



Escola d'Enginyeria de Telecomunicació i
Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

BACHELOR DEGREE THESIS

TITLE: Development of an SDN control plane for Time-Sensitive Networking (TSN) endpoints

DEGREE: Double Bachelor Degree on Network Engineering and Telecommunications Systems Engineering

AUTHOR: Jordi Cros Mompart

DIRECTORS: David Rincon Rivera
Anna Agustí Torra

DATE: 21 June 2021

Títol: Desenvolupament d'un pla de control SDN per terminals de Time-Sensitive Networking (TSN).

Autor: Jordi Cros Mompart

Director: David Rincón Rivera
Anna Agustí Torra

Data: 21 Juny 2021

Resum

Dins el món industrial i d'automatització, com ara indústria 4.0 o busos de comunicació de vehicles, l'extensió d'Ethernet amb capacitats relacionades amb la sincronia i la qualitat de servei (reserves d'ample de banda o garanties de retard) es defineix a una sèrie d'estàndards que descriuen com es podrien unificar les comunicacions deterministes amb les best-effort dins d'una mateixa xarxa. Aquesta extensió a l'Ethernet convencional s'anomena Time-Sensitive Networking (TSN) i ha estat creada pel TSN task group del comitè IEEE 802.1.

Per una altra banda, la gestió del pla de control de les xarxes locals (LAN) evoluciona cap al concepte Software Defined Networking (SDN). En aquesta arquitectura, un controlador centralitzat s'encarrega de configurar els nodes de xarxa per tal de satisfer els requeriments dels usuaris. Les capacitats lògiques del controlador SDN venen a partir d'implementacions de codi obert i propietàries, permetent una gran flexibilitat en la gestió de la xarxa.

Finalment, OPC-UA és una arquitectura que permet la interconnexió de qualsevol tipus de dispositiu industrial o administratiu, i que obre la porta a l'automatització i digitalització dels processos industrials. Ofereix molts graus de llibertat, ja que tan sols defineix interfícies entre elements, la seva estructura de dades i diferents modalitats de transmissió de dades.

L'objectiu d'aquest projecte és desenvolupar un prototip de l'arquitectura de gestió de terminals TSN basada en SDN, tal com es defineix a l'estàndard IEEE 802.1Qcc. Aquest element, la Centralized User Configuration (CUC), es comunicarà a través de OPC-UA amb els dispositius que vulguin enviar fluxos TSN per proporcionar-los la configuració necessària. També es comunicarà amb el controlador de xarxa, per tal d'establir correctament el flux TSN extrem a extrem (des del terminal emissor fins al receptor, passant pels commutadors TSN). Addicionalment, es dissenyaran els dispositius que consumiran el trànsit time-sensitive, que necessiten aplicar la configuració que els arriba de manera automatitzada.

S'ha aconseguit implementar la CUC i els terminals de l'arquitectura seguint l'arquitectura SDN. A més, s'ha deixat preparada la interfície amb el component que falta, la Centralized Network Configuration (CNC).

Title: Development of an SDN control plane for Time-Sensitive Networking (TSN) endpoints

Author: Jordi Cros Mompart

Directors: David Rincón Rivera
Anna Agustí Torra

Date: 06/21/2021

Overview

In the industrial and automation world, such as Industry 4.0 or vehicle communication buses, an Ethernet extension that provides timing and quality of service (bandwidth reservations or delay guarantees) has been defined in a series of standards whose aim is to describe how deterministic, time-sensitive and best-effort communications can use the same network. This Ethernet extension is called *Time-Sensitive Networking* (TSN) and it has been defined by the TSN Task Group of the IEEE 802.1 committee.

The field of the management of the control plane of Local Area Networks (LANs) is evolving towards the *Software Defined Networking* (SDN) concept, based on a centralized network controller. This controller configures all of the network nodes to satisfy user requirements. The capabilities in SDN controllers come from both open-source and proprietary implementations. Due to this, the SDN controller is a flexible element in terms of network management.

Finally, OPC-UA is an architecture that interconnects any kind of industrial or administrative device, with the aim to standardize and automate industrial communications in the application layer. It achieves this goal by only defining the interfaces between elements, its data structure and different communication modes, which allows it to be versatile.

The goal of this thesis is to build a prototype part of the TSN endpoint management architecture, based on SDN as defined in the IEEE 802.1Qcc standard. This element, the Centralized User Configuration (CUC), will establish a session using OPC-UA with the endpoints that want to take part in a TSN flow, so that they can be properly configured, and it will also communicate with the network controller, so that the TSN flow is established end-to-end (from the talker endpoint to the listener endpoint, passing through the TSN switches). Additionally, this thesis includes the design of the endpoints that will generate and listen to the TSN flow, which will be automatically configured.

The results of this thesis include the development of the CUC and the endpoints following the SDN concept. In addition, the interface with the Centralized Network Configuration is prepared for further integrations.

Acknowledgements

Gràcies als meus pares per la vida plena d'oportunitats
que m'han donat, el recolzament i la seva companyia.

Especial agraïment als meus tutors David Rincón i Anna Agustí.
A banda del seguiment, valoro molt el seu rigor en la docència.

A Gabriel, mi compañero de laboratorio, por su ayuda
e interesantísimas aportaciones a este proyecto.

Finally, my sincere love and gratitude to Julie Kim, since
nothing would be the same without her by my side.

*"No one wants to learn by mistakes, but we cannot learn enough from
successes to go beyond the state of the art."* – Henry Petroski

TABLE OF CONTENTS

INTRODUCTION	1
CHAPTER 1. DATA LINK LAYER AND IEEE STANDARDS	5
1.1. Data Link layer	5
1.1.1 IEEE 802.1Q - VLAN	6
1.1.2 IEEE 1588/ 802.1AS – Precision Time Protocol	7
1.2. Time Sensitive Network standards	10
1.2.1 IEEE 802.Qbv – Enhancements for Scheduled Traffic	10
1.2.2 IEEE 802.Qav – Credit Based Shaper	13
1.2.3 Standards in process	14
CHAPTER 2. NETWORK MANAGEMENT AND OPC-UA.....	16
2.1 NETCONF, RESTCONF, YANG and Configuration Data Models	16
2.1.1 Information models and data models	16
2.1.2 YANG.....	18
2.1.3 NETCONF	22
2.1.4 RESTCONF	25
2.2 OPC-UA.....	26
2.2.1 Clients and Servers	29
CHAPTER 3. IEEE 802.1QCC	32
3.1 IEEE 802.1Qcc clause 46 – Time Sensitive Networking Configuration.....	32
3.1.1 UNI Integration	33
3.1.2 Stream Transformation	36
3.1.3 Example workflow.....	37
CHAPTER 4. SOFTWARE AND HARDWARE FOR TSN ENDPOINTS	40
4.1 Kalycito – Linux and TSN using open62541	41
4.2 Linux, kernel and Linux Network Stack	41
4.2.1 General system architecture.....	42
4.2.2 Linux kernel	43
4.2.3 Linux Network Stack	44
4.3 Intel i210 features related to TSN.....	45
CHAPTER 5. ARCHITECTURE DESIGN	46
5.1 Statement of the objectives.....	46
5.2 Centralized User Configuration.....	47
5.2.1 OPC-UA Client.....	48
5.2.2 Logic Unit Center	48
5.2.3 RESTCONF Client.....	57

5.3	Endpoints	58
5.3.1	Address Space definition	59
5.3.2	Interface configuration	62
CHAPTER 6.	IMPLEMENTATION	63
6.1	LAN	63
6.2	Time synchronization between Endpoints and the TSN switch	63
6.2.1	Installation and use of <i>linuxptp</i>	64
6.2.2	Interconnection with the TSN switches	66
6.3	Details on OPC-UA Address Spaces	67
6.4	Endpoint configuration	68
6.5	Prototype integration and set up	70
6.5.1	System requirements	70
6.5.2	Set up of the components	70
CHAPTER 7.	TESTS AND RESULTS	76
7.1	Basic scheduling test and evaluation with OPC-UA Pub/Sub implementation	76
7.1.1	Set up and configuration properties	76
7.1.2	Results	78
7.1.3	Test conclusions	87
7.2	Scheduling performance test with TSN traffic generated by iperf	87
7.2.1	Set up and configuration properties	87
7.2.2	Results	89
7.2.3	Test conclusions	94
CHAPTER 8.	CONCLUSIONS AND FUTURE LINES	96
8.1	Conclusions	96
8.2	Future lines of development	97
8.3	Sustainability considerations	98
REFERENCES		99
GLOSSARY		105
ANNEX A.	IEEE AND TSN STANDARDS	108
A.1	IEEE 802.1d – Spanning Tree Protocol	108
A.2	IEEE 802.1aq – Shortest Path Bridging and Shortest Path First	108
A.3	IEEE 802.1ab – Link Layer Discovery Protocol	109
A.4	IEEE 802.1Qat - Stream Reservation Protocol	110
A.5	IEEE 802.1Qch – Cyclic Queuing and Forwarding	111

A.6	IEEE 802.1Qci – Per Stream Filtering and Policing.....	112
A.7	IEEE 802.1Qbu / 802.3br – Frame Preemption.....	112
A.8	IEEE 802.1Qcr – Asynchronous Traffic Shaping.....	112
A.9	IEEE 802.1Qca – Path Control and Reservation.....	113
A.10	IEEE 802.1CB – Frame Replication and Elimination for Reliability	113
A.11	IEEE 802.1CM – Time Sensitive Networking for Fronthaul	115
A.12	IEEE 802.1AX – Link Aggregation.....	115
ANNEX B. UNI YANG MODULE		116
ANNEX C. PROJECT DEVELOPER’S GUIDE.....		120
C.1	Project Structure.....	120
C.2	CUC	121
C.2.1	OPC-UA Client.....	122
C.2.2	Logic Unit Center	123
C.2.3	RESTCONF Client.....	124
C.3	Endpoints	124
C.3.1	Talker.....	124
C.3.2	Listener	126
ANNEX D. ENDPOINT CONFIGURATION EXAMPLE		128
D.1	iptables	128
D.2	tc qdisc	129
D.2.1	taprio.....	129
D.2.2	etf.....	131
D.2.3	cbs	132
ANNEX REFERENCES		133

LIST OF FIGURES AND TABLES

- Fig. 0.1** Fully centralized architecture applies to an SDN network
- Fig. 1.1** Frame format for 802.3 - Ethernet and 802.11 – WiFi [8]
- Fig. 1.2** Ethernet frame format adapted with the VLAN tag [11]
- Fig. 1.3** BMCA clock distribution tree [12]. The blue lines are the logical tree that distributes the timing
- Fig. 1.4** PTP synchronization establishment [13] [14]
- Fig. 1.5** gPTP port-bounded mechanism [13]
- Fig. 1.6** 802.1Qbv Cycle schema [15]
- Fig. 1.7** 802.1Qbv Cycle schema with guard band [15]
- Fig. 1.8** Management of priority queues [16]
- Fig. 1.9** Credit Based Shaper bandwidth limiter [17]
- Fig. 2.1** Module definition in YANG language
- Fig. 2.2** Header definition of the YANG module
- Fig. 2.3** Core definition of the YANG module
- Fig. 2.4** Configuration data definition of the YANG module
- Fig. 2.5** State data definition of the YANG module
- Fig. 2.6** Instance data of the YANG module
- Fig. 2.7** Schema of a data model server
- Fig. 2.8** OPC-UA schematic overview, from [28]
- Fig. 2.9** OPC-UA over a Time Sensitive Network [29]
- Fig. 2.10** Address Space Node classes [32]
- Fig. 2.11** OPC-UA Pub/Sub logical overview [29]
- Fig. 3.1** Fully Centralized model scheme, from [34]
- Fig. 3.2** Stream Transformation in an endpoint device [34]
- Fig. 4.1** Intel i210 Network Card Interface [36]
- Fig. 4.2** Visual representation of a Linux system [38]
- Fig. 4.3** Packet processing in a Linux system [35]
- Fig. 5.1** Scenario of the project
- Fig. 5.2** CUC overview
- Fig. 5.3** Transmission selection choices [43]
- Fig. 5.4** Stream Identification types [44]
- Fig. 5.5** Sequence encoding and decoding method types [45]
- Fig. 5.6** Error types on the Status group [46]
- Fig. 5.7** Time aware shaper example
- Fig. 5.8** Endpoint overview
- Fig. 5.9** Overview of the architecture of the prototype
- Fig. 6.1** LAN created for the prototype
- Fig. 6.2** gPTP.cfg file
- Fig. 6.3** Captures from ptp4l and phc2sys processes
- Fig. 6.4** SoC-e MTSN 802.1AS synchronization tab
- Fig. 6.5** Wireshark capture of gPTP protocol in interface enp2s0
- Fig. 6.6** Address Space instances for Talker and Listener

Fig. 6.7 Listener's config.json

Fig. 6.8 Talker's config.json

Fig. 6.9 Configuration values for the Talker's interface configuration

Fig. 6.10 Subscription triggered and received in Listener.

Fig. 6.11 config.json for the CUC

Fig. 6.12 CUC waking up process

Fig. 6.13 CUC logs regarding the instantiation of the UNI groups.

Fig. 6.14 CUC generated GCL for the Talker.

Fig. 6.15 CUC sending TSN and Subscription config

Fig. 6.16 Equipment used in the testbed. From left to right: Linux PC running the Talker, TSN switch, Linux PC running the Listener

Fig. 7.1 First scenario, with the TSN flow

Fig. 7.2 Second scenario: similar to the first one, with an additional best-effort traffic.

Fig. 7.3 TSN configuration is disabled for the third test case

Fig. 7.4 Wireshark capture showing TSN flow and gPTP protocol

Fig. 7.5 OPC-UA Pub/Sub traffic delay in milliseconds (Y axis), versus Subscribe message sequence number (X axis)

Fig. 7.6 OPC-UA Pub/Sub traffic jitter in milliseconds (Y axis), versus Subscribe message sequence number (X axis)

Fig. 7.7 Network delay for the case of 1MByte every 1s TSN traffic, in microseconds (Y axis), versus Subscribe message sequence number (X axis)

Fig. 7.8 Network jitter for the case of 1MB every 1s TSN traffic in microseconds (Y axis), versus Subscribe message sequence number (X axis)

Fig. 7.9 OPC-UA Pub/Sub traffic delay (after the modification) in milliseconds (Y axis), versus Subscribe message sequence number (X axis)

Fig. 7.10 OPC-UA Pub/Sub traffic jitter (after modification) in milliseconds (Y axis), versus Subscribe message sequence number (X axis)

Fig. 7.11 Scheduling performed with simultaneous OPC-UA Pub/Sub and best-effort traffics

Fig. 7.12 Delay for the TSN flow (100ms interval) when there is best-effort traffic, in milliseconds (Y axis), versus Subscribe sequence number (X axis)

Fig. 7.13 Desynchronization coming from the Node application

Fig. 7.14 Listener receiving nonscheduled traffic

Fig. 7.15 Delay for the non-TSN configuration case and 1MB every 1s, in milliseconds (Y axis), versus Subscribe message sequence number (X axis)

Fig. 7.16 Jitter for the non-TSN configuration case and 1MB every 1s, in milliseconds (Y axis), versus Subscribe message sequence number (X axis)

Fig. 7.17 First scenario with the iperf traffic. Basic scheduling in a large TAS cycle

Fig. 7.18 Second scenario, with the shortest time slot for a given gate state

Fig. 7.19 Third scenario, for testing the shortest scheduler cycle in the prototype endpoints

Fig. 7.20 Basic scheduling performance with iperf traffic

Fig. 7.21 Some Ethernet frames captured with Wireshark

Fig. 7.22 Wireshark capture with a transmission time slot of 100 μ s for the TSN flow

Fig. 7.23 Incorrect plotting of the 100 μ s time slot scheduling

Fig. 7.24 Scheduling performed in 100 μ s time slots for the TSN filter

Fig. 7.25 Bandwidth obtained by the iperf flow when TSN is active.

Fig. 8.1 Global industrial network scenario

Fig. 8.2 Modification on the Talker endpoint design

Fig. A.1 LLDP payload format [5]

Fig. A.2 Stream Reservation Protocol stack

Fig. A.3 FRER sample schema [15]

Fig. C.1 Workflow of the CUC project

Fig. C.2 Set up of the node that will be Published

Fig. C.3 Binding of the Subscription method on the Listener's Address Space

Table 5.1 Example GCL

INTRODUCTION

Industrial and automation communications are facing fundamental changes in their architecture. Current solutions are mostly based on proprietary protocols (see Profibus [1] or CAN [2]), causing a market fragmentation. Using these protocols, modifying the structure in order to incorporate new elements or connections, always requires a high operational cost. Apart from that, these solutions offer bandwidths that are orders of magnitude below from what Ethernet is offering nowadays. *Time-Sensitive Networking* (TSN) is the natural solution to this situation.

The Time-Sensitive Networking Task Group [3] of the IEEE 802.1 committee aims to provide determinism by releasing new standards that apply to the regular Ethernet. These standards, such as IEEE 802.1Qbv or IEEE 802.1Qav, define how to perform traffic scheduling and shaping in order to bound latencies and guarantee a bandwidth for a given stream. Therefore, the *Quality of Service* (QoS) guarantees are satisfied. This means that by using TSN standards, Ethernet can transport time-sensitive flows and best-effort data in the same network. For example, highly time-sensitive industrial devices, such as sensors or automators, could communicate between them using the same physical Ethernet infrastructure used by office computers, facilitating the design of the networks and reducing operational cost and initial investment.

Software Defined Networking (SDN) [4] is a modern approach to network management which is based on the use of software-based controllers to manage the control plane of a network. The controller is a centralized entity that configures every device to enable a new service in the network, so that other network elements will only need to apply the incoming configuration with no extra logic. Currently, there are already several mature implementations of SDN controllers, such as *OpenDaylight* (ODL), *Operating Network Operating System* (ONOS) or *Ryu*. All of them, generally, provide a generic *northbound* interface to set service requirements and a *southbound* interface to configure elements in the network; they all give the possibility to implement extra modules in them to extend their functionalities. Therefore, one of these modules could manage TSN devices, giving the SDN controller the capacity to serve time-sensitive traffic to its users.

If the necessity of change towards more standardization in the industrial and automation world is combined with the SDN network management architecture, an SDN solution for TSN networks comes up naturally.

One of the most recent TSN standards, IEEE 802.1Qcc *Fully Centralized Network* model [5], defines an SDN-inspired management plane. All of the computational logic of the control plane is performed by two centralized components: the *Centralized User Configuration* (CUC) and the *Centralized Network Configuration* (CNC) modules. The CUC polls traffic requirements and device specifications from endpoints. The CNC receives all the information grouped by the CUC and configures all the network nodes to forward the traffic in a deterministic manner. Both centralized entities perform the functionalities expected in an SDN controller.

Some proprietary implementations of a Fully Centralized Network model exist, such as the one from Cisco for its IE-4000 TSN switches [6], providing a private solution including a CUC, a CNC, TSN bridges and endpoints. Nevertheless, it is a proprietary solution, and since its architecture is closed, it cannot be integrated to any SDN controller as an external module.

Industrial frameworks and software solutions tend to be device-specific. In order to correct this trend and to unify industrial communication protocols, the OPC Foundation released in 2008 the *OPC Unified Architecture* (OPC-UA) [7]. OPC-UA is an application layer protocol aimed at providing a platform-independent architecture in which any kind of device can establish a standardized communication with another endpoint in the network, avoiding translations between different device-specific implementations.

The goal of this thesis is to provide an innovative prototype of an open-based software solution that integrates a CUC and the time-sensitive endpoints. Since the implementation will be based on standards and industry trends, the prototype designed in this project can be considered the base of a bigger design, including a CNC and the integration to any existing SDN controller. **Figure 0.1** shows the scope of the thesis' prototype. In addition, it shows the CNC and the CUC, the modules that can be integrated in an SDN controller.

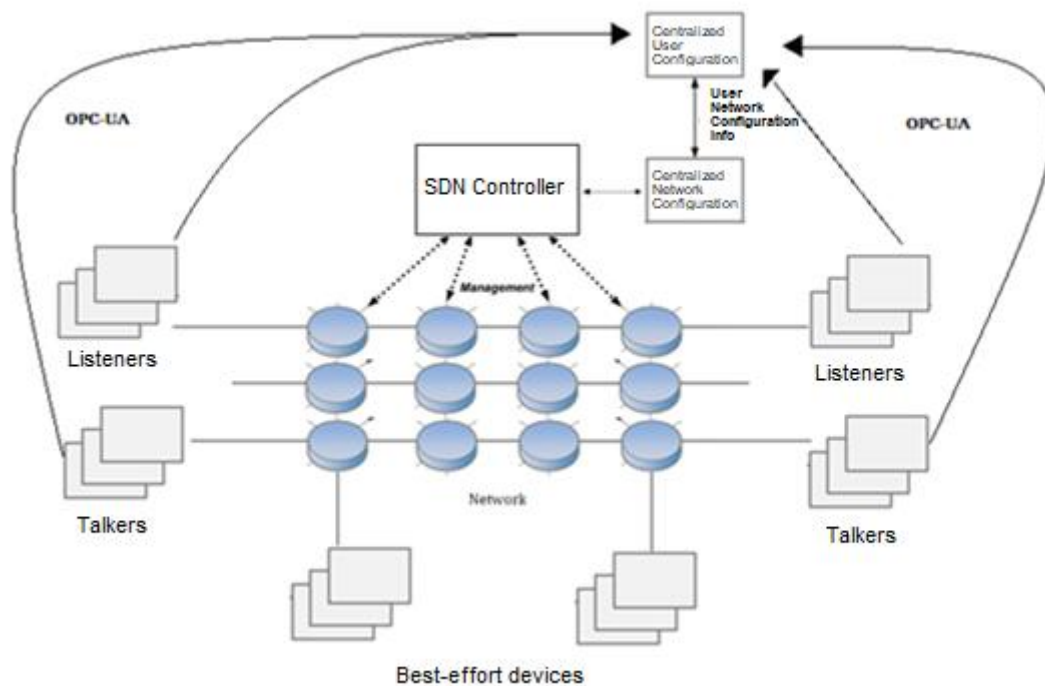


Fig. 0.1 Fully centralized architecture applied to an SDN network.

Additionally, the endpoints will be OPC-UA devices, meaning that configuration process will be based on this protocol. The configuration will allow the endpoints

to take part in the transmission and reception of time-sensitive flows. To the best of our knowledge, our work is the first open-source implementation of the standardized TSN Fully Centralized Network model. This opens the line in the industry to further developments, as there is still a lack of products in the market that aim to integrate a Time Sensitive Network over a Software Defined Networking solution.

Following the objectives stated above, the thesis has been organized in several chapters that can be further divided into three major blocks. The first block aims to contextualize and give all the necessary details to design a proper solution, including the following goals:

1. Chapter 1 focuses on providing the basics of networks, reviewing concepts of the data link layer. This will be followed by a description of the most important standards and protocols required in our design: a time scheduler, a traffic shaper and a time synchronization mechanism in the network.
2. Chapter 2 describes the newest paradigms regarding network management. It introduces concepts such as YANG, NETCONF and RESTCONF, as required by the IEEE 802.1Qcc standard. This section also takes a deeper look into OPC-UA, presenting how the clients and servers communicate and how the data are handled, so that CUC and endpoints are designed appropriately.
3. Chapter 3 focuses on the IEEE 802.1Qcc standard. It also defines important details for the implementation. It includes the definition of the interface between the CUC and the CNC and the workflow that has to be followed by the whole architecture to properly establish a TSN flow in the network.
4. Chapter 4 provides details of the endpoints, because specific hardware is required to generate or receive TSN flows. A detailed description of the Linux system architecture is provided from a networking point of view, to understand how endpoints need to be configured for further automation. Finally, the TSN-capable Intel i210 network interface card (NIC) is reviewed.

The second block focuses on the design and implementation of the prototype, based on all the concepts described in the first block.

5. Chapter 5 starts with a more detailed description of the designed architecture, followed by the specification of both the CUC and the endpoints (Talker and Listener). In-depth guidelines of the interfaces between components and logical insights are provided to refine the design of the implementation.
6. Chapter 6 provides details of the implementation, starting from the deployment of a common Local Area Network (LAN) and a time-synchronized domain among all the devices that take part in it. This is

followed by the instantiation of the OPC-UA endpoints and the details on how their time-sensitive configurations will be performed. At the end of the chapter, a final prototype architecture is presented.

The last block consists of testing the prototype, and discussing the results and the conclusions. Additionally, it proposes future directions that the project may follow with the current development of new TSN standards.

7. Chapter 7 is the presentation of the testbed where the solution has been tested. Practical results of TSN configurations are evaluated to determine the reliability of the configuration computed by the CUC and the configuration of the endpoints.
8. Chapter 8 describes the conclusions obtained throughout the research, development and implementation of the project, and discusses future directions.

The document includes some annexes with additional information. It is recommended to read them to fully understand the software implementation of this project, as well as broaden the context of the project. The annexes are as follows:

1. Annex A includes a list of the standards that are related with Time Sensitive Networking. Other standards are also mentioned because they may interfere with some of the qualities that TSN brings. For example, Spanning Tree Protocol may limit the number of disjoint paths for a TSN stream. Some of the TSN standards specify protocols and methodologies for network elements, such as Frame Replication and Elimination for Redundancy.
2. Annex B provides a description of the UNI YANG module, the data interface that defines the communication between the CUC and the CNC.
3. Annex C is the project developer's guide for the software implementation. Based on NodeJS, CUC can be set up in any device, while the endpoints need to have a TSN compatible interface. This annex aims to describe the project's structure and some of its insights to help future developers.
4. Annex D gives a sample of an endpoint configuration, providing specific details on how the Linux system tools are used. Its content is very important to understand how the devices are automated from the CUC configuration, since these tools are used in the prototype.

CHAPTER 1. DATA LINK LAYER AND IEEE STANDARDS

This chapter describes some technologies and implementations related to the design and deployment of the thesis’ prototype. It provides information from standards that helps the reader to completely understand the current status of TSN.

1.1. Data Link layer

Time Sensitive Networking is aimed to be implemented within the Layer 2 of the OSI model, the Data Link layer. The Data Link layer is responsible for preparing the data that will be transmitted through a physical channel. Its goal is to transmit the data without errors between two adjacent network nodes. To achieve that, it offers medium access mechanisms, by performing device addressing and defining the network topology.

The functionalities that are closer to the physical layer rely on frames, composed of payload and headers (additional information related to routing and forwarding). This frame’s format is not unique, it depends on the physical access they have been designed for. **Figure 1.1** shows the generic Ethernet and WiFi frames. Their definition belongs to the *Medium Access Control* (MAC) sublayer, defined in IEEE 802.3 and 802.11 standards, respectively.

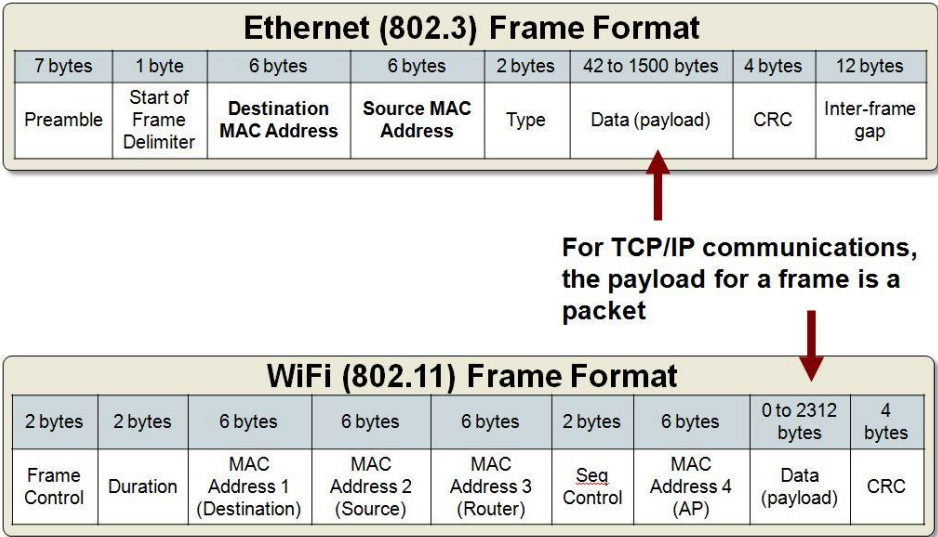


Fig. 1.1 Frame format for 802.3 - Ethernet and 802.11 – WiFi [8].

In a higher level there is *Logical Link Control* (LLC), in charge of offering to upper layers an interface that is independent from the physical device that will be used.

From a practical point of view, we can consider the LLC as the network driver that uses the *Network Interface Card* (NIC) and the MAC handles the protocol performance that will be executed at the system kernel, before sending a payload to the physical device.

The services that the Data Link layer provides are the following [9]:

1. Generic
 - a. Conforming frames from upper layer data packets.
 - b. Frame synchronization
2. LLC
 - a. Flow and error control, commonly used in wireless protocols, since wired Ethernet is nowadays practically error free.
3. MAC
 - a. Medium Access Control (CSMA/CA, CSMA/CD)
 - b. Physical addressing.
 - c. LAN switching, including Spanning Tree Protocol or Shortest Path Bridging.
 - d. Queues and scheduling
 - e. *Quality of Service* (QoS)
 - f. *Virtual LAN* (VLAN)

As we can see, the second OSI layer includes a wide variety of functions. Some of them are not related to our working context, but others are important, such as VLAN, QoS, queuing or scheduling.

IEEE 802.1 group is responsible for developing LAN standards (see [10] for more information about the standards). The number of protocols that this group defines is quite big, so short a list with the ones that are related in some way or another with our project follows.

1.1.1 IEEE 802.1Q - VLAN

TSN relies on VLANs to distinguish different streams and priorities.

A Virtual LAN is an independent logical network that uses a shared physical network, so that several logical networks can coexist. Thanks to this concept, different flows can be maintained as if they were all in separate networks with its

own broadcasting domain. VLANs can be defined in various manners, such as in/out port, MAC address based or other static methodologies, but VLAN tagging, as defined in IEEE 802.1Q, is the predominant solution.

If we consider the Ethernet in **Figure 1.1** frame with the VLAN tag, the frame is modified as **Figure 1.2** shows:

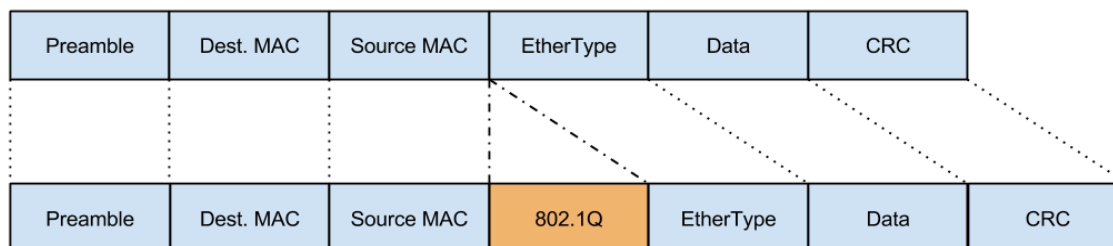


Fig. 1.2 Ethernet frame format adapted with the VLAN tag [11].

802.1Q adds four extra bytes to the original Ethernet frame, and the original's EtherType value is assigned to 0x8100 in order to flag the altered format of the frame and avoid compatibility issues with non-VLAN devices. The 802.1Q tag found in the modified Ethernet frame provides the VLAN information, as follows:

1. User priority (three bits): used by TSN (and other protocols) to determine the priority of the flow in the network.
2. Canonical Format Indicator (one bit)
3. VLAN ID (12 bits): used to uniquely identify the VLAN

1.1.2 IEEE 1588/ 802.1AS – Precision Time Protocol

The goal of this protocol is to synchronize the internal clocks of the elements inside a LAN. These clocks may have distinct features, and the first goal of *generic Precision Time Protocol* (gPTP) is to find the clock with the highest accuracy (Grand Master, GM). That will be the reference for all the other clocks in the network. This process is called *Best Master Clock Algorithm* (BMCA) and it generates a time distribution tree with the GM as root, as **Figure 1.3** shows. In addition, it also handles faults from the Master Clock and resynchronizes all devices.

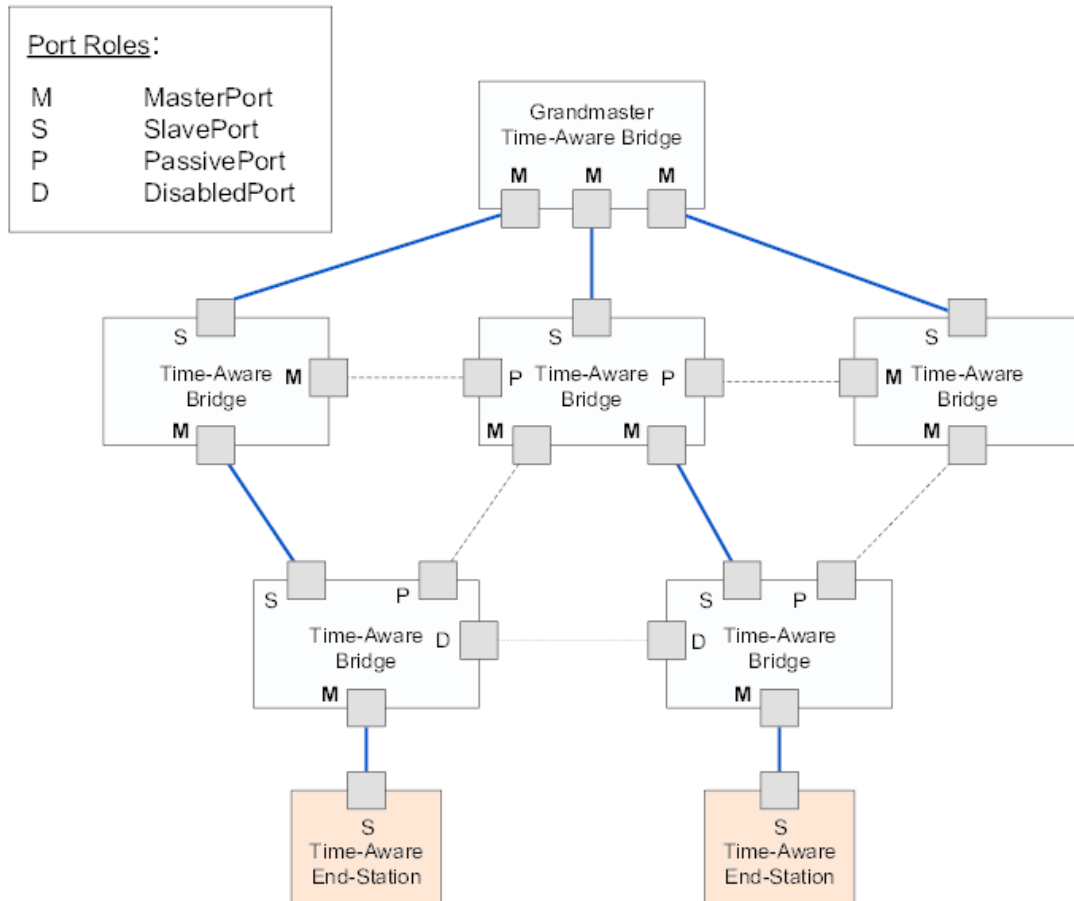


Fig. 1.3 BMCA clock distribution tree [12]. The blue lines are the logical tree that distributes the timing.

Once the *Master Clock* is chosen, other devices will be considered *Slaves* and will be synced to the Master. **Figure 1.3** also shows how network elements set their ports states, to obtain the synchronization from the port with the best timing source.

Once the type of every device's clock is determined, the synchronization process begins, as shown in **Figure 1.4**:

1. Slave clocks periodically receive a *Sync* message from the Master, which includes the time as seen from the Master (T_1), which is received at local time T_2 by the slave, but it is still not correct because of the delay between the slave and the master (offset).
2. Slaves send the *Delay Request* message in the instance T_3 and the Master marks the timestamp before sending it back (T_4) using the *Delay Response*.
3. Slave clocks can calculate their offset with the Master, and use it correctly.

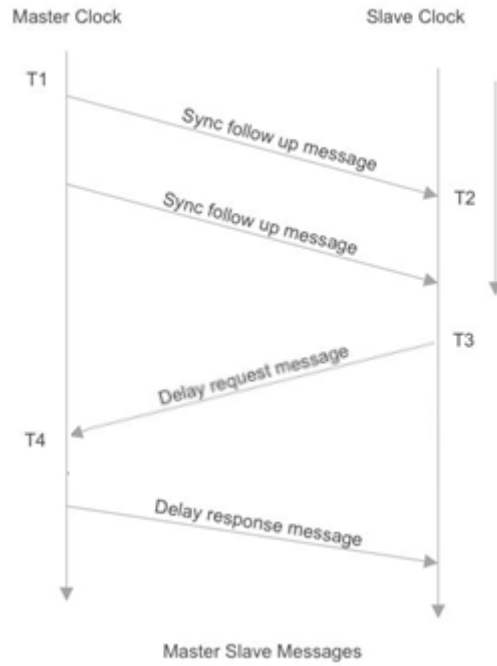


Fig. 1.4 PTP synchronization mechanism [13] [14].

The offset between Master and Slave clocks is determined by the following expressions:

$$\text{Offset} = (T2 - T1) - \text{Path Delay} \quad (1.1)$$

$$\text{Path Delay} = \frac{(T2 - T1) + (T4 - T3)}{2} \quad (1.2)$$

The precision level depends on the network's performance, since a high jitter implies an inaccurate path delay estimation and, consequently, an imprecise offset. Once all clocks are synchronized, PTP/gPTP protocols are only useful to maintain this status in case of failures or disconnections.

In order to improve the accuracy, and since the actual time of emission of the Sync message may not be exactly the same T1 that appears in the message, a Follow Up message (carrying the actual time the previous Sync message was sent) can be used. The process is shown in **Figure 1.5**, where the middle bridge computes its own clock before forwarding a new generated *Follow Up* message. Note that **Figure 1.4** shows the Sync and Follow up transmissions into a single one to simplify the communication between endpoints, even though both are separate transmissions.

