

Time-Sensitive Networking (TSN) is a set of new IEEE standards that aims to provide deterministic real-time communications over Ethernet. TSN also provides centralized online configuration and control architectures which eases the deployment of large networks. Moreover, it enables the reconfiguration of the network in order to adapt to changes in the production chain and make a better use of the network's resources.

In this project we implement a Centralized Network Controller (CNC) as software in charge of the automatic configuration of a set of TSN bridges that make use of TSN standards. We also perform a verification process of the CNC using commercial TSN bridges.

RESUMEN

La constante evolución de la Industria 4.0 requiere una constante mejora de las redes de comunicación industrial para cumplir los requisitos de las operaciones industriales existentes. Los sistemas ciberfísicos hacen posible nuevas formas de producción, creación de valor y optimización en tiempo real, proporcionando medios para la personalización de productos. Para que sea posible, las redes de comunicación industrial deben proporcionar propiedades de respuesta determinista y en tiempo real a la vez de poder adaptarse a los cambios constantes de los requisitos de la línea de producción.

Time-Sensitive Networking (TSN) es un conjunto de nuevos estándares del IEEE que buscan proporcionar determinismo y propiedades de tiempo real en Ethernet. TSN también proporciona nuevas arquitecturas para la configuración y control de la red de manera centralizada, facilitando el despliegue de nuevas redes de gran tamaño. Además, habilita medios para la reconfiguración de la red para adaptarse a los cambios en la línea de producción y así hacer un mejor uso de los recursos de la red.

En este proyecto implementamos un Controlador Centralizado de Red (CNC, del inglés *Centralized Network Controller*) como software encargado de automatizar la configuración de un conjunto de puentes de red (*bridges* en inglés) que usan los estándares de TSN. Además realizamos la verificación del CNC usando puentes de red comerciales.

***Index Terms*—Time-Sensitive Networking, Time-Aware Shaper, Centralized Network Controller, Network Configuration, Real-time**

I. INTRODUCTION

The rise of novel applications such as the Industrial Internet of Things (IIoT) [24], autonomous driving [22] or the intelligent management of utilities [21] is leading to a change in the infrastructures that have traditionally supported industrial applications. Specifically, these applications keep the common characteristics of industrial applications, such as the interaction with the real world and the need for providing a continuous correct service. On top of that, these novel applications are required to function in environments that dynamically change in an unpredictable manner, reconfiguring the available resources to adapt to the new necessities. In the case of distributed systems, one of the resources to reconfigure is the communication network, and this is the main focus of this project.

A. Background and motivation

Enhanced communications is one of the key factors in the emerging industrial applications, such as Industry 4.0, in order to achieve timely delivery and reliability in control data. Ethernet has been widely used in the industry for many years, but even though its speed has been improving over time, there is another key factor to take into account: determinism. A network is considered deterministic when it guarantees that the transmission of data will finish in a previously specified amount of time. Thus, it is an important aspect to consider when working with critical and time-sensitive data. But bare Ethernet lacks determinism as it is based on the *best-effort* principle [23]. However, it is possible to enhance it and provide means for determinism. This has been done in the past by some Ethernet-based proprietary communication technologies [17].

Time-Sensitive Networking (TSN) is a set of new standards that provides means to enable deterministic real-time (RT) communications over Ethernet sitting on layer 2 of the ISO/OSI Model, that is the Data Link layer [18]. TSN also provides centralized online configuration and control architectures which eases the process of configuring large networks and enables the network to reconfigure in order to adapt to changes in the environment.

TSN specifies several standards but we pay special attention to the Time-aware Shaper (TAS) standardised in IEEE Std 802.1Qbv [13], which defines means to enforce a schedule designed to separate the communication on the Ethernet network into repeating cycles of fixed length, following the concept of time-division multiple access (TDMA). In this way, time-critical communication can be separated from non-critical background traffic.

As mentioned, TSN also provides centralized online configuration and control. To do so it proposes a centralised architecture to configure all the aspects of the network, including TAS [14]. This central configuration is done by what they define as the Centralized Network Controller (CNC) which can be seen in Figure 6. Unfortunately, TSN does not specify the implementation of the CNC, thus there is a need to do so, which is the main focus of this project.

B. Project objectives

In this subsection we explain the objectives of this project. Throughout this document, all the objectives indicated in this section will be covered. The goal of this project is to implement a Centralized Network Control element (CNC) to automatically configure the TAS of a set of TSN bridges. In this project, this main goal is going to be divided into three objectives:

1) The implementation must adhere to the specifications of the P802.1Qcw. This standard defines the YANG data model for the TAS, which allows to enforce RT for scheduled traffic. Thus, the first objective of this project is to implement a real prototype of the YANG model that is currently proposed in the standard.
2) The implementation has to meet requirements concerning the user's point of view and the development point of view. From the user's point of view, as the goal is to automatically configure the TAS of the bridges, it is of utmost importance that this process requires the least supervision possible. From the development point of view, the implementation has to enable means of integration with other possible applications, to ease the process of possible functionality changes in the future and other requirements that will be described throughout the document.
3) The implementation and verification of the CNC in this project has to be carried out using commercial TSN bridges as a working environment. By doing this, the CNC will be able to be deployed in a physical system.

C. Tasks of the project

To be able to achieve the previously mentioned objectives the project has been divided into a series of tasks that have been individually completed during its implementation. Here we list the tasks in a chronological order:

- Getting familiar with the set of standards of TSN, paying special attention to the IEEE Std 802.1Qbv, which describes the TAS.
- Understanding the current state of the art of the automatic configuration of TSN bridges.
- Designing a configuration file for containing the configuration parameters necessary for the TAS configuration of the bridges. After that, implementing an application responsible for parsing the configuration file.
- Implementing an application that builds a data instance with the previously parsed parameters that follows the YANG data model specified by the P802.1Qcw.
- Creating an application that automatically connects to the corresponding TSN bridges and applies the previously generated configuration.
- Verifying the correctness in all of the steps that together achieve the automatic configuration of the TASs.

II. Previous Work

In this section we explain the key concepts required to understand the work that we have carried out. First we talk about TSN, its set of standards and which of them are more directly related to this project. Then we mention some tools that TSN makes use of in order to configure the bridges of the TSN network. Finally we introduce the concept of Docker [1].

A. Time-Sensitive Networking - TSN

TSN is a set of standards that are being developed by the TSN task group of the *IEEE 802.1* working group. The goal of these standards is to define mechanisms to provide Ethernet with RT guarantees, fault-tolerance and online management of the network configuration. There are a number of standards that can be combined to build a myriad of different TSN networks. Below we explain the ones that we have taken into consideration while developing this work.

In TSN the communication follows a publisher-subscriber architecture, where the publisher is called a *talker* and the subscribers are called *listeners*. On top of that, TSN organises the communication using *streams*, as standardised in the IEEE Std 802.1Qat Stream Reservation Protocol (SRP) [12]. A stream is a virtual communication channel that connects each talker to all its intended listeners. Streams exhibit the characteristics of the traffic that they convey, such as the period, the frame size or the priority. The priority of the stream identifies the type of traffic that it conveys, namely time-triggered traffic, event-triggered traffic or best-effort traffic. The first two types of traffic are considered sensitive traffic as the expectation is to deliver the messages on time. Time-triggered traffic is one which is initiated at predefined points in time. Event-triggered traffic is initiated as a consequence of the occurrence of a significant event. On the other hand, best-effort traffic is all other kinds of traffic.

