

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE

*DEPARTMENT OF ELECTRICAL, ELECTRONIC AND
INFORMATION ENGINEERING "GUGLIELMO MARCONI" (DEI)*

MASTER DEGREE IN AUTOMATION ENGINEERING
8891

GRADUATION THESIS

in

MECHATRONICS SYSTEMS MODELING AND CONTROL M

**TIME SENSITIVE NETWORKS:
ANALYSIS, TESTING, SCHEDULING
AND SIMULATION**

CANDIDATE:
Luca Cedrini

SUPERVISOR:
eminent prof. **Alessandro Macchelli**

CO-SUPERVISORS:
eng. **Stefano Gualmini**
eng. **Marco de Vietro**

Academic year
2019/2020

Session
IV

Ringraziamenti

Iniziamo dalla fine. Desidero ringraziare i miei correlatori, Stefano Gualmini e Marco De Vietro, per avermi guidato e supportato nel mio lavoro durante il tirocinio. Ringrazio anche il mio relatore, il professor Alessandro Macchelli, che si è sempre mostrato gentile e disponibile.

Ulteriori ringraziamenti vanno ai miei compagni di avventura di questi anni universitari, che sono diventati a tutti gli effetti miei amici. Da quelli magistrali, Andrea, Lori, Giova, Eusebio, Ros, Martina e Baruz, a quelli con cui ho condiviso solo la triennale, Steve, Mula, Fre e Ilaria. Sono sicuro che, anche se prenderemo strade diverse, riusciremo sempre a ritrovarci.

Degli anni del liceo voglio ricordare in particolare Nick e Giulia. Ringrazio anche Sici e Benuz, con la speranza di poter presto festeggiare come si deve da Caruso.

Ringrazio poi gli amici di una vita, con cui ormai sono di famiglia sin da quando eravamo bambini: Balbo e Mattia. E non c'è davvero bisogno di aggiungere altro.

Finiamo dall'inizio. Ringrazio la mia sempre presente famiglia per avermi cresciuto ed educato. Ringrazio i miei nonni, Alfredo ed Elena, Enzo e Marisa, ringrazio i miei zii, Andrea e Morena, e ringrazio i miei genitori e mia sorella, Marco, Lucia ed Elena.

È senz'altro grazie ad ognuno di voi che sono riuscito a raggiungere questo traguardo.

Luca

Abstract

The industrial automation world is an extremely complex framework, where state-of-the-art cutting edge technologies are continuously being replaced in order to achieve the best possible performances.

The criterion guiding this change has always been the productivity. This term has, however, a broad meaning and there are many ways to improve the productivity, that go beyond the simplistic products/min ratio.

One concept that has been increasingly emerging in the last years is the idea of interoperability: a flexible environment, where products of different and diverse vendors can be easily integrated together, would increase the productivity by simplifying the design and installation of any automatic system.

Connected to this concept of interoperability is the Industrial Internet of Things (IIoT), which is one of the main sources of the industrial innovation at the moment: the idea of a huge network connecting every computer, sensor or generic device so as to allow seamless data exchange, status updates and information passing.

It is in this framework that Time Sensitive Networks are placed: it is a new, work-in-progress set of communication standards whose goal is to provide a common infrastructure where all kinds of important data for an industrial automation environment, namely deterministic and non deterministic data, can flow.

This work aims to be an initial step towards the actual implementation of the above-mentioned technology. The focus will therefore be not only on the theoretical aspects, but also on a set of practical tests that have been carried out in order to evaluate the performances, the required hardware and software features, advantages and drawbacks of such an application.

Keywords: Time Sensitive Networks (TSN), Industrial Internet of Things, Industry 4.0, industrial communication

Abstract

Il mondo dell'automazione industriale è un ambiente estremamente complesso, dove nuove e innovative tecnologie vengono continuamente impiegate al fine di ottenere le migliori prestazioni possibili.

Il criterio guida di questo cambiamento è sempre stato la produttività. Questo termine ha tuttavia un largo significato, e ci sono molti modi di incrementare la produttività, che vanno oltre il semplicistico rapporto prodotti/min.

Un concetto emergente negli ultimi anni è quello di interoperabilità: creare un ambiente flessibile, caratterizzato da una facile integrazione di dispositivi di diversi produttori, aumenterebbe la produttività semplificando il lavoro di progetto e installazione di qualsiasi sistema automatico.

Collegato al concetto di interoperabilità c'è l'Industrial Internet of Things (IIoT), che è una delle principali fonti di innovazione industriale al momento: l'idea di una grande rete che mette in comunicazione calcolatori, sensori e ogni generico dispositivo per permettere un continuo scambio di dati, aggiornamenti e passaggio di informazioni.

È in questo contesto che si collocano le reti TSN: si tratta di un nuovo (ancora in via di sviluppo) gruppo di standard di comunicazione che mirano a fornire un'infrastruttura comune dove tutte le tipologie di dati di interesse industriale, soprattutto quelli deterministici e non deterministici, possono circolare.

Questo lavoro di tesi si propone di essere un passo iniziale verso la concreta implementazione della sopracitata tecnologia. Pertanto l'attenzione sarà concentrata non solo sugli aspetti teorici, ma anche su una serie di test pratici che sono stati svolti per valutare le prestazioni, l'hardware e il software richiesti, i vantaggi e gli svantaggi di una tale applicazione.

Parole chiave: Time Sensitive Networks (TSN), Industrial Internet of Things, Industria 4.0, comunicazione industriale

Contents

1	Introduction	13
2	Background	23
2.1	The network stack and the Internet	23
2.1.1	The internet's jargon	23
2.1.2	The network's core	25
2.1.3	A network's parameter: delay	27
2.1.4	Layered architecture	31
2.1.5	Link to TSN	36
2.2	New trends in the automation world	36
2.2.1	Introduction to Industry 4.0	36
2.2.2	Core idea of Industry 4.0	37
2.2.3	OPC UA	41
2.3	Time Sensitive Networks	44
2.3.1	Time synchronization	46
2.3.2	Time aware shaper	48
2.3.3	System's configuration	53
2.3.4	Credit-based shaping	55
2.4	The scheduling problem	58
2.4.1	ILP-Based Joint Routing and Scheduling	60
2.4.2	No-wait packet scheduling	64
3	Tests	68
3.1	Introduction to the tests	68
3.2	Setup	70
3.3	Precision Time Protocol for Linux	74
3.4	Time aware shaper	80
3.4.1	Linux's output interface	81
3.4.2	Relevant qdiscs	83
3.4.3	Intel's code	88
3.4.4	TAPRIO	89
3.4.5	MQPRIO	96
3.4.6	Hardware limit	99

3.5	Credit-based shaper	102
3.6	Troubleshooting	104
3.6.1	Kernel's version	105
3.6.2	Topology of the test	107
3.6.3	Crash of VNC	110
3.6.4	Intel's code	111
3.7	Summary	113
3.7.1	Requirements	114
4	Simulations	116
4.1	Simulation libraries on MATLAB/Simulink	118
4.1.1	Routing	120
4.1.2	Packets	123
4.1.3	Input port	125
4.1.4	Output port	126
4.1.5	Switch	129
4.1.6	Host	130
4.1.7	Network	131
4.1.8	Validation	132
4.2	Implementation of No-wait packet scheduling on MATLAB	140
4.2.1	TimeTabling	141
4.2.2	NWPS_Heuristic	143
4.2.3	Schedules	144
4.2.4	Multi-objective optimization	146
4.2.5	Performance	147
4.3	Test simulations	151
4.3.1	Simple network	151
4.3.2	Generic network	156
4.3.3	Final considerations	159
5	Conclusions	163
5.1	Summary	164
5.1.1	Analysis	164
5.1.2	Testing	165
5.1.3	Scheduling	166
5.1.4	Simulation	167
5.2	Future developements	168
5.2.1	Long-term objectives	168
5.2.2	Short-term objectives	169
	Bibliography	170

List of Figures

1.1	TSN testbed at the Hannover Fair, 2018	18
1.2	machine-to-machine communication	19
2.1	Structure of a packet	24
2.2	An example of a network	27
2.3	Comparison between two protocol stacks	32
2.4	An example of the active layers in a transmission	35
2.5	Characteristics of the four industrial revolutions	37
2.6	Smart factory	38
2.7	Automation pyramid	40
2.8	IT support	41
2.9	OPC UA client	42
2.10	OPC UA meta model	43
2.11	Main TSN standards	44
2.12	Key elements of TSN	45
2.13	Principle scheme of PTP	47
2.14	Time Aware Shaper	49
2.15	The use of a Guard Band	51
2.16	System's configuration	54
2.17	The computed schedule	55
2.18	Example of CBS for one queue	57
3.1	Setup for the tests	69
3.2	Setup for the tests	70
3.3	pc-a and its kernel version	72
3.4	Working frequencies of the cores	73
3.5	A typical graph of Wireshark	74
3.6	Master's ptp4l	77
3.7	Master's phc2sys	77
3.8	Slave's ptp4l	78
3.9	Slave's phc2sys	78
3.10	Test in the master	80
3.11	Test in the slave	80

3.12	The time aware shaper	81
3.13	Linux's output interface	82
3.14	Default queueing discipline	83
3.15	A piece of code from sample-app-taprio.c	89
3.16	A piece of code from sample-app-taprio.c	90
3.17	Output of the tc script	93
3.18	Histogram of the interpacket latency	93
3.19	Zoom of Fig.3.18	94
3.20	Wireshark's graph	94
3.21	First test with MQPRIO	96
3.22	Second test with MQPRIO	97
3.23	Wireshark of the second test with MQPRIO	97
3.24	Test with PFIFO_FAST	99
3.25	Second test with TAPRIO	100
3.26	Zoom of Fig. 3.25	101
3.27	Wireshark of the second test with TAPRIO	101
3.28	Credit based shaper	103
3.29	Wireshark of the first test with CBS	103
3.30	Wireshark of the second test with CBS	104
3.31	Configuration file for kernel 5.8	106
3.32	Configuration file for kernel 5.9	106
3.33	Currently loaded kernel modules	107
3.34	Output of sample-app-taprio.c	108
3.35	Test with the switch	109
4.1	Comparison between different simulation methods	116
4.2	The SimEvents library	119
4.3	Configuration of a generation block	123
4.4	Configuration of a generation block	124
4.5	Input port	126
4.6	Output port	127
4.7	Schedule	127
4.8	A three ports switch	129
4.9	Collected data	130
4.10	Generic network	131
4.11	Generic network	133
4.12	Measured interpacket latency	134
4.13	Zoom of the interpacket latency	134
4.14	Measured TSN latencies	135
4.15	Measured best effort latencies	135
4.16	Zoom of the best effort latencies	136
4.17	Experimental setup	137
4.18	Latencies for TSN packets	138
4.19	Latencies for BE packets	139

4.20	Outputs of the function: 1	147
4.21	Outputs of the function: 2	148
4.22	Outputs of NWPS_Heuristic_mo	149
4.23	Latencies for TSN packets	151
4.24	Simple network	152
4.25	Taskset	152
4.26	Results of the simulation	153
4.27	Schedule and generation	154
4.28	Generic network	156
4.29	Results about the generic network	157
4.30	Average waiting times	157
4.31	Results about the generic network with 200 tasks	158

Chapter 1

Introduction

Overview on the main communication techniques in industrial automation

Automatic machines are the core of any industrial plant: they are designed in such a way to be able to work autonomously for as long as possible. The idea is to let the machine work without any need of human intervention, which definitely increases productivity, lowers the costs for the staff and the risks associated with any industrial environment.

This means, though, that every action, every step of the productive process must be integrated together, creating a system that pertains several different domains: mechanical, electrical, electronic, pneumatic and so on. Also, devices belonging to the same automatic machine, that handles the same productive process, may be placed dozens of meters apart and still need to communicate or be coordinated with the highest of the precisions, in order to maximize the efficiency.

Flexibility and interoperability also play an important role in modern automation. Market's needs often require to be able to quickly change the configuration of the machine, the format of the product or also to build even more complex systems out of several automatic machines. It is fundamental, therefore, to start from the design to take into account all the features and the future challenges that the system is going to face, and provide it with the necessary technology to survive the test of time and make it a reliable and durable product.

As we said, industrial plants are made of numerous different devices that work in the same environment and generally tend to modify it. Automatic machines, in particular, are characterized by a high level of planning and design of the tasks for every device by which they are composed, which implies an extreme level of synchronization.

Clearly this couldn't be done if the devices were all mute and unable to communicate one with the other. There has to exist at least one channel of

communication through which they can pass information, data, status and many other working parameters.

The trivial example is represented by the system bus, which is a link connecting all the devices of an automatic machine, starting from the main drives up to the last sensor, to the main control logic, i.e. the PLC. In this particular -yet very close to our framework- example, the communication channel works as a collector of all the operating data from the environment (sensors measurements, control inputs from the HMI, other signals of various kinds) and passes them to the computing unit so that it can compute the control signals for the next cycle; then those control signals flow through the same link to the different destinations in the plant.

In an industrial environment there exists another whole level of communication, that is called Enterprise Resource Planning. ERP is a software system that utilizes a centralized database that contains all the necessary data in one location. The data are generated from the transactions produced by every process enabled by the ERP system. This database is shared across multiple departments to support multiple functions. The shared information allows for easy metric reporting and faster implementations of tasks, such as orders, marketing and bookkeeping. In essence, an ERP system automates processes across departments and helps to streamline common functions such as inventory management, order fulfillment, and order status.

These two systems coexist in most of the productive plants and basically define two different sets of information models, source and destination devices; but they are both needed in order for the whole plant to work properly and to increase the efficiency. As a result, a massive load of information has to be sent and delivered, every second, throughout the whole factory.

One major problem of this approach is that, potentially, a single device may have to communicate to a large set of other devices, possibly not of the same type and even produced by different vendors. Therefore it is required that all the devices act following some sorts of routine or protocol, both hardware and software, when interacting with another device of the system. This in order to ensure that every datum that is sent into the network can be decrypted and understood by everyone who is interested in it.

That's not an easy task because, as we said, the automation industry is a mix of all kinds of technologies and devices; however several entities and organizations have already begun to develop standards and protocols useful not only in the automation field. Among the most important ones there are the International Organization for Standardization (ISO), the Institute of Electrical and Electronic Engineers (IEEE) and the OPC Foundation. Their work deals with different aspects of the industrial communication that we will study more in detail in the next chapters.

Going back to the examples of ERP and the system bus, it is important to highlight that there is a fundamental difference between these two communication models. It consists in the kind of data and in the guarantees

that such data have or do not need to have.

As we discussed before, an automatic machine relies on the strong synchronization between its elements. To do that, the signals to and from the main control unit must be delivered as fast as possible. Mere speed, though, is not the most important feature required: determinism is. We are talking about a deterministic message when the message has *guaranteed* delivery times, with bounded and limited jitter. This means that we are able to compute and guarantee, a priori, that every instance of the message will not break the deadlines that have been assigned to it in terms of latency and intermessage latency (i.e. the time that passes between the send and the delivery and the time that passes between two consecutive deliveries). Determinism does not imply speed nor efficiency, but rather that even in the worst possible case we are sure about the limits of some functional parameters; it is achieved by reducing the randomness that naturally characterizes the physical processes.

On the other hand, the kind of data that an ERP system deals with does not have any needed guarantees about the delivery times. If the information about the productivity of a certain machine, that has to be stored in order to keep track of the performances of the plant, gets delivered to the server with a delay of some milliseconds due to network's congestion, it is not a big deal. Instead, if the photocell sensor fails to send in time to the PLC the information about the product passing in front of it, the actions linked to this event do not get triggered, the machine loses its synchronization and as a consequence the product has to be discarded, since it hasn't been treated properly.

The previous are two simple examples that aim at passing the idea that some messages, and in particular their temporal characteristics, are necessary for the correct behavior of the plant, while for other messages it does not really matter: delays certainly affect the quality of service but do not compromise in any way the operations. We usually refer to the first kind of messages as *time sensitive* messages, while the second class is called *best effort*: we try to achieve the best possible performance, but without any guarantee.

So we have seen that a large amount of data has to flow across the plant, some with "constraints" on their delivery time and some without. The major problem is that if we use a single standard switched network as a mean for all these messages to flow, it is highly unlikely that we can guarantee the delivery time of any message. This result is mostly due to the quantity and the nature of the best effort data: they are in fact messages with very variable dimensions and rate, and they are generally many orders of magnitude more numerous with respect to the time sensitive messages. The danger is that at some point the network will become congested because too high a number of messages has been sent in a small amount of time, and the resources of the network cannot satisfy all the requests in the usual amount of time. The

victims, namely the messages that are going to face an inevitable delay, can be anyone, and there is really no way to predict when or where this scenario will take place. Time sensitive messages will be enqueued in some router along their path towards the destination, behind some best effort messages that got there first; and they will have to wait for the queue to get empty before they can leave it and reach the destination. As one can imagine, it is easy to lose control over what happens and leave things to randomness.

The most widespread solution to this problem is to use two, or more, separate networks, each one dedicated to a particular category among the ones described before. So we have a whole network where only enterprise information flows towards the different departments of the plant, and we have the local networks in the automatic machines that let every component have a “protected” communication channel with its PLC, namely the system buses. System buses are different from a generic network because we need to add some features to make them real-time compatible: those features in general involve reducing the efficiency, the performances or the degrees of freedom in order to gain something in the determinism area.

The advantage of this approach is obviously that we can avoid disturbances on the time sensitive messages by simply eliminating any other message from the network. But this is also kind of a drawback: the fact that we use separate networks in the same plant prevents us from being able to communicate from any device to any other device. Solutions to bridge the two networks in a way that does not affect the properties of the buses exist but are expensive and pretty impractical. In a world where the idea that “everything is connected” and “everything is online” is progressively emerging with the Industrial Internet of Things and Industry 4.0, this perspective seems a bit limiting and old.

That’s where Time Sensitive Networks come into play.

A new technology with new possibilities

Time Sensitive Networks is the name of a series of standards that are being developed by the Institute of Electrical and Electronic Engineers that aim at a better integration of all the traffic classes in the industrial communication framework.

The key idea is to exploit some new mechanisms and routing strategies in order to re-gain control over what happens in large and possibly congested networks. As we said earlier, the fact that usually standard networks are flooded with huge amounts of messages makes it very hard to predict and compute worst case delivery times for any message. Therefore we have to apply a classical tradeoff: we need to lose something in terms of performances in order to be able to impose from the outside some boundary to the latency.

That’s what has been done in fieldbuses, in order to guarantee that every

message flowing in the link will be delivered in time. The real innovation about TSN is that we can choose, a priori, which entities are going to be limited and which ones are going to be left free, subject to the randomness of the network.

The potential of this concept is huge: first of all, it allows us to keep the vital functions of the network for the time sensitive messages, granting that every sensor measurement or control signal will reach its intended destination before its deadline. Moreover, and that's the great innovation, it allows us to use one, unique, large-scale network as a mean for all the messages in a plant or factory. Connecting every single device with rest of the industrial world makes the communication system cutting edge and compliant with the new trends of Industry 4.0.

Compared to the classical fieldbuses, these new techniques bring some remarkable advantages. Increased throughput, performances and efficiency, but most importantly a level of interoperability never seen before in this area, as this technology will be completely vendor-independent, and therefore render every device capable by construction of communicating effortlessly with any other device. At least on the physical level; that's because every device also needs to know which is the format of the data that it is reading. TSN only provides a layer 2 common infrastructure, meaning that we are able to physically transmit every message from any device to any other device. But how the messages are built and read it is up to the sender and the receiver, respectively.

Let's give a trivial example. Let's suppose that a drive needs to send to a PLC its measurements about the current position of the shaft, picked up by its integrated encoder. If the drive records, for some reason, the value using the ASCII standard and the PLC reads the message considering the received value as an unsigned integer, the two values will never correspond, making the message itself useless, even if delivered right on time.

The scenario just depicted normally does not occur because the communication networks of automatic machines are very limited in size, allowing to know precisely which devices need to intercommunicate and to adjust consequently their communication protocols. But the degrees of freedom offered by TSN allow, as we said, to potentially put in communication literally everything. At this point it becomes tough to take into account all the possible devices that could share information and provide them all with a suitable protocol. As a matter of fact, in the previous example it was not specified which PLC was the destination of the position, indeed because we could connect a sensor of a certain automatic machine with the PLC of another one; maybe it is the PLC of the downstream machine that has to take as input the flow of incoming products and needs to know their position on the conveyor belt. Or maybe it could even be the enterprise server, which is supposed to record the operational parameters of every device in the plant, allowing for a more sophisticated and complete statistical analysis and data tracking. In

this last case, then, the message is not supposed to meet deadlines in terms of latency but it does not really matter: as long as we have configured the network in the right way, it will be treated as deterministic or as best effort according to the specific needs of the receiver.

The OPC Foundation has been active for some time on this field, producing standard protocols for industrial communication. It has also contributed to the TSN topic, among other organizations such as Kalycito and Fraunhofer, by developing some testbeds involving deterministic data expressed in a standardized and universal format. Particular reference to [1], where a practical test exposed at the 2018 Hannover Fair is described.



Figure 1.1: TSN testbed at the Hannover Fair, 2018

They tested the real time communications between two industrial PCs directly connected through an ethernet cable. They were able to verify that adopting TSN could grant bounded and deterministic jitters in terms of latencies for the messages, while using the standard network's services the value of the jitter could become quite unpredictable and not controllable.

Putting aside, for the moment, the discussion about ERP and the higher levels of the automation pyramid, let's just consider the productive process, made by one or more coordinated automatic machines. What could be some possible use cases of an actual implementation of TSN in this limited area?

The main application is certainly a real-time machine-to-machine communication (M2M). M2M communication -two machines communicating without human interaction- is reinventing manufacturing by enabling data to be

shared across different control and analytical applications to derive superior operational efficiencies. TSN enhances M2M communication by connecting previously unconnected proprietary controllers. This is made possible through a network of TSN machines (TSN-compliant end points) connected through TSN-enabled switches. M2M communication can enable remote management, operation of equipment/devices through cellular point-to-point connections.

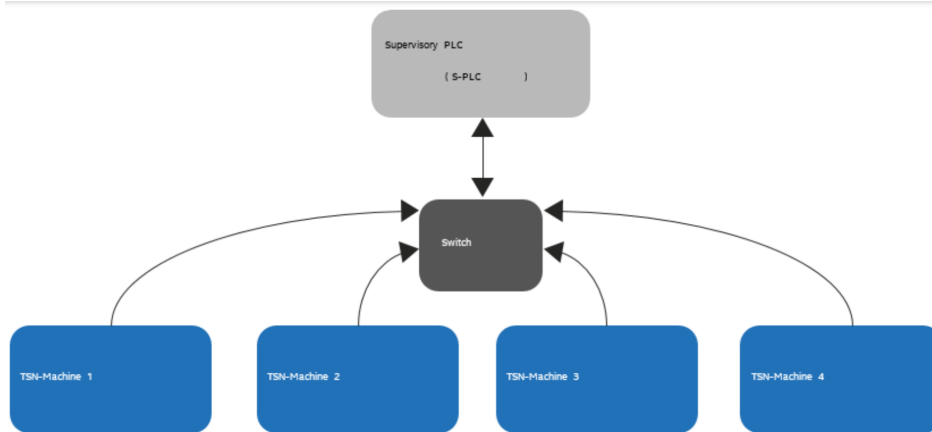


Figure 1.2: machine-to-machine communication

Figure 1.2 shows a production cell with a supervisory PLC coordinating communication across four different TSN machines. Another example of M2M communication can be PLCs communicating with other controllers, conveyor belts, and other control equipment (at the same network layer) to regulate or monitor the production of a product.

TSN could also be exploited in robotic applications. An industrial robot is a programmable, mechanical device used in place of a person to perform dangerous or repetitive tasks with a high degree of accuracy. Based on the operating environment, industrial robots can be classified as fixed (robotic arms), mobile (autonomous guide vehicles) and collaborative (pick and place robots). A key challenge in robotics is the absence of a standard communication protocol. Robotic manufacturers must support many customized protocols, which can lead to increased integration times and costs. Since modern robotics integrates artificial intelligence (AI), machine vision, and predictive maintenance into one system, there is a need for sensors and actuators to stream high bandwidth data in real time. A common solution is to use a specific channel for real-time control (similar to the system buses) and a separate one for higher bandwidth communications (TCP/UDP), just like we highlighted before. For applications that generate high bandwidth traffic (100 MB/s to 1 GB/s), using two separate communication channels becomes inefficient. TSN provides a shared communication channel for high

bandwidth traffic and real-time control traffic.

Lastly, motion control applications have strict delay requirements to ensure that real-time data transmissions can support workload demands. Motion control spans various industrial market segments (discrete industries, process industries, power industry, etc.) and supports dedicated applications such as PLC controllers. Other use cases include controlling the velocity or position of a mechanical device, hydraulic pumps, linear actuators, or electric motors. As the automation industry consolidates its operations, motion controllers need to process more workloads, resulting in a greater need for increased bandwidth and information transparency between different levels in the factory.

For example, next generation PLC machines require response times to be in the low microsecond range. TSN was developed to accommodate these development and represents the next step in the evolution of dependable and standardized industrial communication technology. TSN standards allow the specification of Quality of Service, which enables time-sensitive traffic to efficiently navigate through networks. According to a "Markets and Markets" research report, the market for motion control is expected to reach 22.84 billion dollars by 2022. The main drivers for this adoption will be metal and machinery manufacturing, as industry leaders look to improve speed and accuracy along with increased production.

Even though the perspectives just presented seem appealing and simple to implement, TSN is not immune from drawbacks. First of all, we can currently guarantee the real-time properties of the messages only for limited-sized networks, where no more than 7/8 elements stand between host and destination. This does not mean that we cannot implement that large and unique network for the whole plant that we described before, but rather that real-time messages will be constrained in a local area, while the other kind of messages will have no boundaries. The technical details will be presented in the next chapters.

But, most importantly, the main issue about TSN right now is that it is still an experimental, work-in-progress technology, both in terms of standards definition and hardware support. For this reason most of the work that has been done in this field is related to the theoretical aspects rather than on the development of actual tests and practical solutions. The web has a fair amount of presentations, brochures -even academic papers- describing what we have briefly summarized in the first pages and exploring the theoretical possibilities and implications of this technology; it lacks, though, guides, tutorials or examples on how to perform actual tests starting from the basic elements.

The internship and its main objectives

This thesis work has been realized in collaboration with Marchesini Group. Marchesini is an automation company operating in the field of packaging for pharmaceutical and cosmetic products. It is based in Pianoro (BO) but has several other productive plants in Italy.

One of the main characteristics of Marchesini is the trend to develop internally all the necessary tools useful for the design and production of automatic machines. It is so for the operating system, based on a Linux distribution, for the software development environment and for the programming language. All these features contribute to make Marchesini's machines performing, durable and unique. It is particularly important, for these reasons, to study and comprehend well the necessary requirements, both on the hardware and software sides, for every new technology, in order to be able to integrate it in the company's ecosystem. That was also the case for TSN.

Nobody had ever done anything related to Time Sensitive Networking in the company, but Stefano Gualmini, R&D coordinator and the co-supervisor of this thesis, was interested in deepening the topic. He had previously seen TSN at automation fairs and exhibitions and wanted to explore the possibilities opened by this new technology, considering a future implementation in Marchesini itself.

So the starting goal for this project was to acquire some knowledge in the matter by simply searching the internet for basic information; then, depending also on the results of the research part, start to think about how to build a testbed in order to evaluate the performances and the implementation details. As we already mentioned, it was important to focus on the hardware and software requirements of the system, in order to be able to replicate sooner or later the technology, independently from vendors that may offer, even now, commercial -but specific- solutions.

I began my internship on September 1st, 2020. The first part was held in Casalecchio (BO), at a subsidiary company of Marchesini which usually deals with the training of the new employees. This in order to get started on the complex ecosystem adopted in Marchesini that we outlined before.

In October I was transferred at the company's headquarters in Pianoro, more precisely in the R&D department. Here I started researching and studying the material concerning TSN, as well as figuring out a possible setup to build in order to test them. During this month I also laid the foundations for the simulation environment on Matlab/Simulink that we will present in the final chapter.

Eventually, in November, I moved to Imola (BO) where a software house of Marchesini is located. With the help of Marco De Vietro, we built and configured a simple setup implementing the time sensitive features that we later tested. After three weeks of testing, since I had already finished the mandatory hours of the internship, I completed the simulation part at home.

Outline

So far we've seen that the topic of communication in industrial automation is quite vast and complex. In a generic plant there are two main kinds of data that have to flow: time sensitive and best effort. The current solution adopted in order to grant proper delivery times to the time sensitive data is to use two separate networks, one for the time sensitive flows and the other for the rest. The two networks differ from one another and thus are not interchangeable; fieldbuses need additional features to make it possible to meet the deadlines.

Time Sensitive Networking could allow a successful integration of these two networks, aligning the communication infrastructure with the new paradigms of Industry 4.0 and Industrial Internet of Things. In the last pages a brief description has been provided of what TSN means and what are the main advantages and possibilities; the details and particularities, along with the proper terminology and naming systems, have been intentionally left out.

This work represents an initial step towards a better understanding of the principles behind such a technology, aimed at a possible future implementation. Its structure is the following.

After this introductory chapter we'll be presenting the main theoretical concepts supporting the claims of TSN, such as the communication model, known as stack, and the principles of industrial communication such as the OPC UA protocol. Eventually we will get in the deep about TSN, describing the mechanisms and strategies that make it work.

In the third chapter we are going to describe the tests that have been carried out. The chapter includes the software and hardware features of the setup, along with the results obtained in several different scenarios.

The fourth chapter is dedicated to the simulation. As we will see, a simulation environment could come in hand when we need to evaluate the performances of large networks, which would be too expensive to replicate in real life. The simulation library will be first validated by comparing its results with the experimental ones, and then will be exploited to test some innovative scheduling algorithms that are very useful in the configuration process of a time sensitive network. For this reason the topic is placed after the tests.

In the last chapter we conclude the thesis with a summary of the work done and an additional presentation on the numerous future developments of the work related to TSN.

Chapter 2

Background

2.1 The network stack and the Internet

Today's Internet is arguably the largest engineered system ever created by mankind, with hundreds of millions of connected computers, communication links, and switches; with billions of users who connect via laptops, tablets, and smartphones; and with an array of new Internet-connected devices such as sensors, game consoles, picture frames, and even washing machines.

In this section we are going to provide a description of the basic elements and mechanisms that make it work. Since it is also a particular case of a computer communication network, we are going to highlight the necessary features and characteristics that a communication network should have in order to function properly, with particular attention to those relevant for TSN.

2.1.1 The internet's jargon

The Internet is a computer network that interconnects hundreds of millions of computing devices throughout the world. It can be viewed as an infrastructure that provides services to distributed applications running on different devices. All of these devices are called *hosts* or *end systems*; as of July 2011, there were nearly 850 million end systems attached to the Internet, not counting smartphones, laptops, and other devices that are only intermittently connected to the Internet.

End systems are connected together by a network of *communication links* and *packet switches*. There are many types of communication links, which are made up of different types of physical media, including coaxial cable, copper wire, optical fiber, and radio spectrum. Different links can transmit data at different rates, with the *transmission rate* of a link measured in bits/second. When one end system has data to send to another end system, the sending end system segments the data and sends them into the network. The resulting packages of information, known as *packets* in the jargon of

computer networks, are then sent through the network to the destination end system, where they are reassembled into the original data.

So a packet is basically just a sequence of bits that are grouped together to make sense and actually transmit pieces of information. The information that each packet carries around is not only related to what the sender has to communicate to the receiver, though. Since the packet is moved in the network, additional data need to be added to it in order for the switches to be able to sort all the incoming packets and send them in the right directions. Therefore, standards define a particular structure, namely some fields that necessarily compose a packet, each one filled with relevant information; the first chunk of bits that gets sent is called header, and tells the reader the details of the packets, such as its length, protocol used, source and destination, and possibly some other fields useful to perform error detection.

The following picture shows an example of an IPv4 packet structure.

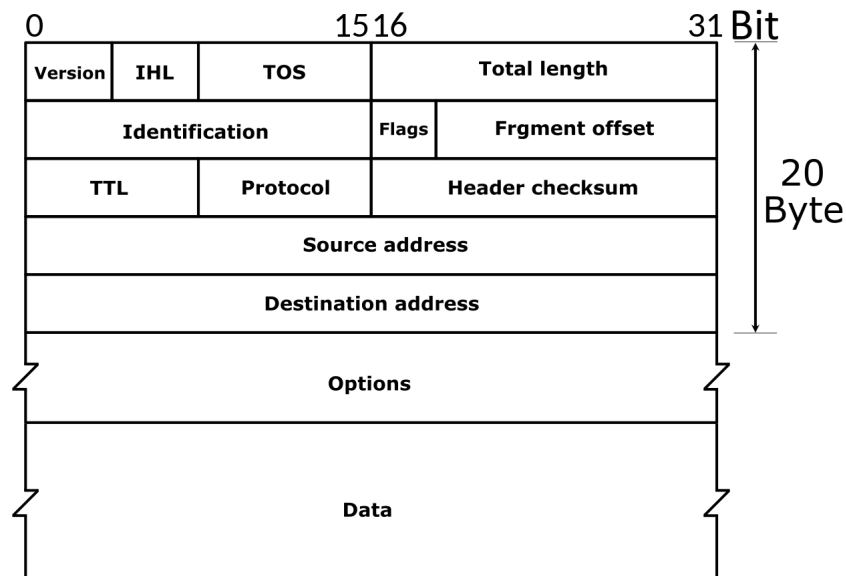


Figure 2.1: Structure of a packet

A packet switch takes a packet arriving on one of its incoming communication links and forwards that packet on one of its outgoing communication links.

The sequence of communication links and packet switches traversed by a packet from the sending end system to the receiving end system is known as a *route* or *path* through the network.

End systems, packet switches, and other pieces of the Internet run protocols that control the sending and receiving of information within the Internet. The Transmission Control Protocol (*TCP*) and the Internet Protocol (*IP*) are two of the most important protocols in the Internet. The IP protocol

specifies the format of the packets that are sent and received among routers and end systems. The Internet's principal protocols are collectively known as TCP/IP.

The typical model for computers communicating on a network is request-response. In the request-response model, a client computer or software requests data or services, and a server computer or software responds to the request by providing the data or service. This model is also named "client/server" communication model.

A different way for devices to communicate on a network is called publish-subscribe, or pub-sub. In a pub-sub architecture, a central source called a broker (also sometimes called a server) receives and distributes all data. Pub-sub clients can publish data to the broker or subscribe to get data from it, or both. Clients that publish data send it only when the data changes (report by exception, or RBE). Clients that subscribe to data automatically receive it from the broker/server, but again, only when it changes. The broker does not store data; it simply moves it from publishers to subscribers. When data comes in from a publisher, the broker promptly sends it off to any client subscribed to that data.

2.1.2 The network's core

In a network application, end systems exchange messages with each other. Messages can contain anything the application designer wants; messages may also perform a control function.

To send a message from a source end system to a destination end system, the source breaks long messages into smaller chunks of data known as packets. Between source and destination, each packet travels through communication links and packet switches.

Packets are transmitted over each communication link at a rate equal to the full transmission rate of the link. So, if a source end system or a packet switch is sending a packet of L bits over a link with transmission rate R bits/sec, then the time to transmit the packet is L/R seconds.

Most packet switches use *store-and-forward* transmission at the inputs to the links. Store-and-forward transmission means that the packet switch must receive the entire packet before it can begin to transmit the first bit of the packet onto the outbound link. Let's consider, for instance, a simple network made by two end systems and a router in between, which takes as input a stream of data from one end system and redirects such stream to the other end system; as we mentioned, data streams are composed by several different packets, which in turn are made by an appropriate number of bits.

Because the router employs store-and-forwarding, the router cannot transmit the bits it has received at any given moment; instead it must first buffer (i.e. "store") the packet's bits. Only after the router has received all of the packet's bits it can begin to transmit (i.e., "forward") the packet onto the

outbound link.

As a consequence, if we refer to R as the bitrate of the link and to L as the number of bits of each packet, we are able to compute the total delay (i.e. the time that passes between the sending of the first bit and the arrival of the last one): $2L/R$. If the switch instead forwarded bits as soon as they arrive (without first receiving the entire packet), then the total delay would be L/R since bits are not held up at the router. But routers need to receive, store, and process the entire packet before forwarding.