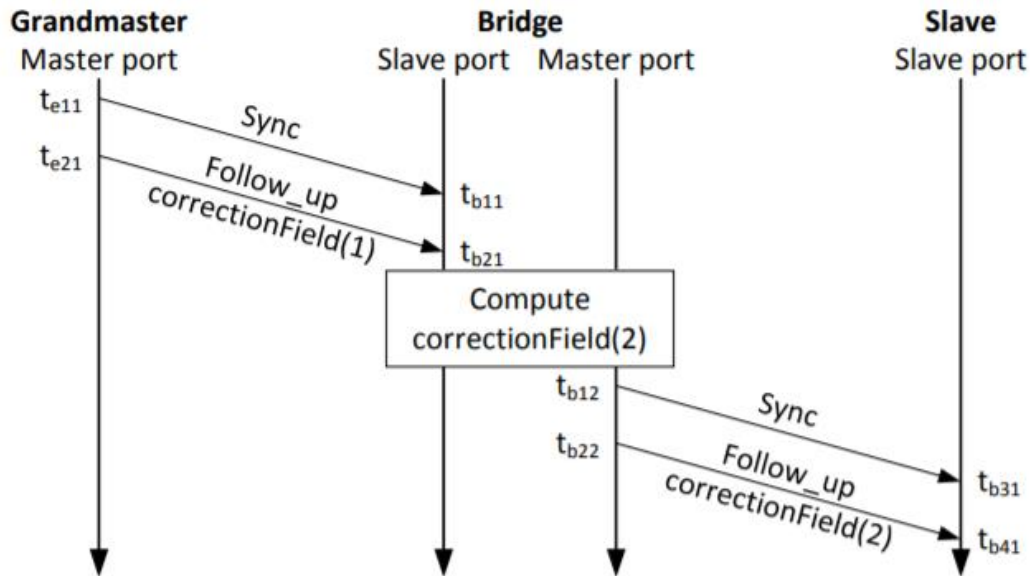


Fig. 1.5 gPTP port-bounded mechanism [13].

All devices that take part in a TSN network must implement gPTP, since an accurate synchronization is crucial for a good performance of the real time standards.

1.2. Time Sensitive Network standards

Since TSN is the most important field of knowledge of this document, there is a need to review all the contents inside 802.1 standards and, specifically, the amendments to 802.1Q standard. Each one of the standards gives different approaches on TSN.

1.2.1 IEEE 802.Qbv – Enhancements for Scheduled Traffic

The *Time Aware Scheduler* (TAS) defines a periodical cycle in which it is possible to specify slices of the link capacity. These slices are slots of time in which the data transmission is reserved only from specified queues. Its objective is to distribute all the different TSN priorities and best-effort flows in this interval and ensure a reservation of the bandwidth for all of them. An example of this idea is shown in **Figure 1.6**.

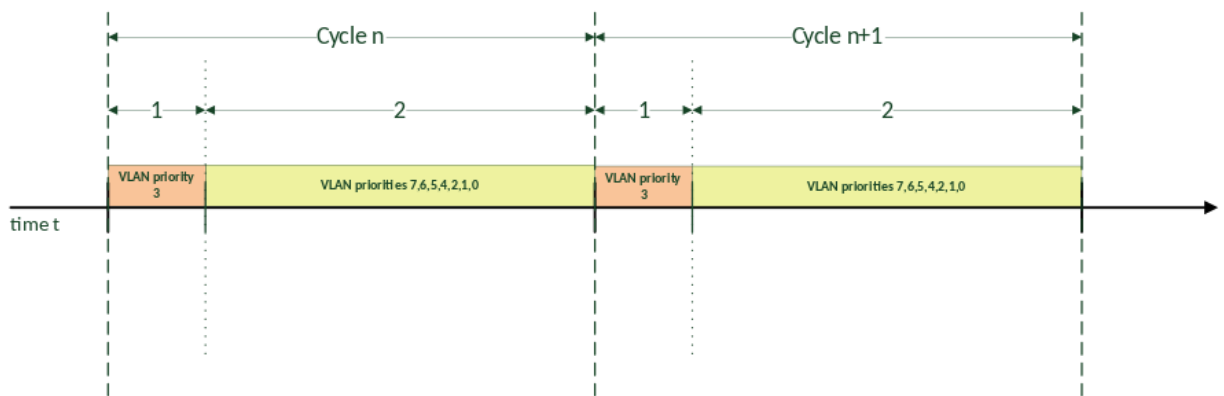


Fig. 1.6 802.1Qbv Cycle schema [15].

The only problems that may emerge from this time distribution is that the sending of the frame starts too close to the next reserved slot. Therefore, the transmission may invade the next slot. In order to avoid that, it is necessary to add a guard interval. The size of the interval should correlate with the transmit time that it takes to send the maximum size Ethernet frame, 1524 bytes. This guard avoids the current data to overlap with the next priority slot. Because of the guard intervals, the transmission time will be reduced, as shown in **Figure 1.7**.

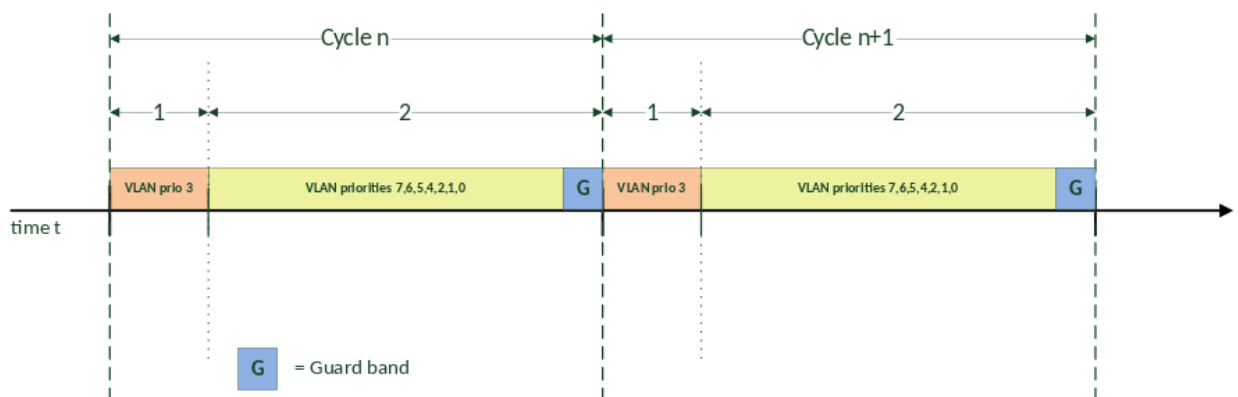


Fig. 1.7 802.1Qbv Cycle schema with guard band [15].

Since the publication of IEEE 802.1Q-2014, a new enhancement proposal named *Enhancements for Scheduled Traffic* (EST) dynamically establishes a status for every different priority. A *gate* is associated with each different traffic priority, and the gate state determines if it is possible to transmit a packet from a certain queue or not. More specifically:

1. Open: frames are selected to be transmitted according to the algorithm in the queue, such as Credit Based Shaper (seen in the following section 1.2.2) or First in First Out (FIFO).
2. Closed: frames from this priority will not be transmitted.

In order to have control over the gates' status, there is a list known as *Gate Control List* (GCL), which every entry in it contains the gate status for all priorities. **Figure 1.8** shows an example with different priorities, all with their own transmission algorithm and a gate controlled by the GCL pointer.

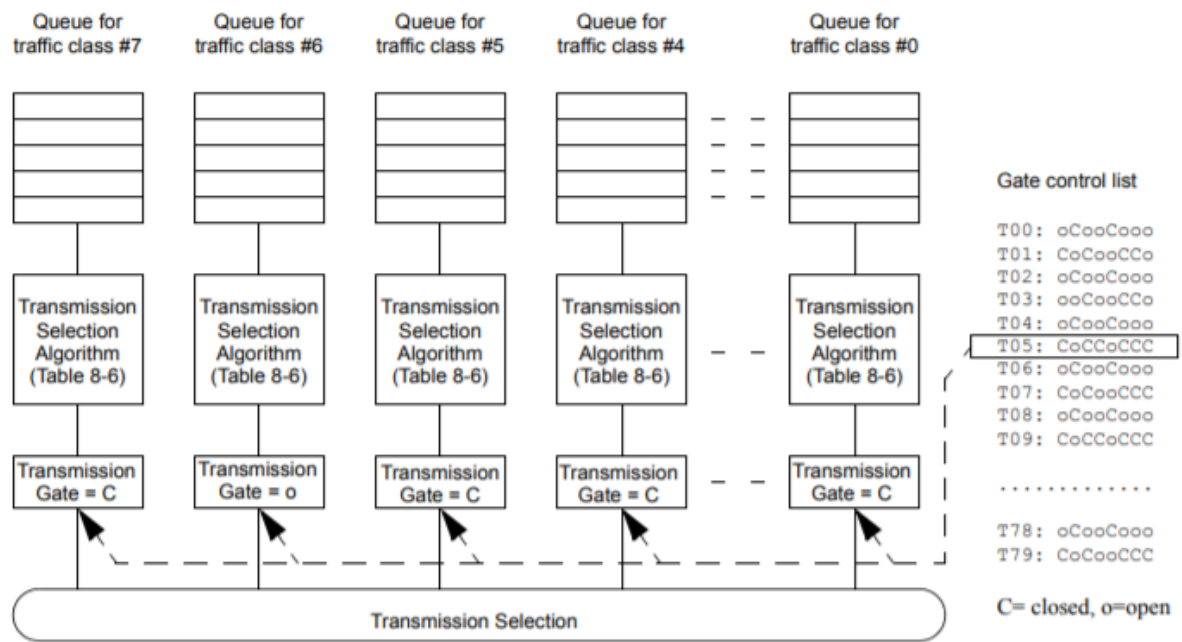


Fig. 1.8 Management of priority queues [16].

Thanks to this gate management, it is possible to prioritize the frames from a given VLAN tag, ensuring a deterministic transmission time and bandwidth. The operations that can be performed on the gates include:

1. SetGateState: changes the gate state of the gate, specifying a duration interval to avoid premature changes.
2. Set-And-Hold-MAC: acts as SetGateState, but it indicates the moment in which the sending of a preemptible frame shall be interrupted (check an introduction to Frame Preemption in section A.7 of the annexes).
3. Set-And-Release-MAC: equivalent to the previous one, but instead of indicating the interruption point it indicates the resume point.

One of the main goals of prototype developed in this thesis is to provide the TSN endpoints with the necessary configuration to perform this traffic scheduling.

1.2.2 IEEE 802.Qav – Credit Based Shaper

Originally, the 802.1Qav standard was independent, but since the release of 802.1Q-2011 we can find it inside 802.1Q standard. The important content of 802.1Qav is the Credit Based Shaper (CBS), which is used as a bandwidth limiter for different traffic queues that follow VLAN's priority codes. The bandwidth limiter relies on several parameters, as follows:

1. *portTransmitRate*: link speed.
2. *idleSlope*: when the queue is not transmitting, it gains credits based on this rate. It can never be higher than the *portTransmitRate*.
3. *transmit*: boolean value that maps the current door status (transmitting or idle). If combined with frame pre-emption, the door status will be idle every time that pre-emption takes part. Because of that combination, the CBS is not losing credits from overheads, for example.
4. *credit*: number of bits available to be sent.
5. *sendSlope*: when the queue is transmitting, it loses credits based on this rate. It is a value obtained straight from *idleSlope* and *portTransmitRate*:

$$\textit{sendSlope} = \textit{idleSlope} - \textit{portTransmitRate} \quad (1.3)$$

6. *transmitAllowed*: boolean value that maps if the credit is positive or negative to determine the possibility to dequeue data.

Figure 1.9 illustrates how the process works and the role of these parameters. More details of this standard may be found in Annex L of the official IEEE 802.1Q-2018 standard [18].

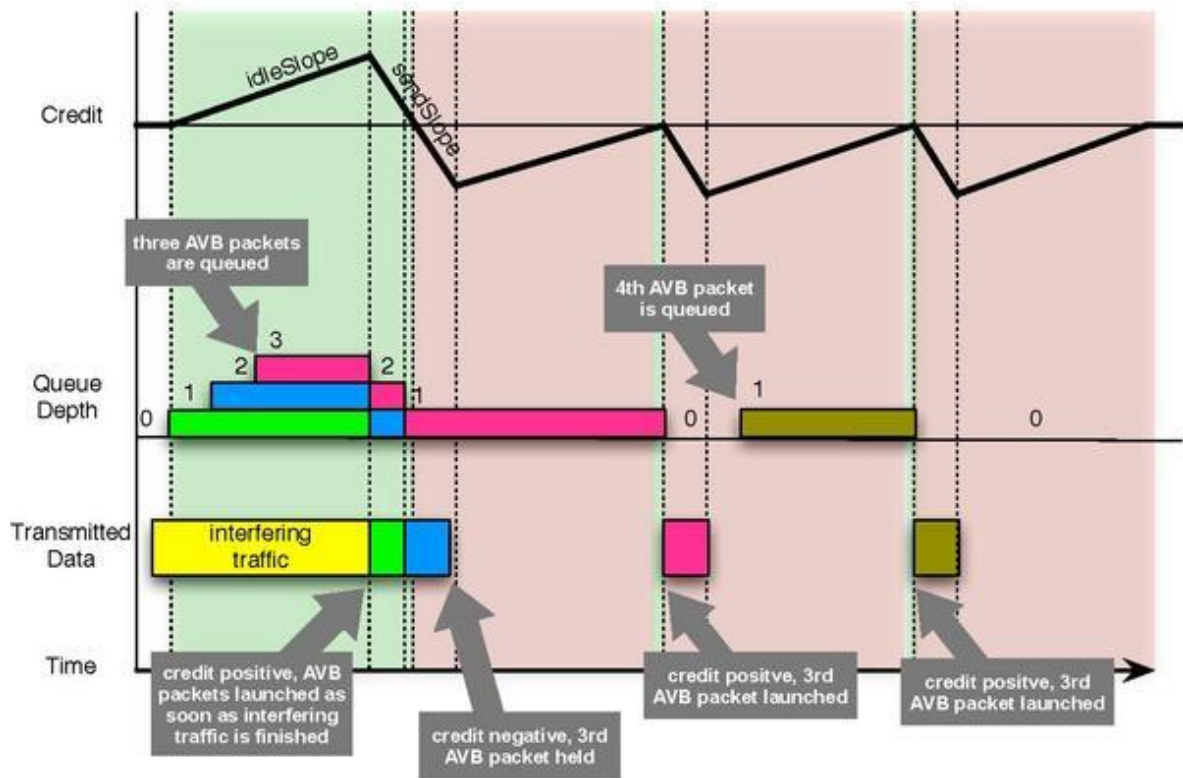


Fig. 1.9 Credit Based Shaper bandwidth limiter [17].

1.2.3 Standards in process

As of June 2021, the TSN IEEE group is still developing new standards and sub standards from 802.1 and 802.1Q, respectively. In the future, some of them may have been already released. Here is a list of all of them [19]:

1. Standalone
 - a. IEC/IEEE 60802 - TSN Profile for Industrial Automation
 - b. P802.1CS - Link-Local Registration Protocol
 - c. P802.1CQ - Multicast and Local Address Assignment
 - d. P802.1DC - Quality of Service Provision by Network Systems
 - e. P802.1DF - TSN Profile for Service Provider Networks
 - f. P802.1DG - TSN Profile for Automotive In-Vehicle Ethernet Communications
2. Amendments
 - a. P802f - YANG Data Model for EtherTypes

- b. P802.1Qcj - Automatic Attachment to Provider Backbone Bridging services
- c. P802.1Qcw - YANG Data Models for Scheduled Traffic, Frame Preemption and Per-Stream Filtering and Policing
- d. P802.1Qcz - Congestion Isolation
- e. P802.1Qdd - Resource Allocation Protocol
- f. P802.1Qdj - Configuration Enhancements for TSN
- g. P802.1ABcu - LLDP YANG Data Model
- h. P802.1ABdh - Support for Multiframe Protocol Data Units
- i. P802.1ASdm - Hot Standby
- j. P802.1ASdn - YANG Data Model
- k. P802.1CBcv - FRER YANG Data Model and MIB
- l. P802.1CBdb - FRER Extender Stream Identification Functions

This chapter is extended in Annex A with all the standards that are not in contact with the current project, but they are still related to time-sensitive networking.

CHAPTER 2. NETWORK MANAGEMENT AND OPC-UA

This chapter presents the technologies that are candidates to be used in the CUC prototype. It starts with the description of YANG, NETCONF and RESTCONF, which are new approaches to network management. In addition, the chapter introduces OPC-UA, a framework that aims to enable the communication between any device in a network.

2.1 NETCONF, RESTCONF, YANG and Configuration Data Models

The goal of this section is to introduce the concepts and protocols related to network management. Traditionally, network management has been based on tools such as scripts, *Command Line Interface* (CLI) and *Simple Network Management Protocol* (SNMP). We will comment how ineffective these tools are in terms of network automation. Additionally, we will introduce the new network management paradigm, based on NETCONF and YANG, for building the TSN prototype [20].

Before describing NETCONF and YANG, it is important to define the difference between information models and data models.

2.1.1 Information models and data models

First, we will describe how CLI, and SNMP have been used for network management [21]. These tools are based on information models and its disadvantages are presented in the following lines.

2.1.1.1 Command-Line Interface (CLI)

CLI requires proprietary implementations for each different model, preventing quality or robust automation. It is sensitive to different modifications on network structure, device configuration or firmware. Moreover, when a device is being configured, there is a period of time in which only part of the configuration is loaded to the component. If an error in the configuration script occurs, then the device will remain corrupted, implying a high operational cost for its recovery.

In conclusion, handling these processes is too expensive and requires complicated programming. Also, this methodology is inefficient in reporting the devices' status to a possible network controller, since polling would be required, a non-efficient technique.

2.1.1.2 SNMP

The SNMP protocol was designed to monitor and manage a network. It is capable of writing changes in the devices' configurations and receiving asynchronous notifications from them. Nevertheless, it is typically only used for fault monitoring and actuation, not for configuring devices. The reason is that SNMP does not offer a standard or automated manner of identifying the different modules in the *Management Information Base* (MIB), forcing the operators to perform that discovery stage for every device. Also, disadvantageously, it relies on UDP transport, yielding the opportunity to lose some critical configuration or operational data.

2.1.1.3 Information models and data models

If we consider a network controller based on either of the tools just mentioned, we would have to use *information models* to maintain the control of each configuration for every different network element.

An information model has a goal of organizing the data at a conceptual level, independent from the different configuration means the device may have. To simplify an information model, any detail referred on the implementation necessary to set up the configuration is avoided. It only handles the actual data that will be configured. This type of model is useful for network designers to describe elements' details. It also supplies operators with a reference to what data has to be set up. If that model becomes more specific, we may get a *data model* [22].

Data models are more closely linked to the implementation than information models; these include structures related to the process of setting up the device, such as rules, restrictions or permissions. The objective of data models is to maintain the controller and device with the same instance configuration data. This data has a strict design on how changes must be done and must ensure that input parameters are set accordingly.

As an example, the configuration process for an Ethernet interface needs the IP address, the default gateway and the subnetwork mask. If an information model is used, the values for all fields are given and the configuration setting is not handled. For the case of a data model, the fields are given in a known object instance. As the configured device knows it, it has the automated process to set up the configuration, because the data received is known and valid.

These new concepts about data models can be better explained by introducing YANG and NETCONF/RESTCONF.

2.1.2 YANG

Yet Another Next Generation (YANG) [23] is an encoding language used to describe data models, maintained by NETMOD (an IETF working group) and specified in RFC 7950.

Its main strength is that it uses a strict syntax to define data models and ease automated reading and writing. To understand more clearly the data models' goals and how YANG achieves them, we will base the explanation in a practical example [24] – the process of creating a data model to describe an Ethernet interface. We will begin with the module definition, which includes brackets with all of the module content inside.

```
module ultraconfig-interfaces {  
  
}
```

Fig. 2.1 Module definition in YANG language.

Add the header information of the module, shown in **Figure 2.2**:

```
yang-version 1.1;  
  
namespace  
  "http://ultraconfig.com.au/ns/yang/ultraconfig-interfaces";  
  
prefix if;  
  
organization  
  "Ultra Config Pty Ltd";  
  
contact  
  "Support: <https://ultraconfig.com.au/contact/>";  
  
description  
  "This YANG module has been created for the purpose of a tutorial.  
  It defines the model for a basic ethernet interface";  
  
revision "2020-01-03" {  
  description  
    "Initial Revision";  
  reference  
    "Learn YANG - Full Tutorial for Beginners";  
}
```

Fig. 2.2 Header definition of the YANG module.

1. `yang-version`: version of the YANG standard used.
2. `namespace`: used to identify the module uniquely, so that URL/URI formats are used, since it is unlikely that two independent developers use the same one.
3. `prefix`: identifies the defined module used when another YANG module imports the current one the use their properties.
4. `organization` and `contact`: provide information about the creators of the module.
5. `description`: short summary about the module's goals and what it includes.
6. `revision`: version control field. Each new version a revision entry is added with the details about the latest changes.

Until this point, there is only a header, with more bureaucratic than useful information. **Figure 2.3** shows the different elements that conform the YANG module's core:

```
typedef dotted-quad {  
  type string {  
    pattern  
    '([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\.){3}'  
    + '([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])';  
  }  
  description  
  "Four octets written as decimal numbers and  
  separated with the '.' (full stop) character."  
}
```

Fig. 2.3 Core definition of the YANG module.

Figure 2.3 shows a new type definition, specifically a type that can be used to instantiate IP addresses or masks (check the regular expression). Since this is basically a string, it is easy to wonder why is it useful to define a new type for it. The reason is that it may contain fields inside, such as *pattern*. This property coerces that any instance of this type must be compliant with a given format. If a value of a *dotted-quad* type is “TSN Network”, it will not be accepted in the instance data, since it can generate configuration failures. Now, understanding the use of new types, we need to locate them in the module. Specifically, all types are found inside the *container*.

A container includes *configuration data*, which are all the fields that can be read or written, such as the value of the IP address or the “up” and “down” interface

status. Also, it can contain *state data*, read-only properties of a module, such as the number of packets transmitted by the given interface.

First, let's take a look at the configuration data, an easy example compared to the ones used by the IEEE standards. In **Figure 2.4** there is a definition of four leaves, which maps information from an interface. Each leaf specifies its own type and description, but also can include various inner properties. As an example, the pattern used in **Figure 2.3** or the *default* flag found in the “*enabled*” property. Notice that the “*address*” and “*subnet-mask*” are defined as *dotted-quad* types, so the pattern designed in it will be applied.

```
list interface {  
  key "name";  
  leaf name {  
    type string;  
    mandatory "true";  
    description  
      "Interface name. Example value: GigabitEthernet 0/0/0";  
  }  
  leaf address {  
    type dotted-quad;  
    mandatory "true";  
    description  
      "Interface IP address. Example value: 10.10.10.1";  
  }  
  leaf subnet-mask {  
    type dotted-quad;  
    mandatory "true";  
    description  
      "Interface subnet mask. Example value: 255.255.255.0";  
  }  
  leaf enabled {  
    type boolean;  
    default "false";  
    description  
      "Enable or disable the interface. Example value: true";  
  }  
}
```

Fig. 2.4 Configuration data definition of the YANG module.

After the definition of the configuration data, the state data definition is presented. All state data is embedded in a list that contains more than a subtype of every element. Every entry on the list will have a specific leaf “*name*” and another named “*oper-status*”. The “*name*” is a common string and “*oper-status*” is an enumeration defined inside the leaf. This enumeration can only be used by the “*oper-status*” variable, different to the type definition defined in **Figure 2.3**. If we take a look to **Figure 2.5**, the “*interface-state*” list has a property “*config*” set to false. This means that the list is a state variable and cannot be modified as a

configuration variable. In addition, the “key” property sets the attribute that will be used as unique reference to the element.

```
list interface-state {  
  config false;  
  key "name";  
  leaf name {  
    type string;  
    description  
      "Interface name. Example value: GigabitEthernet 0/0/0";  
  }  
  leaf oper-status {  
    type enumeration {  
      enum up;  
      enum down;  
    }  
    mandatory "true";  
    description  
      "Describes whether the interface is physically up or down";  
  }  
}
```

Fig. 2.5 State data definition of the YANG module.

With all these definitions, this is a built YANG module. This module can be instantiated with data, like example is shown in the **Figure 2.6**. YANG instances intend to be sent to a network element that has the same YANG module. Since they share the same module definition, a correct configuration is ensured. The following XML contains two distinct interfaces that contain an IP address and a subnet mask assigned to each of them. Since the “enable” field is not included, both will adopt the default state, which is set to “false”.

```

<data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <interfaces xmlns="http://ultraconfig.com.au/ns/yang/ultraconfig-interfaces">
    <interface>
      <name>GigabitEthernet 0/0/0</name>
      <address>10.10.10.1</address>
      <subnet-mask>255.255.255.0</subnet-mask>
    </interface>
    <interface>
      <name>GigabitEthernet 0/0/1</name>
      <address>192.168.1.1</address>
      <subnet-mask>255.255.255.0</subnet-mask>
    </interface>
  </interfaces>
</data>

```

Fig. 2.6 Instance data of the YANG module.

According to this example, the YANG modules specify formats on the configuration of the different protocols that a machine can implement. Additionally, it offers the possibility to give these configurations with a well-formed guarantee to avoid misconfigurations.

Despite having a clear definition of the YANG modules, there is a need to set up an environment and logic on them that sends and receives the instance data. This point is where NETCONF or RESTCONF come in play, since they equip the logic on the establishment and monitoring of the configuration, as well as the status of the devices based on the YANG modules shared by the controller and the device.

2.1.3 NETCONF

NETCONF [25] appeared as an answer to the inefficiency of SNMP and other network configuring protocols, not user friendly or standardized. It needs specific implementations for each device model.

NETCONF is based on a client – server communication, where the controller acts as a client and the network element as the server. In summary, this protocol performs two different operations: *read* the operational status of a device or *update* its configuration. Moreover, it allows the servers to perform actions, configuration or management processes triggered by the clients. In addition, clients can subscribe to notifications, so that they receive the updated information without the need of a request. Normally, every message exchanged between client and server is a *Remote Procedure Call* (RPC) or *RPC-Reply*, the response generated by the server. An RPC-Reply can be a simple *OK* indicating the success of the operation or it may contain the entire device configuration, depending on what is requested. By standard, RPC is encoded in XML and the

communication between client and server is performed by SSH using the port 830, as default.

Before diving into the contents of the different RPCs, the architecture of a server based on data models will be presented, as it helps the understanding of NETCONF and RESTCONF functionalities.

2.1.3.1 Data model-based server architecture

Figure 2.7 summarizes the general architecture of a server, valid for both NETCONF and RESTCONF servers.

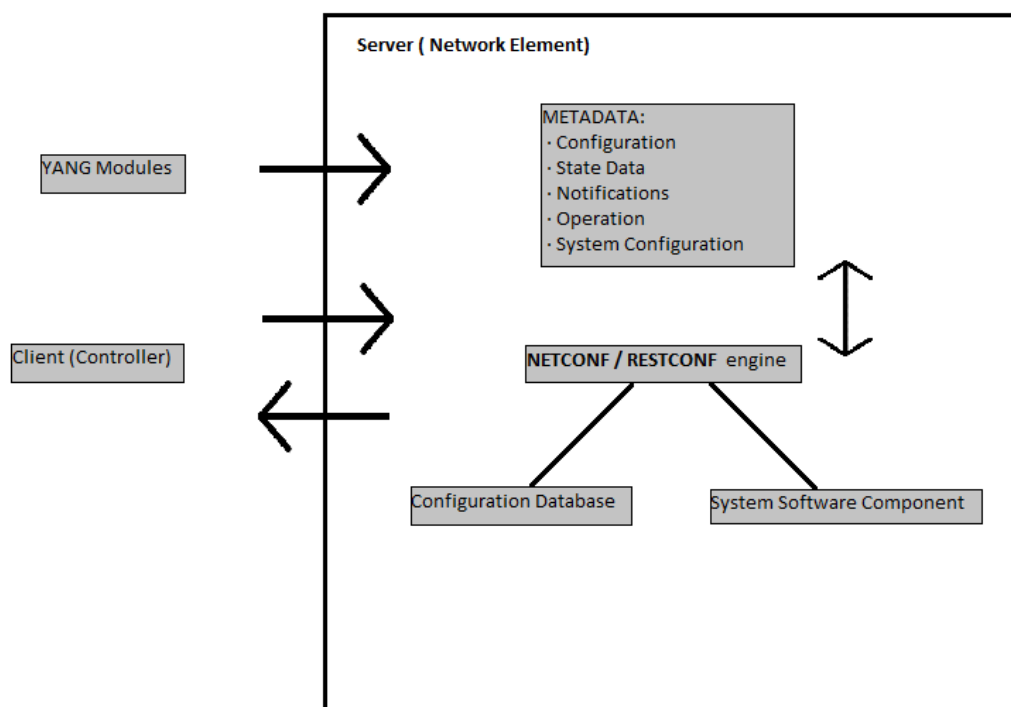


Fig. 2.7 Schema of a data model server.

YANG modules follow a model-based server architecture because, as mentioned, they are used to defining the servers' needs in terms of implementation and monitoring. These YANG modules will be integrated in the system, so that the server's configuration and status can be mapped to the instance YANG data and treated with RPCs. When the NETCONF core receives a request from a client, it parses the content, verifies the YANG models and executes the demanded actions. This process includes setting the *configuration database* and the configuration of the device in a strict controlled manner. If an error occurs, a roll back of the steps is performed by the server. In some cases, the actions will be to poll data to update the client. Others can be to modify any configuration entry in the configuration database, which is the storage that maps the actual configuration of the device. After processing all the different requests

that a client has sent, a single response will be generated, containing the results for all the requests.

2.1.3.2 NETCONF features

This protocol provides several improvements over CLI or SNMP-based management:

1. NETCONF transactions guarantee that the configuration is applied entirely, following the ACID concept used in databases:
 - a. Atomicity: if all changes are not applied, all new configuration will be discarded. This avoids the chance to abort during the middle of the configuration process.
 - b. Consistency: all in one. All the new configurations that are included in the instance YANG data are applied at the same time, avoiding transactional states and increasing the simplicity in servers.
 - c. Independence: multiple clients can perform changes in the configuration without interfering.
 - d. Durability: when a transaction has been performed, there are guarantees of not losing them, even with a total disconnection from the device.
2. NETCONF can manage different contexts, such as global configuration and specific YANG modules.
3. NETCONF servers offer the distinction between distributing data model instances and applying them. This means that a certain configuration may be received and processed by the server, but not applied until the client specifies to do so.

These features come from the fact that NETCONF includes mechanisms to control the configuration database. However, the server does not only have one database; it has three, all of them modifiable:

1. *Startup configuration datastore*: its content is read and applied to the running datastore when the device is powered on. This means that the device will be configured at power on as stated in the content of this database.
2. *Running datastore*: contains the actual configuration of the device.
3. *Candidate datastore*: space to apply changes without affecting the active configuration. When a server receives an RPC *“commit*

configuration”, the content of this datastore will be transferred to the running datastore.

Prior to the RPC exchange between a client and a server, the NETCONF protocol starts by sending a *hello* message, where parties specify the protocol version and expose the supported YANG modules. After this first exchange, RPC and RPC-Reply can be performed. The main different calls are listed below:

1. *get-config*: to obtain a part or the whole configuration datastore.
2. *edit-config*: change the content of the configuration datastore.
3. *copy-config*: copy content from a datastore to another. For example, to set the startup datastore with the content of the running one.
4. *delete-config*: deletes the specified contents of the given datastore.
5. *lock*: locks a datastore, avoiding it to be changed by another client or datastore.
6. *unlock*: releases the locking of a datastore.
7. *get*: returns the running datastore and the device status information.
8. *close-session*: finishes the NETCONF session.
9. *kill-session*: forces the finalization of the NETCONF session.
10. *get-data*: a more flexible manner to get configuration and state information.
11. *edit-data*: also, a more flexible way to modify the contents of a datastore.

As a result of the configuration datastores and the types of operations that RPCs perform to the server, the transactions that include changes in the configuration can be protected from interferences. As an example, the configuration is sent to the configuration datastore (locking it) and the update to the running datastore is triggered. By locking the configuration datastore, others cannot interfere in the current configuration process, so that all configuration processes can be handled in a controlled manner.

2.1.4 RESTCONF

This protocol can be considered an adaptation from NETCONF to the known RESTful APIs, basing the communication on HTTPS requests, instead of a persistent communication like SSH that NETCONF uses. Due to this, RESTCONF [26] is considered a subset of NETCONF, with some differences:

1. RESTCONF considers in its standard both JSON and XML encodings for the requests (YANG data instances). NETCONF only takes XML into account.
2. Due to the nature of HTTPS, it is not possible to split requests in different messages. All of the information is sent together.
3. RESTCONF does not have any manner to validate the changes before applying a new configuration. Instead, RESTCONF implicitly checks the data during the request, validating the YANG module. In other terms, the only datastore available in RESTCONF is the running datastore, so it is not possible to modify the candidate datastore to perform more complex testing before commitment.

To summarize, RESTCONF does not have any notion about the lock, candidate datastore or operation commitment. We could conclude that it is not an optimal protocol to manage different devices, since clients have less control over the servers. Nevertheless, it is interesting to rely on this protocol to communicate with the network controller's northbound interface (explanation extended in Chapter 3).

2.2 OPC-UA

OPC-UA [27] is the evolution from OPC Classic, and can be described as a workspace or architecture that aims to communicate different platforms in a secure way. It is arranged for services in the industrial or business environment, to interconnect every single device to the processes. All communications performed in OPC-UA are done between a client and a server.

The client and the server identity may be confusing at first for some software developers, due to the fact that in automation, all endpoints (such as sensors or actuators), are considered servers. Then, the clients are typically the controllers or managers. The clients obtain as much information as needed from the servers distributed on the industrial floor, being able to have control over all these elements. **Figure 2.8** shows an overview on how a server can receive requests from several different clients. Each of the clients performs an operation with the server, which may include, for example, an action or a data poll.

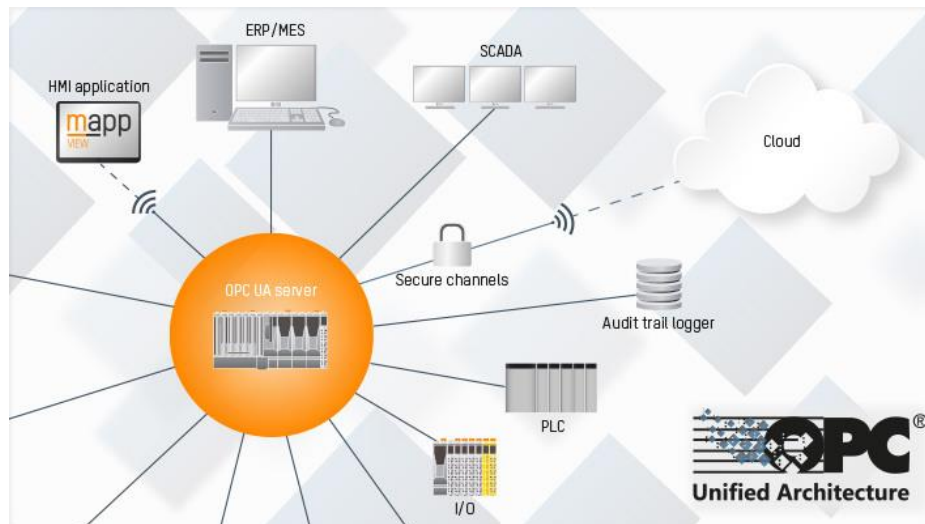


Fig. 2.8 OPC-UA schematic overview, from [28].

The main differences between OPC-UA and OPC Classic are the following ones:

1. OPC Classic is strictly dependent on Microsoft, since the communication relies on DCOM, a technology considered today as deprecated.
2. Insufficient data models: OPC Classic is not able to represent data types and their relations, which may be essential for an accurate industrial design and data treatment.
3. Security: DCOM is not a safe protocol. As an enhancement, OPC-UA bases all its communications in secure channels, using TLS and certificates to authenticate every device.

OPC-UA is designed to connect all levels of data management that may partake in an activity. For example, in an industrial context, for example, OPC-UA can communicate databases, the ERP (SAP, Odoo, among others), sensors, controllers or monitoring devices. OPC-UA aims to be scalable, because it does not provide many specifications on how to configure a device to take part in the system. This offers the possibility to very different elements to take part in the architecture. It intends to be secure, since it uses the TLS protocol for the communications, as well as being resilient, offering alarms, events, discovering, reading or writings, among others.

OPC-UA is a service-oriented architecture, which includes:

1. **Discovery**: clients can get to know which data is available, its relation with other fields and which metadata can be processed.
2. **Publish and Subscribe**: clients use this type of communication to choose how and when they want to receive the desired data. For instance, a Publish/Subscribe communication requires time-sensitive handling.

3. Node: used to create, delete or modify the data structure, known as Address Space, which the server maintains.
4. Methods: nodes used to call functions in the servers.

OPC-UA extends beyond the boundaries for this project, because it brings all the necessary parts to interconnect any system in an industrial network. However, its utilization between endpoints and the controller element of a TSN network may be a viable solution. There also is growing research on the combination of both architectures, which state that implementing OPC-UA over a TSN network may provide real time and loss guarantees between OPC-UA components. Using this solution, there can be a temporal synchronization in the critical processes that may require a strict synchronization with other elements. The idea of OPC-UA over TSN is shown in **Figure 2.9**.