In order to provide timing guarantees to the different types of traffic, TSN proposes the IEEE Std 802.1AS [15] which

Figure 1. Division of communication time in cycles with a fix guard band. From [13]

standardises global clock synchronisation. This is one of the most important standards proposed in TSN as it is required to properly execute many of the other standards proposed by the working group.

Once the network is synchronized we can implement policies to provide the network with RT guarantees with the use of a traffic shaper. Traffic shaping is a bandwidth management method that delays the transmission of certain types of traffic in order to ensure network performance for higher priority traffic. Traffic shapers distribute packets evenly in time in order to smooth out the traffic and avoid overwhelming the buffers in subsequent bridges along the path.

TSN proposes two different traffic shapers that dictate the transmission of RT traffic. On the one hand, TSN proposes the IEEE Std 802.1Qbv Time-Aware Shaper (TAS) [13], which is used to provide hard RT guarantees to time-triggered traffic. On the other hand, TSN proposes the IEEE Std 802.1Qav Credit Based Shaper [11], which provides soft RT guarantees to event-triggered traffic.

Between these two shapers we bring focus to the TAS, as the goal of this project is to automatically configure the TASs within the bridges of a TSN network. In the next section we explain the TAS in more detail.

B. TSN's Time-Aware Shaper

The TAS is a mechanism added onto the Ethernet stack to provide TSN networks with deterministic behaviour. TAS is used to provide timing guarantees to the network of systems that interact with the environment, e.g. an automotive safety system. Let us explain how the TAS works.

In Figure 1 we see the structure of a TSN communication time division that uses the TAS. The communication time is repeated in so called cycles of fixed time lengths. Each cycle is divided in two main slots, one for time-triggered traffic, called the protected window, and another for event-triggered traffic. Also, if there is enough time after transmitting the event-triggered traffic, best-effort frames can be transmitted. In the beginning of the communication cycle we see a fixed guard band, a period of time in which no frames can be transmitted in order to prevent them from interfering with the time-triggered traffic.

In Figure 2 we see a diagram of the transmission gates of the TAS. The frames assigned to a cycle are eventually transmitted through a *transmission gate* (or *gate* for short). The Qbv [13] amendment introduces this concept to enforce the communication in cycles and windows. As you can see in Figure 2 each transmission gate is associated with a queue,

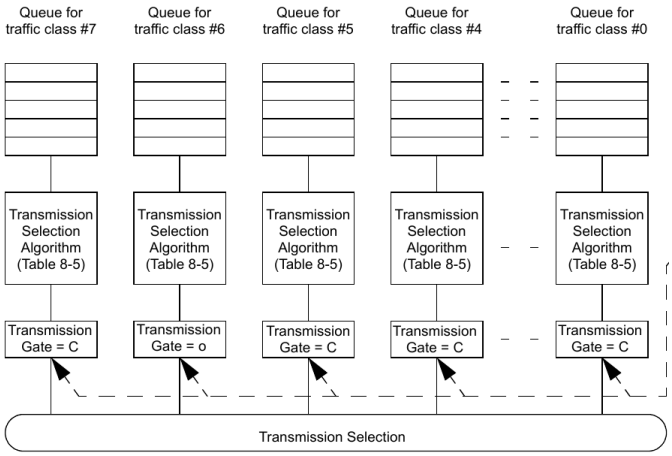

Figure 2. Diagram of the transmission gates of the TAS. From [13]

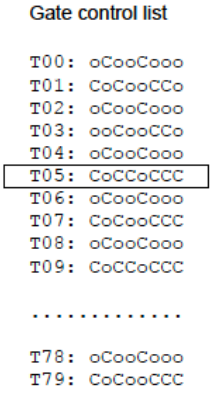

Figure 3. Structure of a Gate Control List (GCL). From [13]

one for each traffic class. The state of the transmission gate determines whether queued frames can be selected for transmission or not. For a given queue, the transmission gate can be in an *open* state or a *closed* state. If the gate is open, queued frames are selected for transmission. On the other side, if the gate is closed, queued frames are not selected for transmission.

In order to define the states of each gate the TAS makes use of the *Gate Control List* (GCL). A GCL is associated with each port and contains an ordered list of gate operations. Each gate operation changes the transmission gate state for each one of the traffic class queues of the port. The GCL also specifies the duration of each gate operation. In Figure 3 we see the structure of a GCL, where each entry is composed by the time interval Tn and the state of each gate: o for open and C for closed. If several gates are open at the same instant of time, the transmission selection algorithm selects which one of them has a higher priority. Each entry of the GCL is read until it reaches the last entry. At this point the GCL is repeated from the beginning in a cyclic manner.

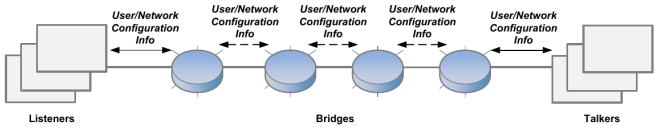

Figure 4. Fully distributed configuration architecture. From [14].

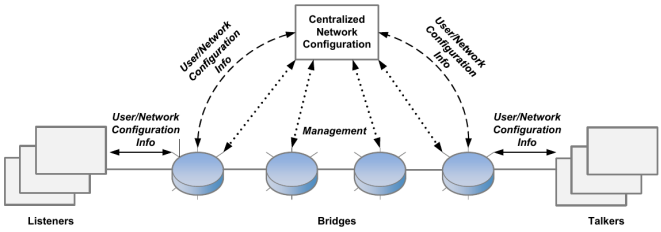

Figure 5. Fully centralized network/distributed user configuration architecture. From [14].

C. Configuration Architectures of a TSN Network

The SRP is an enhancement to Ethernet to provide end-to-end management of resource reservations for data streams requiring guaranteed Quality of Service (QoS) in Local Area Networks (LANs). The IEEE Std 802.1Qcc [14] amendment covers enhancements to the SRP. These enhancements include the capability of creating more traffic classes, the main contribution being the definition of the *scheduled traffic* class which is time-triggered traffic with hard RT requirements. The amendment also defines two new architectures to allow the management of the network requirements in real-time. In the following subsection we explain these new architectures.

1) Configuration Architectures: The SRP proposed in previous standards by IEEE uses a fully distributed approach in order to perform the reservation of resources. The new architectures proposed in TSN allow the management of the network in a centralized manner to have a single vision of the network configuration and the available resources, increasing the efficiency of the configuration process. More specifically, it defines two new architectures. Next we describe both the legacy distributed architecture and the two new ones proposed in TSN.

In Figure 4 we see the *fully distributed model*. As you can see, the bridges are configured in a distributed manner and the Talkers/Listeners exchange the configuration information with the adjacent bridge which then forwards the configuration messages to other bridges in the network.

