MASTER THESIS

TITLE: μTSN-CP: A Microservices-based Control Plane for Time Sensitive Networking

MASTERS DEGREE: Master's degree in Applied Telecommunications and Engineering Management (MASTEAM)

AUTHORS: Gabriel David Orozco Urrutia

ADVISORS: David Remondo
Anna Agusti Torra

DATE: July 7, 2023

Title: μTSN-CP: A Microservices-based Control Plane for Time Sensitive Networking (MASTEAM)
Authors: Gabriel David Orozco Urrutia

Advisors: David Remondo
Anna Agusti Torra

Date: July 7, 2023

Abstract

Time-Sensitive Networking (TSN) is a group of IEEE 802.1 standards that aim at providing deterministic communications over IEEE Ethernet. The main characteristics of TSN are low bounded latency and very high reliability, which complies with the strict requirements of industry and automotive applications. In this context, allocating time slots, configuration paths, and Gate Control Lists (GCLs) to contending TSN streams is often laborious. Software-Defined Networking (SDN) and the IEEE 802.1 Qcc standard provide the basis to design a TSN control plane to face these challenges. However, current SDN/TSN control plane solutions are monolithic applications designed to run on dedicated servers. None of them explores Microservice as a design pattern; these SDN controllers do not provide the required flexibility to escalate when facing increasing service requests. This work presents μTSN-CP, a microservices-based Control Plane (CP) architecture for TSN/SDN that provides superior scalability in situations with highly dynamic service demands. Using a qualitative approach, we evaluate our μTSN-CP solution compared to a monolithic solution in terms of CPU usage, RAM usage, latency, and percentage of successfully allocated TSN Streams. Our μTSN-CP architecture leverages the advantages of microservices, enabling the control plane to scale up or down in response to varying workloads dynamically. We achieve enhanced flexibility and resilience by breaking down the control plane into smaller, independent microservices. The experimental evaluation demonstrates that our TSN-CP outperforms the monolithic solution, with significantly lower CPU and RAM usage, reduced latency, and a higher percentage of successfully allocated TSN Streams. This advancement in TSN/SDN control plane design opens up new possibilities for highly scalable and adaptable networks, catering to the ever-increasing demands of time-sensitive applications in various industries.

To my family and loved ones
whom always find methods to show me
that my only limit is on my own will
posteris lvmen moritvrvs edat

CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

Time Sensitive Networking (TSN) refers to a set of IEEE 802.1 standards to provide deterministic, low-latency, and highly reliable communications over the existing Ethernet. TSN enables applications with different time-critical levels to share transmission resources in Ethernet while meeting their latency, bandwidth, and reliability requirements [1]. One of the main standards of TSN is the IEEE 802.1Qbv, which defines the Time-Aware Shaper (TAS). The TAS establishes Gate Control Lists (GCLs) for each outgoing port of the TSN switches to control which traffic classes can be transmitted at different time intervals. This feature ensures that traffic classes can access the transmission medium in a time-triggered manner, preventing non-critical traffic classes from invading the time slots assigned to time-critical traffic classes and thereby achieving bounded end-to-end latency for these [2]. It is essential to highlight that TAS requires precise time synchronization among all the nodes of a TSN domain (i.e., end stations and TSN switches); this synchronization is achieved with the Precision Time Protocol (PTP) by using the IEEE 802.1AS standard.

The assignment of time slots requires the TAS to know the network topology and the requirements of the different data streams, which end stations demand via the User/Network Interface (UNI). The IEEE 802.1Qcc standard defines three architecture models for getting the topology and user/network requirements and configuring the underlying network switches accordingly: the fully distributed model, the centralized network/distributed user model, and the fully centralized model. This work focuses on the fully centralized approach because it resembles an SDN architecture. It removes the control logic from the network devices and allocates it in Centralized Network Configuration (CNC) and Centralized User Configuration (CUC).

Even though the IEEE 802.1Qcc standard defines the basis for implementing the fully centralized model, it includes general guidelines and does not provide concrete specifications. In the literature, several works contain designs of a centralized control plane for TSN [1, 3, 4, 5]; however, these solutions follow non-scalable monolithic architectures, which are unable to allocate additional resources to specific tasks as needed. Such TSN control plane (CP) implementations waste significant computational resources and increase the time invested to calculate schedules (i.e., the length and position of time slots assigned to different GCLs at the outgoing ports of the TSN switches along network paths).

The concept of microservices is a promising approach to dealing with the scalability issues of monolithic TSN CPs. Microservices are an architectural model that divides a monolithic application into different components, each with a specific functionality [6]. Since these components are smaller than the whole monolithic application, it is easier to add or remove microservice instances [7], enabling resource allocation to highly demanded tasks more efficiently. In the literature, some works use microservices for SDN controllers: in [8], the authors create a cloud-native SDN controller for transport based on the concept of microservices; the work in [9] explores the use of microservices for a CP for open optical networks. However, to the best of our knowledge, there needs to be work exploring the use of microservices to design a TSN CP.

In this work, we propose μTSN-CP, a microservice-based SDN CP architecture for TSN that aims to reduce the time needed to retrieve network topology and stream requirements, calculate valid schedules and configure switches accordingly by optimizing the associated resource usage. Considering the CP's atomic functionalities, μTSN-CP decomposes the

CNC and CUC elements into microservices. Moreover, we implement a prototype of the μTSN-CP elements using Python and JavaScript and execute it on a hybrid cloud running on Amazon Web Services (AWS) using Docker and local instances. We analyze the performance of the μTSN-CP prototype and compare it with a TSN-CP built with a monolithic architecture. In such a comparison, we analyze the qualitative characteristics that improve the computational resource usage (i.e., CPU and RAM) and the time spent calculating new schedules when the TSN network topology changes or new service demands arrive from end stations.

The main contributions presented in this work are:

- A microservice-based TSN CP architecture design to transfer data with mixed time-criticality over Ethernet.
- A microservice-based TSN CP prototype that follows this design.
- The demonstration that our architecture provides increased scalability above a monolithic solution implementation.

The remainder of the document is structured as follows. Chapter 1 presents a conceptual review of the main topics of this document. Section 1.1. refers to the Microservices architecture; it includes a definition of its components and their functionalities, a comparison with a monolithic architecture, and its advantages in terms of resource usage. On the other hand, Section 1.2. explores Software Defined Networks, deepening on the main concepts and benefits compared to traditional networks. After that, in Section 1.3., we explore the theory behind TSN. Chapter 2 explores the literature and all the available CP solutions for TSN Networks. Chapter 3 explores all the pieces that made up the μTSN-CP; it presents a complete description of all the microservices inside the CP solution. In the later chapter 4, we include a Section to validate the solutions delivered by our μTSN-CP 4.1., another Section that describes the system we gave to the EETAC laboratory 4.2., a Section for exploring the different methods and variables that exist to improve the performance of the ILP are analyzed 4.2., and a later Section of the discussion 4.3.. Finally, Chapter 5 presents the work's conclusions.

CHAPTER 1. BACKGROUND

1.1. Microservice Architecture

Microservices is an architectural model for service deployment that aims to divide a monolithic application into multiple independent components, each with a specific function. [10]. The functional division enhances the creation and deletion of microservices and eases the application release process. On the other hand, it also allows allocating computational resources, such as RAM, CPU, Disk, or GPU, to specific microservices. In critical applications, this specific allocation of resources leads to a more efficient response to incoming traffic, as we can assign resources specifically to the applications that consume most of the computational resources.

For the proper functioning of microservices, it is necessary the presence of a specific group of elements that together shape the MSA (*MicroService Architecture*). Even though there is no such thing as a standard for microservices, the literature indicates the general characteristics that shape the MSA.

To understand the characteristics of MSA available in the literature, we classified them into two groups: the *Microservices Peculiarities* and the *MSA Elements*. The Microservices Peculiarities consist of the properties of microservices to follow for dividing a monolithic application into an MSA architecture. In this group, we have:

1. The functional decomposition taking as reference the business model [11]. This decomposition indicates that each microservice must provide something of value to the users or the other microservices while maintaining an adequate size in terms of functionalities. Microservices that bundle many functions return to a monolithic design, while very small microservices separate tightly linked functions, adding management complexity [12].

2. Each microservice is independent of the others; consequently, we need a common communication protocol between microservices such as HTTP [13] or, as in this work, using a message broker such as RabbitMQ.

3. Each microservice must consume from its own database for achieving data isolation [13, 14]. A centralized database implies losing the independence of microservices.

The second group of MSA features is the *MSA Elements*, which ensure the proper functioning of microservices providing security, reliability, and management. These items are:

- *Load Balancer* that distributes network traffic when a microservice is scaled across multiple instances to enhance the capabilities of the architecture [14].

- *API Gateway* is an interface responsible for preventing the establishment of direct communication between users and microservices, hiding the internal structure of the architecture [15].

- *Service Discovery* stores and provides the addresses of the microservices to establish effective communication between the elements of the architecture [16, 17, 18].

- *Circuit Breaker* is responsible for responding to failures when a microservice is not working properly [15, 17, 18].
- *Orchestrator* [13, 16, 17, 18] manages the resources allocated into microservices (*i.e.*, create, delete, replicate, and update microservice instances), this allows to increase and release resources.

1.2. Software Defined Networking

The Software Defined Networking (SDN) paradigm consists of the separation of the two planes of the networks, the data plane and the Control plane [19, 20]. The data plane is the part of the network that carries the user traffic directly. On the other hand, the Control plane is the network division that handles the packages forwarding from one part of the network to another. The decoupling between these two planes means that network administrators can use Software to schedule and control the entire network from a single control panel instead of on a device-by-device basis thus centralizing the control plane operations, which opens the possibility of optimizing the operation of the network (in terms of traffic, delay, energy consumption, etc), as one central entity has a global view of the state of the devices. As well as in Network Function Virtualization (NFV), in SDN the Software is decoupled from the Hardware [19].

A typical SDN architecture is divided into three layers:

1. Applications, responsible for communicating requests for resources or information about the network.
2. Drivers use information from applications to decide how to route a data packet.
3. Network devices receive information from the controller about where to move data.

The physical location of the previously mentioned elements depends on the network architecture characteristics [20].

There are different ways of implementing SDN. Some approaches use the OpenFlow protocol [21] proposed to standardize the communication between the SDN controller and the network devices in an SDN architecture. The literature suggests to enable researchers to test new ideas in a production environment. OpenFlow provides a specification to migrate the control logic from a switch into the controller. It also defines a protocol for communication between the controller and the switches.

On the other hand, there is also the option of using Yet Another Next Generation (YANG) data models and its related protocols network management protocols NETCONF and RESTCONF [22]. YANG is a data modeling language used to model network configuration, state data, and administrative actions; the first release was published on RFC 7950 in August 2016. YANG's main characteristics are its easy readability, hierarchical configuration data models, and reusable types and grouping [23]. It is linked to NETCONF and RESTCONF as both protocols were designed to provide a mechanism to install, manipulate, and delete configurations of network devices using YANG Models. However, they differ in their base protocol; NETCONF goes over SSH, and RESTCONF uses a RESTful API approach [24].

In μTSN-CP we decided to use the combination between YANG models, NETCONF, and RESTCONF as they are the best approach to match the requirements to configure a TSN network's parameters pragmatically. Moreover, there are already YANG models to define the configuration parameters of the TSN interfaces and the communication between the Centralized Network Configuration (CNC) and the Centralized User Configuration (CUC) . As we will see in other sections, NETCONF will be used to configure the MTSN switches from SoC-e [25] directly from an SDN controller, and RESTCONF will be used to send the configurations to the SDN controller through a RESTful API approach over HTTP and to communicate the CNC and the CUC in the UNI interface.

1.3. Time-Sensitive Networking

The literature defines Time-Sensitive Networking as a set of IEEE 802.1 standards that aim to provide deterministic, low-latency, and highly reliable communications over the existing Ethernet. TSN enables applications with different time-critical levels to share transmission resources in Ethernet while meeting their latency, bandwidth, and reliability requirements [1].

Several commercial devices that implement TSN's functionalities are available in the market. Table 1.3. shows an internet exploration performed during the project about the different devices in the market. The list includes the amendments of the 802.1 standards implemented in each device as well as the final functionality of the device (*i.e.,* if it is an end-station or a TSN switch). However, in the laboratory, we had the MTSN kit from the Spanish company SoC-e [26], with their headquarters in the Basque country. Such a device counts with the necessary models for implementing μTSN-CP and a Web interface for configuration tests and constant support of the manufacturer.

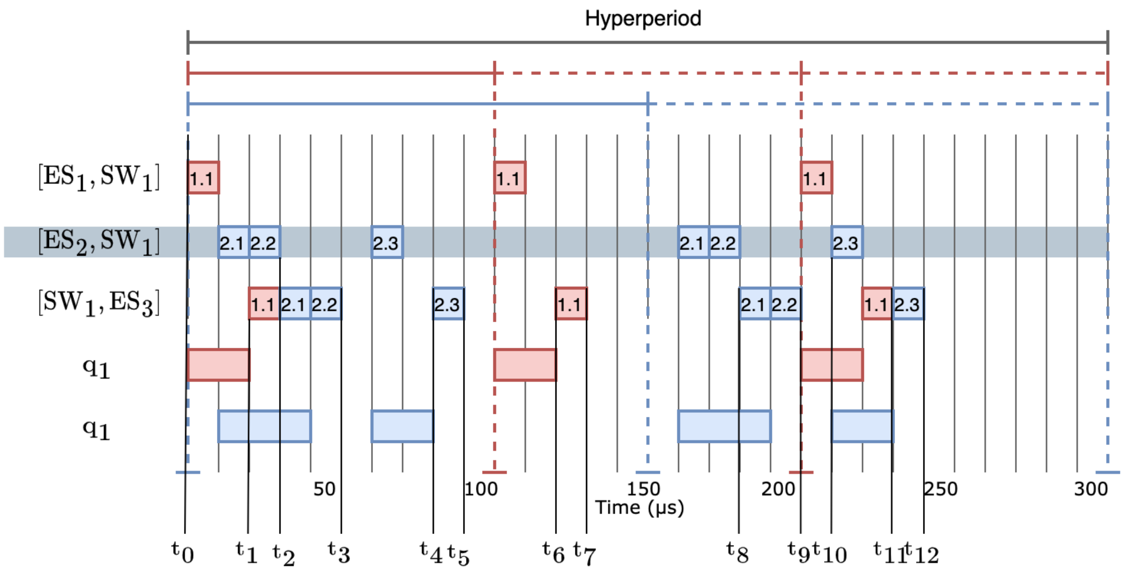


Figure 1.1: Time Aware Shaper Example

One of the main standards of TSN is the IEEE 802.1Qbv, which defines the Time-Aware Shaper (TAS), deeply explored in this work. The TAS establishes GCLs for each outgoing port of the TSN switches to control which traffic classes can be transmitted at different time intervals, being able to discriminate between one or another according to its queue priority. This feature ensures that traffic classes can access the transmission medium in a time-triggered manner, preventing non-critical traffic classes from invading the time slots assigned to time-critical traffic classes and thereby achieving bounded end-to-end latency for these [2]. It is essential to highlight that TAS requires precise time synchronization among all the nodes of a TSN domain (i.e., end-stations and TSN switches); using the IEEE 802.1AS standard, we achieve time synchronization with the Precision Time Protocol (PTP).

Figure 1.1 depicts the general functioning of the TAS for the network presented in Figure 1.2. As we can see, there is no superposition of the frames in the time domain when they are crossing throughout the same link. The TAS should be able to open and close the gates to enable the transmission of a type of traffic. The streams in TSN have several characteristics, such as the communication period, the maximum latency, and the origin and destination; all of these features are strict constraints to consider when generating a TAS-compliant schedule. For instance, even if two streams have different periods, they can never have a frame scheduled to transmit over the same link simultaneously.

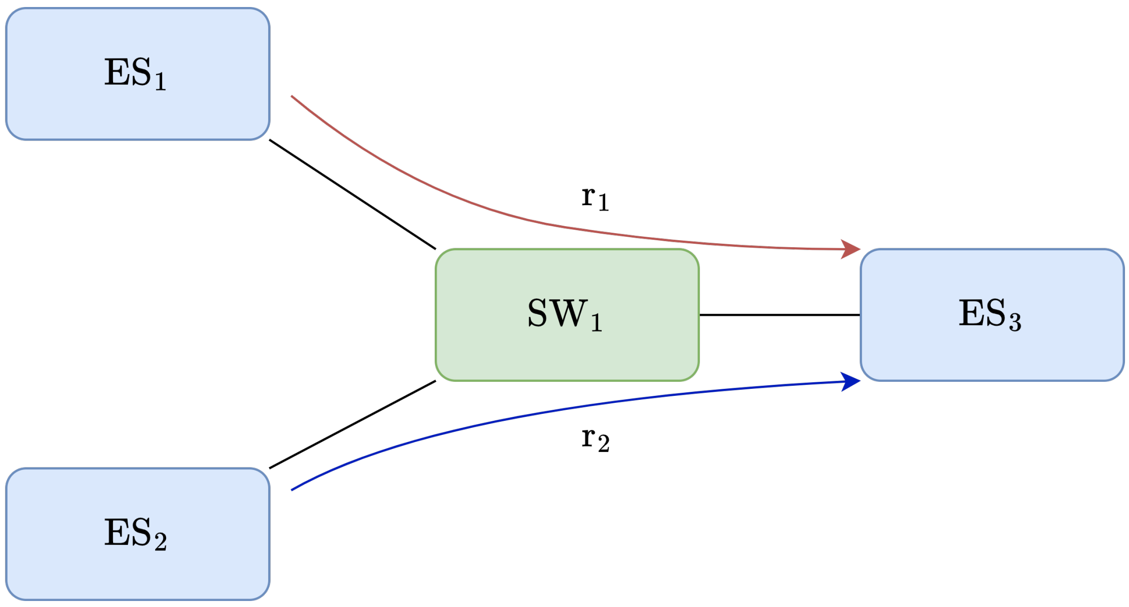


Figure 1.2: Example Network Diagram

Name of the Product	**Company**	**Features**	**Labor**
netX 90N	Hilscher	It has the possibility of synchronizing time and with an interrupt-based event trigger, which allows it to be used as a TSN end-station	end-Station
Trustnode	INNO ROUTE	It has suport of 802.1AS, 802.1Qbv, 802.1Qci and a NETCONF server	Switch TSN
SJA1105TEL	NXP	Support of 802.1AS	Switch TSN
Industrial Ethernet 4000	Cisco	It has 802.1AS, 802.1Qbv According to Datasheet	Switch TSN
TSN-G5008	Moxa	The manufacturer specifies that in the future this device will have TSN capabilities	Switch TSN
TSN machine switch	BR	All of the standad 802.1AS-2019, 802.1Q,802.1Qbv, 802.1Qav, 802.1Qcc,802.1Qbu, 802.1Qci,802.1CB	TSN
TSN Starter Package	TTTech	802.1AS, 802.1Qbv, 802.1Qbu, 802.1Qcc, 802.1CB	Switch TSN
PCIE-0400-TSN	Kotron	- 802.1 AS: timing and synchronization, 802.1 Qbv: traffic scheduling, 802.1 Qbu: frame preemption, 802.1 Qcc: Stream Reservation Protocol enhancements, - 802.1 CB	end-station
RELY-TSN-PCIe	Relyum	IEEE 802.1AS, IEEE 802.1Qbv, IEEE 802.1Qav, IEEE 802.1Qcc, IEEE 802.1CB, IEEE 802.1Qci, IEEE 802.1Qbu and IEEE 802.3br	end-station
MFN 100	TTTech	Not specified	Edge Computer

Table 1.1: Some of the TSN devices available in the literature market (as of June 2023)

CHAPTER 2. RELATED WORKS

Several works have addressed the design of a TSN CP based on the fully centralized model of the IEEE 802.1Qcc standard. The authors of [3] offer an SDN CP for FPGA TSN networks that includes a time-sensitive management protocol and a time-sensitive switching model. However, that contribution resorts to a protocol for the management of the underlying network that is different from the one described in the IEEE 802.1Qcc standard (which uses RESTCONF and NETCONF). Additionally, all the elements of the TSN CP are merged into a monolithic element, which limits the scalability and flexibility of the architecture. In [4], the authors leverage SDN and Object Linking and Embedding for Process Control Unified Architecture (OPC-UA) to build an SDN/TSN CP. In the implementation, they propose four elements (User Registration, Service Registration, Stream Management Component, and an OpenDaylight SDN controller), where each element is running on a different computer. Even though this solution shows a certain degree of platform independence and is not completely monolithic, the authors do not exploit all the potential advantages of a microservice architecture. Also, the Stream Management Component, in particular, performs several tasks that could be split further into separate microservices. On the other hand, the authors of [27] aims to enforce temporal isolation for the flows already foreseen at the network design time in order to avoid affectation to newly created flows. However, their solution does not show any intention of using MSA as a deployment strategy.

On the other hand, the concept of microservices is used in the literature for the design of SDN controllers. The authors of [8] propose μABNO (Application-based Network Operations); this SDN architecture is based on microservices and achieves auto-scalability while enabling cloud-scale traffic load management. They use Kubernetes to orchestrate the containers that execute the microservices and a cloud-native architecture running on the Adrenaline Test-bed platform. In [9], authors build an SDN controller for Open Optical Networks based on microservices. They rely on a microservices architecture with Docker containers and Kubernetes to enable platform-as-a-service network control, with the automated and on-demand deployment of SDN controllers or applications and on-the-fly upgrades or swaps of the Software components.

In summary, there is no work in the literature that explores the use of microservices in the design of a TSN CP: the only existing approaches present a monolithic design that fails to achieve good scalability and flexibility to accommodate varying traffic demands and wastes significant computational resources. In contrast, our μTSN-CP architecture addresses this issue by distributing the main CNC and CUC functionalities described in the IEEE 802.1 Qcc among microservices. This approach allows the system to dynamically adapt the resources needed for just the necessary functions upon changing traffic demands.