Each packet switch has multiple links attached to it. For each attached link, the packet switch has an output buffer (also called an output queue), which stores packets that the router is about to send into that link. The output buffers play a key role in packet switching. If an arriving packet needs to be transmitted onto a link but finds the link busy with the transmission of another packet, the arriving packet must wait in the output buffer. Thus, in addition to the store-and-forward delays, packets suffer output buffer queuing delays. These delays are variable and depend on the level of congestion in the network. Since the amount of buffer space is finite, an arriving packet may find that the buffer is completely full with other packets waiting for transmission. In this case, packet loss will occur: either the arriving packet or one of the already-queued packets will be dropped.

Another duty that switches are charged of is packet forwarding. In the Internet, every end system has an address called an IP address. When a source end system wants to send a packet to a destination end system, the source includes the destination's IP address in the packet's header (see Fig. 2.1). As with postal addresses, this address has a hierarchical structure. When a packet arrives at a router in the network, the router examines a portion of the packet's destination address and forwards the packet to an adjacent router. More specifically, each router has a forwarding table that maps destination addresses (or portions of the destination addresses) to that router's outbound links. When a packet arrives at a router, the router examines the address and searches its forwarding table, using this destination address, to find the appropriate outbound link. The router then directs the packet to this outbound link. The Internet has a number of special routing protocols that are used to automatically set the forwarding tables. A routing protocol may, for example, determine the shortest path from each router to each destination and use the shortest path results to configure the forwarding tables in the routers.

In the following picture an example of what we have just described is depicted. We have two end systems connected to the network, which is made by several interconnected switches. An end system may wish to send to another end system pieces of information by means of packets. Packets travel through a series of switches (not necessarily the same switches) from their source towards their destination. Inside each switch the header of the packet is analyzed in order to compute the address of the next destination

and its associated output port; other operations may be performed, such as error detection.

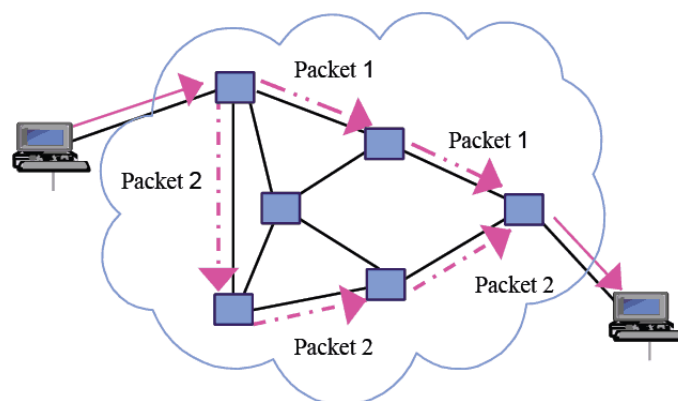


Figure 2.2: An example of a network

End systems (PCs, smartphones, Web servers, mail servers, and so on) connect into the Internet via an access Internet Service Provider. The access ISP can provide either wired or wireless connectivity, using an array of access technologies including DSL, cable, FTTH, Wi-Fi, and cellular. But connecting end users and content providers into an access ISP is only a small piece of solving the puzzle of connecting the billions of end systems that make up the Internet. To complete this puzzle, the access ISPs themselves must be interconnected. This is done by creating what is called a network of networks.

2.1.3 A network's parameter: delay

Ideally, we would like Internet services to be able to move as much data as we want between any two end systems, instantaneously, without any loss of data. Unfortunately, this is an impossible goal, one that is unachievable in reality; we have already seen a mechanism that causes packets to be delayed, i.e. the store-and-forward policy implemented in routers.

As a matter of fact, computer networks necessarily constrain throughput (the amount of data per second that can be transferred) between end systems, introduce delays between end systems, and can actually lose packets due to the physical laws of reality.

Recall that a packet starts in a host (the source), passes through a series of routers, and ends its journey in another host (the destination). As a packet travels from one node (host or router) to the subsequent node (host or router) along this path, the packet suffers from several types of delays at each node along the path. The most important of these delays are the nodal processing delay, queuing delay, transmission delay, and propagation

delay; together, these delays accumulate to give a total nodal delay. The performances of many Internet applications are greatly affected by network delays.

In this subsection we are going to define better these kinds of delay, which represent a useful parameter in the evaluation process of a network's performances.

Let's consider for instance a generic router of a network, which has some input ports that feed it incoming packets, and as many output ports through which it can redirect said packets. We can brake the overall delay in some smaller delays, taking place at each router along the way: so, in general, the longer the path toward the destination, the longer the packet will be constrained to wait. Let's then try to figure out the possible kinds of delays at the router of the example.

When the packet arrives at router A from the upstream node, router A examines the packet's header to determine the appropriate outbound link for the packet and then directs the packet to this link. In this example, the outbound link for the packet is the one that leads to router B. A packet can be transmitted on a link only if there is no other packet currently being transmitted on the link and if there are no other packets preceding it in the queue; if the link is currently busy or if there are other packets already queued for the link, the newly arrived packet will then join the queue.

The time required to examine the packet's header and determine where to direct the packet is part of the processing delay. The processing delay can also include other factors, such as the time needed to check for bit-level errors in the packet that occurred in transmitting the packet's bits from the upstream node to the router under examination. Processing delays in high-speed routers are typically on the order of microseconds or less. After this nodal processing, the router directs the packet to the queue that precedes the link to the next node in the network.

At the queue, the packet experiences a queuing delay as it waits to be transmitted onto the link. The length of the queuing delay of a specific packet will depend on the number of earlier-arriving packets that are queued and waiting for transmission onto the link. If the queue is empty and no other packet is currently being transmitted, then our packet's queuing delay will be zero. On the other hand, if the traffic is heavy and many other packets are also waiting to be transmitted, the queuing delay will be long. It is quite intuitive that the number of packets that an arriving packet might expect to find is a function of the intensity and nature of the traffic arriving at the queue. Queuing delays can be on the order of microseconds up to milliseconds in practice.

Assuming that packets are transmitted in a first-come-first-served manner, as it is common in packet-switched networks, our packet can be transmitted only after all the packets that have arrived before it have been transmitted. We denote the length of the packet by L bits, and denote the transmis-

sion rate of the link from our router to the downstream router by R bits/sec. For example, for a 10 Mbps Ethernet link, the rate is $R = 10$ Mbps; for a 100 Mbps Ethernet link, the rate is $R = 100$ Mbps. The transmission delay is L/R . This is the amount of time required to push (that is, transmit) all of the packet's bits into the link. Transmission delays are typically on the order of microseconds to milliseconds in practice.

Once a bit is pushed into the link, it needs to propagate to the next router or host. The time required to propagate from the beginning of the link to its end is the propagation delay. The bit propagates at the propagation speed of the link. The propagation speed depends on the physical medium of the link (that is, fiber optics, twisted-pair copper wire, and so on) and is in general a little smaller than the speed of light. The propagation delay is therefore the distance between two routers divided by the propagation speed. That is, the propagation delay is d/s , where d is the distance between router A and router B and s is the propagation speed of the link. Once the last bit of the packet propagates to node B, it and all the preceding bits of the packet are stored in router B. The whole process then continues with router B now performing the forwarding. In wide-area networks, propagation delays are on the order of milliseconds.

There's a big difference between transmission delay and propagation delay. Transmission delay is the rate at which we are able to inject bits into the link, whereas the propagation delay is the time that those bits take to cross the link from the beginning to the end. The first parameter is hence a function of the packet's size, while the second is a function of the distance between the two consecutive nodes. In contrast, transmission delay does not depend on the distance nor does propagation time depend on the number of bits.

In summary, we can compute the nodal delay as the summation of four contributions.

$$d_{nodal} = d_{processing} + d_{transmission} + d_{propagation} + d_{queueing} \quad (2.1)$$

The first three terms of the equation mostly depend on the hardware capabilities of the system and on its configuration, as we have explained so far. The most complicated and interesting component of nodal delay is the queuing delay. Unlike the other three delays, the queuing delay can vary from packet to packet. For example, if 10 packets arrive at an empty queue at the same time, the first packet transmitted will suffer no queuing delay, while the last packet transmitted will suffer a relatively large queuing delay (while it waits for the other nine packets to be transmitted). For these reasons this kind of delay is quite unpredictable, and currently object of studies and papers.

Here, for explanation purposes, we are going to provide a few examples about the behavior of the queueing delay, based on some simple assumptions.

First of all, in order to avoid instabilities, we need to make sure not to saturate the network: if we indicate the transmission capacity of the port with R and we assume that packets of the same size (L bits) arrive with constant frequency (a entities per second), we must grant that the ratio aL/R is smaller than 1. Namely, that the sending capabilities of the port are more powerful than the size of the incoming data stream, otherwise the buffer associated with the port will keep growing up to the point it will have to discard some packets.

Furthermore, if we now consider a scenario where the ratio aL/R is smaller than 1 and the packets have all the same size, the behavior of the queueing delay depends on how the packets arrive to the router. If in fact packets arrive with a constant rate and equal interarrival times, each one will have very low waiting times as the previous packet, which has arrived a long time ago, has already finished its transmission. On the other hand, if packets arrive in bursts, each one separated from the last one from the proper amount of time in order to guarantee the non-saturating property that we discussed before, the waiting time will vary greatly from packet to packet. First ones to arrive, in a single burst, will have low (yet increasing) waiting times, while the last one will have to wait a time almost equal to the transmission time of all the previous packets.

In an actual network we cannot assume that packets have all the same size, nor can we attribute some pattern to their interarrival times, so the matter gets more and more complicated. Through this examples, though, it is already possible to sense that there could exist some techniques aimed at the successful reduction of the waiting times of packets in routers: in particular for this case, avoid bursts of messages and keep the sending frequency as constant as possible.

In addition, real queues preceding an output port have a limited size, which greatly depends on the router's capacity and cost. Because the queue capacity is finite, packet delays do not really approach infinity as the traffic intensity (aL/R) approaches 1. Instead, a packet can arrive to find a full queue. With no place to store such a packet, a router will drop that packet; that is, the packet will be lost. From an end-system viewpoint, a packet loss will look like a packet having been transmitted into the network core but never emerging from the network at the destination. The fraction of lost packets increases as the traffic intensity increases. Therefore, performance at a node is often measured not only in terms of delay, but also in terms of the probability of packet loss. It will be then duty of the end systems to detect that packet loss has occurred, through some network mechanisms, and fix the problem, most likely by sending again the packet.

At this point we can compute the parameter called end-to-end delay, i.e. the time that it takes to a packet to travel through the network from its source to its destination. Specifically it is the lowest possible value that this parameter can assume, according to the configuration of the network and its

hardware resources. Assuming that a path composed by N switches exists between source and destination of a certain packet

$$d_{end-to-end} = \sum_{i=1}^N (d_{processing,i} + d_{transmission,i} + d_{propagation,i}) \quad (2.2)$$

As we mentioned, this is the value we obtain in a totally uncongested network, where packets are sent out as soon as they arrive to a node. As the traffic size increases, we are going to feel the effects on the queueing delay at each node, that won't be constant in time nor easily predictable. A possible strategy we could implement to reduce the end to end delay is not to reduce the single nodal delays, but to actually change the routes of the packets in order to achieve an overall higher (and better) network utilization. This solution involves the exploitation of optimization algorithms that will compute the optimal route that packets need to follow in order not to stress some routers but, rather, to take maybe longer paths but through little-used routers. This way we could balance the overall load on the network's elements and reduce on average the end to end delay. This is a very advanced topic, though, that requires good knowledge of optimization problems and numerical computations; it will find more space in the last chapter, where we are going to discuss some interesting new techniques of network's organization.

In addition to delay and packet loss, another critical performance measure in computer networks is end-to-end throughput. It is the rate, instantaneous or average, of bits that an end systems manages to successfully transfer to another end system. Throughput and delays are inversely correlated: if packets suffer delays due to congestion in the network, their arrival rate to the destination will decrease, and so will the throughput.

Throughput may also suffer from the bottleneck effect. If, along the way, there is a transmission link which has a low transmission capacity or needs to be used by several different flows, then the overall throughput will not be greater than the particular throughput of that link.

2.1.4 Layered architecture

So far we have seen that the Internet, and communication networks in general, are extremely complex systems. Packets, switches, routers, applications, communication protocols are all elements that have to be defined, handled and coordinated in order for the whole ensemble to work as intended.

The most important feature about communication networks is that they are built following a layered architecture. This means that we have a hierarchical structure, divided in levels, each one providing certain services to the levels above and exploiting the services provided by the levels below.

That's the main idea; it may be even more complicated, at first, but it leads to a definitely better organization of all the components involved and

brings some additional degrees of freedom that come in hand throughout the life of the network. A layered architecture allows us to discuss a well-defined, specific part of a large and complex system. This simplification itself is of huge value by providing modularity, making it much easier to change the implementation of the service provided by the layer. As long as the layer provides the same service to the layer above it, and uses the same services from the layer below it, the remainder of the system remains unchanged when a layer's implementation is changed: changing the implementation of a service is in fact very different from changing the service itself.

So, to provide structure to the design of network protocols, network designers organize protocols -and the network hardware and software that implement the protocols- in layers; what we are interested in is the service that a layer offers to the layers above and below, the so-called service model of a layer. A protocol layer can be implemented in software, in hardware, or in a combination of the two.

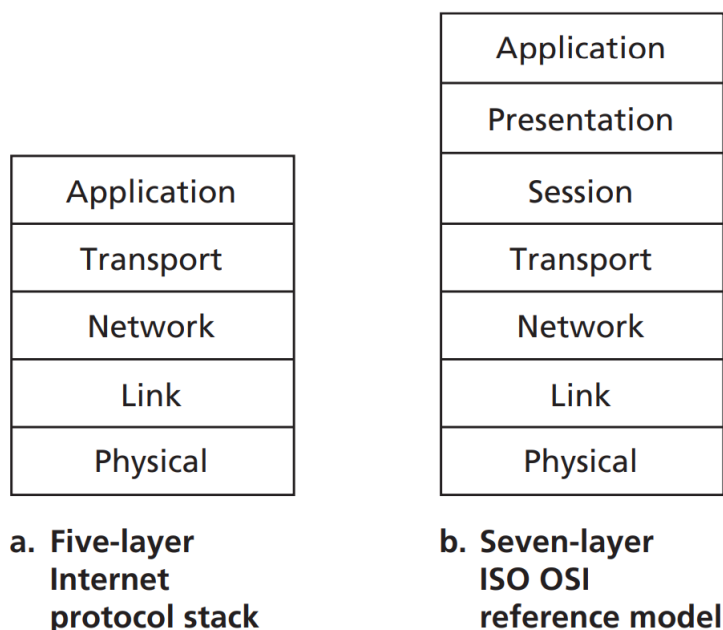


Figure 2.3: Comparison between two protocol stacks

In the late 1970s, the International Organization for Standardization (ISO) proposed that computer networks be organized around seven layers, called the Open Systems Interconnection (OSI) model. The OSI model took shape when the protocols that were to become the Internet protocols were in their infancy, and the inventors of the original OSI model probably did not have the Internet in mind when creating it. Because of its early impact on networking education, the seven-layer model continues to be a reference

point both in networking textbooks and in real applications. The seven layers of the OSI reference model, shown in Fig. 2.3 (b), are: application layer, presentation layer, session layer, transport layer, network layer, data link layer, and physical layer.

Figure 2.3 (a) anticipates that the Internet protocol stack consists of five layers: the physical, link, network, transport, and application layers.

The application layer is where network applications and their application-layer protocols reside. The Internet's application layer includes many protocols, such as the HTTP protocol (which provides for Web document requests and transfers), SMTP (which provides for the transfer of e-mail messages), and FTP (which provides for the transfer of files between two end systems). For instance, certain network functions, such as the translation of human-friendly names for Internet end systems like `www.google.it` to a 32-bit network address, are also done with the help of a specific application-layer protocol, namely, the domain name system (DNS). An application-layer protocol is distributed over multiple end systems, with the application in one end system using the protocol to exchange packets of information with the application in another end system.

The Internet's transport layer transports application-layer messages between application endpoints. In the Internet there are two transport protocols, TCP and UDP, either of which can transport application-layer messages. TCP provides a connection-oriented service to its applications. This service includes guaranteed delivery of application-layer messages to the destination and flow control (that is, sender/receiver speed matching). TCP also breaks long messages into shorter segments and provides a congestion-control mechanism, so that a source throttles its transmission rate when the network is congested. The UDP protocol provides a connectionless service to its applications. This is a simpler service that provides no reliability, no flow control, and no congestion control, but it is definitely lighter with respect to TCP.

The Internet's network layer is responsible for moving network-layer packets known as datagrams from one host to another. The Internet transport-layer protocol (TCP or UDP) in a source host passes a transport-layer segment and a destination address to the network layer. The network layer then provides the service of delivering the segment to the transport layer in the destination host. The Internet's network layer includes the famous IP Protocol, which defines the fields in the datagram as well as how the end systems and routers act on these fields. There is only one IP protocol, and all Internet components that have a network layer must run the IP protocol. The Internet's network layer also contains routing protocols that determine the routes that datagrams take between sources and destinations. The Internet has many routing protocols and, although the network layer contains both the IP protocol and numerous routing protocols, it is often simply referred to as the IP layer, reflecting the fact that IP is the glue that binds the

Internet together.

The Internet's network layer routes a datagram through a series of routers between the source and destination. To move a packet from one node (host or router) to the next node in the route, the network layer relies on the services of the link layer. In particular, at each node, the network layer passes the datagram down to the link layer, which delivers the datagram to the next node along the route. At this next node, the link layer passes the datagram up to the network layer. The services provided by the link layer depend on the specific link-layer protocol that is employed over the link. For example, some link-layer protocols provide reliable delivery, from transmitting node, over one link, to receiving node. Note that this reliable delivery service is different from the reliable delivery service of TCP, which provides reliable delivery from one end system to another. Examples of link layer protocols include Ethernet, WiFi, and the cable access network's DOCSIS protocol. As datagrams typically need to traverse several links to travel from source to destination, a datagram may be handled by different link-layer protocols at different links along its route. For example, a datagram may be handled by Ethernet on one link and by PPP on the next link. The network layer will receive a different service from each of the different link-layer protocols.

While the job of the link layer is to move entire frames from one network element to an adjacent network element, the job of the physical layer is to move the individual bits within the frame from one node to the next. The protocols in this layer are again link dependent and further depend on the actual transmission medium of the link. For each transmitter-receiver pair, the bit is sent by propagating electromagnetic waves or optical pulses across a physical medium. The physical medium can take many shapes and forms and does not have to be of the same type for each transmitter-receiver pair along the path. Examples of physical media include twisted-pair copper wire, coaxial cable, multimode fiber-optic cable, terrestrial radio spectrum, and satellite radio spectrum. Physical media fall into two categories: guided media and unguided media. With guided media, the waves are guided along a solid medium, such as a fiber-optic cable, a twisted-pair copper wire, or a coaxial cable. With unguided media, the waves propagate in the atmosphere and in outer space, such as in a wireless LAN or a digital satellite channel.

The least expensive and most commonly used guided transmission medium is twisted-pair copper wire. For over a hundred years it has been used by telephone networks. Twisted pair consists of two insulated copper wires, each about 1 mm thick, arranged in a regular spiral pattern. The wires are twisted together to reduce the electrical interference from similar pairs close by. Typically, a number of pairs are bundled together in a cable by wrapping the pairs in a protective shield. A wire pair constitutes a single communication link. Unshielded twisted pair (UTP) is commonly used for computer networks within a building, that is, for LANs. Data rates for LANs using twisted pair today range from 10 Mbps to 10 Gbps. The data rates

that can be achieved depend on the thickness of the wire and the distance between transmitter and receiver. Modern twisted-pair technology, such as category 6a cable, can achieve data rates of 10 Gbps for distances up to a hundred meters. In the end, twisted pair has emerged as the dominant solution for high-speed LAN networking.

Like twisted pair, coaxial cable consists of two copper conductors, but the two conductors are concentric rather than parallel. With this construction and special insulation and shielding, coaxial cable can achieve high data transmission rates. Coaxial cable is quite common in cable television systems. Coaxial cable can be used as a guided shared medium. Specifically, a number of end systems can be connected directly to the cable, with each of the end systems receiving whatever is sent by the other end systems.

An optical fiber is a thin, flexible medium that conducts pulses of light, with each pulse representing a bit. A single optical fiber can support tremendous bit rates, up to tens or even hundreds of gigabits per second. They are immune to electromagnetic interference, have very low signal attenuation up to 100 kilometers, and are very hard to tap. These characteristics have made fiber optics the preferred long-distance guided transmission media, particularly for overseas links. Fiber optics is also prevalent in the backbone of the Internet. However, the high cost of optical devices, such as transmitters, receivers, and switches, has blocked their deployment for short-distance transport, such as in a LAN or into the home in a residential access network.

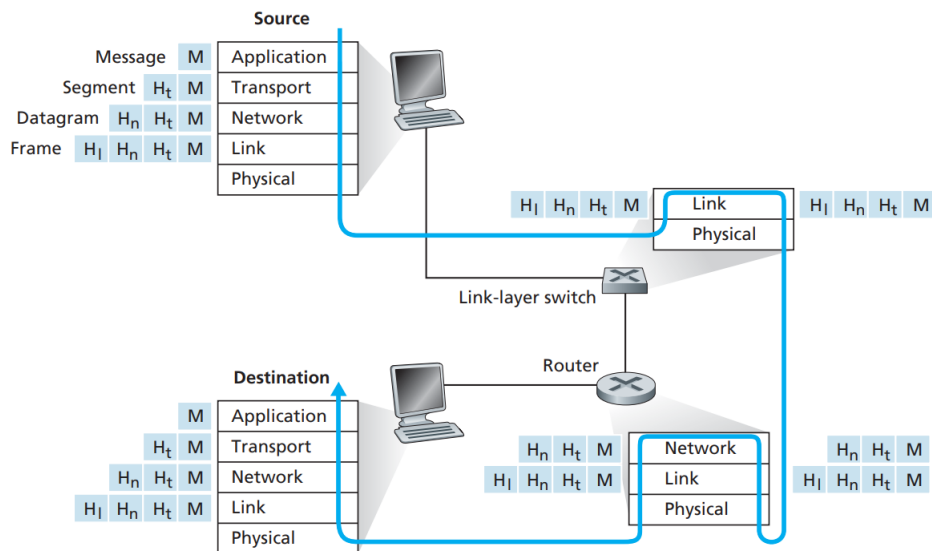


Figure 2.4: An example of the active layers in a transmission

Protocol layering has conceptual and structural advantages. As we have

seen, layering provides a structured way to discuss system components and modularity makes it easier to update system components. It is worth mentioning, however, that some researchers and networking engineers are strongly opposed to layering. One potential drawback of layering is that one layer may duplicate lower-layer functionalities. For example, many protocol stacks provide error recovery on both a per-link basis and an end-to-end basis; in this case, though, being able to detect an error in the single link is much cheaper than having to retransmit again the whole packet from source to destination, so this duality may increase the performances.

A second potential drawback is that functionality at one layer may need information (for example, a timestamp value) that is present only in another layer; this violates the goal of separation of layers.

2.1.5 Link to TSN

In this section we have covered all the notions necessary to have a basic understanding of what computer networks are and how they work. We have noted that they are organized in layers, which give us a structured and modular way of analysis. We have also commented on the fact that the performances usually depend on a large number of factors, most of which are not under direct control nor even predictable with sufficient accuracy.

TSN exploits the layered structure of the networks since, as we will see, deals with a layer 2 particular implementation. According to what we have said, this means that, in our considerations, we do not need to worry about what happens in the higher levels and we only need to use what we get from the physical network.

Another relevant aspect concerning TSN is the delay, which we have analyzed in subsection 2.1.3. It is important to understand why it exists in the system and is inevitable, how it is generated and what is its behavior with the major dependencies.

Eventually, not strictly connected to TSN itself but to the possible implications, it is worth keeping in mind the two communication models for hosts in a network, namely client/server and publisher/subscriber, with the associated features.

2.2 New trends in the automation world

2.2.1 Introduction to Industry 4.0

Industry 4.0 is a recent, strategic initiative whose goal is the transformation of industrial manufacturing through digitalization and exploitation of potentials of new technologies. An Industry 4.0 production system is thus flexible and enables individual and customizable products.

These requirements can be met only by radical advances in current manufacturing technology. Most of their technical aspects are addressed by the application of the generic concepts of Cyber-Physical Systems (CPS) and Industrial Internet of Things (IIoT) to the industrial production systems. The Industry 4.0 "execution system" is therefore based on the connections of CPS building blocks. These blocks are embedded systems with decentralized control and advanced connectivity that are collecting and exchanging real-time information with the goal of identifying, locating, tracking, monitoring and optimizing the production processes.

Industry 4.0, as the first government-led initiative and inspiration for other similar initiatives, comes from Germany, at the beginning of the last decade.

Then, the concept of Industrial Internet has been brought up in North America by the General Electric company in late 2012; it is seen as a tight integration of physical and digital worlds that combines big data analytics with the Internet of Things, and assumes a much broader application area with respect to Industry 4.0.

In France, the concept "Industrie du futur" was introduced as a core of the future French industrial policy. It is based on cooperation of industry and science and built on five pillars: cutting edge technologies, support to the French companies, training, international cooperation and promotion.

China also joined this trend with its "Made in China 2025" program, in 2015, which was largely inspired by the german Industry 4.0.

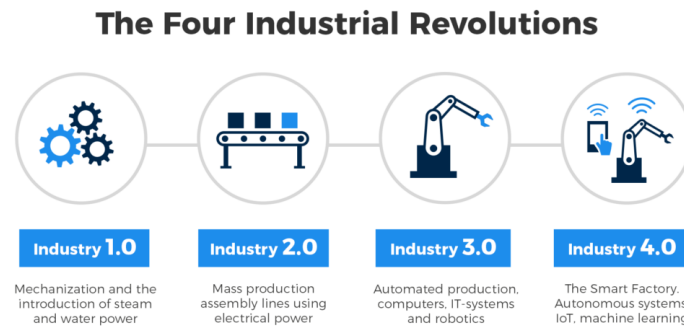


Figure 2.5: Characteristics of the four industrial revolutions

2.2.2 Core idea of Industry 4.0

The basic concept was first presented at the Hannover fair in the year 2011. Since its introduction, Industry 4.0 is in Germany a common discussion topic in research, academic and industry communities at many different occasions.

The main idea is to exploit the potentials of new technologies and concepts such as:

- availability and use of the internet and IoT
- integration of technical processes and business processes in the companies,
- digital mapping and virtualization of the real world
- "Smart" factory including "smart" means of industrial production and "smart" products
- the concept of lean manufacturing, a production method invented by Toyota and aimed at the optimal reduction of production costs and efforts



Figure 2.6: Smart factory

Figure 2.6 shows the Industry 4.0 smart factory. The core process is digital to physical conversion in a reconfigurable manufacturing system. Reconfigurable manufacturing systems are the latest advance in the development of a manufacturing system. First step were fixed production lines with the machines dedicated to the performance of specific tasks so only one product could be produced. Next step were flexible production systems with programmable machines that allowed production of a variety of different products but offered no flexibility in the production capacity. The results of the latest development are reconfigurable manufacturing systems able to adapt their hardware and software components to follow ever-changing market requirements of type and quantity of the products.

Machines in Industry 4.0 factory are Cyber-Physical Systems, namely physical systems integrated with ICT components. They are autonomous systems that can make their own decisions based on machine learning algorithms and real-time data capture, analytics results, and recorded successful past behaviors. Typically, programmable machines are used, with a large share of mobile agents and robots able of self-organization and self-optimization.

Products in such factory are also "smart", with embedded sensors that are used via wireless networks for real-time data collection, for localization, for measuring product state and environment conditions. Smart products also have control and processing capabilities. Thus they can control their logistical path through the production and even control/optimize the production workflow that concerns them. Furthermore, smart products are capable of monitoring their own state during the whole life cycle. This enables active, condition-based maintenance that is especially important for products embedded in larger systems .

In Industry 4.0, the production elements have, beside their physical representation, also virtual identity, a data object that is stored in the data cloud. Such virtual identity can include a multitude of data and information about the product, from documents, to 3-D models, individual identifiers, current status data, history information and measurement/test data.

Important elements of the Industry 4.0 concept are also interoperability and connectivity. A continuous flow of information between the devices and components, Machine-To-Machine interaction (M2M), manufacturing systems and actuators should be established. Hereby the machines, products and factories can connect and communicate via the Industrial IoT (mostly based on wireless network). Another important topic is Human-To-Machine (H2M) collaboration that is necessary as some production tasks are too unstructured to be fully automatized. A lot of research effort is currently also invested in so called collaborative robotics. Here human workers and especially designed compliant robots work together in the execution of complex and unstructured work tasks at the manufacturing production line. Such tasks were done completely manually before. Advanced user interfaces are developed for new forms of M2H communication. They often include teleoperation and are based on augmented reality environments. Between the Industry 4.0 manufacturing technologies, 3D printing is often mentioned as one of the key technologies. In combination with rapid prototyping methods including 3D modelling, a direct digital thread can be established from design to production, facilitating a shorter time from the idea to the product. Until now, however, additive manufacturing processes cannot always reach the same quality as a conventional industrial process and some new materials still need to be developed.

Software tools are crucial for operating of the Industry 4.0 smart factory. Figure 2.7 depicts the well known pyramid structure of support software of

modern production systems.

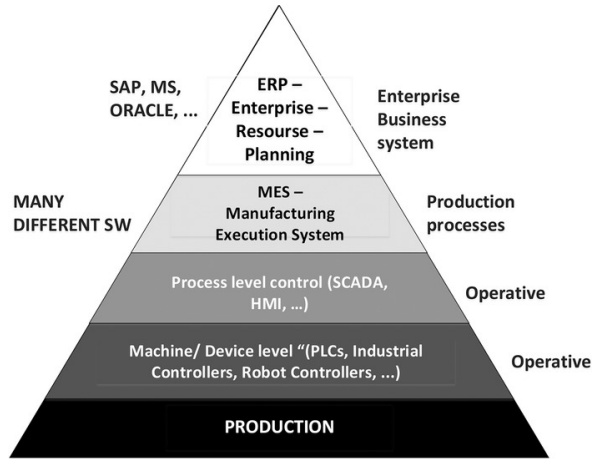


Figure 2.7: Automation pyramid

On the business level, the already mentioned Enterprise Resource Planning (ERP) tool is implemented. ERP supports enterprise-wide planning such as business planning, supply chain management, sales and distribution, accounting, human resource management and similar. Usually commercially available solutions are implemented but they do not support fast adaptation in production planning due to the unplanned events.

The second level in the traditional automation pyramid is Manufacturing Execution System (MES). It supports production reporting, scheduling, dispatching, product tracking, maintenance operations, performance analysis, workforce tracking, resource allocation and similar. It covers aspects such as management of the shop floor and communication with the enterprise (business) systems. Again, most of the software solutions available on the market are centralized and not distributed to the shop floor elements.

The next operative level is process level control based on Supervisory Control and Data Acquisition (SCADA) control system architecture followed by controllers on machine/device level such as Programmable Logic Controllers (PLCs), robot controllers and other controllers. The last level of the automation pyramid is a machine/device level. In opposition to the top two layers, this level has a naturally distributed control level.

ERP and MES tools represent basic software in the company and are used since the nineties. Both systems have typically a modular structure but are centralized in their operation and thus have limited capability for dynamic adaptation of the production plan.

Another important issue is information integration among ERP, MES and other software tools used in the company; problems such as database integration need to be solved and communication protocols need to be defined.

In the end, it can be concluded that for the Industry 4.0 the classical automation structure does not feature the best solution as it is not flexible enough for adapting to the dynamic changes in the order flow and at the shop floor. Distributed MES solution, where most of the functions are decentralized, is expected to be more suitable for the reconfigurable production systems (Fig. 2.8): for full support of reconfigurable systems, a continuous flow of information (vertical and horizontal integration) between all elements should be realized.

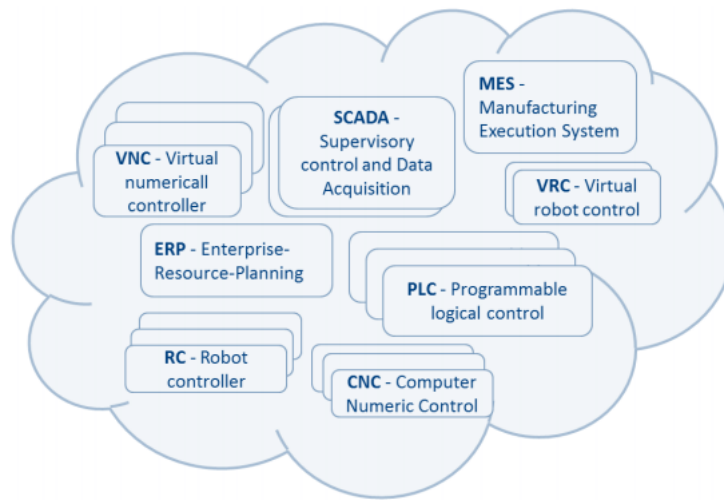


Figure 2.8: IT support

This industry concept is not limited just to the production system but it includes the complete value chain (from suppliers to the customers of one enterprise towards the "Connected World" of all enterprises) and all enterprise's functions and services. It is clear that it is not easy to fulfill these criteria, therefore only some "islands" of the Industry 4.0 concept currently exist.

2.2.3 OPC UA

OPC Unified Architecture (OPC UA) is a vendor-independent communication protocol for industrial automation applications. It is based on the client/server (or even publisher/subscriber) model and allows seamless communication from the individual sensors and actuators up to the ERP system or the cloud. The protocol is platform-independent and features built-in safety mechanisms. Since OPC UA is flexible and completely independent, it is recognized as the ideal communication protocol for the implementation of Industry 4.0.

The OPC Foundation provides specifications for data exchange in industrial automation. There is a long history of COM/DCOM-based specifica-

tions, most prominent OPC Data Access (DA), OPC Alarms and Events (A&E), and OPC Historical Data Access (HDA), which are widely accepted in the industry and implemented by almost every system targeting industrial automation.

The OPC Unified Architecture was born out of the desire to create a true replacement for all existing COM-based specifications without losing any features or performance. Additionally it must cover all requirements for platform-independent system interfaces with rich and extensible modeling capabilities being able to describe also complex systems.

The OPC UA specification is broken into several parts. [UA1] gives an overview and [UA2] explains the security model. [UA4] defines the abstract services, [UA3] the address space model and [UA5] the information model of OPC UA. [UA6] defines the mapping of the abstract services to a concrete technology.

OPC UA specifies an abstract set of services in [UA4] and the mapping to a concrete technology in [UA6]. OPC UA does not specify an API but only the message formats for data exchanged on the wire. A communication stack is used on client- and server-side to encode and decode message requests and responses. Different communication stacks can work together as long as they use the same technology mapping.

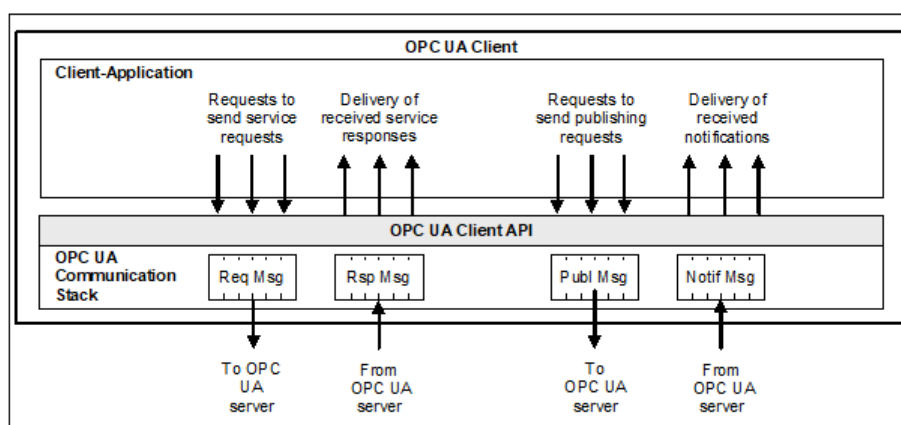


Figure 2.9: OPC UA client

An OPC UA client consists of a Client Implementation using an OPC UA communication stack. The Client Implementation accesses the communication stack using the OPC UA API. Note that the API is not standardized. It may vary for different programming languages and potentially for different communication stacks. Several communication stacks may exist for different operating systems, programming languages and mappings. The clientside communication stack allows the client to create request messages based on the service definitions. The client-side communication stack communicates with a server-side communication stack. The OPC Foundation standard-

ized only this communication. Thus, everybody can develop his or her own communication stack with its own API as well.