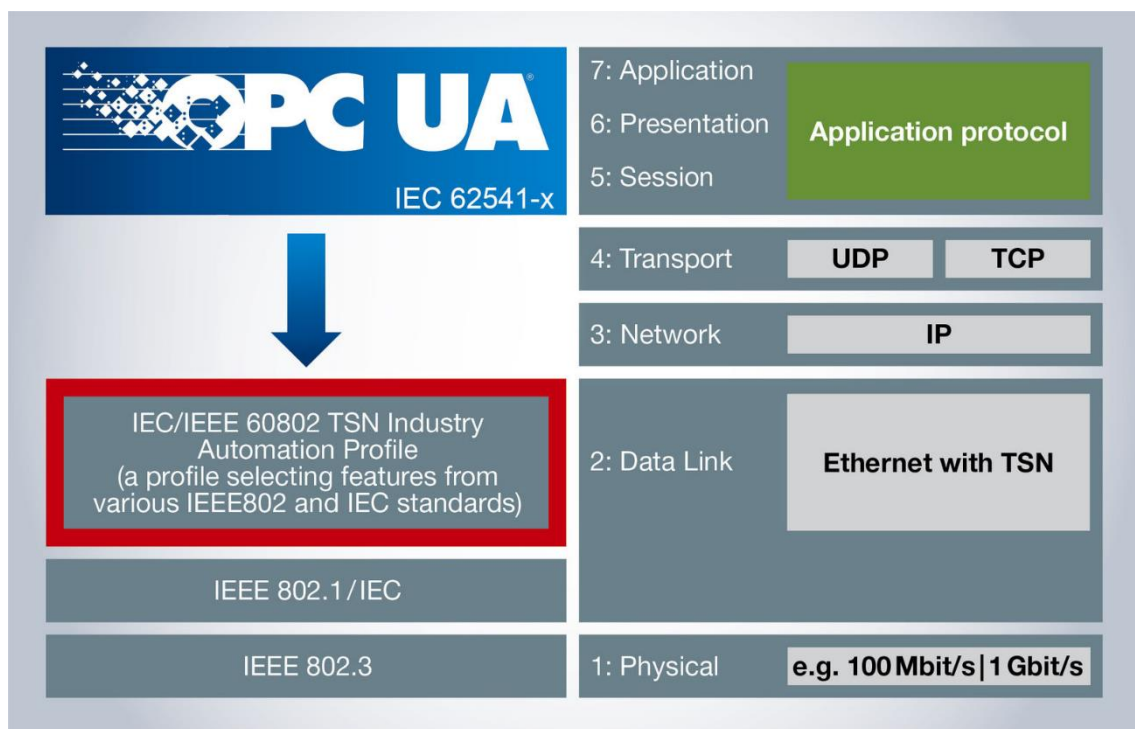


Fig. 2.9 OPC-UA over a Time Sensitive Network [29].

OPC-UA allows communication between any elements in the industrial network. If TSN network is the infrastructure used, then OPC-UA communications become deterministic [30]. In addition, open standards are used throughout all the OSI layers, avoiding any proprietary protocol. By this, it is much easier to combine real time-aware devices with common ones. The time-aware devices need determinism in the network, while others flood the network with best-effort traffic, congesting it.

2.2.1 Clients and Servers

It is important to remark how the concept of client and server differs from what non-automation developers are used to. Clients are the centralized controlling elements and the servers are the distributed endpoints that perform some actions on the industrial network. Due to this, servers are prepared to receive more than one connection from clients to receive data, process some logic or call asynchronous functions [31].

All the communication between the client and a server is based on the server's *Address Space*.

2.2.1.1 Address Space

Address Space is the way that an OPC-UA server structures and presents the content to its clients. It is organized in folders, so that the client has a controlled and orderly access to the contents. For example, it is possible to distribute the content of an Endpoint in two different sections: “*InterfaceConfiguration*” and “*TrafficSpecification*”. The Address Space has different types of entries, called nodes, whose types can be seen in **Figure 2.10**:

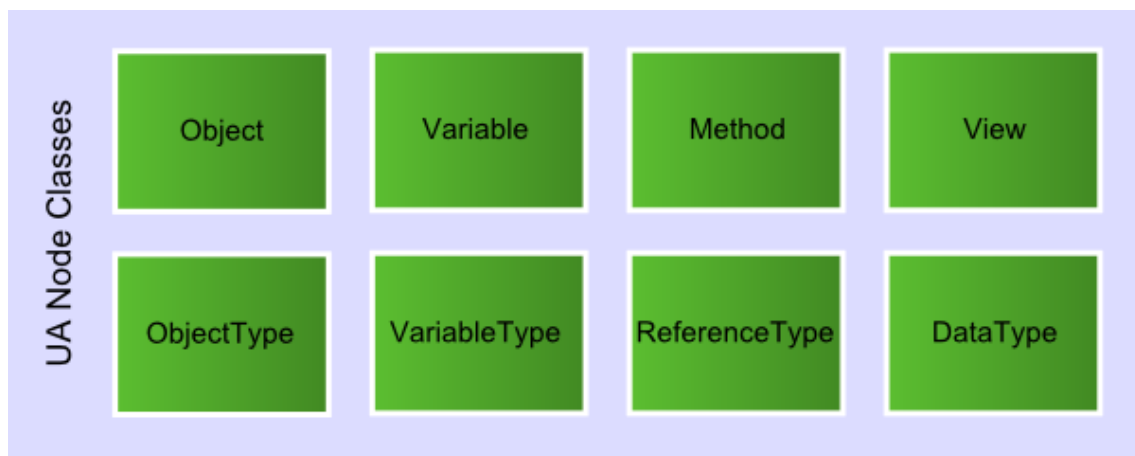


Fig. 2.10 Address Space Node classes [32].

1. *Object*: represents a system, a component or other objects. For example, a network interface card.
2. *Variable*: used to represent objects' properties.
3. *Method*: represents a function that can be called and executed in the server.
4. *ReferenceType*: used to reference other nodes instances.
5. *ObjectType*: straight analogy to objects in Object Oriented Programming.

6. *DataType*: contains all data types that a Variable can adopt.
7. *View*: it restricts the number of visible nodes in an Address Space.

All these nodes in the Address Space are made from one of these classes. The node contents, in general, are attributes and references:

1. *Attributes*: describe the nodes. Read, write and query accessible, also to Subscription and Monitoring services. Every attribute must contain id, name, description, data type and mandatory/optional indicator.
2. *References*: used to relate nodes with other nodes. They are instances of the node class ReferenceType that can be found in the same or other Address Space.

As a conclusion, the OPC-UA servers will maintain the Address Space, which the clients are forced to interact with. All communication between the client and the server will be based on the data in the Address Space. Therefore, there is an important need to design it accordingly to our needs, since we are using our Endpoints as OPC-UA servers and our CUC unit as the client.

2.2.1.2 *Publish/Subscribe*

The Publish/Subscribe technique is a part in the OPC-UA specification. The main goal is to increase the situations in which OPC-UA can be feasible for automation, both for plant communications and cloud-based operations.

Clients subscribe to a certain node in the Server's Address Space. By doing this, the Client will receive any variable changes pushed by the Server, not polling for them. Thus, it offers enormous benefits regarding industrial automation; since anytime new data is set by the Server, it will be instantly sent. This permits the chance to handle the sending with time-sensitive performance [33]. Apart from that, the Servers can publish data to multiple Clients, meaning that the standard gives a lot of flexibility to the implementations it may support.

Figure 2.11 presents an overview of what can be achieved with Pub/Sub from OPC-UA, enabling any kind of non-pollled communication regardless of the network's architecture, since all devices would use OPC-UA.

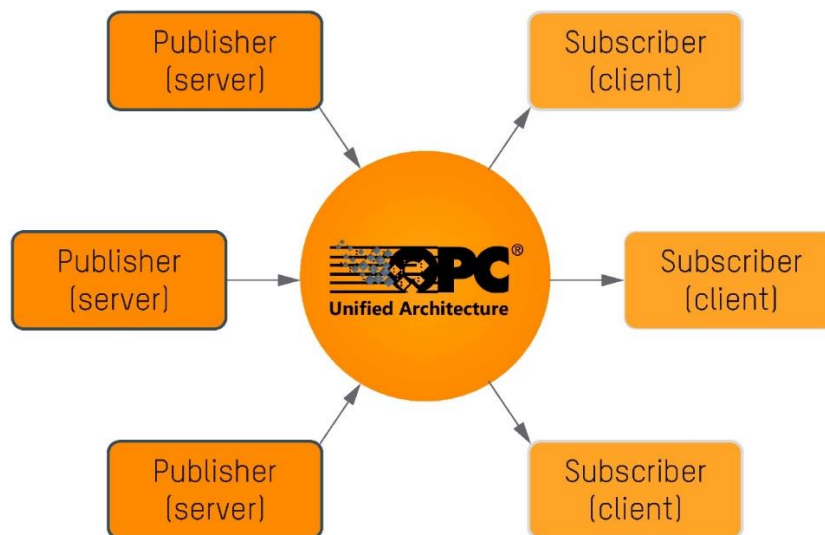


Fig. 2.11 OPC-UA Pub/Sub logical overview [29].

CHAPTER 3. IEEE 802.1QCC

This chapter is where the *Fully Centralized* architecture from the IEEE 802.1Qcc standard is presented. This is the architecture that this thesis' is following. Thus, this standard must be reviewed in detail [34].

3.1 IEEE 802.1Qcc clause 46 – Time Sensitive Networking Configuration

In general terms, the 46th clause from 802.1Qcc exposes the different network architectures for TSN networks. It provides the details of different communications from the control plane, necessary to successfully use the TSN network and its advantages. This project will focus on the content of this clause, specifically the content inside the “*Fully centralized model*”, as seen in **Figure 3.1**.

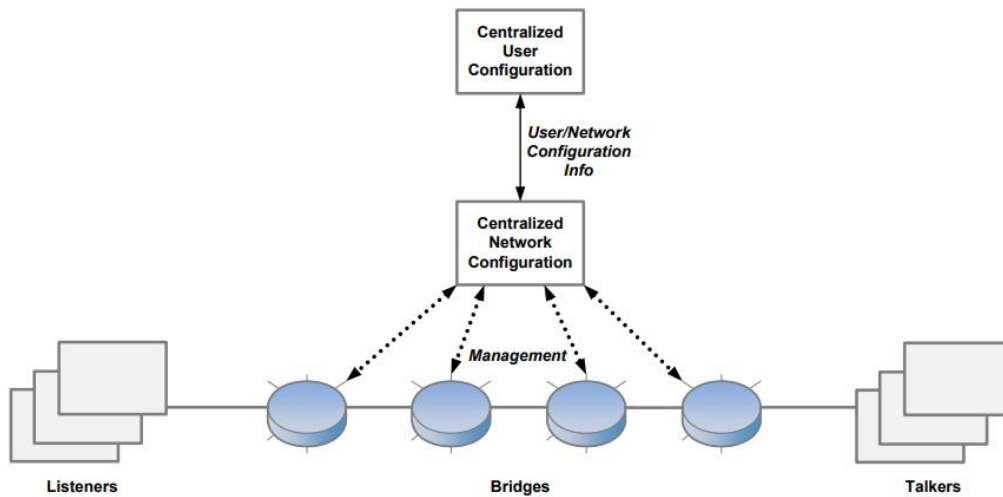


Fig. 3.1 Fully Centralized model scheme, from [34].

The advantage that this architecture has over the others (“*Centralized network / User distributed*” and “*Fully distributed*”) is that it can centralize user’s and network elements’ configuration. This means that the controller will be completely aware of the entire network’s topology, so it can compute an optimal configuration. In fact, this advantage is also found in the “*Centralized network / User distributed*” model; however, the Centralized User Configuration is the key difference, because it manages all the endpoints’ configurations to partake in a successful TSN communication. In the “*Centralized network / User distributed*” the endpoint configuration is managed by the network elements, lacking centralization.

First of all, the CUC discovers all the TSN endpoints in the network, obtaining its capabilities and requirements. With all the requirements obtained for a certain TSN communication, the CUC must give the computed endpoints' configuration to the CNC. The CNC configures the TSN domain in the network and responds the CUC with extra configuration. The CUC will compute and send the different TSN requirements for the Endpoints.

It is important to state that the communication protocol between the endpoints and the CUC falls out of the scope of the IEEE specification. As a result, in **Figure 3.1** such communication does not appear.

Finally, the CUC manages the different endpoints that may share TSN streams by combining their properties and building the User / Network Interface (UNI) (explained in the following Section 3.1.1). The logical process to obtain the data from the endpoints and generate a UNI is the designer's responsibility.

The UNI is the interface that exists between the CUC and CNC (see **Figure 3.1**) and defines all the data exchanged in the communication using YANG data models. As described in Section 2.1, by using YANG modules and NETCONF/RESTCONF based communication, we ensure that the data transmitted is well formatted and validated before being processed. The YANG data model is the encoding of the UNI into a data model detailed below.

3.1.1 UNI Integration

The UNI consists of three high level information groups, considering only the use of a *“Fully Centralized”* model:

1. Talker: elements provided from the CUC that specify a Talker for a given TSN stream.
2. Listener: elements provided from the CUC that specify a Listener of a given TSN stream.

Both the previous groups' data have to be encoded to a YANG instance, coming from the configuration polled from the endpoints (via OPC-UA, for example).

3. Status: elements that are given back from the CNC to the CUC, which specify the result of the configuration of the network elements and part of the configuration needed for Talker and Listeners. They use it to set up their interfaces to emit and receive the TSN stream successfully in the network. It also can contain the failure code in order to give the CUC the enough knowledge to solve the problem with its users.

The unique identifier that must be used for a message is called *“StreamID”*, which identifies the generated configuration over the stream that is going to be transmitted. Conceptually, the communication between CUC and CNC can be interpreted as a request – response process: the CUC sends the Talker and

Listener groups to the CNC and it responds with a Status group. Using the fully centralized model, the CUC may combine multiple groups in a single request. Due to this, the CNC is able to compute the all the requirements, avoiding further costs of reconfiguration and giving the response all in once.

It must be noted that the standard does not specify how these groups should be combined or how the CNC may compute them to offer the best solution to the network. This means that the standard provides only an architecture design, including specified parts, such as the UNI Interface. It also contains unspecified parts that are left to the designer criteria or to future TSN standard releases.

When the CUC is specifying information from a Talker or a Listener, there are two possible actions:

1. Join: an endpoint (talker or listener) joins a stream, so the CNC prepares the network elements to handle the TSN requirements.
2. Leave: alert that the endpoint is no longer taking part of the stream, so that the CNC can perform a resource release.

The CNC should accept the separate reception of the messages, meaning that it may first receive the groups and, afterwards, a call to “Join” or “Leave”. Let’s check the content of each different group in detail, as it will be needed to understand for a good architecture design.

3.1.1.1 Talker

This group specifies the following, grouping the capabilities and requirements that an emitter may request:

1. Talker’s behavior regarding the stream (when it transmits, how, etc.).
2. Talker’s real-time requirements.
3. TSN capabilities of the endpoint’s network interface.

All this information is contained in the following subgroups:

1. StreamID
2. StreamRank
3. EndStationInterfaces
4. DataFrameSpecification
5. TrafficSpecification
6. UserToNetworkRequirements

7. InterfaceCapabilities

Groups “*StreamRank*” and “*TrafficSpecification*” should be in every “*Join*” request from Talkers. It is also recommended that they include the “*UserToNetworkRequirements*” and “*InterfaceCapabilities*”, since they would receive default values that would lower the optimization of the TSN flow performance. “*DataFrameSpecification*” should be also specified in a Talker group, unless the subgroup “*InterfaceCapabilities*” contains “*ActiveDestinationMAC*” and “*VlanStreamIdentification*” meaning that the endpoint is able to perform Stream Transformation (introduced in Section 3.2.1).

3.1.1.2 Listener

This group contains the following, providing details of what a listener user is expecting from a stream and its capabilities:

1. Listener’s network requirements.
2. TSN capabilities of the Listener’s network interface.

All this information, as the Talker’s, can be found in the subgroups below:

1. StreamID
2. EndStationInterfaces
3. UserToNetworkRequirements
4. InterfaceCapabilities

“*UserToNetwork*” requirements and “*InterfaceCapabilities*” should be included in any “*Join*” call in order to avoid the use of default values. “*StreamID*” and “*EndStationInterfaces*” must be included for every operation and for any endpoint type, both talkers and listeners.

3.1.1.3 Status

Status group provides the results of a TSN stream configuration, coming from the CNC. In the “*Fully Centralized*” it is received by the CUC. It is the CUC’s job to process this group data in order to compute the configuration needed for the endpoints involved. It groups the following:

1. StreamID
2. StatusInfo
3. AccumulatedLatency

4. InterfaceConfiguration

5. FailedInterfaces

For every talker or listener, the groups “*AccumulatedLatency*” and “*InterfaceConfiguration*” are different. Then, the separation between “*Status*” instances must be highlighted, so that it is splitted and treated independently in the CUC. “*StreamID*” and “*StatusInfo*” must be included in every response, both for “*Join*” or “*Leave*” events. “*AccumulatedLatency*” must be included in the “*Join*” responses, meanwhile “*InterfaceConfiguration*” is optional. Every time that a group is optional it is recommended to be included anyways, because it provides better details, thus a better network optimization. “*FailedInterfaces*” is an optional subgroup that specifies which and why network interfaces have failed in the configuration.

These three groups, “*Talker*”, “*Listener*” and “*Status*” are mapped onto a YANG module interface that the 802.1Qcc standard provides in one of their annexes. This module interface needs an implementation, that can be found in Annex B of this document. With this YANG module, it is possible to establish a NETCONF or RESTCONF communication between the CUC and the CNC.

3.1.2 Stream Transformation

In order to apply the TSN behavior to the network interface of the endpoints, the network needs a way to identify which streams take part in the TSN domain and which ones are best-effort. Commonly, the TSN streams are identified according to the VLAN ID (see Section 1.1.1). Nevertheless, it is possible that this VLAN ID differs from the endpoint and the network. To mitigate this difference, the groups “*InterfaceCapabilities*” and “*EndStationInterfaces*” are used. This distinction and adaptation between different VLAN IDs may be performed by the endpoint or the gateway switch.

When the endpoint sends the “*InterfaceCapabilities*” it informs the CNC about the Stream Transformation features of the device. Then, the CNC will return the “*InterfaceConfiguration*” subgroup, the stream identifier in the network. With this information, the endpoint can perform the stream transformation, if necessary. This transformation can be seen in **Figure 3.2**.

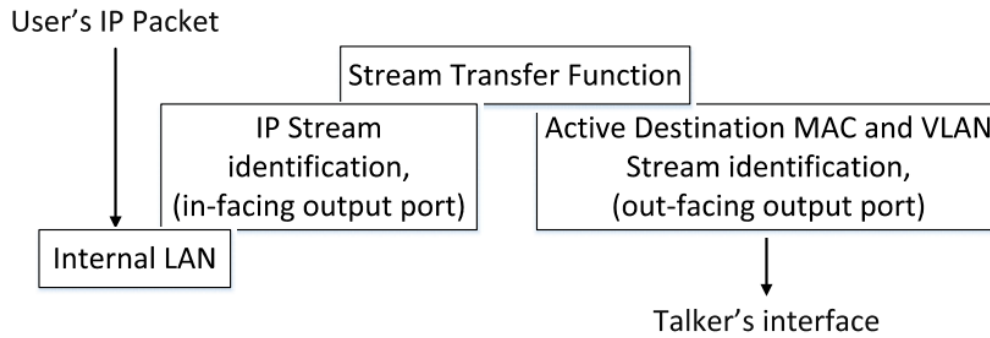


Fig. 3.2 Stream Transformation in an endpoint device [34].

This previous case supposes that the endpoint is not applying any specific VLAN or destination MAC address indicated by the TSN configuration. Due to the lack of this, the Stream Transfer Function uses IP header fields to identify the packets from the given stream and applies the desired values for VLAN ID and/or destination MAC address. In the same manner, the Stream Transfer Function (STF) is able to map the incoming VLAN or MAC in order to place the endpoint's values back for the upper layer application.

3.1.3 Example workflow

The 802.1Qcc standard provides an example case for the “*Fully Centralized*” architecture, showing the needed steps to establish a TSN stream and network configuration. This workflow only considers a single CUC and CNC. Next, each step will be listed in order to give the needed details to extrapolate ideas for our design. Some of the steps do not need to occur sequentially. Rather, they can be performed in a parallel fashion.

3.1.3.1 CUC discovers Talkers and Listeners

The protocol used to communicate the CUC with the endpoints is out of the scope of the 802.1Qcc standard. As we have already described, OPC-UA is a good candidate framework to handle this communication interface. The servers will be the endpoints and the client will be the CUC, that will perform requests against the endpoints' Address Space.

3.1.3.2 CUC reads each endpoint capabilities and requirements

This information includes TSN capabilities, the details about the device and its requirements in order to fill all the UNI fields. All these features must be handled in the Address Space of the endpoints, since the CUC will poll the data from it.

3.1.3.3 CUC generates the UNI instance data

When the CUC realizes it has enough information to perform a request to the CNC, it instantiates the UNI YANG module groups. All the logic that the CUC must perform in order to determine the moment of generating groups and sending to the CNC is out of the specification. It is noticeable that the specification defines the flow, the architecture, some of the interfaces between components and components' functionalities, but it is not specified how the implementation shall be done.

3.1.3.4 CNC discovers network elements

Simultaneously to the first three steps, the CNC discovers the network elements until it is able to describe the network topology, thanks to Link Layer Discovery Protocol (LLDP).

3.1.3.5 CNC reads TSN capabilities

Thanks to the remote management protocol, such as NETCONF, that is based on different YANG modules, specifically designed for TSN standards.

3.1.3.6 CUC sends the Talker and Listener groups to the CNC

The CUC sends a "Join" operation with the talker and listeners groups for a given stream. The subgroups "EndStationInterfaces" identify the endpoints in the network thanks to the MAC address, so that the CNC is able to identify them. The communication protocol between the CUC and the CNC is again out of the scope of the standard. However, our proposal will use RESTCONF, since it is the best option for the communication with network controllers. As commented in Section 2.1.4, RESTCONF uses NETCONF's principles, which brings all the advantages of using YANG modules. Since the information that the CUC does not configure the CNC, it is not needed to use the bigger datastore system that NETCONF provides.

3.1.3.7 CNC configures the TSN domain for the given stream

Assuming that steps in the Sections 3.1.3.4 and 3.1.3.5 are already performed and the CNC knows the network topology and details, the CNC has to compute the configuration of each network element. By this, the incoming TSN stream gets its reservation established in the network. CNC will base the configuration on the location of the endpoints involved and the VLAN ID, which will be provided back in the "Status" groups.

3.1.3.8 CNC configures the stream features

CNC uses its capabilities in order to enable the TSN features to the stream domain. This configuration will give all the real time requirements, if possible. If the configuration is unsuccessful, the following status groups will contain the details in the “FailedInterfaces” subgroup.

3.1.3.9 CNC returns the Status groups

When the CNC is finished with the device configuration, it generates the “Status” groups and gives them back to the CUC.

3.1.3.10 CUC configures each endpoint

If the CUC received the results of a bad “Status” group, it can retry the attempt after rechecking the sending Listener and Talker groups. Assuming that the CUC has received a successful response, it will generate the needed configuration for each endpoint that takes part. It will also send them through the OPC-UA interface, calling the configuration of the interface.

All the logic required to perform all of these steps are in the hands of the designer. This means that it can be implemented in different workflows. As a simple example, the first discovery step can be performed during several minutes before starting to instance the UNI groups. As another example, the prototype designed in this project knows the number of endpoints that participate in the network. Thus, it connects to both of them and proceeds with the pending steps.

CHAPTER 4. SOFTWARE AND HARDWARE FOR TSN ENDPOINTS

This chapter describes the endpoints. It introduces the basics of computer systems and how a Linux system can be managed to configure TSN requirements. This analysis is focused on the Intel i210 network card, used in this thesis' prototype.

The Intel i210 will be an important piece of our testbed, and therefore it deserves a detailed description. It is a network interface designed for real-time Ethernet applications; thus, it can be the ideal network interface for an endpoint in a TSN network [35]. It is a device that provides a hardware-based timestamping mechanism embedded in it. Thanks to this timestamping, it is able to support standards such as 802.1AS (gPTP, for clock synchronization with each network element) or different implementations found for 802.1Qbv and 802.1Qav specifications, all of them mentioned and treated in the other sections of this document.



Fig. 4.1 Intel i210 Network Card Interface [36].

There are several software implementations that take advantage of the physical timestamping of the network card. Some of the implementations come from Intel, others from open-source communities. The majority of them use similar processes and configuration tools, but with different final goals.

One of the objectives of this project is to extract the necessary information from these implementations. After understanding them, we can provide the endpoints with the capacity of applying dynamically the configuration provided by the CUC.

An interesting source found is a prototype from Kalycito, which is helpful in terms of introducing to the configuration of the endpoints' interfaces.

4.1 Kalycito – Linux and TSN using open62541

This project [37] uses two endpoints with Intel i210 network cards and a switch to build a small TSN domain. This domain is used by an OPC-UA open-source software written in C, establishing a Publish/Subscribe communication between endpoints. In order to give the endpoints enough tools to handle a TSN communication, the device must be prepared in the following aspects:

1. The device must have a real-time kernel. For Linux devices, this means that the kernel must be the *low latency* version, if possible. Also, the kernel contains several tools that are only found in latest versions.
2. The network interfaces must be configured to participate in the network. This process is done as with any other network interface (providing the IP address and network mask).
3. Configure the interfaces to support the TSN stream.
4. Start the OPC-UA Pub/Sub to send and receive data between endpoints over the TSN domain created.

Kalycito's project provides the introduction to the tools that will be used to set up the network cards for TSN stream requirements. In addition, it presents certain parts of the Linux operating system and its tools. All the details regarding the used tools will be given later in the document (see Section 5.3.2, Section 6.4 and Annex D), but before that it is interesting to introduce some concepts about Linux and its kernel.

4.2 Linux, kernel and Linux Network Stack

Linux is an operating system used by many developers and researchers. Since it is open-source, different communities, companies, associations and individuals take part on its constant enhancement. They all contribute in adding new features and optimizing the system. At first, it was designed for computers, but nowadays servers, routers, TVs, mobile phones and other embedded systems are Linux-based. For example, the Android operating system is based on the Linux Kernel.

4.2.1 General system architecture

A system architecture is represented in **Figure 4.2**, which is applicable to any operating system. In general terms, the goal of this architecture is to offer computing resources to applications and services.

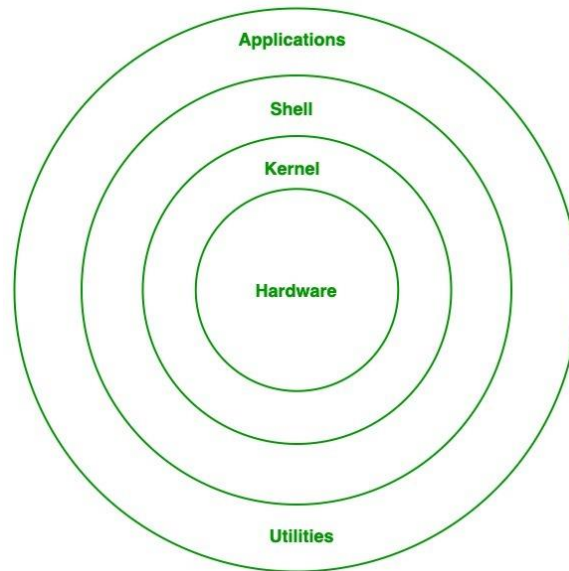


Fig. 4.2 Visual representation of a Linux system [38].

At the center of the figure there is the **Hardware**. Basically, it represents all the physical devices that build up the computer. For example, the Intel i210 network card belongs in the **Hardware**. The responsibility of the kernel is to virtualize all the hardware resources. Because of this, the processes can use them. Also, the kernel handles the conflicts that may occur in order to gain access to the resources.

In an upper layer there are the *Shell* and the *System Library*. Both of them offer an interface to the kernel, so that its complexity is hidden to upper layer users. These upper layer applications will use these interfaces to interact and use the kernel. Then, the kernel will communicate with the physical components. The applications mentioned are, basically, the *System Utilities* and the *Applications*. These applications are, in general, all the different software developed for the user of the system.

Having a look at this general architecture, we may guess that the kernel must perform the packet communication from the software to the NIC. Since this communication will need to comply with TSN requirements, the kernel packet management must be modified accordingly. Also, there has to be a way of identifying a given application traffic in the system kernel, because this specific

traffic needs to be handled. To have a good understanding of all the packet processing starting from an application to the network interface, we will take a deeper look in the Linux kernel. Afterwards, the Linux Networking Stack will be described.

4.2.2 Linux kernel

Linux kernel [39] is the most important piece of the operating system. In order to complement its definition, the kernel takes the responsibility of four different tasks:

1. Memory management
2. Process management
3. Device drivers
4. Security and system calls

Of the 28 million lines of code in the kernel, the majority of it goes to the drivers' section. This means that it ensures compatibility with a lot of different devices, but at the same time, its performance is a little worsened. As we saw in the previous section, the kernel is responsible for sending and receiving data through the network interfaces. Thus, we need to understand how it works to understand the modifications to handle TSN flows.

The kernel is responsible to control up to the *Transport Layer*, meaning that Ethernet and IP Layers are controlled and managed by the system kernel. A kernel user that needs to communicate data to the NIC will create and use a socket [40]. This socket is used to identify the application's traffic in the kernel, allowing to establish rules for this specific flow. Even though it is mentioned that the kernel works in Layers 2, 3 and 4, it is possible to implement higher layers, just by adding this feature to its source code.

To modify Linux kernel's behavior regarding different sockets, we have different interfaces. For instance, *netfilter* is an interface of the kernel used by the module *iptables* that manages the treatment of the packets at the network layer (NAT, firewall, etc.). There is also *netcat*, which enables the user to manage the Transport layer inside the kernel, so that it can be used by other applications. These two examples are configuration tools for both network and transport layers. Since TSN is implemented in the link layer, there is a need to know about the link layer tool, called *qdisc*.

Queuing Disciplines (qdisc) is a tool found between the IP Stack and the network card driver. It is the orchestrator of the traffic that is being sent to the driver's buffers [41]. With this tool it is possible to determine how packets will be sent through the network interface, being able to prioritize and conform the outgoing

traffic. Later in this document this tool will be recovered, since it is the best one to configure the endpoints' interfaces.

Linux kernel is being constantly updated. Its versions are based on an “*a.b.c.d*” format. Nowadays, the newest main version (*a*) is 5.0. This version already includes the tools that are used in this thesis to configure the TSN endpoints.

With this overview of the kernel, focused on the network stack, it is the best moment to describe how Linux processes the network packets.

4.2.3 Linux Network Stack

To develop the endpoints in our prototype, it is important to understand how a Linux system treats the packets before being sent to the NIC. **Figure 4.3** summarizes the process that a packet follows before arriving to the network interface.

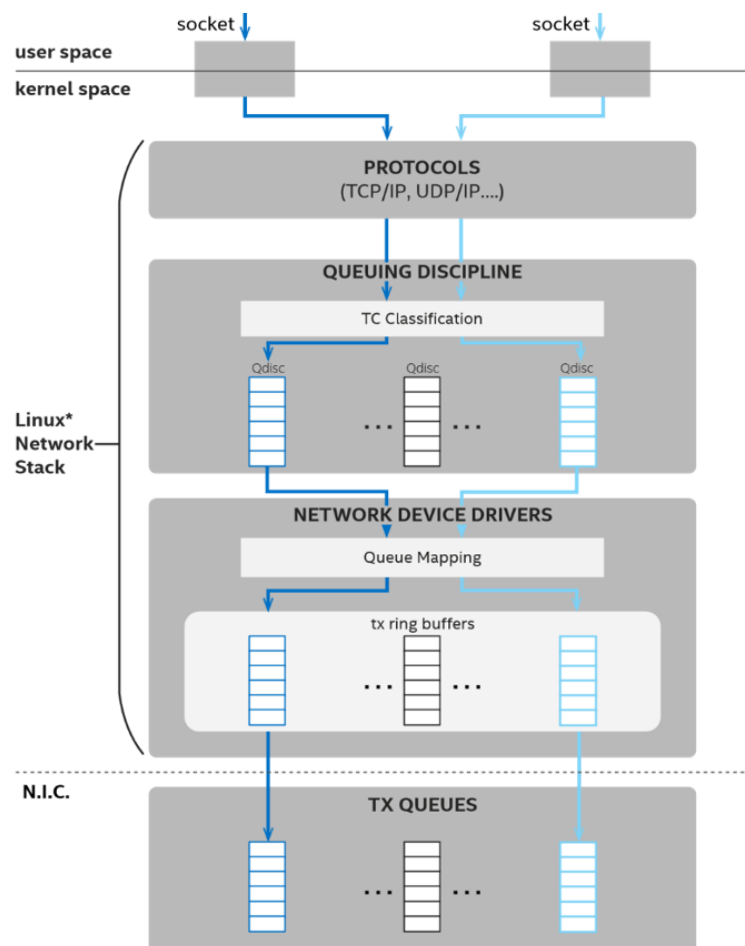


Fig. 4.3 Packet processing in a Linux system [35].

According to the configuration established in the kernel, the data pointed by the socket will be processed by the system and sent. In a more detailed view, the kernel treats the upper layers of the packet (transport and network) and right before the driver's buffers we have the Queuing Disciplines. These queues take care of prioritizing the order of the frames and, basically, act as schedulers/shapers to determine how the frames will be sent to the driver's buffers.

4.3 Intel i210 features related to TSN

In the Intel i210 overview webpage and its datasheet [42], we can find useful information to implement our prototype. Nowadays, there are different kernel interfaces that allow us to configure TSN parameters to some given outgoing traffic. These interfaces are called *TAPRIO* (Time Aware Priority), *CBS* (Credit Based Shaper) and *ETF* (Earliest TxTime First). All these interfaces can be configured with the *qdisc* tool. Actually, *qdisc* will hold different rules, so that traffic that coincides with them is accordingly scheduled [35]:

1. Earliest TxTime First: gives a scheduling temporally based on the transmission queues. It permits the application to determine the time to send the frame. It has a feature called *Launch Time* that gives the opportunity to give the exact time of transmission for TSN applications.
2. TAPRIO: this algorithm performs the scheduling based on priorities as defined in 802.1Qbv (see Section 1.2.1). It includes a way to parse the incoming different priorities to traffic classes perform traffic scheduling with them.
3. CBS: implementation of the 802.1Qav (see Section 1.2.2), acting as a bandwidth limiter and requiring the same parameters as specified in the previous section.

Understanding basic system architecture is recommended to contextualize how the TSN requirements can be satisfied in the endpoints from the thesis' prototype. In addition, tools provided since the release of kernel 5.0 can interact with Intel i210 buffers and become a TSN device. The prototype designed for the thesis needs to use these tools to apply the configuration coming from the CUC.

CHAPTER 5. ARCHITECTURE DESIGN

After the introduction to the basic concepts seen in the previous chapters, we will now define an architecture for the prototype of this project.

5.1 Statement of the objectives

The scenario of the project is the *Fully Centralized* architecture as defined by IEEE 802.1Qcc – clause 46. We will focus our development in the endpoints and the Centralized User Configuration, as illustrated in **Figure 5.1**.

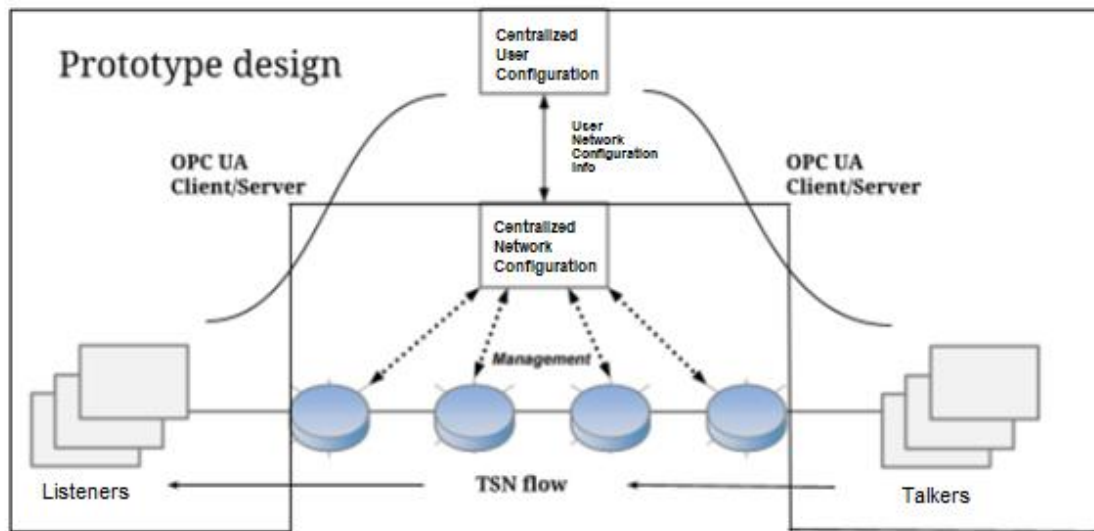


Fig. 5.1 Scenario of the project.

The figure shows that both endpoints will be OPC-UA-capable devices. These devices must be able to offer the needed information to the CUC, to get configured and transmit a data stream with TSN requirements. To understand how this will be performed, we need to analyze how each component will perform its functions.

The prototype consists of the Centralized User Configuration (CUC) and the endpoints. Both solutions can be provided to a future CNC development. When this prototype is integrated with a CNC, all the solution can be used as an SDN module in any current commercial controller.

Since the implementation of the CNC is not the focus of this thesis, the functionality expected in our prototype is focused on the following points:

1. The CUC polls the features required from the Talker and the Listener.
2. The TSN configuration is generated by the CUC and sent to the devices.
3. The Talker device applies the configuration to send traffic scheduled by the CUC logic.

Nevertheless, from the design of the fully centralized architecture described in the IEEE 802.1Qcc standard, the interface between CUC and CNC is also a requirement and needs to be implemented. Even though, it cannot be tested in this prototype, because the CNC prototype developed in parallel to this project is still in an early phase. the design is made considering the communication with the CNC. This allows an easier integration in the future.

5.2 Centralized User Configuration

This module will act as a client for the Centralized Network Configuration and for the endpoints. The CUC must manage the TSN requirements from endpoints and communicate them to the CNC. This is a meticulous process that has to be precise enough to be all TSN flows and requirements from all TSN endpoints.

Basically, the CUC has two interfaces:

1. OPC-UA Interface: responsible of polling the endpoint's requirements and give them back a configuration for their TSN interfaces. The requests must be designed in order to correctly point to the desired Address Space nodes. It will also communicate to the Talker the TSN configuration, so it can autoconfigure the device to perform the scheduled emission of data.
2. User to Network Interface (UNI): This interface exchanges the endpoint's requirements grouped by the CUC and the results obtained in the CNC. The YANG module provided in the standard has to be used. In addition, the most optimal communication protocol to handle the requests is RESTCONF, chosen for this prototype.