CHAPTER 3. ARCHITECTURE DESIGN

We can see the general structure of the implemented TSN SDN prototype in Figure 3.1. The CUC collects the characteristics of the data streams requested by the end-stations and passes on this information to the CNC. The IEEE 802.1Qcc standard defines the interface as User/Network Interface (UNI) for transferring information between the end-stations and the CUC. However, it does not specify rules or protocols to configure that interface. Each implementation can use the appropriate approach; many jobs in the literature widely use OPC-UA to complete communication due to the standardized usage in Industry 4.0 [28]. The information traversing the different layers includes the end-station identifiers, the message size, frequency, and the end-to-end latency requirements for each stream. Within the CNC, the TSN controller receives the information about the requested data streams from the CUC via the UNI employing RESTCONF. The scheduler determines a configuration of the switches that complies with the requirements of the data streams according to the network topology information received from the SDN controller through RESTCONF. The SDN controller distributes the configuration commands to the switches via NETCONF and YANG models. At the same time, it also transfers the topology information obtained from LLDP to the TSN controller via RESTCONF.

Regarding the microservice architecture, Figure 3.2 illustrates the microservices created for mapping the CP functions and the communication tools and interfaces between them. Jetconf gathers the requested data stream information from the CUC, receives input on the allocated streams from Southconf, and sends all of this to Preprocessing. This block combines the information with the topology information obtained from the Topology Discovery function and sends it to the scheduler (ILP calculator). The configuration resulting from the scheduler is sent to Southconf, which forwards it to the Virtual Local Area Network (VLAN) Configurator and the OpenDaylight SDN controller. The configuration of the TSN switches will be per the VLAN-related information received via HTTP and the schedule received via NETCONF.

3.1. Docker

To understand what Docker is, it is necessary to comprehend wider concepts namely, Operative System virtualization (OS Virtualization) and Hypervisor Virtualization. At a glance, virtualization is the process of isolating and abstracting resources of a computer system and generating concurrent subsystems to run parallel over the same Hardware. Each of these Virtual Machines (VMs) runs with a specific allocation of resources (i.e., CPU, memory, disk) from the host Hardware, isolated from the rest of the resources. Virtualization aims to employ multiple smaller servers on a single large server, reducing hosting costs and enhancing resource usage efficiency [29].

In Hypervisor Virtualization, the role of the hypervisor is crucial in managing and allocating Hardware resources to the VMs. The hypervisor, also known as the Virtual Machine Monitor (VMM), can operate directly on the Hardware, known as bare-metal or Type 1 hypervisor, or it can be installed on top of the host operating system (OS), referred to as Type 2 hypervisor. Regardless of the implementation, the hypervisor creates virtual environments where VMs can operate independently, completely isolated from one another

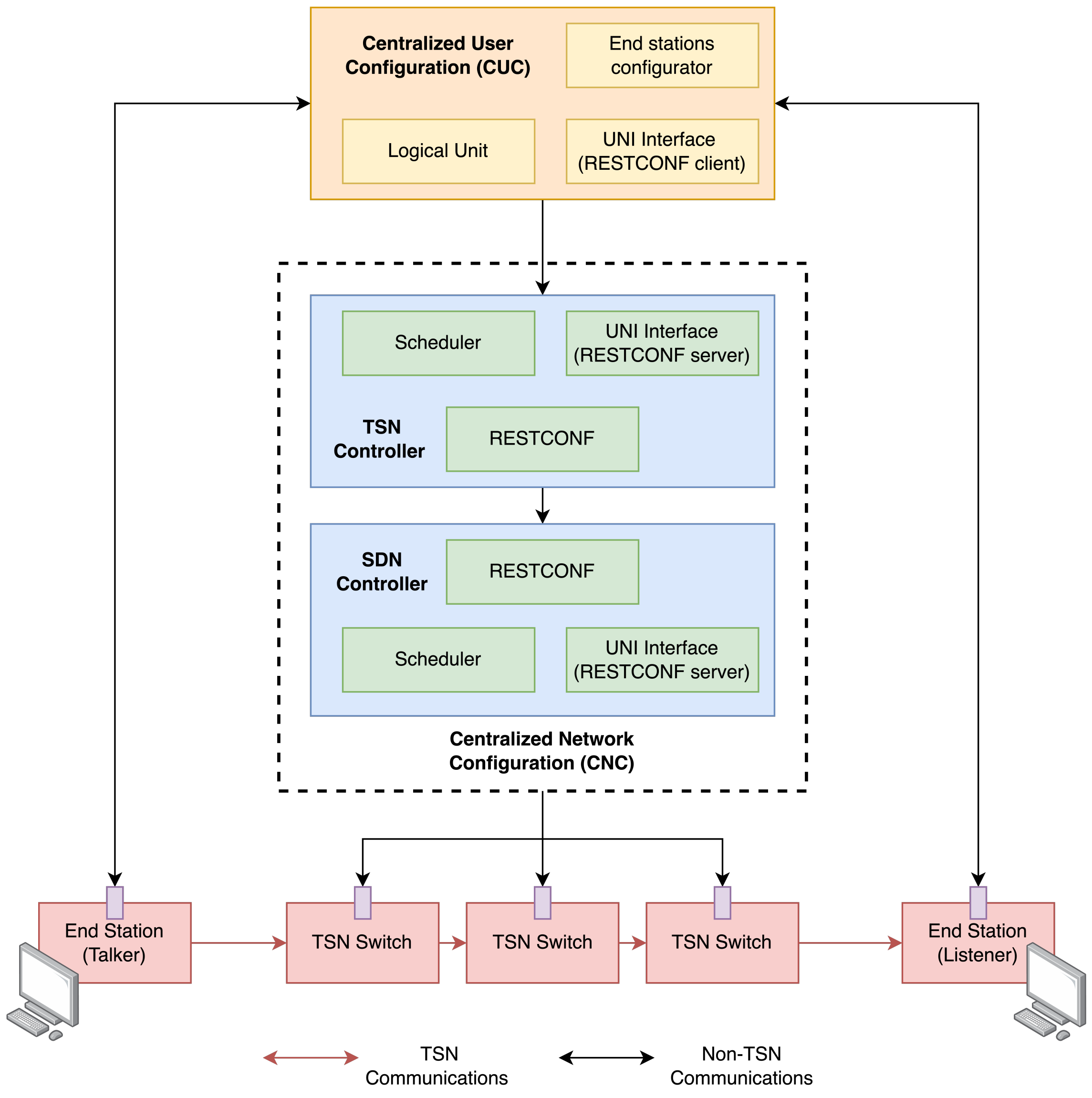

Centralized User Configuration (CUC)
End stations configurator
Logical Unit
UNI Interface (RESTCONF client)
Scheduler
UNI Interface (RESTCONF server)
TSN Controller
RESTCONF
SDN Controller
RESTCONF
Scheduler
UNI Interface (RESTCONF server)
Centralized Network Configuration (CNC)
End Station (Talker)
TSN Switch
TSN Switch
TSN Switch
End Station (Listener)
TSN Communications
Non-TSN Communications

Figure 3.1: Overview of the SDN-TSN prototype

and the host OS. This isolation extends to the operating system itself, allowing different OSes to run simultaneously on the same physical machine.

Virtual Box, VMware Workstation, and Microsoft Hyper-V are popular examples of hypervisor-based virtualization solutions. These platforms offer a wide range of features and capabilities, including the ability to manage and configure VMs, allocate Hardware resources, and even perform live migrations of VMs between physical hosts. Hypervisor virtualization is widely used in enterprise environments, data centers, and cloud computing platforms, providing flexibility, scalability, and efficient resource utilization.

On the other hand, OS virtualization, also known as containerization or operating system-level virtualization, takes a different approach. Instead of creating fully independent VMs, OS virtualization operates on top of the host OS, leveraging its kernel and sharing system resources. In this model, instances are called containers, and each container represents a lightweight and isolated execution environment. Containers share the same OS kernel, libraries, and binaries, which results in reduced overhead and improved performance

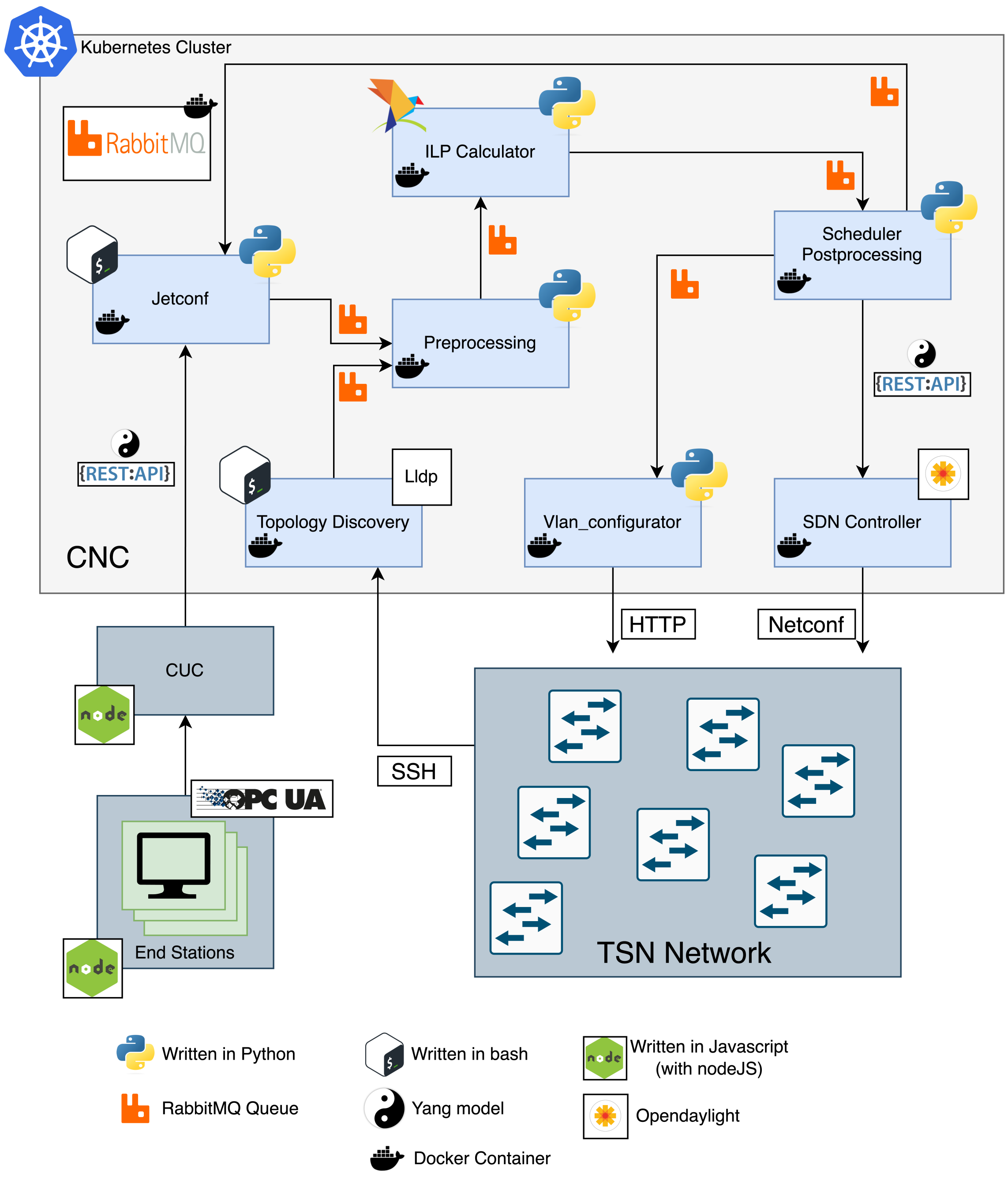


Figure 3.2: Microservices architecture of the CP functions.

compared to hypervisor virtualization.

One of the most prominent platforms for OS virtualization is Docker, which has gained significant popularity in the IT industry. Docker simplifies the packaging and deployment of applications by encapsulating them within containers, along with their dependencies and configuration. Docker containers provide a consistent and reproducible environment, enabling developers to build, ship, and run applications seamlessly across different computing environments. In addition to Docker, other alternatives like Linux containers (LXC) and *Containerd* offer similar containerization capabilities, and several important projects have started to migrate to use *Containerd*, such as Kubernetes.

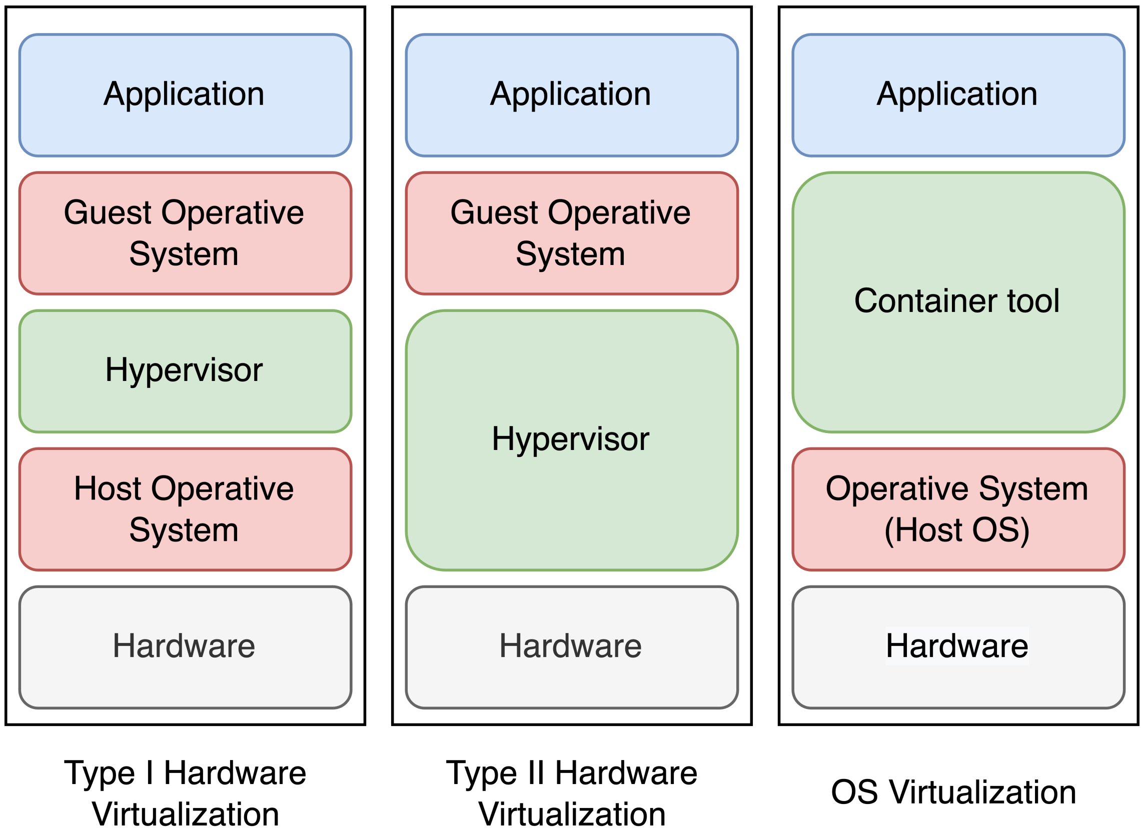


Figure 3.3: Hardware vs OS Virtualizations

The reason behind using Docker in a microservices architecture is to simplify the delivery and management of microservices. Containers are lighter than VMs machines since they already run over the kernel, and one of the design patterns is to put only the necessary Software inside the container. The fact that containers are lighter means that they are created faster and easier than VMs, improving the horizontal scalability of the system (i.e., the possibility of adding extra replicas of an instance in the system) and the failure recovery times.

In μTSN-CP, all of the microservices are running over Docker containers; we explore two different approaches, one with Docker compose and another over Kubernetes, a portable, extensible, open-source platform for managing containerized workloads and services. The first exploration is a prototype, and the second is a theoretical probe-of-concept considering the benefits of adding the extra layer Kubernetes offers. The usage of Docker allowed us to use versions and distributions of Linux to execute each microservices and isolate them in both the development process and the operations maintenance.

3.2. Message Broker

To communicate microservices between themselves, we decided to use a message broker and inter-application communication technology that helps to build a common integration mechanism to exchange information undependable on the language they were written or the platform they ran over. In μTSN-CP, we decided to use RabbitMQ as a message broker because it provides a message queuing service that allows asynchronous processing of messages. When using message queuing, a part of the application pushes notifications to a queue in the message broker, and then the consumers of those messages pull them; when the processing of the message is asynchronous means that messages can stay in the queue without being immediately processed.

Each microservices of μTSN-CP pulls/pushes messages to the appropriated queue. As the application skeleton is written in Python, we used Pika [30], a pure-Python implementation of the AMQP 0-9-1 protocol for message queuing. The Software module to push/pull is the same and is included as a library in each microservices that needs it. Besides, RabbitMQ also runs as a microservice in the cluster, enabling the possibility of scaling it horizontally.

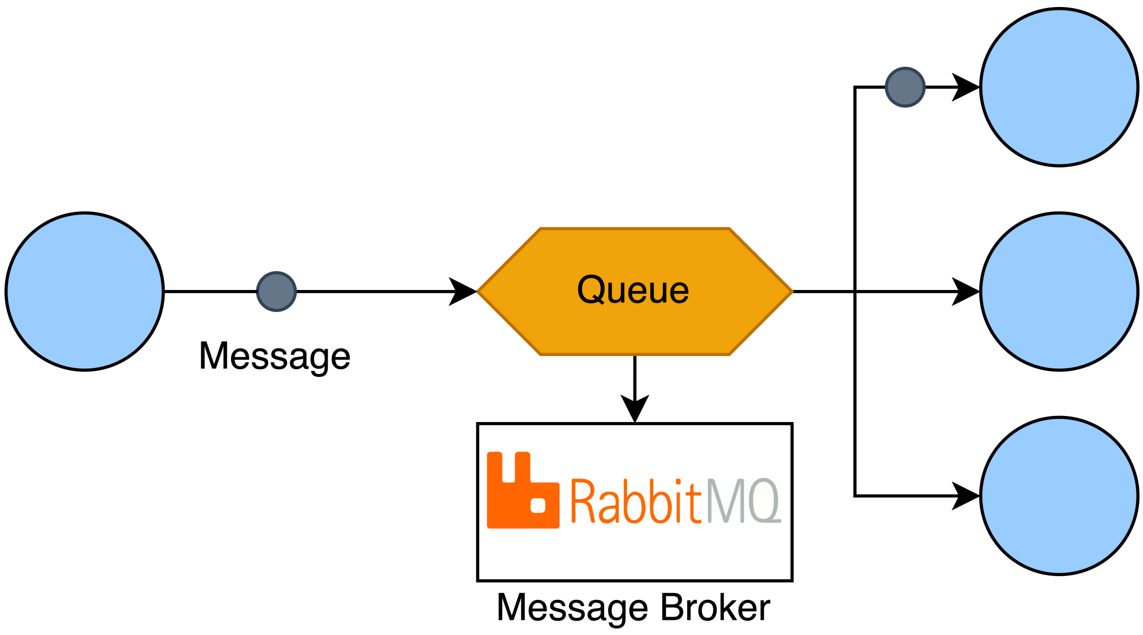


Figure 3.4: Message Queue with RabbitMQ as broker

3.3. Jetconf Microservice

Jetconf microservice is the point of contact between the CUC and the CNC implementing the UNI interface with the YANG model defined in the IEEE standard 802.1Qcc [31]. This microservice will handle the communication of the user requirements to the CNC internal microservices. As input, this microservice gets the JSON payload using a REST/API using RESTCONF, and such payload must match the definitions of the parameters in the YANG module. Some parameters included in the payload are the number of streams and the communication period, maximum latency, size, and smallest and highest transmission offsets. With such requirements, we decided to use Jetconf [32], an implementation of the RESTCONF protocol written in Python3 as a base for the microservice.

In the output, this microservice should communicate the answer to the request payload to the CUC; such output needs to include the feasibility of the communication as a binary value and the offset that each talker should follow to achieve the communication. Additionally, it should send the requested information in a JSON payload using the appropriated

RabbitMQ queue to the Pre-processing microservice. Regarding security, Jetconf includes HTTP/2 over TLS certificated-based authentication to the clients; such certificates should be shared with the client (CUC) to grant the communication appropriately. Figure 3.5 depicts the general operation of Jetconf microservice.

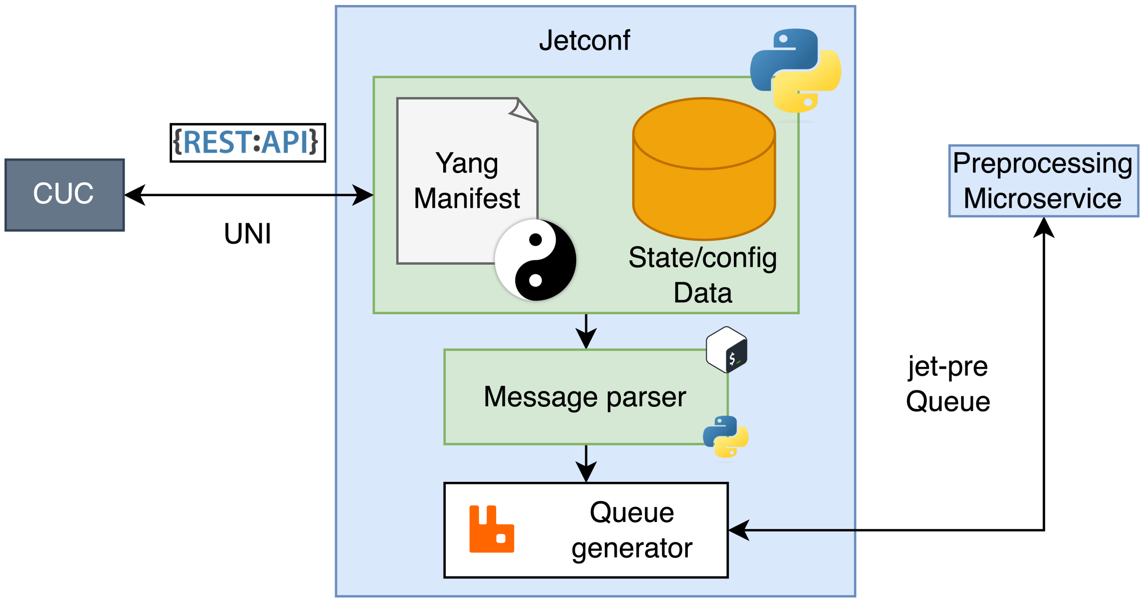


Figure 3.5: Overview of Jetconf Microservice

3.4. Topology Discovery Microservice

The Topology Discovery microservice is in charge of getting the topology of the TSN network and providing it to the other microservices as a network Matrix. For this task, this microservice uses the LLDP [33], a vendor-neutral protocol of the data-link layer of the Open Systems Interconnection (OSI) used for advertising the identity, capabilities, and neighbors. In a nutshell, the Topology Discovery microservice will generate a list of all the available devices in the network by accessing them through ssh to all the available devices in the network and retrieving their LLDP information. We combined bash scripts and Python3 to program this microservice, specifically using the Paramiko [34] library, which implements the SSHv2 protocol. Image 3.6 shows the general architecture of this microservice and its parts.