The server-side communication stack delivers the request messages to the Server Implementation via the OPC UA API. Since the OPC UA API realizes the abstract service specifications, it may be the same as on the client-side. The Server Implementation implements the logic needed to return the appropriate response message. The OPC UA Sever Implementation gets its data from some underlying system. For example, this can be a configuration database, a set of devices or some OPC server.

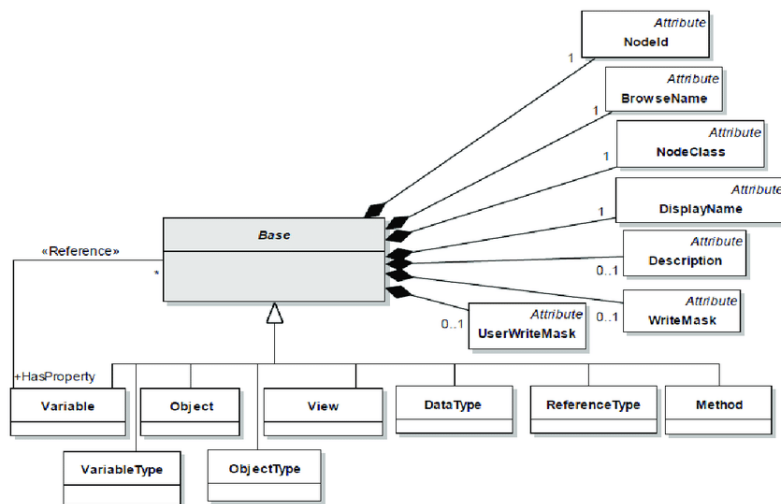


Figure 2.10: OPC UA meta model

The address space model defined in [UA3] is the meta model of OPC UA. The base concept of the meta model is a node. Several node classes are defined specializing the base node class (see Figure 2.10). Each node has a fixed set of attributes depending on the node class. Some attributes are mandatory and some are optional. For example, each node class has a node id uniquely identifying the node while the description attribute is optional.

Relationships between nodes are realized by references. References are no nodes and do not contain any attributes, thus they are a very simple construct. However, each reference is associated to a reference type. Although the meta model already defines a reference type hierarchy and uses those references as inherent part of the meta model (e.g. for defining a type hierarchy), the reference hierarchy is extensible.

The specializations of the base node class represent different concepts of the meta model. An object is a simple node that is typed by an object type. The attributes of the node only contain data describing the object. However, objects are used to represent real-world objects, software objects, etc. These data are stored in several variables referenced by the object. A variable has

a special attribute called value containing the data. Like objects, variables have types, called variable types

Unlike all other attributes, the value attribute has no data type assigned to it. The data type may differ for different variables and therefore each variable points to a data type node representing a data type. Data types are extensible, i.e. each server can define additional data types. Method nodes represent methods in the address space. They contain information on how to call the method (input parameters) and what will be returned (output parameters). View nodes represent an excerpt of the address space. A view typically restricts the data to the needs of a special user group or task and hides unnecessary data. Clients can browse through the address space in the context of a view.

In summary, the OPC UA meta model allows to define an information model by defining object, variable and data types as well as reference types. The specification already defines the base information model in [UA5] already containing several base types. Vendors can extend this model to create their own information model.

OPC UA is strictly related to the new trends of Industrial Internet of Things and Industry 4.0 and all the other topics discussed so far as it provides an efficient communication protocol for all the devices that may need to exchange information in an industrial environment.

2.3 Time Sensitive Networks

Standard	Area of Definition	Title of Standard
IEEE 802.1ASrev, IEEE 1588	Timing and synchronization	Enhancements and performance improvements
IEEE 802.1Qbu and IEEE 802.3br	Forwarding and queuing	Frame preemption
IEEE 802.1Qbv	Forwarding and queuing	Enhancements for scheduled traffic
IEEE 802.1Qca	Path control and reservation	Path control and reservation
IEEE 802.1Qcc	Central configuration method	Enhancements and performance improvements
IEEE 802.1Qci	Time-based ingress policing	Per-stream filtering and policing
IEEE 802.1CB	Seamless redundancy	Frame replication and elimination for reliability

Figure 2.11: Main TSN standards

The third section is finally dedicated to a proper and detailed introduction to Time Sensitive Networks.

By "TSN" we denote a set of network standards being developed by the Institute of Electrical and Electronic Engineers. These standards describe a series of mechanisms and strategies, that can be implemented in a communication network, whose goal is to gain some deterministic control and organization capabilities over the traffic flows in the network itself.

With particular reference to the ISO/OSI model discussed in section 2.2 (see also Figure 2.3), these rules are related to the layer 2, i.e. the network layer.

Fig. 2.11 shows the main standards involved. The second and third columns of the table briefly present the topic of the standard. It can be seen that these mechanisms are quite different one with the others and involve as many different aspects of network communication. It must also be noted that most of the standards are still under direct development by the IEEE organization, and therefore they are likely to be subject to various updates in the future as the current versions are not the final ones.

As we mentioned, the final purpose of a time sensitive network is to provide a higher degree of determinism about the delivery times of certain messages, without changing too much the topology or the features of the network. This means that, by adding some higher level mechanisms and by properly configuring them, we are able to guarantee the maximum value of the latency of the time sensitive packets. Another important property is that we can choose a priori which packets are going to be subject to said constraints and which ones are going to be free, with a perfect integration of the two in one single network.

The standards shown in Fig. 2.11 are the rigorous way to approach the topic, but conceptually we can group them in three big key elements: time synchronization, traffic scheduling and system configuration.

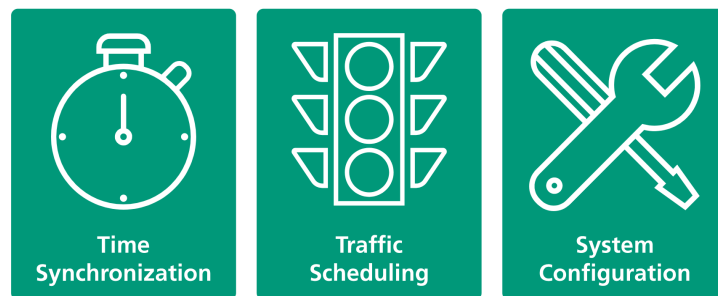


Figure 2.12: Key elements of TSN

Later in this section we are going to address these three elements, along with the related standards and solutions, providing a general idea of what a time sensitive enabled network features and how it works and can be configured.

We are not going to enter in the details about the actual implementation,

as they mostly depend on the hardware and software platforms that host the time sensitive network; furthermore these details will be addressed in a complete way for our configuration in the chapter dedicated to the practical tests.

2.3.1 Time synchronization

Time synchronization is one of the main building blocks of a working time sensitive network.

All the elements of a network, and in general all the devices that are able to receive and send messages, are equipped with at least one clock, in order to be able to measure the passing of time. The problem is that, even if all the clocks start at the exact same time, an inevitable drifting phenomenon will occur; namely that, due some inaccuracies in the measurements, relative errors between the clocks will be generated and added up, until the measurement offset becomes visible and potentially harmful.

Since what we are trying to build is a time sensitive network with a high temporal accuracy required, it only makes sense that the nodes of our network need to be synchronized in order to work properly. The exact reasons why this is necessary are going to be more clear after the next subsection, where we are going to see how we benefit from the fact that the clock share the same concept of time.

The idea of time synchronization, and in particular of distributed clocks in a connected system, is not a new feature of TSN. Standards and protocols already existed, which provided the same service that we need.

One simple example is the Network Time Protocol (NTP). Active from the mid-eighties, it allows clock synchronization between computer systems over packet-switched, variable-latency data networks. The clocks that employ this protocol generally have an offset of a few hundreds of milliseconds with respect to the Coordinated Universal Time (UTC).

Even though NTP is quite simple and already implemented, it cannot be used in a time sensitive framework, and the reason is obviously its inaccuracy. As we said the mean value of the offset is about some hundreds of milliseconds, which is far too high for our purposes; the jitter of this delay is also a problem, as it is extremely variable and depends on several factors, and we can never be really sure about the maximum value of the offset itself. In addition, there is no need to be synchronized with the the world reference clock (UTC in this case): for a restricted set of devices such as the one we find in a production plant, the reference clock could be anyone, and whether this reference clock is in sync with the UTC does not really matter.

Approaching our solution, the IEEE 1588 standard describes the Precision Time Protocol (PTP), a precursor of the one actually implemented in time sensitive networks. Its main drawback is that it does not feature that level of interoperability that is requested in the new standards.

So, with reference to Fig. 2.11, we are going to describe the IEEE 802.1 As-Rev and its gPTP (generalized Precision Time Protocol). As we said, and as it is also reported in the table, the standard is strictly related to the previous IEEE 1588.

The first important feature of this protocol is the so-called Best Master Clock Algorithm. It is a routine executed at the start, when the protocol is invoked, which allows to select between the available devices the one with the best characteristics to be the master, i.e. the reference clock in the system; all the other clocks, as a consequence, will be slaves. Each clock will send a message to the network to detect other clocks, and then perform a data set comparison. This compares data strings from each device and determines which clock is best to maintain the timing network. Usually the relevant parameters that are compared are the quality of the time source, namely its accuracy and jitter, and the position of the clock; it is best in fact to select a clock that is physically put in the middle of the network, so as to be able to reach everyone of the slaves virtually in similar times.

Of course it is possible to choose manually the master clock, but this routine provides a way to automatize the choice, also with generally better results in terms of performances.

The BMCA has also some fault tolerance features as, if the current best master happens to have a fault and to not be available anymore, it can be executed again to find the second best master clock and to avoid the scenario where the network lacks a reference time.

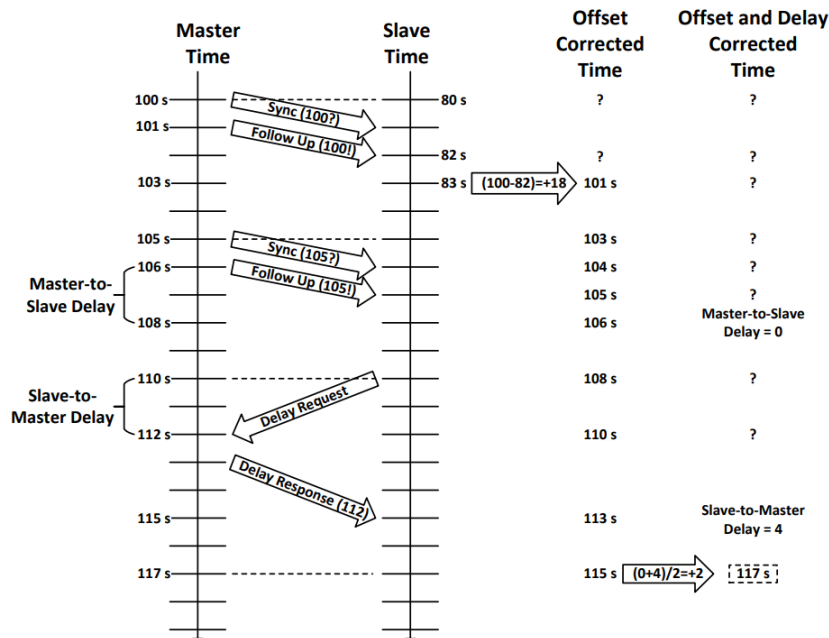


Figure 2.13: Principle scheme of PTP

Figure 2.12 shows how gPTP works. The grandmaster clock periodically issues a sync packet containing a timestamp reflecting when the packet left the grandmaster clock. The grandmaster may also issue a follow up packet containing the timestamp for the sync packet. The use of a separate follow up packet allows the grandmaster to accurately timestamp the sync packet on networks where the departure time of a packet cannot be known accurately in the initial sync message. Clocks that use a separate follow up packet are known as two-step clocks. Some master clocks, known as one-step clocks, can provide an accurate timestamp in the sync message itself, avoiding the need for a second follow-up message.

A slave clock receives the grandmaster's sync packet and timestamps the packet's arrival time using its own clock. The difference in the sync packet's departure timestamp and the sync packet's arrival timestamp is the combination of the slave clock's offset from the master and the network propagation delay. By adjusting its clock by the offset measured at this point, the slave clock can reduce the time difference between it and the master to the network propagation delay only.

Under the assumption that propagation delay is symmetrical, the slave clock can compute it and compensate for it. This is done by issuing a delay request packet that is timestamped on departure from the slave. The master clock receives and timestamps this delay request packet, and the arrival timestamp is sent back to the slave clock in a delay response packet. The difference between these two timestamps is used to calculate the network propagation delay. This delay request-response mechanism can measure the end-to-end path delay between the master and slave clocks and can operate over paths that include non-time aware switches.

IEEE 802.1 As-Rev introduces support for multiple concurrent timescales and other fault tolerant features, which make it the most advanced standard for industrial operations.

As one may guess, the performances suffer some degradation effects as the network becomes bigger and bigger. Currently the best guaranteed performance that can be achieved is an offset lower than 1 microsecond, provided that the maximum distance from the master clock to any slave is not greater than seven nodes. This makes sense, as the accuracy of the timestamps and the delay requests inevitably decreases if the synchronization packets have to travel through lots of nodes before arriving at the destination. However, for limited-sized networks, a 1 microsecond delay is more than acceptable to perform some real time tasks successfully.

2.3.2 Time aware shaper

The most important and fundamental mechanism that Time Sensitive Networks are based upon is the so-called Time Aware Shaper (TAS).

TAS is a kind of mask for the output interface of a single port of any

device in the system. It allows us to control the output packet that will use the communication link with a timely precision and some other features.

The 802.1Qbv standard describes this mechanism and other features connected with it. Figure 2.13 shows the principle scheme of a Time Aware Shaper.

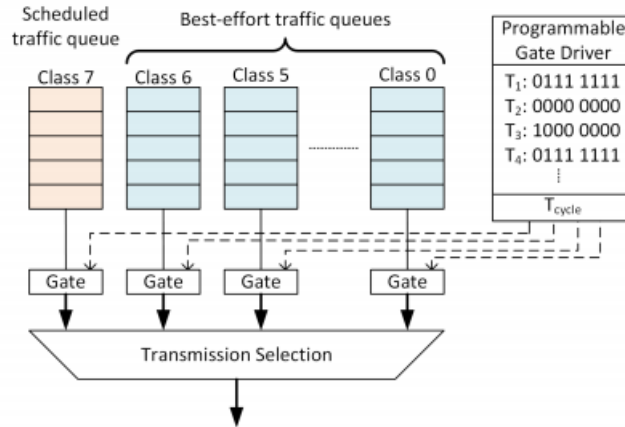


Figure 2.14: Time Aware Shaper

The Time Aware Shaper basically consists in a list of named queues, all connected to the transmission selection block that actually sends the packets into the communication medium. Between the output of each queue and the transmission selection block there is a gate, namely a controlled mechanism that allows to block the packets or let them pass.

The packets that need to be sent out through the port implementing TAS are first enqueued in one of the named queues. In particular, the PCP (a field of the VLAN identifier present in the Ethernet header) is inspected in order to figure out which queue is to be assigned to the packet. The queues are in fact divided in classes, each one connected to a particular value of the PCP field so that all the packets reporting the same number get enqueued in the same class. Queue classes generally are made by just one queue, but it is possible for multiple queues to be assigned to the same class.

In the example we have 8 queue classes, from class 0 to class 7, each one linked with a specific queue.

Queues generally work following a First In First Out model. This means that when a packet arrives at the queue it gets past the queue only if no one else is currently enqueued; otherwise it is put in the last position of the queue, with the the packets arrived before it having a higher exit priority.

The gates connecting the outputs of the queues to the transmission selection block are controlled by means of the Gate Driver, which has the duty of opening or closing the access. The Gate Driver is in turn driven by a

so-called Gate Control List. As we can appreciate from the figure, the GCL is a table containing all the relevant information needed to drive the gates: every line defines a time window, and it is characterized by a duration and by a binary number. The duration indicates the time after which the line has to be updated with the next one in the list, while the binary number represents a bitmask of the combination of open and closed gates for the queues. Usually the GCL has a limited number of lines, that are cyclically repeated; therefore the summation of the durations of the time windows gives the cycle time of the whole operation.

In the example of Fig. 2.13 we have four time windows, each one characterized by its own duration, from T_1 up to T_4 . The bitmask of the first entry of the GCL has an integer value of 127, but that's not relevant: what's important is the combination of zeros and ones whose purpose is to tell to the Gate Driver which gates to close and which gates to open. Specifically, from the initial moment of the cycle, for a time offset of T_1 , all the gates but the one associated with the class 7 queue will be open, while the last one will be closed. This because the bit number seven has value zero, while bits from zero to six have value one.

Similarly, when the second window is activated, namely from time T_1 to time T_1+T_2 after the cycle's start, no queue will be open. This means that every packet in every queue will have to wait this whole window without being transmitted. Even though this window does not make much sense for now, the GCL represented in the example can actually be employed in a time sensitive network; the reason why it is important to have this pause between two windows will be more clear later.

Eventually we have one window of duration T_3 where only queue 7 is open and the last window of duration T_4 equal to the first one.

As we said, this TAS just described is the main tool to be used in a time sensitive network because it has some interesting properties. The first property is that we can separate packets in different queues by means of their PCP field. As it is also depicted in Fig. 2.13, the natural distinction would be between time sensitive and best effort packets.

Recall that we have two main kinds of data flowing in the network. Some of them have deadlines concerning their delivery time that have to be met, while for the others it is not vital to respect any particular constraint, but nonetheless it is important to try and achieve the best possible performance. We call the first kind of data TSN, while the second kind is named best effort.

So the obvious thing we could do would be to tag in different ways the TSN packets and best effort packets in order for them to be sorted in different queues. Then, by correctly switching the gate configuration we would be able to decide whether to send as output a TSN packet or a best effort one.

That's pretty much the main idea behind time sensitive networks. Let's go back and take another look at the schedule in Fig. 2.13, considering now

that queue 7 is dedicated to TSN data while all the others are dedicated to best effort data. Analyzing again the GCL we can see that the third window is completely dedicated to TSN, while the first and the fourth are dedicated to best effort data. Clearly it is not possible to send seven packets at the same time in the communication link, therefore another mechanism will have to be employed in order to choose the most suitable packet, but that's not relevant right now. In this way we managed to successfully separate TSN packets from the best effort ones, which can also be seen as a disturbance, and to dedicate one transmission window to the transmission of just TSN.

This is not trivial at all, especially if we consider the rates and speeds involved in this kinds of processes. Usually, even with multiple queues like in this case, we would have lots of best effort packets mixed with a few TSN packets, most likely distributed with a random pattern among the queues. In this scenario, even if we precisely knew the position of each packet in the queues, it would be very tough to come up with some strategy that grants us similar levels of control without completely changing the system.

Let's now consider the second window, when all the gates are closed and no packet is allowed to leave its queue. This window is called Guard Band, and it is useful to maximize the performances in terms of jitter, at the cost of a little waste in the bandwidth. Current output interfaces in fact do not support preemption of packets. This means that if a packet has already started its transmission, the source cannot stop it, and continue later, until it is concluded. Preemption would be quite complex to implement and has to be supported both on the sending and receiving ends; devices must be aware that there is a chance that the packet they are receiving can be interrupted and resumed later. An easier alternative to preemption is blocking every possible packet before a TSN window, so as to be sure that no packet is transmitting at the start of the window. This way we are sure that TSN packets will leave the devices right at the start of their associated window, if they are available, without the need to abruptly truncate the previous best effort packet, which would certainly cause it to be discarded. Obviously the drawback is that we lose something in terms of available bandwidth, as we are forbidding the system to send anything during that window.

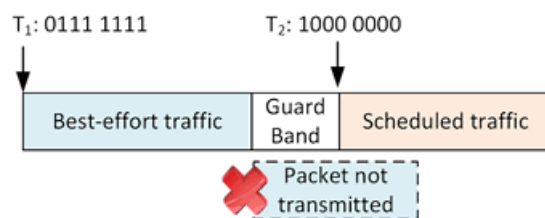


Figure 2.15: The use of a Guard Band

Ethernet frames have a maximum dimension in terms of bytes that they

cannot exceed. This dimension is 1500 bytes plus 26 bytes for the header, for a total of 1526 bytes, i.e. 12208 bits. Then, in order to be sure that every previously transmitted packet has finished its transmission even in the worst scenario, we arrange Guard Bands 12208 bits long: namely their duration is the time needed to send in the link that number of bits, which of course depends on the transmission capacity of the interface. For instance, in a system with 100 Mb/s capacity, the duration of the Guard Bands will be slightly higher than 120 microseconds, while for 1 Gb/s systems it will be ten times lower.

It is clear that the more the cycle is filled with TSN windows, the more we are going to need guardbands and so the more bandwidth we are going to waste. Also, the switching itself between a configuration and another one requires computational overhead for the system, that is not free, i.e. some bandwidth is going to be wasted anyway. Therefore particular attention must be paid in the generation of the GCL, in order to avoid this from happening; for instance it would be good practice trying to organize the traffic so that all the TSN packets pass in one single window, rather than with sparse windows throughout the whole cycle time.

In a time sensitive network every device, hosts and routers, is supposed to implement a Time Aware Shaper on every output port; or at least on every port in which we want both TSN and best effort packets to flow. If for some output ports we do not implement a TAS, those ports will only be used to carry best effort data since, as we have said, we couldn't guarantee the delivery times.

Therefore we have also to assign an appropriate GCL to every port of every device. The operation of producing a suitable GCL, that from now on will be called schedule, for every node in the system is called scheduling. The scheduling problem is a quite complex topic, which will be detailed in section 2.4; here we are going to give a simple idea of what it means and what are the main issues.

Let's consider a generic network where, among the other nodes, a talking host sends packets to a listening host; packets have to flow through one router before arriving to their destination. The schedules we are interested in are the one of the talking device and the one for the output port of the router. Following the argument presented about the guardbands, we can draft two schedules similar to the one in Fig. 2.14, namely with one dedicated window for TSN packets preceded by a guardband, and best effort windows elsewhere. We cannot, though, pick the exact same schedules for both the devices: we first need to consider the talking host and define a suitable window that is activated when the TSN packets are available; then we have to take into account the transmission delay and, therefore, activate the TSN window of the router from the moment the packets get to the router until the moment that they have all left. Once we know the offset and duration of the TSN windows for both the schedules, we can compute the appropriate

offsets for the guardbands and compute accordingly the offsets for the best effort windows.

Clearly the matter gets more and more complicated if we consider more nodes or more complex patterns of packets; it is also important to have extreme accuracy in the computation of the trajectories and of the related activation times for the windows, in order to fully exploit the control capability provided by the TAS and so to maximize the performances.

It is also worth noting that, in order to be able to compute an effective schedule, all the features of the TSN packets have to be known a priori: frequency, path, dimensions and so on.

This example is useful to give the idea that the scheduling problem for a time sensitive network is a complex task, not conceptually but rather from an implementative point of view. This problem is the subject of many studies and academic papers, and typically its solution involves optimization methods and algorithms belonging to the operative research field.

At this point we can also see why it is so important that all the devices in the network are correctly synchronized, namely why we need Precision Time Protocol. The Time Aware Shaper, as the name itself says, is a tool that strongly relies on the measure of the time that the device on which it is implemented has. In particular, the coordination with the other devices in the system is the element that ultimately allows TSN packets to meet their assigned deadlines. If, for instance, the clock of a node along the path has a delay of even some fraction of a millisecond with respect to the master clock, TSN packets will arrive during the guardband window or even before, and as a consequence they will be constrained to wait unnecessarily. This would in fact reduce the efficiency of our network.

2.3.3 System's configuration

The last important feature of a time sensitive network is a generalized and centralized configuration system. As we have seen, time sensitive networks require a little bit more of effort for the configuration step with respect to a classical network: TSN flows need to be registered, a proper schedule must be computed and distributed to all the nodes, which have to behave according to the time aware shaper model.

The standard IEEE 802.1 Qcc deals with this problem and presents some mechanisms related to its solution. The idea is to rely upon a centralized tool that implements in software the configuration steps needed to get the network ready. The following figure shows the main acting elements.

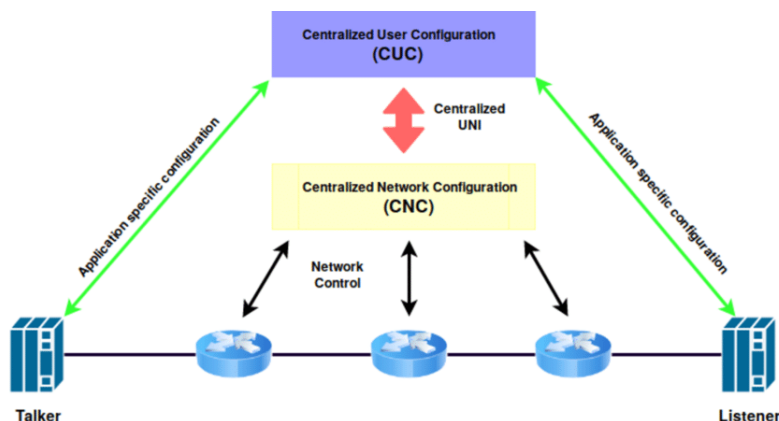


Figure 2.16: System's configuration

The standards define two application programs that run in the system: the Centralized User Configuration (CUC) and the Centralized Network Configuration or Controller (CNC).

As the name suggests, the CNC controls the TSN bridges in the network. The CNC is an application that can be run on any hardware and has two primary responsibilities. First, it is responsible for determining routes and scheduling the TSN flows through the bridged network. Second, it is responsible for configuring the TSN bridges for TSN operation.

On the other hand, CUC is more of an interface of the system with the user, and it is run on the end devices rather than on the core devices in the network. The CNC communicates with the CUC to receive the communications requirements that the network must provide. The CNC considers all the requests, computes the route for each communication flow, schedules the end-to-end transmission for each TSN flow, and finally uploads the computed schedule to each TSN bridge. As part of the schedule computation, the CNC provides a unique identifier for each TSN flow. This unique identifier is used by the TSN bridges to differentiate one TSN flow from another. The unique identifier includes the destination MAC address and the VLAN ID and the class. With these three items, the TSN bridges can identify the TSN flow and transmit the flow according to the correct schedule.

These two elements are supposed to be a simple and integrated interface for the user to interact with. By means of their cooperation the setup of the network can be automatized as much as possible, without the need for the end user to know the details.

The presentation made by Cisco [7] in 2017 describes the typical workflow in setting up a time sensitive network. First of all, through the tools CUC and CNC, we need to issue a request to invoke the Link Layer Discovery Protocol, in order for the network to its configuration with all the nodes and connections.

Then the engineer has to register in the CUC the details of the TSN flow that he's interested in: dimension, frequency, path and so on. At this point the CNC will compute a suitable schedule capable of meeting the requirements requested by the engineer concerning the TSN flows. Note that this step may fail as there may not exist a feasible solution for the scheduling problem.

Stream ID	Destination MAC	Size	Period	Talker Name	CoS	VLAN	Talker Transmit Window	Receiver Name	Receiver Window	Transmission Duration
Flow 1	03:00:5E:A0:03:e9	64	1ms	tsn-TL1	5	3000	350-363µs	tsn-TL2	463-476µs	100µs
Flow 2	03:00:5E:A0:03:eA	64	1ms	tsn-TL2	5	3000	850-863µs	tsn-TL1	963-976µs	100µs

Figure 2.17: The computed schedule

The user may also want to see the computed schedule before starting the network. Figure 2.17 shows the result with all the relevant information concerning the two TSN flows, 1 and 2.

Eventually, if the expected performances are satisfactory, the schedule will be distributed and the TSN packets will start to flow in the network.

What has just been presented is the ideal goal described in the standards. In practice the situation is quite different as a general, vendor-independent system configuration tool does not exist yet. Commercial solutions do exist, but are clearly limited as they do not offer that level of interoperability and flexibility seeked in the standards. They do furthermore depend on the specific hardware provided by the individual producers (Cisco, for example).

2.3.4 Credit-based shaping

So far we have seen that the Time Aware Shaper is the main mechanism to be used in order to control the output flow of a port in a more deterministic way.

It is not mandatory, though, to employ only the TAS in our output interfaces. Let's consider for instance the example in Fig. 2.14: during the first and the fourth time windows, seven queues are open at the same time, but it is not possible for seven different packets to be sent in the communication medium simultaneously. Therefore an additional mechanism, able to choose the most suitable packet in these kinds of situations, is needed.

In this subsection we are going to describe the Credit-based shaper, which can be such a tool, even if it is more related to the original concept of time sensitive networks rather than a current optimal solution.

The idea of time sensitive networks wasn't born as it is, but it is derived from what is called Audio/Video bridging (AVB). The first developers weren't interested in obtaining bounded and deterministic delivery times,

but rather to get an effective synchronization on the transmission of audio and video data over communication networks.

The problem was that, when transmitting a video over internet for example, the end receiver needed to receive audio packets along with their corresponding video packets in order to be able to reconstruct the media with the perfect synchronization between the two tracks. It was therefore important to avoid bursts of consecutive packets all belonging to the same class, and rather to get a good mix of the two types. Then, the first attempts to build traffic shaping mechanisms in the output interface of the network's nodes were made, leading to what is now the Time Aware Shaper.

Even now, in some documents, TSN and AVB are two interchangeable acronyms, like for instance in the datasheet of components such as the Network Interface Controller, in order to indicate or not the hardware's support for that technology.

As we said, Credit-based shaping is a possible solution to the AVB problem because it exploits dynamic priorities; its reference standard is IEEE 802.1 Qav. We define two traffic classes, typically named class A and class B, and we assign to each one of them a different output queue. Note that this mechanism can be used alone or in parallel with another one, e.g. in two of the best effort queues of the TAS in Fig. 2.14. Each class has a dynamic priority parameter that is called credit; since it is a priority, every time that the parameter of A or B is positive and greater than the one of the other class, its packet is eligible for transmission if no one else is currently transmitting.

Queues' priorities are dynamic, therefore change in time. In particular we also need to define two extra parameters for each class: a *sendSlope* and a *idleSlope*. Then, as the names suggest, the credit of each class will be decreased with a rate equal to *sendSlope* whenever packets from its queue are occupying the communication medium, while it will be increased with a rate *idleSlope* when another queue is transmitting. Also, whenever a queue is empty, its credit is reset to zero.

So the idea is quite simple: we are basically using a negative feedback in order to obtain balance in the sending rates of the two queues, by lowering the priority as long as a queue is transmitting. Therefore there won't exist a queue which always has the highest priority, but they will swap this role. The following figure shows one example of the behavior of the credit for one queue, along with the number of packets in the queue and the transmission permissions.

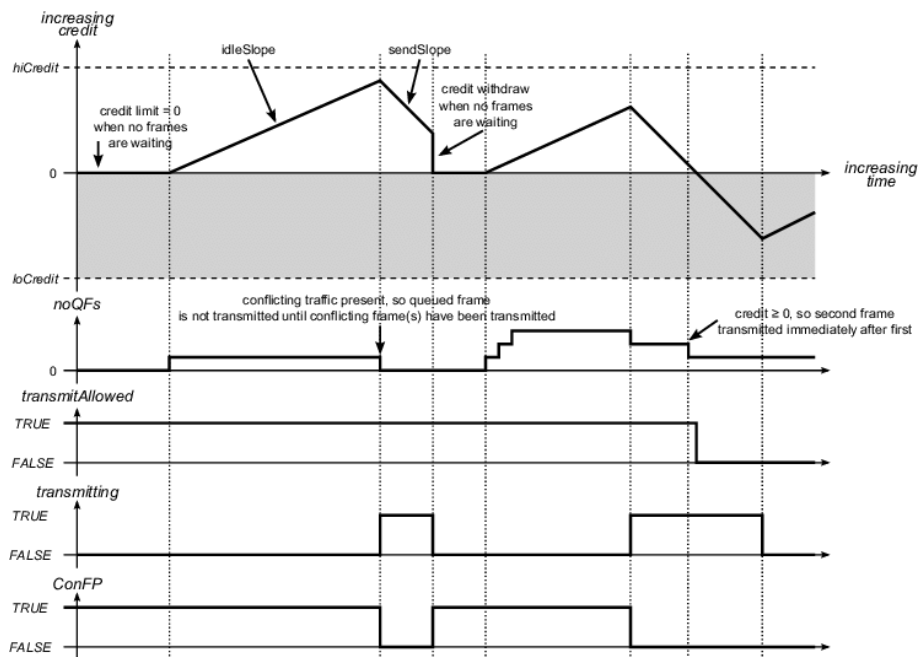


Figure 2.18: Example of CBS for one queue

The four slope parameters for the two queues are usually dimensioned according to the available bandwidth at the port. The equation is the following

$$\alpha_{send,A} + \alpha_{idle,A} = BW \quad (2.3)$$

We have one degree of freedom on one parameter, and then the other is automatically defined according to eq. 2.3. Furthermore, for queue B, the slopes are swapped, namely the idleSlope is the sendSlope of queue A and so on.

As a result we are able, by employing this mechanism, to avoid bursts of packets belonging to the same class, which allows us to obtain a good synchronization between the flows. In Audio Video Bridging this is extremely useful and solves the problem of long transmissions on a single medium.

For time sensitive applications, though, Credit-based Shaping is not the most suitable mechanism, as it is not possible to exactly control the output flow of a port. So it has been presented not as an alternative to the TAS, but rather as an example of another traffic shaping mechanism that can possibly be used in parallel with the Time Aware Shaper. It has also been useful to the initial developments of proper Time Sensitive Networks.

Lastly, the next chapter will present some tests about the Credit-based shaper.

2.4 The scheduling problem

In the previous parts of this chapter we have listed the main features of a Time Sensitive Network described in the standards: time synchronization, gating mechanisms and system's configuration. However, these three key features only are not sufficient for a time sensitive network to be properly working. The missing element is indeed a suitable schedule.

As we have seen, so far we have described the basic tools that allow us to completely control the traffic patterns in a network; how to use these tools may be clear for an easy and simple network, but a general and effective technique to handle all possible scenarios hasn't been provided.

This because TSN standards themselves do not include sections dedicated to a specific method to solve this problem, leaving its solution for the great part to the users or to the vendors of TSN technology. That is also the reason why this topic is treated in a different section.

By definition, the scheduling problem has the goal of producing a suitable Gate Control List for every output port in the network so that the final deadlines for TSN packets are met each and every cycle. Recall from section 2.3.2 that a Gate Control List is made by a number of lines equal to the fixed number of windows in a period, and each line contains the duration and the combination of open and closed gates for that window. The input data for this problem are the network's features such as its topology, number of nodes, transmission rates, computation times, ad so on and so forth; also, the complete set of TSN data has to be known a priori, with each packet generally characterized by a source, a destination, a size and a period.

Typically the solution of this problem is computed offline, before starting the operations of the network; then, by means of the configuration tools that we have seen in the related subsection, the schedule is distributed among the nodes and becomes effective until another updated schedule is produced.

So basically, like every other scheduling, our problem is a resource allocation problem: we have a limited set of resources, (i.e. nodes and cables in our network) and a set of tasks that need those resources. Each task needs to use a subset of resources for a certain amount of time, one at a time in a fixed order, and occupies them so as no one else can use them when it is using them. Since the resources are limited, in general we cannot satisfy immediately all the requests, but we have to come up with strategies aimed at a correct ordering of the tasks. In the end, a solution is acceptable if the deadlines of every TSN packet are met, namely if every packet manages to reach its destination within its maximum latency. Furthermore, among the possibly many acceptable solutions, it is common practice to select a criterion which allows us to choose the best solution that minimizes or maximizes our evaluation criterion; this allows to make a more reasoned choice and usually to obtain better performances. Therefore scheduling problems can be found, most of the time, as optimization problems.

A common example of scheduling problem is the one concerning the processes that need to be run on the CPU, especially in a real time operating system, which is a common feature in the automation world: every process needs to perform some computations within a short period of time, and the amount of operations that the CPU can execute is limited. So the scheduler has to compute the precise order in which processes get to use the CPU, similarly to how packets use the Ethernet cables in our framework.