Figure 5.2 summarizes the aforementioned points about the CUC.

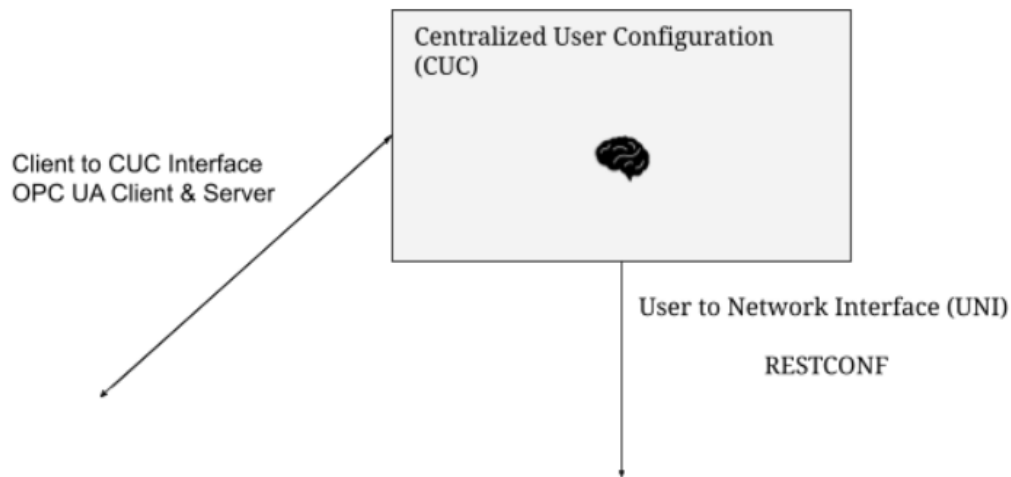


Fig. 5.2 CUC overview.

5.2.1 OPC-UA Client

This is the module of the CUC that is responsible of communicating with the Endpoints. On the one hand, it requests the OPC-UA servers the information that they have established as Talkers or Listeners. This information contains the enough data to generate UNI instance data in the Logic Unit Center (LUC). On the other hand, when the CNC has replied the CUC and the LUC has generated the endpoints' configurations, the OPC-UA Client will provide the endpoints all these computed TSN fields and trigger its establishment on the endpoints.

To successfully retrieve and send data with the OPC-UA servers, the Client must get to know the organization of the objects in the Address Space. By this, it can identify each node successfully. The definition of the Address Space will be described in the following section.

5.2.2 Logic Unit Center

This part of the CUC generates YANG instances for those TSN flows that have configuration for at least one Talker and one Listener. This means that the LUC is responsible of managing the different TSN stream details and treat them independently. This module uses the RESTCONF Client to push the configuration and receive a response from the CNC. With it, it will generate the Endpoint's configuration. This configuration consists in the Gate Control List (shown in **Figure 1.8**) and shaping mechanisms.

5.2.2.1 UNI Instance details

The UNI's interface structure is defined previously in this document (see Section 3.1.1), but with a more general overview. The YANG module definition can also be found in Annex B, which uses the YANG interface from the IEEE standard to provide a YANG module ready to be used between clients and servers.

This section is aimed to detail the origin and description for all different fields that conform the UNI requests. It defines every property that needs to be generated in the CUC.

The “*request*” container contains the fields below:

1. stream-id: unique identifier for the TSN flow. It is the key value for all main groups, so it is mandatory in all of them. It has to be compliant with a regular expression, in which the first 48 bits are the Talker's MAC and the last 16 correspond to a pseudo randomly generated UID.
2. stream-rank:rank: value that determines if the stream is considered as high priority (0) or best-effort (1). Included in the Talker group.
3. end-station-interfaces: specified by both endpoints
 - a. mac-address: interface from which the TSN flow will be sent or received.
 - b. interface-name: name of the TSN interface.
 - c. data-frame-specification: list that may contain the following fields with stream identification goals and the use of redundancy.
 - i. index: identifier of the list element.
 - ii. ieee802-mac-addresses: gives the MAC address of the talker and the possible listeners.
 - iii. ieee802-vlan-tag: specifies the VLAN that the talker uses for the stream.
 - iv. ipv4-tuple: if the endpoints do not know Ethernet details, they can use this field to provide their IPv4 address.
 - v. ipv6-tuple: same case as the previous field, but providing an IPv6 address instead.
4. traffic-specification: this fields are specified by the talker
 - a. interval: contains two fields, numerator and denominator. By dividing them, the number of seconds for a time window is obtained.

For example, knowing a numerator is equal to four and denominator equal to five, it means that the interval is 0.8 seconds long.

- b. max-frames-per-interval: number of frames that will be sent during an interval.
- c. max-frame-size: maximum size of a frame in bytes.
- d. transmission-selection: specifies which shaper to use. The choices can be found in **Figure 5.3**, where CBS is one of the options. In addition to CBS, Strict priority bases the frame transmission purely based on queue priorities and Enhanced Transmission Selection assigns a bandwidth percentage to every traffic class.

Transmission selection algorithm	Identifier
Strict priority (8.6.8.1)	0
Credit-based shaper (8.6.8.2)	1
Enhanced Transmission Selection (ETS) (8.6.8.3)	2
Reserved for future standardization	3–254
Vendor-specific Transmission Selection algorithm value for use with DCBX (D.2.8.8)	255
Vendor-specific	A four-octet integer, where the most significant 3 octets hold an OUI or CID value, and the least significant octet holds an integer value in the range 0–255 assigned by the owner of the OUI or CID.

Fig. 5.3 Transmission selection choices [43]

- e. time-aware:
 - i. earliest-transmit-offset: minimum amount of time from the start of an interval in which the device is able to transmit the frames. It is expressed in nanoseconds.
 - ii. latest-transmit-offset: maximum amount of time from the start of the interval in which the frames can be sent before being considered as deprecated in time. It is expressed in nanoseconds.
 - iii. jitter: maximum difference in time between the previous offsets and the network time (gPTP). It comes from the network synchronization jitter and application synchronization with system clock regarding packet transmission.

5. user-to-network-requirements: the listener may only specify the required latency for the traffic. The number on seamless trees can only be provided by the Talker.
 - a. num-seamless-trees: number of independent paths for every Listener.
 - b. max-latency: maximum value of latency required for the TSN flow, expressed in nanoseconds.
6. interface-capabilities: specified by both endpoints
 - a. vlan-tag-capable: boolean value that indicates if the endpoint supports VLAN tagging.
 - b. cb-stream-iden-type-list: list for the type of Stream Identification, formed by the values OUI/CID + Type (see **Figure 5.4**).

OUI/CID	Type number	Stream identification function	Controlling parameters
00-80-C2	0	Reserved	—
00-80-C2	1	Null Stream identification (6.4)	9.1.2
00-80-C2	2	Source MAC and VLAN Stream identification (6.5)	9.1.3
00-80-C2	3	Active Destination MAC and VLAN Stream identification (6.6)	9.1.4
00-80-C2	4	IP Stream identification (6.7)	9.1.5
00-80-C2	5–255	Reserved	—
other	—	Defined by entity owning the OUI or CID	—

Fig. 5.4 Stream Identification types [44].

- c. cb-sequence-type-list: list for the encoding and decoding types of the *sequence number* for the Ethernet frames (see **Figure 5.5**).

OUI/CID	Type number	Sequence encode/decode method
00-80-C2	0	Reserved
00-80-C2	1	R-TAG (7.8)
00-80-C2	2	HSR sequence tag (7.9)
00-80-C2	3	PRP sequence trailer (7.10)
00-80-C2	4-255	Reserved
Other	—	Defined by entity owning the OUI or CID

Fig. 5.5 Sequence encoding and decoding method types [45].

After reviewing the fields for the *Talker* and *Listener* groups, we will now focus on the *Status* group fields. The CUC has to parse this last group and compute a correct configuration for the endpoints involved.

1. status-info: provides information about the status of the configuration for a given stream.
 - a. talker-status: enumeration that may specify “No Talker”, “Talker ready” or “Failed”.
 - b. listener-status: another enumeration that may be “Listener ready”, “Failed” or “Partial failed”, which means that in scenarios with more than a listener, some of them succeeded and some of them did not.
 - c. failure-code: error code according to **Figure 5.6**.

Failure Code	Description of cause
1	Insufficient bandwidth
2	Insufficient Bridge resources
3	Insufficient bandwidth for traffic class
4	StreamID in use by another Talker
5	Stream destination_address already in use
6	Stream preempted by higher rank
7	Reported latency has changed
8	Egress Port is not AVB capable ^a
9	Use a different destination_address (i.e., MAC DA hash table full)
10	Out of MSRP resources
11	Out of MMRP resources
12	Cannot store destination_address (i.e., Bridge is out of MAC DA resources)
13	Requested priority is not an SR Class (3.259) priority
14	MaxFrameSize [item a) in 35.2.2.8.4] is too large for media
15	msrpMaxFanInPorts [item f) in 35.2.1.4] limit has been reached
16	Changes in FirstValue, other than AccumulatedLatency, for a registered StreamID
17	VLAN is blocked or filtered on this egress Port ^b
18	VLAN tagging is disabled on this egress Port (untagged set)
19	SR class priority mismatch
20	Enhanced feature cannot be propagated to original Port
21	MaxLatency exceeded
22	Nearest Bridge cannot provide network identification for stream transformation
23	Stream transformation not supported
24	Stream identification type not supported for stream transformation
25	Enhanced feature cannot be supported without a CNC

Fig. 5.6 Error types on the Status group [46].

2. accumulated-latency: integer value that represents the value in picoseconds of the worst-case latency for a stream. The talker receives the maximum delay over all the listeners. If received by a listener, only the latency to this same listener is given.
3. group-interface-configuration:interface-list: list that contains different configurations for every combination of MAC address and interface-name of a device.
 - a. group-interface-id: tuple with the MAC address and the interface-name.

b. config-list:

- i. index: identifier for the element of the list
- ii. ieee802-mac-address: source and destination MAC addresses. Tend to be the same as the request, even though the destination MAC address may be turned to multicast.
- iii. ieee802-vlan-tag: specifies the VLAN tag that will be used in the network for this TSN stream.
- iv. ipv4-tuple: IPv4 address of the received to use for Stream Identification purposes.
- v. ipv6-tuple: analogous of the previous field for a IPv6 address.
- vi. time-aware-offset: instant if time determined by the CNC. The Talker must transmit its data at this precise moment. This value is between the “earliest-transmit-offset” and “latest-transmit-offset” given by the same Talker (see Section 5.2.2.1.4.e).

5.2.2.2 Gate Control List

The CUC also has a logical part aimed to compute the entries of the GCL [16] of the TAS (see Section 1.2.1). To automate the GCL creation, the CUC must manage every variable that may be involved in the scheduling of the endpoints.

Because of the complexity derived from the calculation of the GCL, a simplified sample will be shown, in order to describe the process that the UNI needs to take to create this field.

Traffic specification Traffic 1:

- Interval: 100 ms
- Max frames per interval: 1
- Max frame size: 500 bytes
- Earl/Lat transmit offsets: 10/20 ms
- Jitter: 1 ms

Traffic specification Traffic 2:

- Interval: 1 s
- Max frames per interval: 2
- Max frame size: 500 bytes
- Earliest/Latest transmit offsets: 250-300 ms
- Jitter: 1 ms

A sample response from the CNC for all these values could be the one below. Note that both traffics get a VLAN priority assigned and a transmit time. If we compare them to the earliest/latest transmit offsets, all of them can transmit at their correct moment.

1. time-aware-offset traffic 1: 10 ms
2. VLAN priority traffic 1: 6
3. Time-aware-offset traffic 2: 250 ms
4. VLAN priority traffic 2: 5

With these values combined with the ones obtained from the Talker, the CUC must be able to compute the Time Aware Scheduler, as shown in **Figure 5.7**.

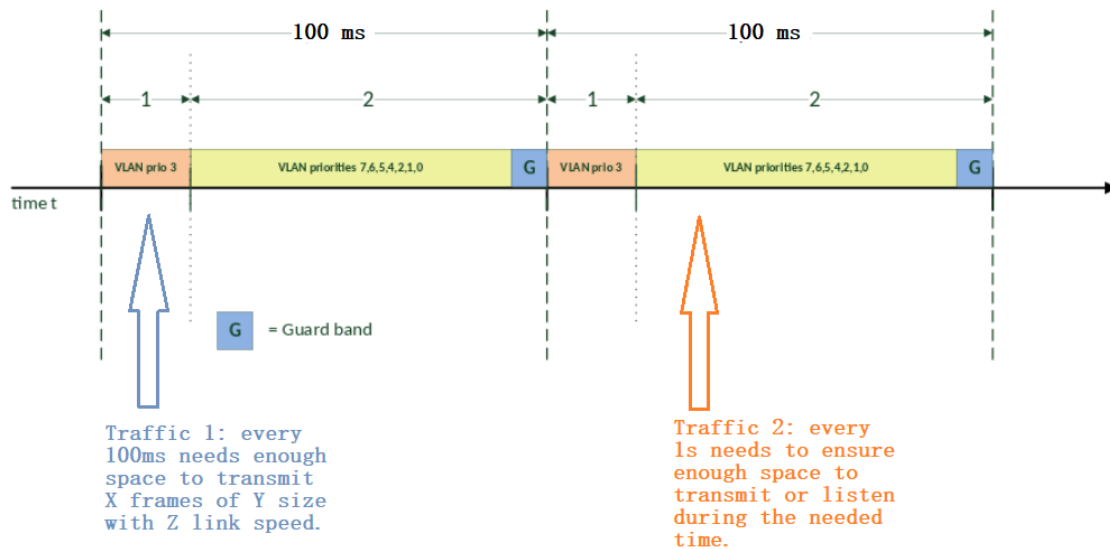


Fig. 5.7 Time aware shaper example.

1. The time slots in which the TSN traffic can be transmitted coincides with the “time-aware-offset” that the CNC is providing to the talker. This restriction would imply that the second traffic “Traffic 2” is splitted in another priority. By this, both TSN traffics are able to transmit at the exact “time-aware-offset” the CNC has proposed.
2. There is a need to specify how long a gate status will be held. In other words, each GCL value needs to provide the amount of time that the gates will have a state mapped (either transmit or idle).

3. There is a need to have synchronization regarding the instant of time in which the TAS cycle begins. If not, the “*time-aware-offset*” cannot be placed correctly inside. This can be done easily by matching the “*interval*” time given from the Talker with the cycle duration of the traffic scheduler.

In conclusion, the CUC needs to specify the following for a GCL: TAS interval duration and GCL. This last element includes the status of the gates for each priority and the amount of time. **Table 5.1** shows an example of a GCL that complies with the aforementioned requirements.

Table 5.1 Example GCL

	0	1	2	3	4	5	6	7
g0 (10ms)	1	1	1	1	1	0	0	1
g1 (1ms)	0	0	0	0	0	0	1	1
g2 (39ms)	1	1	1	1	1	0	0	1
g3 (1ms)	0	0	0	0	0	1	0	1
g4 (49ms)	1	1	1	1	1	0	0	1

1. g0: best-effort traffic, since we know frames from priorities 5 and 6 will not be transmitted during this time. Open from 0 to 10ms.
2. g1: priority 6 transmission interval. Open from 10 to 11 ms.
3. g2: best-effort traffic, open from 11 to 50 ms.
4. g3: priority 5 transmission. Open from 50 to 51 ms.
5. g4: best-effort traffic, open until the end of the cycle.

An alternative, for example, is that a CBS assigns a higher throughput and weight to the priority 5 queue. By doing this, that particular traffic would be prioritized against others. Other approaches can also comply with the requirements.

It is important to note that the computation of the GCL is not simple. The IEEE provides a paper with guidelines on how to implement the computation [47]. Since our focus is on the design of the prototype and not on the scheduling, a simple GCL computation is implemented in the prototype.

5.2.3 RESTCONF Client

As seen in Section 2.1.4, RESTCONF relies on the HTTP client/server architecture. This forces the same configuration data to not be splitted in different requests, since there is no datastore other than the *running datastore*. The datastore directly sets the configuration to the device. This means that our RESTCONF client shall send only fully prepared requests, including the talker and listener's information all in once. In addition, to enhance the data control (one of the lacks of RESTCONF), different assertions should be checked by the LUC before sending a request:

1. Check the information coming from the OPC-UA client and parse all the fields.
2. Use the parsed fields in order to build an instance YANG module (described in Annex B).
3. Use a YANG validator to check the instanced data against the YANG module definition.

Apart from preparing the data to be send later to the CNC, the RESTCONF client also may request configuration from it. Then, it needs to have an interface with the *Logic Unit Center* to perform any kind of request. Keep in mind that this interface has been designed and implemented. However, it is not used by the final prototype, because a prototype of a CNC is under development.

5.2.3.1 HTTP/2 details

Since RESTCONF is implemented over HTTP/2 [48], it requires *HTTPS Mutual Authentication*. This means that a *TLS Handshake* authenticating both the server and the client must be performed. In order to do this, we need to build a root certificate for the RESTCONF server and different certificates signed by it. This client certificates can be used by clients.

In addition, since requests are performed as a RESTful service, API endpoints should be defined by the CNC to give the clients the best detail about how to use them. In the current state of the CNC, these endpoints are defined as follows.

1. "*https://{host}:{port}/restconf/data/ieee802-dot1q-tsn-types-upc-version:tsn-uni*"
 - POST method
 - Body type: JSON
 - Content expected: instance of the YANG module referenced in Annex B.
 - Response: HTTP status code

2. *"https://{host}:{port}/restconf/data/ieee802-dot1q-tsn-types-upc-version:tsn-uni"*

- Same endpoint used, but with a GET method.
- Response type: JSON
- Response content: current status groups from the UNI, including TSN information to configure the endpoints.

As previously mentioned, at the moment of finishing this document, there is no CNC prototype, since it is under development. This means that all the API endpoints cannot be fully provided and the generation of the information from the CNC is mocked. For this project, the CUC uses sample values that can later be used to configure the endpoints as if they were real.

5.3 Endpoints

Our endpoints are software-based, running on common PCs equipped with an Intel i210 network interface card. On the one hand, the endpoints need to establish their features and requirements in their Address Spaces to offer them to the CUC. On the other hand, they need to receive the information from the CUC and perform an automated TSN configuration of its interface. It has two virtual interfaces against the rest of the setup, as follows:

1. OPC-UA Interface: the other side of the CUC OPC-UA Interface. This device needs to prepare an optimal Address Space in order to publish all the requirements for the TSN stream. Also, a method to configure itself with the incoming configuration from the CUC, when the endpoint is a Talker. If it is a Listener, there is a need to set up the traffic receiver in order to evaluate the TSN flow.
2. TSN Interface: the interface that will be used to transmit or receive the TSN stream, having its performance adapted to the configuration received by the CUC.

Both interfaces can be seen in **Figure 5.8**.

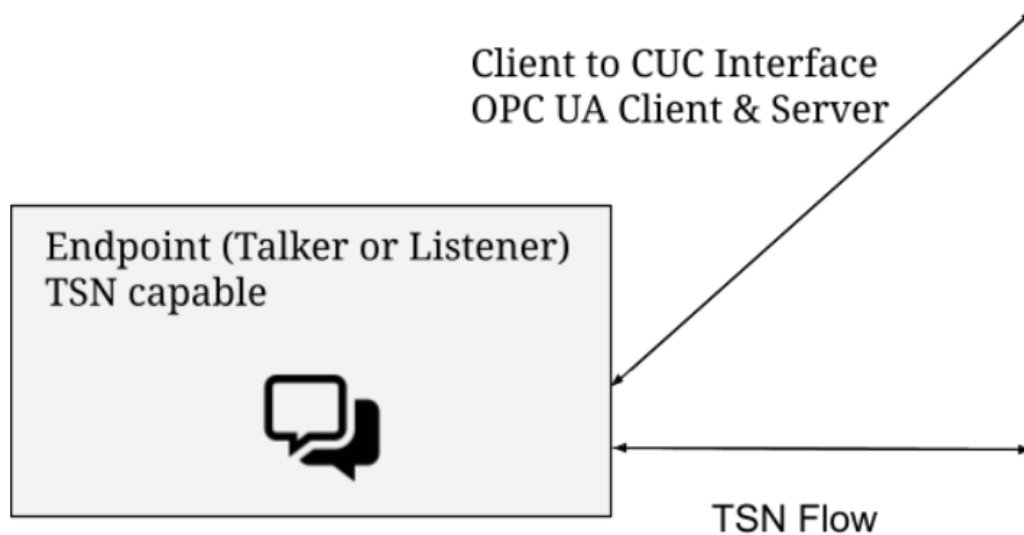


Fig. 5.8 Endpoint overview.

The workflow that the endpoints follow to obtain configuration and apply it is defined below.

1. On wake up, set the interface details and flow requirements at the Address Space of an OPC-UA instanced server.
2. Flush any previous TSN configuration that may be applied to the TSN Interface and load the actual ones.
3. Enable write permissions on the configuration variables coming from the CUC, so that they can be under the endpoint's control.
4. Expose a *method* node to trigger the applying of the new earned configuration or start a receiver, depending on endpoint's nature.
5. TSN Flow will be established.

5.3.1 Address Space definition

Most of the variables that need to be in the OPC-UA servers will be basic well-known types. However, the aforementioned *method* node will be used to apply the received configuration. All the properties described below make a direct reference to the different fields on the UNI defined in Section 3.1.1.

5.3.1.1 *EndpointFeatures*

It contains the device's features and requirements used both by a talker and a listener.

1. `endpointType`: String that determines if the endpoint is Talker or Listener.
2. `identificationType`: UInt32 reserved for future use for Stream Identification.
3. `interfaceName`: String with the name of the TSN interface of the endpoint.
4. `macAddress`: String that contains the MAC address of the TSN interface.
5. `maxDelay`: UInt32 that expresses the maximum latency required in nanoseconds. Reserved for future use.
6. `redundancy`: Boolean expressing the requirement of path redundancy. Reserved for future use.
7. `streamId`: String that uniquely identifies the TSN flow.
8. `streamIdTypes`: String reserved for future use for Stream Identification.
9. `vlanCapable`: Boolean to determine if the endpoint manages VLANs.

The talkers, in addition, have extra variables in *EndpointFeatures* in order to specify the traffic to be transmitted:

1. `earliestTransmitOffset`: UInt32 expressed in nanoseconds.
2. `intervalDenominator`: UInt32 that gives the numerator of a rational value to express the time interval in seconds.
3. `intervalNumerator`: UInt32 that gives the denominator of the same previous value.
4. `jitter`: UInt32 expressed in nanoseconds.
5. `latestTransmitOffset`: UInt32 expressed in nanoseconds
6. `maxFrameNumber`: UInt32 during the interval
7. `maxFrameSize`: UInt32 of the TSN frames.
8. `priority`: UInt32 that states the priority for the given TSN stream.

9. transmissionSelection: UInt32 identifying the transmission selection algorithm.

5.3.1.2 *TSNInterfaceConfig*

In addition, as stated in the previous lines, the Address Space also needs an object to store the configuration from the CUC and trigger its setting. The variables that will be used in that object, called *TSNInterfaceConfig*, are as follows:

1. LaunchConfig: Method that triggers the setting of the TSN configuration and the flow transmission.
2. gclGates: [UInt32] array of gate status. It indicates which priority is able to dequeue data.
3. gclGatesTimeInterval: [UInt32] time associated to each state in the previous variable.
4. interval: UInt32 expressing the interval duration of the scheduler.
5. latency: UInt32 maximum latency among all listeners. Reserved for future use
6. vlanIdValue: UInt32 that determines the VLAN ID to be used.

5.3.1.3 *PublishObject*

Moreover, for testing purposes, talkers need to have a changing variable that will be published through the TSN Flow to listeners. The object is called *PublishObject* and it only has one variable, which its content defines the traffic specification of the emitting TSN flow based on OPC-UA Publish/Subscribe:

1. publishObjectData: String

5.3.1. *SubscribeClientObject*

Finally, the listeners need an entry point on their Address Spaces in order to start a subscription or any other data receiver. To do that, the following object is present in the Address Space. It is called *SubscribeClientObject* and contains only a variable that will trigger an OPC-UA Client instance to Subscribe for the talker's changing variable.

If the TSN flow is not an OPC-UA based communication, the suitable receiver is powered on, such as an iperf¹ server.

1. initSubscription: Method
2. interval: UInt32 (only used by OPC-UA Subscription)

5.3.2 Interface configuration

Interface configuration will be an automated process that will parse the new CUC configuration and apply it on the specified interface. The details of this automation can be found in Section 6.4 and Annex D.

After making definition on how both components are, it is a good moment to introduce the overall architecture. The CUC and the endpoints are contained in **Figure 5.9**, representing the logical view of the prototype.

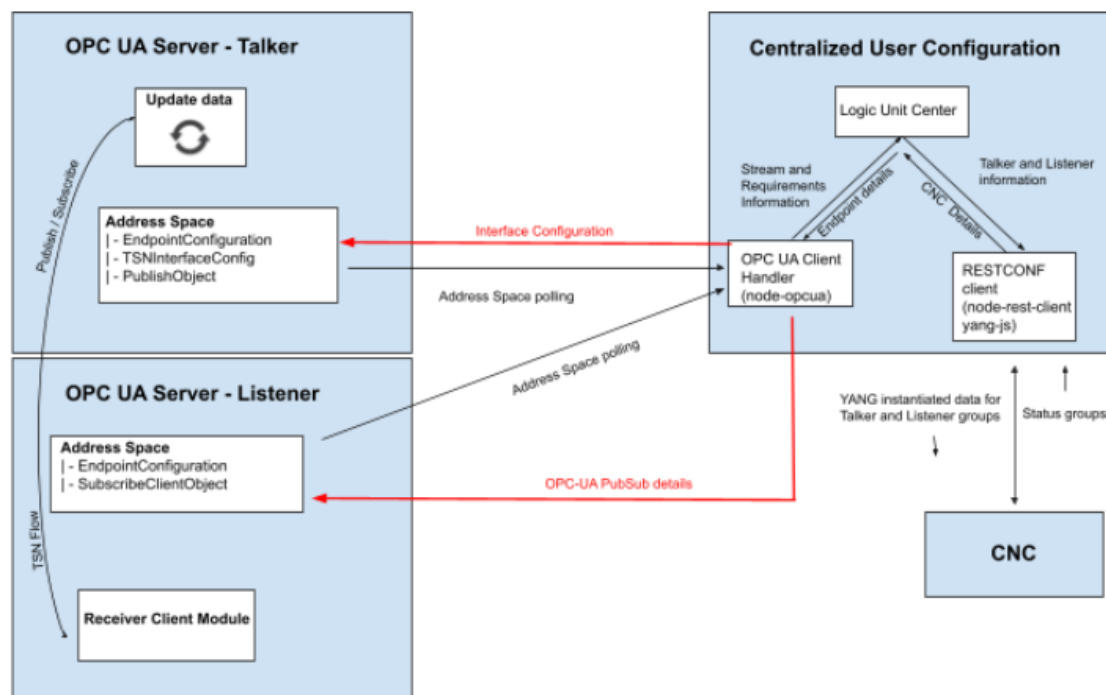


Fig. 5.9 Overview of the architecture of the prototype

¹ Iperf is a client-server-based tool that helps to determine the available bandwidth between participants. In addition, it provides jitter details. It is used to test the TSN configuration because it constantly generates a given bandwidth. This means that packets will always be ready to be sent, a different approach than the OPC-UA Publish/Subscribe traffic.

CHAPTER 6. IMPLEMENTATION

This chapter presents the process to set up the developed environment. It begins with the network, followed by the time synchronization between participants in the prototype. In addition, the instance of the used Address Spaces and an endpoint sample configuration are shown. To integrate all of these parts, a general prototype performance is presented in the last section of this chapter.

6.1 LAN

This step consists in the connection between elements of the prototype: two endpoints (Linux PCs), the CNC/CUC (another PC) and a SoC-e MTSN switch [49]. **Figure 6.1** illustrates the network topology, together with additional data (IP and MAC addresses, interfaces, ports).

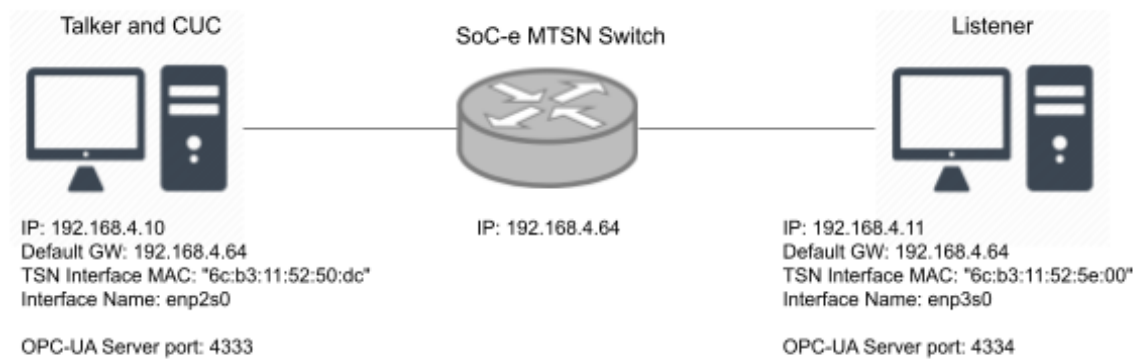


Fig. 6.1 LAN created for the prototype.

Endpoints (Linux systems) need to get their interfaces configured in order to be in the network, as any other device taking part in a LAN. Note that the CUC and the Talker are located in the same physical machine. This decision is taken only for resource optimization, since the CUC can be set up in any other device connected to the network.

6.2 Time synchronization between Endpoints and the TSN switch

In previous works [50] [51], there were several unsuccessful attempts to synchronize the clocks of the Intel i210 NICs and the SoC-e MTSN switches. We now know that the switches' implementation of the 802.1AS standard comes from the "*linuxptp*" [52] package. This is a relevant information, since our endpoints can run the same software, thus minimizing compatibility issues.

6.2.1 Installation and use of *linuxptp*

From the README of the package:

1. Using git, clone the repository from <http://git.code.sf.net/p/linuxptp/code> *linuxptp*.
2. Open a terminal on the root of the cloned folder and type the command *make* (a known tool to specify the compiling process of C programs). Call the command *make install* in order to bring the tools *ptp4l* and *phc2sys* to the system path. As an alternative, it is possible to install *linuxptp* by typing the command *sudo apt install linuxptp*.
3. To initialize it, the command
4. “*sudo ptp4l -i enp3s0 -f configs/gPTP.cfg --step-threshold=1 -m*”, where “-i” gives the interface going to be synchronized and the flag “-m” gives output to the terminal. The flag “-f” specifies the configuration file, getting the standard gPTP parameters, as shown in **Figure 6.2**.

```
#
# 802.1AS example configuration containing those attributes which
# differ from the defaults. See the file, default.cfg, for the
# complete list of available options.
#
[global]
gmCapable          1
priority1          248
priority2          248
logAnnounceInterval 0
logSyncInterval    -3
syncReceiptTimeout  3
neighborPropDelayThresh 800
min_neighbor_prop_delay -20000000
assume_two_step     1
path_trace_enabled  1
follow_up_info      1
ptp_dst_mac         01:80:C2:00:00:0E
network_transport   L2
delay_mechanism     P2P
transportSpecific    1
```

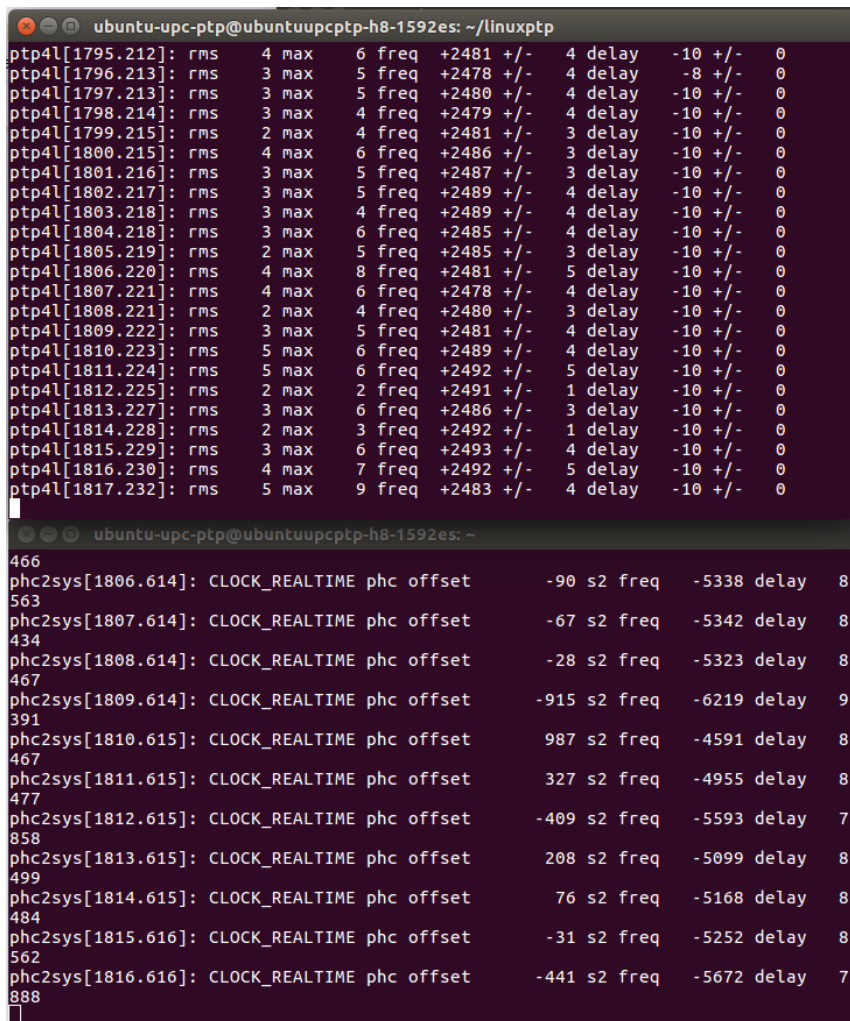
Fig. 6.2 gPTP.cfg file.

If we enable the 802.1AS in both endpoints, their network interfaces become synchronized. For more details on the use of the “*ptp4l*” tool, check the manual

page [53]. To apply this network interface clock synchronization to the actual system clock, the *phc2sys* tool [54] is used with the following command:

```
“sudo phc2sys -s enp3s0 -c CLOCK_REALTIME --step_threshold=1 \ --transportSpecific=1 -w -m -l 7”
```

The “-s” flag indicates the master clock and “-c” will be the slave, meaning that the system clock will map its value from the network interface. The “*transportSpecific*” flag is needed to communicate with *ptp4l* when it uses gPTP. “-w” means a wait flag to wait for *ptp4l* values. The last value, “-l”, is the log level, set to the maximum.



```

ubuntu-upc-ptp@ubuntuupc-ntp-h8-1592es: ~/linuxptp
ptp4l[1795.212]: rms 4 max 6 freq +2481 +/- 4 delay -10 +/- 0
ptp4l[1796.213]: rms 3 max 5 freq +2478 +/- 4 delay -8 +/- 0
ptp4l[1797.213]: rms 3 max 5 freq +2480 +/- 4 delay -10 +/- 0
ptp4l[1798.214]: rms 3 max 4 freq +2479 +/- 4 delay -10 +/- 0
ptp4l[1799.215]: rms 2 max 4 freq +2481 +/- 3 delay -10 +/- 0
ptp4l[1800.215]: rms 4 max 6 freq +2486 +/- 3 delay -10 +/- 0
ptp4l[1801.216]: rms 3 max 5 freq +2487 +/- 3 delay -10 +/- 0
ptp4l[1802.217]: rms 3 max 5 freq +2489 +/- 4 delay -10 +/- 0
ptp4l[1803.218]: rms 3 max 4 freq +2489 +/- 4 delay -10 +/- 0
ptp4l[1804.218]: rms 3 max 6 freq +2485 +/- 4 delay -10 +/- 0
ptp4l[1805.219]: rms 2 max 5 freq +2485 +/- 3 delay -10 +/- 0
ptp4l[1806.220]: rms 4 max 8 freq +2481 +/- 5 delay -10 +/- 0
ptp4l[1807.221]: rms 4 max 6 freq +2478 +/- 4 delay -10 +/- 0
ptp4l[1808.221]: rms 2 max 4 freq +2480 +/- 3 delay -10 +/- 0
ptp4l[1809.222]: rms 3 max 5 freq +2481 +/- 4 delay -10 +/- 0
ptp4l[1810.223]: rms 5 max 6 freq +2489 +/- 4 delay -10 +/- 0
ptp4l[1811.224]: rms 5 max 6 freq +2492 +/- 5 delay -10 +/- 0
ptp4l[1812.225]: rms 2 max 2 freq +2491 +/- 1 delay -10 +/- 0
ptp4l[1813.227]: rms 3 max 6 freq +2486 +/- 3 delay -10 +/- 0
ptp4l[1814.228]: rms 2 max 3 freq +2492 +/- 1 delay -10 +/- 0
ptp4l[1815.229]: rms 3 max 6 freq +2493 +/- 4 delay -10 +/- 0
ptp4l[1816.230]: rms 4 max 7 freq +2492 +/- 5 delay -10 +/- 0
ptp4l[1817.232]: rms 5 max 9 freq +2483 +/- 4 delay -10 +/- 0

ubuntu-upc-ptp@ubuntuupc-ntp-h8-1592es: ~
466 phc2sys[1806.614]: CLOCK_REALTIME phc offset -90 s2 freq -5338 delay 8
563 phc2sys[1807.614]: CLOCK_REALTIME phc offset -67 s2 freq -5342 delay 8
434 phc2sys[1808.614]: CLOCK_REALTIME phc offset -28 s2 freq -5323 delay 8
467 phc2sys[1809.614]: CLOCK_REALTIME phc offset -915 s2 freq -6219 delay 9
391 phc2sys[1810.615]: CLOCK_REALTIME phc offset 987 s2 freq -4591 delay 8
467 phc2sys[1811.615]: CLOCK_REALTIME phc offset 327 s2 freq -4955 delay 8
477 phc2sys[1812.615]: CLOCK_REALTIME phc offset -409 s2 freq -5593 delay 7
858 phc2sys[1813.615]: CLOCK_REALTIME phc offset 208 s2 freq -5099 delay 8
499 phc2sys[1814.615]: CLOCK_REALTIME phc offset 76 s2 freq -5168 delay 8
484 phc2sys[1815.616]: CLOCK_REALTIME phc offset -31 s2 freq -5252 delay 8
562 phc2sys[1816.616]: CLOCK_REALTIME phc offset -441 s2 freq -5672 delay 7
888

```

Fig. 6.3 Captures from *ptp4l* and *phc2sys* processes

Figure 6.3 shows two captures from the terminals. The first capture shows *ptp4l* syncing the clock with the GrandMaster clock, which is the Talker/CUC device. The offset values between clocks ranges from two to five nanoseconds. The second capture shows messages from the *phc2sys* process. It is noticeable that sometimes it loses the synchronization, reaching values of offset close to a millisecond. This may be a problem. Other computers where we tested the same

set up do not provide these peaks, meaning that the reason may be found in the physical hardware of the device. Specifically, on the Listener machine. It is important to remind these peaks during the results of the tests reported in Chapter 7.