This is the general performance of the microservice in terms of the steps followed from the moment we start the communication to the moment the information is retrieved from the devices and parsed to the Preprocessing Microservice:

1. Signal to activate the microservice from Preprocessing microservice to the internal topology generation module.
2. The topology generator module activates the retrieval of the topology information using the Paramiko module. It uses a Paramiko to send the commands of the following code sniped to every device on the network to get the information of neighbors

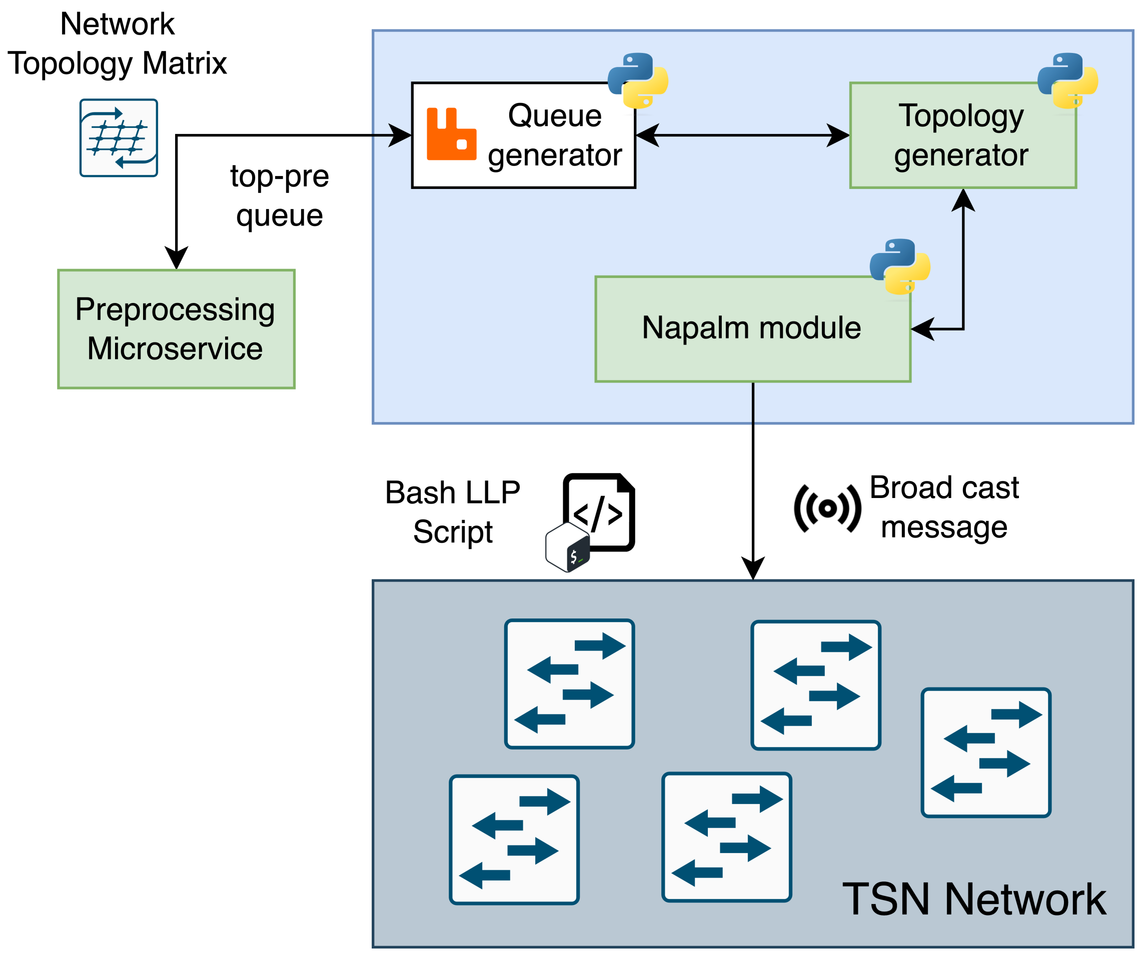


Figure 3.6: Overview of Topology Discovery Microservice

of each device in the network. Raw data is collected and stored as files in the microservice.

3. The topology generator module gets all the files, cleans the information and translates it into a Network Topology Matrix.

4. Once the network information is ready, Topology Discovery Microservice generates a JSON message in the RabbitMQ queue and sends it to the Preprocessing Microservice.

On the other hand, this is the code implemented in the microservice to retrieve the topology of the network:

```
for ip in addresses:
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    ssh.connect(ip, username='soc-e', password='soc-e')
    ssh_stdin, ssh_stdout, ssh_stderr = \
    ssh.exec_command('sudo lldpcli show neighbors')
    time.sleep(1)
    data = ssh_stdout.readlines()
    with open('devices/topology_'+ ip + '.txt', 'a') as f:
        for line in data:
            f.write(str(line) + '\n')
```

3.5. Preprocessing Microservice

This microservice is the entry point between the data the Jetconf Microservice provides and the Topology Discovery Microservice provides. In Figure 3.7, we can see the input values inside this Microservice from the previous Microservices. Those values are retrieved from the respective RabbitMQ queues. Two modules are doing the functionalities of the microservice. The first module is the Dijkstra Module, which implements the Dijkstra algorithm to get the shortest path possible from origin to destination for the routes in the schedule. It is important to highlight that in our μTSN-CP, the module for calculating the path from sources to destinations and the module for calculating the schedule are in different microservices, as seen in other sections. The reason behind this design criteria is to reduce the number of variables involved in the mathematical formulation of the ILP problem for the schedule and reduce the complexity and the computation time to find a solution. Furthermore, this design pattern enables us to easily adapt the microservice to use different algorithms to calculate the path for the streams (e.g., Floyd-Warshall and Johnson's algorithms).

The other primal functionality of Preprocessing Microservice is to prepare the input parameter for the principal Microservice in the architecture, the ILP calculator. This preprocessing job is necessary to reduce the number of tasks assigned to the ILP calculator. As Figure 3.7 depicts, the output parameters for the microservice correspond to parameterized resultant values from the Dijkstra module and the previous microservices. Besides, maintaining specific functionalities in that microservice enhances the possibility of using different approaches implemented as a microservice instead of the ILP calculator, simply following the black box design criteria and respecting the input and outputs. Finally, as in the previous microservices, the way to communicate the results to the ILP Microservice is by a RabbitMQ queue.

3.6. ILP Calculator Microservice

The ILP Calculator Microservice takes its name from the usage of Integer Linear Programming (ILP) , a mathematical optimization and feasibility programming method in which a set of constraints (represented as linear inequalities) and a linear optimization function describe a problem. The variables used in the definition of the linear constraints create a space of solution that is filled with the set of possible solutions that respect the constraints inequalities. The larger the number of variables and constraints, the more complex is the solution space and the harder to get a solution to the scheduling problem. In our μTSN-CP, all of the characteristics of TSN TAS are included in the mathematical model. We adopted the ILP Model described in [35] and implemented it with Python 3 using Pyomo, an open-source optimization modeling language. In this section, we deepen the characteristics of the TAS as we explore the ILP formulation problem.

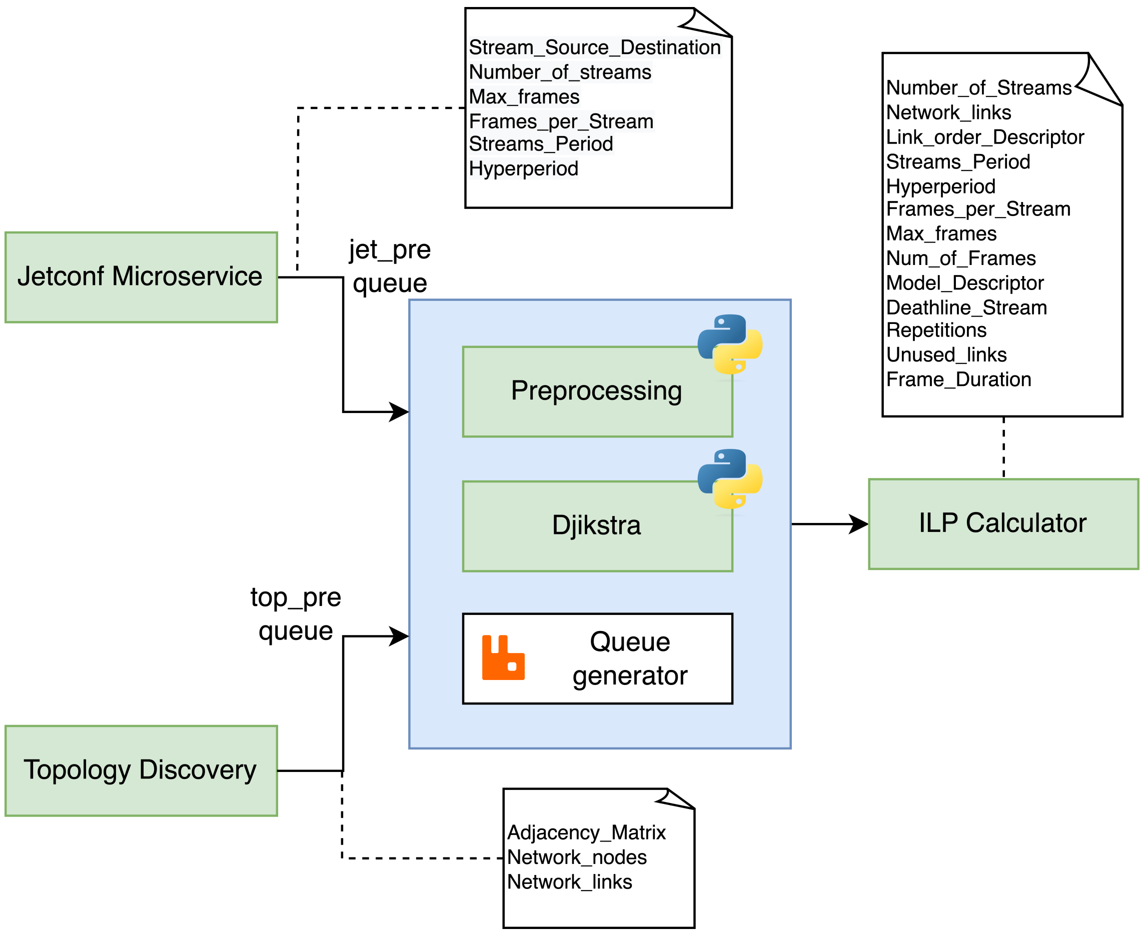


Figure 3.7: Overview of Preprocessing Microservice

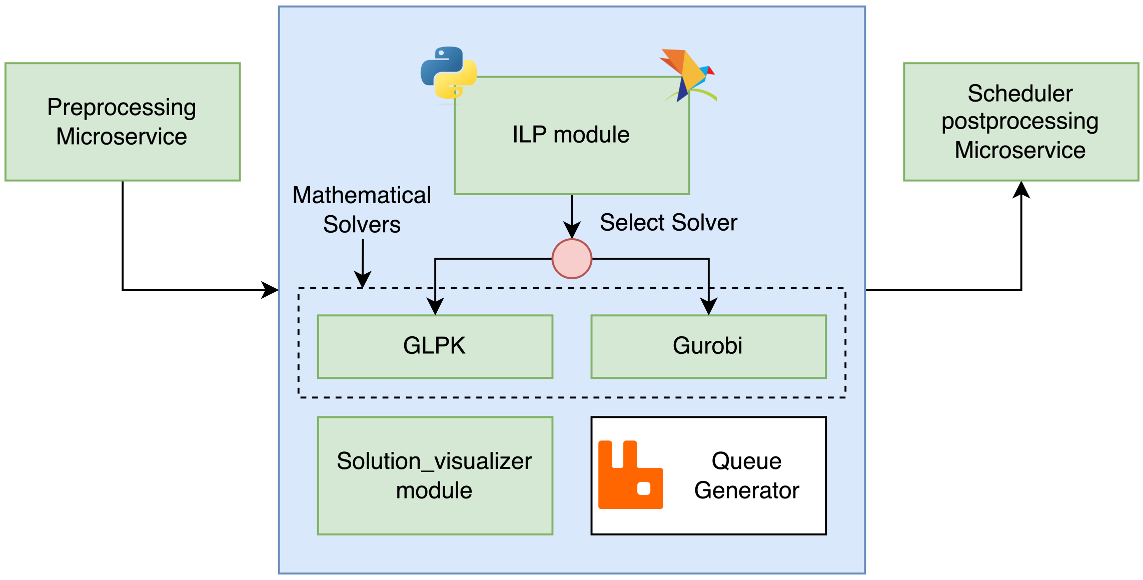

Preprocessing Microservice
ILP module
Mathematical Solvers
Select Solver
GLPK
Gurobi
Solution_visualizer module
Queue Generator
Scheduler postprocessing Microservice

Figure 3.8: Overview of ILP Calculator Microservice

3.6.1. ILP Model

The first point to regard is the necessity of reducing the values for the total number of excess queues denoted from now as K, and the total number of extra end-to-end latency occasioned by interfering streams indicated as Λ. The article defines excess queues as the number of times a stream has to be divided into separate sub-frames in the time domain. K is mathematically defined as the summation of all the excess queues $(\mathrm{K}_{a,b}-1)$ in all of the used TSN ports of the devices in the network $[\mathrm{V}_a,\mathrm{V}_b] \in \mathbf{E}$. On the other hand, Λ is defined as the summation of all of the subtractions of the end-to-end latency of each stream (λ_i) and the latency of each stream as if any other stream wouldn't have interrupted the transmission ($\underline{\lambda_i}$) for all of the streams in the scheduling problem ($\mathrm{s}_i \in \mathbf{S}$). Consequently, the optimization function in 3.1. The weight of the constants $C1$ and $C2$ will determine the prioritization of reducing one parameter over the other.

$$C1 \sum_{[\mathrm{V}_a,\mathrm{V}_b]\in\mathbf{E}} (\mathrm{K}_{a,b}-1) + C2 \sum_{\mathrm{s}_i\in\mathbf{S}} (\lambda_i - \underline{\lambda_i}) \tag{3.1}$$

Constraint 3.2 represents that the number of queues used for all of the streams in a certain link is lower-bounded by the number of queues used for each stream in that link $\mathrm{s}_i^{[\mathrm{v}_a,\mathrm{v}_b]}.\rho$. Equation 3.3 defines λ_i to be the total time used for transmitting the stream s_i from the moment the first frame is transmitted in the first link of the path until the last frame is fully received at its destination.

$$s.j. \qquad \mathrm{k}_{a,b} \geq \mathrm{s}_i^{[\mathrm{v}_a,\mathrm{v}_b]}.\rho \qquad \forall \mathrm{s}_i^{[\mathrm{v}_a,\mathrm{v}_b]} \in \tag{3.2}$$

$$\lambda_i = \mathrm{f}_{i,k}^{\mathrm{s}_i.t}.\phi + \mathrm{f}_{i,k}^{\mathrm{s}_i.t}.L - \mathrm{f}_{i,1}^{\mathrm{s}_i.s}.\phi \qquad \forall \mathrm{s}_i \in S \tag{3.3}$$

Constraint 3.4 ensures that the latency of each stream λ_i is upper bounded by its deadline $\mathrm{s}_i.D$. Consequently, in the results provided by the model, all of the streams will arrive within their deadline. Equation 3.5 ensures that every stream has to be fully scheduled within its period $\mathrm{s}_i.T$. Equation 3.6 is to guarantee that every frame traversing a link $[\mathrm{v}_a,\mathrm{v}_b]$ in its path to the destination has to start transmitting after the same frame has been fully received on the previous link of the path $[\mathrm{v}_x,\mathrm{v}_a]$, even if the worst synchronization error δ is present.

$$\lambda_i \leq \mathrm{s}_i.D \qquad \forall \mathrm{s}_i \in S \tag{3.4}$$

$$\mathrm{f}_{i,m}^{[\mathrm{v}_a,\mathrm{v}_b]}.\phi \leq \mathrm{s}_i.T - \mathrm{f}_{i,m}^{[\mathrm{v}_a,\mathrm{v}_b]}.L \qquad \forall \mathrm{f}_{i,m}^{[\mathrm{v}_a,\mathrm{v}_b]} \in F \tag{3.5}$$

$$\mathrm{f}_{i,m}^{[\mathrm{v}_a,\mathrm{v}_b]}.\phi \geq \mathrm{f}_{i,m}^{[\mathrm{v}_x,\mathrm{v}_a]}.\phi + \mathrm{f}_{i,m}^{[\mathrm{v}_x,\mathrm{v}_a]}.L + \delta \qquad \forall \mathrm{f}_{i,m}^{[\mathrm{v}_a,\mathrm{v}_b]}, \mathrm{f}_{i,m}^{[\mathrm{v}_x,\mathrm{v}_a]} \in F \tag{3.6}$$

Equation number 3.7 enforces that each link can transmit only one frame at a time and that the frames in a stream must be transmitted in order. In constraint 3.8 is represented as the general parameter of the hyper period (i.e., defined as the Least Common Multiple of all of the periods of the streams in the schedule) $\mathrm{hp}_{i,j} = lcm(\mathrm{s}_i.T, \mathrm{s}_j.T)$. The set of translations of stream s_i and stream s_j are denoted as A and B. All possible values in A and B are denoted as α and a β, respectively. However, it is essential to highlight that in the real implementation in Python, each stream in the schedule will have its own set of variables representing the repetitions within the hyper period, so each stream will have its own sets of variables for representing the repetitions within its hyper period.

$$f_{i,m}^{[v_a,v_b]}.\phi + f_{i,m}^{[v_a,v_b]}.L \leq f_{j,n}^{[v_a,v_b]}.\phi \qquad \forall f_{i,m}^{[v_a,v_b]}, f_{i,n}^{[v_a,v_b]} \in F^2, m < n \tag{3.7}$$

$$A = \left\{0, 1, ..., \frac{hp_{i,j}}{s_i.T} - 1\right\}, \quad B = \left\{0, 1, ..., \frac{hp_{i,j}}{s_j.T} - 1\right\} \tag{3.8}$$

As it is possible to have overlapping in time and link within that repetitions, constraints 3.9 and 3.10 avoid this from happening by stating that $f_{i,m}^{[v_a,v_b]}$ must finish before $f_{j,n}^{[v_a,v_b]}$ starts, for all of the possible values of the repetitions of both streams in the hyper period (A and B). Equations 3.9 and 3.10 represent opposite cases; if one of the equations is fulfilled, the other should not because it indicates that one Frame starts before the other, and that is why for those scenarios, there is a variable $\sigma \in 0, 1$ that represents this model disjunction. M is a large value used for prioritizing the variable in the inequality. In the Python implementation, σ is a N dimensional array, each dimension being the set of repetitions of each stream.

$$\alpha.s_i.T + f_{i,m}^{[v_a,v_b]}.\phi + f_{i,m}^{[v_a,v_b]}.L \leq \beta.s_j.T + f_{j,n}^{[v_a,v_b]}.\phi + M.\sigma \tag{3.9}$$

$$\beta.s_i.T + f_{j,n}^{[v_a,v_b]}.\phi + f_{j,n}^{[v_a,v_b]}.L \leq \alpha.s_i.T + f_{i,m}^{[v_a,v_b]}.\phi + M.(1 - \sigma) \tag{3.10}$$

$$\forall f_{i,m}^{[v_a,v_b]}, f_{j,n}^{[v_a,v_b]} \in F^2, \forall \alpha \in A, \forall \beta \in B \tag{3.11}$$

We achieve TAS determinism by allowing only one stream to be present in a queue at any moment. None of the frames of any stream can enter into a queue while another frame is being queued. Constraints 3.12 and 3.13 achieve this characteristic in cases where two streams share the same egress port in a switch. ω is an auxiliary variable representing the disjunction between 3.12 and 3.13. Using this variable, we guarantee that only one of the two constraints must be fulfilled for each pair of streams with the same link.

$$\alpha.s_i.T + f_{i,m}^{[v_a,v_b]}.\phi \leq \beta.s_j.T + f_{j,n}^{[v_a,v_b]}.\phi + M(\omega + \varepsilon_{i,j}^{[v_a,v_b]} + \varepsilon_{j,i}^{[v_a,v_b]}) \tag{3.12}$$

$$\beta.s_j.T + f_{j,n}^{[v_a,v_b]}.\phi \leq \alpha.s_i.T + f_{i,m}^{[v_x,v_a]}.\phi + M(1 - \omega + \varepsilon_{i,j}^{[v_a,v_b]} + \varepsilon_{j,i}^{[v_a,v_b]}) \tag{3.13}$$

$$\forall f_{i,m}^{[v_x,v_a]}, f_{i,m}^{[v_a,v_b]}, f_{j,n}^{[v_x,v_a]}, f_{j,n}^{[v_a,v_b]} \in F^4, \forall \alpha \in A, \forall \beta \in B \tag{3.14}$$

The variable $\varepsilon_{i,j}^{[v_a,v_b]}$ represented in 3.15 captures whether or not two streams are assigned in the same queue. The variable will be 1 only if $s_i^{[v_a,v_b]}$ is assigned to a queue strictly less than $s_j^{[v_a,v_b]}$. Such of behaviour for $\varepsilon_{i,j}^{[v_a,v_b]}$ is depicted in equations 3.16 and 3.17.

$$\varepsilon_{i,j}^{[v_a,v_b]} = \begin{cases} 1, & s_i^{[v_a,v_b]}.\rho < s_j^{[v_a,v_b]}.\rho \\ 0, & \text{otherwise} \end{cases} \tag{3.15}$$

$$s_j^{[v_a,v_b]}.\rho - s_i^{[v_a,v_b]}.\rho - M(\varepsilon_{i,j}^{[v_a,v_b]} - 1) \geq 1 \tag{3.16}$$

$$s_j^{[v_a,v_b]}.\rho - s_i^{[v_a,v_b]}.\rho - M.\varepsilon_{i,j}^{[v_a,v_b]} \leq 0 \tag{3.17}$$

$$\forall s_i^{[v_a,v_b]}, s_j^{[v_a,v_b]} \in S^2 \tag{3.18}$$

Finally, the equations 3.19 and 3.20 are for taking into account the cases in which two streams s_i, s_j that uses a same egress port of a TSN switch arrived at the different ingress port. In that case, there can be a synchronization error between the two internal clocks of both ports, disrupting the connection. Consequently, the maximum synchronization error δ is included in the equations to prevent this error from happening.

$$\alpha.s_i.T + f_{i,m}^{[v_a,v_b]}.\phi + \delta \leq \beta.s_j.T + f_{j,n}^{[v_y,v_a]}.\phi + M(\omega + \varepsilon_{i,j}^{[v_a,v_b]} + \varepsilon_{j,i}^{[v_a,v_b]})) \tag{3.19}$$

$$\beta.s_i.T + f_{j,n}^{[v_a,v_b]}.\phi + \delta \leq \alpha.s_j.T + f_{i,m}^{[v_y,v_a]}.\phi + M(1 - \omega + \varepsilon_{i,j}^{[v_a,v_b]} + \varepsilon_{j,i}^{[v_a,v_b]})) \tag{3.20}$$

$$\forall f_{i,m}^{[v_x,v_a]}, f_{i,m}^{[v_a,v_b]}, f_{j,n}^{[v_y,v_a]}, f_{j,n}^{[v_a,v_b]} \forall F^4, \forall \alpha \in A, \forall \beta \in B \tag{3.21}$$

3.6.2. Implementation details

In an ILP program, there are two pieces, the model itself that, in our case, we implemented with Pyomo, and the other is the mathematical solver that has to be installed in the same Docker container as the model. There is a vast amount of solvers available, and some of them are Open Source. Table 3.6.2. depicts some of the most used ILP solvers available, including a small description to differentiate between them and the license type necessary for their usage.

```
#opt=SolverFactory('gurobi',solver_io="Python")
opt=SolverFactory('glpk')
self.instance = self.model.create_instance()
self.results = opt.solve(self.instance)
self.instance.solutions.load_from(self.results)
```

It is important to remark that the solver doesn't maintain any relationship with the Pyomo code, and changing between one another is as easy as changing a line in the code (Code Sniped A.1 shows how to do it). However, the installation process differs, as some are available as Open Source GitHub repositories, others are available as PIP packages for Python, and the others have to be downloaded with Conda [36] package manager for Python. In our scenarios, we analyzed GLPK and Gurobi. Later sections deepen this comparison.