The scheduling of a Time Sensitive Network, like any other scheduling problem, is classified as a NP-hard problem, which implies that the time complexity of an algorithm is not polynomial. This means that the time required from a computer, to compute an acceptable (maybe optimum) solution, increases exponentially with the number of tasks and in general with the complexity of the problem. This, in addition to the variety of possible ways to approach a solution, is probably the main reason why the topic is not described in the standards. Instead it is widely treated in the academic world, where numerous papers and work do exist which go in the deep of such an optimization problem.

For this thesis project several scheduling algorithms have been studied, and two of them are going to be described in the next paragraphs. Furthermore, a large part of the chapter concerning to the simulation is dedicated to the implementation and evaluation of a scheduling algorithm.

One last thing worth mentioning is the fact that with the term scheduling we actually mean two separate "programs", i.e. the actual scheduling of the gate controllers and the routing. Routing is the set of rules that the packets follow in order to correctly reach their destination: usually a routing algorithm gives an ordered set of nodes which compose the path for a given packet towards its destination. The most common kind of routing algorithm gives the shortest path between two any given nodes in a connected network (Dijkstra's is a famous example), but there are also other kinds of algorithms that aim at minimizing the congestion, e.g. redirecting some packets through less congested and possibly longer paths.

So scheduling and routing can become a big, unique problem, which belongs to a specific class of scheduling problems. These problems are way more complex with respect to the two individual problems considered separately but allow for more degrees of freedom in the solutions. The other class of scheduling problem is instead composed by the routing and scheduling problems solved separately, which means that the computational load is lighter but the solution space is restricted with respect to the previous case.

We conclude this presentation part about scheduling with what we expect to have as a solution of the problem and some considerations about it. The structure of the schedule that we expect to find will probably be similar to the one presented in Fig. 2.14, as it has been highlighted. Our target schedule will feature some TSN windows, and each one is supposed to be preceded by a so-called guardband (with all the gates closed) in order to

guarantee the property that the transmission medium is free at the start of the TSN window. So we will have at least three windows: one for TSN data, one guardband and the remaining for best effort packets. In general, the number of TSN windows will depend on the characteristics of the problem but, even if the schedules had the same structure, the offset and duration of each window would be very different and specific to the port. Also, for reasons already widely explained, the number of needed guardbands will affect the overall throughput of our system; therefore in the optimization problem some additional constraint is necessary to guarantee that TSN windows get scheduled as close as possible, in order to increase the throughput.

We proceed now to the description of two scheduling algorithms, namely the ILP-Based Joint Routing and Scheduling algorithm [13] and the No-wait packet scheduling algorithm [14].

2.4.1 ILP-Based Joint Routing and Scheduling

As we said earlier, this is an optimization problem that aims at solving at the same time both the scheduling and the routing problems for a time sensitive network.

First of all we need to name the set of inputs for our problem. We define the following entities:

- V , the set of nodes in the network. Some of them are going to be connected one with the other.
- E is the set of edges that represent the connections between nodes; if node i and node j have a connections then there will be an element in E which is (i,j)
- D is the set of swithing delay information. It contains the delays that packets suffer from their traveling through the network
- F is the set of real time tasks. If we assume that it is made by a number K of tasks, each task f_k belonging to F will be characterized by:
 - s_k , the source of the flow
 - d_k , the destination of the flow
 - ct_k , the cycle time of the flow
 - rsl_k , the required slot lenght, namely the duration of its TSN windows necessary to let the packets flow
 - ml_k , the maxmimum latency for the packets allowed by the application

Then we need to express the needs of our schedule as constraints for the optimization problem. There are four kinds of constraints that we are

going to take into account: routing constraints, scheduling along a path constraints, resource constraints and application constraints.

We start with the routing constraints and we define a help binary variable x_{ijk} , which is equal to one if the flow k uses the edge (i,j) on its path, and it is equal to zero otherwise. Using this variable we express the routing constraints, namely that packets need to be generated in their sources and travel through the network towards their destination

$$\forall f_k \in F : \quad \sum_{j \in V | (s_k, j) \in E} x_{s_k, jk} - \sum_{j \in V | (j, s_k) \in E} x_{j s_k k} = 1 \quad (2.4)$$

$$\forall f_k \in F, \forall i \in V - [s_k, d_k] : \quad \sum_{j \in V | (i, j) \in E} x_{ijk} - \sum_{j \in V | (j, i) \in E} x_{jik} = 0 \quad (2.5)$$

$$\forall f_k \in F, \forall i \in V : \quad \sum_{j \in V | (i, j) \in E} x_{ijk} \leq 1 \quad (2.6)$$

The last equation expresses the constraint that a flow shall pass through one link at most once in a period.

Later we define the constraints of the scheduling along a path. We define here what we call scheduling variables:

- t_{ijk} is an integer which defines the beginning of a time slot on link (i,j) of flow f_k ; it belongs to the interval $[0; ct_k - 1]$
- o_{ijk} is a positive integer variable that states an offset of a time slot as a number of full cycles of length ct_k

In this way we can express the position of a time slot on link (i,j) as $t_{ijk} + o_{ijk} * ct_k$. As a consequence we have:

$$\forall f_k \in F, \forall (i, j) \in E : \quad t_{ijk} + o_{ijk} \leq x_{ijk} * M \quad (2.7)$$

$$\begin{aligned} \forall f_k \in F, \forall i \in V - [s_k, d_k] : \\ \sum_{j \in V | (i, j) \in E} (t_{ijk} + o_{ijk} * ct_k) - \sum_{j \in V | (j, i) \in E} (t_{jik} + o_{jik} * ct_k) \\ \geq msd_i * \sum_{j \in V | (i, j) \in E} x_{ijk} \end{aligned} \quad (2.8)$$

M in the first equation is a big arbitrary constant, and it is useful to put the scheduling variables to zero if the flow does not use link (i,j) . The second equation, instead, expresses the constraint that the scheduled windows on two consecutive nodes along the path do have to take into account the worst case delay of the link that connects them, namely msd_i .

The resource constraints have to make sure that any two scheduled TSN windows do not overlap one with the other, granting that we do not require from the system any more resources than it has. We define the binary variable a_{ijkluv}

$$\begin{aligned}
& \forall (f_k, f_l) \in F \times F | l > k, \forall (i, j) \in E \\
& \forall (u, v) \in \left\{ u \in \mathbb{N} | u \leq \frac{\text{lcm}(ct_k, ct_l)}{ct_k} \right\} \times \left\{ v \in \mathbb{N} | v \leq \frac{\text{lcm}(ct_k, ct_l)}{ct_l} \right\} \\
& (t_{ijl} + v * ct_l) - (t_{ijk} + u * ct_k) \\
& \geq rsl_k - M * (3 - a_{ijkluv} - x_{ijk} - x_{ijl}) \\
& (t_{ijk} + u * ct_k) - (t_{ijl} + v * ct_l) \\
& \geq rsl_l - M * (2 + a_{ijkluv} - x_{ijk} - x_{ijl})
\end{aligned} \tag{2.9}$$

The binary variable is useful to activate one constraint or the other. In particular, we are imposing that for any two given windows on the same port, either the first ends before the start of the second or the other way around, therefore without any overlapping.

The last set of constraints concerns the deadlines imposed by the application to the TSN flows.

$$\begin{aligned}
& \forall f_k \in F : \\
& \sum_{j \in V | (i, d_k) \in E} (t_{jd_kk} + o_{jd_kk} * ct_k) - \sum_{j \in V | (s_k, j) \in E} (t_{s_kjk} + o_{s_kjk} * ct_k) \\
& \leq ml_k - rls_k
\end{aligned} \tag{2.10}$$

Equation 2.10 limits the latency of the actual transmission of packets with the constraint imposed by the application.

Eventually we need to provide a suitable cost, namely a scalar function, that depends on the parameters of our solutions, which is going to be minimized.

$$\begin{aligned}
& \min \sum_{f_k \in F} \left(\sum_{j \in V | (j, d_k) \in E} (t_{jd_kk} + o_{jd_kk} * ct_k) - \sum_{j \in V | (s_k, j) \in E} (t_{s_kjk} + o_{s_kjk} * ct_k) \right)
\end{aligned} \tag{2.11}$$

The cost function of equation 2.11 is the sum of the latencies of all the TSN flows. Reducing the latency is good practice as it introduces some fault

tolerant properties in the system; as it may be intuitive, finishing the time sensitive tasks as soon and as quickly as possible is far better than having them circulating in the network for a long time, even if they meet the deadlines. Then this cost function can be updated with specific coefficients for the latency of each packet in order to weight them differently according to an arbitrary, and decided by the user, set of priorities.

To summarize, what we have just described is a proper integer linear program, namely an optimization problem which has a linear cost function and whose solution must be an integer. In particular, equations from 2.4 to 2.10 express the constraints of the problem which in turn represent the physical and logical characteristics of a scheduling problem; equation 2.11, instead, expresses the cost associated to each particular solution, which has to be minimized. Furthermore, while the constraints equations need to be kept pretty much the same, the cost function can be adapted and chosen according to the specific needs of the application; in general, changing the cost function means changing also the optimal solution that the algorithm will provide.

Since ILP problems are quite popular, specific software tools already exist whose purpose is just to solve efficiently these kinds of instances. In [13], in particular, the Gurobi software is used; Gurobi is a commercial optimization solver, able to work with integer linear programs, quadratic programs and other kinds of similar optimization problems. A brief and deeper explanation on how the software works was not provided, but the results concerning some test cases were shown. In particular comparisons were made with simpler problems or simpler versions of this one, e.g. with routing and scheduling solved separately: the results were in fact better with the presented joint routing and scheduling integer linear program, which was able to achieve lower latencies and higher network utilizations.

The main drawback of this approach, however, is that its overall capacity is quite limited and, in turn, the time and resources required to compute the optimal solution are very high. We mentioned that a scheduling problem is a NP-hard problem and therefore quite difficult to solve. In the paper a powerful computing unit with 128 GB of RAM was employed for over a couple of hours, just to compute the solutions for a very restricted set of tasks; the simulated scenario had up to 30 packets scheduled for each cycle. This makes sense as all the combinations of possible solutions are virtually infinite, but becomes quite limiting if we want to apply it to a practical scenario.

For these reasons we are now going to consider a different kind of solution for our problem, namely a heuristic algorithm, which does not grant an optimal solution but allows to solve more complex problems.

2.4.2 No-wait packet scheduling

This subsection is dedicated to the presentation of the no-wait packet scheduling algorithm [14] for the solution of the scheduling problem in a time sensitive network. We have seen that an optimal solutions could lead to excessive execution times and resources required; the presented algorithm, being a heuristic solution, is able to reduce the computational complexity and to make tractable and solvable larger problems with respect to the previous case.

The algorithm is actually derived from an already existing approach to some scheduling frameworks, which is the job shop scheduling. Job shop scheduling is a well known solution for the organization of jobs for the machines of a factory, because it has some interesting features that make it particularly suitable for this framework.

Recall that we have a known set of TSN packets which have to travel in the network and need to use the limited set of resources, namely switches and cables; the goal is to assign slices of time to each packet in each switch and cable in its path, so that it can successfully reach its destination within the end of the cycle.

Conceptually the algorithm is quite simple, and it is based upon two guiding principles that we need to follow while building the schedule: each packet has to be scheduled as soon as possible, and also it never has to wait along its path, as the name suggests. This means that, ideally, as soon as a packet is generated, it finds a free "hallway" that directly connects its source to its destination: this property is achieved by means of a correct positioning of TSN windows on all the nodes in its path.

The function that performs this task is called "TimeTabling". We start from an ordered sequence which contains all the TSN flows that we need to schedule; afterwards, starting from the beginning and then proceeding in order, we "place" every packet in the network assuming that it leaves the source right at the start of the period. By "placing a packet" we mean to schedule the appropriate TSN windows in each node of the path: these windows must have a duration necessary to let the whole packet pass, and so it will be proportional to the size; they also need to be delayed taking into account the transmission, propagation and computation times that will inevitably affect the latency of the packet. When a window in a port is dedicated to a certain packet, the port is occupied and therefore cannot be used by another packet. If, somewhere along the path, we find that at least one port is occupied and so the packet that we are placing would be constrained to wait, we need to restart the scheduling of the packet itself, assuming that it leaves the source a little while later. This because we need to satisfy the property that no packet is ever constrained to wait, for any reason. So we have to postpone the departure of packets as long as we do not find their path occupied by some previously placed TSN packets. Hence

the initial order that we use to place packets has a sort of priority role: the first packets to be placed will in fact be able to leave instantaneously, while the last ones will be, in general, constrained to leave after a while, because the network is already partially occupied by other instances. Once we have finished placing all the packets in the sequence, we can first of all check whether or not they all fit in the assigned period; if they do, we can also compute other two outputs for the timetabling algorithm which are the critical flow and the span. The critical flow is the one flow, among the list of TSN flows, which arrives at destination last; the span is in turn the arrival time of the critical flow.

Algorithm 1: TimeTablin algorithm

```

function TimeTabling(sequence);
for each task in sequence do
    start time = 0 ;
    while start time < cycle time do
        if every node in task's route is free then
            | put TSN windows;
        else
            | start time = start time + delta;
            | break;
        end
    end
end
critical flow = last flow to finish;
span = arrival time of critical flow;
generation = start time of each flow;
return critical flow, span, generation

```

The proposed no-wait packet scheduling algorithm consists in an iterative research, among all the possible sequences of tasks, of the one that minimizes the span. We have a function that, given one sequence, computes critical flow and span, namely the timetabling; what we have to do is iteratively change the input sequence of the timetabling to find the best possible span. In particular the paper describes a tabu search based on neighborhoods: we take the current solution and the current critical flow, and then we create a neighborhood of that solution by swapping every flow preceding the critical flow with the critical flow itself and by placing the critical flow before every flow preceding it in the current sequence. These two kinds of operations are named respectively *swapping* and *insertion*, and allow to obtain a neighborhood made by up to $2n-2$ sequences, where n stands for the number of tasks. Once we have a new neighborhood, we analyze every component of the neighborhood (by means of the timetabling) and select the one which has the best span as the current solution. The "tabu" part comes into play whenever we find a solution which has the same critical flow as the previous

ones; we have in fact to record in a list a certain number of flows, which correspond to the critical flows of the last current solutions. The proposed algorithm requires not to consider a solution as the current one if its critical flow belongs to the tabu list, unless it has the best span encountered so far. The dimension of the tabu list has to be decided by the user and it is a useful parameter to tune the algorithm.

Another parameter that the user needs to chose is the number of unsuccessful iterations. The algorithm is in fact programmed to stop if it reaches a certain number of consecutive neighborhoods visited without improving the currently best solution. This measure is adopted because searching all the possible sequences, namely all the permutations of n tasks, is virtually impossible.

Algorithm 2: No-wait packet scheduling algorithm

```

function NWPS(TaskSet, limit);
initial solution = randomOrdering(TaskSet);
[critical flow, span, generation]=TimeTabling(initial solution);
best solution = initial solution;
current solution = initial solution;
while insuccessful iterations < limit do
    neighborhood = generateNeigh(current solution, critical flow);
    for each solution in neighborhood do
        | [critical flow, span, generation]=TimeTabling(solution);
    end
    current solution = best in neighborhood;
    if current solution better than best solution then
        | best solution = current solution;
        | insuccessful iterations = 0;
    else
        | insuccessful iterations = insuccessful iterations + 1;
    end
end
return best solution

```

A remark about this no-wait packet scheduling algorithm (which holds also for the ILP-Based joint routing and scheduling problem) is that its output is just an ordered sequence of tasks, each one characterized by a starting time; this specified starting time is the appropriate offset in the period at which the packet should leave its source in order to avoid any kind of waiting, according to the no-wait scheduling. Even if there is a one-to-one correspondence, that is not equal to a proper schedule, as the one shown in Fig. 2.14. In order to obtain a regular schedule, some additional processing needs to be made: we need to reconstruct the paths of packets and place the TSN windows where they need to be placed; then we need to schedule guardbands before every TSN window, so as to grant the fact that

the cable is free at the beginning of everyone of them; eventually we have to fill the remaining parts of the period with best effort windows. All of that must be compatible with the format of the time aware shaper, namely the gating mechanism must be able to correctly read the information that we are providing to it. As we said, this post-processing operation is quite trivial and less time consuming with respect to the actual execution of the no-wait packet scheduling, but it is still necessary make the system work.

In the end, No-wait packet scheduling is a good scheduling algorithm as it tries to schedule every packet as soon as possible; this compression of the schedules reduces the number of guardbands, therefore increasing the throughput for best effort data. Furthermore, the fact that TSN tasks are the first to be dealt with in any cycle grants us some fault tolerant properties, as we have time to compensate for possible inaccuracies in the model. Eventually, it would be very easy to change the schedules, after an initial computation, to take into account some additional tasks required by the application: instead of recomputing all the schedules from the beginning, it suffices to simply add the new flows at the end of the current sequence. As a result, new windows will be placed in the system without affecting those already placed in the previous computations. That's another reason why we try to minimize the span, i.e. to have more time available to possibly place other tasks.

In the third chapter the no-wait packet scheduling algorithm will be implemented and tested in simulations: some additional considerations about its effectiveness, advantages and disadvantages will be presented and supported by the obtained results.

Chapter 3

Tests

This chapter is dedicated to the description of the tests about Time Sensitive Networks that have been carried out in order to evaluate performances and implementation details.

3.1 Introduction to the tests

As we mentioned in the first pages of this thesis, the topic of Time Sensitive Networking is well treated in the academic world and in general in the web from a theoretical point of view. The documentation available is very useful in order to understand the working principles of this new technology with its implications, advantages and drawbacks. Some of it has been directly accessed and is thus present in the bibliography section after the final chapter.

On the other hand, what is actually missing is a set of reference works concerning the actual implementation of a time sensitive network. This is mainly due to the fact that the standards themselves, defining the mechanisms and tools needed to successfully implement them in a flexible way, are yet to be finished. Therefore the information necessary to start the production of software and hardware components from the main producers is incomplete or not up to date.

There are, though, a few works reporting and describing practical tests which require quite common hardware and software components, that are therefore feasible and suitable for the purposes of this thesis. They are not meant to be employed in an actual industrial environment, but rather to be educative and illustrative demos about TSN.

One example can be found on the Kalycito's web page [9]. This company has long been active in this field, and has provided a brief tutorial on how to setup two computers to have a deterministic communication, also supporting OPC UA. It is based on the tests performed at the automation fair in Hannover in 2018, described in the whitepaper also available in the webpage.

But the most useful contribution to this thesis has been provided by Intel, which also has a long term committment to the development of TSN tools. On the web page is available a detailed presentation concerning TSN, with its features and possible use cases [10]. Moreover, the page refers to the Intel's account on GitHub, where a series of demos is present [11].

The GitHub page is organized in three branches, two of them dating back to April 2019 and the master one created in October 2020, while I was researching the topic. The newest branch is actually not complete and is waiting to be provided in the next future with the suitable code and explanatory guide; not to mention that the required hardware is brand new and not too easy to find on short notice. Hence we focused on the second branch, the one named "apollolake-i".

This branch is in turn divided in four demos, and has a very detailed and in-depth guide which is supposed to cover all the steps, from getting the hardware to evaluating the results. For this reason we chose to follow the instructions of the guide and recreate the tests from Intel, with the side goal of understanding the mechanisms and the implementation details adopted in order to make the test work.

The topology of the tests is quite simple: two devices connected with an Ethernet cable on one of their ports, with possibly a router in between; one of them acts as a talker, namely sends information, and the other one as a listener. Both devices are equipped with an Intel processor belonging to the Apollolake family, with an I210 as Network Interface Controller (NIC); they are also supposed to run a version of the Linux operating system, which the usual tool of choice in any experimental work such as this one, thanks to its flexibility and configurability. The following picture shows the required setup for the tests.

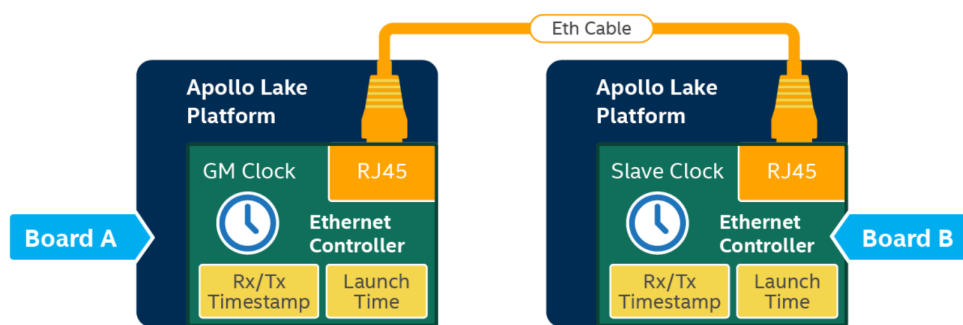


Figure 3.1: Setup for the tests

As we said, the test allows the presence of a switch connecting board A and board B. However the switch had to be TSN-compatible, and we did

not have one; so we tested the simpler configuration shown in Fig. 3.1.

The four mentioned demos, that make the test, have the following topics:

- Precision time protocol
- Credit-based shaping
- Time aware shaper
- TSN with OPC UA

Every demo has a dedicated folder from which the necessary code can be downloaded. The user guide contains the information about how to use the provided code; in particular the user can choose whether to use pre-made scripts or to insert manually all the configuration commands from the Linux's terminal. Clearly the second option is the one that provides more insight on what has been done, so that's the one that has been adopted; also, the Linux's packets could be installed manually, as we did, or by means of a Yocto project, a special file containing the configuration software needed.

3.2 Setup

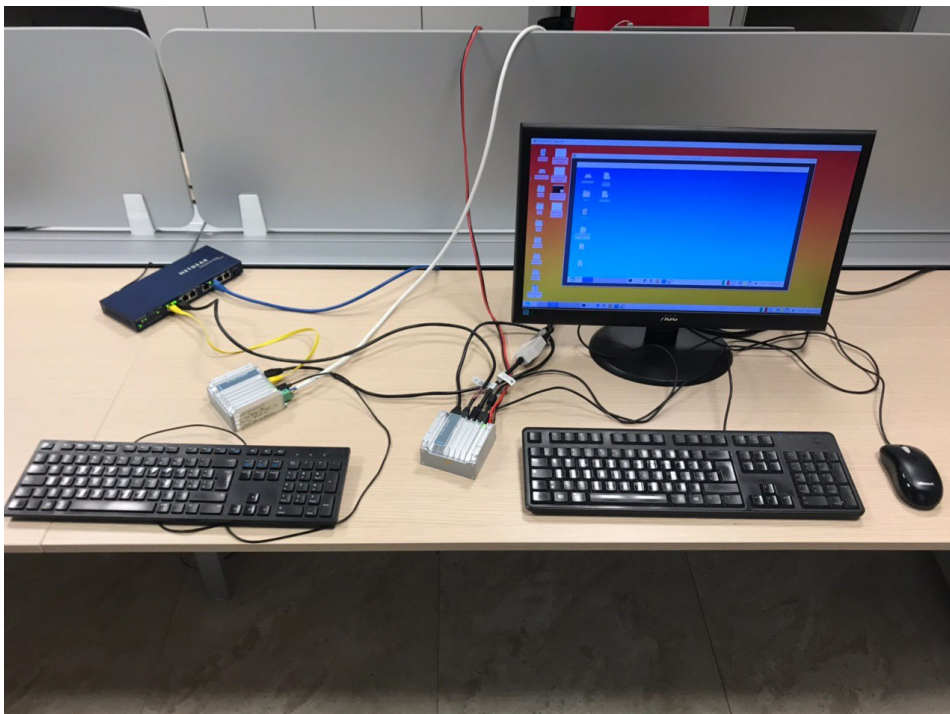


Figure 3.2: Setup for the tests

The previous picture shows the setup that we built to perform the tests. On the desk there are two identical industrial PCs by Siemens, model 127E, which satisfy the hardware requirements of the test, namely they have a Intel processor belonging to the family of the Apollolakes and have a I210 NIC.

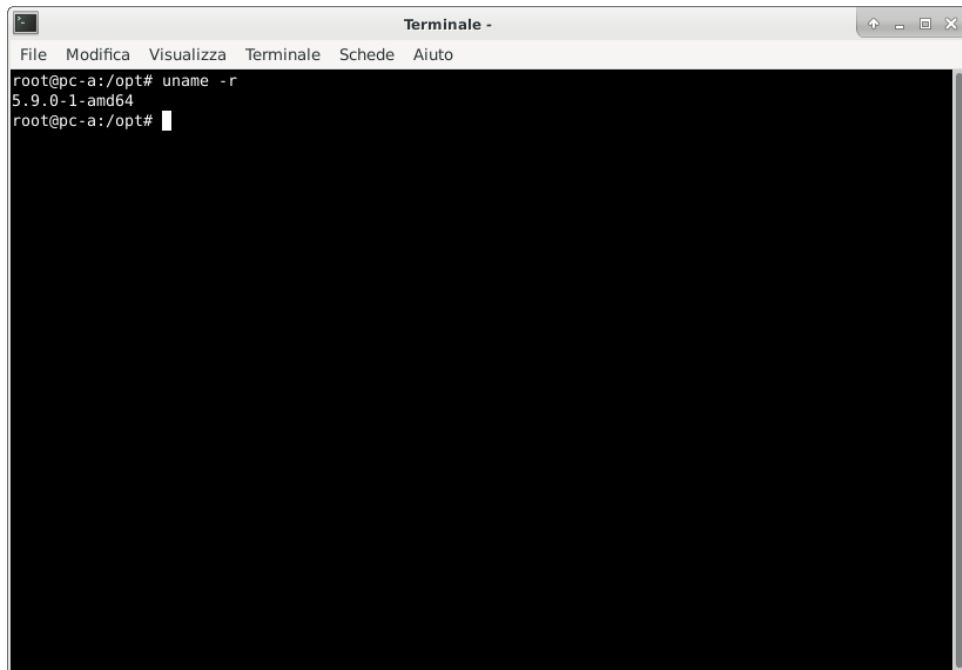
They have two ethernet ports each, which are used to establish a direct connection and to access the Internet; the second ports of the devices are mutually connected via the ethernet cable, while the first ones are both connected to the switch on the left, which is in turn connectet to the network and provides both with information from the Internet.

The router is also useful to carry the VNC packets. VNC is a graphical desktop-sharing system used to remotely control another computer; we launched a server application in the PC that was not connected to the monitor, and then we launched a client application on the other one in order to be able to see both desktops on one screen. By selecting the first port as the access point for the client, we made all the VNC packets flow through the router, namely a longer path but not interfering with the time sensitive communications on the direct link. The desktops have been assigned different colors, blue and red, in order to better distinguish them, and they are both visible on the monitor in Fig. 3.2. In particular, the PC with the blue desktop on the left has been named "pc-a", while the other, on the right and with red desktop, "pc-b".

The operating system installed on both PCs is Linux Debian; we installed four different versions of the kernel, for reasons that will be more clear later. The kernel versions are the following:

- 4.19
- 4.19-rt
- 5.08
- 5.09

Right after the boot it was possible to access the GRUB's menu' and select the desired version for the kernel. If nothing had been done, after a timeout the operating system would be launched with the latest used version of the kernel. The following picture shows the terminal's output of pc-a about its running kernel version.



```
Terminale -
File Modifica Visualizza Terminale Schede Aiuto
root@pc-a:/opt# uname -r
5.9.0-1-amd64
root@pc-a:/opt#
```

Figure 3.3: pc-a and its kernel version

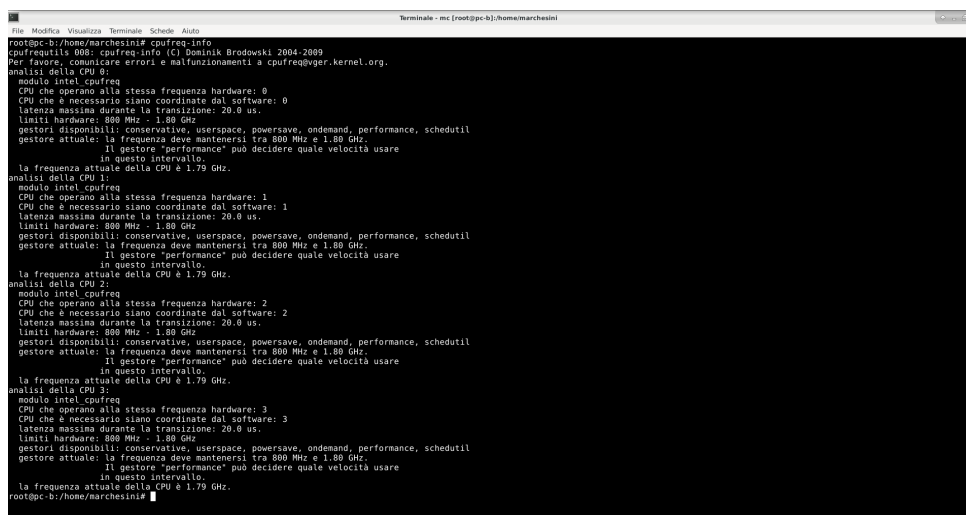
Then we setup the IP addresses of the four ports on our system. We organized them in the following way (note that the names in the system for the ethernet ports were eno1 and eno2 for the first and the second port respectively):

1. pc-a
 - eno1: 192.168.31.32
 - eno2: 192.168.3.32
2. pc-b
 - eno1: 192.168.31.33
 - eno2: 192.168.3.33

As we said, we used eno2 on both the devices for the time sensitive communications, while eno1 was connected to the router for internet connection and for the VNC application.

The last configuration step described in the user guide was related to the CPU optimization, namely some settings that allow the CPU to perform better exploiting all of its hardware capabilities, especially in a real-time framework. In particular we had to check that the so-called C-states of the CPU were disabled: it is a power setting available in the BIOS that limits

the energy consumption when the CPU is idle, but reduces its responsiveness and overall performance. This option is normally enabled, but since these PCs had already been used in the factory, in order to maximize the performance they had already been optimized in this sense. Then we also increased the operating frequency to the maximum allowable value, eliminating the safety margin. As a result, by requesting the information about the cpu, the terminal showed this output:



```
File Modifica Visualizza Terminale Schede Aiuto
Terminale - mc (root@pc-b):/home/marchesini
root@pc-b:~/home/marchesini# cpufreq-info
cpufrequtils 008: cpufreq-info (C) Dominik Brodowski 2004-2009
Per favore, comunicare errori e malfunzionamenti a cpufreq@kernel.org.
analisi della CPU 0:
 modulo intel_cpufreq
 CPU che operano alla stessa frequenza hardware: 0
 CPU che è necessario siano coordinate dal software: 0
 latenza massima durante la transizione: 20.0 us.
 limiti hardware: 800 MHz - 1.80 GHz
 gestori disponibili: conservative, userspace, powersave, ondemand, performance, schedutil
 gestore attuale: la frequenza deve mantenersi tra 800 MHz e 1.80 GHz.
                    Il gestore "performance" può decidere quale velocità usare
                    in questo intervallo.
 la frequenza attuale della CPU è 1.79 GHz.
analisi della CPU 1:
 modulo intel_cpufreq
 CPU che operano alla stessa frequenza hardware: 1
 CPU che è necessario siano coordinate dal software: 1
 latenza massima durante la transizione: 20.0 us.
 limiti hardware: 800 MHz - 1.80 GHz
 gestori disponibili: conservative, userspace, powersave, ondemand, performance, schedutil
 gestore attuale: la frequenza deve mantenersi tra 800 MHz e 1.80 GHz.
                    Il gestore "performance" può decidere quale velocità usare
                    in questo intervallo.
 la frequenza attuale della CPU è 1.79 GHz.
analisi della CPU 2:
 modulo intel_cpufreq
 CPU che operano alla stessa frequenza hardware: 2
 CPU che è necessario siano coordinate dal software: 2
 latenza massima durante la transizione: 20.0 us.
 limiti hardware: 800 MHz - 1.80 GHz
 gestori disponibili: conservative, userspace, powersave, ondemand, performance, schedutil
 gestore attuale: la frequenza deve mantenersi tra 800 MHz e 1.80 GHz.
                    Il gestore "performance" può decidere quale velocità usare
                    in questo intervallo.
 la frequenza attuale della CPU è 1.79 GHz.
analisi della CPU 3:
 modulo intel_cpufreq
 CPU che operano alla stessa frequenza hardware: 3
 CPU che è necessario siano coordinate dal software: 3
 latenza massima durante la transizione: 20.0 us.
 limiti hardware: 800 MHz - 1.80 GHz
 gestori disponibili: conservative, userspace, powersave, ondemand, performance, schedutil
 gestore attuale: la frequenza deve mantenersi tra 800 MHz e 1.80 GHz.
                    Il gestore "performance" può decidere quale velocità usare
                    in questo intervallo.
 la frequenza attuale della CPU è 1.79 GHz.
root@pc-b:~/home/marchesini#
```

Figure 3.4: Working frequencies of the cores

With a frequency of 1.80 GHz per core, we reached the maximum value. Before that, the working frequency was about the half with respect to the maximum.

Eventually, as the last common feature to all the demos, we installed Wireshark on the PCs, but most importantly on the receiver device. Wireshark is an open-source packet sniffer: it is an application that is capable of detecting the activity at the specified port of the system and has some tools useful to analyze it. We employed it mainly to measure the incoming packets through the direct connection between the two PCs, where time sensitive data were supposed to flow, and to view the results by means of plots.

Figure 3.5 shows the activity measured by Wireshark, installed on pc-b, on eno1, i.e. the packets incoming from pc-a related to the VNC server, carrying the information about its desktop. In particular we activated Wireshark, then launched the client VNC application which established the connection with the other PC, opened a few windows and eventually closed the client application. What we see on the graph is reasonable with what has been done, namely at the start there is nothing flowing on the cable, then the amount of packets transmitted each second is extremely dependent on what we were doing on the other desktop. After closing all the communications,

the packets' rate goes back to zero.

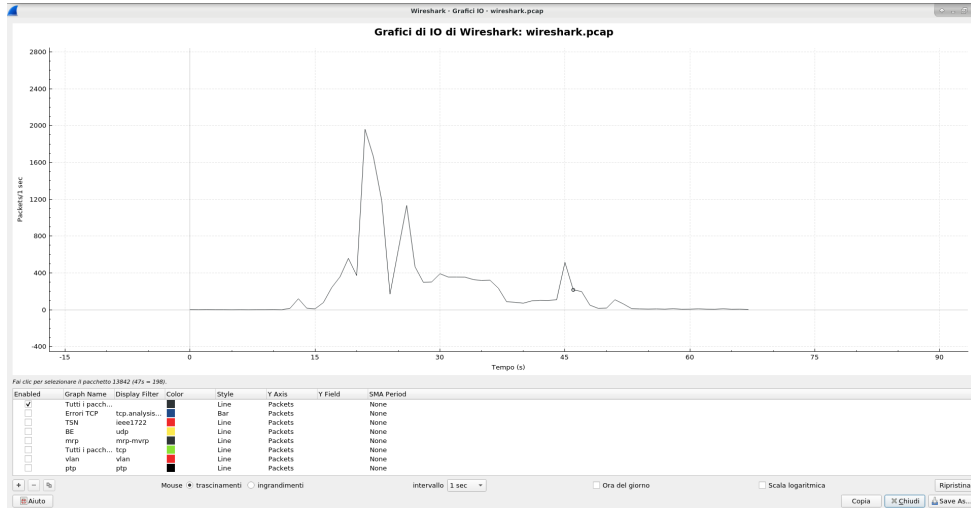


Figure 3.5: A typical graph of Wireshark

3.3 Precision Time Protocol for Linux

The first test performed was related to the Precision Time Protocol. It is a protocol that allows a set of devices to come to an agreement about the measure of the time in each one of them, in a distributed way; refer to section 2.3.1 for a more detailed explanation about PTP. In the user guide this is the first demo described after the setting up chapter.

In Linux there exists a user space utility that is called *ptp4l*, which stands for Precision Time Protocol for Linux. It implements the PTP in an automatic way, and it only needs to be activated by a command on the terminal or, more simply, by means of a script. We downloaded the latest version of source code from the internet, compiled it and put it in the */opt* folder.

As a result of its activation, the two physical clocks of the devices are going to be synchronized with the best accuracy. The problem is that what we are interested in is the synchronization between the two system clocks, which are the ones actually used by the operating system. The difference is that the hardware clock is maintained by an actual clock, powered by a battery; this implies that this clock persists a reboot. However, accessing the hardware clock requires an I/O operation, and therefore can be expensive in terms of performance; for this reason it is seldom used other than at the start of the operating system. As a consequence, there is actually an offset between the system clock and the hardware clock, which means that *ptp4l* alone is not sufficient to achieve a true and useful synchronization.