6.2.2 Interconnection with the TSN switches

To enable 802.1AS on the TSN switches, the configuration webpage must be used. We can login using *admin* as user and *soc-e* as password. Then, if we navigate to the *Advanced* mode and click on the *Synchronization* tab, we can see the screenshot shown in **Figure 6.4**.

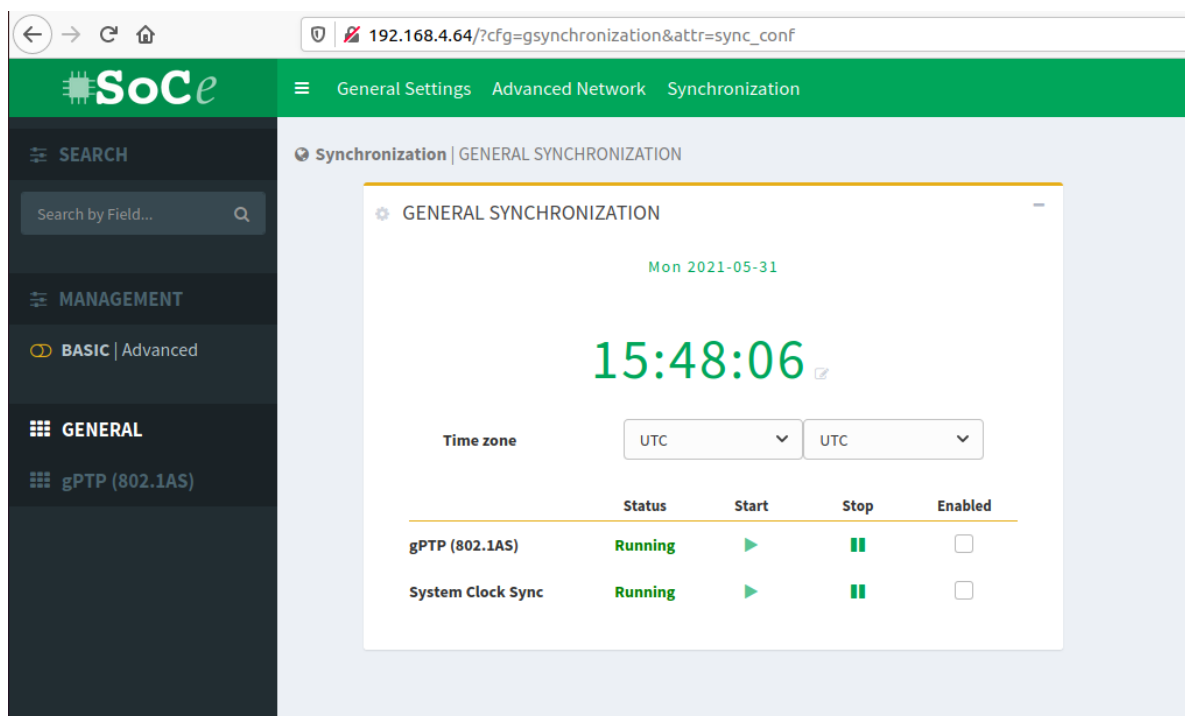


Fig. 6.4 SoC-e MTSN 802.1AS synchronization tab.

After making sure that the *linuxptp* daemons in the TSN switches and the endpoints are running, they will synchronize. **Figure 6.5** is a Wireshark capture that shows the interface *enp2s0* from one of the endpoints.

278	11.048234458	System-0_08:a0	LLDP_Multicast	PTPv2	68 Path_Delay_Resp_Follow_Up Message
279	11.077043528	Shenzhen_52:5e:00	LLDP_Multicast	PTPv2	58 Sync Message
280	11.077250904	Shenzhen_52:5e:00	LLDP_Multicast	PTPv2	90 Follow_Up Message
281	11.202087140	Shenzhen_52:5e:00	LLDP_Multicast	PTPv2	58 Sync Message
.....					
▶ Frame 277: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface 0					
▶ Ethernet II, Src: System-0_08:a0 (70:f8:e7:d0:08:a0), Dst: LLDP_Multicast (01:80:c2:00:00:0e)					
▼ Precision Time Protocol (IEEE1588)					
▶ 0001 = transportSpecific: 0x1					
.... 0011 = messageId: Path_Delay_Resp Message (0x3)					
.... 0010 = versionPTP: 2					
messageLength: 54					
subdomainNumber: 0					
▶ flags: 0x0200					
▶ correction: 0,000000 nanoseconds					
ClockIdentity: 0x70f8e7fffd008a0					
SourcePortID: 4					
sequenceId: 480					
control: Other Message (5)					
logMessagePeriod: 127					
requestreceiptTimestamp (seconds): 1620314566					
requestreceiptTimestamp (nanoseconds): 741007207					
requestingSourcePortIdentity: 0x6cb311fffe525e00					
requestingSourcePortId: 1					

Fig. 6.5 Wireshark capture of gPTP protocol in interface enp2s0.

The dissection of the PTP frame shows how the frame is being generated in the TSN switch. It computes its own correction and resends a new frame (see Section 1.1.2). We now have all the elements in the scenario synchronized, and the endpoints need to get their TSN configuration and start streaming or receiving the flows.

6.3 Details on OPC-UA Address Spaces

As stated during the course of this document, there is a need to design the Address Spaces, so the CUC can poll variables, update configuration and call the self-configuration methods. Thus, the CUC can provide all the data to configure the TSN domain.

After creating all the nodes and starting the servers, the Address Space structure can be obtained by using a free OPC-UA client named “freeOpcUaClient”. In order to get access to the variables, the CUC needs the *NodeId* values, shown in **Figure 6.6**.

▼	...	1:EndpointFeatures	ns= 1;i= 1000
>		1:earliestTransmitOffset	ns= 1;i= 1016
>		1:endpointType	ns= 1;i= 1002
>		1:identificationTypes	ns= 1;i= 1009
>		1:interfaceName	ns= 1;i= 1004
>		1:intervalDenominator	ns= 1;i= 1012
>		1:intervalNumerator	ns= 1;i= 1011
>		1:jitter	ns= 1;i= 1018
>		1:latesTransmitOffset	ns= 1;i= 1017
>		1:macAddress	ns= 1;i= 1003
>		1:maxDelay	ns= 1;i= 1006
>		1:maxFrameNumber	ns= 1;i= 1013
>		1:maxFrameSize	ns= 1;i= 1014
>		1:priority	ns= 1;i= 1010
>		1:redundancy	ns= 1;i= 1005
>		1:streamId	ns= 1;i= 1001
>		1:streamIdTypes	ns= 1;i= 1008
>		1:transmissionSelection	ns= 1;i= 1015
>		1:vlanCapable	ns= 1;i= 1007
▼	...	1:PublishObject	ns= 1;i= 1026
>		1:publishObjectData	ns= 1;i= 1027
▼	...	1:TSNInterfaceConfig	ns= 1;i= 1019
>	(x)	1:LaunchConfig	ns= 1;i= 1020
>		1:gclGates	ns= 1;i= 1023
>		1:gclGatesTimeDuration	ns= 1;i= 1024
>		1:interval	ns= 1;i= 1022
>		1:latency	ns= 1;i= 1021
>		1:vlanIdValue	ns= 1;i= 1025

▼	...	1:EndpointFeatures	ns= 1;i= 1000
>		1:endpointType	ns= 1;i= 1002
>		1:identificationTypes	ns= 1;i= 1009
>		1:interfaceName	ns= 1;i= 1004
>		1:macAddress	ns= 1;i= 1003
>		1:maxDelay	ns= 1;i= 1006
>		1:redundancy	ns= 1;i= 1005
>		1:streamId	ns= 1;i= 1001
>		1:streamIdTypes	ns= 1;i= 1008
>		1:vlanCapable	ns= 1;i= 1007
▼	...	1:SubscribeClientObject	ns= 1;i= 1010
>	(x)	1:InitSubscription	ns= 1;i= 1012
>		1:interval	ns= 1;i= 1011

Fig. 6.6 Address Space instances for Talker and Listener.

The figure includes an overview of all the variables that the OPC-UA servers manage. For more detail on each field, see Section 5.3.1. For all fields used in the sample demo, the namespace is the same, set as 1. The identifier value “i” points to each variable node.

6.4 Endpoint configuration

Setting up the endpoints to act as Talkers or Listeners in a TSN stream requires the configuration received from the OPC-UA client to be mapped to different configuration commands. In fact, as shown in **Figure 6.6**, variables *LaunchConfig* and *InitSubscription* are method types. This means that a call to these nodes will trigger a function. Therefore, we need to implement the configuration function to perform the following steps:

6.5 Prototype integration and set up

This section links all the previous steps in order to fully instantiate the prototype of the thesis. In order to do that, it is recommended to read Annex C, which specifies the developer's guide of the software implementation [55], giving details on how the flow is explicitly implemented and how it could be modified for different setups. Anyways, the current implementation is the one used for the model presented during this section.

6.5.1 System requirements

All endpoints that take part in the environment are required to have the following:

1. Ubuntu 20.04 [56] or any distribution that uses a Linux kernel 5.0 or newer.
 - a. Older distributions from Ubuntu can be used, but a manual kernel and *iptables* update is required.
2. Intel i210 Network Interface Card.
 - a. Time synchronization running in all endpoints and the SoC-e MTSN switch, as described in Section 6.2.

Apart, all systems that take part on the architecture need to:

1. Install *Node* [57], which includes the *npm* packet manager, in order to install all applications on the devices.
2. Be in the same LAN (see Section 6.1).

With all these conditions satisfied, the software implementation can be set up in the components. It is important to state that the scenario may work even though the time synchronization is not set in the network. However, results would be totally unpredictable, since the time offsets between elements is unknown and, therefore, deterministic performance cannot be guaranteed.

6.5.2 Set up of the components

Even though the prototype considers the CUC and the Talker in the same computer, the setting up of the components is treated separately, because both software implementations are independent and it can be easily separated by modifying the configuration files.

In order to install all the required external libraries of the Node projects for any of the components (Talker, Listener and CUC), call command "*npm i*" in the following directories:

1. /TSN-CNC-CUC-UPC/CUC
2. /TSN-CNC-CUC-UPC/ENDPOINTS/Talker
3. /TSN-CNC-CUC-UPC/CUC/Listener

Then, this prototype will install the Talker/CUC solutions to the same PC and the Listener software solution will be set up in the other machine.

6.5.2.1 Endpoints

The “*config.json*” file provides different parameters regarding the endpoint properties and stream identifier to publish or subscribe. The Listener configuration file is shown in **Figure 6.7**.

```
1  {  
2      "macAddress": "6c-b3-11-52-5e-00",  
3      "type": "LISTENER",  
4      "streamId": "6c-b3-11-52-50-dc-ff-ff",  
5      "interface": "enp3s0",  
6      "talkerEndpointUrl": "opc.tcp://192.168.4.11:4333/TSNInterface"  
7  }
```

Fig. 6.7 Listener’s config.json.

If we test TSN flow that is used is the OPC-UA Publish/Subscribe, the Listener needs to specify the URL of the Talker that will publish the TSN stream. For the Talker case, **Figure 6.8** shows the configuration properties.

```
1  {  
2      "macAddress": "6c-b3-11-52-50-dc",  
3      "type": "TALKER",  
4      "streamId": "6c-b3-11-52-50-dc-ff-ff",  
5      "interface": "enp2s0",  
6      "interval": 1,  
7      "dataLength": 1  
8  }
```

Fig. 6.8 Talker’s config.json.

In order to easily modify the traffic specifications (only for the OPC-UA Publish/Subscribe case), the property “*interval*” sets the amount of time in

seconds in which a variable is updated periodically, triggering the Publish of the “*publishObjectData (ns=1i=1027)*” node. The length of this Address Space node is set by the field “*dataLength*” in the configuration file, in Mbytes.

After setting the configuration files for both endpoints, they can be run by the command “*node ./index.js*” in their respective root folders. The Talker, after waking up, will set the values of its Address Space and start modifying the “*publishObjectData*”. The Listener, instead, sets its Address Space properties and waits for an action from the CUC.

If the action of the CUC is not triggered in the Listener, the OPC-UA Subscription does not occur, thus the TSN flow generated by the Node application is never sent through the network. This is a valid use case to test the TSN features with other tools, such as *iperf*.

Once the CUC sends the configuration back to the devices, the Talker configures the interface based from its own information and the one received by the CUC, confirming it with the logs, as shown in **Figure 6.9**.

```
Handle all process to configure i210 board
TAS Interval: Variant(Scalar<UInt32>, value: 1000000000)
GCLGates: Variant(Array<UInt32>, l= 2, value=[3,5])
GCLGatesDuration: Variant(Array<UInt32>, l= 2, value=[100000000,900000000])
Latency: 100
Vlan ID: Variant(Scalar<UInt32>, value: 1997)
Configuration applied to the interface successfully.
```

Fig. 6.9 Configuration values for the Talker’s interface configuration.

The Listener, when it receives the action from the CUC, will perform a Subscribe to the Talker’s variable in its Address Space, as follows in **Figure 6.10**.

```
subscription started for 2 seconds - subscriptionId= 254205
-----
Received
Received
Received
Received
Received
Received
Received
Received
Received
Received
```

Fig. 6.10 Subscription triggered and received in Listener.

Note that the implemented prototype can be tested with OPC-UA traffic, but also with any kind of traffic the Talker would want to give guarantees. We only need to reset the TSN requirements in the Talker to use a different traffic.

6.5.2.2 Centralized User Control

After installing all the npm packages, we set the OPC-UA Client URLs in the file “*config.json*”, shown in **Figure 6.11**.

```
1  {  
2    "endpointUrlTalker": "opc.tcp://localhost:4333/TSNInterface",  
3    "endpointUrlListener": "opc.tcp://192.168.4.10:4334/TSNInterface",  
4    "cncUrl": "RFU"  
5  }
```

Fig. 6.11 *config.json* for the CUC.

In this example the URL of the Talker is in the localhost, the same machine as the CUC. Both CUC and Talker can run separately if this variable points to the Talker OPC-UA Server in a different machine.

Apart from the OPC-UA Servers addresses, there is also the URL to the CNC, the RESTCONF server, mocked. However, for development purposes, it can be changed to use the RESTCONF Client module (see Annex C).

After setting correctly all the Endpoints' URLs, the project can be started by calling the command “*node ./index.js*” in the root directory of the CUC project. If the OPC-UA Servers are awake, we can see how the connection is established by checking the logs, as shown in **Figure 6.12**.

```
C:\Program Files\nodejs\node.exe .\CUC\index.js  
Polling Talker features and requirements...  
Talker information received, waiting for Listener...  
Listener information received. Parsing now both obtained datagroups.
```

Fig. 6.12 CUC waking up process logs.

Once the Servers' Address Systems has been accessed, the CUC generates a YANG instance group and sends it to the CNC, as shown in **Figure 6.13**.

```
UNI Talker and Listener groups have been instantiated. Sending request to the CNC.
***MOCKED*** Response from the CNC received. Parsing the configuration...
```

Fig. 6.13 CUC logs regarding the instantiation of the UNI groups.

After receiving the network configuration from the CNC, the CUC computes the Talker's GCL and sends it to the Talker, including the *timeOffset* and variables to help the Endpoint support several flows and interfaces, such as *interface*, *vlanId*, *streamId* and *macAddress*. It also has the *latency* to the listener (mocked property, which is supposed to be obtained from the CNC response), as shown in **Figure 6.13**.

```

v talkerConfig: {gcl: {...}, vlanId: 1997, streamId: '6c-b3-11-52-50-dc-ff-ff', i
> gcl: {interval: 1000000000, states: Array(2), duration: Array(2)}
  interface: 'enp2s0'
  latency: 100
  macAddress: '6c-b3-11-52-50-dc'
  streamId: '6c-b3-11-52-50-dc-ff-ff'
  timeOffset: 20
  vlanId: 1997

```

Fig. 6.14 CUC generated GCL for the Talker.

After the configuration is successfully sent by the CUC, it triggers the methods to the endpoints, starting the TSN flow.

```

Sending configuration to Talker...
Sending subscription details on Listener...
Configuration received and applied by the Talker.
Subscribe established on Listener.

```

Fig. 6.15 CUC sending TSN and Subscription config.

Figure 6.16 shows the testbed scenario running at EETAC's laboratory C4-325.



Fig. 6.16 Equipment used in the testbed. From left to right: Linux PC running the Talker, TSN switch, Linux PC running the Listener.

CHAPTER 7. TESTS AND RESULTS

This chapter describes the tests performed on the prototype and the results obtained. All test reports are provided with the following information:

1. Test objective
2. Set up and configuration properties
3. Results
4. Test conclusions

The two main tests reported here are:

1. Basic scheduling test and evaluation with OPC-UA Pub/Sub implementation
2. Scheduling performance test with TSN traffic generated by iperf

All tests performed and its results can be found in the project repository [58]. This resource contains more tests than the ones reported. This is because this document aims to give the most remarkable results.

In the following figures and charts, the TSN flow is always colored green, the best-effort traffic is orange and gPTP traffic (which is very light and it cannot be seen in most of the figures) is dark.

7.1 Basic scheduling test and evaluation with OPC-UA Pub/Sub implementation

The goal of this test is to evaluate the scheduling that is being performed by the Talker. The test is performed using the OPC-UA traffic generated in the Javascript program. To achieve this, different traffics and different congestion status are used. An analysis of the transmission and system delays is provided to bound the performance of the implementation. In addition, the use of a Node application to generate high-sensitive traffic is tested.

7.1.1 Set up and configuration properties

The prototype is tested with several traffic scenarios:

1. First, only TSN traffics are sent by the device. This helps to determine the behavior of the scheduling mechanism and the traffic generation

implementation (handled by a Node application). In order to put the system under stress, transmission times are reduced and the traffic is analyzed. **Figure 7.1** provides more details of the scenario:

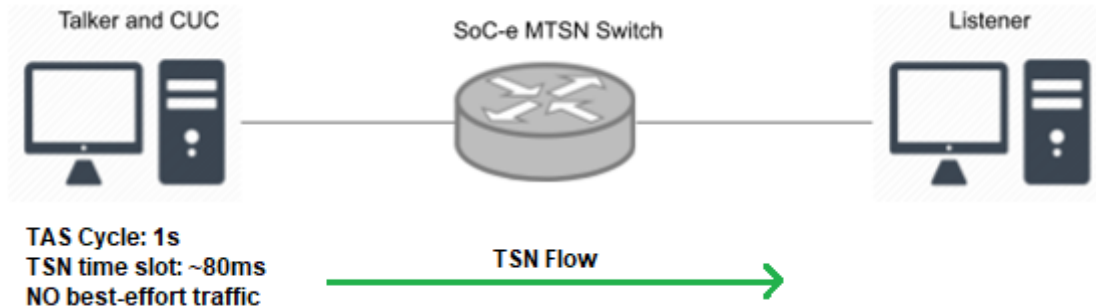


Fig. 7.1 First scenario, with the TSN flow.

2. In a second step, similar TSN traffics are used combined with a best-effort flooding. This is to determine how the time slots of the scheduler are used by the different priorities, proving the correct performance of the configuration received by the CUC. In addition, the behavior of the generated TSN traffic is analyzed. **Figure 7.2** illustrates the scenario:

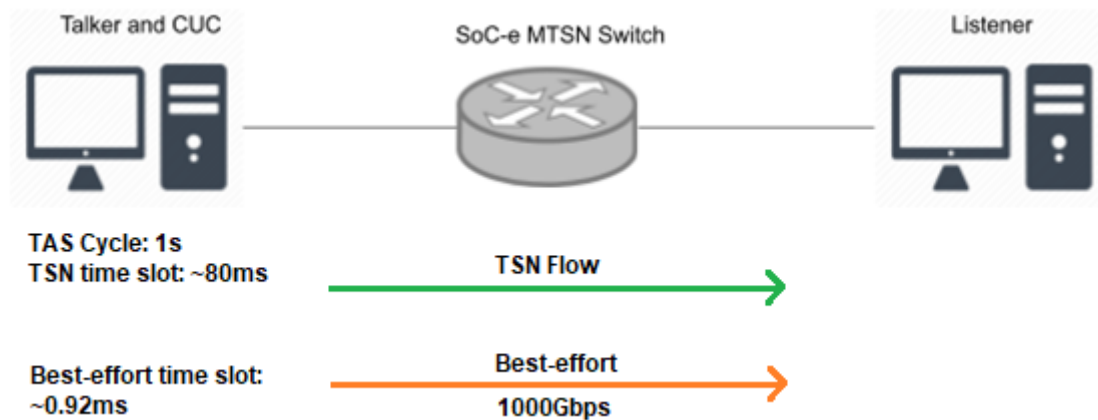


Fig. 7.2 Second scenario: similar to the first one, with an additional best-effort traffic.

3. Finally, the second case is repeated without any TSN configuration on the Talker. Because of this, both performances can be compared and extract conclusions from them. **Figure 7.3** presents this scenario.

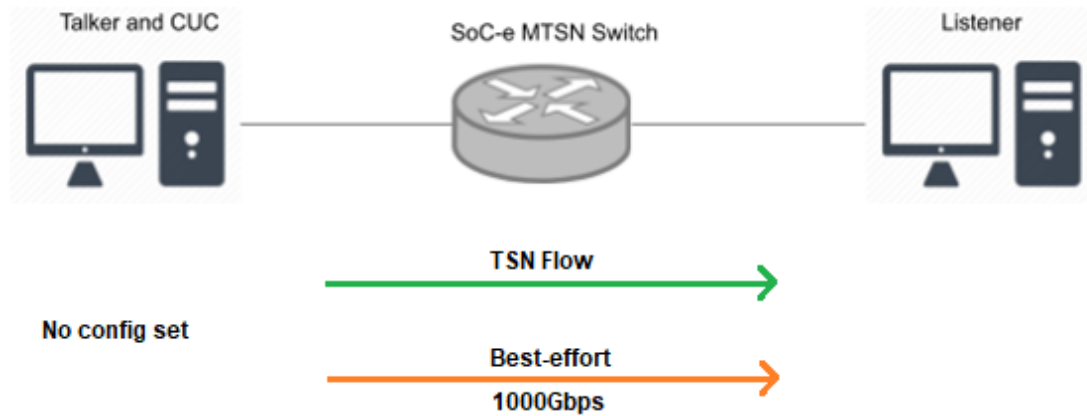


Fig 7.3 TSN configuration is disabled for the third test case.

7.1.2 Results

When a traffic composed of a 1 MByte file sent every one second cycle, we obtain **Figure 7.4**, where we see the received packets at the Listener in a time graph.

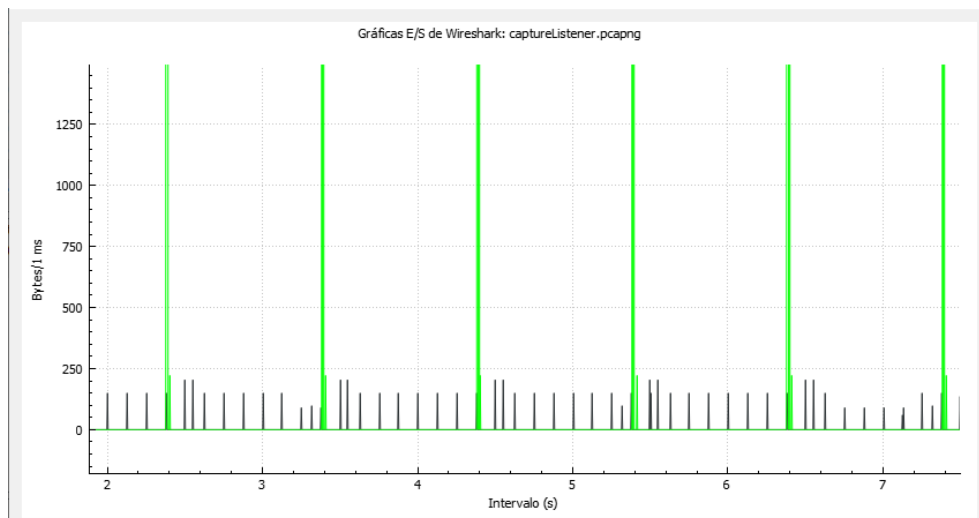


Fig. 7.4 Wireshark capture showing TSN flow and gPTP protocol.

We can see two different traffics in the capture. The green one is the OPC-UA TSN flow, where the Listener receives 1 MByte in every cycle (1 second). The dark traffic is gPTP and, since it is configured as the highest priority, it can be transmitted at any time, during any TAS defined interval. Thanks to this, the clock synchronization in the network is maintained. If the gPTP traffic was considered only in a time window, there would be a time offset between the sending from the application level and the forwarding from the NIC, causing failures when trying to maintain synchronization.

In addition, this traffic can also provide details on the delay of the Node application traffic. It is based on the logs that both Talker and Listener report for every TSN packet sent or received in the OPC-UA server and client, respectively. By considering the sending/receiving timestamps, delay and jitter figures can be built. The delay and jitter of every received Subscribe message is plotted in **Figure 7.5**. This traffic is transmitted from Talker to Listener every one second after being configured by the CUC. The time slot for the TSN traffic is set to 80.028 ms.

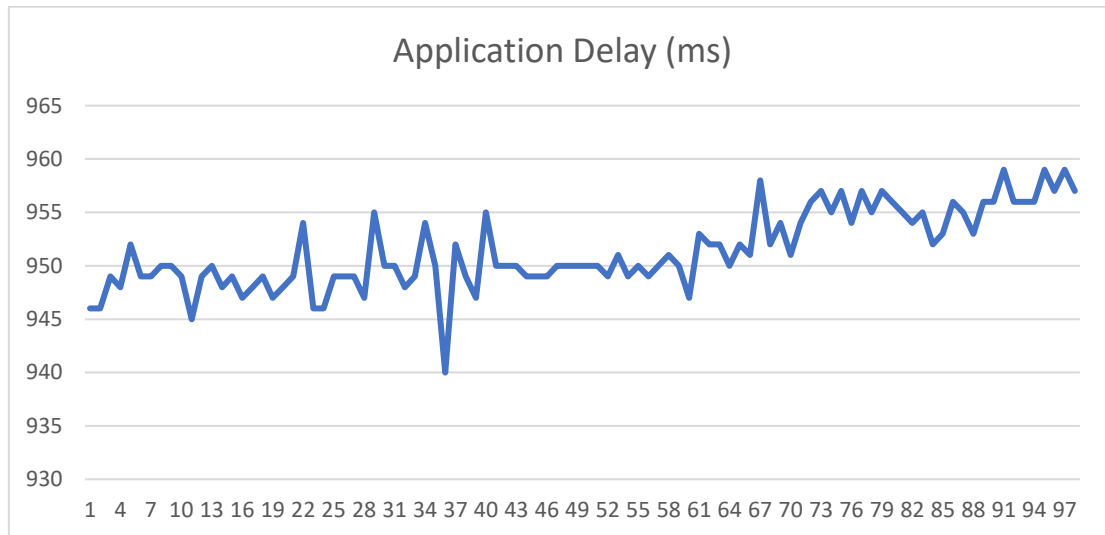


Fig. 7.5 OPC-UA Pub/Sub traffic delay in milliseconds (Y axis), versus Subscribe message sequence number (X axis).

At first, the most remarkable result is that the delay reaches very high values, just below a second. The reason may be the traffic scheduling and its synchronization with the Node application. When the transmitted packet is generated in the application level, it is transmitted to the Linux kernel and stays in the *queuing disciplines* until the scheduler chooses it. Then, since our TAS has chosen a time interval of one second, it may be possible that the packet arrives to the queue right after it has been its time to transmit. In addition to this, the time slot set for the TSN flow is adjusted to the “*maxFrameSize*” and “*maxFrameNumber*” set in the Talker. This means that the slot set is not big enough to dequeue more than one Subscribe frame. Consequently, the following Subscribe messages will be shifted one entire cycle of the scheduler, providing delays close to the cycle duration, one second.

The Application jitter (the variation of the delay for every packet compared to the average delay) is shown in **Figure 7.6** and most of the results are between ± 6 ms. Since we are checking from server application to client application, this variance may come from the Talker’s application or from the network.

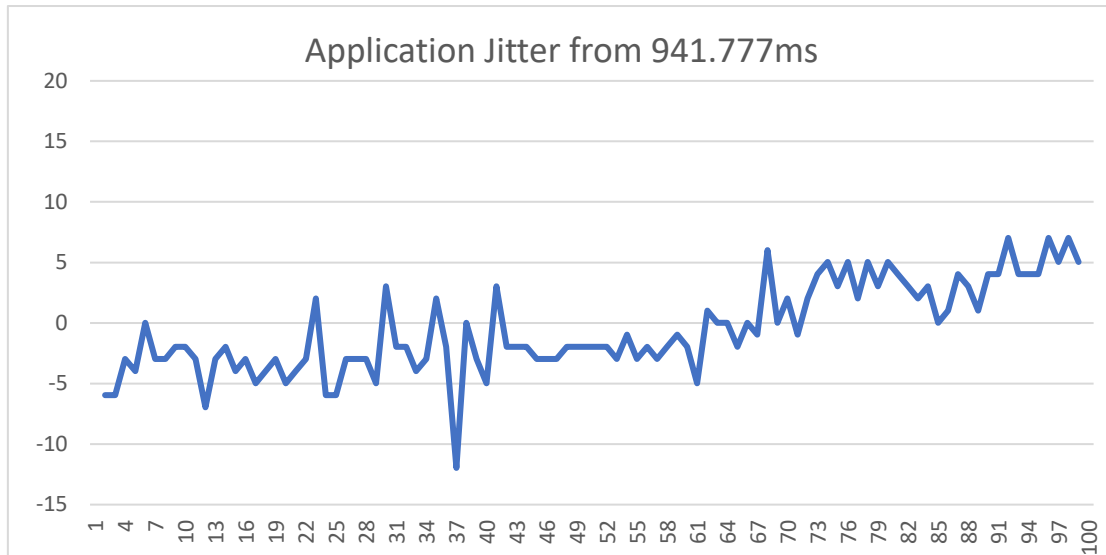


Fig. 7.6 OPC-UA Pub/Sub traffic jitter in milliseconds (Y axis), versus Subscribe message sequence number (X axis).

From the same Wireshark capture used in **Figure 7.4**, the timestamps performed by the Intel i210 cards at the sending and receiving instants can be checked and compared. By checking every first packet timestamp of every interval, the end-to-end delay and jitter of the network can also be determined, as shown in **Figure 7.7**.

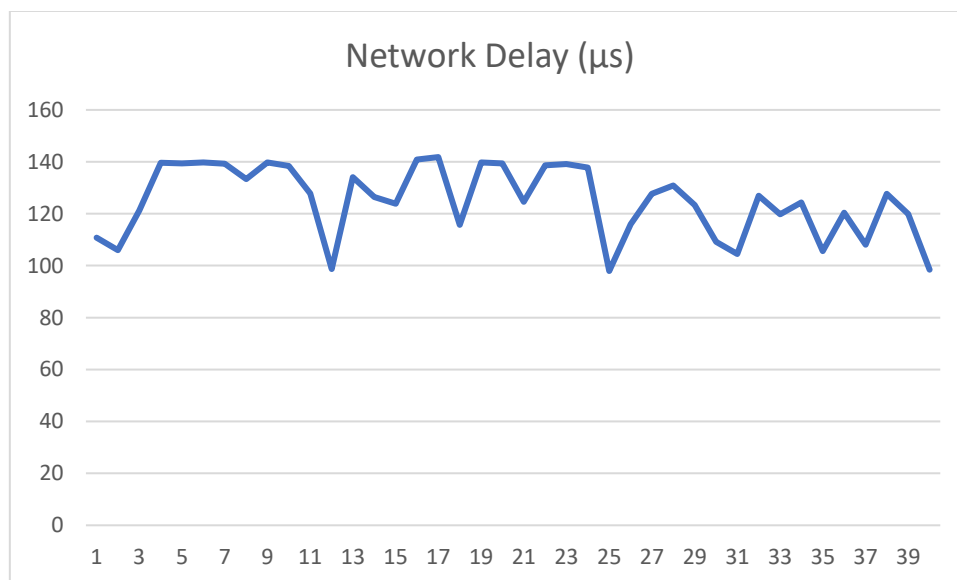


Fig. 7.7 Network delay for the case of 1MByte every 1s TSN traffic, in microseconds (Y axis), versus Subscribe message sequence number (X axis).

At first look, the network delay is much smaller than the Application delay and it is very reasonable, since the network is not congested. Once a packet is

transmitted, it will not find any full buffers during the path, so its delay will be the minimum. Regarding the jitter, it is expected that the delay of different frames does not differ too much, as we can see in **Figure 7.8**.

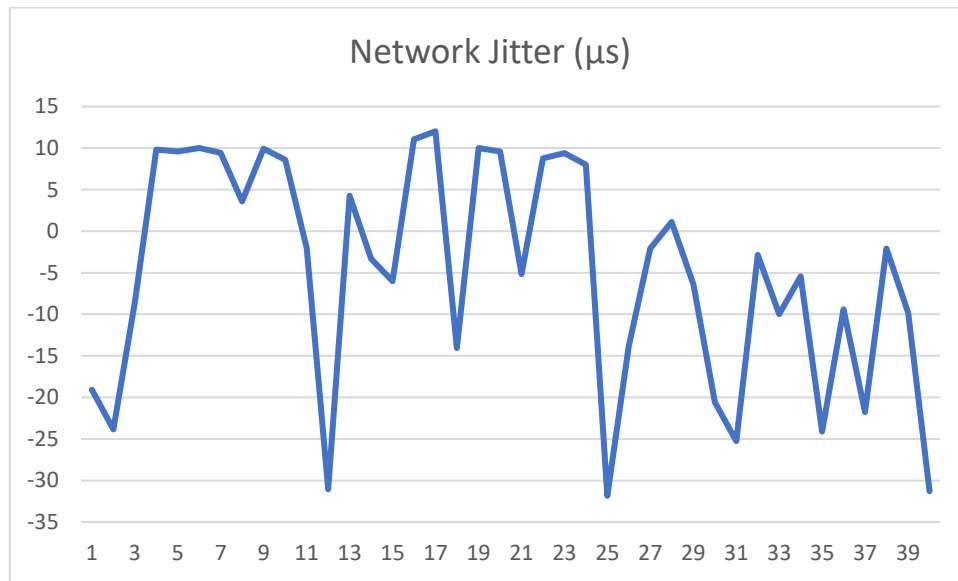


Fig. 7.8 Network jitter for the case of 1MB every 1s TSN traffic in microseconds (Y axis), versus Subscribe message sequence number (X axis).

Since the transmission delay and jitter are much lower than the Node App latency, it seems that the traffic generation is not performed optimally. To optimize it, the Talker prototype has been modified. It determines the moment in which the Publish will happen, and it places the frame generation before the start of the appropriate TAS slot. The result of this change is shown in **Figure 7.9**.

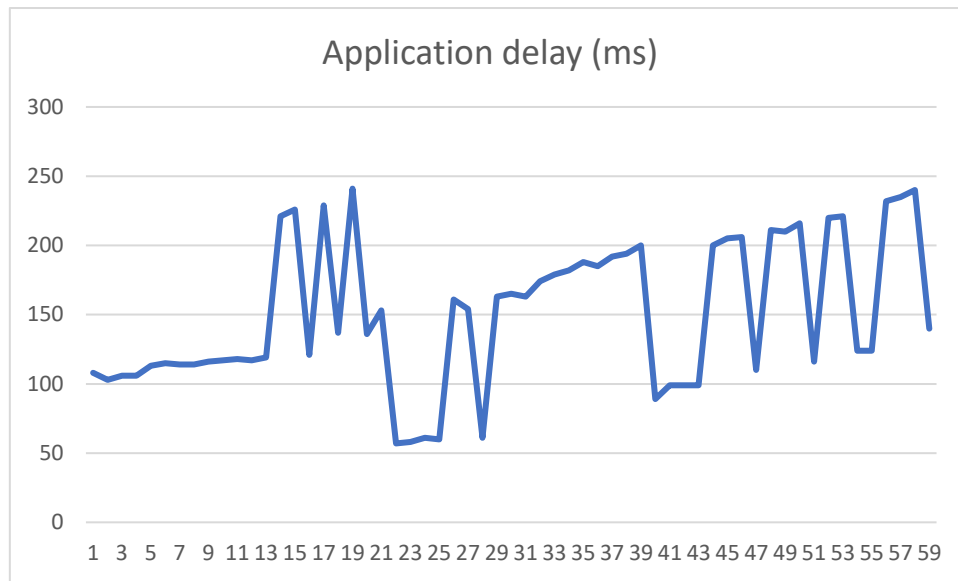


Fig. 7.9 OPC-UA Pub/Sub traffic delay (after the modification) in milliseconds (Y axis), versus Subscribe message sequence number (X axis).

The delay has decreased importantly. Nevertheless, there is still a lack of strict synchronization in the implementation of the OPC-UA Publish traffic. It always leads to high values of latency. The jitter that can be observed in **Figure 7.10** is also higher than before. Since the TAS is the same, this behavior must come from a lack of synchronization in the Node App.