Solver Name	License Type	Description
AMPL	Free reduced version, Pay license	Focus on maintainability, integrates a common language for analysis and debugging [37]
PIco	Open Source, available in pip	Allows you to enter an optimization problem as a high-level model, with painless support for vector and matrix variables and multidimensional algebra [38]
CBC	Open Source	An open-source mixed-integer program (MIP) solver written in C++. CBC is intended to be used primarily as a callable library to create customized branch-and-cut solvers [39]
GLPK	Open Source	It is a set of routines written in ANSI C and organized in the form of a callable library. Is intended for solving large-scale Linear Programming (LP) and mixed-integer programming (MIP) [40]
Gurobi	Student license Pay license	According to their website, Gurobi claims to be the fastest and most powerful mathematical programming solver available for your Linear Problems [41]

Table 3.1: Different Mathematical Problem Solvers considered in this project

However, we decided to use GLPK in the final design for the ILP microservice as it is and Open Source implementation and Gurobi due to the transient nature of microservices and Docker containers (i.e., they can be replaced for another container when they present an error); it requires the acquisition of a special license for being used inside of Docker containers.

Regarding the architecture of the microservice by itself, several small pieces work together to perform the full task; the two most important pieces are the ILP module, described above, and the Solution Visualizer. The Solution Visualizer is a module also written in Python whose principal task is to facilitate the exploration of the solution that the ILP is provided to the current scheduler problem. This module references what is described in the motivation scenario but adds the network topology and information of the different modules.

3.7. Scheduler Postprocessing

The last microservice in the pipeline of the stream is the Scheduler Postprocessing Microservice which processes the information containing the schedule and network configuration coming from the ILP Microservice. Figure 3.9 depicts the general interaction between the sub-modules of the microservice and the other microservices. As in the previous elements, we implemented RabbitMQ in the microservice for communicating with the ILP microservice. Once the JSON containing the information is received and parsed, it is sent to the *ConfGen* submodules, The *VLAN configurator*, and *TAS configurator*. The

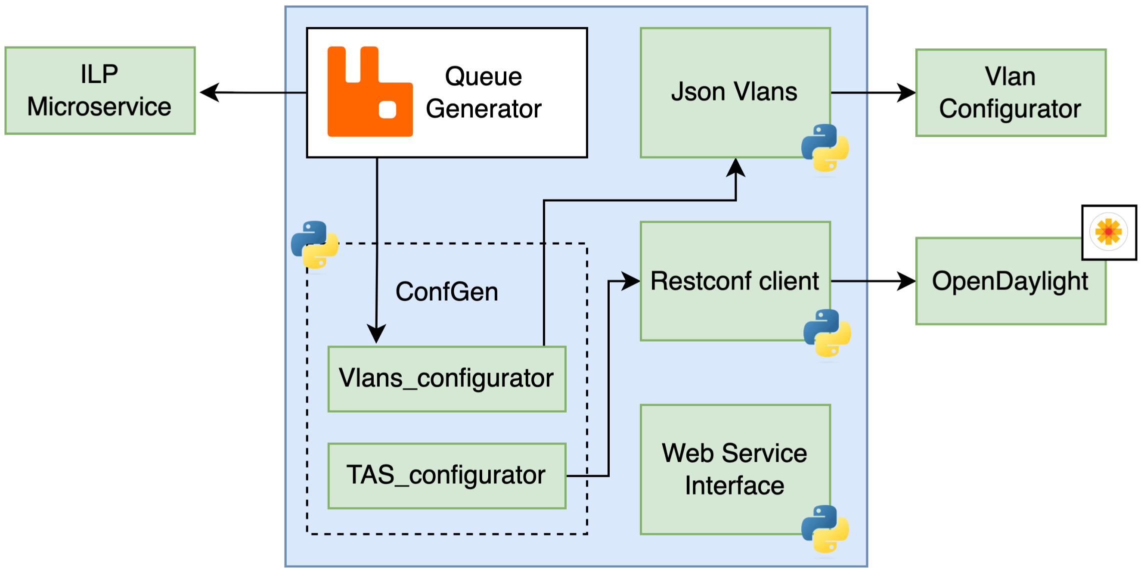


Figure 3.9: Overview of Scheduler Post Processing Microservice

first one is responsible for generating the information necessary for configuring the VLAN in the TSN Switches; this is out of the scope of the TSN standard and is performed by the VLAN configurator microservice. The second one creates the JSON payload containing the configuration to be sent through RESTCONF to the Opendaylight container.

Specifically, the *TAS configurator* sub-module uses the IEEE 802.1Qcc standard *ieee802-dot1q-sched* YANG module. This module defines all of the parameters for the TAS TSN switch setting; we mapped each to an output of the previous microservices. Normally, the *ieee802-dot1q-sched* module defines the configuration for one single TSN Device. However, the SDN controller provides an inventory system for storing the network information and YANG capabilities of all the devices managed in the SDN controller domain; this leads to centralizing the configuration of all the elements into a single configuration payload sent by RESTCONF. The SDN controller will parse this unified payload and will generate NETCONF sessions in each device covered in the configuration to set its *ieee802-dot1q-sched* YANG model. If the controller is not implemented in the architecture, the system would need to generate a NETCONF payload for each network device in the network to be managed by the CNC, which would be harder to implement, maintain and monitor since NETCONF goes through SSH and not in HTTP as RESTCONF.

Lastly, we included a web server that prompts the scheduler and network configuration as depicted in Figures 4.1, 4.2, 4.3. This web server is accessible by the system IP address (i.e., Kubernetes cluster IP, Docker-compose IP) with port 8080.

3.8. VLAN configurator

It is important to note that in TSN, all flow communications from source A to destination B are encapsulated within VLANs defined in the IEEE 802.1q standard. Therefore, all the VLAN configurations must be carried out in the devices, which entails designing a microservice that generates the necessary orders for the devices to apply these configurations.

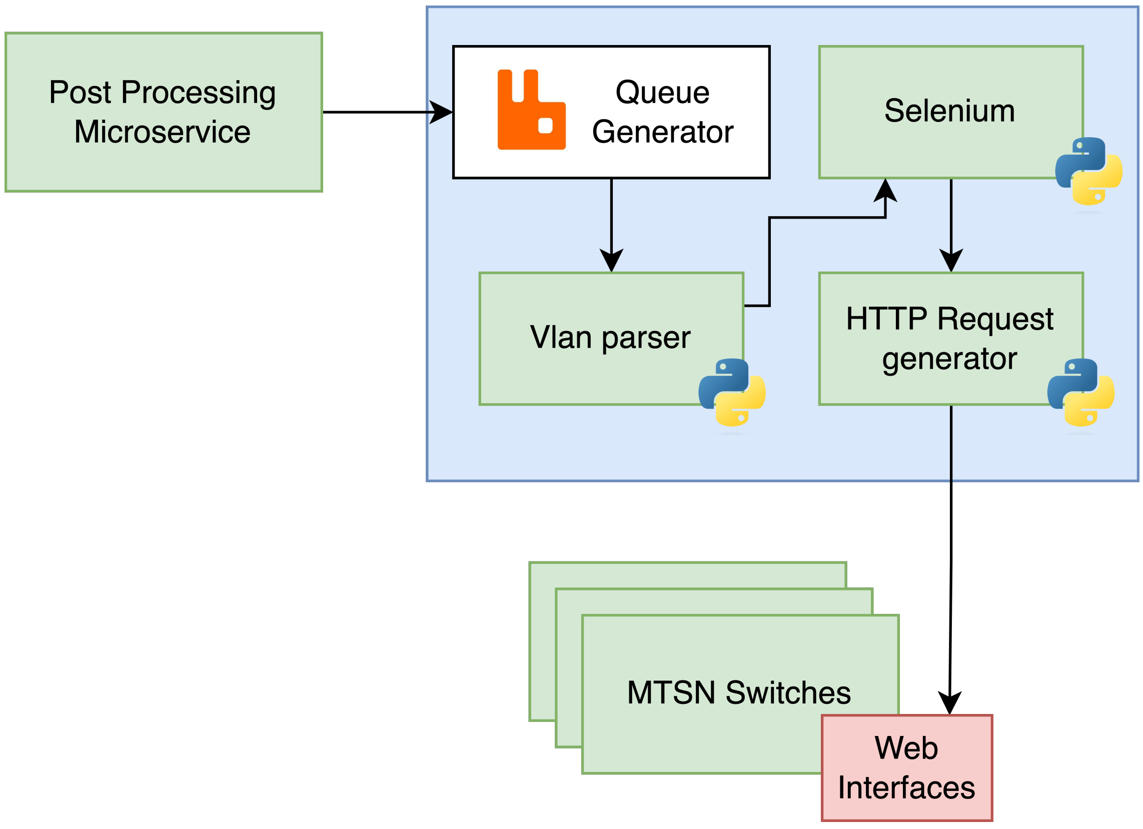


Figure 3.10: Overview of VLAN Configurator Microservice

Ideally, an SDN communication interface, such as those used in the 802.1Qbv protocol, would be used to configure the TAS using YANG and RESTCONF/NETCONF models. However, the MTSN Kit devices do not have these characteristics. Therefore, the VLAN configuration microservice was specifically designed to be used in the MTSN Kit. Although the MTSN Kit has a web interface for configuring VLANs, this microservice takes as input the sources and destinations of each flow received via RabbitMQ and performs the necessary configurations using Selenium in Python. Figure 3.10 depicts the functioning of this microservice.

This means this microservice will not work with equipment other than the MTSN Kit. However, due to the modular nature of the architecture, it is possible to build another microservice that does the configuration using YANG models. It will only be required to create the configuration using the model provided by the IEEE [42].

3.9. SDN controller

Finally, the SDN controller is the microservice that acts as the interface between the CP and the network. It has a RESTCONF-type communication interface to communicate with the post-processing microservice and a NETCONF communication interface to communicate with the computers on the network. The main reason why this microservice is used for communication with TSN switches instead of a direct NETCONF communication between the post-processing microservice and network equipment is due to the simplicity and benefits of an SDN controller. Figure 3.11 shows the operation diagram of the SDN controller microservice.

In our case, we choose OpenDaylight as the SDN controller for our CP. OpenDaylight includes a module for YANG models and RESTCONF/NETCONF that allows storing all exposed models in a device inventory. By incorporating the list of devices with their respective IP addresses, the controller automatically creates a data model that allows all devices to be accessed simultaneously using a single RESTCONF communication.

Therefore, it is only necessary to send a message containing all the configurations that must be applied to the equipment to configure the TAS with the offsets provided by the ILP. The process that this microservice follows is as follows:

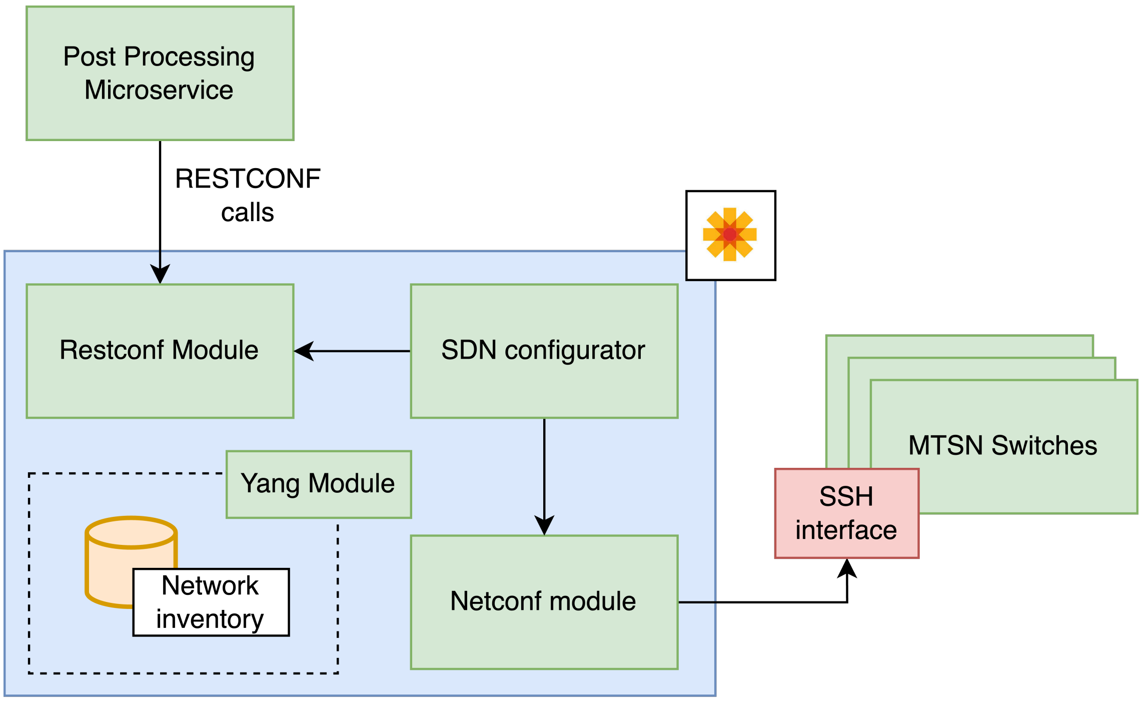


Figure 3.11: Overview of SDN Controller Microservice

- The post-processing microservice sends the necessary configuration, including device IP addresses, using the RESTCONF protocol via REST API-like communication.
- OpenDaylight creates an inventory of all the computers on the network and generates a data model that includes all the YANG models exposed in the NETCONF interfaces of the computers.
- Using NETCONF, OpenDaylight accesses all computers and configures them by following the directions provided by the post-processing microservice.

CHAPTER 4. ANALYSIS

This chapter consists of three sections. The first section aims to validate the results the ILP solver provides of the μTSN-CP. To do this, we present a stream and topology visualizer developed as part of the code within the post-processing microservice. This display shows information about the frames that make up each stream as well as the temporal distribution of these frames during the hypercycle time period. It also includes a system topology diagram and a color display system that eases traffic visualization on each link.

The second section describes the system we deployed on the EETAC laboratory. This topology includes the computers where the microservices, the end-stations, and the switches are deployed. Moreover, as well as distinguishing between the management and data layers. Considering that the computer that contains the microservices, it is essential to remember that this system is modular, and as long as the communication between the microservices and the switches is guaranteed, the computer is easily replaceable by a cloud infrastructure using technologies such as Kubernetes or others container management systems such as Fargate, Openshift, or Docker Swarm.

In the third section, the different methods and variables that exist to improve the performance of the ILP are analyzed, understanding performance as the speed at which the system provides a solution. In addition, we included a qualitative analysis of how to enhance such characteristics by using tools typical microservices architectures to achieve the best possible performance.

4.1. Scheduling Solution Inspection

This subsection aims to verify that the results obtained by the combination between the Dijkstra Algorithm output and the ILP scheduling are correct. As we have specified in the previous chapter, the main objective of the μTS-CP is to be able to perform all the necessary calculations to find a specific and viable solution for a set of flows, taking into consideration that each stream will have an origin and destination, as well as characteristics in terms of bandwidth and maximum latency. These requirements are imperative, and it is impossible under any circumstances that a solution to a stream scheduling problem does not meet their needs. We implemented a scheduling Solution Visualizer included in the post-processing microservice to verify this goal.

As we can see in the Figures 4.1 4.2 4.3, the scheduling problem viewer includes:

1. **Network topology:** specifying the network nodes and the links between them. This element is located in the upper left corner of the image.

2. **Description of the Stream matrix:** This includes a set of parameters such as the matrix of network links, the number of frames in each Stream, the period of the flows, specification of the order of the links to be used according to the Dijkstra algorithm for each Stream. These are written in two-dimensional vectors and dictionaries; the order in the array is the stream identifier. The parameters are in the upper right corner of the diagram in red rectangles.

3. **Gantt Diagram:** This defines each link with a specific color; on the horizontal axis, we have the time in milliseconds, while on the vertical, we have the frames of each stream in each link. The notation described is *('S': Frame stream number, 'L': Link number through which the frame is traversing, 'F': Frame order within the stream)*, this graph is located at the bottom of the scheduling problem viewer.

Below there are three sections showing examples of the output provided by the visualizer after finding a solution to the scheduling problem. It is important to note that these examples were tested theoretically and using a microservice that generates random networking and TSN scheduling problems.

4.1.1. First TSN Example

In this example, we present the features of the Scheduling problem included in Figure 4.1:

- Four-node network, where Node 1 connects all other nodes.
- Two frames, each with a single subframe. One of the frames has a period of 5000ms, while the other has a period of 2500ms.
- The fifth red box at the top left of the diagram contains an array of arrays that describes the paths from source to destination obtained using Dijkstra's algorithm. The first array refers to the set of routes of all the flows. The second subarray shows the links that each stream must traverse from its destination to its origin, and each subarray contains a tuple of the two nodes that make up the link. In this example, the first stream traverses from Node 2 to Node 3, passing through Node 1, while the second stream traverses from Node 0 to Node 3, also passing through Node 1.

In the Gantt chart at the bottom of the Figure 4.1, the Y axis represents the offset of each subframe of each stream in a specific link. The X-axis represents time in milliseconds. To improve visualization, the visualizer draws each time slot with a different color when they correspond to the same link.

This solution fulfills all of the conditions of a TSN scheduling; there is no overlap between frames in the same temporal space and the same link; this is clear since no frame shares the same temporal moment with another frame of the same color. In addition, the example fulfills all the normal characteristics of any transmission: the streams pass through the links in sequential order. In the case of stream 1, which has half the period of stream 0, you can see the repeats in the hyper period after 2500 milliseconds.

4.1.2. Second TSN Example

Here we have the following characteristics that describe the TSN scheduling problem of Figure 4.2.

- 5-node network, where Node 2 connects all other nodes. Also, nodes 3 and 4 have a direct link between them.