So, from the package management interface of Debian, we installed the utility *phc2sys*. It is a program which is usually supposed to be used alongside *ptp4l* in order to synchronize two clocks on the same device, namely the system clock and the hardware clock.

At this point the process should be clear. We have four clocks in the system, and we want the time measurements of the two system clocks to be as close as possible. In order to do so, we first need to synchronize the two hardware clocks, from the master device to the slave device. Then, in each device, we need to synchronize the physical clock with the system clock, but in two opposite ways: in the master device we have to make the physical clock follow the system clock, while in the slave device it is the system clock the one who follows (the physical clock). In this way we create a chain connecting the master clock to every possible slave clock belonging to any node of the network.

The two utilities need to be invoked from the terminal using the following commands. Therefore, every time that we needed to synchronize the devices we would open two terminal windows, named respectively "ptp4l" and "phc2sys".

In the master node:

```
/opt/linuxptp/ptp4l -i eno2 -A -2 -m & (3.1)
```

```
/opt/linuxptp/phc2sys -s CLOCK_REALTIME (3.2)  
-c eno2 -O 0 -w -m &
```

The first part of the commands is there to reach the folder where the source code of the utilities is stored; then, the flags of 3.1 stand for:

- i: the interface's name on which to send PTP packets
- A: to select delay mechanism automatically. Start with end-to-end (E2E) and switch to peer-to-peer (P2P) when a peer delay request is received
- m: show the output directly in the terminal

while for 3.2:

- s: specify the source of time, in this case the system clock is called `CLOCK_REALTIME`
- c: specify the target, in this case the physical clock at the port `eno2`
- O: specify the offset between master and slave to be 0 seconds

-
- w: wait until the ptp4l is in synchronized state
 - m: show the output directly in the terminal

Instead, in the slave node the following commands need to be entered:

```
/opt/linuxptp/ptp4l -i eno2 -A -2 -s -m & (3.3)
```

```
/opt/linuxptp/phc2sys -s eno2  
-c CLOCK_REALTIME -O 0 -w -m & (3.4)
```

The flags have the same meaning as the ones in 3.1 and 3.2. There are, however, two important differences:

- in 3.3 the -s flag specifies that that device is the slave. Note that we could have avoided specifying a slave at all, enabling the Best Master Clock Algorithm to select autonomously which device is most suitable to assume this role
- in 3.4 CLOCK_REALTIME and eno2 are switched, meaning that the relationship between the physical clock and the system clock is inverted, i.e. in this case the system clock follows the physical clock

In the next pages we are going to show the outputs of these commands.

```

PTP4L
File Modifica Visualizza Terminale Schede Aiuto
PTP4L x PHC2SYS
root@pc-a:/opt/linuxptp# ptp4l -i eno2 -A -2 -m &
[1] 993
root@pc-a:/opt/linuxptp# ptp4l[164.315]: selected /dev/ptp1 as PTP clock
ptp4l[164.354]: port 1: INITIALIZING to LISTENING on INIT_COMPLETE
ptp4l[164.355]: port 0: INITIALIZING to LISTENING on INIT_COMPLETE
ptp4l[171.025]: port 1: LISTENING to MASTER on ANNOUNCE_RECEIPT_TIMEOUT_EXPIRES
ptp4l[171.025]: selected local clock e0dca0.ffe.65e375 as best master
ptp4l[171.025]: port 1: assuming the grand master role

```

Figure 3.6: Master's ptp4l

```

PHC2SYS
File Modifica Visualizza Terminale Schede Aiuto
PTP4L x PHC2SYS
root@pc-a:/opt/linuxptp# phc2sys -s CLOCK_REALTIME -c eno2 -w -m -0 0 &
[1] 1051
root@pc-a:/opt/linuxptp# phc2sys[351.935]: eno2 sys offset      540 s0 freq  -6655 delay  1587
phc2sys[352.936]: eno2 sys offset      541 s2 freq  -6654 delay  1591
phc2sys[353.936]: eno2 sys offset      517 s2 freq  -6137 delay  1565
phc2sys[354.936]: eno2 sys offset       40 s2 freq  -6459 delay  1597
phc2sys[355.937]: eno2 sys offset     -173 s2 freq  -6660 delay  1586
phc2sys[356.937]: eno2 sys offset     -153 s2 freq  -6692 delay  1602
phc2sys[357.937]: eno2 sys offset     -123 s2 freq  -6708 delay  1585
phc2sys[358.938]: eno2 sys offset     -69 s2 freq  -6691 delay  1564
phc2sys[359.938]: eno2 sys offset      -4 s2 freq  -6646 delay  1596
phc2sys[360.938]: eno2 sys offset     -13 s2 freq  -6657 delay  1594
phc2sys[361.939]: eno2 sys offset     -21 s2 freq  -6668 delay  1592
phc2sys[362.939]: eno2 sys offset       0 s2 freq  -6654 delay  1598
phc2sys[363.940]: eno2 sys offset       2 s2 freq  -6652 delay  1597
phc2sys[364.940]: eno2 sys offset      10 s2 freq  -6643 delay  1603
phc2sys[365.940]: eno2 sys offset     -37 s2 freq  -6687 delay  1559
phc2sys[366.941]: eno2 sys offset       8 s2 freq  -6653 delay  1585
phc2sys[367.941]: eno2 sys offset      -9 s2 freq  -6668 delay  1553
phc2sys[368.941]: eno2 sys offset      29 s2 freq  -6633 delay  1599
phc2sys[369.942]: eno2 sys offset     -32 s2 freq  -6685 delay  1552
phc2sys[370.942]: eno2 sys offset      13 s2 freq  -6649 delay  1591
phc2sys[371.942]: eno2 sys offset       3 s2 freq  -6656 delay  1596
phc2sys[372.943]: eno2 sys offset     -18 s2 freq  -6676 delay  1591
phc2sys[373.943]: eno2 sys offset      -4 s2 freq  -6667 delay  1575
phc2sys[374.943]: eno2 sys offset       0 s2 freq  -6664 delay  1596
phc2sys[375.944]: eno2 sys offset      10 s2 freq  -6654 delay  1572
phc2sys[376.944]: eno2 sys offset       3 s2 freq  -6658 delay  1588
phc2sys[377.945]: eno2 sys offset      -8 s2 freq  -6668 delay  1585
phc2sys[378.945]: eno2 sys offset     -18 s2 freq  -6681 delay  1571
phc2sys[379.945]: eno2 sys offset     -11 s2 freq  -6679 delay  1555
phc2sys[380.946]: eno2 sys offset      33 s2 freq  -6638 delay  1600
phc2sys[381.946]: eno2 sys offset       8 s2 freq  -6654 delay  1588
phc2sys[382.946]: eno2 sys offset     -23 s2 freq  -6682 delay  1565

```

Figure 3.7: Master's phc2sys

```

PTP4L
File Modifica Visualizza Terminale Schede Aiuto
PTP4L
root@pc-b:/opt/linuxptp# ptp4l -i eno2 -A -2 -s -m &
[1] 1072
root@pc-b:/opt/linuxptp# ptp4l[394.682]: selected /dev/ptp1 as PTP clock
ptp4l[394.731]: port 1: INITIALIZING to LISTENING on INIT COMPLETE
ptp4l[394.731]: port 0: INITIALIZING to LISTENING on INIT COMPLETE
ptp4l[395.911]: port 1: new foreign master e0dca0.ffff.65e375-1
ptp4l[399.911]: selected best master clock e0dca0.ffff.65e375
ptp4l[399.911]: port 1: LISTENING to UNCALIBRATED on RS SLAVE
ptp4l[400.913]: master offset 253168187 s0 freq +0 path delay 568
ptp4l[401.913]: master offset 253164667 s1 freq -3520 path delay 953
ptp4l[402.913]: master offset -5227 s2 freq -8747 path delay 953
ptp4l[402.913]: port 1: UNCALIBRATED to SLAVE on MASTER CLOCK SELECTED
ptp4l[403.913]: master offset 755 s2 freq -4339 path delay 568
ptp4l[404.913]: master offset 2367 s2 freq -2494 path delay 148
ptp4l[405.913]: master offset 1703 s2 freq -2448 path delay 146
ptp4l[406.914]: master offset 990 s2 freq -2650 path delay 146
ptp4l[407.914]: master offset 532 s2 freq -2811 path delay 99
ptp4l[408.914]: master offset 204 s2 freq -2980 path delay 78
ptp4l[409.914]: master offset 32 s2 freq -3090 path delay 78
ptp4l[410.914]: master offset -9 s2 freq -3122 path delay 52
ptp4l[411.914]: master offset -41 s2 freq -3156 path delay 21
ptp4l[412.914]: master offset -1 s2 freq -3129 path delay -11
ptp4l[413.915]: master offset -29 s2 freq -3157 path delay -11
ptp4l[414.915]: master offset -34 s2 freq -3171 path delay -11
ptp4l[415.915]: master offset -37 s2 freq -3184 path delay -11
ptp4l[416.915]: master offset -2 s2 freq -3160 path delay -13
ptp4l[417.915]: master offset 9 s2 freq -3150 path delay -13
ptp4l[418.915]: master offset 1 s2 freq -3155 path delay -14
ptp4l[419.915]: master offset -10 s2 freq -3166 path delay -14
ptp4l[420.915]: master offset -22 s2 freq -3181 path delay -14
ptp4l[421.916]: master offset 6 s2 freq -3159 path delay -14
ptp4l[422.916]: master offset -14 s2 freq -3177 path delay -14
ptp4l[423.916]: master offset 10 s2 freq -3158 path delay -14
ptp4l[424.916]: master offset 15 s2 freq -3150 path delay -15
ptp4l[425.916]: master offset -5 s2 freq -3165 path delay -15
ptp4l[426.916]: master offset 4 s2 freq -3158 path delay -15
ptp4l[427.917]: master offset -21 s2 freq -3181 path delay -15
ptp4l[428.917]: master offset 2 s2 freq -3165 path delay -16

```

Figure 3.8: Slave's ptp4l

```

PHC2SYS
File Modifica Visualizza Terminale Schede Aiuto
PTP4L
root@pc-b:/opt/linuxptp# phc2sys -s eno2 -c CLOCK_REALTIME -O 0 -w -m &
[1] 1096
root@pc-b:/opt/linuxptp# phc2sys[446.595]: CLOCK_REALTIME phc offset 254862982 s0 freq +0 delay 1501
phc2sys[447.597]: CLOCK_REALTIME phc offset 254863962 s1 freq +978 delay 1504
phc2sys[448.598]: CLOCK_REALTIME phc offset -11 s2 freq +867 delay 1525
phc2sys[449.599]: CLOCK_REALTIME phc offset 9 s2 freq +984 delay 1504
phc2sys[450.599]: CLOCK_REALTIME phc offset -39 s2 freq +939 delay 1600
phc2sys[451.600]: CLOCK_REALTIME phc offset -7 s2 freq +959 delay 1597
phc2sys[452.600]: CLOCK_REALTIME phc offset -364 s2 freq +600 delay 2340
phc2sys[453.600]: CLOCK_REALTIME phc offset 16 s2 freq +871 delay 2338
phc2sys[454.601]: CLOCK_REALTIME phc offset 98 s2 freq +958 delay 2353
phc2sys[455.601]: CLOCK_REALTIME phc offset 149 s2 freq +1038 delay 2385
phc2sys[456.601]: CLOCK_REALTIME phc offset 88 s2 freq +1022 delay 2292
phc2sys[457.602]: CLOCK_REALTIME phc offset 400 s2 freq +1360 delay 1589
phc2sys[458.602]: CLOCK_REALTIME phc offset -294 s2 freq +786 delay 2215
phc2sys[459.602]: CLOCK_REALTIME phc offset 201 s2 freq +1193 delay 1592
phc2sys[460.603]: CLOCK_REALTIME phc offset -354 s2 freq +698 delay 2286
phc2sys[461.603]: CLOCK_REALTIME phc offset -98 s2 freq +885 delay 2291
phc2sys[462.603]: CLOCK_REALTIME phc offset 25 s2 freq +897 delay 2361
phc2sys[463.604]: CLOCK_REALTIME phc offset 108 s2 freq +1022 delay 2268
phc2sys[464.604]: CLOCK_REALTIME phc offset 31 s2 freq +978 delay 2322
phc2sys[465.604]: CLOCK_REALTIME phc offset 401 s2 freq +1357 delay 1588
phc2sys[466.605]: CLOCK_REALTIME phc offset 40 s2 freq +1116 delay 1574
phc2sys[467.605]: CLOCK_REALTIME phc offset -99 s2 freq +989 delay 1584
phc2sys[468.605]: CLOCK_REALTIME phc offset -136 s2 freq +923 delay 1623
phc2sys[469.606]: CLOCK_REALTIME phc offset -66 s2 freq +952 delay 1585
phc2sys[470.606]: CLOCK_REALTIME phc offset -452 s2 freq +546 delay 2389
phc2sys[471.606]: CLOCK_REALTIME phc offset 381 s2 freq +1244 delay 1612
phc2sys[472.607]: CLOCK_REALTIME phc offset -243 s2 freq +734 delay 2338
phc2sys[473.607]: CLOCK_REALTIME phc offset 17 s2 freq +921 delay 2344
phc2sys[474.607]: CLOCK_REALTIME phc offset 108 s2 freq +1017 delay 2238
phc2sys[475.608]: CLOCK_REALTIME phc offset 25 s2 freq +965 delay 2383
phc2sys[476.608]: CLOCK_REALTIME phc offset 363 s2 freq +1310 delay 1606
phc2sys[477.608]: CLOCK_REALTIME phc offset -277 s2 freq +780 delay 2308
phc2sys[478.609]: CLOCK_REALTIME phc offset -128 s2 freq +846 delay 2351
phc2sys[479.609]: CLOCK_REALTIME phc offset 8 s2 freq +944 delay 2364
phc2sys[480.609]: CLOCK_REALTIME phc offset 67 s2 freq +1005 delay 2378
phc2sys[481.610]: CLOCK_REALTIME phc offset 383 s2 freq +1341 delay 1624
phc2sys[482.610]: CLOCK_REALTIME phc offset 39 s2 freq +1112 delay 1556

```

Figure 3.9: Slave's phc2sys

In the first screenshot we can see that the node recognizes that it is the master, after seeing that the only other node in the network has been launched in slave mode. For all the simulations we set pc-a to be the master and pc-b to be the slave. Then it stops producing outputs because there is nothing to tell to the user; sync packet will be periodically sent in the network to let the slaves mode adjust their time.

In the second screenshot, the output of the `phc2sys` utility continuously shows the measured offset between the system clock and the physical clock. Recall that in this case the system clock is the "master". We can see that roughly every second the estimates are updated. In particular we are interested in the column of numbers right after the word "offset"; that's the measure, in nanoseconds, of the delay between the two clocks in the moment when the check was performed.

The same holds for the third and fourth image, where the same outputs for the slave node are shown. In particular it is clear that the starting values of the two clocks are very different, as the initial offset is quite big; then, though, thanks to PTP, the physical clocks start to converge and reach a sort of "consensus".

According to what we have said we can finally compute the overall delay between one system clock and the other as the summation of the three relative delays shown in the figures above. It is possible to qualitatively appreciate that this value, in steady state, is always less than one microsecond, since the measures are shown in nanoseconds.

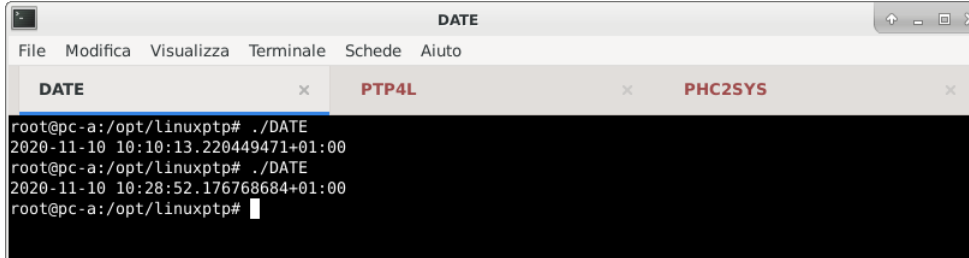
This is a good result as it provides a simple and efficient way to synchronize the clocks.

For the tests we created two scripts, named "master_synchronization" and "slave_synchronization" and containing 3.1, 3.2 and 3.3, 3.4 respectively, to be launched whenever necessary. Recall that it is not mandatory to specify a priori which is the master node, and therefore it is sufficient to create one general script and let the nodes decide by themselves who is to assume the role of master.

As far as the tests on this particular feature are concerned, we couldn't repeat the ones described in the user guide (in the first demo) as they required access to the pins of the NIC, and since ours was integrated in the chipset it wasn't possible.

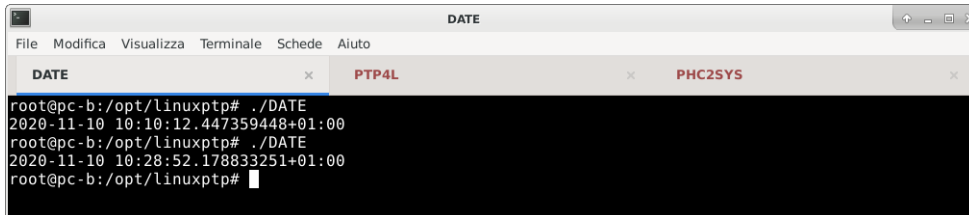
However we managed to arrange a simple and simplistic way to determine whether or not there had been improvements from the use of PTP, by means of the Linux `date` command, which prints the current date with nanosecond precision. After a weekend where both the PCs had been turned off and disconnected from any time source, we invoked the `date` command on both the PCs virtually at the same time, i.e. pressing the enter key on the two keyboards (see Fig. 3.2) as simultaneously as possible, before and after calling the PTP. This is just a qualitative way to assess the effects of PTP as the two commands cannot be entered at the exact same time. The following

figures show the results.



```
root@pc-a:/opt/linuxptp# ./DATE
2020-11-10 10:10:13.220449471+01:00
root@pc-a:/opt/linuxptp# ./DATE
2020-11-10 10:28:52.176768684+01:00
root@pc-a:/opt/linuxptp#
```

Figure 3.10: Test in the master



```
root@pc-b:/opt/linuxptp# ./DATE
2020-11-10 10:10:12.447359448+01:00
root@pc-b:/opt/linuxptp# ./DATE
2020-11-10 10:28:52.178833251+01:00
root@pc-b:/opt/linuxptp#
```

Figure 3.11: Test in the slave

As it can be seen, before invoking the PTP the delay between the two system clocks is about 0.8 seconds, while with the PTP enabled it becomes less than 2 milliseconds. The previously analyzed outputs show that in general it does not exceed 1 microsecond, so the difference is due to the noisy conditions of the experiment. However we can still appreciate the fact that a better synchronization has been achieved.

We also exploited Wireshark to measure the amount of packets sent between one host and the other by the PTP; the measured rate is approximately of 1 packet per second, which matches the updating frequency of the delays, according to what is shown in figures from 3.6 up to 3.9.

Other tests have been later performed on other machines (not our two industrial PCs), with the support of an oscilloscope, in order to precisely evaluate the performances: other than confirming the 1 microsecond accuracy, the tests revealed that even in case the PTP is turned off for a relatively short period of time, the measured drifting between the clocks is rather slow, allowing a fault tolerant behavior.

3.4 Time aware shaper

Following the order used in the second chapter to present the main features of TSN, we are going in this section to describe the characteristics and results of the tests performed about the time aware shaper; in the user guide from Intel, the next tests belong to the third demo.

Recall that the time aware shaper is a mechanism that allows us to divide the time in slices, and assign each time slice to one specific queue among the set of available output queues in the output interface of the device; we can also control the access of the traffic classes to the queues in order to separate packets with different PCP tags in the header. With this tool, and a proper scheduling, we can guarantee the delivery times of time sensitive packets in a network where -potentially- a huge amount of best effort data is flowing. Refer to section 2.3.2 for a more detailed description; for clarity, we report again the principle scheme of a time aware shaper.

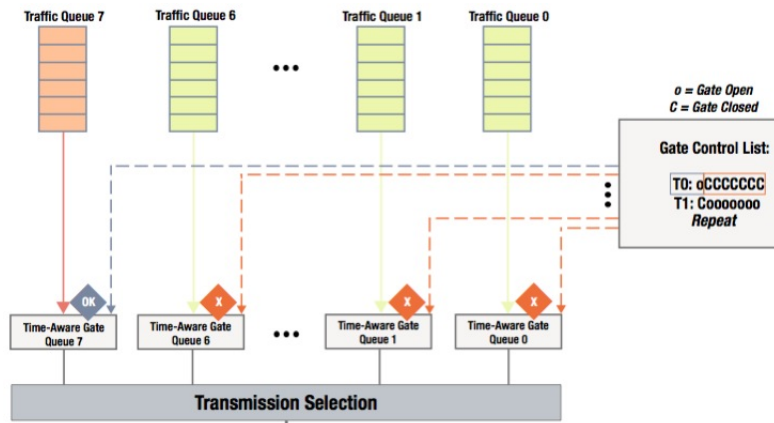


Figure 3.12: The time aware shaper

The time aware shaper, in Linux, is implemented as a *qdisc*, which stands for *queueing discipline*. It is a feature of the particular Linux's implementation of the network stack.

3.4.1 Linux's output interface

As an operating system, Linux has inside its kernel some functionalities to handle incoming and outgoing flows of information. Figure 3.13 shows a scheme for the output interface.

The overall system that is in charge of these activities is called Traffic Control, and has a userspace utility, which can be invoked by means of the command *tc*. *tc* provides mechanisms to control the way packets are sent and received, it provides a set of functionalities such as shaping, scheduling, policing and dropping network traffic.

The main element of this Linux packet scheduler are the queuing disciplines (Qdisc), which are network traffic disciplines to create queueing rules for reception and transmission. There are ingress and egress Qdisc for reception and transmission respectively. The egress Qdisc provides shaping, scheduling and filter capabilities for data transmission from the network protocol layers. On the other hand, the ingress Qdisc provides filter and

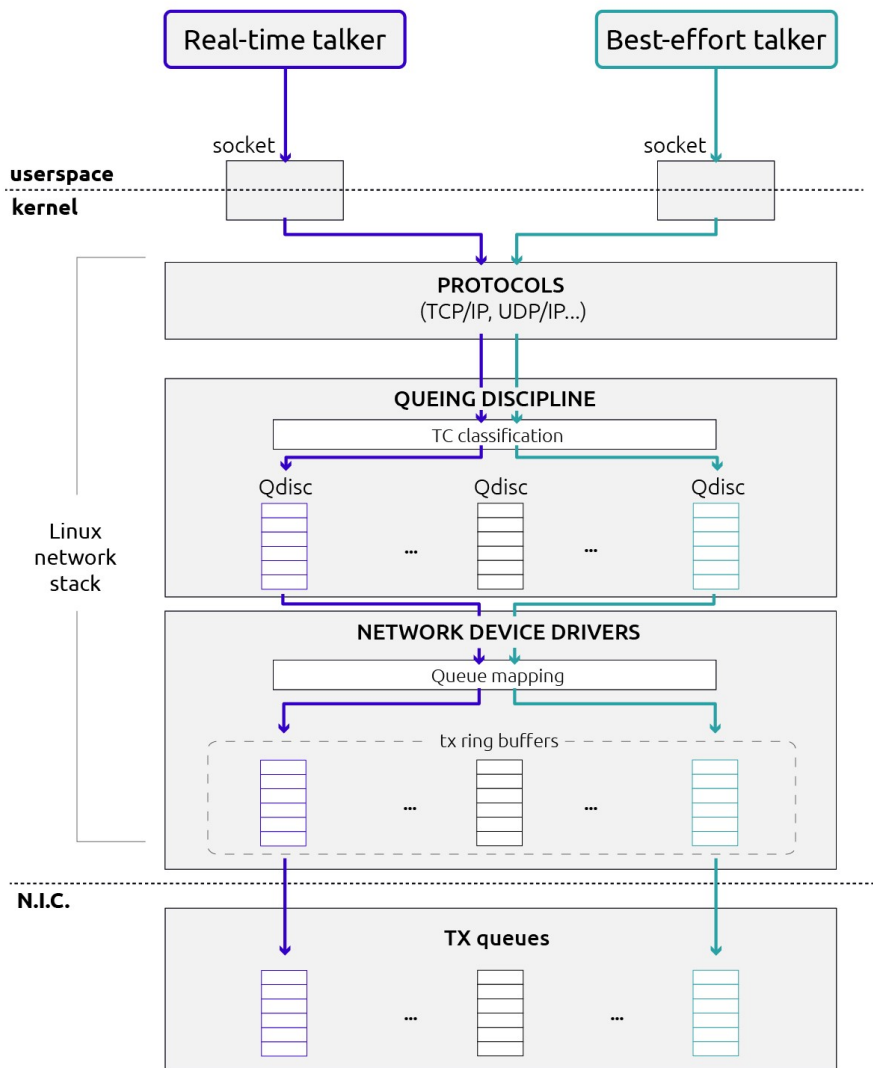


Figure 3.13: Linux's output interface

dropping features for the reception of packets; ingress Qdiscs, however, are not important for our purposes and therefore will not be considered.

For the egress Qdisc there are two basic types of disciplines: classless Qdisc and classful Qdisc. The classless Qdisc is simple and does not contain another Qdisc, so there is only one level of queuing; the classless Qdisc only determines if the packet is classified, then delayed or sent. The classful Qdisc can contain another Qdisc, so there could be several levels of queues; in this case, there may be different filters to determine from which Qdisc packets will be transmitted. We are going to focus mainly on classful Qdiscs.

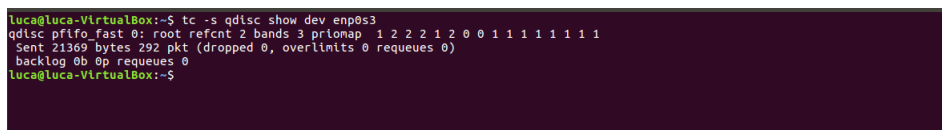
Qdisc can be then used to shape the outgoing traffic in a more determin-

istic way. For a classless Qdisc, the default discipline is the PFIFO_FAST, which corresponds to a classic fifo queue: the first packet to arrive is the first to be served as soon as one packet can leave the queue.

In Fig.3.13 it can be seen that the kernel space is organized in queues, whose access is handled by the chosen queueing discipline. We mentioned that the tc utility can be used to configure the traffic control system. For instance, by entering the following command in the terminal

```
tc -s qdisc show dev eth0 (3.5)
```

the output will be the currently working queueing discipline at the port named eth0 (in our system the ports were named eno1 and eno2).



```
luca@luca-VirtualBox:~$ tc -s qdisc show dev enp0s3
qdisc pfifo_fast 0: root refcnt 2 bands 3 priomap  1 2 2 1 2 0 0 1 1 1 1 1 1 1 1
Sent 21369 bytes 292 pkt (dropped 0, overlimits 0 requeues 0)
backlog 0b 0p requeues 0
luca@luca-VirtualBox:~$
```

Figure 3.14: Default queueing discipline

The previous picture has been taken on another PC running Debian, and we can see that the default queueing discipline is the PFIFO_FAST (in this case the system's name for the output port was enp0s3).

3.4.2 Relevant qdiscs

After introducing the traffic control utility in Linux and the concept of queueing discipline, we can in this subsection focus on the most important Qdisc for our purposes.

As we mentioned earlier, the time aware shaper mechanism is implemented by means of a particular queueing discipline called TAPRIO, which stands for time aware priority. So, in order to correctly initiate a time sensitive application, we need to configure the traffic control system of Linux to use this specific Qdisc, via the tc utility.

In this subsection, in order to make a proper comparison between different strategies about how to handle the outgoing packets, we are also going to describe another queueing discipline, named MQPRIO. It stands for multiple queues priority, and it is characterized by the same traffic separation as TAPRIO but it does not allow the user to choose which queue gets to send its packets out: instead it is based on a fixed priority model, where the queue with the highest priority, which has at least one packet enqueued, gets to send it out.

Going back to TAPRIO, the following is the required command needed to correctly configure it in the system.

```

tc  qdisc  replace  dev  eth0  parent  root  handle  100  taprio
    num_tc  3
    map    2  2  1  0  2  2  2  2  2  2  2  2  2  2  2  2
    queues 1@0  1@1  2@2
    base-time 1528743495910289987
    sched-entry S 01 300000
    sched-entry S 02 300000
    sched-entry S 04 400000
    clockid  CLOCK_TAI

```

(3.6)

As we can see the command itself is quite complicated and has a lot of fields to be filled, therefore it has been broken down to several rows in order to ease the explanation of every single part.

In the first row, as we already mentioned, we need to invoke the `tc` utility; in particular what we want to do is to replace the currently active queueing discipline with one of our choice, i.e. `taprio`. `eth0` is the name of the output port on which we want to act (in our system we used `eno2`), while the handle is a simple number useful to identify the Qdisc.

The second row defines the number of queue classes that the system is going to create; in this case we want to have three different classes. TAPRIO supports up to 16 queue classes, and each one of them has to be assigned at least one physical queue on the Network Interface Controller, so it is good practice to first check in the configuration file how many hardware queues are available.

Then we need to specify the mapping that exists between the traffic classes of Linux and the previously defined queue classes. To this end, TAPRIO uses the priority field of the `sk_buff` (socket buffer, SKB) structure. The SKB is the internal kernel data structure for managing packets. Since the SKB is a kernel structure, it cannot be directly set from user space. One way of setting it from user space is to use the `SO_PRIORITY` socket option by the sending application so that every packet sent by that specific socket will belong to the same traffic class. `sk_buff` is a 4-bits field carried around by packets; the 16 possible combinations are therefore associated to 16 different traffic classes. In the third row we need to insert, for each ordered traffic class, the corresponding queue class that we choose to be linked with it. So in this case we have 16 numbers that can range from 0 to 2, namely the three queue classes that we defined in the previous row. In particular we choose to assign traffic class 2 to queue class 1, traffic class 3 to queue class 0 and all the other traffic classes to queue class 2.

Next we need to link the queue classes to the hardware classes provided by the NIC. In this assignment we have to make sure that every queue class has at least one hardware queue and not to leave gaps in the process. Again we have to proceed following the order of the classes and this paradigm, which can be a little confusing at first: in general we will always have a structure like this "count@offset". Count indicates the number of hardware queues assigned to that specific class, while offset is the numeric identifier of the first queue assigned; the other ones will be the next in order.

So in the example we assign one queue to queue class 0, starting from the offset 0 (namely the first hardware queue); then we assign to queue class 1 the next available queue (i.e. the one at offset 1); eventually we choose to assign to queue traffic 2 the two remaining queues, which start from offset 2.

"Base-time" indicates the start time for the activation of TAPRIO. The measure is expressed in nanoseconds and is referred to the clock specified in the last row; in this case it says `CLOCK_TAI`, which is the International Atomic Time. We want to highlight that this parameter is extremely important because it is the starting time for the scheduling cycles. In order for the system to work properly, it is crucial that all the nodes have synchronized clocks (issue handled by PTP) but also that they have the same start time for the cycles (or at least an integer multiple of the cycle time). Not having an agreement on this parameter would make every other packet's synchronization mechanism useless, as it would be the same as if the clocks weren't synchronized; the implications have been largely discussed in section 2.3.1. This is also the reason why it needs to be specified in nanoseconds.

Eventually, the last three rows that we haven't described yet are used to enter the schedule for the time aware shaper. The first part is fixed and indicates that that row is an entry of the schedule; then we have two integers. The first one is the bitmask related to the open-closed combination of the gates, reported as an integer, while the second one is the duration, always in nanoseconds, of the time window. First of all we can compute the cycle time of the schedule as the summation of all the time windows: in this case we have 1 millisecond cycles.

After that, we can extract from each row the information about which queue classes are going to be open and which ones are going to be closed in that window. In the example we configure TAPRIO so as to have one 300 microseconds window where only the first class has access to the communication medium, followed by another 300 microseconds window for the second queue class and eventually a 400 microseconds window for traffic class 2.

Note that this number does not match with the queue identifier, since we need ones when the queue is open and zeros when it is closed, and therefore the two are completely unrelated. It would be more clear if we translated the integers in binary form, as we do next:

-
- 01 = 001
 - 02 = 010
 - 04 = 100

It is also important to note that any combination of this 3-bits sequence is supported. This means that any number from 00 to 07 can be entered as a legit bitmask, and then its associated binary form will have to be looked at. Four of the five combinations that have not been listed above involve multiple queues being open at the same time; the most important combination, though, is 00. It is in fact the one that allows us to close all the gates and to block every enqueued packet from going out, implementing therefore the so-called guardbands that have been described in section 2.3.2. They are basically used in place of a preemptive mechanism before a time sensitive window, in order to let all the best effort transmissions terminate in time; this way we are able to guarantee that the communication medium is free at the start of every TSN window.

Once we have specified all of the listed parameters for the system, the traffic control utility installs TAPRIO in the output interface of the desired port and the schedule is going to be active. Of course the number of entries and their content are a degree of freedom of the user, which has to carefully produce them, either by hand or using some algorithm, as discussed in section 2.4.

Next we are going to describe MQPRIO, which is another kind of priority-based queueing discipline. It is simpler with respect to TAPRIO as it does not feature the gates system, but the packets to be sent out whenever the communication medium is free are chosen by directly considering the highest priority queue.

The following command, which is also derived from the `tc` utility, is needed to install MQPRIO on one output port.

```
tc qdisc add dev eth0 parent root handle 100 mqprio
    num_tc 3
    map 2 2 1 0 2 2 2 2 2 2 2 2 2 2 2
    queues 1@0 1@1 2@2
    hw 0
```

(3.7)

We can see that it needs much less information, compared with TAPRIO (3.6); this because it is a simpler queueing discipline. First of all there is no need of a schedule, as the behavior of the interface does not depend on time. Furthermore the information about the start time and the reference clock are unnecessary, since devices do not need to be coordinated and synchronized.

Both TAPRIO and MQPRIO have been tested and in the next subsection will be presented individually and in comparison one with the other; even now, though, we could imagine that the fact that MQPRIO is simpler on paper implies a common tradeoff, i.e. that its performances cannot be as good as the ones we would expect from TAPRIO, which also needs additional support features such as PTP. In particular one issue that is not addressed by MQPRIO is preemption: TAPRIO allows to schedule idle periods (i.e. guardbands) in place of packet preemption to get the same results, but MQPRIO does not have anything preventing best effort packets from delaying the transmission of time sensitive packets.

Eventually in this subsection we need to introduce one last queueing discipline, that is ETF (Earliest Tx Time First). Right in the manual, we find that ETF allows applications to control the instant when a packet should be dequeued from the traffic control layer into the netdevice. If offload is configured and supported by the network interface card, it will also control when packets leave the network controller.

ETF achieves that by buffering packets until a configurable time before their transmission time (i.e. txtime, or deadline), which can be configured via an option that is called delta. The qdisc manages to order packets by their txtime so that they will be dequeued following the (next) earliest txtime first. It relies on the SO_TXTIME socket option and the SCM_TXTIME CMSG in each packet field to configure the behavior of time dependent sockets.

As we said, ETF is supposed to work alongside its hardware version which has to be implemented in the Network Interface Controller and is called Launch Time feature. The I210 NIC is one of the few NICs that currently support this feature and for this reason is the most suitable for TSN applications. Therefore, by employing these two mechanisms, we are able to further improve the accuracy, both from the kernel's side and the network device's side, with which packets leave the system. Note also that ETF is not an alternative to TAPRIO or MQPRIO, but something to be used in parallel.

We report below an example of command needed to install ETF in the output port. The structure is the same as the ones seen before but with the fields needed for etf; in particular in the last row we activate the offload mode, meaning that the NIC supports Launch Time.

```
tc qdisc replace dev eth0 parent root 1 etf
    clockid CLOCK_TAI
    delta 300000
    offload
```

(3.8)

3.4.3 Intel's code

The software employed in the tests has some "general" components that belong to the Linux's framework, such as its queueing disciplines; these features have been presented in the previous subsection and are also described in the manual. There are, though, "particular" components as well, namely some userspace applications created by Intel specifically for these tests. They are meant to be compiled and invoked by the user to ease the configuration of the system and to actually send and receive packets. We analyzed the code provided and extracted the particular features and implementation details that we are going to list below, and that have already been partially covered.