In Figure 5 the *centralized network/distributed user model* is depicted. With respect to the previous configuration, the bridges share the traffic requirements of the Talkers/Listeners with the CNC instead of propagating them through the adjacent bridges. The CNC has a complete view of the physical topology of the network, which enables the CNC to centralize complex computations.

Finally, in Figure 6 we show the *fully centralized model*. In this case, there is again a CNC, which centralizes the final transmission of the configuration information to the bridges, as in the previous model. The difference is, the user configuration

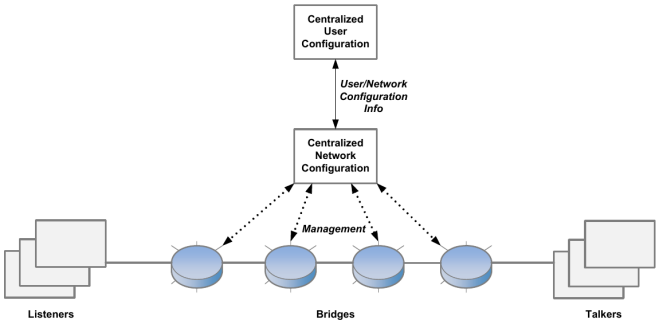

Figure 6. Fully centralized configuration architecture. From [14].

is also centralized in the Centralized User Controller (CUC).

These three network models are possible in TSN. However, since the fully distributed architecture was designed prior TAS, this architecture does not support the configuration of TAS. Thus, only the *centralized network/distributed user model* and the *fully centralized model* enable the use of this traffic shaper. For this reason, in this project we will only consider these last mentioned models.

2) Elements of the centralized models:: An application can request changes in the traffic which are then translated into changes in the network configuration. This is enabled by the User/Network Interface (UNI) which is responsible for managing the exchange of configuration information between the user side and the network side. The user side is represented by the Talkers and Listeners, i.e. the end stations. The rest of the network components, the bridges, represent the network side. On the one hand, in the fully centralized configuration architecture, the UNI is placed between the CUC and the CNC. On the other hand, in the centralized network/distributed user configuration architecture, the UNI is placed between then end-devices (Talker/Listener) and the first bridge that they are connected to.

The Centralized Network Configuration (CNC) is defined in the Qcc [14] amendment as a centralized component that configures network resources on behalf of TSN applications (users).

Finally, the Centralized User Controller (CUC) is an application that establishes a common ground between the CNC and the end devices. It communicates with the CNC to request the specific requirements that the end-points need to enable a deterministic communication.

D. NETCONF and YANG

In order to be able to configure the TASs of the bridges it is necessary to have a communication protocol to transfer the configuration information. NETCONF is a protocol standardized by the IETF (Internet Engineering Task Force) and it provides mechanisms to install, manipulate and delete the configuration of network devices. It uses operations called *Remote Procedure Calls* (RPCs) to execute certain configuration procedures on the desired device such as, get/set configuration, modify configuration, etc.

NETCONF follows a client-server architecture. In each of the TSN bridges there is a NETCONF server that enables the interaction with a client. The client can send commands in the form of RPCs that allow the client to act on the configurations. The client in this case is the CNC, which connects to the NETCONF server of the corresponding bridge. The CNC can send any NETCONF command, for instance a *get-config* RPC (used to retrieve the current configuration from a specified datastore), which is then processed by the NETCONF server.

All the RPCs are used with reference to a *datastore*. In NETCONF, datastores are a conceptual place to store and access information. A datastore might be implemented, for example using files, a database, flash memory locations or combinations. The objective of using datastores is to get a device from its initial default state into a desired operational state. NETCONF defines three datastores: *running*, *startup* and *candidate*, but users can define any number of additional datastores.

To prepare the configuration information to be sent via NETCONF it is possible to use YANG, a data modeling language. In comparison with other modeling languages such as UML (Unified Modeling Language), YANG is specifically targeted to the needs of configuration management. It provides a human readable representation, hierarchical configuration data models, reusable types and groupings (defined data structures that can be used in multiple YANG modules) and support for the definition of RPC operations among other capabilities.

It is important to note two different concepts within YANG. On the one hand, there is the *YANG module*, a file containing the specifications of a data model. On the other hand, there is the *YANG instance* (or instantiation), which contains the values of a configuration following the corresponding YANG module.

Another important aspect about YANG is that it is a data modeling language that is protocol independent. That is, a YANG data model or instance can be converted in any encoding format (i.e. XML or JSON) that the network configuration protocol supports.

Therefore, with the use of NETCONF and YANG we are able to configure the bridges according to a previously generated configuration in an automated manner. In Section III we go more in depth on how NETCONF is used and how the configuration is stored in a YANG data model.

E. Docker

In this project we use Docker [1] to implement our CNC. Docker is a set of platform as a service products that use operating system level virtualization to deliver software in packages called containers. A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Docker images are built using a so called *Dockerfile*, a file containing a set of instructions used to build the image itself.

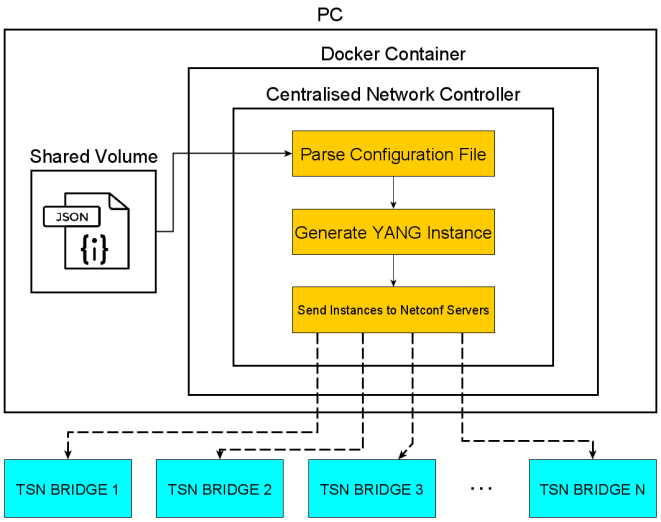

Figure 7. Diagram of the project's implementation strategy.

Once the image is created, its execution results in a container (an image at runtime).

During the development of this project we realized that there were a significant amount of software dependencies in order to make the CNC meet all its requirements. Thus, Docker provides us high maintainability, as we can deploy the container anywhere without installing the dependencies. In Section IV we explain how the Docker image is built and how the CNC is executed inside the container.

III. Implementation Strategy

In this section we describe the implementation strategy of the project. To have a better understanding we will refer to the diagram in Figure 7 in which the general implementation strategy of the project is depicted.

Starting from a general perspective, the CNC implemented in this project is executed in a PC. In this PC we have a Docker container. Inside the Docker container runs the software program that is in charge of the automatic configuration of the bridges' TAS. Note that TSN does not specify if the CNC is a physical entity or software. In this project we implement the CNC as software, thus from this moment on, when we talk about the CNC we refer to the software that is implemented in this project which is in charge of the automatic configuration of the bridges' TAS.

The CNC executes a series of processes: the parsing of the configuration file; the generation of the YANG instances for each specified bridge; and finally the transmission and deployment of the resulting YANG instances to the corresponding TSN bridges.