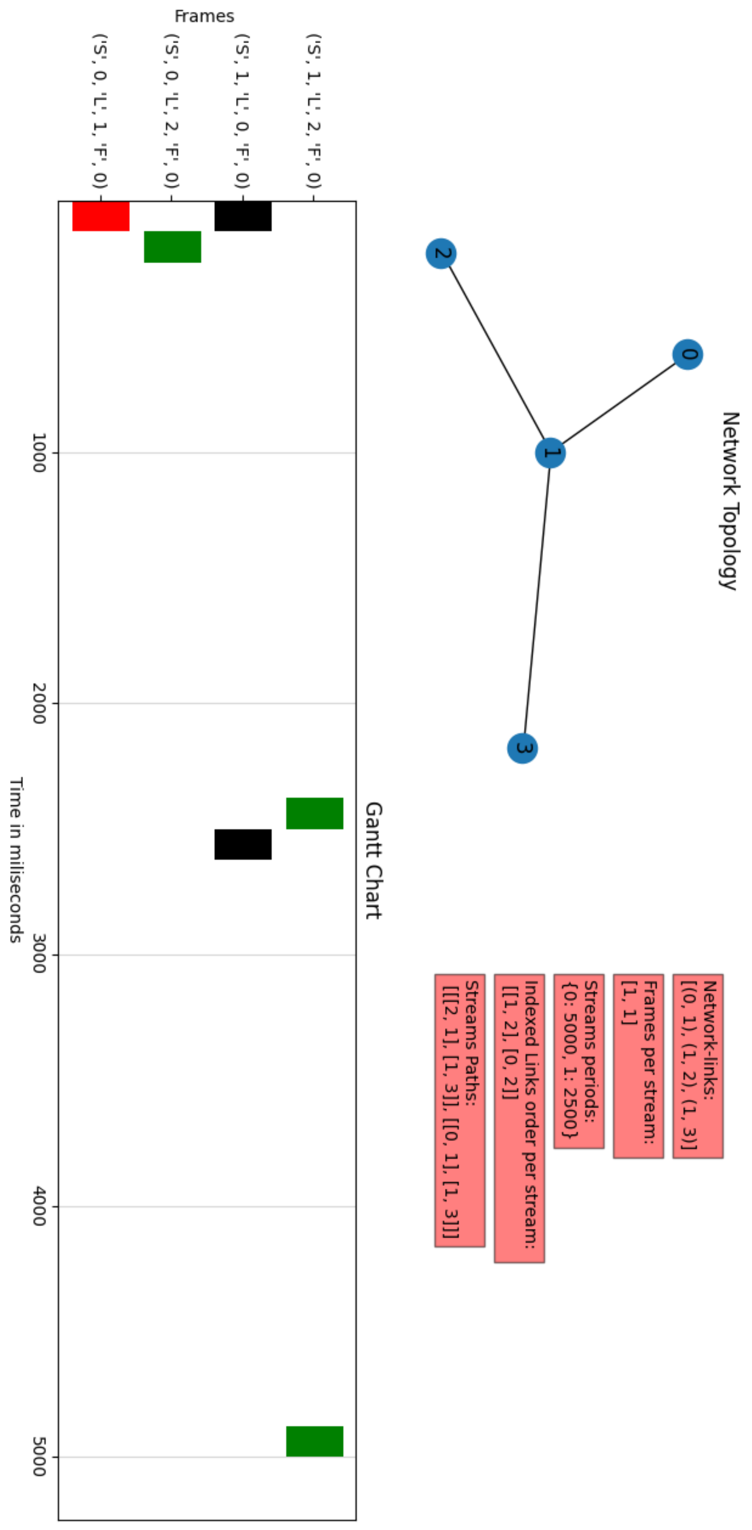


Figure 4.1: First Scheduler problem

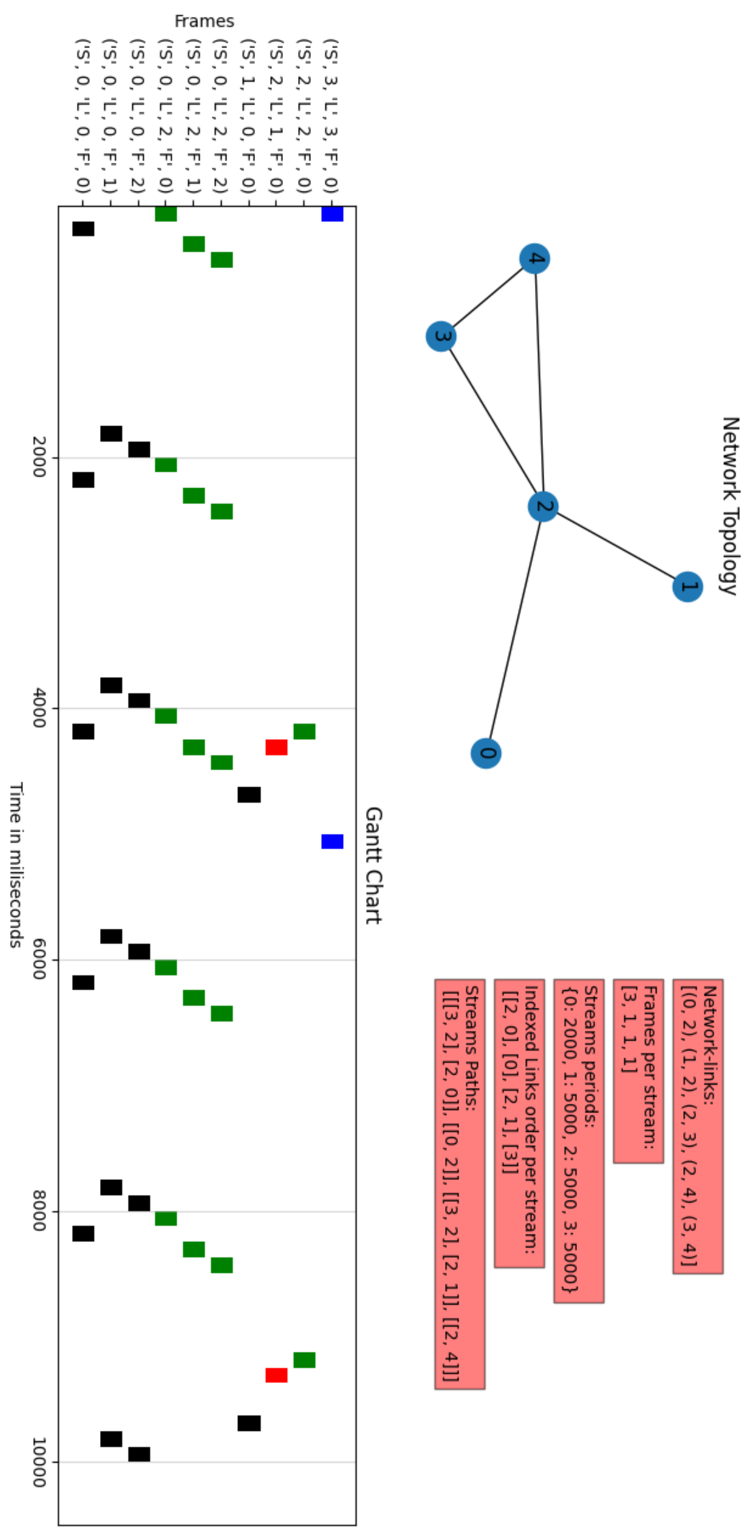


Figure 4.2: Second Scheduler problem

- Four streams, the first consists of 3 frames, while the others have only one frame. The first frame has a period of 2000ms, while the others have periods of 5000ms, which makes the hyper period 10000ms.
- The stream distribution is as follows:
  - Stream 0: Origin at Node 3 and destination at Node 0, passing through Node 2.
  - Stream 1: Origin at Node 2 and destination at Node 0.
  - Stream 2: Origin at Node 3 and destination at Node 1.
  - Stream 3: Origin at Node 2 and destination at Node 4.

This example is noticeably more complicated than the previous one. It involves a larger number of frames, a larger network, and more conditions to fulfill for the ILP due to the repetitions of some streams within the hyper period. However, when looking at the image, we can appreciate that there is no overlapping of streams in any of the links simultaneously, even considering the repetitions in the hyper period. In addition, the system can predict when a stream will repeat itself and occupy a temporary space in a link, thus avoiding configuring another stream at that moment or in its multiples.

It is interesting to note that stream 0 never interrupts stream 1, even though they both use Link 0. However, the most crucial moment occurs with streams 0 and 2 on Link 2. Although stream 2 could have been transmitted consecutively, this option would interfere with when stream 2 transmits over Link 2. Therefore, the system decides to wait, even if this occurs in the third repetition period.

4.1.3. Third TSN Example

Here we have the following characteristics that describe the TSN scheduling problem of Figure 4.3:

- 5-node network, where Node 0 connects the other nodes, but several nodes have direct connections to each other.
- Four streams, two of them made up of 3 frames; the other two have only one frame. The first two frames have periods of 2000ms, while the others have periods of 5000ms, making the hyper period 10000ms as in the previous example.
- The Stream distribution is as follows:
  - Stream 0: Origin at Node 2 and destination at Node 3, passing through Node 0.
  - Stream 1: Origin at Node 3 and destination at Node 1.
  - Stream 2: Origin at Node 3 and destination at Node 1.
  - Stream 3: Origin at Node 4 and destination at Node 2.

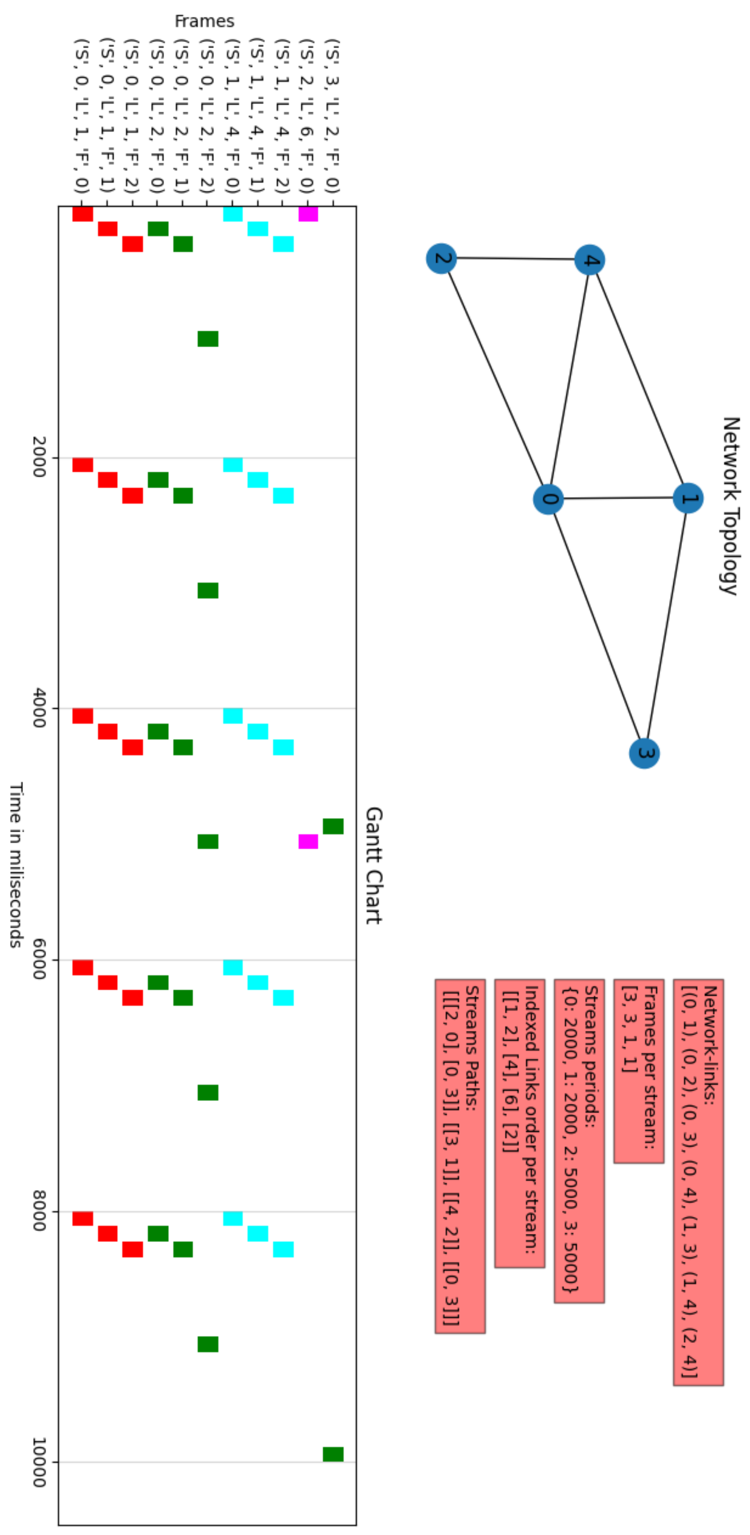


Figure 4.3: Third Scheduler problem

In this example, we have increased the number of frames included and used a topology with a higher number of links to verify that the solutions delivered by our solver meet all the constraints for a TSN scheduling system. We can notice that in none of the links, at any time, does the location of one frame overlap with another, even considering the repetitions that the frames may have within the hyper period. It is important to note that our second link, which uses the most streams, keeps its properties intact, ensuring that no two frames from streams 3 and 0 are ever found occupying the same temporary space.

4.2. Laboratory Setup

In this section, our objective is to graphically show the distribution of the elements that make up the laboratory assembly to test the correct functioning of our solution. Our solution functions when we can accurately configure the switches with the appropriate VLAN settings. In other words, it transforms the offsets into settings for the MTSN kit equipment.

Below is a detailed list of the elements that make up Figure 4.5. First, it is necessary to distinguish between the data plane and the control plane. The data plane consists of all the components within the red block, as the diagram legend specifies. As we are in a TSN architecture, these elements have Hardware interfaces capable of communicating according to the IEEE 802.1 standards that define TSN.

Among these elements, we find:

1. The two TSN switches (SW0, SW1) that make up the MTSN kit. As mentioned above, these switches can communicate using TSN standards. Each Switch has 4 TSN interfaces and communicates with the other through the ETH3 interface and its respective end-station through the ETH0 interface. Because TSN is used for communication, switches must have some time synchronization mechanism, such as PTP, built into the system of the switches.
2. The End Stations: In our assembly, we used two computers with Linux, specifically Ubuntu 22.01 LTS. These computers can communicate according to IEEE 802.1 standards through the Intel i210 NIC. In addition, they have the necessary configuration to perform PTP synchronization through Linux PTP within the Linux Kernel. These devices communicate with each other for industry-standard communications using the OPC UA protocol. It is important to highlight that design and prototyping are not part of our work; the deliverable documents of the Bachelor's degree thesis of Jordi Cros include a full description of them [43].

On the other hand, the elements inside the blue block correspond to the control plane. These elements are not required to use TSN for communication but are responsible for orchestrating and managing the data plane elements. Among these elements, we find:

1. The interfaces of Port Z of the TSN switches, these NICs are used only for management tasks. NETCONF protocol exposed the YANG models, and since SSH encapsulates NETCONF, we can also use SSH for controlling purposes.
2. A layer two Switch that allows us to have all the elements of the control plane within the same network. Therefore, if the TSN network is larger, all the Z Ports of the TSN

switches should be connected to this Switch or to any switch array that abstracts the communication between them into a single network.

3. μTSN-CP, the block in the upper center of the diagram, represents our prototype. We deployed it on a single computer with Docker-compose installed in the lab deployment; this highlights another advantage of using microservices, as our controller can run on any machine with Docker installed. It is important to mention that this block is simply an abstraction, and to take full advantage of the microservices architecture; the µTSN-CP should ideally be in a private or public cloud, running on some controller orchestration tool like Kubernetes, Openshift, or Docker Swarm. Carrying out this transition in the laboratory assembly is perfectly possible. It is only necessary to guarantee that the interface exposed by the controller and the rest of the elements of the control plane can communicate on the same network.

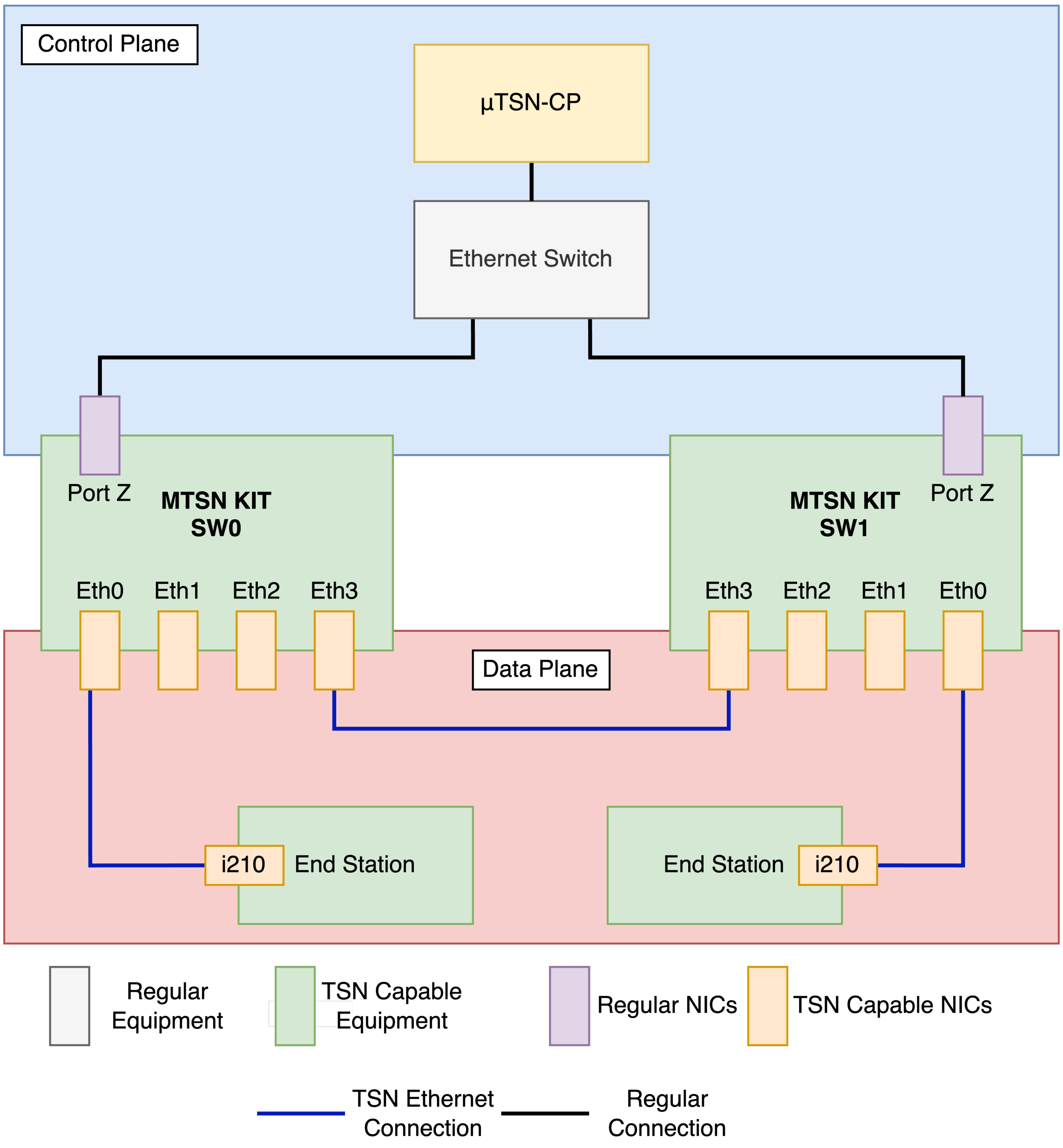


Figure 4.4: Laboratory Setup description

4.3. Analysis and Results

In this section, we compare the existing methods to improve the performance and resource usage of the microservices-based SDN controller for TSN. First, it is essential to remember that at the system core, there is a microservice with an ILP model that defines the constraints and the objective function of the TSN stream scheduling problem. Said microservice uses a predefined solver to find the solution; it can be any available in the state-of-the-art of ILP Solvers, including those shown in Table 3.1. This work compares GLPK and Gurobi as chosen solvers for the ILP model in terms of the time they take to offer a solution for a given number of streams in a defined topology.

Secondly, another condition that can affect the performance of our system is the Hardware of the machine in which the ILP calculator microservice runs. Such parameters include the number of processor cores, available RAM, and processor frequency are factors to consider. Therefore, we conducted tests with different machines where the Hardware parameters changed. For this, we use the following machines:

1. A local laptop with a 4-core Intel Core i7-7820HK processor with a maximum frequency of 3.9GHz, and 16 GiB of RAM.
2. An Elastic Compute Cloud (EC2) instance on Amazon Web Services (AWS) of type T3.small with a 2-core Intel Xeon Scalable processor with a maximum frequency of 3.1 GHz and 2 GiB of RAM [44].
3. An EC2 instance on AWS of type M5Zn large with a 4-core Intel Xeon Scalable processor with a maximum frequency of 4.5 GHz and 16 GiB of RAM [45].

We chose these machines for having a point of comparison in which the number of cores, the clock speed of the same, and the RAM will vary. It should be noted that, according to the literature, the actual performance you will get from using multiple cores in an ILP problem depends on several factors, such as the nature of the optimization problem, the size of the model, the availability of memory resources, and the specific system configurations [46]. In the test, like in the previous one, it will be verified, which is the option that provides a solution to the planning problem in the shortest time.

Finally, with the results of the two previous experiments, we select the best ILP model solver and compare the two machines with the best performance to identify what characteristics the machine running the microservice should have, both in terms of Hardware resources as a solver.

The characteristics of the experiments were the following:

1. All were made using the topology shown in Figure 4.5.
2. For flows, a custom microservice was used that creates the inputs of the Jetconf microservice, generating an input sent through the RabbitMQ queue.
3. All streams have a deadline of 500ms.
4. The periods of the streams vary between 125ms, 250ms, and 500ms with the same probability.

Number of Streams	GLPK				Gurobi			
	Average	Min	Max	Std. dev.	Average	Min	Max	Std. dev.
15	4.768	1.895	27.772	5.621	0.620	0.542	1.410	0.182
20	23.113	10.431	55.086	12.329	0.758	0.700	0.926	0.047
25	58.503	30.810	80.636	13.622	0.978	0.909	1.439	0.111

Table 4.1: Seconds to find a solution - GLPK vs. Gurobi including Average, minimum and maximum values, and standard deviation

5. It is assumed that all links are over a 1Gb/s channel.

6. The streams can be composed of 1 to 3 Frames with the same probability.

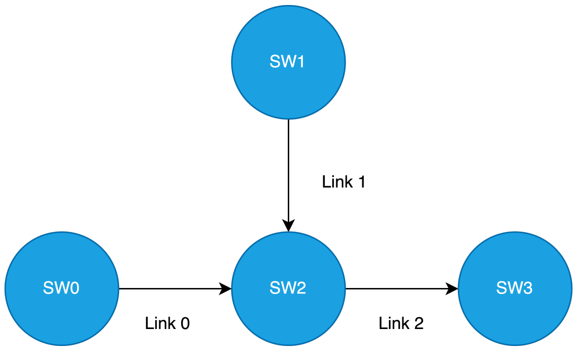


Figure 4.5: Topology used for tests

4.3.1. ILP Solvers Comparison

It is essential to mention that the literature already indicates what the comparison between GLPK and Gurobi will yield. Various articles show that Gurobi generally offers better resolution times and performance than GLPK, especially for large and complex problems [47]. Regarding our test, we increased streams from 15 to 25 streams per problem with jumps of 5 streams between each test. We repeated each test 100 times for each number of streams value.