We downloaded from GitHub two programs, named *scheduler.py* and *sample-app-taprio.c*. As we said they are meant to be executed by the user in order to perform the test and automatize the process of configuration of the system.

In general, in order not to have to modify the code, these programs need to be called with several arguments, some of which are text files where, for instance, the schedule can be specified. This is the case for the *scheduler.py*, the python program that has to be called to start the test. The name itself can be a bit misleading as it does not generate the schedule, but it deals with the configuration of the queuing disciplines according to a schedule that the user must provide. This program in fact takes as input the name of the interface on which the user wants to set the qdisc, plus two text files: one for the schedule itself and one for the mapping of the traffic classes with the queue classes. In the main, if the schedule file is a proper file, the program starts to build the command to install TAPRIO (similar to 3.6), otherwise it gets ready to install MQPRIO. Then the program creates another command to install the ETF qdisc. All the information about these queueing disciplines is provided by means of the two text files, which have to be created according to some "protocol" in order for the program to read them correctly. For instance the mapping file has four columns: in the first we have to put the traffic classes, in the second the corresponding queue classes, in the third the word "ETF" (if we want to install it) and in the fourth its pre-fetching time.

We want to highlight that this is a sort of local protocol, which means that we have to generate the text files following its rules but it is not specified in the general standards of TSN; it applies only to the *scheduler.py*. It would be in fact possible to configure by hand the queuing disciplines, using the commands discussed in the previous subsection.

The second program is *sample-app-taprio.c*. This program is more complex as it is responsible for handling the TSN packets and can be used both to send them and to receive them. With one input flag it is possible to specify whether the particular instance should be working as a talker or as a listener; then, according to the choice, the program launches a different thread which

creates either a receiving socket or a sending socket. In the receiving case all is left to specify is the output port through which the program should listen and some other less relevant parameters about the plots. In the sending case, instead, it is necessary to provide, first of all, the IP address of the receiving device; then, also a text file which specifies details about the generation of TSN packets and their traffic class must be added. We can in fact create a generation pattern which will be cyclically repeated, with a certain number of packets sent per cycle at the specified time windows; then we have also to assign the packets a specific traffic class.

As we mentioned in the previous subsection, it is possible to tag the generated packets and assign them to a specific traffic class by using the `SO_PRIORITY` option when creating a socket.

```
if (setsockopt(fd, SOL_SOCKET, SO_PRIORITY, &priority,
              sizeof(priority))) {
    pr_err("Couldn't set socket priority: %m\n");
    goto no_option;
}
```

Figure 3.15: A piece of code from `sample-app-taprio.c`

Figure 3.15 shows the portion of code in `sample-app-taprio.c` that uses the `SO_PRIORITY` socket option to specify the desired priority for all the packets leaving the socket just created. This means that an instance of this application is able to generate only one type of TSN data, namely packets belonging to one traffic class. In order to have multiple TSN data streams it is necessary to invoke as many instances of `sample-app-taprio.c`.

When called in transmit mode, the program is also able to timestamp the packets, so that on the receiving side it is possible to compute the latency, namely the time it takes to one packet to travel from source to destination. Another important measure computed in the destination, namely in the device where the program is called in receiving mode, is the interpacket latency: it is the time that passes between two consecutive arrivals of packets belonging to the same traffic class. Actually, as we will see later, interpacket latency is the main parameter considered for the evaluation of the results.

The code and a more accurate description of it is available of the GitHub account of Intel [11].

3.4.4 TAPRIO

Here we finally describe in deep the test carried out about the time aware shaper, exploiting the TAPRIO queueing discipline in Linux.

Recall also that we followed the user guide provided by Intel [11], and this is the third demo; refer to Figure 3.2 for the hardware configuration. We had two devices, one sender and one receiver, which were supposed to exchange TSN and best effort packets via the same communication medium.

The objective of the test was to verify that the TSN mechanism named Time Aware Shaper is in fact able to handle large amounts of packets and at the same time guarantee the delivery times of the time sensitive ones, especially in congested situations. To this end, some additional software had to be installed in order to perform the test. In particular, from the package handler, we installed the utility *iperf3*, which we later used to create the best effort traffic. Iperf3 has client and server functionality, and can create data streams to measure the throughput between the two ends in one or both directions. Typical iperf output contains a time-stamped report of the amount of data transferred and the throughput measured.

The following picture shows the problem that we were trying to solve.

```

Terminale -
File Modifica Visualizza Terminale Schede Aiuto
~C
-- pc-b ping statistics ---
12 packets transmitted, 12 received, 0% packet loss, time 27ms
rtt min/avg/max/mdev = 3.412/4.100/5.010/0.563 ms
root@pc-a:/home/marchesini# ping pc-b
PING pc-b (192.168.2.33) 56(84) bytes of data:
64 bytes from pc-b (192.168.3.33): icmp_seq=1 ttl=64 time=0.648 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=2 ttl=64 time=0.529 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=3 ttl=64 time=0.769 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=4 ttl=64 time=0.811 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=5 ttl=64 time=0.683 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=6 ttl=64 time=0.956 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=7 ttl=64 time=0.976 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=8 ttl=64 time=0.842 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=9 ttl=64 time=0.578 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=10 ttl=64 time=0.759 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=11 ttl=64 time=0.756 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=12 ttl=64 time=0.269 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=13 ttl=64 time=0.748 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=14 ttl=64 time=0.777 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=15 ttl=64 time=0.646 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=16 ttl=64 time=0.777 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=17 ttl=64 time=0.760 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=18 ttl=64 time=5.01 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=19 ttl=64 time=5.54 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=20 ttl=64 time=3.28 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=21 ttl=64 time=4.37 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=22 ttl=64 time=4.14 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=23 ttl=64 time=4.33 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=24 ttl=64 time=3.35 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=25 ttl=64 time=4.34 ms
64 bytes from pc-b (192.168.3.33): icmp_seq=26 ttl=64 time=5.17 ms
^C
-- pc-b ping statistics ---
26 packets transmitted, 26 received, 0% packet loss, time 236ms
rtt min/avg/max/mdev = 0.269/1.992/5.536/1.802 ms
root@pc-a:/home/marchesini#

```

Figure 3.16: A piece of code from sample-app-taprio.c

We used the *ping* utility, which simply sends periodically a packet to the specified IP address, reporting also the time taken to reach the destination and come back. In this trivial test we first called the ping when no one else was actually using the ethernet cable to communicate with pc-b, and after a while we called the already mentioned utility iperf3 to generate a large amount of packets, also directed to pc-b. As we can see, the round trip time for the ping packets, which at the beginning had -even if affected by a high variance- a low mean value, becomes ten times bigger.

This happens because the output interface of the device is flooded with packets that are supposed to be sent through the same port. Conflicts arise between said packets due to the fact that the hardware resources of the port cannot satisfy all the requests, and therefore the port itself acts a bottleneck. The generic ping packet arrives at the output port where the PFIFO_FAST queueing discipline is installed; it is put in a queue where, possibly, many other packets have arrived before and so it has to wait a long time before having the chance to leave the device.

So in the test we recreated a similar scenario, where we had large amounts of generic packets and a few important packets (like the ping packets in the previous example) directed towards pc-b, with the goal of keeping the latency of the important packets bounded despite the traffic.

We also want to highlight that these conflicts are "local" to the output port of pc-a: once packets leave it, they are already ordered and do not have to wait anymore before reaching their destination. This is mainly due to the simple configuration adopted in our test; in a generic network all the devices experience conflicts, most of all routers, as they have multiple input ports and they have to handle the parallel flows from each one of them. This is also the reason why the most relevant parameter that we measured was interpacket latency rather than pure latency. In this case latency is not affected by the conflicts as when packets leave the port they have already been solved. Interpacket latency, instead, directly depends on the conflicts because, if a packet is constrained to wait, its interarrival time from the previous one is going to increase. Again, these remarks apply to this particular case: in general latency is affected by conflicts as well. For instance, when a packet needs to wait in a router along its path toward the destination its latency increases. But in our configuration this cannot happen. We will return to this topic later and in the next chapter.

In order to simplify even more the test, we created several scripts to be executed either in pc-a or in pc-b, containing the commands that we explained before. First of all we used PTP to synchronize the clocks of the devices, setting pc-a as the master node and pc-b as the slave node. Then we activated the listening applications in the listening device, namely pc-b, for best effort data (iperf3 in server mode) and for the TSN packets (sample-app-taprio.c in receiving mode). In pc-a we first activated iperf3 in client mode, which acted as the best effort talker; next we invoked the scheduler.py program in order to properly configure TAPRIO and ETF on eno2. Eventually we called sample-app-taprio.c in transmit mode to produce and send TSN packets; we defined two instances of this program in order to send two different traffic classes of TSN data. After a while we killed all the involved processes and invoked the last script, which interpreted the data files produced by sample-app-taprio.c in order to plot the results about interpacket latency. A .pcap file was also available to be opened with Wireshark in order to see all the activity at eno2 in pc-b.

The details about the TSN flows are the following. Through the configuration file for sample-app-taprio.c we set 1 millisecond cycles where two streams of time sensitive packets were sent twice per cycle. In particular they were arranged like this within the millisecond:

- 100 microseconds: generate a packet with traffic class 5
- 200 microseconds: generate a packet with traffic class 3

-
- 600 microseconds: generate a packet with traffic class 5
 - 700 microseconds: generate a packet with traffic class 3

In the TAPRIO queuing discipline we mapped the traffic classes in the following way:

- traffic class 5 to queue class 0
- traffic class 3 to queue class 1
- all the other traffic classes to queue class 3

The schedule that we installed with TAPRIO had to be compatible with the traffic patterns and, since this was a simple configuration, was made by hand.

- from 0 to 100 microseconds: open queue class 3 (bitmask 08)
- from 100 to 200 microseconds: open queue class 0 (bitmask 01)
- from 200 to 300 microseconds: open queue class 1 (bitmask 02)
- from 300 to 600 microseconds: open queue class 3 (bitmask 08)
- from 600 to 700 microseconds: open queue class 0 (bitmask 01)
- from 700 to 800 microseconds: open queue class 1 (bitmask 02)
- from 800 to 1000 microseconds: open queue class 3 (bitmask 08)

The three lists reported are defined in three configuration files: the last two are needed to complete the tc command, similar to 3.6. The first one is particular to the application as it defines the generation pattern of time sensitive data. By looking at them we can see that packets belonging to the same traffic class are generated at regular intervals of half a millisecond, and the system is configured so that to send them out as soon as they are available.

Thus the expected result is to have interarrival times for packets of the same class of 500 nanoseconds. We could also expect a rate decrease for the best effort data and in general of the global rate at the port: we are in fact dedicating specific time windows to the transmission of one packet at a time, therefore wasting some bandwidth. To this end, we must also note that we did not provide for guardbands in the schedule as TAPRIO itself is able to make up for them; before the transmission of every packet it computes the time it takes for it to be completely transmitted and, if there isn't enough, does not even start it. In this way, at the beginning of every new time window, the communication medium is free.

The next picture shows the output of the script that we created to setup TAPRIO, which employed scheduler.py.

```

Terminale - mc (root@pc-a):/opt/TEST_A_max_bandwidth/DEMO-3-TSN/TAPRIO
File Modifica Visualizza Terminale Schede Aiuto
root@pc-a:/home/marchesini# mc
root@pc-a:/opt/TEST_A_max_bandwidth/DEMO-3-TSN/TAPRIO# ./master_python_scheduler
kernel_sched_rt_runtime us = -1
Deleting any existing qdisc...
Adding etf qdisc on queue 0...
Adding etf qdisc on queue 1...
Base time set to 30s from now (ns): 1605801131000000000
qdisc taprio 100: root refcnt 9 tc 4 map 3 3 3 1 3 0 3 2 3 3 3 3 3 3 3
queues offset 0 count 1 offset 1 count 1 offset 2 count 1 offset 3 count 1
clockid TAI base-time 0 cycle-time 0 cycle-time-extension 0 base-time 1605801131000000000 cycle-time 1000000 cycle-time-extension 0
index 0 cmd S gatemask 0x8 interval 100000
index 1 cmd S gatemask 0x1 interval 100000
index 2 cmd S gatemask 0x2 interval 100000
index 3 cmd S gatemask 0x8 interval 200000
index 4 cmd S gatemask 0x8 interval 100000
index 5 cmd S gatemask 0x1 interval 100000
index 6 cmd S gatemask 0x2 interval 100000
index 7 cmd S gatemask 0x8 interval 200000
qdisc pfifo 0: parent 100:4 limit 1000p
qdisc pfifo 0: parent 100:3 limit 1000p
qdisc etf 8003: parent 100:1 clockid TAI delta 5000000 offload on deadline_mode off skip_sock_check off
qdisc etf 8004: parent 100:2 clockid TAI delta 5000000 offload on deadline_mode off skip_sock_check off
root@pc-a:/opt/TEST_A_max_bandwidth/DEMO-3-TSN/TAPRIO#

```

Figure 3.17: Output of the tc script

We can see the mapping of all the traffic classes to the queue classes, and below the time windows of the schedule, with the associated durations and bitmasks.

As we mentioned, sample-app-taprio.c is programmed, when it is in receiving mode, to keep logs about the temporal information of all the incoming TSN packets; another program provided by Intel deals with the analysis of these logs and plots the results as a histogram. On the horizontal axes we have different values of interpacket latency and on the vertical axis we have, in logarithmic scale, the number of samples that have that value on interpacket latency, with nanosecond accuracy

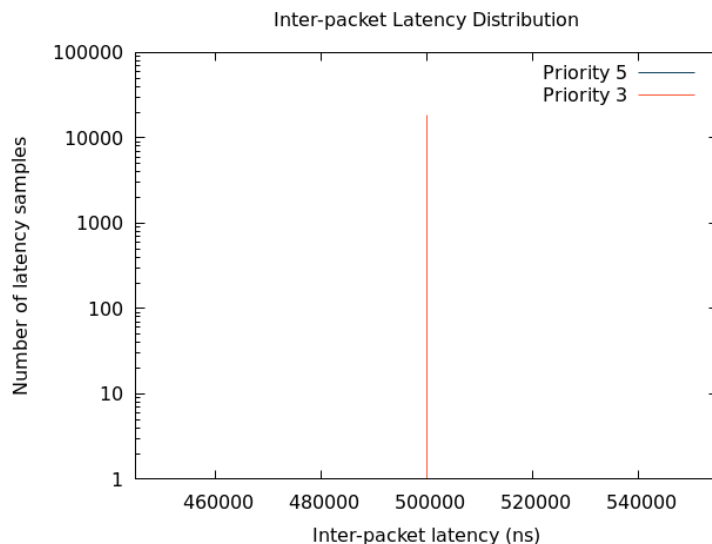


Figure 3.18: Histogram of the interpacket latency

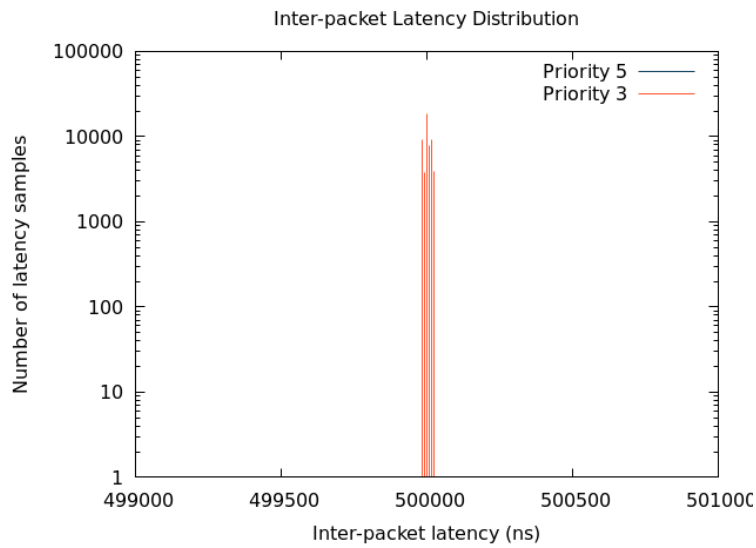


Figure 3.19: Zoom of Fig.3.18

Before getting to the analysis of these results, we report the graph by Wireshark of the measured activity at the input port of pc-b. On the horizontal axis we have the time of the measurement while on the vertical axis we have the average rate of information in packets per second. Below there is a sort of legend, as we can filter the packets according to some criteria: in this case we are able to tell the TSN packets from the others as they have a VLAN tag and we plot their rate in red; the blue line, instead, represents all the packets. So the best effort rate is the blue line minus the red one.

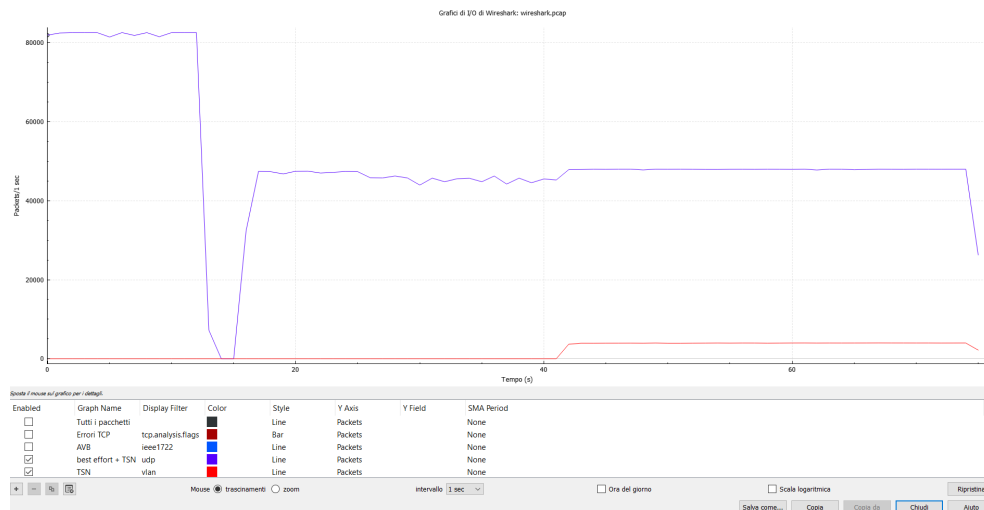


Figure 3.20: Wireshark's graph

We want to highlight that we repeated several times the same exact test, and every time the results were very close to the ones reported in Figures 3.18, 3.19 and 3.20.

The first two figures show the histogram for interpacket latency, where the time in the horizontal axis is reported with nanosecond accuracy: we can see that TSN packets have interpacket latencies very close to the expected value of half a millisecond, i.e. 500000 nanoseconds. The deviation, probably due to noise, from the exact value does not exceed a few dozen of nanoseconds, which is absolutely acceptable for any kind of application. In addition it can be noticed that said deviations can have both plus and minus signs, namely there are samples with interpacket latency greater or even smaller than half a millisecond. This is due to the fact that if a packet is slightly late and both the previous and the next are right on time, the first interpacket latency is a bit greater than 500 nanoseconds while the second is a bit smaller; therefore it is reasonable to expect a "triangular" shape for the histogram. This also implies that the vast majority of packets has an extremely high accuracy.

Let's from now on focus on the Wireshark's graph (Fig. 3.20), which is extremely meaningful as it gives a general overview of what happens in the system.

We can see that at the start the communication medium is occupied by best effort data, as the red line shows that the TSN's rate is zero. Furthermore we can spanometrically assess that the global output rate of the port saturates the capacity of the ethernet cable. In the iperf3 utility we in fact set the size of best effort data to be 1500 bytes which is, as we recall, the maximum allowable dimension for an ethernet packet. We did this to put ourselves in the worst scenario, where best effort data could possibly occupy the communication medium, and therefore constraining all the other packets to wait, as long as possible. So if we compute $1500 \times 8 \times 82000$, which is the initial bitrate, we get 984 million bits per second, i.e. almost the one Gigabit per second of the cable.

In this initial part, but also in the next one, the rate itself of the best effort data has some fluctuations: this is due to the fact that the system tries to send out as many packets as it can but in the meantime it has to perform other tasks such as schedule processes, execute them and so on. Therefore there is no guarantee on the rate's value.

Approximately 15 seconds from from the start we invoked the tc utility to setup TAPRIO. We can see that for a couple of seconds the transmission of all kinds of packets is interrupted; this behavior is reasonable as the system in this period is updating its output interface, replacing the current queueing discipline with a new one.

After the transient, the transmissions resume but with a clear difference: the rate is now limited with respect to the one we had before. That's one of the main drawbacks that we mentioned: the fact that we are dedicating some specific time windows to the transmission of single packets reduces the

time available to all the others to be sent, therefore decreasing the overall capacity of the port. The behavior of the best effort packets is the same as before, only restricted to limited windows within the cycle. The plot makes sense also because, qualitatively, the rate is almost halved, according to the schedule we provided where almost half of the period (4/10) was dedicated to TSN data.

After 40 seconds from the start, TSN packets begin to flow in their dedicated time windows, and their rate is consequently constant at 4000 packets per second, namely twice per millisecond for each stream, as intended.

In conclusion, the transmission of the selected kinds of packets is not affected at all by the other types of traffic that may use the same medium, allowing a deterministic communication. This is the purpose of Time Sensitive Networks.

3.4.5 MQPRIO

This subsection is dedicated to the test performed with the MQPRIO queuing discipline, described in section 3.4.2. We said that this is not a TSN-compatible mechanism, but we used it to provide a comparison with respect to the TAPRIO qdisc used in the previous subsection.

The features of the test are the similar to the previous one, so refer to the related subsection for the traffic patterns details. This time, though, we did not provide a schedule and installed MQPRIO instead of TAPRIO; we also disabled the ETF queuing discipline, which was useful to provide more determinism to the actual transmission time of packets.

First we executed the test without the interfering presence of the best effort packets, so with the TSN data alone; these were the results.

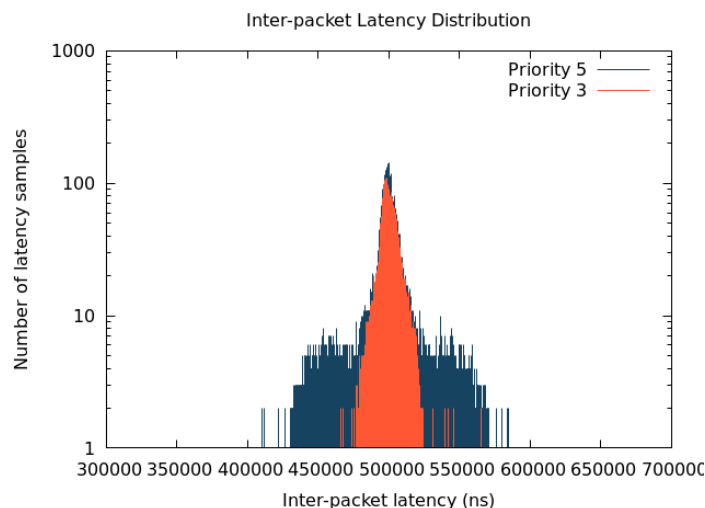


Figure 3.21: First test with MQPRIO

The performances of the system shown in Fig. 3.21 are, even though definitely worse with respect to the case with TAPRIO, not that bad considering the requirements of an industrial application. The vast majority of packets has an interpacket latency between 450 and 550 microseconds, and the more we get close to half a millisecond the more samples we have.

However this kind of solution is not suitable to be used, because in this test the TSN packets are the only ones flowing in the ethernet cable, which is therefore way more oversized than it needs to be.

In order to have a proper comparison with TAPRIO, we must recreate the same operating conditions, i.e. we need to add the best effort packets.

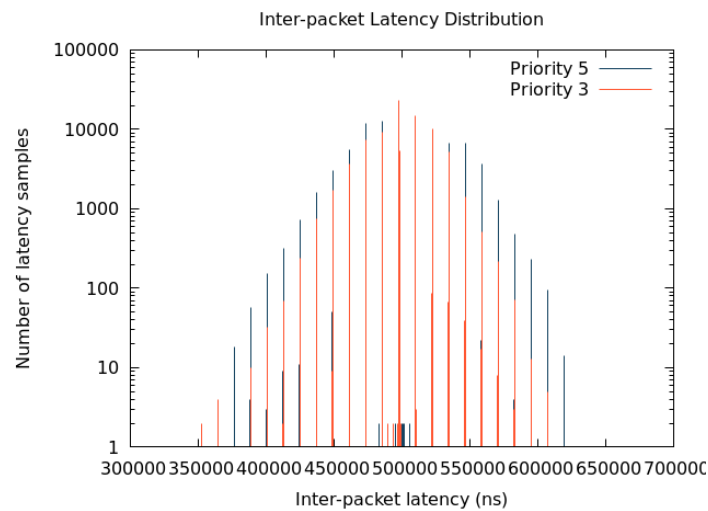


Figure 3.22: Second test with MQPRIO

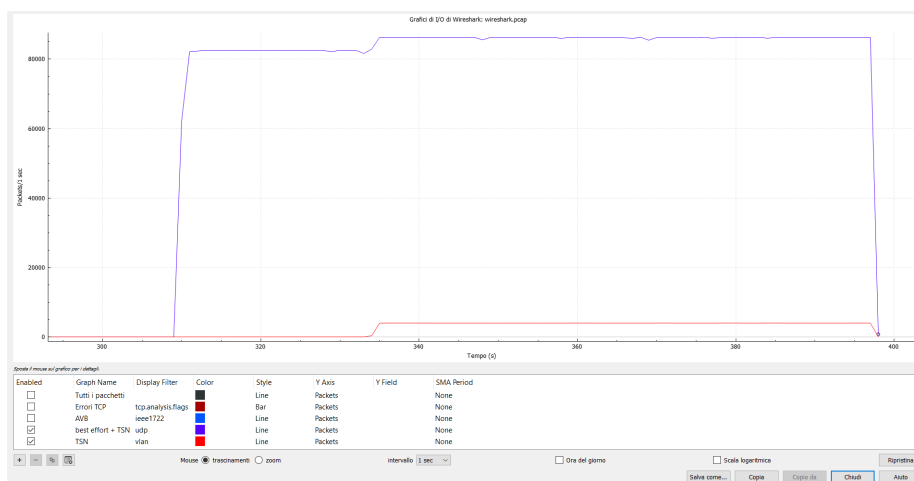


Figure 3.23: Wireshark of the second test with MQPRIO

This second test shows that MQPRIO is way more sensitive to the presence of disturbances: if in fact TSN packets were the only ones in the system, then the performances could even be acceptable, but if instead there is interfering traffic the interpacket latency loses completely its deterministic properties. We can see that now we have a meaningful number of samples with up to 150 microseconds of difference from the nominal value of 500. On the other hand, from Fig. 3.23, we can gather that the output capacity of the port remains unchanged: we do not lose the two setup seconds and most importantly we do not have any further limitation on the bandwidth. Recall that the blue line represents the sum of TSN packets plus best effort packets. At the start we have the usual number of roughly 82000 best effort packets per second; then, when TSN data start to flow, the overall number of packets increases and it is slightly smaller than 86000.

What happens is that, since the system does not have dedicated windows, the communication medium is occupied by a packet as soon the previous one has finished its transmission: in this way the number of bits that actually go out every second remains constant at the maximum value allowed by the hardware capabilities. As we said, the queueing discipline chooses the next packet according to the priority of the queue where it is waiting, and queues are filled according to the traffic priority, so TSN packets have a higher priority with respect to the others. The problem in this case is that if a best effort packet is in transmission in the moment that a TSN one is created and has to be sent out, the time sensitive packet needs to wait until the end of the current transmission. Furthermore, recall that this is the worst possible scenario in terms of waiting times as the dimension of the best effort packets is the biggest allowed for ethernet packets. So, in order to be able to always send something, we lose the capability of sending out immediately the TSN packets as soon as they are produced; therefore they are constrained to wait and their interpacket latency isn't deterministic anymore, as Fig. 3.22 shows.

A side note about Figure 3.23: when the TSN packets start to flow it may seem that the overall output capacity of the port has increased, but that's not true. TSN packets are way smaller with respect to best effort ones, so with the same amount of bits/second the number of TSN packets is higher than the number of best effort packets. Therefore the measure of packets/second increases, while the bit/seconds does not. Confirming this, the new value of packets/second is a bit smaller than $82000+4000$, which means that less best effort packets flow every second, and in their place the 4000 TSN packets do.

Eventually we report the results of a test conducted with the default queueing discipline, namely PFIFO_FAST. The system is expected to behave even worse with respect to MQPRIO, as it is not able to tell apart the time sensitive packets from the others.

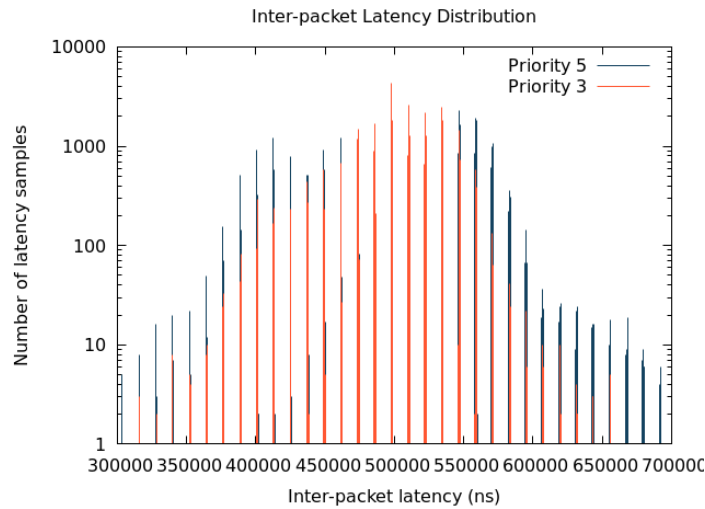


Figure 3.24: Test with PFIFO_FAST

What happens in this case is that TSN packets get lost in the first-in-first-out queue of the output interface, among the numerous best effort packets; therefore, whenever the system has to choose which packet to send out next, it can only take the first entry of the queue, which is probably a best effort packet.

Eventually a comparison between Figures 3.22 and 3.23 with Figures 3.18, 3.19 and 3.20 is extremely helpful and meaningful in order to comprehend the features, advantages and drawbacks of these queueing disciplines, in particular TAPRIO. We can see that with TAPRIO we can guarantee in all circumstances the interpacket latency value, at the cost of a waste of bandwidth and an increased complexity in the system, due to the fact that we need to compute a priori a suitable schedule (which could become non-trivial in large networks). With MQPRIO, instead, system's configuration is easier and we do not lose bandwidth; also, we can still shape the traffic so that TSN packets have a higher priority with respect to the others, and can therefore circulate with approximately the intended patterns even if the best effort queues are filled with packets. But we cannot guarantee anymore an effective bound on the interpacket latency; for this reason MQPRIO cannot be employed in a time sensitive network.

3.4.6 Hardware limit

After presenting the main features of MQPRIO, in this subsection we go back to TAPRIO and present another test about it. This time we changed the schedule and the traffic patterns in order to explore the limits of the hardware, namely the maximum performances allowed by the physical capabilities of our system.

In particular we chose to send out the two traffic classes traffic class, number 3 and number 5, ten times each milliseconds; therefore we modified the previous schedule dividing each window by five, obtaining 200 microseconds cycles. The traffic pattern was the following:

- 100 nanoseconds: generate a packet with traffic class 5
- 25 microseconds: generate a packet with traffic class 3
- 100 microseconds: generate a packet with traffic class 5
- 125 microseconds: generate a packet with traffic class 3

The schedule was the following:

- from 0 to 25 microseconds: open queue class 0 (bitmask 01)
- from 25 to 50 microseconds: open queue class 1 (bitmask 02)
- from 50 to 100 microseconds: open queue class 3 (bitmask 08)
- from 100 to 125 microseconds: open queue class 0 (bitmask 01)
- from 125 to 150 microseconds: open queue class 1 (bitmask 02)
- from 150 to 200 microseconds: open queue class 3 (bitmask 08)

Clearly the mapping of the queues was the same. Next we show the results.

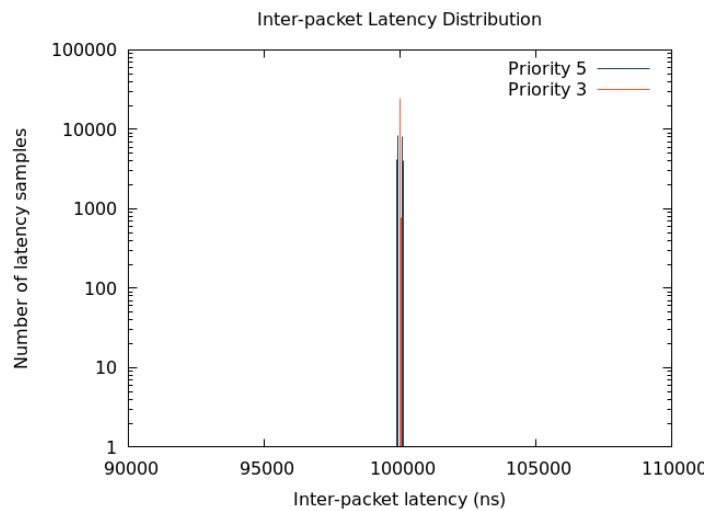


Figure 3.25: Second test with TAPRIO

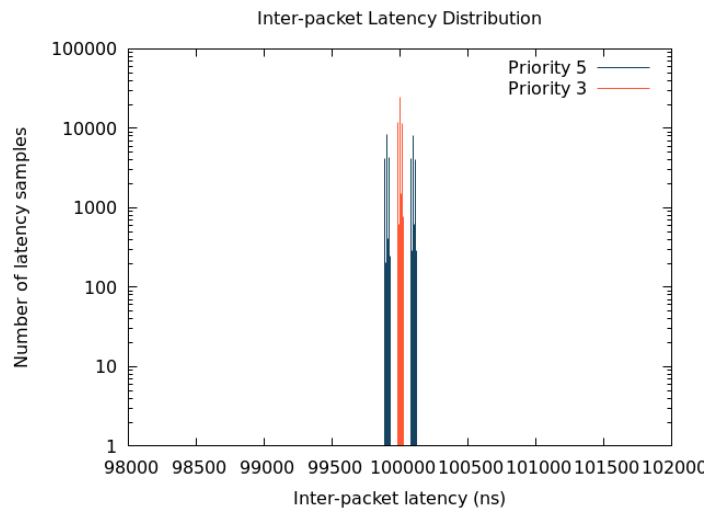


Figure 3.26: Zoom of Fig. 3.25

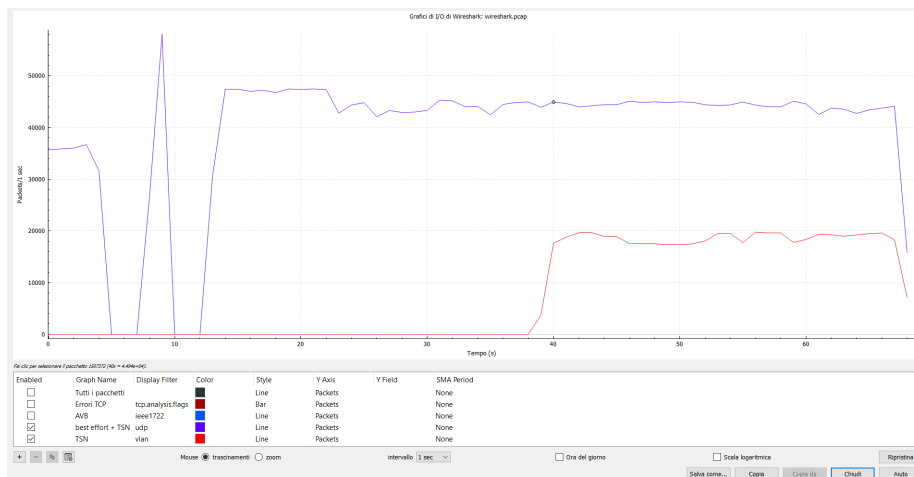


Figure 3.27: Wireshark of the second test with TAPRIO

Figures 3.25 and 3.26 show results comparable with the ones obtained in the first and nominal test. However from Figure 3.27 we can see that the Wireshark’s graph indicates that the sending rates of the TSN packets is not constant at all: even from the user’s point of view, the system seemed to struggle in the execution of all the tasks in parallel. We can in fact see that the red line approximates the 20000 packets per second which would be the target value, but it seldom manages to reach it: this may be due to the fact that the custom program from Intel is not optimized to work in such heavy conditions, while the already existing utility iperf3 had no problems in this sense.