As we can see in Figure III, the PC also contains a *Shared Volume*. This is a directory inside the PC that can also be accessed by the Docker container. In this directory we find the configuration file which the user can modify directly from the PC without having to access the Docker container. This feature is explained in detail in Section IV-D.

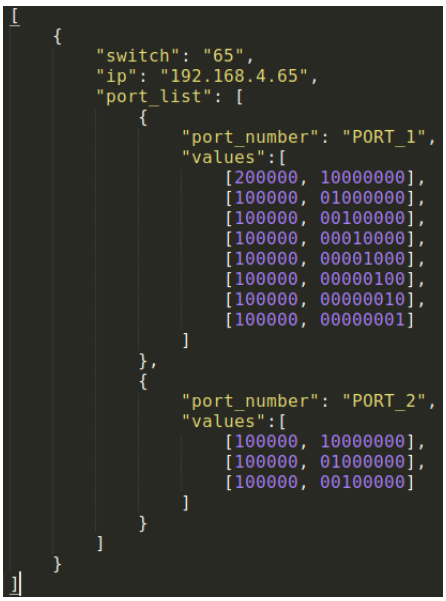

Figure 8. Example of the content of the configuration file using a JSON format.

A. *Configuration File*

The goal of this project is to automatise the configuration of the TAS of a series of bridges that together form part of a TSN network. In order to configure each one of the bridges we use a configuration file that contains all the necessary information to configure TAS in each bridge.

Figure 8 shows an example of the content of a configuration file. Specifically, the parameters that the configuration file must contain are the values that form the GCL that, as explained in Section II, contains the states of the gates of a port and the duration of those states. The configuration file must contain the GCL for each port of each bridge that form the TSN network.

We need to configure some other parameters in the configuration file on top of the ones previously described. To start with, we need to have the IP address of each bridge to be able to establish a connection with each one of them. With respect to other implementations of automatic configurations that are in the state of the art, we have implemented the CNC so that it can establish a connection with the bridges in an automatic way. In this way, our implementation helps us meet our second objective: "From the user's point of view, as the goal is to automatically configure the TAS of the bridges, it is of utmost importance that this process requires the least supervision possible."

Other configuration parameters that are indicated in the configuration file are the *switch name* and the *port number*. On the one hand, the switch name is used to reference the bridge that needs to be configured. On the other hand, the port number is necessary for the bridge to know to which port corresponds to each GCL. For instance, the bridges used in this project use this nomenclature: *PORT_X*, where *X* is a number between 0 and 3 that identifies one of the four TSN ports of the bridge.

The configuration file follows a previously defined format. For this, we decided to use JSON which stands for JavaScript Object Notation and it is a standard text-based format for representing structured data based on JavaScript object syntax. Even though it closely resembles JavaScript object literal syntax, it can be used independently from JavaScript. It is commonly used for transmitting data in web applications but it has been widely used in other applications [3].

The advantage of using this format is that it is highly human-readable and easy to understand. Also, it is good practice to operate with a commonly used format, especially when dealing with distributed systems, in order to increase the interoperability among systems. In this way, if we want to use an application to automatically generate the configuration files we only need to know the format of its output JSON file in order to use it.

As we can see in Figure 7, the CNC starts by reading the configuration file that is stored in the shared volume. Then it parses the information and translates it into another format, more specifically, it translates it into an XML file, the reason being that the NETCONF servers in the bridges use XML encoding to parse all the information received throuhg the RPCs. The filename of this XML corresponds with the switch name specified in the configuration file so that the CNC knows which file to use in the following steps of the automatic configuration. Additionally, the XML follows the data model specified by the TAS YANG model following our first objective: "The implementation must adhere to the specifications of the P802.1Qcw. This standard defines the YANG data model for the TAS, which allows to enforce RT for scheduled traffic. Thus, the first objective of this project is to implement a real prototype of the YANG model that is currently proposed in the standard."

Later on we explain how the information from the configuration file is translated into an XML file.

B. *The YANG Instance*

As we have said in the previous subsection, the CNC implemented in this project first reads the configuration file and parses the configuration information from it to then generate a YANG instance of the ports' TAS.

The IEEE specifies the *YANG scheduled traffic* module as part of the P802.1Qcw [7]. Bridges and end-systems that support scheduled traffic use this model to understand the configuration instances received from the CNC. Therefore, when a TSN device that implements TAS needs to be configured, a new YANG instance is created following the YANG scheduled traffic module and then it is sent to the NETCONF server of the corresponding device as an RPC. Then, the NETCONF server validates the YANG instance following the corresponding YANG module and, if it is correct, it applies the configuration.

That being said, in this project we implement a CNC that automatically generates YANG scheduled traffic instances that are then sent to the corresponding NETCONF servers in the bridges.

Later on in Section IV we explain in detail how the configuration instance is generated.

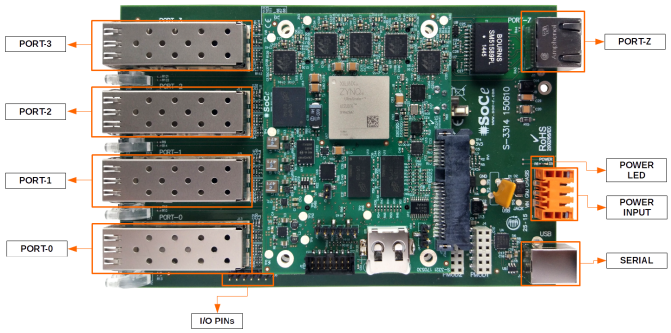

Figure 9. Picture of a SoC-e SMART MPSoC bridge. From [16]

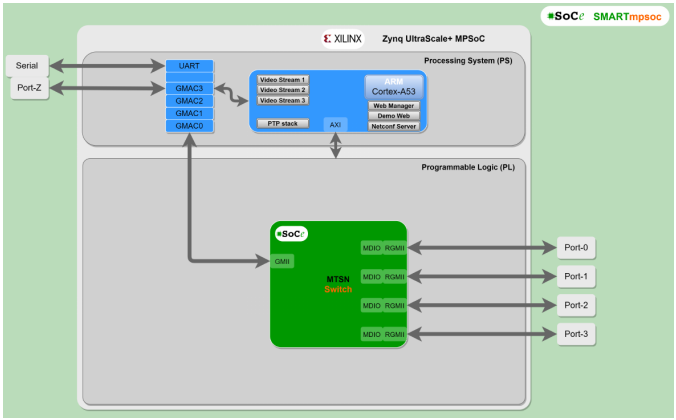

Figure 10. Diagram of the MTSN Kit. From [16]

C. NETCONF

As we have explained, the generated YANG instances containing a new configuration need to be sent and applied to the corresponding bridges. When the CNC completes the generation of the YANG instances that contain the configurations of the bridges, the CNC sends the YANG instances and requests for the bridges to modify their configuration through an RPC. To do so, it is necessary to use a *copy-config* RPC. This RPC contains the YANG instance generated previously as the data payload and it is sent to the NETCONF servers. This is done for each bridge that needs to be modified in the network.

In section IV we describe how the CNC is able to establish a connection with the bridges and how the RPCs are sent.