Indeed, in agreement with the literature, Figure 4.6 shows that the performance of Gurobi is considerably superior to that of GLPK. While the time to find a solution using GLPK increases considerably, reaching up to 27 seconds maximum and 4.76 seconds average in the first test with 15 streams, up to a maximum of 80.63 seconds in the test with 25 streams. It is important to mention that in Figure 4.6 as well as in the rest of the graphical comparison, the colored bar is the average value, and the interval denoted in each bar is between the minimum and maximum values. On the other hand, Gurobi times were never higher than 1.5 seconds, with average values of less than 1 second for all tests. On the other hand, an aspect that greatly differentiates the two solvers is the variance that they deliver. Table 4.1 depicts the average time to solution value and the maximum and minimum values, and the Standard Deviation (SD). Considering this last value, Gurobi offers higher performance for our ILP model and, in turn, provides greater consistency in its performance.

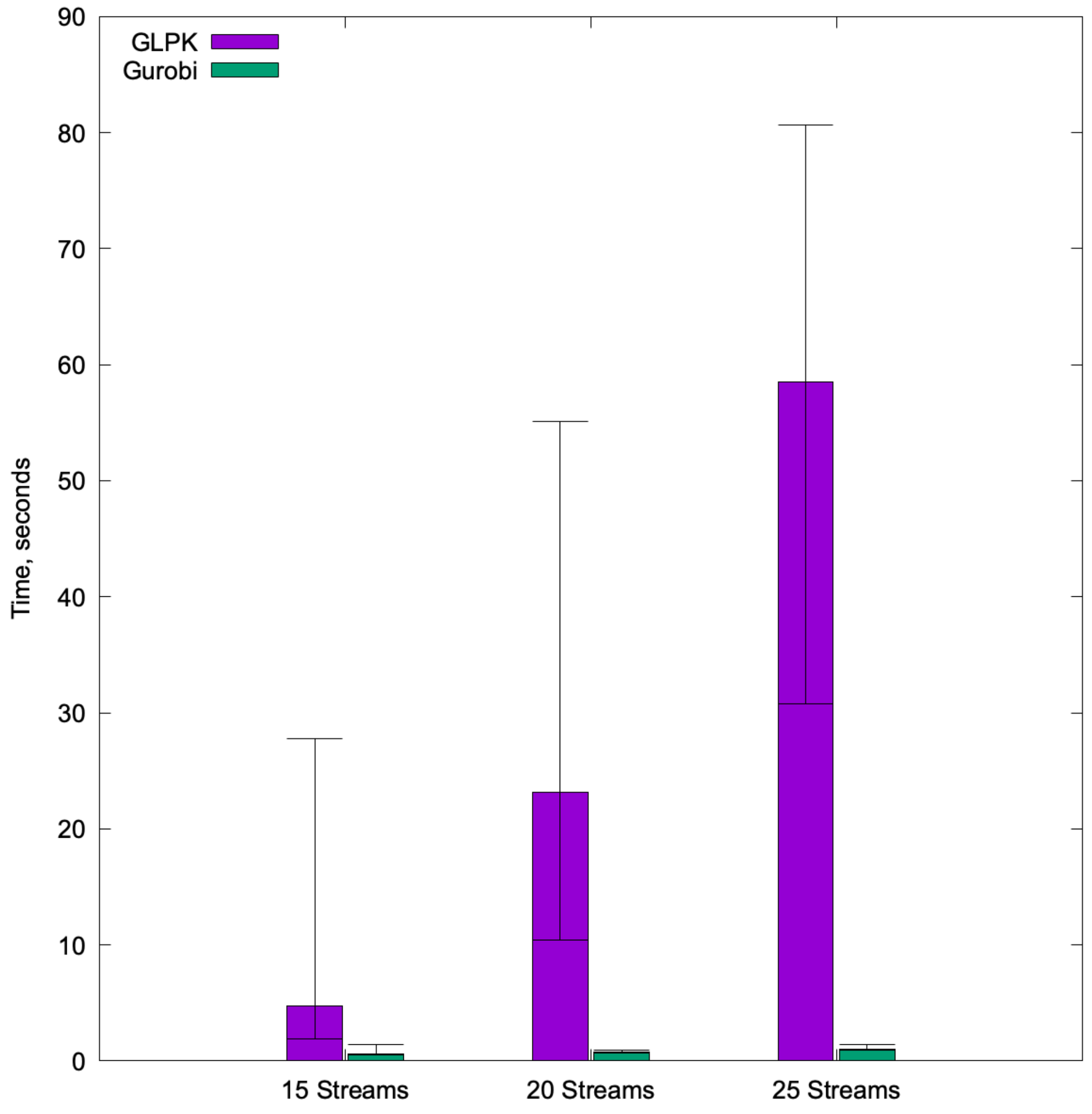


Figure 4.6: Seconds to find a solution - GLPK vs. Gurobi

4.3.2. Hardware configurations Comparison - Using GLPK

We carried out this experiment following the same patterns as the previous one, but this time the final test was with 30 streams within the planning problem. As ILP solvers, we used GLPK to determine how much the Hardware affects timing to get a solution to the scheduling problem. We considered RAM, the number of processor cores, and its operating frequency as Hardware features.

This time, in search of simplicity and since the values change considerably between different streams number, we decided to present 3 subgraphs within the Figure 4.7. Each subgraph refers to a defined number of streams and has a Y-axis that covers different ranges. The figure depicts that the best results were obtained using the M5Zn large instance, although the difference with the local machine is not too big. The worst were those obtained with the T3 small instance, which presents much higher solution times than the other two machines.

The T3 instance has completely different Hardware characteristics than the other two, it has much less RAM, fewer processor cores, and perhaps the most important feature, its processor frequency is much lower than the other two machines. On the other hand, the local machine and the M5Zn large only differ in their clock frequency, having the same number of cores and the same RAM.

This test indicates that the Hardware characteristics significantly affect the time to obtain a result. The tables 4.2, 4.3, and 4.4 show the average, maximum, and minimum values for each test as well as the SD of the values collection. We highlight that all the machines' results present considerable variations attributable to the less consistent performance of GLPK. The next test, where we only used Gurobi as an ILP solver, shows the best performance consistency.

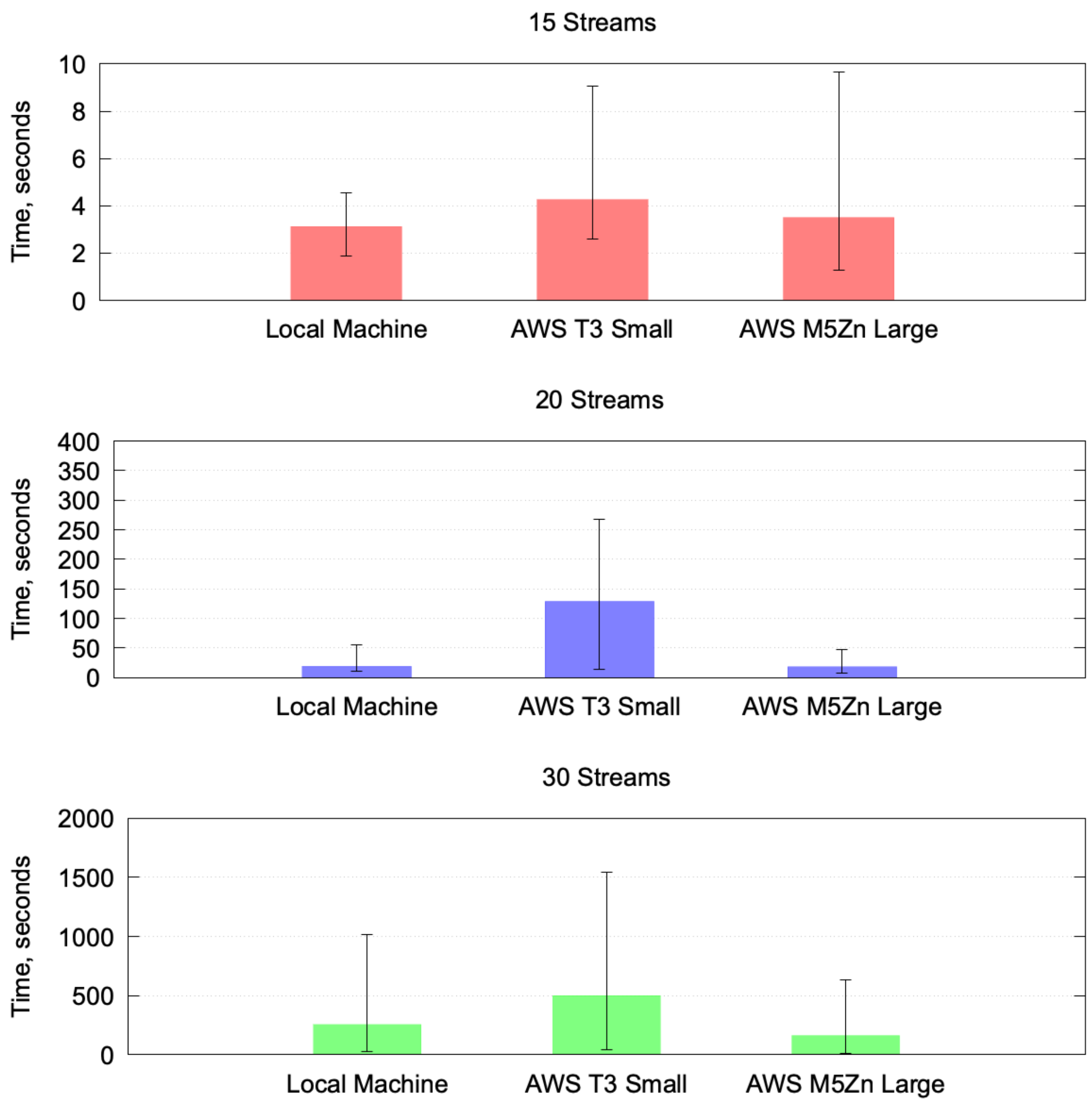


Figure 4.7: Seconds to find Solution - Local Machine vs T3 Micro vs M5Z Large - Using GLPK

Instance	Average	Min	Max	Std. dev.
Local Machine	3.125	1.895	4.560	0.798
AWS T3 Small	4.263	2.584	9.051	1.484
AWS M5Zn Large	3.503	1.287	9.672	2.426

Table 4.2: Seconds to find a solution - 15 Streams Local Machine vs. T3 small vs. M5Zn Large - Using GLPK

Instance	Average	Min	Max	Std. dev.
Local Machine	19.154	10.431	55.086	11.244
AWS T3 Small	129.062	13.994	267.103	75.260
AWS M5Zn Large	18.370	7.488	48.252	11.461

Table 4.3: Seconds to find a solution - 20 Streams Local Machine vs. T3 small vs. M5Zn Large - Using GLPK

4.3.3. Hardware configurations Comparison - Using Gurobi

Finally, in this test, we decided to use the two instances that showed the best performance in the previous test, the Local machine and the M5Zn Large instance on AWS. In addition, we used Gurobi as an ILP solver to obtain the best possible performance. It is essential to emphasize that according to AWS, M5zn instances deliver the highest all-core turbo CPU performance from Intel Xeon Scalable processors in the cloud; therefore, they are specifically designed to deliver the highest frequency in their processors. As we described at the beginning of the section, the local machine and the M5zn Large instance only differ in their processor frequency since the amount of RAM and the number of cores are the same.

Using Gurobi, the time to find solutions is considerably fast; for that reason, to differentiate both machines, we increase the number of streams in the same planning problem in all the tests. Therefore, the considered problems are 40, 45, and 50 streams. As Figure 4.8 depicts, the M5Zn instance performs much better than the local machine for all tests. On the other hand, the difference between the maximum and the minimum values in the tests is considerably larger in the local instance. That fact also indicates that the consistency of the performance with the M5Zn Instance is greater; this is also visible in the comparison of the SD for both cases. Table 4.5 shows the average time to obtain a solution, the maximum and minimum values, and the SD obtained in the 100 repetitions of the tests.

This indicates that the clock frequency of the CPI for our particular problem is a transcendental parameter when it comes to reducing the time to obtain a solution. All the tests indicate that the best combination to obtain the lowest load times is with Gurobi and the M5Zn Large instance.

Instance	Average	Min	Max	Std. dev.
Local Machine	255.013	30.810	1013.215	302.593
AWS T3 Small	501.388	45.194	1546.723	478.889
AWS M5Zn Large	162.642	12.672	634.854	135.549

Table 4.4: Seconds to find a solution - 30 Streams Local Machine vs. T3 small vs. M5Zn Large - Using GLPK

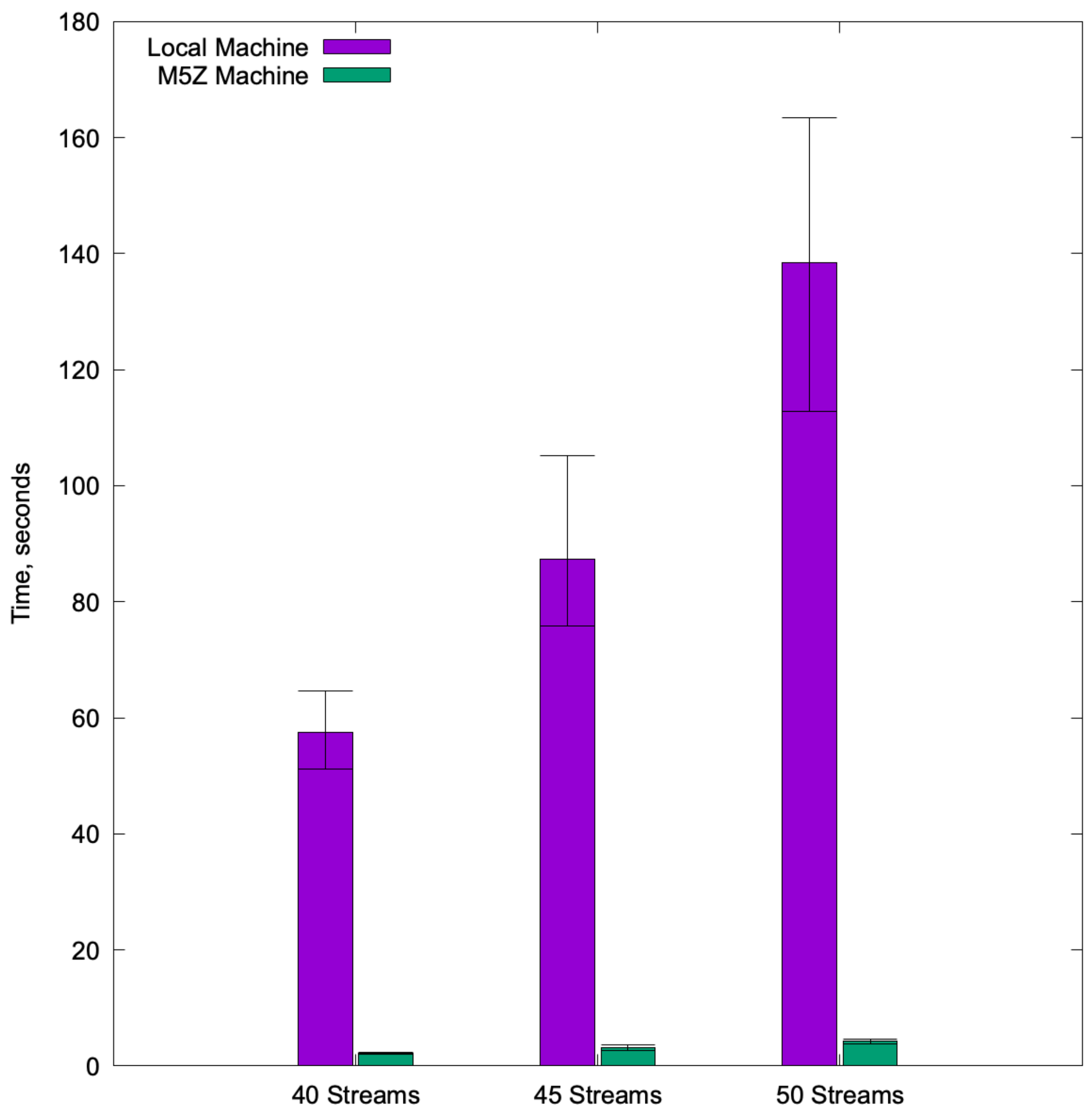


Figure 4.8: Seconds to find Solution - Local Machine vs. M5Z Using Gurobi

We have made various comparisons that have allowed us to observe that certain features have a greater impact on improving system performance. In particular, we have concluded that the processor's operating frequency is directly related to the execution time needed to solve the TSN problem. Using the best virtual machine available on AWS, the M5Zn Large, we have achieved significantly higher performance than the local machine and T3 small. Also, of the two solvers we tested, Gurobi offers the best performance with one significant difference, which heavily influences processing time.

4.4. Microservices as deployment strategy

Using microservices allows us to take full advantage of these features. In particular, since the ILP microservice is the one that consumes the most resources, we can optimize the resources if we run the ILP on a specially designed machine with a high frequency of operation. This optimization is not possible in a monolithic architecture since all the pieces of

Number of Streams	Local Machine				M5Zn Large			
	Average	Min	Max	Std. dev.	Average	Min	Max	Std. dev.
40	57.570	51.236	64.660	4.675	2.201	2.024	2.328	0.089
45	87.403	75.829	105.118	8.256	3.070	2.673	3.598	0.212
50	138.381	112.855	163.418	14.569	4.255	3.779	4.573	0.247

Table 4.5: Seconds to find a solution - Local Machine vs M5Zn Large - Using Gurobi

the Software are tightly integrated, and separating these components would be impossible. However, microservices are specifically designed to allow for this separation.

To achieve the ideal scenario, the ILP runs on a high-capacity Hardware computer, with the maximum operating frequency for its processor and using Gurobi. In contrast, the rest of the elements run on general-purpose machines; we can use container orchestration tools such as Kubernetes or Openshift.

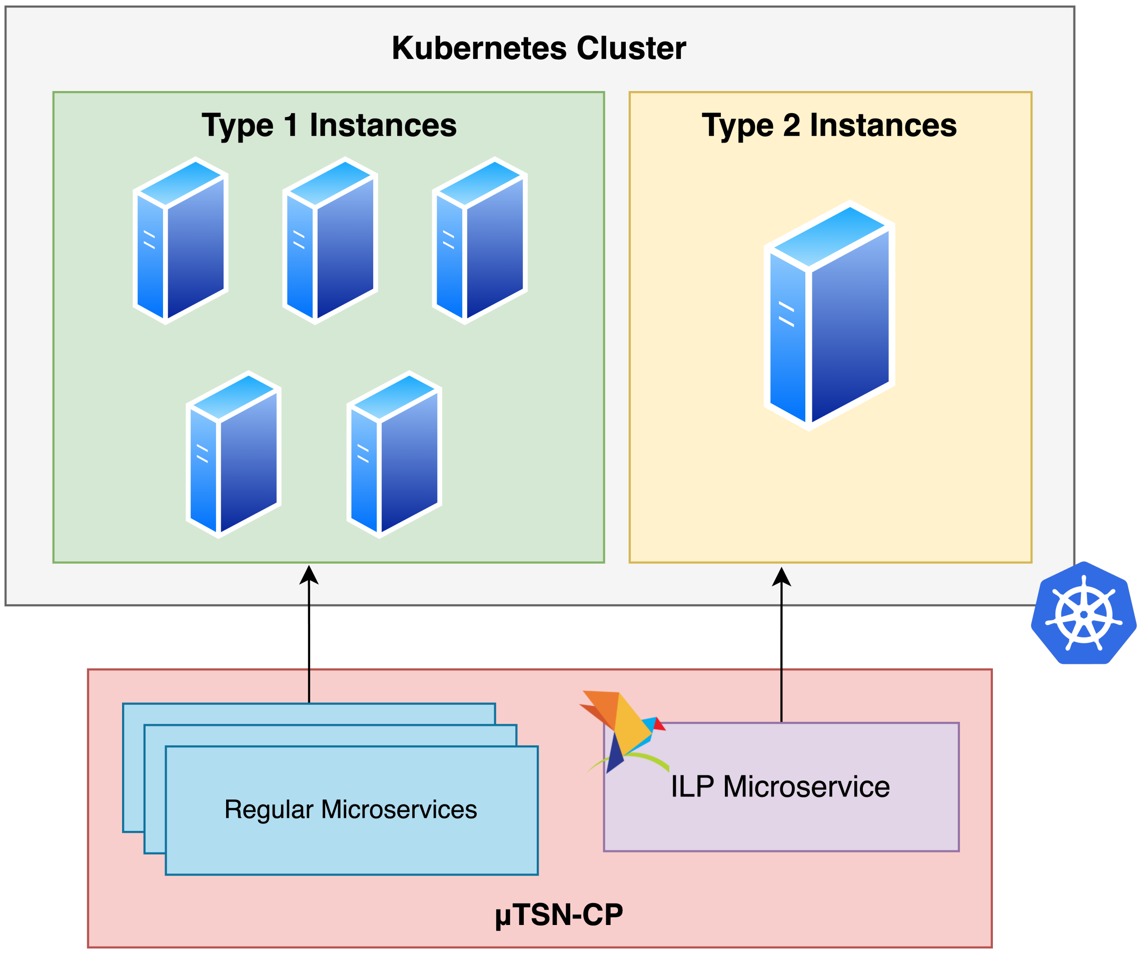


Figure 4.9: Microservices Distribution in a Kubernetes Cluster

Kubernetes and Openshift are two related technologies but with significant differences. Kubernetes is a widely adopted, highly scalable, open-source container orchestration platform. It provides robust tools for container cluster management, application deployment, and autoscaling. On the other hand, Openshift is a Kubernetes-based application platform developed by Red Hat that offers an additional layer of added value by providing a more complete and enterprise-ready experience. Openshift incorporates Kubernetes but adds additional features and functionality, such as deployment automation, application lifecycle management, and a more intuitive and simplified approach to application deployment. In short, while Kubernetes is a more basic and flexible option, Openshift is focused on offering a complete and enterprise solution based on Kubernetes.

Both solutions allow you to manage where and when the containers containing the microservices are deployed and store their configuration parameters and secrets. They also provide components to expose the microservices to the outside and mechanisms to select the nodes on which the microservices can be deployed.

The ideal scenario to deploy our TSN CP consists of a Kubernetes cluster comprising two groups of nodes. The first group is made up of machines specifically designed to operate at high frequency (for example, the AWS M5Zn machine group), while the second group is general purpose and is used to run all other microservices. Figure 4.9 describes that approach.