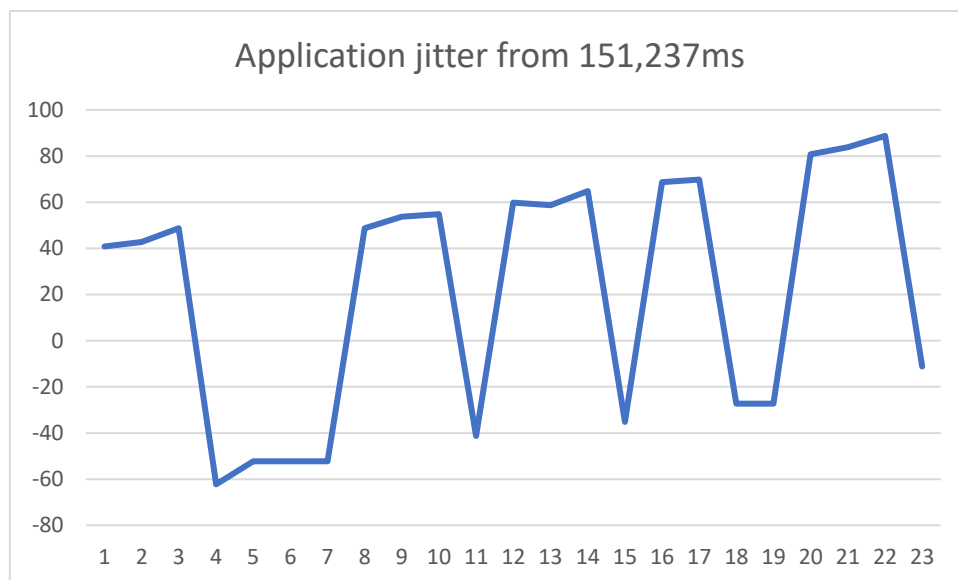


Fig. 7.10 OPC-UA Pub/Sub traffic jitter (after modification) in milliseconds (Y axis), versus Subscribe message sequence number (X axis).

From this traffic, we can conclude that our prototype cannot provide with time synchronization to services that are not attached to the system clock. If the Node

application was fully synchronized and transmitted at the same time the TAS slot began, the delays would look more similar to the network ones. Thus, the TSN behavior would be real.

The latency analysis on the first traffic has determined that the temporal requirements cannot be satisfied with the Javascript generated traffic, and therefore a different application has to be used. However, there is still a need to check the behavior of the scheduler when best-effort traffic is generated. **Figure 7.11** shows the results of adding a best-effort traffic. We can see the 1s interval set up in the initial configuration and it respects the slots set for every priority, whether TSN flow or best-effort traffic.

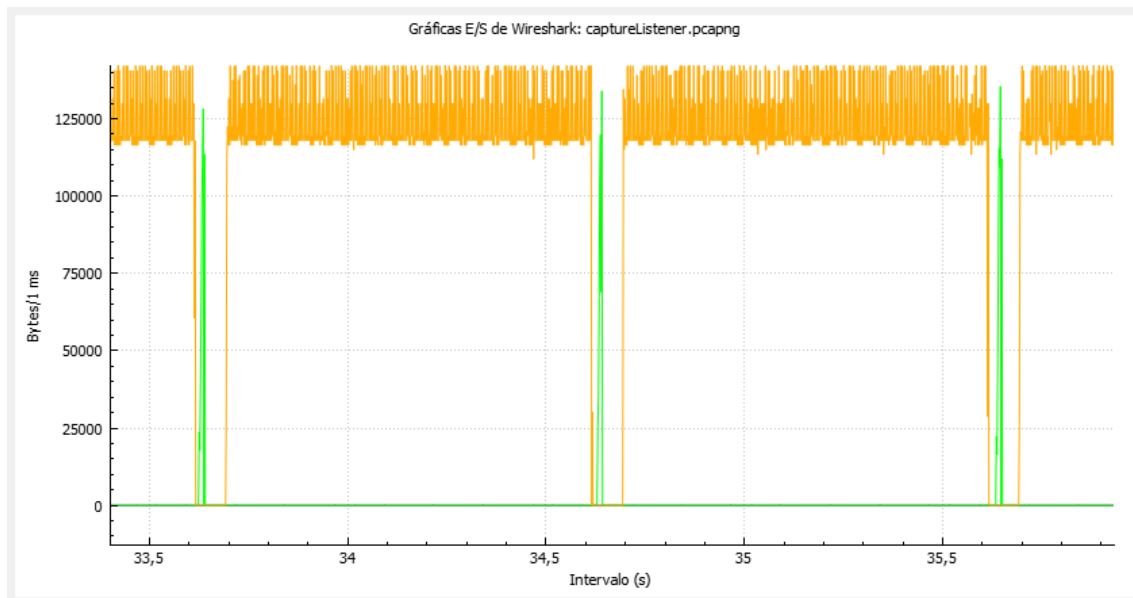


Fig. 7.11 Scheduling performed with simultaneous OPC-UA Pub/Sub and best-effort traffics.

From the figure, we can conclude that the time-aware scheduler is definitely working in our Talker endpoint. The performance of the system seems correct, although the performance of the traffic generator is not good.

If we analyze again the delay and jitter for the OPC-UA Pub/Sub traffic, the results are similar to the first round of tests (these results are not shown). Also, lowering the scheduler cycle used to 100ms (instead of one second), the delay and jitter results are still bad. And we can see that delays behave erratically, with somewhat stable regions followed by sudden decreases or increases, as shown in **Figure 7.12**.

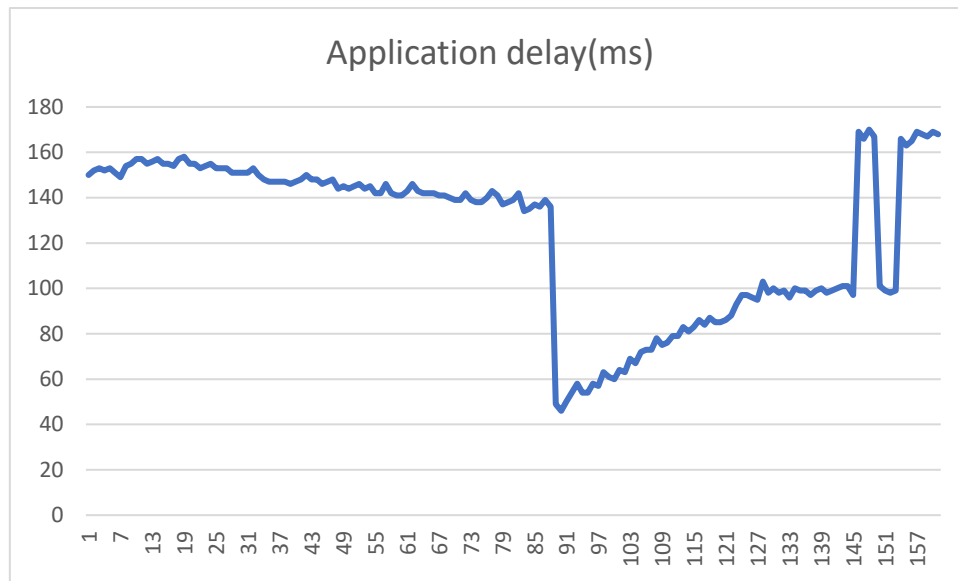


Fig. 7.12 Delay for the TSN flow (100ms interval) when there is best-effort traffic, in milliseconds (Y axis), versus Subscribe message sequence number (X axis).

These gaps, such as the one that happens around packet 86, could be caused by the fact that the Node application cannot synchronize, and therefore one of the Publish packets has been created in a moment in which the TAS slot was about to trigger, minimizing the time this message waits in buffer. After it, the subsequent packets keep drifting in time, increasing again the delay. This event can be seen in **Figure 7.13**.

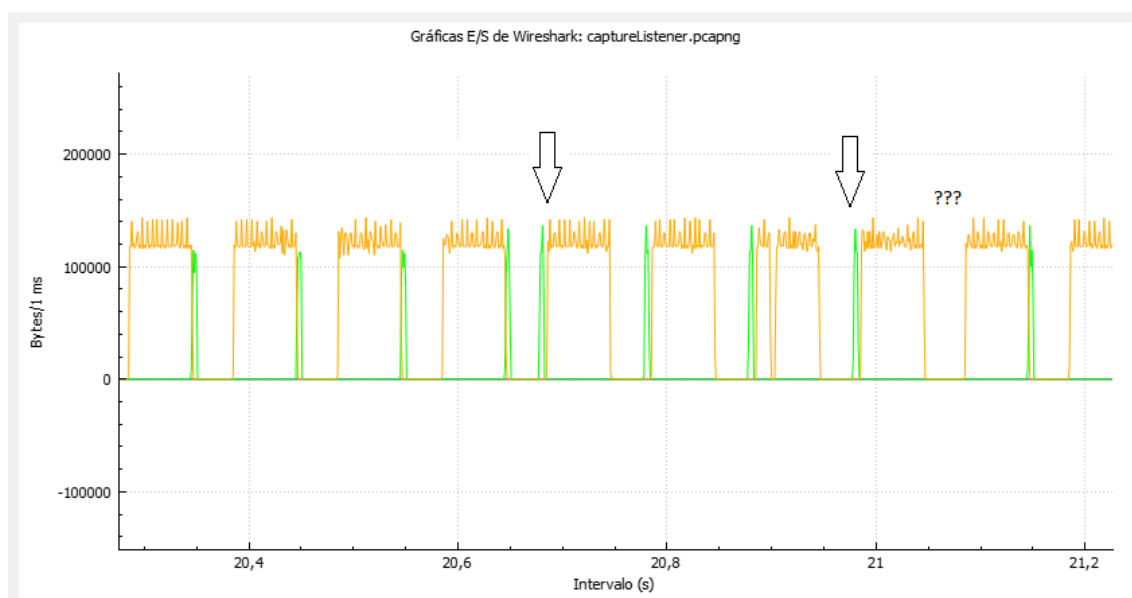


Fig. 7.13 Desynchronization coming from the Node application.

The arrows in **Figure 7.13** point to undefined behavior of the traffic generation. Note that one of these frames (the one pointed by the first arrow) is sent in the same interval than the previous one. This leads to a delay gap between the consecutive frames. In addition, the second misplaced frame leaves a slot without any transmission. If the Node App generated synchronously the frames, this would not happen. The scheduling is working properly, but TSN traffic is not being correctly generated.

Now that we seem to have isolated the problems to the traffic generator, it may be possible that the lack of synchronization in the Talker endpoint Node application generates a worse behavior with the TSN configuration applied. Because of that, we will compare now delay and jitter with some of the previous used traffics without any TSN configuration.

Going back to the traffic used in the first test (1 s interval), **Figure 7.14** shows that the scheduling is not being performed.

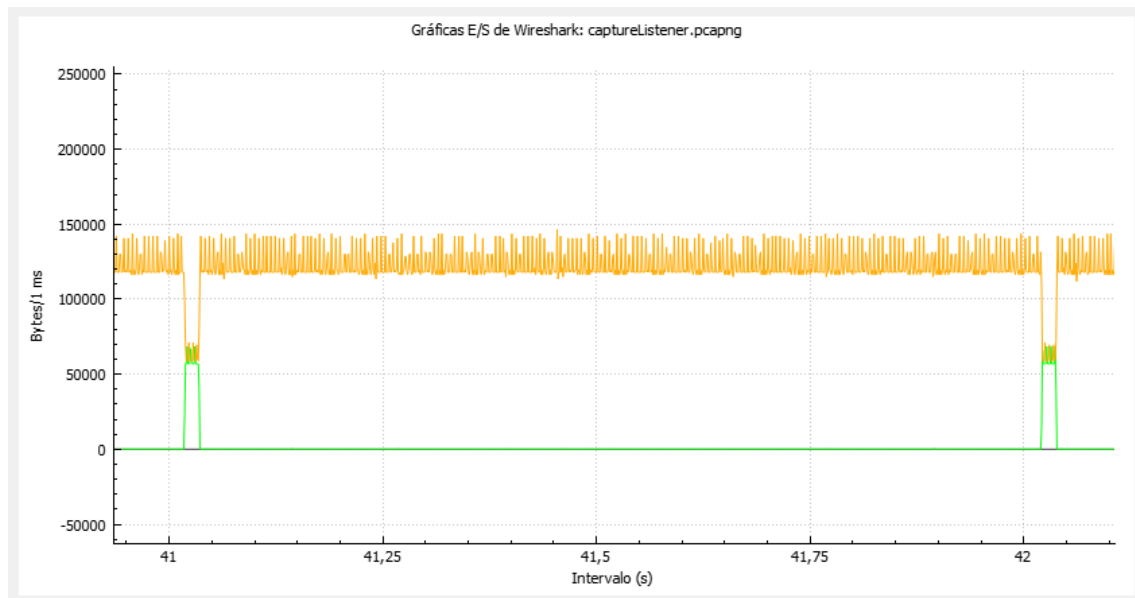


Fig. 7.14 Listener receiving nonscheduled traffic.

A best-effort traffic is being constantly transmitted with a bitrate of 1 Gbps. Iperf is the tool used to generate this traffic. When the OPC-UA traffic appears, just after $t = 41$ s and $t = 42$ s, the bandwidth is shared – we can see how the increase in the green traffic (OPC-UA) is compensated by the decrease in yellow traffic (best-effort).

Regarding delay and jitter, even though the interface is full of best-effort traffic, it may still have some degree of determinism, since no other elements are flooding the network. Because of that (and the fact that the Talker is not able to fully synchronize with TSN configuration), the delay is similar or even lower, as seen in **Figure 7.15** and **Figure 7.16**.

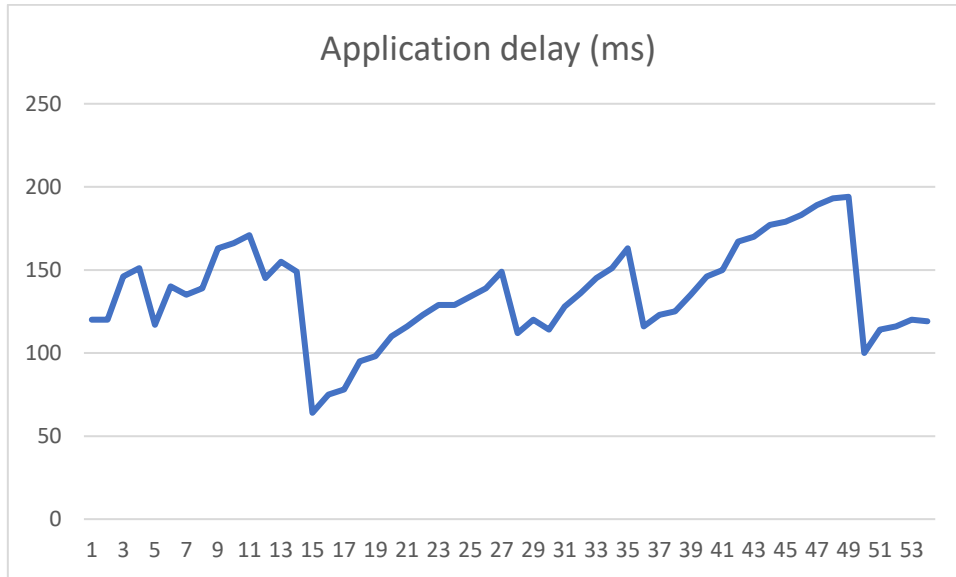


Fig. 7.15 Delay for the non-TSN configuration case and 1MB every 1s, in milliseconds (Y axis), versus Subscribe message sequence number (X axis).

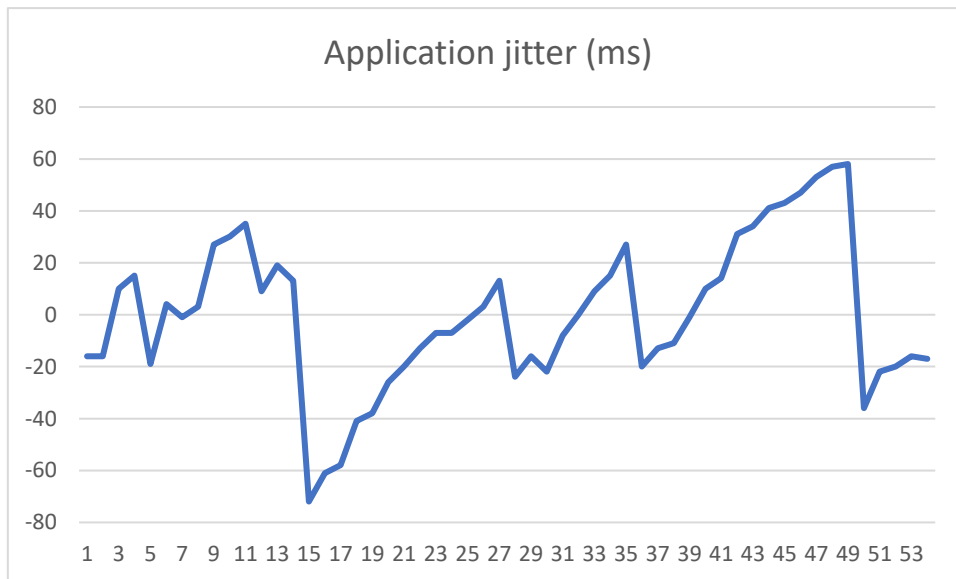


Fig. 7.16 Jitter for the non-TSN configuration case and 1MB every 1s, in milliseconds (Y axis), versus Subscribe message sequence number (X axis).

The shape of the delay and jitter graphs cannot be described, because there is not any known configuration in the NIC and all traffics are treated as best-effort. Because of that, the scheduling performed is FIFO, meaning that the frames are sent as they arrive to the buffers. This is not a deterministic behavior that can be described. As mentioned, the average value of the delay is similar to the one tested with time-scheduling. This confirms that the Node application generated traffic is not deterministic at all.

7.1.3 Test conclusions

After testing the system limits with this set up, the main conclusions are:

1. The traffic scheduling is performed successfully by the Talker, allowing gPTP to transmit at any time slot of the TAS and slotting the different traffic priorities, as shown in **Figure 7.4** and **Figure 7.11**.
2. The OPC-UA Publish/Subscribe traffic generated by the Node application Publish does not generate traffic synchronously. Strict TSN requirements will never be achieved with it.
3. This scenario does not permit the testing of shorter time intervals, since the traffic generation already brings dozens of milliseconds of jitter. With this condition, it is not possible to test schedulers in the order of the microseconds.
4. If the generated traffic is not scheduled correctly, the applied TSN configuration does not guarantee determinism to the system. In fact, the system gets worse in terms of latency and jitter.
5. The computation of the TSN configuration for the endpoints (as described in Section 6.5) works properly. This means that the architecture designed achieves to establish a TSN domain. Even though, it still needs to be tested at smaller time scales.

Therefore, we must change the data source for the TSN flow and evaluate the TSN performance of the Talker endpoint. In the following test, the traffic generator is switched from a Node application to *iperf*.

7.2 Scheduling performance test with TSN traffic generated by iperf

The goal of this test is to bring the system to its limits regarding time scale. To achieve this, two TCP/UDP traffics (with constant bitrate) are generated with iperf. One is considered the best-effort and the other the TSN flow. By lowering the time scale, the minimum supported TAS interval and priority slot can be identified for the current system.

7.2.1 Set up and configuration properties

Similarly to the previous test, different traffics and congestion conditions are applied. The scenarios include:

1. A basic scheduling with an interval of one second, to check that the incoming TSN configuration is still valid when the generator is changed to iperf. **Figure 7.17** shows the scenario of the test:

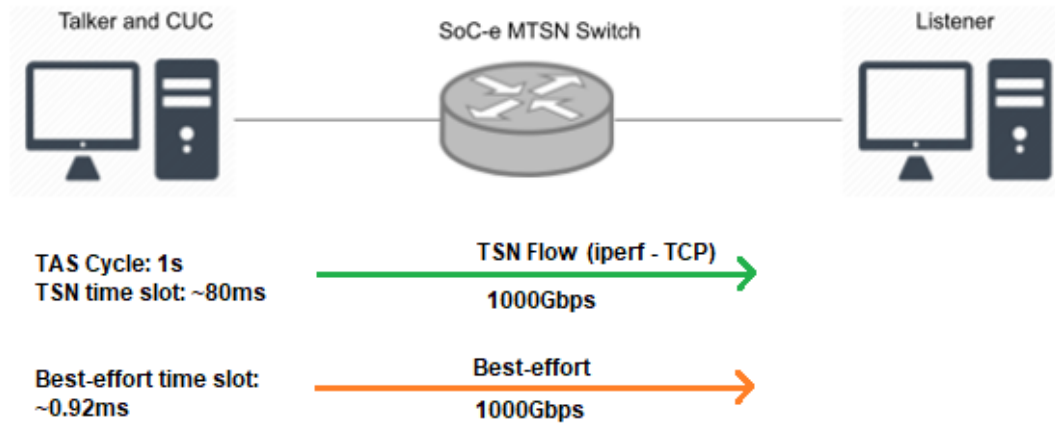


Fig 7.17 First scenario with iperf traffic. Basic scheduling in a large TAS cycle.

2. In the same time interval, the time slot is reduced until a reasonable minimum is found. **Figure 7.18** provides more details of the traffics used:

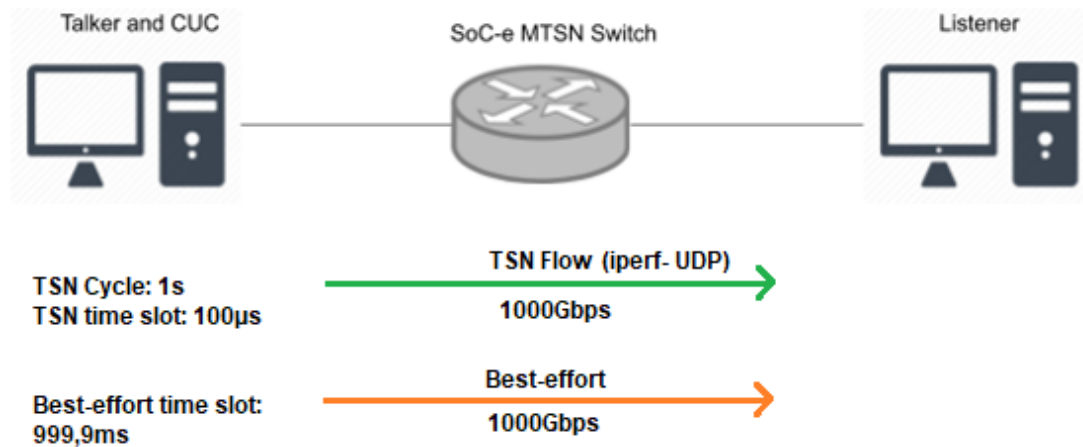


Figure 7.18. Second scenario, with the shortest time slot for a given gate state.

3. The TAS interval is reduced to 200μs (two slots of 100μs), the hypothetical shortest period obtained in the previous case. **Figure 7.19** shows the traffics used and how the scheduler's cycle is reduced.

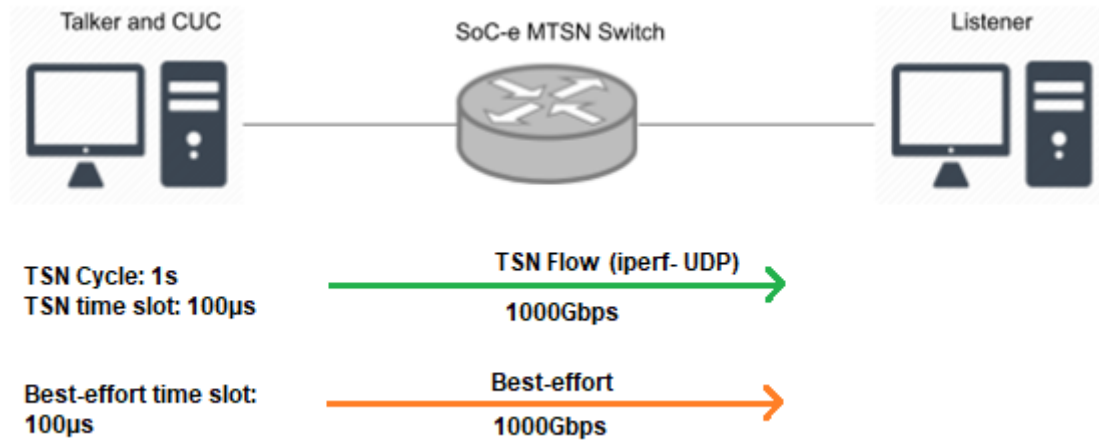


Fig. 7.19 Third scenario, for testing the shortest possible scheduler cycle in the prototype endpoints.

The TCP/UDP traffics used are both generated with the iperf tool between Talker and Listener. Due to the constant bitrate generation, set to 1 Gbps in both cases, the evaluation of the scheduling performed by the endpoint is optimal. This means that the TSN traffic does not depend on a synchronized frame generation, since frames are constantly generated in the maximum network bandwidth and frames will be available from the beginning until the end of the time slot. By this, it is possible to keep reducing the time slots and intervals of the scheduler and evaluate the use of these slots by the talker endpoint.

7.2.2 Results

The first thing to check is if the incoming configuration from the CUC is still correctly set for the iperf traffics. **Figure 7.20** shows how the interface starts transmitting both traffics without any kind of scheduling. Between time instants 12:35:41 and 12:35:42, the TAS is set and starts performing the time slots.

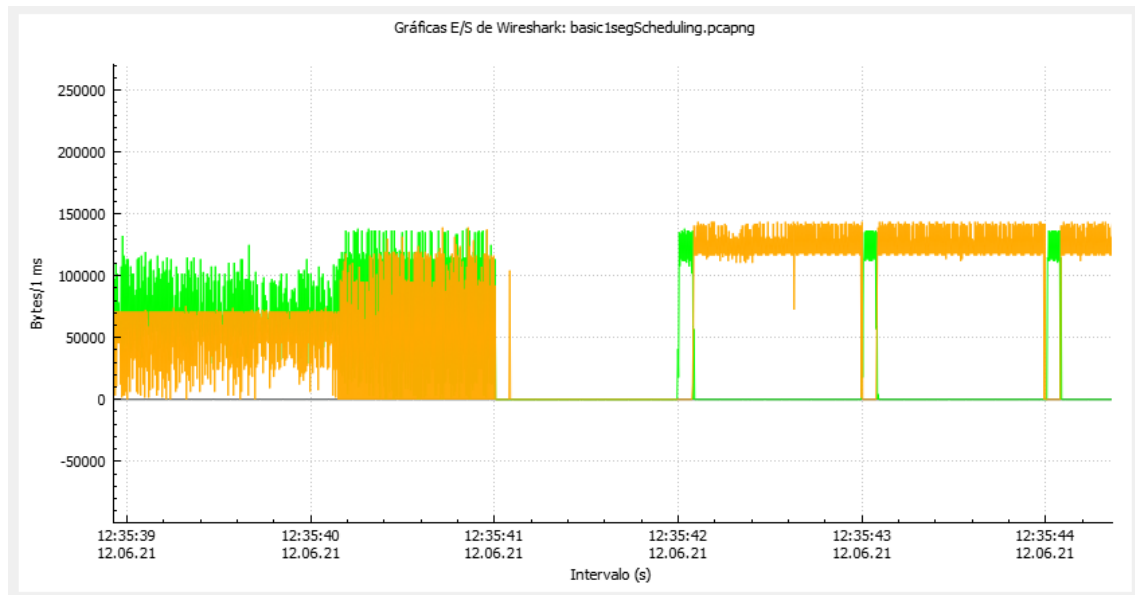


Fig. 7.20 Basic scheduling performance with iperf traffic.

During the scheduling, the TSN configuration defines that the scheduling starts exactly when the system clock ticks a new second. **Figure 7.20** does not provide enough detail of the scheduling starting time. Since the Wireshark plotting system time sensitivity is in the order of milliseconds, it is better to check directly the reception timestamps from **Figure 7.21**.

4996...	12:35:42,000113044	192.168.4.10	192.168.4.11	4333	2962	TCP	
TSN transmission							
5007...	12:35:42,080349037	192.168.4.10	192.168.4.11	5001	1512	UDP	Transition to best-effort
5008...	12:35:42,083012244	192.168.4.10	192.168.4.11	4333	11650	TCP	
Best-effort transmission							
5754...	12:35:43,000332228	192.168.4.10	192.168.4.11	4333	1514	TCP	Transition to TSN flow
5755...	12:35:43,001659381	192.168.4.10	192.168.4.11	5001	1512	UDP	

Fig. 7.21 Some Ethernet frames captured with Wireshark.

The theoretical time when the scheduler starts is $12:35:42,000000000$. However, the first frame of the TSN flow is received by the Listener $113.044 \mu\text{s}$ later. By checking the network delay calculated in **Figure 7.7**, its average value is $110.728 \mu\text{s}$. If both values are subtracted, $2.316 \mu\text{s}$ are obtained. This value is not exact, because the taken delay in the network is the average value. The $2.316 \mu\text{s}$ that the Talker has delayed its ideal transmission can perfectly come from the *phc2sys* synchronization offset, observable in **Figure 6.3**. This synchronization may be higher and worse results may happen in other tests, because *phc2sys* suffers offset peaks in the order of $100 \mu\text{s}$.

The TSN flow slot duration given by the CUC is 80,028 ms. Between instants 12:35:42,080349037 and 12:35:42,083012244 there is a transitory state during 2,663 ms. Here, the TAS is switching from one time slot to the next one, but it transmits both traffic classes during this small period of time. In addition, in the following transition (back to the TSN time slot) there is a transitory state during 1,327 ms. Considering the network delay, the first TSN frame has been transmitted 221,500 μ s late. This high value may come from the overall desynchronization between clocks, that includes:

1. Desynchronization between Talker's system clock and Talker's NIC clock (microseconds).
2. Offset between Talker's NIC and Listener's NIC clocks (nanoseconds).
3. Desynchronization between Listener's NIC clock and Listener's system clock (microseconds).

It is possible to test that the current performance fulfills the TSN requirements in terms of scheduling and time precision. By shortening the used time slots and intervals, a good analysis can now be performed (as opposed to the NodeJS App generated traffic case). Note that for shorter intervals, the TSN flow is switched to use UDP, with the same port 4333. TCP is only used with the longer time slots. This is because the acknowledge timers and sliding windows of the protocol may the sending of the data during short intervals.

First, to find the minimum usable time slot inside any TAS cycle, the transmission time slot for the TSN flow has been shortened until it is 100 μ s long, as shown in **Figure 7.22**. Again, the precision of Wireshark plotting is not enough to manage the microsecond scale, forcing to check the packets with Wireshark.

4677...	13:50:15,736711153	System-o_08:a0	LLDP_Multicast	60	PTPv2
4677...	13:50:15,736796443	System-o_08:a0	LLDP_Multicast	90	PTPv2
4677...	13:50:15,861725744	System-o_08:a0	LLDP_Multicast	60	PTPv2
4677...	13:50:15,861796060	System-o_08:a0	LLDP_Multicast	90	PTPv2
4677...	13:50:15,935248787	MS-NLB-PhysServer-0...	Broadcast	66	0xf1c1
4677...	13:50:15,986745358	System-o_08:a0	LLDP_Multicast	60	PTPv2
4677...	13:50:15,986815634	System-o_08:a0	LLDP_Multicast	90	PTPv2
4677...	13:50:16,000231086	192.168.4.10	192.168.4.11	4333	1512 UDP
4677...	13:50:16,000231271	192.168.4.10	192.168.4.11	4333	1512 UDP
4677...	13:50:16,000231331	192.168.4.10	192.168.4.11	4333	1512 UDP
4677...	13:50:16,000231389	192.168.4.10	192.168.4.11	4333	1512 UDP
4677...	13:50:16,000231447	192.168.4.10	192.168.4.11	4333	1512 UDP
4677...	13:50:16,000231504	192.168.4.10	192.168.4.11	4333	1512 UDP
4677...	13:50:16,000366407	192.168.4.10	192.168.4.11	4333	1512 UDP
4677...	13:50:16,000366495	192.168.4.10	192.168.4.11	4333	1512 UDP
4677...	13:50:16,108457004	Shenzhen_52:5e:00	LLDP_Multicast	68	PTPv2
4677...	13:50:16,108579418	System-o_08:a0	LLDP_Multicast	68	PTPv2
4677...	13:50:16,108632132	System-o_08:a0	LLDP_Multicast	68	PTPv2
4677...	13:50:16,111800719	System-o_08:a0	LLDP_Multicast	60	PTPv2
4677...	13:50:16,111820103	System-o_08:a0	LLDP_Multicast	90	PTPv2
4677...	13:50:16,236828997	System-o_08:a0	LLDP_Multicast	60	PTPv2
4677...	13:50:16,236852763	System-o_08:a0	LLDP_Multicast	90	PTPv2

Fig. 7.22 Wireshark capture with a transmission time slot of 100 μ s for the TSN flow.

The transmission time window is set during $135.409\text{ }\mu\text{s}$, a reasonable value given the synchronization precision of the network and system clocks. Shorter time slot durations are not transmitting any frame, $100\text{ }\mu\text{s}$ is the minimum configurable time slot duration. We can conclude that the TSN prototype can work properly with $100\text{ }\mu\text{s}$ time slots at the jitter level measured from the gPTP and the *phc2sys* processes.

To finish with this testbench, the TSN flow and the best-effort traffic will be set both in $100\text{ }\mu\text{s}$ time slots inside a $200\text{ }\mu\text{s}$ cycle of the scheduler. By this, the performance of the endpoint is tested using its smallest time slots in a TAS cycle. **Figure 7.23** shows the output of the Wireshark graph. Due to the fact that the slots are $100\text{ }\mu\text{s}$ long, the sensitivity of the Wireshark's plotting system cannot distinguish any scheduling.

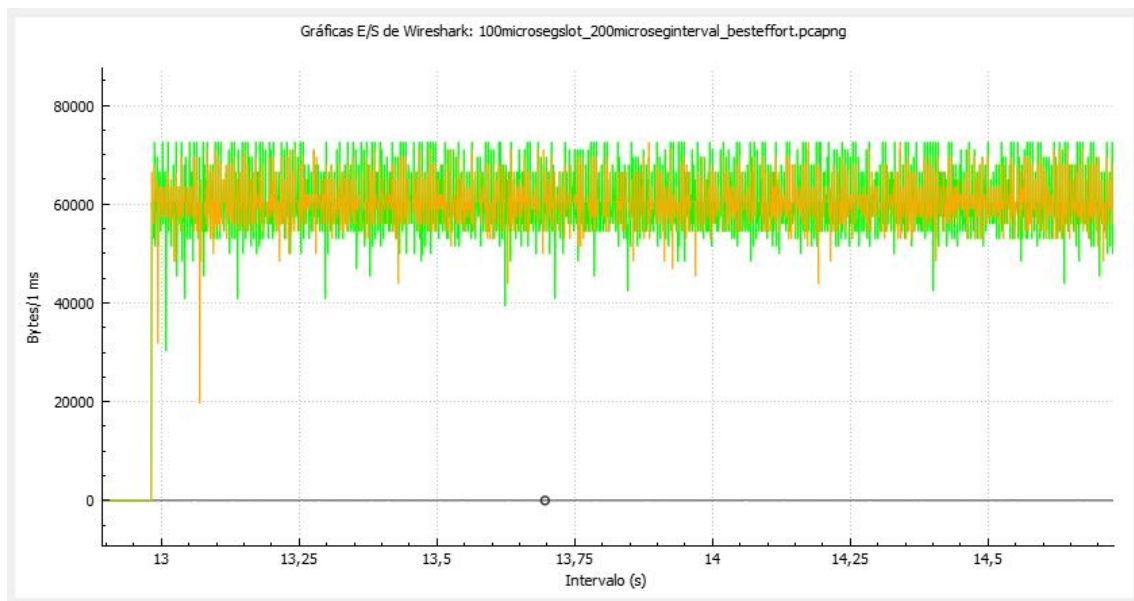


Fig. 7.23 Incorrect plotting of the $100\mu\text{s}$ time slot scheduling.

Again, it is better to check the capture directly, as shown in **Figure 7.24**:

4457...	14:22:40,000074332	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000074360	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000074387	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000074407	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000074435	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000074462	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000074483	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000086436	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000265377	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000265397	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000265424	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000265444	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000265472	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000265500	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000265520	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000276572	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000458129	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000458157	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000458177	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000458204	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000458232	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000458253	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000458281	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000469003	192.168.4.10	192.168.4.11	4333	1512	UDP
4457...	14:22:40,000632558	192.168.4.10	192.168.4.11	4333	1512	UDP

Fig. 7.24 Scheduling performed in 100 μ s time slots for the TSN filter.

Thick lines separate cycles, approximately every 200 μ s. The TSN flow is perfectly placed in its assigned transmission slot. In addition, the receiving timestamp from **Figure 7.24** marks the duration of the intervals.

1. First cycle (frames from 14:22:40,000074332 to 14:22:40,000086436): 191.045 μ s, in which only 12.104 μ s (first group of eight frames) there is a transmission of the TSN flow (it should be during 100 μ s).
2. Second cycle (frames from 14:22:40,000265377 to 14:22:40,000276572): 192.752 μ s. Only 11.195 μ s (second group of eight frames) are used to send TSN flow.
3. Third cycle (frames from 14:22:40,000458129 to 14:22:40,000469003): 174.429 μ s of time interval, emitting the TSN flow during 10.874 μ s (last group of eight frames).

If we compare these values with the measured desynchronization (in the order of tens or hundreds of microseconds), the TAS cycle durations make sense. Nevertheless, the time during which the TSN flow is being sent is too short. The difference with the results shown in **Figure 7.22** is that since the best-effort traffic has been added and the TAS interval lowered to 200 μ s, we may have surpassed the system limits.