IV. Implementation

In this chapter we explain in detail how the project was completed. The implementation of the CNC itself is explained, covering all the aspects that we introduced in Section III. Next we explain the different software tools that we have used in order to carry out the CNC.

A. Hardware

One of the objectives of this project is to integrate our CNC with real commercial TSN bridges. The devices that we have used are two *SoC-e SMART MPSoC* bridges [9], which are bridges that support the main TSN standards. More specifically, the following are the most important standards implemented by the switches for this project:

1) IEEE Std 802.1AS for the for global clock synchronisation.
2) IEEE Std 802.1Qav which defines the Credit Based Shaper.
3) IEEE Std 802.1Qbv for the TAS that supports scheduled traffic.
4) IEEE Std 802.1Qcc for network management, i.e. NETCONF for managing YANG data.
5) P802.1Qcw for the YANG Scheduled Traffic module specification.

In Figure 9 we see a picture of the board. There are four external TSN ports (from PORT-0 to PORT-3) and one service port (PORT-Z). As the operating system, the bridges run an embedded Linux OS in which they run some default applications. Looking at Figure 10 we can see a diagram of an MTSN Kit. On the bottom the *Programmable Logic* section is displayed and on top is the *Processing System* section. As you can see, in the processing system section there are three applications running, the most important one being the NETCONF server which is responsible for managing all the incoming RPCs from our CNC. Another application that is executed in the processing system is the *Web Manager*, a web based graphical interface used to configure the switch. In fact, the TAS can be configured using this graphical interface, which in this project is what we want to automate.

Apart from the bridges we also make use of a separate machine to run the CNC implemented in this project. During the implementation and verification phase we used a PC. However, the CNC can run on any system capable of running Docker and that has an Ethernet port to connect to the bridges.

B. The CNC

In this section we describe the implementation of the CNC which is responsible for the automatic configuration of the bridges' TAS. As a reminder, TSN does not specify if the CNC is a physical entity or software. In this case we implemented it as software that can be executed on any device capable of running Docker.

Before entering into detail about the implementation, bear in mind that the CNC is written in C, although as explained earlier, we also make use of JSON and XML encoding to format some files.

The following sections explain each one of the processes that the CNC carries out. Although they are explained separately, the CNC executes all of them in series, meaning that it forms part of a single executable.

*1) **Parsing of the Configuration File**:* As explained previously in Section III-A, the configuration file is formatted using JSON, so the first step is to be able to read this type of format and extract all the data from it. To this end, we use a light-weight library written in *C* [20] that enables the parsing of any JSON file making use of a few basic functions. The parsing of the JSON file consists in reading each json

node of the JSON file. A json node is a generic container of elements inside a JSON stream. It can contain fundamental types (integers, booleans, floating point numbers or strings) and complex types (arrays and objects). Thus, the functions from the JSON library provide us with a way to read the nodes from the configuration file.

First we open the configuration file as a readable file with the use of basic functions found in the standard input-output C library *stdio.h*. Once the file is open we use the following function provided by the JSON library:

```
json *json_parse(const char *);
```

With this function all the json nodes of the *json* file are stored in a variable of type *json*, which is then used to go through all the values specified in the configuration file using other functions from the library. Here are some examples of functions that the library provides:

- *json json_next(const json)*: Points to the child node of the one specified as a parameter.
- *json json_item(const json , size_t)*: Locates a child node by offset.
- *double json_number(const json)*: Returns the value of the current node as a numeric value.

So, with the use of these functions we are able to temporarily store the information inside the program to then make use of it in the next step, which is the generation of the YANG instance.

2) ***Generation of the YANG Instance****:* Once the parsing of the configuration file is complete, the next step is to make use of the acquired data to build the YANG instance. As said previously, the YANG instance is formatted using XML encoding and it follows the data model specified by the YANG scheduled traffic module (P802.1Qcw).

As this data model is already specified, the structure of the XML file is exactly the same each time a new configuration is applied except for the values that have been specified in the configuration file. This makes the procedure of printing the XML file a straightforward process by utilizing once again the functions from the standard input-output library from *C* (*stdio.h*).

At the end of this process, the expected result is to have an XML file generated for each one of the bridges that have to be configured. All these files are stored in a separate folder in the working directory of the CNC so the next process can access the XML files. Moreover, this allows the user to access these generated XML files for validation purposes and other possible reasons.

3) ***Transmission of the RPCs****:* The next step that the CNC has to perform is the transmission of the RPCs to the bridges.

As we have already explained, each bridge executes its own Netconf server. These servers respond to any NETCONF messages sent by the client, which in this case is the CNC. In order to transmit the RPC messages, the CNC must first establish a connection with the bridges' servers. In addition, since we want our CNC to configure the network in an autonomous way, this connection has to be established without user supervision.

The connection is established through Secure Shell (SSH) [8], a cryptographic network protocol for operating network services. The connections are managed automatically by the CNC with *libnetconf2* [4]. This library provides all the necessary functions so as to achieve the SSH connections and the transmission of the RPCs to the NETCONF servers in the bridges.

As explained in the previous section, the YANG instances previously generated are stored in a folder inside the working directory of the CNC. To apply this configuration we need to use an RPC which copies an entire configuration to the desired datastore.

As we explain in Section II, NETCONF possesses three default datastores, namely startup, candidate and running. In this project we configure the NETCONF servers in the bridges to only use the running datastore. Therefore, whenever the CNC transmits a new configuration to a bridge, this new configuration is directly applied to the running datastore.

The RPC that we have to use to apply a new configuration to the datastore is the *copy-config* RPC. Using a function provided by *libnetconf2* we build the RPC by adding the generated XML file to it and specifying the datastore in which we want to apply the configuration. After that, we use a second function also from *libnetconf2* to send the RPC.

These functions explained above perform checks in order to see if the RCPs were generated and sent correctly. Apart from these checks, once to *copy-config* RPC has been successfully applied, we also perform another RPC on the NETCONF servers to acquire the current configuration using the *get-config* RPC. The result of this RCP, as a response from the NETCONF server, is another YANG instance containing the information of the current configuration. In this way, we can then check if the bridges have actually been updated with the desired configuration.

After this process has been executed for each one of the specified bridges in the configuration file, the automatic configuration of the TAS in the bridges has finished.

C. CMake

In this project we also made use of CMake [19]. CMake is an open-source set of tools designed to build, test and package software. It is used to control the software compilation process using simple platform and compiler independent configuration files.

As mentioned previously in this document, to build the CNC there are a number of software dependencies that need to be installed. When it comes to compiling source code that has third-party dependencies, the instructions to give to the compiler can get very extensive and incomprehensible. Thus, with the help of CMake we ease the compilation process of the CNC and organize the project's source code.

Any project based on CMake always contains the *CMakeLists.txt* file, which describes how the project is structured, the list of source files to compile and what CMake should generate out of it. Cmake reads the *CMakeLists.txt* file and produces the desired output.

In our case we have one major dependency that needs to be specified in the *CMakeLists.txt* file, that is Libnetconf2.

Luckily, most of the complex open-source libraries provide a file that instructs CMake where to find necessary files that are installed in the host machine in order to be able to compile the solution. By utilizing this file together with the CMake function `find_package()`, we instruct CMake to find the required packages installed on the system to be able to build the CNC's executable. This same procedure is used with other dependencies like *libssh* [5] (used to establish an SSH connection between two devices) and *libyang* [6] (used to parse and validate YANG formatted files). Naturally, we also need to specify our source code in order for it to be compiled.