To achieve this in Kubernetes, we can use labels on the cluster nodes to differentiate them into type 1 (general purpose) and type 2 (specialized). Then we can use a tag named ”nodeSelector” as shown in the following code:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ilp-microservice
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ilp-microservice
  template:
    metadata:
      labels:
        app: ilp-microservice
    spec:
      nodeSelector:
        type: 2
      containers:
        - name: ilp-microservice
          image: ilp-microservice:latest
          ports:
            - containerPort: 8080
```

In the example above, we specify that the ILP pod should run on a node labeled type 2, which indicates that you should select a node from the pool of specialized machines with high operating frequencies.

In this way, using the tags and the ”nodeSelector” in Kubernetes, we can ensure that the microservices are deployed in the appropriate groups of nodes, allowing efficient use of resources and optimizing system performance.

Although we mentioned before that the ideal scenario would be to use Gurobi for our deployment, it is important to note that this Software is paid. However, Gurobi offers a free student license that can be obtained by request, specifying its use and the educational institution in which it will be used. It is important to note that due to containers' temporary nature, it is impossible to link a student account to a group of containers.

In cases like these, Gurobi offers an option that must be purchased by the institution and has an economic cost that was not considered in the project's development. However, considering the features in this document, it is possible to use Kubernetes and deploy the ILP microservice with any resolver mentioned in this work.

CHAPTER 5. CONCLUSIONS

In this project, we explored the usage of SDN and an MSA to implement a functional prototype of a TSN CP. We tested such prototype using real equipment for the CUC, the CNC, the end-stations with the Intel i210 NICs and the TSN switches of MTSN kit. It is important to highlight, that for the best of our knowledge, this is the first time that a SDN controller for TSN is designed and implemented by following a MSA.

Besides, we explore the characteristics of microservice architecture and its advantages compared to monolithic architecture. Specifically, we discover that it can be used to achieve scalability granularity, assigning computational resources to the specific microservices that consume the most resources. Moreover, the MSA also enables the option to have a different development and integration process between microservices. As we have seen so far, replacing the microservices with others that perform the same functionality but with a different approach is always possible. For instance, the VLAN generator module can be modified in the future by the YANG models for configuring VLANs once SoC-e has implemented it in the MTSN Kit.

We discover the necessity of prioritizing the frequency oscillation of the processor of the machines that are running our ILP implementation of the Raagard ILP model for the TAS. This is because solving a Linear mathematical problem is a single-thread task. It doesn't matter how many cores the processor has available; it only will use one single thread. That was the main reason for the M5Zn large AWS machine outperforming the local machine and not the available resources. However, in a real scenario, it will be possible to allocate specifically that microservice to a working device with the highest oscillation frequency to achieve better results thanks to MSA.

As allocating resources specifically to some microservices is possible, an MSA architecture can achieve better results regarding the number of Streams processed with the same amount of computational resources. Such affirmation is based on the fact that we can have granularity in the assignation of resources. At the same time, in a monolith, we can not allocate resources specifically to some parts of software.

Finally, this project allowed us to explore a plethora of technologies that will be useful in our professional careers, such as Docker, Kubernetes, Pyomo, GLPK, Gurobi, NETCONF, RESTCONF, well as many others. This project was developed in several programming languages, such as Python, Javascript, and Bash scripts.

5.1. Future Work

The modular design provided by the MSA allows the flexibility to replace individual components in future work as long as the new microservice receives the same inputs and delivers the same outputs as the old one. In this case, we suggest replacing the ILP Solver microservice with another that uses Artificial Intelligence (AI) algorithms to generate solutions to scheduling problems. This new microservice could be trained using a dataset generated from the results obtained by the current microservice.

Using AI algorithms instead of the ILP Solver can offer several significant benefits regarding the time required to solve scheduling problems. AI algorithms can learn patterns and automatically optimize the solution-generation process. By training the algorithm with a dataset based on the results of the current microservice, you can take advantage of existing knowledge and potentially improve the efficiency and accuracy of the generated solutions.

Furthermore, by replacing the ILP Solver microservice with one based on AI, different machine-learning approaches and techniques can be explored to address scheduling problems. This could open up new possibilities and allow the adaptation of the system to different scenarios and specific requirements.

5.2. Sustainability Considerations

SDN controllers based on microservices are crucial for efficient and environmentally friendly networking. Energy efficiency is prioritized through intelligent workload distribution, dynamic scaling, and efficient scheduling algorithms. Integrating renewable energy sources reduces reliance on fossil fuels. Scalability and flexibility are achieved through microservices architecture, allowing optimal resource allocation and preventing overprovisioning. Specifically, any virtualization technique optimizes the energy usage of the hardware since the number of on devices can be reduced to cope with lesser load. Additionally, lifecycle management practices ensure the use of recyclable materials and consider end-of-life disposal options. By optimizing energy consumption, leveraging renewable energy, enabling scalability, and incorporating lifecycle management, TSN SDN controllers promote sustainability in network infrastructure.

5.3. Ethical and Security Considerations

The use of Software-Defined Networking (SDN) in Time-Sensitive Networking (TSN) raises important ethical considerations. It is necessary to ensure the privacy and security of the data transmitted. Equity and justice in access to network resources must also be guaranteed, avoiding discriminatory practices. In addition, environmental implications must be addressed, and a sustainable approach to using SDN in TSN must be sought. In addition to these ethical concerns, security considerations must also be addressed to protect the network infrastructure, ensure data privacy and integrity, prevent unauthorized access, and comply with regulatory requirements. Robust security measures, access controls, encryption techniques, and incident response procedures should be implemented while selecting reliable vendors and regularly updating software and firmware. By addressing both ethical and security considerations, organizations can make responsible decisions to maximize the benefits and minimize the risks of SDN in TSN.

5.4. Acknowledgment

This research has been partially funded by the Agencia Estatal de Investigación of Ministerio de Ciencia e Innovación of Spain under projects PID2019-108713RB C51 MCIN/ AEI /10.13039/501100011033 and TSI-064100-2022-6, and by the European Regional Development Fund of the European Union in the framework of the ERDF Operational Program of Catalonia 2014-2020 under the Research Project 001-P-001643 Agrupació Emergent Looming Factory (IU16-011733 BAMPLA).

ACRONYMS

3GPP	3rd Generation Partnership Project
AI	Artificial Intelligence
API	Application Programming Interface
AWS	Amazon Web Services
CNC	Centralized Network Communications
CP	Control Plane
CUC	Centralized User Communications
EC2	Elastic Compute Cloud
GCLs	Generic Cell Libraries
IEEE	Institute of Electrical and Electronics Engineers
ILP	Integer Linear Programming
LXC	Linux Containers
LP	Linear Programming
MIP	Mixed Integer Programming
NETCONF	Network Configuration Protocol
NFV	Network Functions Virtualization
OLE	Object Linking and Embedding
OPC	OLE for Process Control
OPC-UA	OPC Unified Architecture
OSI	Open Systems Interconnection
OS	Operating System
PTP	Precision Time Protocol
SD	Standard Deviation
SDN	Software-Defined Networking
REST	Representational State Transfer
RESTCONF	REST-based Configuration Protocol
TAS	Time Aware Shaper
TSN	Time-Sensitive Networking
UNI	User Network Interface
VLAN	Virtual Local Area Network
VMs	Virtual Machines
VMM	Virtual Machine Monitor
YANG	Yet Another Next Generation

BIBLIOGRAPHY

[1] T. Gerhard, T. Kobzan, I. Blöcher, and M. Hendel, “Software-defined flow reservation: Configuring ieee 802.1 q time-sensitive networks by the use of software-defined networking,” in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2019. doi: https://doi.org/10.1109/ETFA.2019.8869040 pp. 216–223. 1, 5

[2] M. L. Raagaard and P. Pop, “Optimization algorithms for the scheduling of ieee 802.1 time-sensitive networking (tsn).” Tech. Univ. Denmark, Lyngby, 2017. 1, 6

[3] W. Quan, W. Fu, J. Yan, and Z. Sun, “Opentsn: an open-source project for time-sensitive networking system development,” in *CCF Transactions on Networking*, vol. 3, 2020. doi: https://doi.org/10.1007/s42045-020-00029-8 pp. 51–65. 1, 9

[4] T. Kobzan, I. Blöcher, M. Hendel, S. Althoff, A. Gerhard, S. Schriegel, and J. Jasperneite, “Configuration solution for tsn-based industrial networks utilizing sdn and opc ua,” in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1. IEEE, 2020. doi: https://doi.org/10.1109/ETFA46521.2020.9211897 pp. 1629–1636. 1, 9

[5] M.-T. Thi, S. B. H. Said, and M. Boc, “Sdn-based management solution for time synchronization in tsn networks,” in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1. IEEE, 2020. doi: https://doi.org/10.1109/ETFA46521.2020.9211923 pp. 361–368. 1

[6] D. Gallipeau and S. Kudrle, “Microservices: Building blocks to new workflows and virtualization,” in *SMPTE Motion Imaging Journal*, vol. 127, no. 4. SMPTE, 2018. doi: https://doi.org/10.5594/JMI.2018.2811599 pp. 21–31. 1

[7] F. Moradi, C. Flinta, A. Johnsson, and C. Meirosu, “Conmon: An automated container based network performance monitoring system,” in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2017. doi: https://doi.org/10.23919/INM.2017.7987264 pp. 54–62. 1

[8] C. Manso, R. Vilalta, R. Casellas, R. Martínez, and R. Muñoz, “Cloud-native sdn controller based on micro-services for transport networks,” in *2020 6th IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2020. doi: https://doi.org/10.1109/NetSoft48620.2020.9165377 pp. 365–367. 1, 9

[9] Q. P. Van, H. Tran-Quang, D. Verchere, P. Layec, H.-T. Thieu, and D. Zeghlache, “Demonstration of container-based microservices sdn control platform for open optical networks,” in *2019 Optical Fiber Communications Conference and Exhibition (OFC)*. IEEE, 2019, pp. 1–3. 1, 9

[10] J. Thönes, “Microservices,” in *IEEE software*, vol. 32, no. 1. IEEE, 2015. doi: https://doi.org/10.1109/MS.2015.11 pp. 116–116. 3

[11] A. Boubendir, E. Bertin, and N. Simoni, “A vnf-as-a-service design through micro-services disassembling the ims,” in *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*. IEEE, 2017. doi: https://doi.org/10.1109/ICIN.2017.7899412 pp. 203–210. 3

[12] S. Klock, J. M. E. Van Der Werf, J. P. Guelen, and S. Jansen, “Workload-based clustering of coherent feature sets in microservice architectures,” in *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2017. doi: https://doi.org/10.1109/ICSA.2017.38 pp. 11–20. 3

[13] J. Rufino, M. Alam, J. Ferreira, A. Rehman, and K. F. Tsang, “Orchestration of containerized microservices for iiot using docker,” in *2017 IEEE International Conference on Industrial Technology (ICIT)*. IEEE, 2017. doi: https://doi.org/10.1109/ICIT.2017.7915594 pp. 1532–1536. 3, 4

[14] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, “From monolithic to microservices: An experience report from the banking domain,” in *IEEE Software*, vol. 35, no. 3. IEEE, 2018. doi: https://doi.org/10.1109/MS.2018.2141026 pp. 50–55. 3

[15] S. Newman, *Building microservices*. O’Reilly Media, 2021. ISBN 978-1-4919-5035-7 3, 4

[16] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert, “Microservices,” in *IEEE Software*, vol. 35, no. 3. IEEE, 2018. doi: https://doi.org/10.1109/MS.2018.2141030 pp. 96–100. 3, 4

[17] F. Montesi and J. Weber, “Circuit breakers, discovery, and api gateways in microservices,” in *arXiv preprint arXiv*, 2016. doi: https://doi.org/10.48550/arXiv.1609.05830 3, 4

[18] F. Gutierrez, *Spring Boot Messaging*. Springer, 2017. ISBN 1484212258 3, 4

[19] K. Benzekki, A. El Fergougui, and A. Elbelrhiti Elalaoui, “Software-defined networking (sdn): a survey,” in *Security and communication networks*, vol. 9, no. 18. Wiley Online Library, 2016. doi: https://doi.org/10.1002/sec.1737 pp. 5803–5833. 4

[20] S. Rowshanrad, S. Namvarasl, V. Abdi, M. Hajizadeh, and M. Keshtgary, “A survey on sdn, the future of networking,” in *Journal of Advanced Computer Science & Technology*, vol. 3, no. 2. Science Publishing Corporation, 2014. doi: https://doi.org/10.1002/sec.1737 pp. 232–248. 4

[21] A. Lara, A. Kolasani, and B. Ramamurthy, “Network innovation using openflow: A survey,” in *IEEE communications surveys & tutorials*, vol. 16, no. 1. IEEE, 2013. doi: https://doi.org/10.1109/SURV.2013.081313.00105 pp. 493–512. 4

[22] B. Claise, J. Clarke, and J. Lindblad, *Network programmability with YANG: the structure of network automation with YANG, NETCONF, RESTCONF, and gNMI*. Addison-Wesley Professional, 2019. ISBN 0135180392 4

[23] M. Jethanandani, “Yang, netconf, restconf: What is this all about and how is it used for multi-layer networks,” in *Optical Fiber Communication Conference*. Optica Publishing Group, 2017, pp. W1D–1. 4

[24] L. Richardson and S. Ruby, “Restful web services.” O’Reilly Media, Inc., 2008. ISBN 9780596529260 4

[25] SOC-e, “MTSN-Kit: A comprehensive multiport TSN setup,” Retrieved from https://soc-e.com/mtsn-kit-a-comprehensive-multiport-tsn-setup/. 5

[26] “Soc-e mtsn kit,” Jun 2022. [Online]. Available: https://soc-e.com/mtsn-kit-a-comprehensive-multiport-tsn-setup/ 5

[27] L. Leonardi, L. L. Bello, and G. Patti, “Exploiting software-defined networking to improve runtime reconfigurability of tsn-based networks,” in *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2022. doi: 10.1109/ETFA52439.2022.9921723 pp. 1–4. 9

[28] M. Schleipen, S.-S. Gilani, T. Bischoff, and J. Pfrommer, “Opc ua industrie 4.0-enabling technology with high diversity and variability,” in *Procedia Cirp*, vol. 57. Elsevier, 2016. doi: https://doi.org/10.1016/j.procir.2016.11.055 pp. 315–320. 11

[29] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, “Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors,” in *Proceedings of the 2Nd ACM SIGOPS/EuroSys european conference on computer systems 2007*, 2007. doi: https://doi.org/10.1145/1272996.1273025 pp. 275–287. 11

[30] “Pika library,” Jun 2022. [Online]. Available: https://pika.readthedocs.io/en/stable/ 15

[31] “Ieee standard for local and metropolitan area networks–bridges and bridged networks – amendment 31: Stream reservation protocol (srp) enhancements and performance improvements,” Jun 2022. [Online]. Available: https://standards.ieee.org/ieee/802.1Qcc/5784/ 15

[32] “Jetconf,” June 2022. [Online]. Available: https://jetconf.readthedocs.io/en/latest/ 15

[33] P. Congdon, “Link layer discovery protocol and mib,” in *V1. 0 May*, vol. 20, no. 2002, 2002, pp. 1–20. 16

[34] M. Zadka, “Paramiko,” in *DevOps in Python*. Springer, 2019, pp. 111–119. 16

[35] M. L. Raagaard and P. Pop, “Optimization algorithms for the scheduling of ieee 802.1 time-sensitive networking (tsn),” in *Tech. Univ. Denmark, Lyngby, Denmark, Tech. Rep*, 2017. 18

[36] “Conda,” Jun 2022. [Online]. Available: https://docs.conda.io/en/latest/ 22

[37] “Ampl,” Jun 2022. [Online]. Available: https://ampl.com/ 23

[38] “Pico,” Jun 2022. [Online]. Available: https://www.swmath.org/software/2252 23

[39] “Cbc,” Jun 2022. [Online]. Available: https://www.coin-or.org/Cbc/ 23

[40] “Glpk,” Jun 2022. [Online]. Available: https://www.gnu.org/software/glpk/ 23

[41] “Gurobi,” Jun 2022. [Online]. Available: https://www.gurobi.com/ 23

[42] YangModels, “IEEE 802.1Q Bridge Yang Model,” GitHub repository, July 2023, Accessed on July 2, 2023. [Online]. Available: https://github.com/YangModels/yang/blob/main/standard/ieee/published/802.1/ieee802-dot1q-bridge.yang 25

[43] J. Cros, “Development of an sdn control plane for time-sensitive networking (tsn) end-points,” https://upcommons.upc.edu/handle/2117/348819, 2021. 33

[44] “T3 aws instances,” July 2023. [Online]. Available: https://aws.amazon.com/ec2/instance-types/t3/ 35

[45] “M5 aws instances,” July 2023. [Online]. Available: https://aws.amazon.com/ec2/instance-types/m5/ 35

[46] K. Chakrabarty, “Test scheduling for core-based systems using mixed-integer linear programming,” in *IEEE Transactions on Computer-aided design of integrated circuits and systems*, vol. 19, no. 10. IEEE, 2000. doi: https://doi.org/10.1109/43.875306 pp. 1163–1174. 35

[47] B. Meindl and M. Templ, “Analysis of commercial and free and open source solvers for linear optimization problems,” in *Eurostat and Statistics Netherlands within the project ESSnet on common tools and harmonised methodology for SDC in the ESS*, vol. 20, 2012. 36

APPENDICES

APPENDIX A. PYANG STRUCTURE OF CUSTOM YANG MODEL

This section includes the yang tree used for implementing the communication between the CUC and the CNC. This tree is a modification of the original module *ieee802-dot1q-tsn-types* provided by the IEEE. To build this tree we used Pyang an easy to install python library included in *pip*. The full code is in the Github repository.

Listing A.1: ieee802-dot1q-tsn-types-upc-version@2018-02-15.yang

```
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-reuirements
     |  |  |  |  +--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)
|     +---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
|     |                 +--ro time-aware-offset?        uint32
```

APPENDIX B. HOW TO EXECUTE THE CODE

This annex includes the necessary commands and recommendations to execute the code inside of docker-compose.

B.1. Docker and Docker compose Installations

First it is necessary to install Docker and Docker-compose. Depending on the host operative system you can follow this series of steps. All of these commands have to be executed within the Command Line Interface.

B.1.1. Installation over Linux

First make sure that your system repositories are up to date:

```
sudo apt update
sudo apt upgrade
```

Then is necessary to have the prerequisites (*i.e.,* curl apt-transport-https ca-certificates software-properties-common) installed

```
sudo apt-get install curl \
apt-transport-https \
ca-certificates \
software-properties-common \
```

Go to add the necessary Docker repositories, this is to have the option of downloading directly using the linux package manager. The first to add is GPG

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
sudo apt-key add -
```

Then the Docker repository:

```
sudo add-apt-repository "deb [arch=amd64]\
https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) stable"
```

Then update the repositories again:

```
sudo apt update
```

It is necessary to do it from the correct repository, then we will use this command to check if it is the case:

```
apt-cache policy docker-ce
```

The output should be as follows:

```
docker-ce:
    Installed: (none)
    Candidate: 16.04.1~ce~4-0~ubuntu
    Version table:
         16.04.1~ce~4-0~ubuntu 500
               500
               https://download.docker.com/linux/
               ubuntubionic/stableamd64packages
```

Install Docker with apt packet manager:

```
sudo apt install docker-ce
```

Check that everything is configured and docker is running with the following commands:

```
sudo systemctl status docker
docker-version
```

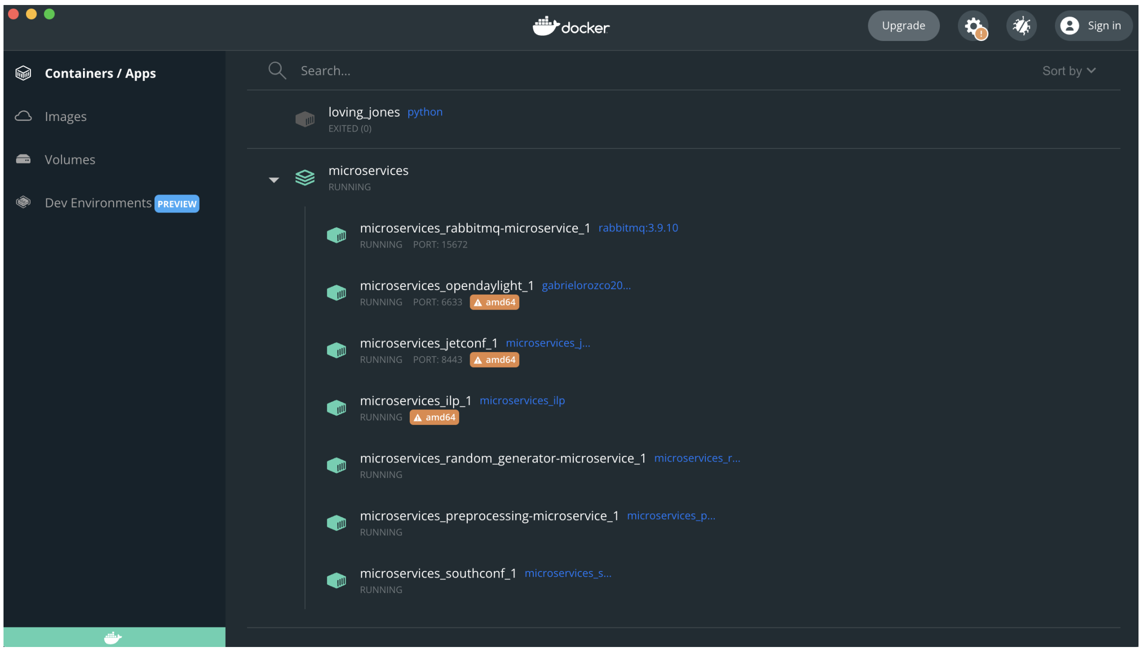
With those commands Docker should be up and running for using within a local machine.