Regarding the bandwidth, it is expected that the value obtained for the TSN flow is equivalent to the transmit rate multiplied by the transmit time. iperf computes

and present this value in the console output, as shown in **Figure 7.25**: 469 Mbps. The theoretical value is 940Mbps (for Gigabit Ethernet links [59]). This means that the bandwidth reservation performed is accurate, because the TSN flow takes the 50% of the scheduled interval.

```

-----
Server listening on UDP port 4333
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[ 3] local 192.168.4.11 port 4333 connected with 192.168.4.10 port 37241
[ ID] Interval      Transfer    Bandwidth    Jitter    Lost/Total Datagrams
[ 3] 0.0-11.9 sec   665 MBytes  469 Mbits/sec  0.097 ms   0/474396 (0%)

```

Fig. 7.25 Bandwidth obtained by the iperf flow when TSN is active.

7.2.3 Test conclusions

The use of iperf traffic in order to determine the prototype characteristics has been useful. It has allowed to lower the time interval to the *queuing disciplines* limit. We have also seen that the desynchronization of gPTP and *phc2sys* is a source of the inaccuracy that has been measured. The main conclusions are:

1. Iperf traffic generation has permitted to find the system limits, since it is a much better traffic generator than the OPC-UA Publish/Subscribe traffic in the NodeJS App.
2. Our scheduler is able to support 100 μ s time slots. This means that the minimum cycle for a whole TAS cycle is 200 μ s for the current implementation in Linux Systems.
3. Desynchronization, mostly from *phc2sys* services, brings inaccuracy to the length of the intervals and the time slots. Since the desynchronization offset may reach hundreds of microseconds, the endpoints used in the prototype should enhance in performance to lower it. By increasing its computational resources and minimizing the number of running processes and services, endpoints should get their synchronization more precise.
4. Bandwidth reservation is possible for constant bitrate flows. The bitrate obtained is always equivalent to the percentage that the transmission time slot takes inside the interval, multiplied by the bitrate of the TSN flow.
5. The transmission jitter is also limited. Its maximum value is the result of subtracting the TAS cycle length from the length of the slot assigned to the TSN flow. For example, in our prototype, we have measured a

minimum interval of the TAS is 200 μs and a minimum time slot of 100 μs , so the maximum jitter that the TSN configuration offers is 100 μs .

CHAPTER 8. CONCLUSIONS AND FUTURE LINES

8.1 Conclusions

Our goal was to design a prototype of SDN-controlled TSN endpoints. This work has applications in scenarios related to time-sensitive networking, such as Industry 4.0, Smart Vehicles, Internet of Things, or 5G, among others).

Our prototype is able to integrate in an SDN-controlled network that unifies all industrial systems to the same infrastructure, guaranteeing the requirements of the systems that need strict temporal requirements. This means that all devices in a factory will be able to communicate optimally, despite sharing communication resources with other systems and avoiding network congestion. Our prototype is based on standards. This is something important in this sector, since most of the implementations are device-dependent and its interoperability is usually not guaranteed.

The proposed solution uses OPC-UA to configure any device in an Ethernet network to establish time-sensitive domains (TSN flows) and guarantee a given QoS. The prototype designed in this project is able to dynamically configure TSN domains across the network. Furthermore, the use of OPC-UA enhances the solution's adaptability, since it can be used in a wide variety of contexts. In short, any device can easily run OPC-UA and communicate with the CUC to take part in any TSN domain. To the best of our knowledge, this is the first open-source implementation able to do that.

In relation with the performance of the prototype, the CUC is able to give configuration to the endpoints. The CUC provides the endpoints configuration regarding the scheduling they need to perform in order to satisfy the time requirements. This proves that the architecture is valid and the CUC is able to manage time-sensitive endpoints in a network. With the integration of a CNC, overall network performance can be tested; however, for this project, we have limited ourselves to use the current implementation without a network controller to find the endpoints' time-sensitive capacities.

The tested endpoints are able to perform traffic scheduling, ensuring bandwidth reservation to the TSN flows that may require it. The endpoints are able to perform scheduling in scales below the microsecond, using a minimum time slot size of 100 μ s and, consequently, a minimum TAS cycle of 200 μ s (two 100 μ s time slots). We must emphasize that the endpoints' clock synchronization is not ideal, with inaccuracies close to 100 μ s if the offsets from all components are added. Therefore, if endpoint PCs with better clocks were used, with lower *phc2sys* synchronization offsets, the overall performance of the prototype should improve.

The performance of the traffic generator in the NodeJS App is not accurate. Therefore, a different environment has to be used to generate time-sensitive

OPC-UA traffic. In order to stress the system and find its performance boundaries, we used iperf traffic, which worked quite well.

8.2 Future lines of development

The prototype, based on the *fully centralized* architecture, can be extended in many ways. Once a CNC implementation is available and the communication between CUC and CNC is fully specified, both can join to complete the model specified in the IEEE 802.1Qcc standard. This complete model can be added in an SDN controller. **Figure 8.1** shows an overview of an industrial network scenario and where our prototype would take part.

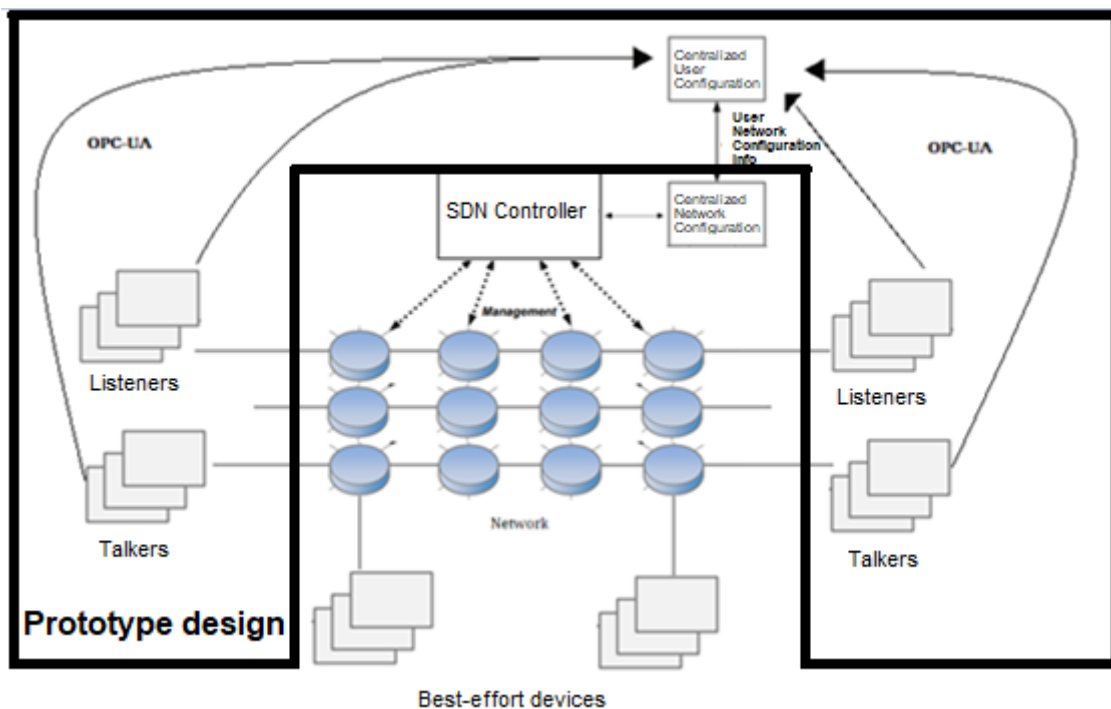


Fig. 8.1 Global industrial network scenario.

Aside from that, the tests performed for the NodeJS Application in the Talker endpoint determine that the OPC-UA traffic generation is not performed optimally. Even though different traffic sources have been used to get to know the TSN performance of the Talker, the ideal scenario would be to have all devices communicating by using OPC-UA.

The current problem in our solution is that Javascript does not offer strict time management. It inherently introduces tens of milliseconds of jitter. As a solution, a new module needs to be implemented to generate OPC-UA traffic with rigorous time synchronization locked with the system clock. We would further modify **Figure 5.8**, which would give us **Figure 8.2**.

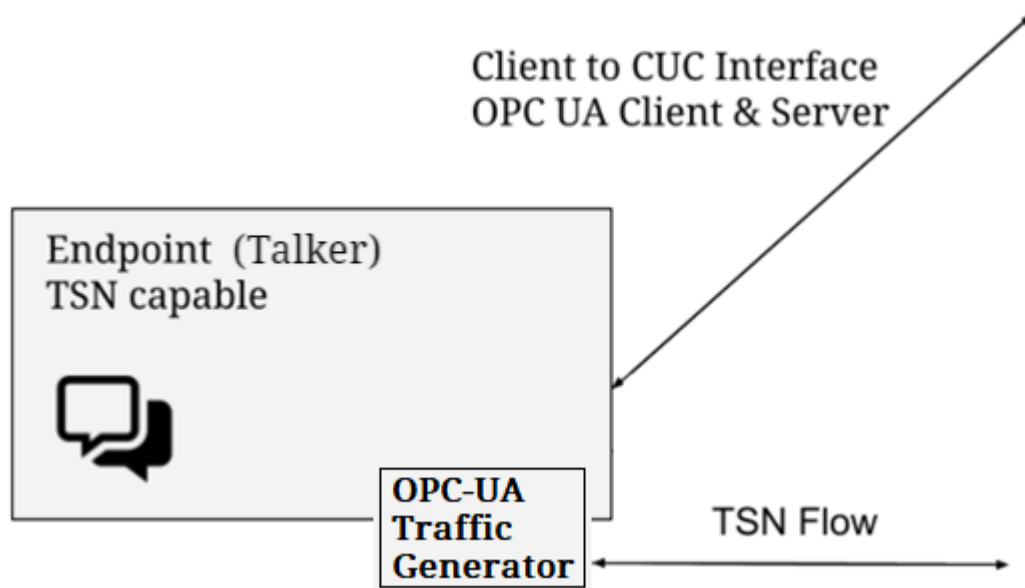


Fig. 8.2 Modification on the Talker endpoint design.

8.3 Sustainability considerations

Nowadays, the majority of the industrial infrastructure is outdated. Old devices and technologies tend to be inefficient in terms of power consumption and performance. By replacing these devices, energy is saved and the power efficiency is higher. Besides the devices' qualities, these devices are commonly used in independent systems inside the all-embracing industrial framework. Joining all of them in the same network results in saving physical resources, such as links and bridges. Also, the controllers of these systems can be unified in a single physical device, because it is able to communicate directly with all components participating in the network.

In addition, by using software-defined networks, all the network elements can be managed by the controller. This means that power management can be performed, so that unused devices can be turned off, only using the necessary number of resources.

TSN alone does not provide quantifiable benefits to power consumption. In fact, some of its features (e.g. path redundancy) may introduce extra processing and load into the network. Nevertheless, TSN is the key solution, as it is capable of joining all components in one network. Therefore, the benefits presented in the current paradigm are all possible to the use of time-sensitive standards.

REFERENCES

- [1] Nelly Ayllon, "Profinet vs ethernet: definitions and a comparison" [Online]. Available: <https://us.profinet.com/profinet-vs-ethernet-definitions-and-a-comparison/> [Accessible 06 07 2021] [Published 06 26 2020]-
- [2] CSS Electronics, "CAN Bus Explained – A Simple Intro (2021)" [Online]. Available: <https://www.csselectronics.com/screen/page/simple-intro-to-can-bus/language/en>. [Accessible 06/07/2021] [Last update 2021].
- [3] VMWare, "Software-Defined Networking" [Online]. Available: <https://www.vmware.com/topics/glossary/content/software-defined-networking> [Accessed 06 07 2021].
- [4] Time Sensitive Networking Task Group, "Time-Sensitive Networking (TSN) Task Group" [Online]. Available: <https://1.ieee802.org/tsn/>. [Accessed 06 01 2021].
- [5] IEEE Standards Association, "Bridges and Bridged Networks. Amendment 31: Stream Reservation Protocol (SRP) Enhancements and Performance Improvements", Chapter 46 "Time-Sensitive Networking (TSN) Configuration" [Online]. Available (private) https://standards.ieee.org/standard/802_1Qcc-2018.html [Accessed 06 01 2021] [Published 06 14 2018].
- [6] Cisco, "Cisco IE-4000 product family" [Online]. Available: https://www.cisco.com/c/en/us/td/docs/switches/lan/cisco_ie4000/tsn/b_tsn_ios_support/b_tsn_ios_support_chapter_01.html#concept_hxr_gz3_1b [Accessed 06 01 2021] [Last update: 11 11 2018].
- [7] OPC Foundation, "OPC Unified Architecture" [Online]. Available <https://opcfoundation.org/about/opc-technologies/opc-ua/> [Accessed 06 01 2021].
- [8] Microchip Developer, "Data Link Layer (Layer 2)" [Online picture]. Available: https://microchipdeveloper.com/local--files/tcpip:tcp-ip-data-link-layer-layer-2/ethernet_wifi_frames.JPG [Accessed 06 07 2021].
- [9] mespresso, "Ethernet. Estandar IEEE 802, subcapas LLC y MAC" [Online]. Available: <https://mespresso.wordpress.com/2017/02/08/ethernet-estandar-ieee-802-subcapas-llc-y-mac/> [Accessed 06 01 2021] [Published 02 07 2018].
- [10] IEEE Standards Association, "Bridges and Bridged Networks" [Online]. Available (private): https://standards.ieee.org/standard/802_1Q-2018.html [Accessed 06 01 2021] [Published 05 07 2018].
- [11] Sebastian Wiesinger, Network Engineering – Stack Exchange, "Why and how are Ethernet Vlans tagged?" [Online]. Available: <https://networkengineering.stackexchange.com/questions/6483/why-and-how-are-ethernet-vlans-tagged?>

- how-are-ethernet-vlans-tagged [Accessed 06 01 2021] [Published 02 25 2014].
- [12] Hyung-Taek Lim, Daniel Herrscher, Martin Johannes Walzl and Firas Chaari, "Performance Analysis of the IEEE 802.1 Ethernet Audio/Video Bridging Standard", figure 2 [Online]. Available: https://www.researchgate.net/publication/262323002_Performance_Analysis_of_the_IEEE_8021_Ethernet_AudioVideo_Bridging_Standard [Accessed 06 07 2021] [Published 03 2012].
- [13] Henning Puttnies, Peter Danielis, Enkhtuvshin Janchivnyambuu and Dirk Timmermann, "A Simulation Model of IEEE 802.1AS gPTP for Clock Synchronization in OMNeT++" [Online]. Available: <https://easychair.org/publications/open/Q4kL> [Accessed 06 01 2021].
- [14] Silicon Labs, "The IEEE 1588 Standard" [Online]. Available: <https://www.silabs.com/whitepapers/ieee-1588-standard> [Accessed 06 01 2021].
- [15] Wikipedia, "Time-Sensitive Networking" [Online]. Available: https://en.wikipedia.org/wiki/Time-Sensitive_Networking, section 6 [Accessed 06 01 2021].
- [16] IEEE Standards Association, "Bridges and Bridged Networks", section 8.6.8.4 [Online]. Available (private) https://standards.ieee.org/standard/802_1Q-2018.html [Accessed 06 01 2021] [Published 05 07 2018].
- [17] Wikipedia, "Credit-based fair queuing" [Online]. Available: https://en.wikipedia.org/wiki/Credit-based_fair_queuing [Accessible 06 07 2021].
- [18] IEEE Standards Association, "Bridges and Bridged Networks", section 8.6.8.2 [Online]. Available (private) https://standards.ieee.org/standard/802_1Q-2018.html [Accessed 06 01 2021] [Published 05 07 2018].
- [19] Time Sensitive Networking Task Group, "Time-Sensitive Networking (TSN) Task Group" [Online]. Available: <https://1.ieee802.org/tsn/>, "Ongoing TSN Projects" section. [Accessed 06 01 2021].
- [20] Benoit Claise, Joe Clarke and Jan Lindblad, "Network Programmability With YANG", Chapter 1 "The Network Management World Must Change: Why Should You Care?" and Chapter 2 "Data Model-Driven Management", pp. 2-95, Pearson Addison-Wesley, USA [Published in 01 18 2019].
- [21] SNMP Center, "Why use NETCONF/YANG when you can use SNMP and CLI?" [Online]. Available: <https://snmpcenter.com/why-use-netconf/> [Accessed 06 01 2021].

- [22] Benoit Claise, Joe Clarke and Jan Lindblad, "Network Programmability With YANG", Chapter 3 "YANG Explained", pp. 96-149, Pearson Addison-Weasley, USA [Published in 01 18 2019].
- [23] IETF, "RFC 3444 – Information Models and Data Models" [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc3444> [Accessed 06 01 2021] [Published 01 2003].
- [24] Matt Albrecht, "Learn YANG. Full Tutorial For Beginners" [Online]. Available: <https://ultraconfig.com.au/blog/learn-yang-full-tutorial-for-beginners/> [Accessed 06 01 2021] [Published 01 03 2020].
- [25] Benoit Claise, Joe Clarke and Jan Lindblad, "Network Programmability With YANG", Chapter 4 "NETCONF, RESTCONF, and gNMI Explained", pp. 158-189, Pearson Addison-Weasley, USA [Published in 01 18 2019].
- [26] Benoit Claise, Joe Clarke and Jan Lindblad, "Network Programmability With YANG", Chapter 4 "NETCONF, RESTCONF, and gNMI Explained", pp. 190-213, Pearson Addison-Weasley, USA [Published in 01 18 2019].
- [27] "AGM" – Laiarroz Elektronika, "OPC: Desde el clásico al nuevo OPC-UA" [Online]. Available: <https://larraioz.com/articulos/opc-desde-el-clasico-al-nuevo-opc-ua> [Accessed 06 01 2021] [Published 10 04 2016].
- [28] B&R Automation, "OPC UA" [Online]. Available: <https://www.br-automation.com/es-es/tecnologias/opc-ua/> [Accessed 06 07 2021].
- [29] B&R Automation, "OPC UA para el control de movimiento, seguridad y aplicaciones en tiempo real" [Online]. Available: <https://www.br-automation.com/es-es/tecnologias/opc-ua/opc-ua-para-el-control-de-movimiento-seguridad-y-aplicaciones-en-tiempo-real/> [Accessed 06 01 2021].
- [30] TTTech Industrial, "Time Sensitive Networking and OPC UA (OPC UA over TSN)" [Online]. Available: <https://www.tttech-industrial.com/technologies/opc-ua-over-tsn/> [Accessed 06 01 2021].
- [31] John S Rinaldi, "OPC UA Client vs Server" [Online]. Available: <https://www.rtautomation.com/rtas-blog/opc-ua-client-vs-server/> [Accessed 06 01 2021] [Published 10 16 2018].
- [32] Unified Automation, "UA Bundle SDK .NET", section "Address Space Concepts" [Online]. Available: <https://documentation.unified-automation.com/uasdkdotnet/2.5.2/html/L2UaAddressSpaceConcepts.html> [Accessed 06 01 2021].
- [33] Exorint, "The introduction of opc ua publish-subscribe and its importance to manufacturers" [Online]. Available: <https://www.exorint.com/en/blog/the-introduction-of-opc-ua-pubsub->

- publish-subscribe-and-its-importance-to-manufacturers [Accessed 06 01 2021].
- [34] IEEE Standards Association, “Bridges and Bridged Networks. Amendment 31: Stream Reservation Protocol (SRP) Enhancements and Performance Improvements”, Chapter 46 “Time-Sensitive Networking (TSN) Configuration” [Online]. Available (private) https://standards.ieee.org/standard/802_1Qcc-2018.html [Accessed 06 01 2021] [Published 06 14 2018].
- [35] Intel Corporation, “Adopting Time-Sensitive Networking (TSN) for Automation Systems”, section “TSN Products from Intel” [Online]. Available: <https://software.intel.com/content/www/us/en/develop/articles/adopting-time-sensitive-networking-tsn-for-automation-systems-0.html> [Accessed 06 01 2021] [Last updated 03 13 2020].
- [36] Intel Corporation, “Controladora Ethernet Intel® I210” [Online]. Available: <https://www.intel.es/content/www/es/es/products/details/ethernet/gigabit-controllers/i210-controllers.html> [Accessible 06/07/2021].
- [37] Gopiga S K, Keerthivasan A S, Nikhil Vannan K, Selva Suba Jenifer J, Shriya Chaurasia, Suriya Narayanan P V and Thangavaila K T, “How to run OPC UA PubSub on real-time Linux and TSN using open62541” [Online]. Available: <https://www.kalycito.com/how-to-run-opc-ua-pubsub-tsn/> [Accessed 06 01 2021].
- [38] ankita_saini, “Introduction to Linux Operating System” [Online]. Available: <https://www.geeksforgeeks.org/introduction-to-linux-operating-system/> [Accessed 06 01 2021] [Published 02 18 2021].
- [39] Rubén Velasco, “Kernel Linux, descubre cómo es el corazón de este sistema operativo” [Online]. Available: <https://www.softzone.es/programas/linux/kernel-nucleo-linux/> [Accessed 06 01 2021] [Last update 03 30 2021].
- [40] Grégoire Monet, “From Sockets to NIC: A Big Picture” [Online]. Available: <https://medium.com/swlh/from-sockets-to-nic-a-big-picture-7494356cfcd4> [Accessed 06 01 2021] [Published 12 10 2020].
- [41] Dan Siemon, “Queuing in the Linux Network Stack” [Online]. Available: <https://www.linuxjournal.com/content/queueing-linux-network-stack> [Accessed 06 01 2021] [Published 11 23 2013].
- [42] Intel Corporation, “Intel® Ethernet Controller I210 Datasheet”, Cover of the document [Online]. Available: <https://datasheet.octopart.com/WGI210IS-S-LJXX-Intel-datasheet-138896048.pdf> [Accessed 06 01 2021] [Published 06 2018].

- [43] IEEE Standards Association, “Bridges and Bridged Networks”, section 8.6.8, figure 8.6 [Online]. Available (private) https://standards.ieee.org/standard/802_1Q-2018.html [Accessed 06 01 2021] [Published 05 07 2018].
- [44] IEEE Standards Association, “Frame Replication and Elimination for Redundancy”, section 9.1.1.6, table 9.1 [Online]. Available (private): https://standards.ieee.org/standard/802_1CB-2017.html [Accessed 06 01 2021] [Published 09 28 2017].
- [45] IEEE Standards Association, “Frame Replication and Elimination for Redundancy”, section 10.5.1.6, table 10.2 [Online]. Available (private): https://standards.ieee.org/standard/802_1CB-2017.html [Accessed 06 01 2021] [Published 09 28 2017].
- [46] IEEE Standards Association, “Bridges and Bridged Networks. Amendment 31: Stream Reservation Protocol (SRP) Enhancements and Performance Improvements”, section 46.2.5.1.3, table 46.15 [Online]. Available (private) https://standards.ieee.org/standard/802_1Qcc-2018.html [Accessed 06 01 2021] [Published 06 14 2018].
- [47] Ramon Serna Oliver, Silviu S. Craciunas, Wilfried Steiner, “IEEE 802.1Qbv Gate Control List Synthesis using Array Theory Encoding” [Online]. Available: https://www.researchgate.net/publication/324476736_IEEE_8021Qbv_Gate_Control_List_Synthesis_using_Array_Theory_Encoding [Accessed 06 11 2021] [Published 04 2018].
- [48] Ilya Grigorik and Surma, “Introduction to HTTP/2” [Online]. Available: <https://developers.google.com/web/fundamentals/performance/http2> [Accessed 06 01 2021] [Last update 12 02 2019].
- [49] SoC-e, “1G MTSN-Multiport TSN Switch IP Core” [Online]. Available: <https://soc-e.com/mtsn-multiport-tsn-switch-ip-core/> [Accessed 06 01 2021].
- [50] Josep Oriol Castaño Cid, “Proves amb equipament Time-Sensitive Networking (TSN)” [Online]. Available: <https://upcommons.upc.edu/handle/2117/119626> [Accessed 06 20 2021] [Published 07 27 2018].
- [51] Joan Feliu Castaño Cid, “Design and proof of concept of a centralized controller for time-sensitive networks” [Online]. Available: <https://upcommons.upc.edu/handle/2117/119626> [Accessed 06 20 2021] [Published 07 13 2018].
- [52] Vinicius Gomes, “TSN Documentation Project for Linux”, section “Synchronizing Time with Linux* PTP” [Online]. Available: <https://tsn.readthedocs.io/timesync.html> [Accessed 06 01 2021]

- [53] Ubuntu manuals, “ptp4l” [Online]. Available: <http://manpages.ubuntu.com/manpages/xenial/man8/ptp4l.8.html> [Accessed 06 01 2021].
- [54] Ubuntu manuals, “phc2sys” [Online]. Available: <http://manpages.ubuntu.com/manpages/cosmic/man8/phc2sys.8.html> [Accessed 06 01 2021].
- [55] Gabriel David Orozco Urrutia and Jordi Cros Mompart, “TSN-CNC-CUC-UPC” [Online repository]. Available (private access): <https://github.com/gabriel-david-orozco/TSN-CNC-CUC-UPC>.
- [56] Ubuntu, “Ubuntu 20.04.2.0 LTS (Focal Fossa)” [Online downloadable OS Image]. Available: <https://releases.ubuntu.com/20.04/> [Accessed 06 01 2021] [Released 04 23 2020].
- [57] Node JS, “NodeJS Download for Windows” [Online downloadable development software]. Available: <https://nodejs.org/en/> [Accessed 06 01 2021] [Released 04 21 2020].
- [58] Jordi Cros Mompart, “Development of an SDN control plane for Time-Sensitive Networking (TSN) endpoints” Downloadable content [Online]. Available: <https://drive.google.com/file/d/1DPcwr7XWd9zfzqvXiy0uo7xjvsNd7YTX/view?usp=sharing> [Accessed 06 20 2021] [Published 06 20 2021]
- [59] Gigabit Wireless, “What is the actual maximum throughput on Gigabit Ethernet?” [Online]. Available: <https://www.gigabit-wireless.com/gigabit-wireless/actual-maximum-throughput-gigabit-ethernet/> [Accessed 06 13 2021].

GLOSSARY

ACID	Atomicity, Consistency, Independence and Durability
API	Application Programming Interface
BMCA	Best Master Clock Algorithm
BS	Base Station
CBS	Credit Based Shaper
CDP	Cisco Discovery Protocol
CID	Caller Identifier
CLI	Command Line Interface
CNC	Centralized Network Configuration
CQF	Cyclic Queuing and Forwarding
CSMA/CA	Carrier Sense Multiple Access / Collision Avoidance
CSMA/CD	Carrier Sense Multiple Access / Collision Detection
CUC	Centralized User Configuration
DCOM	Distributed Component Object Model
ERP	Enterprise Resource Planning
EST	Enhancements for Scheduled Traffic
ETF	Earliest TxTime First
ETS	Enhanced Transmission Selection
FDP	Foundry Discovery Protocol
FIFO	First In First Out
FRER	Frame Replication and Elimination for Reliability
GCL	Gate Control List
gPTP	Generic Precision Time Protocol
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IS-IS	Intermediate System to Intermediate System
ISIS-PCR	IS-IS Path Control Reservation
ITU-T	International Telecommunication Union - Telecommunication
JSON	JavaScript Object Notation
LAG	Link Aggregation Group
LAN	Local Area Network
LLC	Logical Link Control
LLDP	Link Layer Discovery Protocol
LUC	Logic Unit Center
MAC	Medium Access Control
MB	MegaByte

MIB	Management Information Base
MMRP	Multiple MAC Registration Protocol
MRP	Multiple Registration Protocol
MSRP	Multiple Stream Reservation Protocol
MVRP	Multiple VLAN Registration Protocol
NAT	Network Address Translation
NETCONF	Network Configuration Protocol
NIC	Network Interface Card
ODL	OpenDaylight
ONOS	Operating Network Operating System
OPC-UA	OPC Unified Architecture
OS	Operating System
OSI	Open Systems Interconnection
OUI	Organizational Unique Identifier
PCE	Path Computation Element
PHY	Physical Interface
PSFP	Per Stream Filtering and Policing
Pub/Sub	OPC-UA Publish / Subscribe communication
QDISC	Queuing Disciplines
QoS	Quality of Service
REST	Representational State Transfer
RESTCONF	Network Configuration Protocol over REST
RFC	Request for Comments
RPC	Remote Procedure Call
SDN	Software Defined Networking
SNMP	Simple Network Management Protocol
SPB	Shortest Path Bridging
SPF	Shortest Path First
SRP	Stream Reservation Protocol
SSH	Secure Shell
STF	Stream Transfer Function
STP	Spanning Tree Protocol
TAI	International Atomic Time
TAPRIO	Time Aware Priority
TAS	Time Aware Scheduler
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TLV	Type Length Value
TSN	Time Sensitive Networking
UDP	User Datagram Protocol
UID	Unique Identifier
UNI	User to Network Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

UTC	Universal Time Coordinated
VID	VLAN Identifier
VLAN	Virtual Local Area Network
WiFi	Wireless Fidelity
XML	Extensible Markup Language
YANG	Yet Another Next Generation

ANNEX A. IEEE AND TSN STANDARDS

This annex contains all the protocols and standards that may be related to the project, but do not have any impact in its development. Some of them belong to the field of the Link Layer protocols, such as *Spanning Tree Protocol*, *Shortest Path Bridging / Shortest Path First* and *Link Layer Discovery Protocol*. The following ones are closer to Time Sensitive Networking, trying to provide new features to the setting of a TSN domain and flows in a network. Others are mentioned since they overlap in some aspects.

By introducing these standards, including all the ones mentioned in Chapter 1, we want the reader with a general overview of the whole TSN-related content described in this project. Most of them will take part in the future lines of this field and others are mentioned in order to state a possible overlapping in functionalities.

A.1 IEEE 802.1d – Spanning Tree Protocol

Spanning Tree Protocol (STP) [1] is a generic used protocol in LAN, aimed to avoid loop forwarding issues at layer 2 networks.

1. From link cost, priority or MAC address, the *root switch* is determined inside a LAN.
2. The rest of the switches set the *root port*, which is the one with minor cost to the root switch.
3. For every link, we select as *designed port* the one with less cost to the root switch.
4. If in a link there is a designed port and the other one is not the root port, this last will be *disabled* from being used.

SPT is implemented in each different VLAN in the infrastructure. As VLANs are being used to distinguish between different TSN streams, it is important not to perform SPT [2] in them. By not using Spanning Tree Protocol, the virtual network does not get its links disabled, permitting redundancy and other features needed to comply with TSN standards.

A.2 IEEE 802.1aq – Shortest Path Bridging and Shortest Path First

The reason to consider these protocols is that some TSN standards (specifically 802.1Qbv and 802.1Qca) mention them and there could be a need of an analysis of them for future lines after this project.

The Shortest Path Bridging (SPB) [3] standard aims to be a substitute of STP, offering several advantages:

1. More scalability and convergence speed.
2. More robust meshed topologies (no loop avoidance)
3. Forwarding redundancies, easing the recovery process against failures and load distribution.

Even though this protocol looks better than STP, it is recommended not to use it either, since it still interferes with critical TSN functionalities.

Apart from SPB, there is also Shortest Path First (SPF) with all its variants, since they value the different link features for future decisions.

A.3 IEEE 802.1ab – Link Layer Discovery Protocol

This protocol, also known as *Station and Media Access Control Connectivity Discovery* is fundamental for the implementation of a TSN network. It lets devices announce its identity, properties and adjacent connections in the network. It is the standardization of several proprietary protocols, such as Cisco Discovery Protocol (CDP) or Foundry Discovery Protocol (FDP).

Typically, information obtained thanks to Link Layer Discovery Protocol (LLDP) [4] is stored in the Management Information Base (MIB). This information can be queried using, typically, Simple Management Network Protocol (SNMP). Information obtained contains the following:

1. System name and description.
2. Port name and description.
3. VLANs in the device.
4. Management IP.
5. System capabilities (including TSN features).
6. Low level information (LLC/PHY).
7. Link aggregations.

Within a certain interval, devices multicast their LLDP data in Type Length Value (TLV) format and a given multicast address.

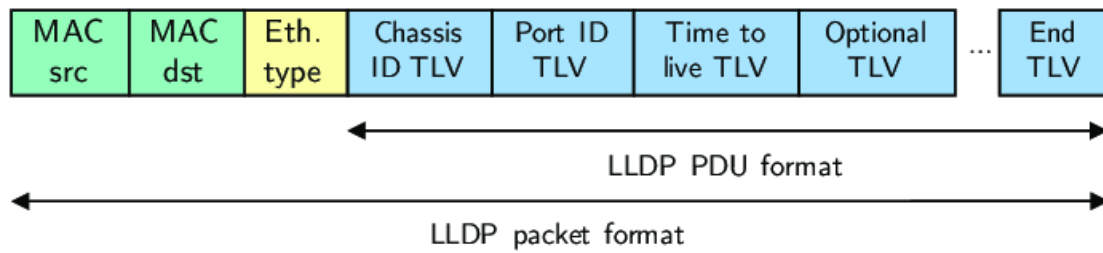


Fig. A.1 LLDP payload format [5].

Software Defined Networks (SDN) data plane is orchestrated by the controller, so it needs to receive all LLDP packets from all network elements to perform a good network mapping [6]. The current trend on network management is to use LLDP information by NETCONF clients. This means that the CNC in the TSN architecture is likely to use LLDP information to build the network topology.

A.4 IEEE 802.1Qat - Stream Reservation Protocol

Since 2011, this standard is included in 802.1Q. It defines the *Stream Reservation Protocol* (SRP) [7]. Basically, it manages data flows from the emitter to the receiver, including the network bridges. By this way, it is possible to perform end-to-end reservations on the resources in order to guarantee bandwidth and latency. Endpoints can declare themselves as talkers or listeners, propagating the information to the network elements. These nodes will check if the bandwidth and latency requirements can be satisfied with other current active flows and forward the request if possible. If not possible, they communicate the failure reason back to the announcer.

SRP is used over *Multiple Mac Registration Protocol* (MMRP), *Multiple VLAN Registration Protocol* (MVRP) and *Multiple Stream Registration Protocol* (MSRP). All of them rely on *Multiple Reservation Protocol* (MRP). MRP is used to propagate all the information through the LAN, upper protocols are used to carry the information related to reservations.

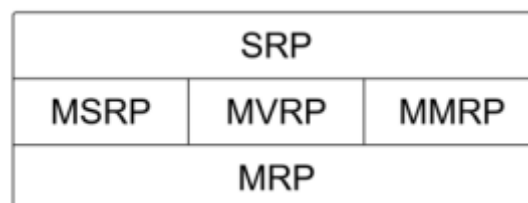


Fig. A.2 Stream Reservation Protocol stack.

All the information shared with SRP is used by network elements in order to implement the traffic scheduling and shaping correctly. Essentially, SRP operates in the following manner:

1. Talker announces the flux of data being sent.
2. Every network element that takes part in the path registers the flow.
3. Worst latency case is computed (with the collaboration of 802.1AS to synchronize all clocks).
4. The flow domain is created.
5. Resource reservation is performed.
6. Future announcements may not be performed, since the resources are busy.

A.5 IEEE 802.1Qch – Cyclic Queuing and Forwarding

This substandard is on Annex T of the IEEE 802.1Q-2018 [8]. It specifies a traffic scheduling method that may offer a deterministic latency on TSN networks. The principle in which *Cyclic Queuing and Forwarding* (CQF) operates is that flows are transmitted and loaded to the queues in a cyclic way. This means that time is divided in equal parts and in each cycle the nodes in the network perform a transmission and a reception. It is important that each network device is synchronized, in order to apply the same time intervals. By this way, if we consider that the maximum delay between nodes is the duration of the interval, it is easy to get the total maximum delay on the flow, by the following expression, where h is the number of hops and d interval duration.

$$delay = (h + 1) \cdot d \quad (\text{A.1})$$

It is very important the correct decision of the interval duration, keeping in mind all the congestion status the nodes may suffer, they must ensure a time synchronized transmission. If not, the whole CQF system becomes invalid. A correct implementation of this protocol needs to consider more parameters, such as the non-homogeneity of links in the network, making the previous expression more complex.

A.6 IEEE 802.1Qci – Per Stream Filtering and Policing

This substandard, also found in the IEEE 802.1Q-2018 [9], brings the capacity of filtering frames at the ingress ports depending on the reception time, bandwidth and other ratios [10]. Thanks to that filtering, there is a protection against bad use of the bandwidth and excessive burst size from malicious endpoints. By this way, these traffics are not retransmitted to the network, avoiding congestion or collapse. *Per Stream Filtering and Policing* (PSFP) can be applied in three different tables:

1. *Stream Filter*: ordered list of filters that determine the actions applied to the frames of a same flux.
2. *Stream Gate*: has parameters for every different flux priority, independently from the stream identifier.
3. *Flow Meter*: based on properties of fluxes, such as Committed Information Rate, Committed Burst Size, among others.

A.7 IEEE 802.1Qbu / 802.3br – Frame Preemption

Included in the IEEE 802.1Q-2014 standard and describes the methodology in which a frame transmission can be interrupted. For example, to avoid frames to overlap with the guard interval, and resume it later in the next interval, when the priority of the flow allows it to be sent again. It needs specific detail of the physical channel it is working on, that is the reason why some part of it is described in the Ethernet standard, IEEE 802.3.

In summary, the partial frame is ended with a CRC and the receiving switch stores all the content in a buffer, waiting for the arrival of the rest of the frame. Furthermore, it permits the reduction of the guard interval, optimizing the use of the bandwidth when traffic scheduling is applied [11].

A.8 IEEE 802.1Qcr – Asynchronous Traffic Shaping

This standard provides the processes for network elements in order to perform traffic shaping between links. The goal is to satisfy the needs of the flow, which may require deterministic latency or zero congestion loss. The difference between this standard and other TSN standards is that this one holds the traffic shaping in a non-deterministic network, meaning that there is no clock synchronization between devices [12].

Thus, this standard may look for similar goals than this project, but not in the same network scenario.

A.9 IEEE 802.1Qca – Path Control and Reservation

Included in the IEEE 802.1Q-2014 document [13]. It provides an extension of the forwarding protocol Intermediate System to Intermediate System (IS-IS) and contains the following functionalities:

1. Tree establishment to forward frames.
2. Use of IS-IS to communicate the bandwidth computed by the Path Computation Element (PCE).
3. Redundancy in the tree establishment.

Each VLAN ID (VID) is associable to one or more trees and other logic included in IS-IS, specifically the Path Reservation Control. This independence can be maintained between different VIDs. PCE is external to IS-IS protocol and aims to determine and describe explicitly the forwarding tree. There can be more than one PCE in a network and each one manages a traffic in a route, avoiding the overloading of links. This can entail not using the shortest paths.