However we can see something not very much reasonable happening in the output port as well. As soon as the TSN packets start to flow, the best effort number decreases apparently without any reason. Again, this may be due to the fact that the task that we are requiring the system to execute is heavier than it can handle. Confirming this, also the responsiveness and reactivity of the system through the desktop experienced slowdowns during the test, which then disappeared as soon as the involved tasks were killed.

In any case 20 packets every millisecond is a lot of information to be sent by a single device, so we can still conclude that the TAPRIO queuing discipline could be successfully employed in an industrial environment, provided that all the other features that we mentioned have been taken care of. To this end, we can also see this subsection as an introduction to section 3.6, which is supposed to list all the problems and inconsistencies that we encountered in the execution of these tests, which so far may have seemed simple and straightforward but actually were not so immediate.

3.5 Credit-based shaper

This section is dedicated to the test performed concerning the credit based shaper, presented in section 2.3.4. Recall that CBS is neither a necessary feature nor a sufficient one in order to correctly build a time sensitive network; it is not present on the standards (see Fig. 2.11), and can be employed at most in parallel to actual TSN mechanisms such as the time aware shaper. This because, similarly to MQPRIO (even if they are in fact two different traffic shaping mechanisms), they do not have the necessary properties in terms of determinism and granted upper bounds on the transmission times.

So the reasons why we have this section in a work concerning TSN are basically two. First of all, as we said, the CBS technology is a forerunner of what is now called Time Sensitive Networking, as it was the first to deal with strategies to shape the traffic in a more deterministic way. Second, it was part of the Intel's project about TSN: the second demo in the user guide described a simple test, similar to the ones that we already presented, and also provided some code on its GitHub account. In particular, this demo was presented as the second one, namely before the one concerning the time aware shaper; however, for the reasons that we just mentioned, we report the result of this test as the final topic, just for the sake of completeness.

The test is based upon additional software components provided by OpenAvnu, which is another organization active in the field of Time Sensitive Networking. These programs implement a stream reservation protocol, based on a time-varying credit. This stream reservation is aimed at keeping constant the sending rate of packets, spacing them out as much as possible in order to avoid bursts.

In the next picture we report again the working principle of CBS, with

the typical behavior of the credit for a class.

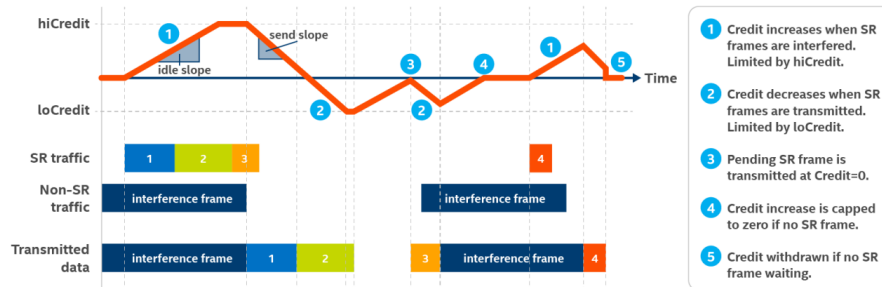


Figure 3.28: Credit based shaper

Then we report the results of the tests: we employed the iperf3 utility to generate disturbing best effort traffic, and then set up a time sensitive traffic class, firstly without any traffic shaping mechanism. The required rate was of 8000 packets per second.

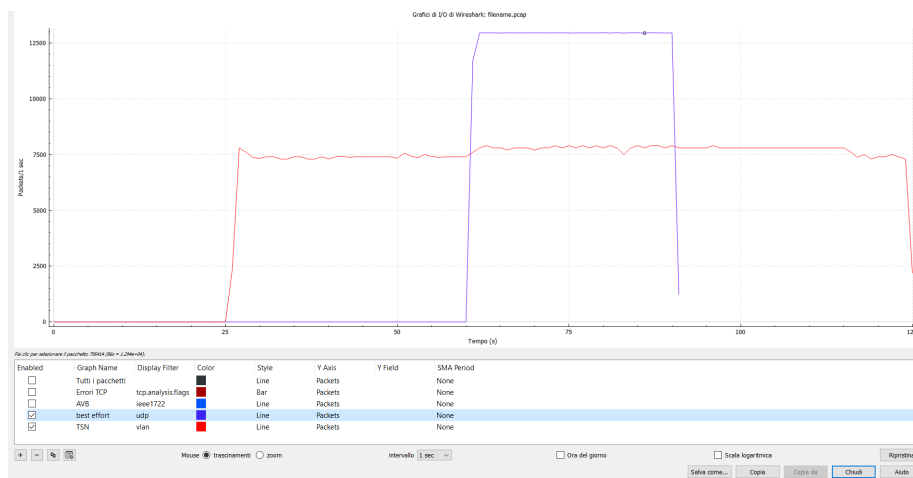


Figure 3.29: Wireshark of the first test with CBS

In the following test, instead, we activated the stream reservation feature and linked it to the time sensitive packets.

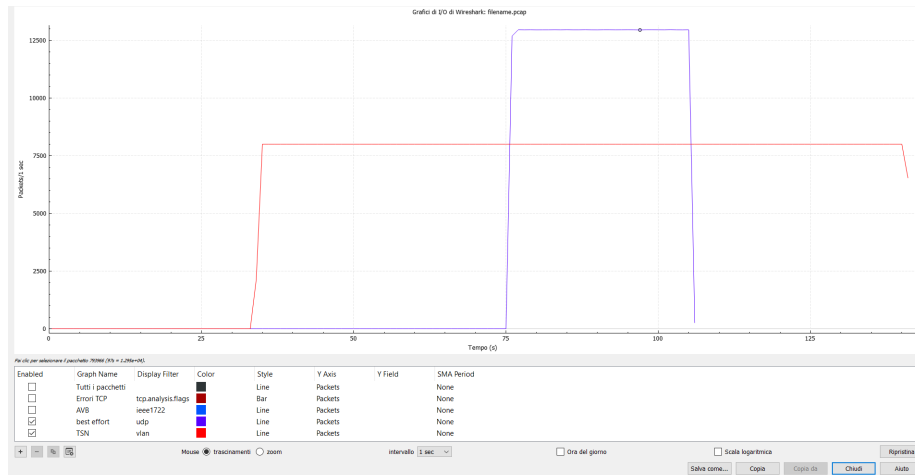


Figure 3.30: Wireshark of the second test with CBS

As usual, the blue line indicates the rate of best effort packets while the red one the rate of important time sensitive data. It is clear that whitout a traffic shaping mechanim, the rate of packets is subject to the non deterministic behavior of the port, which causes the sending rate to have several and large oscillations around the target value. In Fig. 3.29 this behavior can be observed.

On the other hand, if we adopt credit based shaping, we can see that the frequency of the time sensitive packets is consistently fixed at 8000 packets per second. This test uses in addition the ETF queueing discipline, described in the previous section, to speed up the sending process.

We want to highlight that a constant sending rate does not imply bounded transmission times nor bounded and deterministic interpacket latencies. Consider for instance Fig.3.23, where the results of the test performed using the MQPRIO qdisc are shown: the sending rate of the TSN packets is constant because the priority system of the queueing discipline manages to send them out with a relatively low delay with respect to the time they are generated. However, as we can see from Fig. 3.22, the interpacket latencies for the same packets are all but deterministic, with values ranging from 350 to 650 microseconds. We could expect a similar behavior for these packets as well, even though we were not provided with a piece of code able to actually compute latencies; this is not in fact the ultimate purpose of credit based shaping and, therefore, the reason why it is not part of the TSN mechanisms.

3.6 Troubleshooting

Approaching to the end of the chaper dedicated to the tests performed, we are going in this section to list every problematic feature and implementation

issue that we encountered in the execution of the tests.

So far, the relevant topics have been presented in such a way to be more comprehensible and logically ordered, without following the actual order in which they were made. Also, the results and procedures employed to get them may have seemed simple and straightforward, but in reality we had to study in deep the code and its features to solve some issues and actually get to the end.

Some of these problems are local to the code provided by Intel, therefore only meaningful to the execution of the tests, while some other problems are more global and thus more relevant, especially in relation to a future implementation in an industrial environment.

The user guide was partially useful, as it couldn't contain every single piece of information necessary to complete the tests. So we arranged our solutions by researching the internet for similar problems or by directly looking at the code in order to correct the mistakes and inconsistencies.

3.6.1 Kernel's version

The first important issue that we are going to address is related to the version of the kernel of the Linux's operating system installed on the two industrial PCs.

Recall from the first section of this chapter that the two 127E PCs were running a Debian distribution, with the possibility of choosing between four versions of the kernel. We started, in fact, performing the tests with the oldest one, namely the 4.19. We were able to correctly verify the effectiveness and the accuracy of the precision time protocol, but we encountered a warning when executing the code from Intel. In particular, after the call to the scheduler.py program, the terminal would give as output the following warning: "taprio qdisc not found". On the other hand, the tests involving MQPRIO worked as intended on the 4.19.

After some research we figured out that the problem was the support, from the operating system, to the specific queueing discipline. Being a newly implemented feature, TAPRIO is not supported by the old versions of the kernel, in particular in the ones before the 5.3. Not only that: TAPRIO is linked to a kernel module that has to be loaded when the tc utility is invoked, and before that it has to be made available in the configuration files of the kernel.

In the Linux's system files it is possible to find the configuration file of the kernel, which contains the information about the enabled modules. By accessing it, it is possible to verify the support to a queueing discipline. In particular, the kernel module for TAPRIO is named sch_taprio. It can be found in the queueing section, as the figures in the next page show.

```

Terminale - mc [root@pc-8]:boot
root@config:~# cat /etc/sysctl.d/99994_34_L11622+19_1641/9664_*(48856/233626b)_0919_6x96A
CONFIG_MAC802154=m
CONFIG_NET_SCHED=y
#
# Queuing/Scheduling
#
CONFIG_NET_SCH_CBQ=m
CONFIG_NET_SCH_HTB=m
CONFIG_NET_SCH_HFSC=m
CONFIG_NET_SCH_ATF=m
CONFIG_NET_SCH_PRIO=m
CONFIG_NET_SCH_MULTIQ=m
CONFIG_NET_SCH_RED=m
CONFIG_NET_SCH_SFQ=m
CONFIG_NET_SCH_SFB=m
CONFIG_NET_SCH_SFBQ=m
CONFIG_NET_SCH_TEOL=m
CONFIG_NET_SCH_TBF=m
CONFIG_NET_SCH_CBS=m
CONFIG_NET_SCH_ETF=m
# CONFIG_NET_SCH_TAPRIO is not set
CONFIG_NET_SCH_DRED=m
CONFIG_NET_SCH_DSMARK=m
CONFIG_NET_SCH_NETEM=m
CONFIG_NET_SCH_DRR=m
CONFIG_NET_SCH_MQPRIO=m
CONFIG_NET_SCH_SRRIO=m
CONFIG_NET_SCH_CKCKE=m
CONFIG_NET_SCH_DFD=m
CONFIG_NET_SCH_CODEL=m
CONFIG_NET_SCH_FQ_CODEL=m
CONFIG_NET_SCH_CAREM=m
CONFIG_NET_SCH_FQ=m
CONFIG_NET_SCH_HFQ=m
CONFIG_NET_SCH_PIE=m
# CONFIG_NET_SCH_FQ_PIE is not set
CONFIG_NET_SCH_INGRESS=m
CONFIG_NET_SCH_PLUGD=m
CONFIG_NET_SCH_ETS is not set
# CONFIG_NET_SCH_DEFAULT is not set
#
# Classification
#
CONFIG_NET_CLS=y
CONFIG_NET_CLS_BASIC=m
CONFIG_NET_CLS_U32=m
CONFIG_NET_CLS_ROUTE4=m
CONFIG_NET_CLS_MSM=m
1|lute 2|alva 3|arica 4|mpz 5|coia 6|posta 7|erca 8|lmin 9|Aprfen 10|sci

```

Figure 3.31: Configuration file for kernel 5.8

```

Terminale - mc [root@pc-8]:boot
root@config:~# cat /etc/sysctl.d/99994_34_L11688+35_1643/9648_*(48860/234724b)_0884_9x954
CONFIG_GLOWPAN_NHC_FRAGMENT=m
CONFIG_GLOWPAN_NHC_HOP=m
CONFIG_GLOWPAN_NHC_IPV6=m
CONFIG_GLOWPAN_NHC_MOBILE=m
CONFIG_GLOWPAN_NHC_ROUTING=m
CONFIG_GLOWPAN_NHC_UDP=m
CONFIG_GLOWPAN_GHC_EXT_HDR_HOP=m
CONFIG_GLOWPAN_GHC_UDP=m
CONFIG_GLOWPAN_GHC_TCP=m
CONFIG_GLOWPAN_GHC_EXT_HDR_DEST=m
CONFIG_GLOWPAN_GHC_EXT_HDR_FRAG=m
CONFIG_GLOWPAN_GHC_EXT_HDR_ROUTE=m
CONFIG_IIEEE802154=m
# CONFIG_IIEEE802154_NI802154_EXPERIMENTAL is not set
CONFIG_IIEEE802154_SOCKET=m
CONFIG_IIEEE802154_GLOWPAN=m
CONFIG_MAC802154=m
CONFIG_NET_SCHED=y
#
# Queuing/Scheduling
#
CONFIG_NET_SCH_CBQ=m
CONFIG_NET_SCH_HTB=m
CONFIG_NET_SCH_HFSC=m
CONFIG_NET_SCH_ATF=m
CONFIG_NET_SCH_PRIO=m
CONFIG_NET_SCH_MULTIQ=m
CONFIG_NET_SCH_RED=m
CONFIG_NET_SCH_SFQ=m
CONFIG_NET_SCH_SFB=m
CONFIG_NET_SCH_SFBQ=m
CONFIG_NET_SCH_TEOL=m
CONFIG_NET_SCH_TBF=m
CONFIG_NET_SCH_CBS=m
CONFIG_NET_SCH_ETF=m
CONFIG_NET_SCH_DRED=m
CONFIG_NET_SCH_DSMARK=m
CONFIG_NET_SCH_NETEM=m
CONFIG_NET_SCH_DRR=m
CONFIG_NET_SCH_MQPRIO=m
CONFIG_NET_SCH_SRRIO=m
CONFIG_NET_SCH_CKCKE=m
CONFIG_NET_SCH_DFD=m
CONFIG_NET_SCH_CODEL=m
CONFIG_NET_SCH_FQ_CODEL=m
CONFIG_NET_SCH_CAREM=m
CONFIG_NET_SCH_FQ=m
CONFIG_NET_SCH_HFQ=m
CONFIG_NET_SCH_PIE=m
1|lute 2|alva 3|arica 4|mpz 5|coia 6|posta 7|erca 8|lmin 9|Aprfen 10|sci

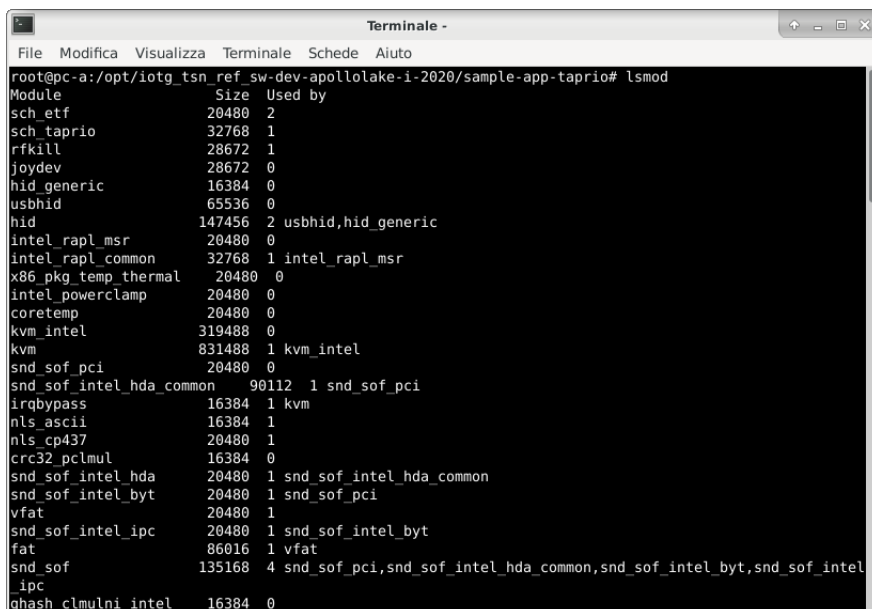
```

Figure 3.32: Configuration file for kernel 5.9

We gathered that in our compiled 5.8 kernel the `sch_taprio` module was available but not enabled, therefore the same `tc` instruction did not work with this kernel as well; it would have needed to be set and recompiled in order for it to work. So we decided to install another kernel, namely the 5.9, where the same module was already enabled. It was also necessary to update the traffic utility *iproute2* to the latest version. We can see the enabled support to other mentioned queueing disciplines such as `ETF` and `MQPRIO`, which are quite recent as well even if not as much as `TAPRIO`.

We accessed, just to check, the configuration file of the 4.19 version and the previous option about `sch_taprio` was completely missing.

After successfully setting the TAPRIO qdisc we invoked the *lsmod* utility in order to see the currently loaded kernel modules, and the output correctly showed the `sch_taprio` module among the most recently activated.



```
Terminale -
File Modifica Visualizza Terminale Schede Aiuto
root@pc-a:/opt/iotg_tsn_ref_sw-dev-apollo-lake-i-2020/sample-app-taprio# lsmod
Module                Size  Used by
sch_etf                20480  2
sch_taprio             32768  1
rfkill                 28672  1
joydev                 28672  0
hid_generic            16384  0
usbhid                 65536  0
hid                    147456  2 usbhid,hid_generic
intel_rapl_msr         20480  0
intel_rapl_common     32768  1 intel_rapl_msr
x86_pkg_temp_thermal  20480  0
intel_powerclamp      20480  0
coretemp              20480  0
kvm_intel              319488  0
kvm                    831488  1 kvm_intel
snd_sof_pci            20480  0
snd_sof_intel_hda_common 90112  1 snd_sof_pci
irqbypass             16384  1 kvm
nls_ascii              16384  1
nls_cp437              20480  1
crc32_pclmul          16384  0
snd_sof_intel_hda     20480  1 snd_sof_intel_hda_common
snd_sof_intel_byt     20480  1 snd_sof_pci
vfat                   20480  1
snd_sof_intel_ipc     20480  1 snd_sof_intel_byt
fat                    86016  1 vfat
snd_sof                135168  4 snd_sof_pci,snd_sof_intel_hda_common,snd_sof_intel_byt,snd_sof_intel
_ipc
ghash_clmulni_intel  16384  0
```

Figure 3.33: Currently loaded kernel modules

The user guide is not very much clear about this compatibility problem, which in contrast represents a relevant issue for a possible implementation of a time sensitive network: it is mandatory to use a new version of the operating system in order to set TAPRIO.

Of course TAPRIO is just one realization of the general concept of time aware shaper, which can be implemented in other ways and in other operating systems. However it is currently the best solution to build an output interface which provides so much freedom in the shaping of the traffic, i.e. compatible with the standards, in particular considering the lack of information about it on the internet.

3.6.2 Topology of the test

The next topic is not related to an actual issue as much as it is a consideration about the test itself and the mechanisms needed.

Recall that the system is made by a talker device and by a listener device, directly connected by an ethernet cable. Large amounts of information packets are produced in the talker device and, as a consequence, conflicts about which packet should be sent first are created. The time aware shaper mechanism is needed to solve said conflicts and to build a sequence of packets that flow towards the listening device. In general it is supposed to exploit the syn-

chronization between devices to minimize, thanks to coordinated schedules, the latencies of the TSN packets. However, as we have already highlighted, the conflicts are present only in the talking device and nowhere else; therefore an actual synchronization between the devices is not needed. Also, the main parameter used to evaluate the results is the interpacket latency, which is measured by the same clock, i.e. the one in the listening device. Thus, whether it was synchronized with the other one or not, it did not really matter.

As a consequence, even though the user guide suggested to do it, we performed a test without first invoking the PTP and we got the same exact result, as we expected. We are not showing the usual histogram, but instead the output shown by the sample-app-taprio.c program.

```

tsn listener
File Modifica Visualizza Terminale Scheda Aiuto
tsn listener
be listener
plot
TSN VLAN Priority 5 - Seq: 63342 Latency = -710176534 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 3 - Seq: 57046 Latency = -710187137 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 5 - Seq: 62342 Latency = -710200080 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 3 - Seq: 57047 Latency = -710160208 ns Inter-packet latency = 500016 ns
TSN VLAN Priority 5 - Seq: 63344 Latency = -710175210 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 3 - Seq: 57048 Latency = -710155586 ns Inter-packet latency = 499984 ns
TSN VLAN Priority 5 - Seq: 63345 Latency = -710203067 ns Inter-packet latency = 500024 ns
TSN VLAN Priority 3 - Seq: 57049 Latency = -710126537 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 5 - Seq: 63346 Latency = -710226205 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 3 - Seq: 57050 Latency = -710221656 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 5 - Seq: 63347 Latency = -710203305 ns Inter-packet latency = 499984 ns
TSN VLAN Priority 3 - Seq: 57051 Latency = -710155577 ns Inter-packet latency = 500016 ns
TSN VLAN Priority 5 - Seq: 63348 Latency = -710193977 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 3 - Seq: 57052 Latency = -710165419 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 5 - Seq: 63349 Latency = -710196610 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 3 - Seq: 57053 Latency = -710160020 ns Inter-packet latency = 499992 ns
TSN VLAN Priority 5 - Seq: 63350 Latency = -710173477 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 3 - Seq: 57054 Latency = -710159612 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 5 - Seq: 63351 Latency = -710198704 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 3 - Seq: 57055 Latency = -710158136 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 5 - Seq: 63352 Latency = -710172395 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 3 - Seq: 57056 Latency = -710150147 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 5 - Seq: 63353 Latency = -710182711 ns Inter-packet latency = 500016 ns
TSN VLAN Priority 3 - Seq: 57057 Latency = -710159353 ns Inter-packet latency = 500008 ns
TSN VLAN Priority 5 - Seq: 63354 Latency = -710219639 ns Inter-packet latency = 500008 ns
TSN VLAN Priority 3 - Seq: 57058 Latency = -710188772 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 5 - Seq: 63355 Latency = -710187222 ns Inter-packet latency = 499984 ns
TSN VLAN Priority 3 - Seq: 57059 Latency = -710154957 ns Inter-packet latency = 500016 ns
TSN VLAN Priority 5 - Seq: 63356 Latency = -710175900 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 3 - Seq: 57060 Latency = -710154674 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 5 - Seq: 63357 Latency = -710198403 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 3 - Seq: 57061 Latency = -710155938 ns Inter-packet latency = 499992 ns
TSN VLAN Priority 5 - Seq: 63358 Latency = -710221545 ns Inter-packet latency = 500008 ns
TSN VLAN Priority 3 - Seq: 57062 Latency = -710157296 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 5 - Seq: 63359 Latency = -710152098 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 3 - Seq: 57063 Latency = -710155442 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 5 - Seq: 63360 Latency = -710176515 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 3 - Seq: 57064 Latency = -710154301 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 5 - Seq: 63361 Latency = -710171141 ns Inter-packet latency = 500016 ns
TSN VLAN Priority 3 - Seq: 57065 Latency = -710155272 ns Inter-packet latency = 500008 ns
TSN VLAN Priority 5 - Seq: 63362 Latency = -710184476 ns Inter-packet latency = 500008 ns
TSN VLAN Priority 3 - Seq: 57066 Latency = -710156843 ns Inter-packet latency = 500000 ns
TSN VLAN Priority 5 - Seq: 63363 Latency = -710206650 ns Inter-packet latency = 499984 ns
TSN VLAN Priority 3 - Seq: 57067 Latency = -710154074 ns Inter-packet latency = 500000 ns
Terminato
root@pc-b:/opt/TEST_B/DEMO-3-TSN/TAPRIO#

```

Figure 3.34: Output of sample-app-taprio.c

The program is supposed to print the latency and interpacket latency of every newly arrived TSN packet. As we can see the interpacket latency is quite consistently of 500 microseconds, which is the expected value; the pure latency, though, has negative values, which does not really make sense. This is due to the fact that there is a relatively big offset between the two system clocks used to register the departure and arrival times. In this case it seems that pc-a's clock is late with respect to pc-b's, therefore the difference between the two measures is still negative even though the arrival indubitably happens after the departure.

Another feature that could be added to the system was a TSN-compatible switch; the user guide describes the already presented TAPRIO tests also in a configuration containing a TSN switch between the talker and the listener. TSN switches are normal switches that implement a time aware shaper. The reasoning is then the same that we have already done: conflicts may arise due to the fact that numerous packets can arrive at the switch at the same

time via different ports and require to be sent through the the same port; the time aware shaper is the tool used to solve this conflicts by means of a suitable schedule that takes into account the a priori known incming traffic patterns.

So we would have needed to provide a schedule, which in this case was extremely simple thanks to the topology of the test, to the switch and then repeat the procedure that we presented before. We had some options about TSN-compatible switches from different vendors but we did not end up buying one, mostly because of their cost: the tests would have been, for sure, more complete with a switch in the middle but, as we said, the main mechanisms of TSN could be evaluated even without a switch. Furthermore it was not actually necessary for the switch to be TSN-compatible: we said that the conflicts are already solved in the talker device and, since the network is very simple, the switch wouldn't have conflicts at all. Therefore a time aware shaper isn't necessary and so a normal switch could work.

As a result we performed the same test on the eno1 port, the one connected to the switch, creating a 3-hops path between the devices. As we expected, the results were the same. The only difference was due to the fact that the switch had a limited capacity (100 Mb/s) and therefore acted as a bottleneck, further downsizing the maximum throughput of our talker device. But, as long as the physical constraints were met, the test worked exactly as the one without a switch (and as the one with a TSN switch described in the user guide).

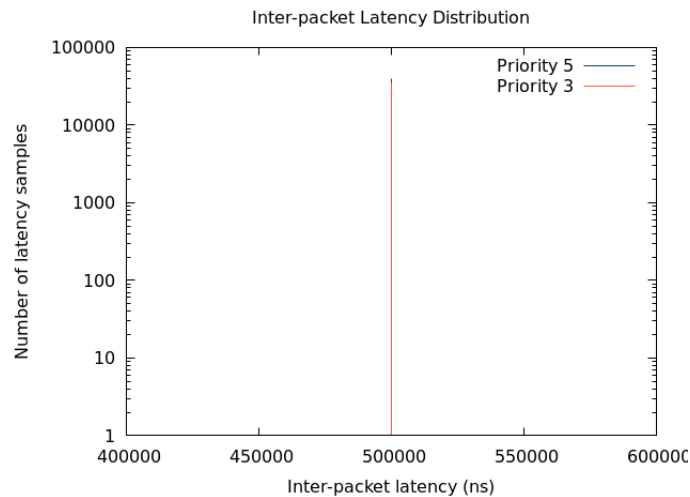


Figure 3.35: Test with the switch

An important remark about the two cases just reported. It was the particular and simple configuration of our system that allowed us to not employ some of the previously described TSN mechanisms such as time synchroniza-

tion and time aware shaping. In general, in a generic network, they both need to be used by every node, in order to support every single time sensitive flow from every possible source to every possible destination.

3.6.3 Crash of VNC

Here we are going to describe some issues directly connected with the use of the TAPRIO queueing discipline. The objective is to highlight that, as part of the TSN features, the new technologies may still present some compatibility issues with the ecosystem that need fixing.

Our initial configuration wasn't the one shown in Fig. 3.2: we simply had one computer connected to the monitor and to the network, which was pc-b, and the other only connected to the first by means of one ethernet cable on eno2. We needed internet connection occasionally on one PC, so we chose to have it on pc-b and to keep the network as simple as possible by only employing two elements. As a consequence, all the packets that needed to be exchanged between pc-a and pc-b, other than the ones for the test, had to flow on the only connection. That included the PTP packets and, most importantly, the packets of VNC, which allowed us to see the desktop of pc-a on the monitor connected to pc-b.

We were able to successfully perform the tests about PTP and MQPRIO with this configuration but we noticed that as soon as we installed TAPRIO on pc-a, namely the server for the VNC application, its desktop experienced slowdowns and losses of responsiveness, up to a complete block. So we couldn't manage to start the talking process to send the data, because we couldn't control the system anymore.

At first we thought that this issue affected the whole system, but actually it was a matter of communication of the VNC packets between client and server. In some way, the change of the system's output interface prevented the VNC to establish a stable channel of communication, which made it difficult to send the information about the desktop from pc-a to pc-b and the commands in the opposite direction, and eventually it would kill the process. Consider for instance Fig. 3.20, where the output activity for pc-a is shown by means of a Wireshark's graph. Recall that right after the call to the tc utility to set TAPRIO, the system had a couple of seconds of pause needed to reset its output interface; during this interval no packet was allowed to leave the device, and that includes the VNC packets. This phenomenon must have interfered with the VNC application and caused the problem.

So, the main reason why we added the switch in our configuration was to provide an alternative path that could be used by VNC to carry its packets, otherwise we would have needed an additional monitor, together with another keyboard and another mouse, in order to be able to directly access to pc-a. Since the tc utility and TAPRIO act on the single port, enabling

the queueing discipline on an output port does not affect the behavior of the other port. As a result we were able to successfully install TAPRIO and still manage to see and control the desktop of pc-a and send commands to it through eno1.

As we already mentioned in the previous subsection, the switch that we used did not have any relevant feature, in particular concerning TSN. We could have added another direct link between the PCs, but in this way we would have lost the connection to the internet. It was not strictly necessary for the tests but we used it, first of all, to download the Intel's software from GitHub, and then to download or update the other software components needed for the test; not to mention all the research work about the issues that we are describing in this section, most of which wouldn't have been solved without external help from the community. So we connected the eno1 ports of the two PCs to the switch, and then we used another port of the switch for the internet cable. In this way both the PCs had direct access to the network, which facilitated any possible update or download of code.

In conclusion we discovered that TAPRIO may cause some compatibility issues with the processes that use its output port. As far as VNC is concerned, probably a patch could fix this problem, but the important takehome message is that this queueing discipline is quite "invasive", as we have seen, and the correct behavior of other programs must be ensured before employing it.

3.6.4 Intel's code

So far we have covered the most important and general issues about our tests. From now on, instead, we are going to focus on the problems related to the code provided by Intel (in particular the `sample-app-taprio.c` application), which still needed to be solved in order to perform the tests correctly. In fact, though, they are less important, but we are going to briefly describe them for the sake of completeness.

A phenomenon that we have already had the chance to experience is the loss of packets from the talker. Consider for instance Fig.3.27, which is related to the test where twenty packets for each milliseconds were supposed to be sent out. We can see that the sender struggles to keep up with this sending rate and sometimes it fails in producing the packet; therefore the actual rate is slightly lower than 20000 packets per second; a similar thing happened even in the nominal test, when the number of packets per millisecond was four.

If a packet is not produced, the related instance of its TSN flow is missing; therefore the measured time between the previous arrival and the next one becomes twice the time it is supposed to be. We could observe this behavior by enlarging the horizontal axis in the resulting histogram: we had a number of samples with 1 millisecond of interpacket latency.

In the nominal case, however, the number of lost packets was very low compared to the total number of packets, less than 1%. Then, increasing the required frequency, this percentage increases as well, as shown in Figure 3.27.

This situation occurred both with TAPRIO and with MQPRIO as queueing disciplines, so it is probably due to the `sample-app-taptio.c` program, which is the application that has to create and send the packets: apparently, if called several times with a high operating frequency, it could happen that a few packets got lost in the system and did not actually leave the device. Furthermore, with lower frequencies and higher cycle times, we did not experience at all this phenomenon.

A second issue that we encountered with this application occurred whenever we wanted to launch the application in the listening mode. The process would get instantly killed with the error "free(): double free detected in teache2" printed on the terminal. This was an actual bug on the writing of the code for the program, as it used the function `popen` to open a process and then invoked the `free` function on the opened process without any apparent reason. Therefore the system couldn't execute the action, would give the error and kill the process. It seemed unusual to find such a kind of error, but we searched the code for every possible call of `free`, we found the wrong one, deleted it and recompiled the program. This particular line was in a portion of the code which was supposed to be executed only when the program was called in the listening mode, so we weren't able to detect the problem at the start, but only after a while. However in the end we could observe that the change had been effective and the process wouldn't get killed.

We also noticed that the `sample-app-taprio.c`, in talking mode, wasn't able to actually send the messages unless the `iperf3`, that we used to generate the best effort traffic utility had been called before, at least once after the last reboot. It did not make much sense as the two things seemed unrelated.

What happened was that the program relied on the `ARP` system call, which consists in a table where the MAC addresses of the neighbour nodes are stored; the problem was that the entries of the table are reset at every boot, and they are compiled whenever there is a communication between the PC and the related node of the network. So, at the start, the `sample-app-taprio.c` program would look at the table, wouldn't find any MAC address corresponding to its target IP address and as a consequence would stop. The problem was partially solved if we first called the `iperf3` utility because it updated the arp table automatically. Actually even a simple ping was enough to update the table.

So we decided to update manually the arp table on `pc-a` with the MAC address of `pc-b`. Recall that we organized the commands in a series of scripts to be called; right before invoking the `scheduler.py` program, which dealt with the installation of TAPRIO or MQPRIO, we inserted the following command:

```
arp -s pc-b e0:dc:a0:65:01:c5 (3.9)
```

where `pc-b` was the name in the system associated with the IP address of `eno2` on `pc-b`, i.e. `192.168.3.33`, while the following was its MAC address. In this way, since this script had to be launched before `sample-app-taprio.c`, the arp table was always updated when it was needed.

We actually found a similar problem on the receiving end. The `sample-app-taprio.c` couldn't get the incoming messages unless the `tcpdump` utility was active. We used `tcpdump` to listen to the port and produce the `.pcap` file that could later be opened with Wireshark, so again the two things did not seem directly related. However, apparently, the program relied on `tcpdump` to listen the time sensitive packets. Since we thought that this issue was in a way similar to the first one, and due to a lack of time, we did not dig any further trying to solve it.

3.7 Summary

This section concludes the chapter dedicated to the presentation of the tests about TSN with a brief overview of the objectives and the results obtained; in the end we are going to provide a clear list of all the requirements and features needed to perform the test, and in general to implement a time sensitive network.

There aren't, on the internet, a lot of tutorials and demonstrations concerning an actual implementation of time sensitive networks, so in the beginning it was difficult to find something helpful to start with. Luckily, Intel has provided a complete set of tests about TSN, along with a detailed user guide. We chose to follow and repeat these tests since they basically implemented what we had in mind as a first topology: two devices, in the most simple of networks, exchanging TSN information. Furthermore the required hardware and operating system (Linux) were compatible with what we had.

We explained in the second chapter that TSN is a set of standards defining some mechanisms and strategies, both hardware and software, that can be implemented in a network to obtain determinism in the transmission times of packets, which otherwise would be subject to a lot of random effects and disturbances. The abovementioned standards can be grouped in three main sets: time synchronization, time aware traffic shaping and system configuration.

The tutorial from Intel allows the user to test the first two key elements of TSN, namely time synchronization and time aware traffic shaping. System configuration cannot be correctly evaluated because the topology of the test is trivial and also because the current solutions are still vendor-dependent and not so much generalized.

So we started by installing the software component that allowed two or more nodes in a network to reach consensus on the time measured of the individual system clocks: this software implemented the Precision Time Protocol. We correctly measured that, among the two nodes that we had available, the maximum offset between the two system clocks did not exceed half of a microsecond, which is an outstanding result. The protocol is already implemented and can be enabled by means of a user-friendly utility in Linux; in particular it is provided with a Best Master Clock Algorithm which automatically selects the best candidate on the network to assume the role of master. This feature allows, for instance, the user to create the same script and call it at the boot in every node of the network; as a result, after about 20 seconds, the whole network will be completely synchronized with the highest of accuracies. Precision Time Protocol is probably the most useful feature discovered in this work, at least at the moment, due to the fact that the time sensitive networks technology is currently waiting for the standards to be finished and for the vendors' support to be updated.