B.1.2. Installation over OSx and Windows

For installing Docker over Mac and Windows it is better idea to use Docker Desktop the installation is straight forward from the website of Docker:

https://docs.docker.com/desktop/

Once installed you will have a GUI similar as the presented in Figure B.1. Moreover, all of the commands from the *cli* can be used as in the Linux distribution.



```
loving_jones  python
EXITED (0)

microservices
RUNNING
    microservices_rabbitmq-microservice_1  rabbitmq:3.9.10
    RUNNING   PORT: 15672
    microservices_opendaylight_1  gabrielorozco20...
    RUNNING   PORT: 6633  amd64
    microservices_jetconf_1  microservices_j...
    RUNNING   PORT: 8443  amd64
    microservices_ilp_1  microservices_ilp
    RUNNING  amd64
    microservices_random_generator-microservice_1  microservices_r...
    RUNNING
    microservices_preprocessing-microservice_1  microservices_p...
    RUNNING
    microservices_southconf_1  microservices_s...
    RUNNING
```

Figure B.1: Docker Desktop Graphical interface

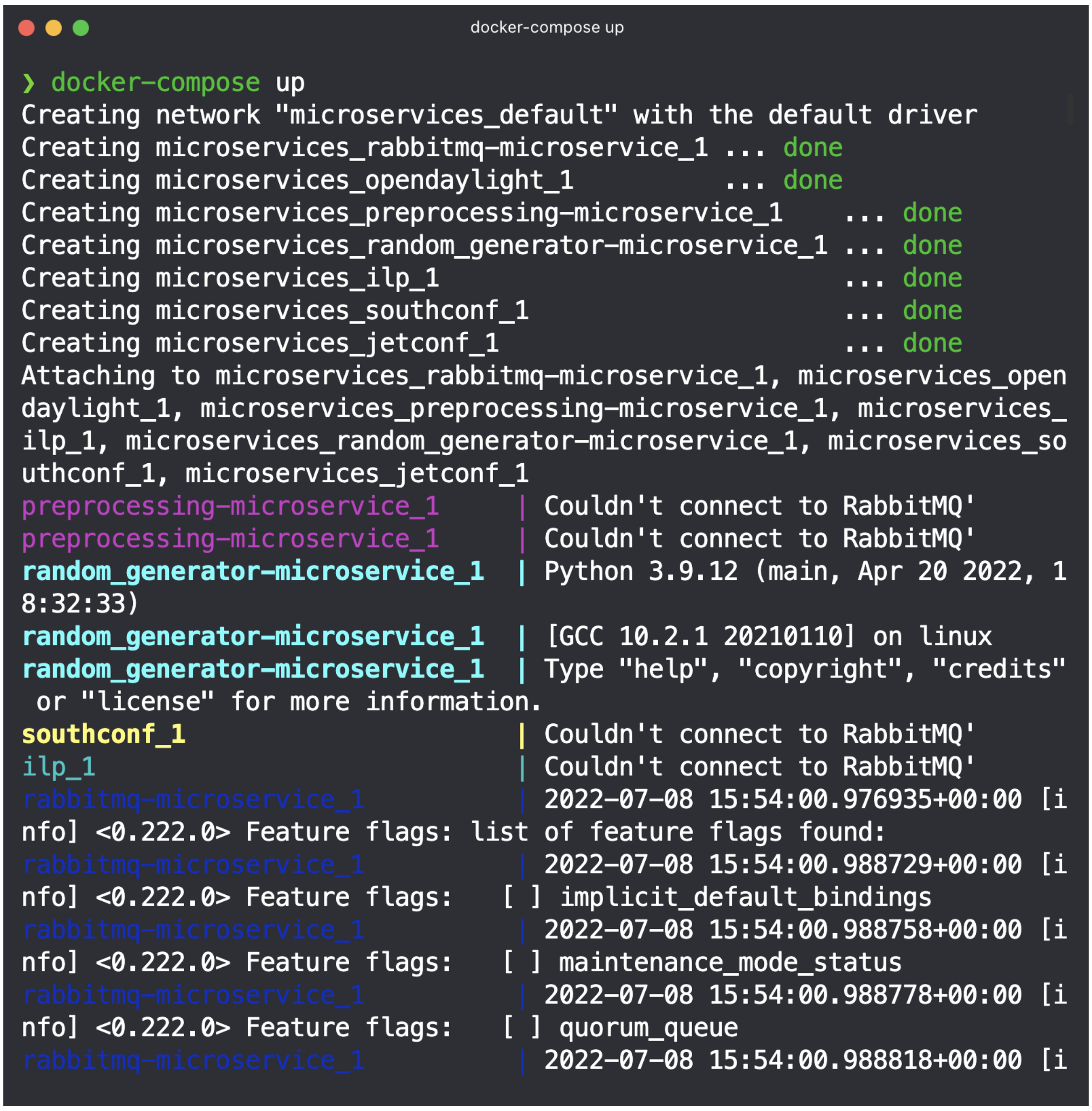
B.1.3. How to execute the code under Docker compose

To execute the code we can use a set of commands necessary from docker.

In the file structure of the repository we have to go move to the proper folder in our repository and execute appropriated docker command. Supposing the code is in the home directory it should be as follows:

```
cd TSN–CNC–CUC–UPC/CNC/Microservices
docker–compose up
```

This command will go to the *docker-compose.yml* file and look for the configuration of the docker infrastructure to deploy, the output of the command should be as depicted in Figure B.2.



```
❯ docker-compose up
Creating network "microservices_default" with the default driver
Creating microservices_rabbitmq-microservice_1 ... done
Creating microservices_opendaylight_1          ... done
Creating microservices_preprocessing-microservice_1    ... done
Creating microservices_random_generator-microservice_1 ... done
Creating microservices_ilp_1                           ... done
Creating microservices_southconf_1                     ... done
Creating microservices_jetconf_1                       ... done
Attaching to microservices_rabbitmq-microservice_1, microservices_open
daylight_1, microservices_preprocessing-microservice_1, microservices_
ilp_1, microservices_random_generator-microservice_1, microservices_so
uthconf_1, microservices_jetconf_1
preprocessing-microservice_1      | Couldn't connect to RabbitMQ'
preprocessing-microservice_1      | Couldn't connect to RabbitMQ'
random_generator-microservice_1   | Python 3.9.12 (main, Apr 20 2022, 1
8:32:33)
random_generator-microservice_1   | [GCC 10.2.1 20210110] on linux
random_generator-microservice_1   | Type "help", "copyright", "credits"
 or "license" for more information.
southconf_1                       | Couldn't connect to RabbitMQ'
ilp_1                             | Couldn't connect to RabbitMQ'
rabbitmq-microservice_1           | 2022-07-08 15:54:00.976935+00:00 [i
nfo] <0.222.0> Feature flags: list of feature flags found:
rabbitmq-microservice_1           | 2022-07-08 15:54:00.988729+00:00 [i
nfo] <0.222.0> Feature flags:   [ ] implicit_default_bindings
rabbitmq-microservice_1           | 2022-07-08 15:54:00.988758+00:00 [i
nfo] <0.222.0> Feature flags:   [ ] maintenance_mode_status
rabbitmq-microservice_1           | 2022-07-08 15:54:00.988778+00:00 [i
nfo] <0.222.0> Feature flags:   [ ] quorum_queue
rabbitmq-microservice_1           | 2022-07-08 15:54:00.988818+00:00 [i
```

Figure B.2: Docker compose up command output

The file *docker-compose.yml* contains the each of the definitions for the microservices. For instance, Code Snipped B.1 shows the part of the code that corresponds to the preprocessing microservice. the *dockerfile* shows the path for the Dockerfile (i.e., a file that defines the steps followed to deploy the container), *volumes* generates a volume mirror between a path within the host machine and the Docker container, this is extremely useful for deploying and testing new code within the container. Additionally, *depends_on* indicates that none of the microservices should be executed if the *rabbitmq-microservice* is not ready. The rest of the file is available at the code repository.

Listing B.1: Code Sniped of Docker-compose preprocessing Microservice

```
preprocessing-microservice:
  build:
    context: .
    dockerfile: Preprocessing_microservice/Dockerfile
  volumes:
    - ./Preprocessing_microservice:/preprocessing
  stdin_open: true
  tty: true
  depends_on:
    - rabbitmq-microservice
```

We can check containers are up and running with the *fig:DockerComposeUP* command from the cli. The Image B.3 shows the proper output. This command will show the containers running (related to the *docker-compose.yml* file) the command they are executing, the state, and the ports they are listening.

The open ports are the way of communicating within microservices and for the out-site world to access to the microservices. For instance, the container that corresponds to the Jetconf Microservice has the port 8443 open for receiving https traffic from the outside world. This port is necessary since Jetconf exposes the RESTCONF API in this port for the UNI interface with the CUC.

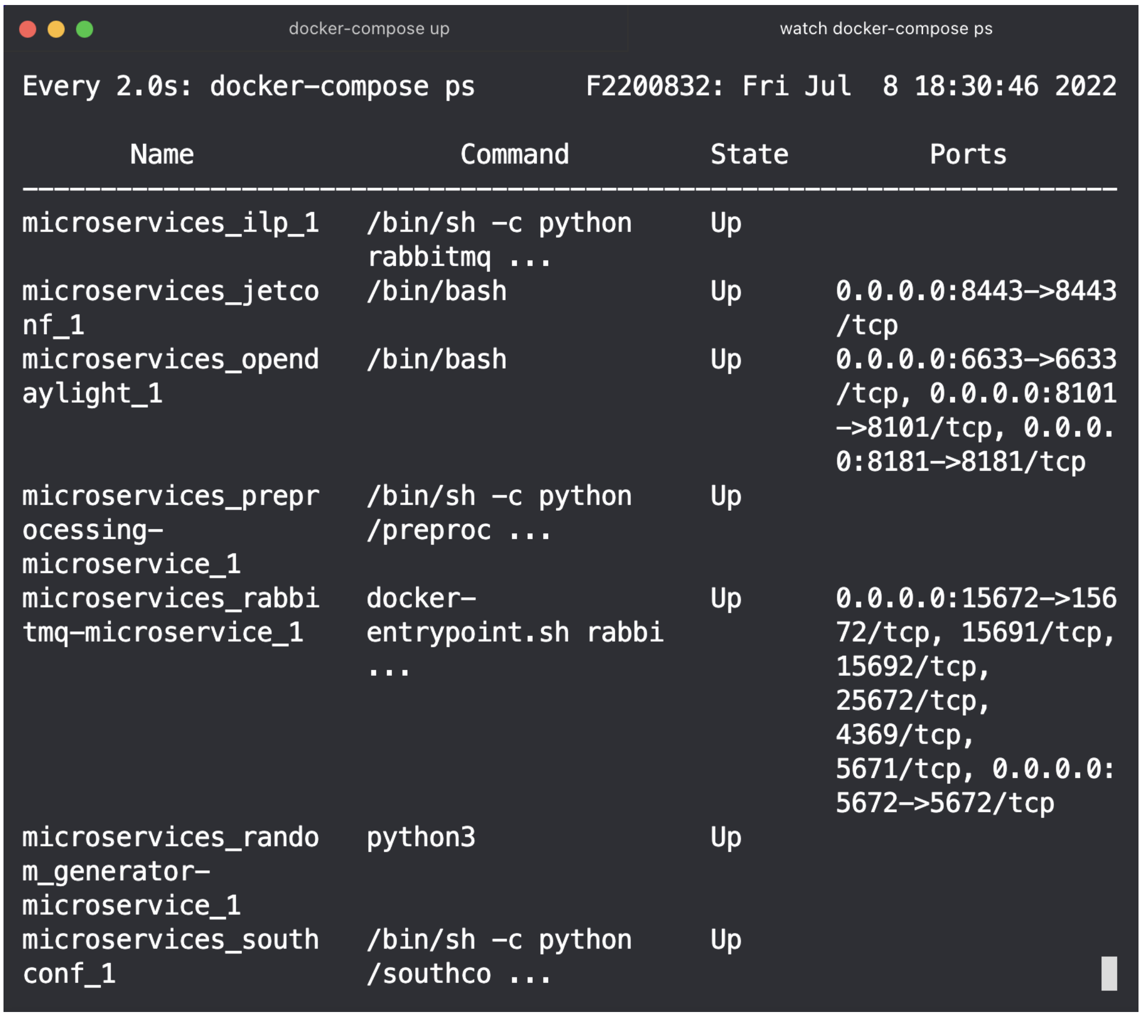
Additionally, it is possible to get inside the Command Line Interface of the docker container if something there needs to be done. For that we can use the command with the following structure:

```
docker-compose exec <Name of the microservice> bash
```

Figure B.4 depicts the usage of the command for accessing to two of the docker containers of the Jetconf and ILP Calculator Microservices.

Finally, to bring down the docker containers and turn-off the system is necessary to use the *docker-compose down* command.

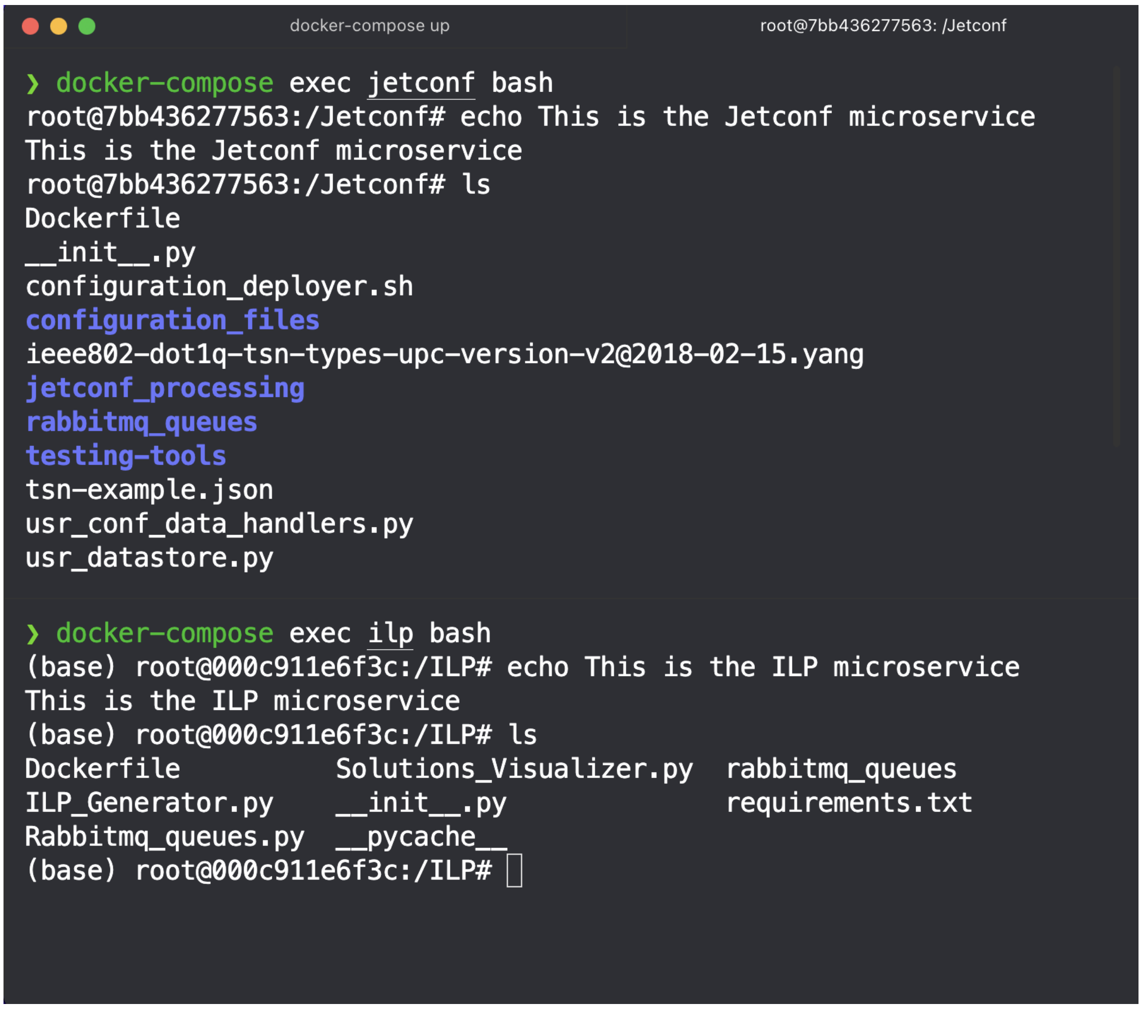
Even thought there are far more docker commands used for implementing μTSN-CP with the provided information in this Annex is enough to the system to work.



```
Every 2.0s: docker-compose ps       F2200832: Fri Jul  8 18:30:46 2022

       Name                    Command            State         Ports
----------------------------------------------------------------------
microservices_ilp_1   /bin/sh -c python      Up
                      rabbitmq ...
microservices_jetco   /bin/bash              Up      0.0.0.0:8443->8443
nf_1                                                 /tcp
microservices_opend   /bin/bash              Up      0.0.0.0:6633->6633
aylight_1                                            /tcp, 0.0.0.0:8101
                                                     ->8101/tcp, 0.0.0.
                                                     0:8181->8181/tcp
microservices_prepr   /bin/sh -c python      Up
ocessing-             /preproc ...
microservice_1
microservices_rabbi   docker-                Up      0.0.0.0:15672->156
tmq-microservice_1    entrypoint.sh rabbi            72/tcp, 15691/tcp,
                      ...                            15692/tcp,
                                                     25672/tcp,
                                                     4369/tcp,
                                                     5671/tcp, 0.0.0.0:
                                                     5672->5672/tcp
microservices_rando   python3                Up
m_generator-
microservice_1
microservices_south   /bin/sh -c python      Up
conf_1                /southco ...
```

Figure B.3: Docker compose ps command output



```
❯ docker-compose exec jetconf bash
root@7bb436277563:/Jetconf# echo This is the Jetconf microservice
This is the Jetconf microservice
root@7bb436277563:/Jetconf# ls
Dockerfile
__init__.py
configuration_deployer.sh
configuration_files
ieee802-dot1q-tsn-types-upc-version-v2@2018-02-15.yang
jetconf_processing
rabbitmq_queues
testing-tools
tsn-example.json
usr_conf_data_handlers.py
usr_datastore.py

❯ docker-compose exec ilp bash
(base) root@000c911e6f3c:/ILP# echo This is the ILP microservice
This is the ILP microservice
(base) root@000c911e6f3c:/ILP# ls
Dockerfile          Solutions_Visualizer.py  rabbitmq_queues
ILP_Generator.py    __init__.py              requirements.txt
Rabbitmq_queues.py  __pycache__
(base) root@000c911e6f3c:/ILP# 
```

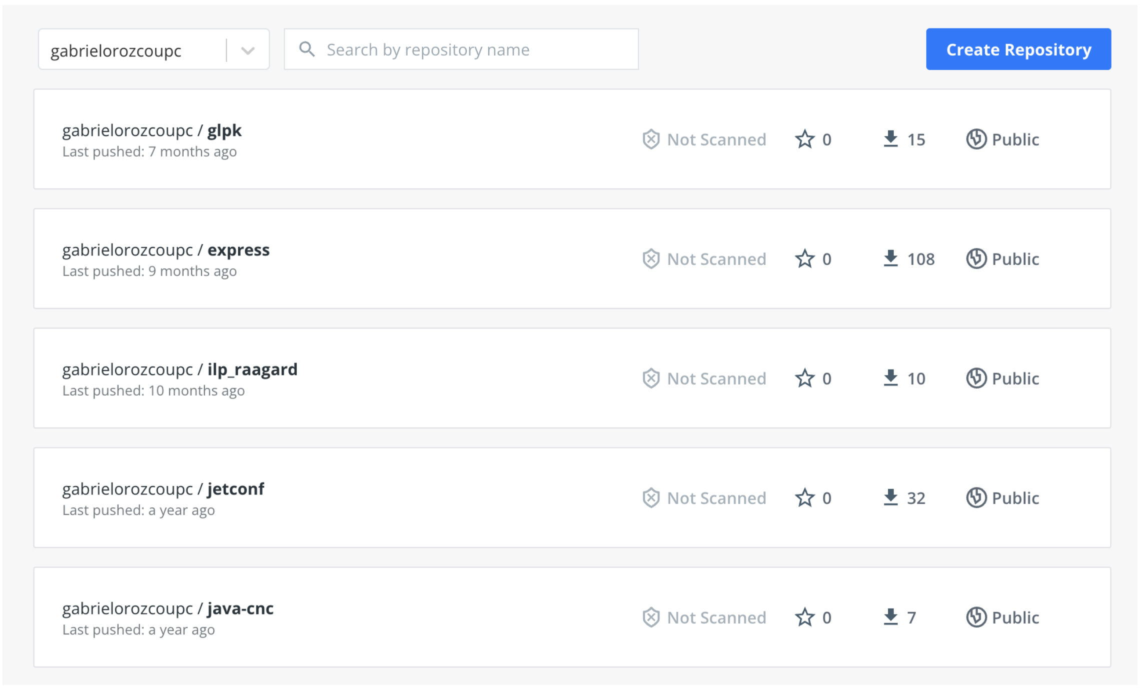
Figure B.4: Accessing two container with docker-compose exec command

APPENDIX C. ABOUT DOCKER IMAGES

The work is not only reflected in the code repository in GitHub, we also have an essential component that has a particular repository the Docker Images. Each one of this software pieces are the building block for the architecture. Even thought, by themselves the Docker image does not contain any code, each microservice needs a particular environment in which to run. For instance, the ILP microservice needs the Pyomo library and GLPK to be installed on the system to run correctly. Those images contain the majority are in a repository in order to be downloaded every-time the system needs runs and they are selected to be used in the *Dockerfile* or in the *docker-compose.yml*. The repository can be in many places such as Docker Container Registry, Amazon Container Registry or Dockerhub. In our case we decided to use directly Dockerhub as the Image repository.

Figure C.1 depicts some of the created images. Such images can be accessed in the following repository:

https://hub.docker.com/repository/docker/gabrielorozcoupc.



gabrielorozcoupc / **glpk** Last pushed: 7 months ago	Not Scanned	0	15	Public
gabrielorozcoupc / **express** Last pushed: 9 months ago	Not Scanned	0	108	Public
gabrielorozcoupc / **ilp_raagard** Last pushed: 10 months ago	Not Scanned	0	10	Public
gabrielorozcoupc / **jetconf** Last pushed: a year ago	Not Scanned	0	32	Public
gabrielorozcoupc / **java-cnc** Last pushed: a year ago	Not Scanned	0	7	Public

Figure C.1: Some of the images available in the Dockerhub repository