Once the *CMakeLists.txt* file is ready, we can run it and generate the compiler instructions for it to build the program as specified in the instructions. Once the building process has finished we end up with a binary file which corresponds to the CNC.

D. Docker

In Section II-E we explained what Docker is and why we use it in this project. In this section we explain how we created the image that contains the CNC and all of its dependencies. Further on we also explain how to make use of the Docker image turning it into a container and how to interact with it to execute the CNC.

Docker uses a file called *Dockerfile* to specify all the instructions to build an image. It always begins with a `FROM` instruction, which specifies the parent image form which we are building. In this case, as our solution was built for a Linux environment, we use Ubuntu 18.04 as the base image. After specifying the base image we can start installing the packages needed for the system in order for it to be able to install all the dependencies and run the CNC. As the image is created from an Ubuntu image as a base image, we can simply instruct Docker to run `apt-get`, which is a command-line tool that helps in handling packages in Linux. This instruction is given to Docker with the `RUN` command as shown below:

```
RUN apt-get update && \
    apt-get dist-upgrade -y && \
    apt-get install -y \
    systemd \
    cmake \
    git \
    curl \
    libpcre3 \
    libpcre3-dev \
    zlib1g-dev \
    libssl-dev \
    build-essential \
```

These are the packages needed to be able to install the dependencies explained previously, that is *libnetconf2* and *libssh* among others. To install these dependencies we first need to copy the files from the host computer into the new image with Docker's `COPY` command. In the folder where the Dockerfile is stored we also place a folder called *libs* where we store all the source files of the dependencies. In this way, when we use the `COPY` command we can instruct it to copy the whole folder into the image like so:

```
COPY ./libs /libs/
```

The first parameter in the command above indicates where the source files are, that is, the folder in the host computer running Docker. The second command indicates where the files will be copied inside the Docker image once it has been built.

After that, the next step is to instruct Docker to compile the source code form the dependency libraries that have just been copied. For each one of the libraries we run the following instruction:

```
# Install libyang
RUN mkdir ./libs/libyang/build && \
    cd ./libs/libyang/build && \
    cmake .. && \
    make && \
    make install
```

Finally, we proceed to build the CNC. In the *libs* folder we also store the source code for our CNC application. Thus, we can also install the application in a similar way to how we instruct Docker to install the dependencies:

```
# Install cnc
RUN mkdir ./libs/cnc/build && \
    cd ./libs/cnc/build && \
    cmake .. && \
    cmake --build . --target all
```

At this point the *Dockerfile* has all the instructions necessary to build the image. Once we run the `build` command in a terminal the image will start to be created. Note that this command has to be executed in the directory where the *Dockerfile* is stored:

```
docker build --rm -t aservera/cnc:latest .
```

Once this process has finished the image is ready to use. As said previously in Section II-E, a Docker image becomes a container at runtime. So, the next step is to execute the image and turn it in a running container.

Going back to Figure 7 we can see that the Docker container has a *Shared Volume*. This volume is a file directory that can be accessed both by the container and the host machine running the Docker container. In this way, the configuration file can be stored in the PC and the user can modify it outside Docker with any text editor and also the CNC application can access from inside the Docker container.

In order to be able to execute the Docker container with the *Shared Volume* shown in Figure 7 it is necessary to use *Docker Compose*, a tool provided by Docker for defining and running multi-container Docker applications and managing the application's services. Basically, this tool is used by defining some configuration parameters in a file called *docker-compose.yml*. We do not go into much detail about how this tool works, but we need to note that using this tool we can

define the shared volume and execute the container. This is done running the following command in the directory where the *docker-compose.yml* files is stored:

```
docker-compose up -d
```

Once this is executed, the CNC Docker container will be up and running on a background process inside the host machine. At this point the user can modify the configuration file and then execute the CNC by using these two commands:

```
docker exec -it cnc ./bin/bash
./libs/cnc/build/cnc
```

The first command opens a terminal in the Docker container and the second one executes the CNC itself.

The source code for the project and the Docker implementation can be found in our git repository [2].

V. Verification

In this section we describe the processes that we have carried out during the project in order to verify that the different parts of the CNC are properly implemented. This section is divided into several subsections, each one referring to an independent part that can be independently evaluated. Furthermore, the overall automatic configuration as a whole is also verified.

A. Generation of the YANG Instance

This section refers to the first process that takes place during the execution of the CNC. This first part is comprised of the parsing of the configuration file and the later generation of the YANG instance.

In order to verify the correct execution of these two parts we only need to check if the parameters introduced in the configuration file are then correctly translated into each generated YANG instance for each bridge. To do this verification, we have included instrumentation code that prints the values of the parameters on the terminal where the configuration is executed. At the same time, this allows the user to check if the parsing of the configuration file has been carried out correctly and that the parameters introduced are the desired ones. Below we explain a simple verification example that is carried out on a bridge.

In Figure 8, referenced previously to explain the configuration file, we show the configuration file used to perform the first test. As we see, it is a simple configuration containing only one bridge (switch 65) with parameters for two ports (PORT_1 and PORT_2. Once we execute the CNC inside Docker we get the following print out in the terminal (Figure 11).

As we can see, the values of the parameters shown in Figure 10 correspond with the values indicated in the configuration file. Also, the CNC generates an XML file containing the YANG instance of the configuration. This file is stored in the Docker container under *cnc/generated-configs/* and it is the one used to build the *copy-config* RPC that is later sent to the corresponding bridge.

On top of the instrumentation prints and the generated XML files, another way that the generation of the YANG instance

```
root@d854feedec3f:/libs/cnc/build# ./cnc
Switch:65
        Ip: 192.168.4.65
        Port Number: PORT_1
                Period: 200000
                Gate States: 10000000
                Period: 100000
                Gate States: 1000000
                Period: 100000
                Gate States: 100000
                Period: 100000
                Gate States: 10000
                Period: 100000
                Gate States: 1000
                Period: 100000
                Gate States: 100
                Period: 100000
                Gate States: 10
                Period: 100000
                Gate States: 1
        Port Number: PORT_2
                Period: 100000
                Gate States: 10000000
                Period: 100000
                Gate States: 1000000
                Period: 100000
                Gate States: 100000
XML instance generated correctly: 65.xml
```

Figure 11. Print out generated by the CNC after parsing the configuration file.

is checked is using *yanglint*, one of the dependency libraries for the CNC which enables the parsing of YANG data models and also validates them. So, when a new YANG instance is uploaded to the bridges' servers, *yanglint* automatically checks if the instance is correct after parsing it. In this way, bridges can discard erroneous configurations.

B. Transmission of the RPCs

Once the generation of the YANG instance is verified we can proceed to the verification of the next step, that is the transmission of the RPC commands to the bridges.