In general terms, the ISIS-PCR announces in the network the bandwidth assignments. If MSRP does not assign these assignments, ISIS-PCR can do it. Then, it performs the following functionalities [14]:

1. Route control to not use always the shortest path.
2. Communicate to other network elements the bandwidth reservations for every flow.
3. Control over failures and overloading thanks to path redundancy, the standard introduced next.
4. Sending of control information thanks to ISIS-PCR extension.

A.10 IEEE 802.1CB – Frame Replication and Elimination for Reliability

This specification [15] contains a description of the processes needed in order to avoid loss of frames in a network either by congestion, link failure or other issues. In general terms, identification and replication mechanisms for frames are provided to perform redundancy in the network and elimination of duplicated frames. With all these handlings, it is easy to realize that a failure in the network would not bring any issue on the flow transmission, since it would be forwarded through two disjoint paths. Thus, the loss probability is highly reduced.

Frame Replication and Elimination for Reliability (FRER) performs the following functions:

1. Packet replication: duplicates frames and forwards them in two disjoint paths. This increments the resiliency of the whole network against the flow. If both duplicated frames reach the destination, it also performs the deletion of one of them.
2. Unicast and Multicast: possibility of one or more listeners in the configuration.
3. Flexible positioning: FRER can be applied in any of the nodes of the path for a flow.
4. Error detection: provides mechanisms to the receivers to detect the failure of one of the redundant flows. Apparently, it can be difficult when the correct flow is being received, even from another interface.
5. Interoperability and backwards compatibility: with similar previous protocols and devices that does not support FRER.

Some of these features are shown in the **Figure A.3**, in where there are nodes that generate a replication and some others perform the deletion of duplications.

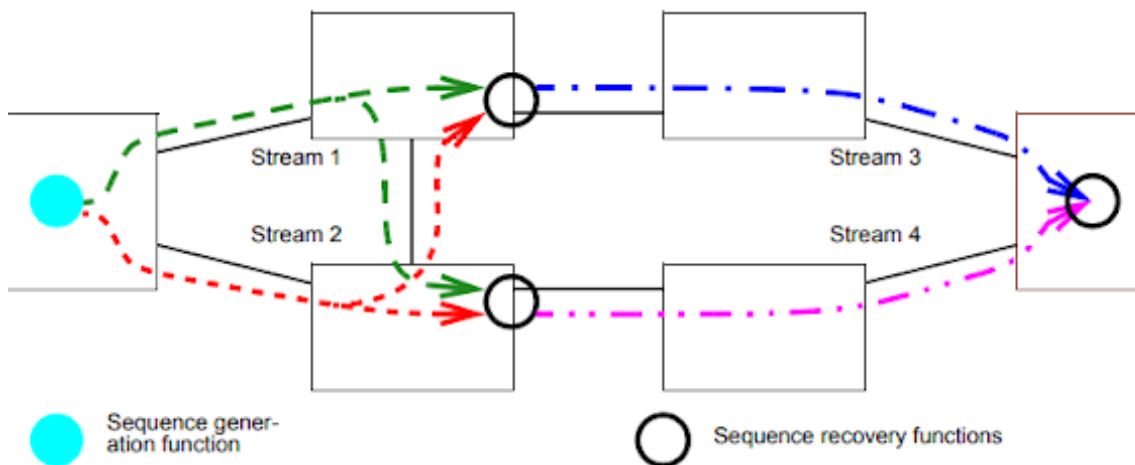


Fig. A.3 FRER sample schema [15].

This standard does not include the procedures to find disjoint paths in the network. Instead, it relies on the IEEE 802.1QCa, already commented in this document (see Section A.9).

A.11 IEEE 802.1CM – Time Sensitive Networking for Fronthaul

This standard [16] aims to define different profiles that select features, configurations and default protocols for the network elements used in the *fronthaul* of mobile networks (4G, 5G and beyond). This kind of streams need important time-sensitive requirements. Options from Layers 1 and 2 from OSI are specified, both for network elements and endpoints, referring to the following aspects:

1. VLAN 802.1Q
2. MAC service specification found at IEEE 802.1AC.
3. MAC/PHY specifications from IEEE 802.3.
4. Traffic interleaving, found in 802.3br.
5. Frame preemption (see section A.7).
6. Temporal synchronization with 802.1AS (see section 1.1.2).
7. Telecom profile specification, from ITU-T G.8275.1.
8. Synchronous Ethernet, specified in the ITU-T G.8261, G.8262 and G.8264.

In addition, fronthaul requirements are specified. Specifically, two classes of requirements, depending on the functionality of the base stations (BS). In addition, it describes how the network elements affect the quality of service of the fronthaul and the impact that the flow control supposes.

A.12 IEEE 802.1AX – Link Aggregation

Even though Link Aggregation may not be directly related to TSN, this standard [17] brings the possibility to aggregate two point-to-point links to form a *Link Aggregation Group* (LAG). Thanks to this, the MAC client considers it as a single link. By this way, the bandwidth is enhanced for those traffics that may need it and the resiliency against failures is higher on the virtual link. Since the LAG is considered as an only link, all route reservation protocols, such as IEEE 802.1Qca and others about deterministic latency, such as IEEE 802.1Qch will consider it as so.

ANNEX B. UNI YANG MODULE

The following tree is a representation of all the fields in the designed YANG module, found in the project's repository [18]:

“Module: ieee802-dot1q-tsn-types-upc-version

```

+--rw tsn-uni
  +--rw stream-list* [stream-id]
    | +--rw stream-id      stream-id-type
    | +--rw request
    | | +--rw talker
    | | | +--rw stream-rank
    | | | | +--rw rank?   uint8
    | | | +--rw end-station-interfaces* [mac-address interface-name]
    | | | | +--rw mac-address      string
    | | | | +--rw interface-name  string
    | | | +--rw data-frame-specification* [index]
    | | | | +--rw index            uint8
    | | | | +--rw (field)?
    | | | | | +--:(ieee802-mac-addresses)
    | | | | | | +--rw ieee802-mac-addresses
    | | | | | | | +--rw destination-mac-address? string
    | | | | | | | +--rw source-mac-address?   string
    | | | | | +--:(ieee802-vlan-tag)
    | | | | | | +--rw ieee802-vlan-tag
    | | | | | | | +--rw priority-code-point? uint8
    | | | | | | | +--rw vlan-id?           uint16
    | | | | | +--:(ipv4-tuple)
    | | | | | | +--rw ipv4-tuple
    | | | | | | | +--rw source-ip-address?   inet:ipv4-address
    | | | | | | | +--rw destination-ip-address? inet:ipv4-address
    | | | | | | | +--rw dscp?                uint8
    | | | | | | | +--rw protocol?            uint16
    | | | | | | | +--rw source-port?         uint16
    | | | | | | | +--rw destination-port?    uint16
    | | | | | +--:(ipv6-tuple)
    | | | | | | +--rw ipv6-tuple
    | | | | | | | +--rw source-ip-address?   inet:ipv6-address
    | | | | | | | +--rw destination-ip-address? inet:ipv6-address
    | | | | | | | +--rw dscp?                uint8
    | | | | | | | +--rw protocol?            uint16
    | | | | | | | +--rw source-port?         uint16
    | | | | | | | +--rw destination-port?    uint16
    | | | +--rw traffic-specification
    | | | | +--rw interval
    | | | | | +--rw numerator?   uint32
    | | | | | +--rw denominator? uint32
    | | | | +--rw max-frames-per-interval? uint16
    | | | +--rw max-frame-size?          uint16

```



```

| | | | +--rw transmission-selection?  uint8
| | | | +--rw time-aware!
| | | |   +--rw earliest-transmit-offset?  uint32
| | | |   +--rw latest-transmit-offset?  uint32
| | | |   +--rw jitter?                  uint32
| | | | +--rw user-to-network-requirements
| | | | +--rw num-seamless-trees?  uint8
| | | | +--rw max-latency?         uint32
| | | | +--rw interface-capabilities
| | | |   +--rw vlan-tag-capable?      boolean
| | | |   +--rw cb-stream-iden-type-list*  uint32
| | | |   +--rw cb-sequence-type-list*   uint32
| | +--rw listeners-list* [index]
| | | +--rw index                uint16
| | | +--rw end-station-interfaces* [mac-address interface-name]
| | | | +--rw mac-address        string
| | | | +--rw interface-name     string
| | | | +--rw user-to-network-requirements
| | | | +--rw num-seamless-trees?  uint8
| | | | +--rw max-latency?         uint32
| | | | +--rw interface-capabilities
| | | |   +--rw vlan-tag-capable?      boolean
| | | |   +--rw cb-stream-iden-type-list*  uint32
| | | |   +--rw cb-sequence-type-list*   uint32
| | +---x compute-request
| +--ro configuration
| | +--ro status-info
| | | +--ro talker-status?  enumeration
| | | +--ro listener-status? enumeration
| | | +--ro failure-code?   uint8
| | +--ro failed-interfaces* [mac-address interface-name]
| | | +--ro mac-address     string
| | | +--ro interface-name  string
| | +--ro talker
| | | +--ro accumulated-latency?  uint32
| | | +--ro interface-configuration
| | | | +--ro interface-list* [mac-address interface-name]
| | | | | +--ro mac-address      string
| | | | | +--ro interface-name   string
| | | | | +--ro config-list* [index]
| | | | | | +--ro index          uint8
| | | | | | +--ro (config-value)?
| | | | | | | +---:(ieee802-mac-addresses)
| | | | | | | | +--ro ieee802-mac-addresses
| | | | | | | | | +--ro destination-mac-address?  string
| | | | | | | | | +--ro source-mac-address?      string
| | | | | | +---:(ieee802-vlan-tag)
| | | | | | | +--ro ieee802-vlan-tag
| | | | | | | +--ro priority-code-point?  uint8
| | | | | | | +--ro vlan-id?              uint16

```

```

/ /      +---:(ipv4-tuple)
/ /      | +---ro ipv4-tuple
/ /      |   +---ro source-ip-address?    inet:ipv4-address
/ /      |   +---ro destination-ip-address? inet:ipv4-address
/ /      |   +---ro dscp?                  uint8
/ /      |   +---ro protocol?              uint16
/ /      |   +---ro source-port?           uint16
/ /      |   +---ro destination-port?      uint16
/ /      +---:(ipv6-tuple)
/ /      | +---ro ipv6-tuple
/ /      |   +---ro source-ip-address?    inet:ipv6-address
/ /      |   +---ro destination-ip-address? inet:ipv6-address
/ /      |   +---ro dscp?                  uint8
/ /      |   +---ro protocol?              uint16
/ /      |   +---ro source-port?           uint16
/ /      |   +---ro destination-port?      uint16
/ /      +---:(time-aware-offset)
/ /      |   +---ro time-aware-offset?     uint32
/ +---ro listener-list* [index]
/ |   +---ro index                        uint16
/ |   +---ro accumulated-latency?         uint32
/ |   +---ro interface-configuration
/ |   |   +---ro interface-list* [mac-address interface-name]
/ |   |   |   +---ro mac-address          string
/ |   |   |   +---ro interface-name       string
/ |   |   +---ro config-list* [index]
/ |   |   |   +---ro index                  uint8
/ |   |   |   +---ro (config-value)?
/ |   |   |   +---:(ieee802-mac-addresses)
/ |   |   |   |   +---ro ieee802-mac-addresses
/ |   |   |   |   |   +---ro destination-mac-address? string
/ |   |   |   |   |   +---ro source-mac-address?    string
/ |   |   |   +---:(ieee802-vlan-tag)
/ |   |   |   |   +---ro ieee802-vlan-tag
/ |   |   |   |   |   +---ro priority-code-point?   uint8
/ |   |   |   |   |   +---ro vlan-id?                uint16
/ |   |   +---:(ipv4-tuple)
/ |   |   |   +---ro ipv4-tuple
/ |   |   |   |   +---ro source-ip-address?    inet:ipv4-address
/ |   |   |   |   +---ro destination-ip-address? inet:ipv4-address
/ |   |   |   |   +---ro dscp?                  uint8
/ |   |   |   |   +---ro protocol?              uint16
/ |   |   |   |   +---ro source-port?           uint16
/ |   |   |   |   +---ro destination-port?      uint16
/ |   |   +---:(ipv6-tuple)
/ |   |   |   +---ro ipv6-tuple
/ |   |   |   |   +---ro source-ip-address?    inet:ipv6-address
/ |   |   |   |   +---ro destination-ip-address? inet:ipv6-address
/ |   |   |   |   +---ro dscp?                  uint8
/ |   |   |   |   +---ro protocol?              uint16

```

```
| | | +--ro source-port?      uint16
| | | +--ro destination-port?  uint16
| | | +---:(time-aware-offset)
| | | +--ro time-aware-offset?  uint32
| +---x deploy-configuration
| +---x undeploy-configuration
| +---x delete-configuration
+---x compute-all-configuration
+---x deploy-all-configuration
+---x undeploy-all-configuration
+---x delete-all-configuration"
```

ANNEX C. Project developer's guide

This annex provides details to developers and any reader that is interested on knowing details about the implementation. It covers the project structure and the explicit implementation, mentioning every function.

C.1 Project Structure

Currently, the project is held in a private repository found in [18]. In the following lines, the project structure is shown as a tree.

“

TSN-CNC-CUC-UPC

```

- CNC
  | - **Not on the scope of this project**
- CUC
  | - index.js
  | - logic
  |   | - core.js
  | - opc-ua-client
  |   | - opcua-client.js
  | - package-lock.json
  | - package.json
  | - resources
  |   | - ca.pem
  |   | - client-certificate.pem
  |   | - client-key.pem
  |   | - clientCertificate.pem
  | - restconf-client
  |   | - restconf-client.js
  | - utils
  |   | - arrayUtils.js
  |   | - gate-control-list
  |   |   | - gateControlListUtils.js
  |   | - yang
  |   |   | - ieee802-dot1q-tsn-types-upc-version@2018-02-15.js
  |   |   | - ieee802-dot1q-tsn-types.js
  |   |   | - json-samples
  |   |   |   | - cncResponse.json
  |   |   |   | - fieldsNotUsed.json
  |   |   |   | - sampleData.json
  |   |   |   | - talker.json
  |   |   | - yangUtils.js
- ENDPOINTS
  | - Listener
  |   | - config.json

```

```

| | | — index.js
| | | — opc-ua-server
| | |   | — opcua-server.js
| | | — package-lock.json
| | | — package.json
| — package-lock.json
| — Talker
|   | — config.json
|   | — index.js
|   | — opc-ua-server
|   |   | — opcua-server.js
|   | — package-lock.json
|   | — package.json
— README.md
— Yang_models
  | — example-jukebox@2016-08-15.yang
  | — ieee802-dot1q-tsn-types-upc-version@2018-02-
15(COMPLETE_VERSION).yang
  | — ieee802-dot1q-tsn-types-upc-version@2018-02-15.yang
  | — ieee802-dot1q-tsn-types.json
  | — ieee802-dot1q-tsn-types.yang
  | — ieee802-dot1q-tsn-types@2018-02-15.yang
  | — ietf-inet-types.yang
  | — ietf-yang-types.yang
  | — README.md
”

```

Note that there are two main folders that correspond to our projects: *CUC* and *ENDPOINTS*. The first one is a project itself, but the *ENDPOINTS* one contains two projects: *TALKER* and *LISTENER*.

C.2 CUC

The Centralized User Configuration project logic is based on the three following components: the OPC-UA Client, the Logic Unit Center and the RESTCONF Client.

1. CUC needs to know the OPC-UA Server addresses it will manage.
2. After polling their identity and configurations, the information is handled at the LUC.
3. It sends the generated information through the RESTCONF Client.
4. Finally, it generates the TSN Endpoint's configuration, sent through the OPC-UA Client. This process already described is shown in **Figure C.1**, which is the *index.js*, the entry file of the CUC Node project.

```

console.log("Polling Talker features and requirements...");
let talkerFeatures = await opcUaClient.connectOpcUaServer(config.endpointUrlTalker);
console.log("Talker information received, waiting for Listener...");
let listenerFeatures = await opcUaClient.connectOpcUaServer(config.endpointUrlListener);
console.log("Listener information received. Parsing now both obtained datagroups.");

let streamId = logicHandler.receiveDataFromOpcUaServer(talkerFeatures);
if(streamId == -1) console.log("Aborting. Obtained TSN configuration is not complete.");
streamId = logicHandler.receiveDataFromOpcUaServer(listenerFeatures);
if(streamId == -1) console.log("Aborting. Obtained TSN configuration is not complete.");

let response = logicHandler.checkStreamInformationReadyAndSend(streamId);

if(response) {
    logicHandler.generateGclAndSendConfig([SIMPLE_GCL]);
} else {
    "No response from CNC. Aborting."
}

```

Fig. C.1 Workflow of the CUC project.

After reviewing the general flow that the CUC is set into, we can check the details of every step.

C.2.1 OPC-UA Client

opcua-client.js file.

C.2.1.1 *connectOpcUaServer(endpointUrl)*

This function connects to the specified *endpointUrl* and, after creating a session against it, reads all known Address Space variables. After obtaining all of these variables, it places them in a single object instance.

*Note that if your own implementation has different node IDs, you would need to change them in this function.

C.2.1.1 *sendConfigToEndpoint(endpointUrl, config, isTalker)*

This method writes the computed TSN configuration to the specified *endpointUrl*. Since Talkers and Listeners need different configuration values, the boolean value *isTalker* is specified as a parameter. After writing all the necessary configuration to the Endpoints, a method is triggered on them to apply it. This triggering is the one that starts the performance of the TSN flow.

C.2.2 Logic Unit Center

core.js file.

C.2.2.1 *receiveDataFromOpcUaServer(receivedData)*

This function checks the presence of every needed field from a Talker or a Listener, reporting any possible error. If information is complete, all the data is pushed to dynamic arrays called *talkerInformation* and *listenerInformation*. The streamId obtained from the OPC-UA Server is returned.

C.2.2.2 *checkStreamInformationReadyAndSend(idStream)*

This function is called when there is a certainty that a stream is ready to be configured. The *idStream* is the identification used to locate the configurations that will be sent. If configurations for a Talker and a Listener are found, the *generateUniGroups* function is called to generate the instance of the YANG module (presented in the Annex B). Once the YANG instance is created a validated through the submodule *yangUtils.js*, it is sent through the RESTCONF client.

*Since the current status of the CNC implementation does not provide the interface yet, the requests and the configuration results are mocked. More information will be provided in the following RESTCONF Client section.

C.2.2.3 *generateUniGroups(ctrTalker, ctrListener)*

This method generates a JSON encoded variable that maps and instance of our designed YANG module.

C.2.2.4 *parseConfigurationData()*

Used in order to decode the response JSON containing the Status groups (see 3.1.1) from the CNC. It fetches the incoming configuration and adds it to the correct entry of the *talkerInformation* and *listenerInformation* variables and returns *true* if the operation is successful. However, if the incoming configuration data contains an error code, it returns *false*.

C.2.2.5 *generateGclAndSendConfig(gclType)*

This function computes the Gate Control List for a given Talker. It looks for all of its Streams and performs a time-scheduling design. Currently, it generates a simple TAS between a TSN traffic and best-effort classes.

After its computation, the GCL is returned.

C.2.3 RESTCONF Client

In the current implementation, the RESTCONF Client implements only one method to post a configuration to the CNC. RESTCONF requires mutual authentication, this means that our client needs a certificate that can be verified by the CNC, the server. The certificate used comes with the project, but needs to be changed when it is connected to a different CNC.

*Note that having a certificate trusted by the CNC is essential, since all HTTP/2 communications are forced to establish a secure session based on Mutual Authentication. If a certificate is not provided, the communication will never work.

C.2.3.1 *restconfRequest(body)*

This method posts the instance data generated by the LUC method *generateUniGroups*. It establishes a session with the CNC and checks that the response is successful. This function is currently commented, because despite of being able to send the configuration to the CNC successfully, there was no configuration returned. All kinds of request can be coded from this function changing the desired parameters.

Also, if another CNC or YANG module is used, remind to check the path used in the current implementation: */restconf/data/ieee802-dot1q-tsn-types-upc-version:tsn-uni*

C.3 Endpoints

Both Talker and Listener follow the same project structure, where the main file is the *opcua-server.js*. When it starts running, it initializes an OPC-UA server. During the initialization, all the Address Space variables are set, reading values from *config.json* when the method *post_initialize* is triggered.

After the initialization, the only interface these projects are providing are through the Address Space calls. By this. The CUC reads/writes data or triggers some function in the Endpoints. Since the Talker and Listener projects are not the same, it is better to check them separately to give better details.

C.3.1 Talker

Apart from the base OPC-UA sever, the Talker has a process that updates an Address Space node. This node data is the one that is sent through the Pub/Sub communication between Talker and Listener. It also needs to perform autoconfiguration thanks to the method triggering from the CUC.

C.3.1.1 *post_initialize()* Talker peculiarities

As mentioned, apart from having extra nodes indicating the traffic specification, it performs a periodical update in one of their variables. The following figure shows how this value *rawData* is being updated following an interval obtained from the *config.json*. It also registers the variable to the Address Space, to make it available for Subscriptions. Note that the updated variable is printed with a timestamp, for debugging purposes.

```
//Object to publish
//Declare InterfaceConfig for retrieved config
let rawData = new opcua.Variant({dataType: opcua.DataType.String, value:"INIT"});
let ctr = 0;
setInterval(function(){
    rawData = new opcua.Variant({dataType: opcua.DataType.String, value:ctr + generatePayload()});
    console.log("Publishing new variable change: "+ ctr + ", " + new Date().toISOString())
    ctr++;
}, config.interval*1000)
const publishObject = namespace.addObject({
    organizedBy: addressSpace.rootFolder.objects,
    browseName: "PublishObject"
});
namespace.addVariable({
    componentOf: publishObject,
    browseName: "publishObjectData",
    dataType: "String",
    value: {
        get: function () {
            return rawData;
        }
    }
});
```

Fig. C.2 Set up of the node that will be published.

Notice that the *rawData* variable is assigned with the expression *ctr + generatePayload()*.

C.3.1.2 *generatePayload()*

This function generates a fixed length payload by getting the *dataLength* from the *config.json* file. After generating the payload, it is returned as a String. This function is used to generate the fixed length payload.

C.3.1.3 *configureInterface(interfaceName, gclGates, gclGatesTimeDuration, interval, latency, vlanIdValue)*

This is the method triggered by the OPC-UA Client requests. It reads all the values previously set by the CUC, such as *interfaceName*, *gclGates*, *gclGatesTimeDuration*, *interval*, *latency* and *vlanIdValue*. With all this values, this function sets the configuration using *iptables* and *tc qdisc* tools. For more information regarding these tools, see Annex D.

This function is only valid for Ubuntu devices with a kernel version +5.0 (to have *taprio*, *etf* and *cbs* tools available) and the Intel i210 as the NIC. If you are using any other kind of operating system or distribution, it is important to check how the network management and traffic scheduling is performed, since this is specific for this combination of OS and NIC and would not work with different Endpoints.

Even though this configuration is configuring the OPC-UA Pub/Sub flow by default, it can be modified to consider any other traffic source by simple modifications on the scheduling commands performed.

C.3.2 Listener

The Listener also prepares all its variables for the CUC. Apart from that, it manages an OPC-UA client in the *opcua_client.js* that performs a subscription to the Talker's *rawData* node. Thanks to this client, OPC-UA Pub/Sub communication is possible between Talker and Listener.

C.3.2.1 *post_initialize()* Listener peculiarities

The following figure shows how the Listener sets a node to trigger an OPC-UA Subscription:

```
const launchConfig = namespace.addMethod(subscribeClientObject, {
  browseName: "InitSubscription"
});
launchConfig.bindMethod((inputArguments, context, callback) => {
  //Create opcUA client that requests subscription data and prints it
  subscriptionClient.connectOpcUaServer(config.talkerEndpointUrl, interval);
  callback(console.log("Done"));
});
```

Fig. C.3 Binding of the subscription method on the Listener's Address Space.

By this, when the CUC triggers it, the Listener subscribes to the Talker data, starting the TSN flow.

C.3.2.2 *connectOpcUaServer(endpointUrl, interval)*

It creates an OPC-UA subscription based on the interval specified as a parameter. On a successful subscribe, the data is printed with an application timestamp for debugging goals. This method can be commented if any different traffic source its used, such as *iperf*.

All the code provided in [18] is provided with in-code comments and logs for proper understanding.

ANNEX D. ENDPOINT CONFIGURATION EXAMPLE

This annex gives the most specific details regarding how an endpoint is configured by using the *iptables* and *tc qdisc* tools. It provides the necessary information in order to comprehend how the commands are built. It is intended to be consulted during the reading of the Chapters 4 and 5 (see Sections 4.3 and 6.4), since they contain the necessary information to understand the following content.

D.1 iptables

This tool [19] can be used as a Network Address Translator (NAT), a firewall and many other IP packet-related functionalities. Nevertheless, this project only uses a *socket priority* mapping.

As default, the Linux kernel assigns as '0' the priority to every socket that is being handled. As a consequence, flow treatment tools will not be able to distinguish between the different communications that are being served. This means that no scheduling would be possible. The objective of the use of *iptables* is to map the desired traffics into socket priorities. As a result, for example, the desired TSN flow will be mapped in a given socket priority, a different one than the best-effort traffic. To map a given traffic to a given socket priority, a command like the following has to be called:

```
"Sudo iptables -t mangle -A POSTROUTING -p udp --dport 7788 -j CLASSIFY -set-class 0:3"
```

1. The *-t mangle* flag means that the rule that is going to be applied will be placed in the mangle table. The mangle table is the one aimed to modify IP headers or other parameters of the socket.
2. *-A POSTROUTING* flag adds the following rule to the POSTROUTING stream. This means that the rule will be applied right after the routing of the packet has been performed.
3. *-p udp* and *-dport 7788* are the conditions to apply the rule. This specific one specifies that if the transport protocol is *UDP* and the port used is *7788*, the following rule will apply. A lot of different conditions may be requested, such as destination IP filtering, egress port filtering, among many others.
4. *-j CLASSIFY* flag is aimed to modify the socket priority of the traffic and it always requires the parameter *-set-class*, which in this case is *0:3*. This means that the priority of the selected sockets will be changed to 3.

It is important to remark that iptables follow a sequential assignment. This means that the first established rule that meets with the traffic's specifications (such as destination port) will be applied. Then, it is important to set first the most specific rules, followed by the best-effort assignments.

By several commands filtering and modifying the socket priorities on the system traffics, the *qdisc* tool will be able to distinguish between them on the scheduling, explained in continuation.

D.2 tc qdisc

This subsection of the Annex describes the different use of the *qdisc* tools named *taprio*, *etf*, and *cbs*. *Taprio* is used to perform traffic scheduling, *etf* for deterministic transmit time and *cbs* for traffic shaping. The use of these three commands is described in the following lines, based on sample configuration commands to describe how each parameter is set.

D.2.1 taprio

This tool [20], as mentioned, performs traffic scheduling based on the standard IEEE 802.1Qbv (see section 1.2.1). A common command that may give a new scheduling mechanism is:

```
"Sudo tc qdisc replace dev enp3s0 parent root handle 100 taprio
num_tc 3
map 2 2 1 0 2 2 2 2 2 2 2 2 2 2 2 2
queues 1 @0 1 @1 2 @2
base-time 1000000000
sched-entry S 01 3000000000
sched-entry S 02 7000000000
flags 0x1
txtime-delay 1000000
clockid CLOCK_TAI"
```

1. The first line initiates the command that adds (replaces the default one) a new qdisc to the interface *enp3s0*. This qdisc will be considered as the root discipline and it will be identified by the number *100*. It is stated that *taprio* is the tool that will be used to specify the qdisc performance.
2. *num_tc 3* refers to the number of traffic classes that are considered in the traffic scheduling. It is common to use as many as the amount of different transmit queues that a NIC has. In this example, considering we have an Intel i210 that has four transmit queues, the decision is to take three different traffic classes. This is because the third and fourth queue of the network interface are best-effort queues and there is no logical distinguishment between them.

3. *map 2 2 1 0 2 2 2 2 2 2 2 2 2 2 2 2* is a mapping from Linux socket priorities to the considered traffic classes. Thanks to the second line, the *taprio* mechanism is considering three traffic classes, 0, 1 and 2. This specific line maps the incoming socket priorities to the given number. Socket priorities can take values from 0 to 15. The actual mapping is the following:
 - a. Socket priorities 0 and 1 → Traffic class 2
 - b. Socket priority 2 → Traffic class 1
 - c. Socket priority 3 → Traffic class 0
 - d. Socket priorities from 4 to 15 → Traffic class 2
4. *queues 1@0 1@1 2@2* is the mapping of the traffic classes to the driver queues. For every traffic class a rule must be specified based on the *queueAmount@queueOffset*.
 - a. Traffic class 0 → sent to one queue, the first one (offset 0).
 - b. Traffic class 1 → sent to one queue, the second one (offset 1).
 - c. Traffic class 2 → best effort traffic, sent to two queues, the third and the fourth (amount 2, offset 2).
5. *sched-entry S 01 300000000* and *sched-entry S 02 700000000* are the definition of the traffic scheduler. Each line specifies an entry to the Gate Control List, S stands for setting the door state. 01 is a priority mask explained below and the 300000000 is the number of nanoseconds maintaining the state. To comprehend how the traffic classes are selected in each *sched-entry*, you should check how the byte is represented in binary to see which bytes are set to one. For example:

A byte set to 5 → 0 0 0 0 0 1 0 1 → Traffic classes 0 and 2

Then, all the time aware scheduler can be determined:

- a. For the first 300 milliseconds the only gate open is for the traffic class 0
- b. For the next 700 milliseconds, the only gate open is for the traffic class 1.

As a result, the interval of the scheduler can be determined by adding the duration of each state. In this case, the interval is 1 second.

6. The *flags 0x1* is a flag used to perform offload to the network interface. This means that part of this processing may be performed by the same hardware of the NIC, a feature compatible with Intel i210 that optimizes the performance.
7. *txtime-delay 1000000* is the approximate amount of time that the packets take from the queuing disciplines to the network interface's queues. Set by default to 1µs.
8. *clockID CLOCK_TAI* is to specify which is the clock that the scheduling will rely on. The *CLOCK_TAI* is equivalent to the system clock, so if it is synchronized via *gPTP* (see section 1.1.2 and 6.2), the scheduler will use the network time.

By analyzing this configuration, it is possible to realize that any kind of scheduling may be performed. If the previous command were used for testing purposes, note that there is no gate control list entry that specifies some space for the best-effort traffic (priority 2).

D.2.2 etf

As a quick reminder, this tool [21] configures the initial transmit time for a given traffic class, as follows:

```
"Sudo tc qdisc replace dev enp3s0 parent 100:1 etf
clockid CLOCK_TAI
delta 500000
offload"
```

1. The *dev* selected must be the same that the one selected for *taprio*.
2. *parent 100:1* specifies that the following qdisc will be applied to the qdisc 100:1, which is the traffic class 0 defined in *taprio* (this means that 100:2 is the traffic class 1 and 100:3 would be the traffic class 3).
3. *clockid CLOCK_TAI* is, again, the clock of reference for the following *delta* parameter.
4. *delta 500000* is the delay amount. This means that when a window is open for traffic class 0, it will hold 500 microseconds the start of the sending.
5. *offload* means that this will be performed by the same network interface.

D.2.3 cbs

This tool [22] is aimed to shape the output of a given traffic priority and limit its bandwidth. The command looks as below:

```
"Sudo tc qdisc replace dev eth0 parent 100:2 cbs  
idleslope 98688  
sendslope -901312  
hicredit 153  
locredit -1389  
offload 1"
```

1. As *etf*, it is placed under a given traffic class from *taprio*. In this example, the second traffic class is set to have a credit-based shaper in its output.
2. *idleslope* and *sendslope* are parameters than can be check previously in this document (see section 1.2.2).
3. *hicredit* and *locredit* is the maximum and minimum number of credits that the shaper can accumulate in excess or in debt.

By checking *taprio*, *cbs* and *etf*, all of them can be used according to the incoming TSN configuration, so that a TAS, a CBS and a controlled transmit time can be mapped to a working endpoint. With the help of *iptables* and the socket priorities, it is possible to distinguish all the traffic to be treated by these three tools.

ANNEX REFERENCES

- [1] Área de Ingeniería Telemática – Universidad de Navarra, “Spanning Tree Protocol” [Online]. Available: https://www.tlm.unavarra.es/~daniel/docencia/rba/rba11_12/slides/07-STP.pdf [Accessed 06 01 2021].
- [2] Cisco, “About Time-Sensitive Networking”, section “CNC to Bridge Control Plane” [Online]. Available: https://www.cisco.com/c/en/us/td/docs/switches/lan/cisco_ie4000/tsn/b_tsn_ios_support/b_tsn_ios_support_chapter_01.pdf [Accessed 06 01 2021].
- [3] Wikipedia, “IEEE 802.1aq” [Online]. Available: https://es.wikipedia.org/wiki/IEEE_802.1aq [Accessed 06 01 2021].
- [4] Wikipedia, “Link Layer Discovery Protocol” [Online]. Available: https://en.wikipedia.org/wiki/Link_Layer_Discovery_Protocol [Accessed 06 01 2021].
- [5] Frederik Hauser, Mark Schmidt, Marco Häberle and Michael Menth, “P4-MACsec: Dynamic Topology Monitoring and Data Layer Protection with MACsec in P4-Based SDN”, figure 3 [Online]. Available: https://www.researchgate.net/publication/340112340_P4-MACsec_Dynamic_Topology_Monitoring_and_Data_Layer_Protection_with_MACsec_in_P4-Based_SDN [Accessed 06 07 2021].
- [6] Leonardo Ochoa-Aday, Cristina Cervello-Pastor, ´ Member, IEEE, and Adriana Fernandez-Fernández, “Current Trends of Topology Discovery in OpenFlow-based Software Defined Networks” [Online]. Available: <https://upcommons.upc.edu/bitstream/handle/2117/77672/Current%20Trends%20of%20Discovery%20Topology%20in%20SDN.pdf> [Accessed 06 01 2021].
- [7] Levi Person, “Stream Reservation Protocol” [Online]. Available: https://avnu.org/wp-content/uploads/2014/05/AVnu_Stream-Reservation-Protocol-v1.pdf [Accessed 06 01 2021] [Published 11 03 2014].
- [8] IEEE Standards Association, “Bridges and Bridged Networks”, annex T [Online]. Available (private) https://standards.ieee.org/standard/802_1Q-2018.html [Accessed 06 01 2021] [Published 05 07 2018].
- [9] IEEE Standards Association, “Bridges and Bridged Networks”, section 8.6.5.1 [Online]. Available (private) https://standards.ieee.org/standard/802_1Q-2018.html [Accessed 06 01 2021] [Published 05 07 2018].
- [10] Erich Brockard, “Everything You Need To Know About TSN Sub-Standards & How To Combine Them” [Online]. Available:

- <https://blog.ebv.com/combining-tsn-sub-standards-knowhow/> [Accessed 06 01 2021] [Published 07 27 2016].
- [11] Josep Oriol Castaño Cid, “Proves amb equipament Time-Sensitive Networking (TSN)”, section 1.1.2.3, Bachelor Degree Thesis, UPC [Online]. Available: <https://upcommons.upc.edu/bitstream/handle/2117/121567/memoria.pdf> [Accessed 06 01 2021] [Published 07 06 2018].
- [12] Time Sensitive Networking Task Group, “P802.1Qcr – Bridges and Bridged Networks Amendment: Asynchronous Traffic Shaping” [Online]. Available: <https://1.ieee802.org/tsn/802-1qcr/> [Accessed 06 01 2021] [Last update 07 09 2020].
- [13] IEEE Standards Association, “Bridges and Bridged Networks”, Chapter 45 “Path Control and Reservation” [Online]. Available (private) https://standards.ieee.org/standard/802_1Q-2018.html [Accessed 06 01 2021] [Published 05 07 2018].
- [14] Josep Oriol Castaño Cid, “Proves amb equipament Time-Sensitive Networking (TSN)”, section 1.1.3.3, Bachelor Degree Thesis, UPC [Online]. Available: <https://upcommons.upc.edu/bitstream/handle/2117/121567/memoria.pdf> [Accessed 06 01 2021] [Published 07 06 2018].
- [15] IEEE Standards Association, “Frame Replication and Elimination for Redundancy” [Online]. Available (private): https://standards.ieee.org/standard/802_1CB-2017.html [Accessed 06 01 2021] [Published 09 28 2017].
- [16] IEEE Standards Association, “Time-Sensitive Networking for Fronthaul” [Online]. Available (private): https://standards.ieee.org/standard/802_1CM-2018.html [Accessed 06 01 2021] [Published 05 07 2018].
- [17] Time Sensitive Networking Task Group, “802.1AX-2020 – Link Aggregation” [Online]. Available: <https://1.ieee802.org/tsn/802-1ax-rev/> [Accessed 06 01 2021] [Last update 01 16 2019].
- [18] Gabriel David Orozco Urrutia and Jordi Cros Mompart, “TSN-CNC-CUC-UPC” [Online repository]. Available (private access): <https://github.com/gabriel-david-orozco/TSN-CNC-CUC-UPC>.
- [19] Herve Eychenne, “iptables (8) – Linux man page” [Online]. Available: <https://linux.die.net/man/8/iptables> [Accessed 06 01 2021].
- [20] Vinicius Costa Gomes, “tc-taprio (8) – Linux man page” [Online]. Available: <https://man7.org/linux/man-pages/man8/tc-taprio.8.html> [Accessed 06 01 2021] [Published 09 25 2018].

- [21] Jesus Sanchez-Palencia and Vinicius Costa Gomes, “tc-etf (8) – Linux man page” [Online]. Available: <https://man7.org/linux/man-pages/man8/tc-etf.8.html> [Accessed 06 01 2021] [Published 07 05 2018].
- [22] Vinicius Costa Gomes, “tc-cbs (8) – Linux man page” [Online]. Available: <https://www.man7.org/linux/man-pages/man8/CBS.8.html> [Accessed 06 01 2021] [Published 09 17 2017].