As explained in previous sections, the CNC first establishes an SSH connection with the bridges that need to be configured. After that, the CNC sends two RPC commands: *copy-config* and *get-config*. The verification for this process can be done by visualising the output of the CNC in the terminal where it is executed. In both the SSH connection and the transmissions of the RPCs there is an output on the terminal. In fact, the SSH connection in some cases needs the user to enter the password to establish a secure connection. Therefore, the terminal will provide information on whether the SSH connection has been successful and, when it comes to the RPCs, the *get-config* command will finish with a print out of the current configuration for the bridge after applying it previously. By checking the output the user can manually check if the configuration was carried out correctly.

To verify this process we use the same input example as the previous section (Figure 8). Executing the CNC application prompts the user to insert the SSH password. After introducing it, the CNC sends the *copy-config* RPC and later the *get-config* RPC. Once the last RPC is completed, the response with the configuration is printed on the terminal. As expected, the configuration parameters received in the *get-config* operation correspond to the ones specified in the configuration file.

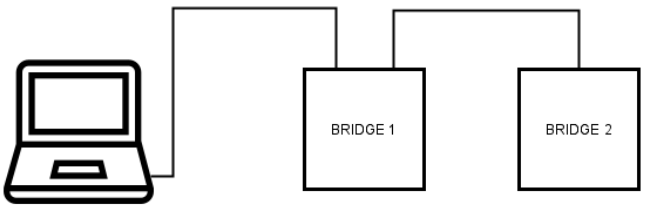

Figure 12. Diagram of the connections for the testing of the configuration of multiple bridges.

C. Configuration for Multiple Bridges

Until now, we have verified the correct operation of both the generation of the YANG instance and the transmission of the RPC to the bridges. To do so we have only used one bridge. Thus, in this section we carry out a verification of the proper configuration of bridges using a scenario with two bridges to ensure that the whole configuration process is also performed correctly with more than one bridge.

To perform this test we connect two bridges together, as shown in Figure 12, by plugging an Ethernet cable between them using one of the ports. Then, one of the bridges is connected to the host machine where the CNC application is executed. This way, the CNC will be able to establish a connection between all of the necessary bridges.

The execution of the CNC application is performed as in the previous tests. The only difference is that the configuration file now has two bridges specified. The CNC is able to parse the parameters of each bridge independently and then apply the generated configurations.

Once the CNC has started we can see the instrumentation prints that show that the configuration parameters have been parsed correctly. After that, the CNC application prompts the user with the SSH password for each bridge that is being configured. If the connection is successful then the RPCs are transmitted to each bridge and the final configuration is also shown in the terminal to confirm the correct execution of the automatic configuration.

D. Configuration of the output ports of bridges

So far we have verified the automatic configuration of the TSN bridges. Nonetheless, in order to verify the correct operation of the system we next check whether the configuration loaded in the bridges is applied in the output ports. To do that, we inject traffic in the network and we check how the traffic is shaped by the bridges.

To this aim we use *Wireshark* [10], a network protocol analyzer that listens to any traffic that is received through the specified network interface on the device that you are running it on. In this case, we execute Wireshark in the same PC as the CNC and we listen to one of the ports of the bridge that we want to configure.

In order to carry out this verification, we must first decide on a proper configuration for the port that we want to analyse using Wireshark. Once the configuration is loaded, we execute a frame generator application that transmits traffic through the port that we want to analyse. Finally, this traffic is received by the PC through the interface that Wireshark is supervising, which allows us to capture the traffic to analyse it.

First, we perform a basic test that consists of checking if the frames sent by the bridge are transmitted or not depending on their priority. We must recall that TAS specifies which queue of the output port of a bridge is open and which is closed at each moment of time. We must also note that frames with a specific priority are buffered in the queue with said priority prior to their transmission. Therefore, if a queue with priority i is open, frames with priority i are transmitted, while if queue i is closed, frames with priority i are buffered in the queue. For instance, we apply the following configuration for port *PORT_0*:

```
"switch": "TEST_64",
"ip": "192.168.4.64",
"port_list": [
    {
        "port_number": "PORT_0",
        "values":[
            [500000, 00000001],
            [500000, 00000000]
    }
[...]
```

Given this configuration, *PORT_0* of the corresponding bridge can only send frames of priority *0* as indicated in the GCL. After applying the configuration, we connect to the bridge via SSH and we generate Ethernet frames with a priority different than *0*. In this case we use a priority of *5* and, as a result, Wireshark does not register any received frames, meaning that the bridge could not send any of them. We also performed the same test but sending frames with priority *0* and, unlike the previous step, Wireshark registers frames received on the PC, which is the expected result.

Next we test another configuration to verify if the time periods of the slots match with the applied configuration. To this end, we apply the following configuration:

```
[...]
{
    "port_number": "PORT_3",
    "values":[
        [1000000, 00100001],
        [1000000, 00000000],
        [1000000, 00000000],
        [1000000, 00000000],
        [1000000, 00000000],
        [1000000, 00000000],
        [1000000, 00000000],
        [1000000, 00000000]
    ]
}
[...]
```

As we can see, *PORT_3* has 8 different slots of 1.000.000 ns each, but only the first one has open gates. More specifically, the gates that are open are for priority 5 and 0. This means that, when transmitting packets of priority 5 or 0, TAS only allows

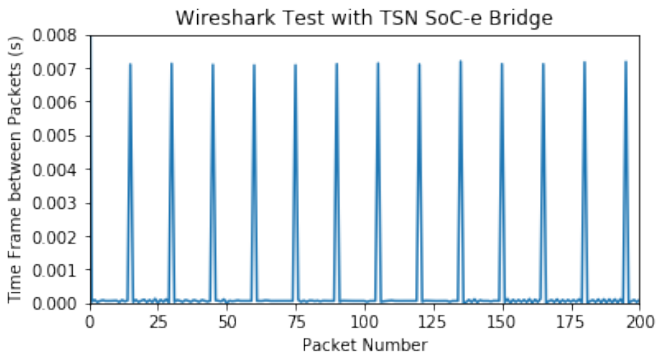

Figure 13. Plot of the Wireshark test performed on a TSN SoC-e bridge.

the transmission of these packets when the first slot is active. Thus, the bridge can only transmit frames with priority 5 or 0 for 0.001 seconds (1.000.000 ns) and then the transmission stops until the rest of the slots have finished, which equals 0.007 s (7 x 0.001 s). Once all the slots are executed, the bridge starts again with the first one, transmitting frames with priority 5 and 0 again. We must note that in our next experiment we do not transmit any frames with priority 0, even though the gate is open. The reason is that the communication between the bridge and the PC requires the transmission of frames with priority 0. Thus, if we configure bridges to never allow the transmission of these frames, the communication is lost.

Now we generate Ethernet packets with priority 5 in the bridge with the configuration previously mentioned. At the same time we run Wireshark and register the packets that the bridge sends. Wireshark allows us to see the time elapsed between the reception of frames. This time is also known as *step*. Using this output we can plot the *step* value for each transmitted packet, which we can see in Figure 13. On the plot we have the *step* value on the *Y* axis and the packet number on the *X* axis. We can see periodic peaks with values close to 0.007 seconds. These peaks represent the total time in which the packets are not being transmitted, which matches the slots with the closed gates. The rest of the time frames are being transmitted in a continuous manner, which is why the time elapsed between frames in the rest of the plot is low.

VI. Conclusion

The aim of the fourth industrial revolution, known as Industry 4.0, is the development of intelligent processes capable of planning, predicting, controlling and producing in an autonomous manner. Industrial network communications play an important role in the progress of this industry as they require RT guarantees and the ability to reconfigure the network to utilise its limited resources efficiently.

With the TSN new set of standards it is possible to provide a network with RT guarantees and adaptability. On the one hand, TSN provides hard RT guarantees to the network by means of TAS, a standard that allows to configure bridges so they can forward scheduled traffic. On the other hand, thanks to TSN's centralized online configuration and control architecture we are able to reconfigure the network. Specifically, this network architecture proposes the use of a central element named CNC, which has a view of the whole network, allowing it to make complex configuration decisions in a reasonable time.

In this document we have described the implementation and verification of a CNC, the function of which is to automatically configure the TAS of TSN bridges. This automatic configuration eases the deployment of new large networks by automatically configuring all the TSN devices that are provided with scheduled traffic enhancements (TAS). Moreover, it enables the reconfiguration of the network in order to adapt to changes in the production chain and make a better use of the network's resources.

This implementation consists of a C program that executes a series of processes in order to automatically configure the TAS of a set of TSN bridges. The first process is the parsing of a configuration file specified by the user, in which the configuration parameters of the bridges are specified. The second process is responsible for the generation of the YANG instances that contain the configuration data for each TSN bridge. Finally, the last process establishes a connection with the bridges and sends the YANG instances in the form of an RPC so that the NETCONF servers in the bridges can apply the new configuration.

At the beginning of this work we established three objectives. In order to meet our first objective, described in Section I-B, this implementation adheres to the current specifications of the TSN standards. Furthermore, it implements the currently proposed specification of the YANG data model for the TAS as defined in the ongoing standardization effort P802.1Qcw from IEEE.

Additionally, to satisfy our second objective specified in Section I-B, the implementation has been developed to require the least supervision possible from the user, as the intention is to configure the TAS of a set of TSN bridges in an automatic manner. We have also focused on enabling the CNC to integrate with future developments by creating a Docker container in which the CNC is executed. This feature eases the deployment process of the CNC and also eases the process of extending or modifying the functionalities of the CNC.

In order to meet our third objective explained in Section I-B, we carry out the verification of the CNC. Specifically, we have first verified each one of its processes separately: the reading of the configuration file, the generation of the YANG instances and the transmissions of the RPCs to the NETCONF servers. On top of that, we have verified the correct integration of all the previous mechanisms and the correctness of our application. We have done this by checking that the configuration loaded in the bridges is applied in the output ports of the bridges by injecting traffic in the network and checking how the traffic is shaped by the bridges. This has allowed us to see that the configuration provided to the CNC was properly deployed in all the bridges.

Finally, it is important to note that both the implementation and the verification process have been carried out using commercial switches that have all the necessary TSN standards to be able to use TAS. This ensured that the CNC is ready to be deployed in a real TSN network.

VII. Future Work

In this final section we make mention of possible future tasks that could be carried out to continue with this project in order to enhance it and provide more uses for it.

First, we can improve the automatic SSH connections to the bridges in order to require even less supervision from the user. In this implementation, the connection is established using a password as the authentication process. This could be improved by using authentication by *public key*. Public key authentication is a way of logging into an SSH account using a cryptographic key rather than a password. This way, the CNC could establish an SSH connection with each bridge without having to ask the user to introduce a password. However, it is still necessary to generate the public key for each bridge. This can be a tedious process if the network is very large, but it would only be necessary once for each bridge which is still a better option than introducing a password every time you want to establish a connection with a bridge.

The majority of time-critical systems require a certain degree of reliability and fault-tolerance. In this aspect, a future work could be to carry out the replication of the CNC and implementing mechanisms for the synchronization and consistency between them.

References

[1] Docker's web page. https://www.docker.com/.

[2] Git repository of the project. https://github.com/andreuservera/cnc.

[3] JSON (JavaScript Object Notation) a Lightweight Data-Interchange Format. https://www.json.org/json-en.html.

[4] Libnetconf2 – The NETCONF protocol library. https://github.com/CESNET/libnetconf2.

[5] Libssh - The SSH Library. https://www.libssh.org/.

[6] Libyang - a YANG parser and toolkit. https://github.com/CESNET/libyang.

[7] P802.1Qcw – YANG Data Models for Scheduled Traffic, Frame Preemption, and Per-Stream Filtering and Policing.

[8] SSH Secure Shell Home Page. https://www.ssh.com/academy/ssh.

[9] System-on-Chip engineering S.L. SoC-e MTSN Kit: A comprehensive multiport tsn setup. https://soc-e.com/mtsn-kit-a-comprehensive-multiport-tsn-setup/.

[10] Wireshark, a network protocol analyzer. https://www.wireshark.org/.

[11] IEEE Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams. *IEEE Std 802.1Qav-2009 (Amendment to IEEE Std 802.1Q-2005)*, pages C1–72, 2010.

[12] IEEE Standard for Local and metropolitan area networks–Virtual Bridged Local Area Networks Amendment 14: Stream Reservation Protocol (SRP). *IEEE Std 802.1Qat-2010 (Revision of IEEE Std 802.1Q-2005)*, pages 1–119, 2010.

[13] IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic. *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)*, pages 1–57, 2016.

[14] IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks – Amendment 31: Stream Reservation Protocol (SRP) Enhancements and Performance Improvements. *IEEE Std 802.1Qcc-2018 (Amendment to IEEE Std 802.1Q-2018 as amended by IEEE Std 802.1Qcp-2018)*, pages 1–208, 2018.

[15] 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.

[16] System-on-Chip engineering S.L. SoC-e MTSN Kit User Guide. v20.1. 2020.

[17] I. Álvarez, A. Ballesteros, M. Barranco, D. Gessner, S. Djerasevic, and J. Proenza. Fault Tolerance in Highly Reliable Ethernet-Based Industrial Systems. *Proceedings of the IEEE*, 107(6):977–1010, June 2019.

[18] IEEE 802.1. Time-Sensitive Networking (TSN) Task Group.

[19] Open-Source. Cmake. https://cmake.org/overview/.

[20] D. Ranieri. Lightweight JSON library for C. https://github.com/davranfor/c/tree/master/json.

[21] R. Salazar, T. Godfrey, N. Finn, C. Powell, B. Rolfe, and M. Seewald. Utility applications of time sensitive networking white paper. *Utility Applications of Time Sensitive Networking White Paper*, pages 1–19, 2019.

[22] S. Samii and H. Zinner. Level 5 by Layer 2: Time-Sensitive Networking for Autonomous Vehicles. *IEEE Communications Standards Magazine*, 2(2):62–68, JUNE 2018.

[23] B. M. Wilamowski and J. D. Irwin. *Industrial Communication Systems (The Industrial Electronics Handbook)*. Second edi edition, 2011.

[24] M. Wollschlaeger, T. Sauter, and J. Jasperneite. The Future of Industrial Communication: Automation Networks in the Era of the Internet of Things and Industry 4.0. *IEEE Industrial Electronics Magazine*, 11(1):17–27, March 2017.