We then discovered that the main mechanism upon which time sensitive network are based, namely the time aware shaper, could be implemented in Linux by means of a queueing discipline, which is basically the criterion adopted by the operating system to sort the packets present in its output interface. We verified that, if properly installed and configured, this mechanism could actually grant limited transmission time and, most importantly, interpacket latency in the communication between our two PCs, especially in presence of disturbance traffic on the same port. We also made comparisons with other queueing disciplines and traffic shaping mechanisms, and were able to successfully prove that, in order to reach the level of determinism sought in the standards, the time aware shaper is a necessary feature, as any other method is way too sensitive to disturbances and random effects. We experienced, though, compatibility problems and issues deriving from the direct use of this new queueing technology, which means that it is not that trivial to employ and the ecosystem as well has to evolve to fully integrate a time aware shaper.

As we said, the last key element is completely missing from the test, and in particular the part related to the scheduling of the gates in the network. In our case this task could be performed by hand, due to the very simple nature of the tests.

In conclusion we can state that the results obtained were satisfying and useful to better understand the features of Time Sensitive Networking.

3.7.1 Requirements

In order to perform our tests we had to get ourselves two computers with the following hardware features:

-
- Network Interface Controller Intel I210: this device is needed as it is one of the few supporting TSN. In particular it supports the hardware offload mode for the Earliest Transmission-Time First queueing discipline, which allows to have determinism in the actual sending times of packets
 - processor belonging to the Intel Apollolake family. In particular our model was a E3940 cpu, with the possibility of performing some optimization operations to maximize the performances at the expense of reliability. In general every example of TSN application employs an Apollolake cpu.

Instead, on the software's side we had:

- Linux Debian 5.9. The version of the kernel of the operating system has to be at least 5.3 in order to support the newest queueing disciplines such as TAPRIO. Furthermore the module `sch_taprio` must be enabled in the configuration file of the kernel, so that the `qdisc` can be successfully installed on one (or both) output ports of the system
- `ptp4l` and `phc2sys`. These are the two utilities necessary to perform the time synchronization on any Linux-based computing unit.

Additionally, not relevant to TSN in general but to the execution of the tests, we had:

- the utility `iperf3`, which was used in client mode to generate huge amounts of packets that acted as best effort traffic; it was used in server mode in the listening device
- `tcpdump`, used to monitor the activity at the receiving port of the listening device. The utility produced a `.pcap` file that could be opened with the program `Wireshark`, which allowed to plot the data in a more comprehensible way
- software from Intel's account on GitHub. In particular we exploited two programs, named `scheduler.py` and `sample-app-taprio.c`.

Chapter 4

Simulations

The third chapter is dedicated to the presentation of the simulation work that has been done about Time Sensitive Networks. This simulation part was developed after the practical tests described in the previous chapter; the main goal was to build a virtual setup which would make it easier and cheaper to test and verify the behavior of larger and more complex networks, as it is one intrinsic property of any simulation. In particular, the feature that we are going to focus on and that we want to test on a large and generic network is the scheduling, which, as we mentioned, is one of the least treated topics concerning time sensitive networking.

There is a good amount of references on the Internet describing several different simulation tools and strategies useful to be used in our framework. In particular, [16] provides a very useful comparison between some simulation methods, highlighting for each one advantages and drawbacks.

Method	Completeness	expansibility	Ease of use	Repeatability	Economy
Physical Simulation	M	L	M	H	L
Hardware in the loop simulation	H	M	H	H	L
OPNET	H	M	M	M	M
NS-2	H	M	M	M	M
NS-3	H	M	H	M	M
OMNET ++	M	M	H	M	H
MATLAB Simulink	L	H	H	M	H
Monte Carlo	L	M	M	L	H
Collaborative Simulation	H	M	L	M	M

Figure 4.1: Comparison between different simulation methods

Figure 4.1 shows such a comparison: among the listed methods we are going to focus on OMNET++ and, most of all, on MATLAB/Simulink. OMNET++ is a simulation library and framework, based on C++, which is particularly suitable for the simulation of networks in general. As a matter of fact, several papers exist which describe some kinds of simulations carried out about time sensitive networks such as [17]. In this paper the simulation

environment is described; OMNET++ provides the basic building blocks that implement the time gating mechanisms and so allow to create and simulate simple networks. The results of these simulations show that using a time aware shaper, instead, for instance, of a queueing system based on strict priority, grants better performances in terms of latency and determinism for TSN data. These simulations focus primarily on the gating mechanisms, such as the Time Aware Shaper, neglecting the time synchronization protocols that, as we described in section 2.3.1, are a necessary feature of TSN, assuming that a time synchronization is already implemented and works perfectly. Even though that's a reasonable assumption, supported by the experimental tests that we presented in chapter 3, we still have to keep in mind that time synchronization is not free. Furthermore, due to the limits of PTP, we are going to be constrained to consider limited-sized network by default.

The key feature of these kinds of simulations is that they are "events" simulations. Unlike usual simulations of physical systems, the protagonists of these simulations are discrete and finite entities which are subject to instantaneous events. Events can create, move and, in general, manipulate the entities, which in turn have the ability to "move" within the simulation environment. There is no need for time-varying continuous state variables whose evolution is described by some integral equations system as, e.g., there is in a mass-spring-damper setup. It is clear that a events simulation is more suitable for a time sensitive network as we can associate to each real flow a virtual entity, and program its motion within the network using events. These concepts will be explained in detail later, but the goal of the first introductory part is to highlight the differences in the simulation environment needed to simulate a time sensitive network.

The main drawback of OMNET++ is its low expansibility, most of all in terms of coding new functions. Recall that our final objective is to program and test a scheduling algorithm, therefore we need quite high computational capabilities from our simulation environment to implement one of the algorithms described in section 2.4.

Mainly due to this reason, we chose to use the MATLAB/Simulink platform to implement our simulation; not to mention that our knowledge and experience about the particular software was far better with MATLAB/Simulink than with OMNET++.

The main issue about MATLAB/Simulink, however, is its low completeness in the matter of time sensitive networks, as it can be also gathered from Fig. 4.1: the software is provided with event simulations libraries, but they are not specific for TSN simulations. Therefore, even though the results of a simulation about TSN would be the same and with the same properties, the starting point would have to be at a lower level; this means that we will first have to create from scratch the basic building blocks for TSN, and then use them to create our test networks. In OMNET++ those blocks are integrated

in the libraries that can be downloaded from the website, which are called the "INET framework".

As a result we can state that the general goal of this chapter is duplex. First we need to exploit the existing SimEvents library available in MATLAB/Simulink to create a set of simulation tools suitable for a time sensitive application; even though the concepts and mechanisms that we are going to implement are quite simple on paper, the new simulation libraries will have to be tested and evaluated, in particular by comparing the results of the simulations with the experimental ones reported in other papers or the ones actually obtained in first person, described in the previous chapter. Eventually, we are going to use the simulation to test and evaluate the behavior of a proper TSN network in the scheduling process of a set of input tasks.

The chapter is then structured in the following way. The first section is dedicated to the implementation of the simulation tools on MATLAB/Simulink; as we mentioned, this part is also provided with a series of tests to validate the effectiveness and accuracy of the simulation. The second section presents the implementation, using the MATLAB's environment, of one complete scheduling algorithm, in particular the No-wait packet scheduling presented in section 2.4.2. The last section can be considered as an integration of the first two as it reports the results of several simulations conducted using the libraries created in section one to test the algorithm implemented in section two. The results and the data gathered will be useful to support some additional considerations about No-wait packet scheduling algorithm, also in comparison with the other scheduling algorithm that we have studied (see section 2.4.1).

4.1 Simulation libraries on MATLAB/Simulink

MATLAB is a programmable environment developed by Mathworks that allows to perform complex matrix computations, plot functions of data and implement algorithms. Simulink is an additional package with a graphical interface which allows the simulation of multidomain physical systems. They are widely used in the academic world and also in the automation bachelor and master's degrees.

The two programs share their workspace, but actually the simulation of a time sensitive networks will feature for the great part the libraries of Simulink, in particular the library named SimEvents. As we mentioned, an event simulation is what's required to correctly simulate a time sensitive network, and SimEvents is the Simulink's library that contains the necessary tools.

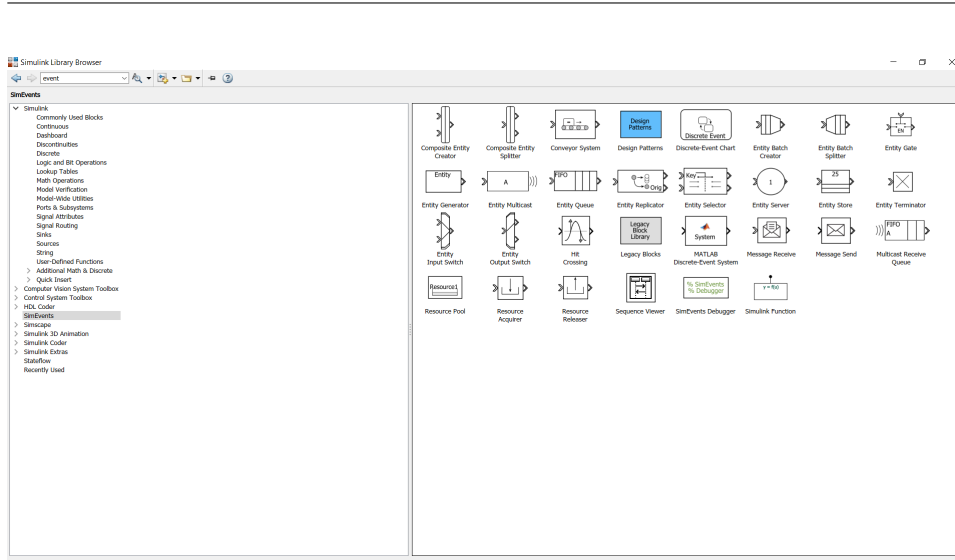


Figure 4.2: The SimEvents library

Figure 4.2 shows the components of the SimEvents library on Simulink. In particular, the most used components will be the following:

- Entity generator, in order to produce the entities of the simulation
- Entity input switch, that allows to merge different flows of entities in one, according to some rule that has to be specified
- Entity output switch, with a dual function with respect to the previous one, namely to separate flows according to some rule
- Entity FIFO queue, which implements a simple queue where the highest output priority is assigned to the packet that arrived first, and so on
- Entity server, which is useful to create delays in the transmission of entities
- Simulink functions, needed to implement some additional functions, also with the help of MATLAB functions

Before getting to the details of Simulink, however, it is necessary to describe a very important function performed by a MATLAB script. When we simulate a network, one of the first things that need to be specified is related to the topology of the network itself, namely the connections between the nodes. In order to efficiently store this information, a matrix is used, usually called adjacency matrix: an adjacency matrix is a square matrix with dimensions equal to the total number of nodes, which has a zero in correspondence of every couple of nodes that are not connected while it has

ones otherwise. Actually it is possible to substitute the ones with other useful information about the specific connection, for instance with the travelling distance between said nodes.

So, once we have a proper adjacency matrix describing the configuration of the network, the first thing to do is to solve the routing problem, namely determine the routes that packets will have to follow in order to get from their sources to their intended destinations. The routing problem is addressed in the next subsection.

4.1.1 Routing

By definition, a routing problem takes as input the information about the topology of a network, in our particular case, and gives as output the routes connecting all the possible combinations of nodes. A route is an ordered sequence of nodes such that between each couple of consecutive nodes there exists a connection and such that, following that sequence, it is possible to correctly reach the destination node starting from the source one.

In general, a routing problem is found as an optimization problem which tries to minimize some cost function; the most common cost is the length of the routes, which in turn gives as a result the shortest paths. There are, however, also algorithms that take into account the overall set of tasks that need to circulate in the network and produce solutions that minimize the congestion in the network, which seldom coincide with the shortest paths.

For large networks, such as the Internet, this problem can become quite complex and requires special addresses (IP addresses) which contain additional information about subnetworks and so on and so forth. Due to the limits imposed by the accuracy of the Precision Time Protocol explained in section 2.3.1, however, in a time sensitive framework the networks are way smaller with respect to the Internet; therefore the routing problem is tractable and very easy to solve. In particular, for the simulation the Dijkstra's algorithm has been exploited. Dijkstra's is a quite famous operative research algorithm that allows to efficiently find the shortest paths in a network of nodes.

Dijkstra's algorithm computes the least-cost path from one node, one at a time, (the source, which we will refer to as u) to all other nodes in the network. Dijkstra's algorithm is iterative and has the property that after the k th iteration of the algorithm, the least-cost paths are known to k destination nodes, and among the least-cost paths to all destination nodes, these k paths will have the k smallest costs. It is important to define the following elements:

- $D(v)$: cost of the least-cost path from the source node to destination v as of this iteration of the algorithm.

-
- $p(v)$: previous node (neighbor of v) along the current least-cost path from the source to v .
 - N' : subset of nodes; v is in N' if the least-cost path from the source to v is definitively known.

Afterwards we have to perform the following computations in order to obtain all the routes from one node to all the other nodes. We have to initialize the vector N' with our starting node, and the vector D with the distances of the neighbors of the starting node. Then we have to select between the remaining nodes the one closer to the starting node, namely the component of D with the lowest distance; we have to add this node to the list of already visited nodes N' and then update the vector D with the smallest distance of the nodes from the starting one. Now this distance can be the original value or another one taking into account the node just added to N' . We need to iteratively repeat this process until the set N' contains all the nodes in the network.

When this loop finishes we have in the set N' all the nodes, and for each node we have its predecessor in the least cost path from the source node. So we can extract the information about the routes by proceeding backwards, namely starting from all the possible destinations and reconstructing the path by adding iteratively the predecessors, up to the one source node.

The process just described has to be repeated considering every node in the network as the starting node; eventually we will have produced the complete set of routes.

In the next page, a simpler pseudocode version of the algorithm is shown, with the purpose of helping the understanding of the basic principles.

Algorithm 3: Dijkstra's algorithm

```
function Dijkstra(Adjacency matrix);  
for each node  $u$  do  
    for each node  $v$  do  
         $N' = \{u\}$ ;  
        if  $v$  is a neighbor of  $u$  then  
             $D(v) = \text{adjacency matrix}(u,v)$ ;  
        else  
             $D(v) = \text{inf}$ ;  
        end  
    end  
    while not all nodes in  $N'$  do  
        find  $w$  not in  $N'$  such that  $D(w)$  is a minimum;  
        add  $w$  to  $N'$ ;  
        for each neighbor  $v$  of  $w$  and not in  $N'$  do  
             $D(v) = \min(D(v), D(w) + \text{adjacency matrix}(w,v))$ ;  
        end  
    end  
    routes = reconstructPaths( $N'$ );  
end  
ROUTES = all the routes for each node in the network;  
return ROUTES
```

We created a MATLAB function implementing the Dijkstra's algorithm, named `Dijkstra.m`. Since, for our purposes, we did not need to know the shortest paths connecting the switches, but we needed the information only about the hosts, we chose to repeat the process only for the hosts. We stored the routes in a three-dimensional matrix that can be accessed by inserting in the first dimension the Id of the source node and in the second dimension the Id of the destination node: the result is the third dimension of the matrix, which is a row vector containing the ordered sequence of Ids that make the requested path. As we said, the input of the function has to be the adjacency matrix and some additional information about which nodes are the hosts, for the reasons just explained.

Our function also lets the user choose whether or not it has to show the progression of the computation, which is a feature useful for the large networks' routing solutions. For the sake of completeness we report, in fact, the approximate execution times of the algorithm as a function of the number of nodes in the network:

- 1 second for 100 nodes
- 8 seconds for 200 nodes
- 29 seconds for 300 nodes

- 86 seconds for 400 nodes
- 200 seconds for 500 nodes

We can notice that the execution times increase exponentially with the number of nodes but, as we already said, it does not constitute a problem as the size of TSN networks has to be limited.

4.1.2 Packets

Let's resume now the discussion about what happens in the Simulink's environment. We are going to treat in this subsection the entities, namely the discrete units that represent the actual packets. In SimEvents, entities can be characterized by several different parameters or fields, which can then be used and manipulated during the simulation; SimEvents also gives us the possibility to use user-defined entities. In the next picture we show the generation block of the entities, which creates instances of our type of entity.

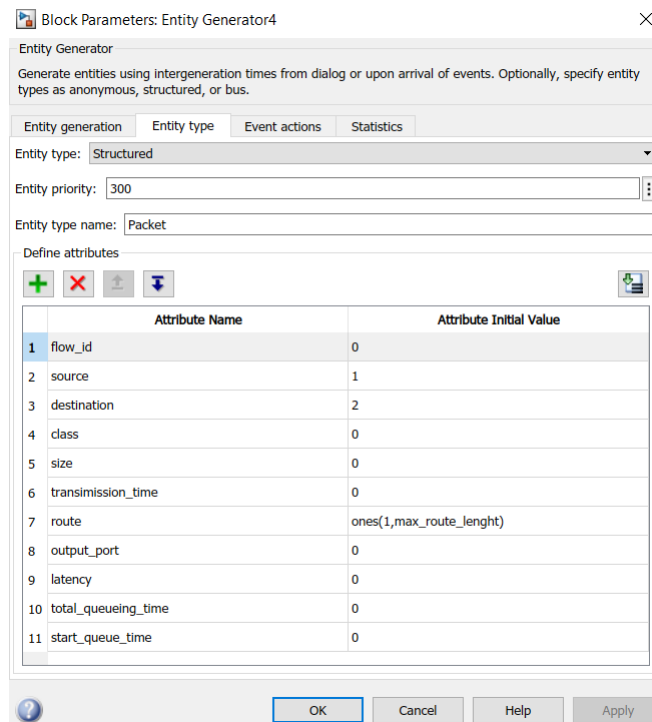


Figure 4.3: Configuration of a generation block

We can see that the configuration panel of the block is divided in tabs. On the current tab the user can specify name of the entity (we chose "Packet") and the set of fields that we mentioned earlier; in SimEvents these fields are called attributes, and can be any kind of variable: integers, real numbers,

vectors or matrices. By the names of the attributes one can already guess their purpose in the simulation. Source and destination fields are going to contain the initial and final node for each packet; class will specify whether the packet belongs to the TSN set or it is a best effort packet; route will contain the route of the packet computed before (and as a matter of fact it is a row vector); the last three attributed aren't strictly necessary to the simulation itself but are used to gather data, such as the amount of time that the packet is kept waiting or its latency.

Attributes need to be initialized in every generation block but can be modified by means of "actions". Actions are simply some lines of code that are executed whenever a packet arrives to a block, when it is waiting inside the block or when it leaves the block; with block we mean the main ones used and shown in Fig. 4.2, such as queues, switches and so on. The user can decide which actions to insert in the simulations and where to put them. Every configuration panel for every block has a tab where the actions specific to that block can be coded, as it can also be seen from Fig. 4.3.

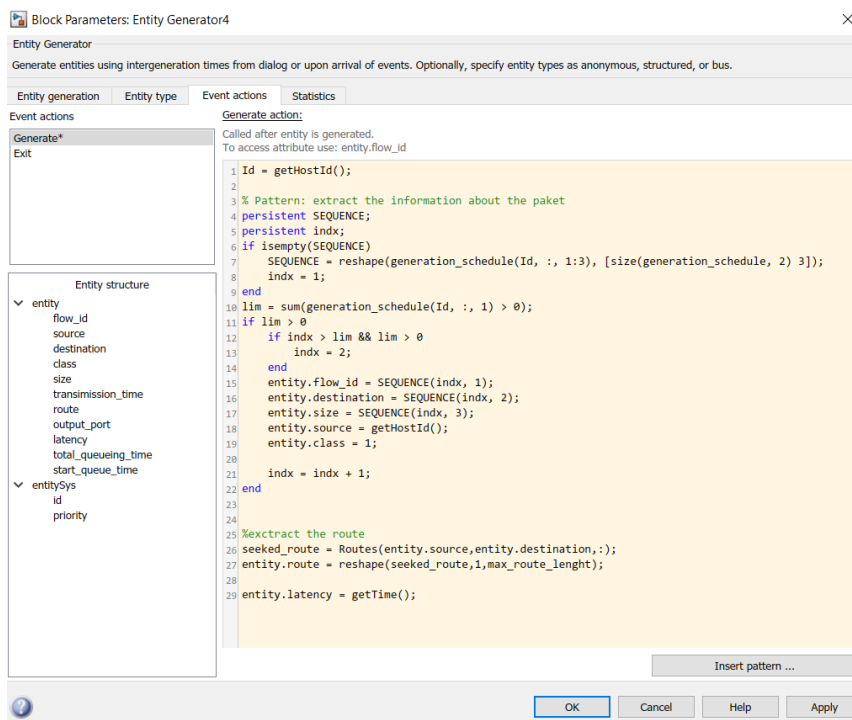


Figure 4.4: Configuration of a generation block

In the figure the tab related to the actions of the configuration panel for the packets generation block is shown. We can see on the top left that event actions can take place either at the generation time or when the packets leave the block, or even in both cases; the action itself that we have programmed

is divided in two parts: in the first part we scan a variable named `generation_schedule`, to extract the information about the packet that has to be sent out next. In this variable we store the ordered sequence of packets that need to be generated; the variable, which is clearly a matrix, also contains the flow Ids, destination and size. In the second part of the action we use the information about source and destination to access the routes matrix and extract the specific route that we need.

Every piece of information that we have extracted from the variables of the workspace is in turn stored in its dedicated field of the packet, so that it can be brought around with it.

The `generation_schedule` also contains the starting time of each packet; these data are used in the first tab of the panel, where the generation problem is addressed. In this tab we can code another action which computes the intergeneration interval between a packet and the next one, in order to correctly send them out with the right timing. Thanks to the persistent variables *SEQUENCE* and *indx*, this procedure is cyclic, meaning that after the packets in `generation_schedule` have been all produced, the block starts again from the beginning.

In the next subsections we are going to describe the key elements or subsystems that make the library for TSN simulation, all of them featuring the basic elements listed before.

4.1.3 Input port

The first complex subsystem that we describe is the input port, which is also useful to fully close the discussion about routing and how it is implemented. This subsystem takes as input a one-directional flow of packets, which can belong to every traffic class; it is the representation of an ethernet input cable. The purpose of the subsystem is to accept the incoming packets and redirect them towards the appropriate output port, simulating the computations that actually take place in a switch.

We make the incoming packets accumulate on a FIFO queue where some actions concerning routing are performed. In order to expand the possibilities of what can be done in the action, we exploit Simulink functions, which allow to create Simulink subsystems and use the computed output as a parameter in the action. In particular, we can import parameters from the simulation inside the action environment using Simulink functions; as Fig. 4.5 shows, we define a Simulink function named *redirectPacket()*, whose output is then going to be used in the entry action of the queue; said function takes as additional inputs two variables from the simulation, namely *Switch_Table* and *Switch_Id*. What happens inside the block is that each incoming packet is analyzed, and in particular its route field is extracted: it is searched to find the position of the switch on the route, using the information about the switch Id; then the next node on the route is known, and the associated

output port is written in the homonym field of the packet. In order to figure out which of the ports of the device assign to the packet, the function examines the switch table, which contains the ports of the switch and the Ids of the nodes connected with them; knowing which is the next destination it is easy to extract the information about the output port.

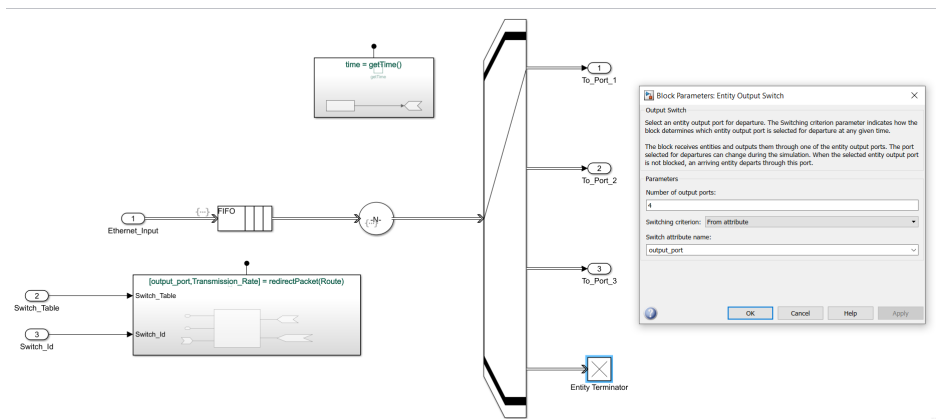


Figure 4.5: Input port

The downstream server is used to impose a delay to the transmission of the packet, which simulates the time that the computing unit of the switch takes to compute the next destination. This delay can be set directly from MATLAB for any input port, and we used the value of $5 \mu s$.

Eventually the output switch is used to divide the packets according to their "output port" field, which was previously filled with the right output port. Figure 4.5 shows this configuration of the switch; then, the single outputs of the switch will be the inputs of the respective output ports. The word switch in this context is used to mean the component of the SimEvents library instead of the component of networks.

4.1.4 Output port

The most important subsystem is the one implementing the output port, as the output port is the key and fundamental place where TSN mechanisms are implemented, particularly the time aware shaper.

The purpose of the output port is to take the packets from the input ports in the device which have been redirected to that output port; then, according to the TSN standards, packets have to be divided in several queues according to their traffic class. The distinction that we are going to consider is between TSN data and best effort data, meaning that we have two queues and two traffic classes. A downstream gating mechanism then chooses which queue gets to send its highest priority packet into the communication medium; as a consequence the output of the port is the packet that actually leaves the

device.

The next figure shows the implementation of the output port

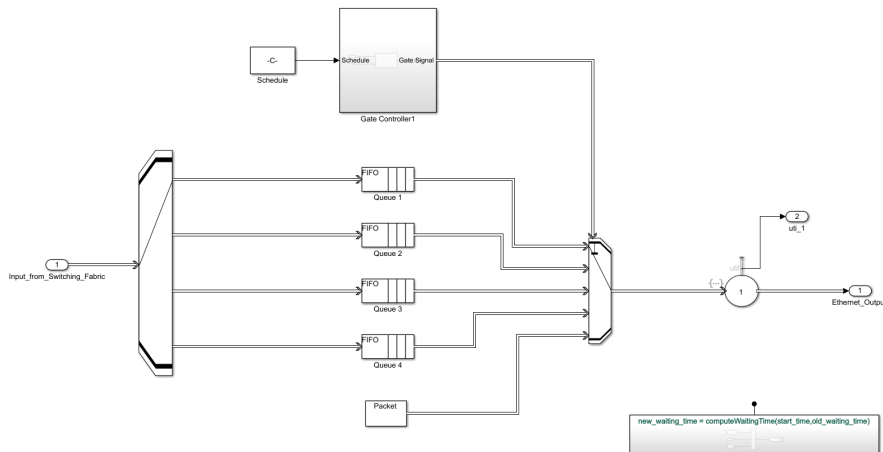


Figure 4.6: Output port

We can directly compare Figure 4.6 with Figure 2.14 and immediately appreciate the similarities. As we said, packets get distributed among the queues by means of an input switch, which redirects the packets in the queues according to their class attribute; the output switch after the queues is controlled by a signal generated by another subsystem, called *Gate Controller*, which in turn is driven by a variable named *Schedule*. The *Schedule* has the following format, which is very similar to the one of Figure 2.14:

```

Command Window
>> reshape(Schedules(1,1,:,:), [151 6])

ans =

      0      0      0      0      0      0.0001200000000000
1.0000000000000000      0      0      0      0.0001200000000000 0.0000060000000000
      0      0      0      0      0.0001260000000000 0.0000010000000000
1.0000000000000000      0      0      0      0.0001270000000000 0.0000090000000000
      0      0      0      0      0.0001360000000000 0.0000790000000000
1.0000000000000000      0      0      0      0.0002150000000000 0.0000040000000000
      0      0      0      0      0.0002190000000000 0.0000340000000000
1.0000000000000000      0      0      0      0.0002530000000000 0.0000040000000000
      0      0      0      1.0000000000000000 0.0002570000000000 0.0007430000000000
      0      0      0      0      0.0010000000000000      0
      0      0      0      0      0      0
      0      0      0      0      0      0
      0      0      0      0      0      0
      0      0      0      0      0      0
      0      0      0      0      0      0

```

Figure 4.7: Schedule

This variable is a matrix with many rows and six columns. Each row represent a time window, while the columns contain the following data:

- the first four column contain the combination of open and closed gates, where the first column usually represents the TSN queue and the others represent the best effort queues

-
- the fifth column contains the offset on the period at which the window starts
 - the sixth column contains the duration of the window

The details and the proper considerations about how this matrix is filled will be presented in the next sections; for the time being what we want to highlight is just the protocol by means of which we can communicate the schedule to the gate controller.

In the *Gate Controller* subsystem this input schedule is analyzed and the control signal for the output switch is generated. If we consider the example of Figure 4.7, the control signal will order to the switch to close all the gates for the first 120 μs of the period, then open the TSN queue for the next 60 μs and so on and so forth.

The downstream server is quite important because it performs several functions. First of all, just like the server of the input port, it simulates the delays in the transmission of packets: in this case it is the time that each packet takes to arrive to the next node in the network. The delay is a function of the size of the packet (one of its attributes), of the transmission rate of the port and on the physical distance between the nodes. The transmission rate of the port is extracted by means of a Simulink function, similarly, again, to what happened for the input port, while the physical distance is assumed to be lower than 30 meters; as a consequence the maximum propagation delay would be of 1 μs .

The second function performed by the server is the update of the attribute related to the waiting time of packets. In the server of the input port, the equivalent of a timer is activated, useful to measure how much the packet is constrained to wait in the switch. This is simply done by writing the exit time from that server (as an exit action) and subtract that time to the time when the packet actually gets past the queue, namely in the entry action of this server. Once the waiting time for the current switch is known, it will have to be added to the previous waiting time, which is stored in the homonymus attribute of the packet.

A side note about the number of queues: we said that we chose to use only two traffic classes and two queues, but in the simulation the queues are four. The number of queues that the user chooses to exploit is not related to the number of queues actually implemented in the system: our choice will simply produce schedules and packets that will affect the first and the last queues, leaving the two middle queues idle at all times. As a matter of fact we can see that the second and third columns of the schedule are always zero, meaning that the associated queues are always closed.

Finally, the fifth entry of the output switch is an entity generator which does not ever produce packets. This because the simulation block cannot intrinsically close all its gates; therefore when we require all the actual gates

to be closed, we open the one connected with this packet generator, which has an infinite intergeneration time, which gives the same result.

4.1.5 Switch

The union of an input port and of an output port makes a port. Several identical ports make a switch. A switch is then a big block which can take in as input a number of flow equal to its number of outputs; both are in turn equal to the number of ports that it is provided with. Packets that arrive at the switch get automatically redirected to the next node specified in their route; this is the exact purpose of an actual switch. As we said, we even take into account delays due to the computations that actually take place inside real switches. Our switches are also TSN-capable, as their output interfaces are provided with the gating mechanisms described in the TSN standards, and so they need proper schedules to be driven according to the desired behavior.

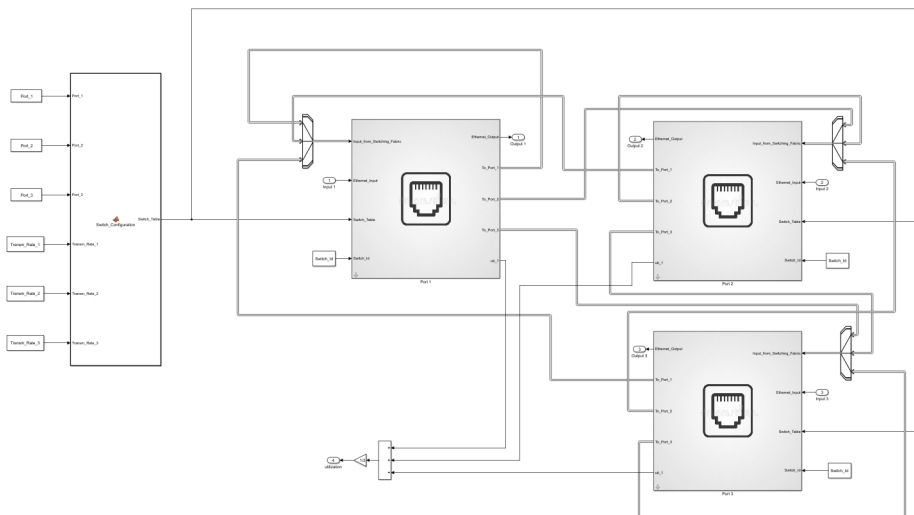


Figure 4.8: A three ports switch

Figure 4.8 shows how a switch is implemented. For the sake of clarity we started to use masks for the subsystems, in order to give a visual idea of what they do. In this case the ports are represented with the image of an ethernet port, because that is what they simulate. The one shown in the example is clearly a three-ports switch as it is featured with three ports. We can see that they are all interconnected so as to allow packets incoming to every port to go out on every port.

In addition to the ports, this subsystem is characterized by a MATLAB function called *Switch_Configuration*: the output of this function is the *Switch_Table* that we already seen in the section dedicated to the input

port. As a matter of fact, this variable is propagated inside every port, where will be directed to the input port and used to perform the routing. The input of the MATLAB function are constants that can be tuned from the mask of the switch itself. We are not going to show it, but by double clicking on the big switch subsystem, which contains the elements in Figure 4.8, a configuration panel opens: inside this configuration panel the user can specify the neighbors of the switch in correspondence with its ports, along with the transmission rates of the ports themselves. As we have seen, this information is appropriately stored and used inside the ports.

By modularly duplicating ports it is possible to create switches with multiple ports. The only thing that has to be kept in mind is that a suitable switch table has to be produced, which takes into account the exact number of ports and is able to correctly communicate the mapping between them and the neighbor nodes. As the number of ports increases, the visual complexity of the subsystems increases, along with the number of connections and lines, making it difficult to understand the roles of the elements. For this reason we show the three ports switch; for the purposes of the simulations, though, we have created switches with up to six ports.

4.1.6 Host

A host is a device with one single port with the duty of generating and receiving packets. We assume that hosts are placed at the boundaries of our networks and that they are connected only with one switch; through routing, is then possible to reach any other host in the network.

There is no need to show how a host is implemented because it is composed by an output port and an input port. Furthermore, since it does not have to redirect packets, the input port does not have any output; instead it is provided with a suitable function whose purpose is to record the data of every received packet and save, at the end of the simulation, those data in the workspace.

```

Command Window
>> Data_1(1:100,:)

ans =

Columns 1 through 8

    0           0           0           0           0           0           0           0
1.0000000000000000 10.0000000000000000 1.0000000000000000 11.0000000000000000 0.0000115000000000 0.0001527000000000 0.0000074000000000 0.0000287000000000
1.0000000000000000 10.0000000000000000 1.0000000000000000 36.0000000000000000 0.0000051900000000 0.0001601000000000 0.0000074000000000 0.0000271000000000
1.0000000000000000 8.0000000000000000 1.0000000000000000 9.0000000000000000 0.0000095999999999 0.0001604000000000 0.0000203000000000 0.0000334000000000
1.0000000000000000 5.0000000000000000 1.0000000000000000 43.0000000000000000 0.0000146999999999 0.0002453000000000 0.0000649000000000 0.0000443000000000
1.0000000000000000 6.0000000000000000 1.0000000000000000 49.0000000000000000 0.0000146999999999 0.0002823000000000 0.0000370000000000 0.0000443000000000
1.0000000000000000 2.0000000000000000 4.0000000000000000 0.0000263306315693 0.0003171000000006 0.0000287906315695 0.0000287906315695
1.0000000000000000 10.0000000000000000 4.0000000000000000 0.000284981474911 0.0004064000000000 0.0000892999999994 0.000371781474912
1.0000000000000000 2.0000000000000000 4.0000000000000000 0.0000321052160743 0.0004133000000024 0.0000669000000024 0.00033252160745
1.0000000000000000 2.0000000000000000 4.0000000000000000 0.000032129287907 0.0004159000000024 0.0000025000000000 0.00033129287907
1.0000000000000000 2.0000000000000000 4.0000000000000000 0.000350992062917 0.0004783000000029 0.0000625000000005 0.00037392062917
1.0000000000000000 2.0000000000000000 4.0000000000000000 0.000369605297970 0.0004997000000033 0.0000214000000004 0.000381005297970
1.0000000000000000 2.0000000000000000 4.0000000000000000 0.000366526447897 0.0005082000000034 0.0000055000000001 0.000381726447898
1.0000000000000000 10.0000000000000000 4.0000000000000000 0.000340787180692 0.0005199000000000 0.0000116999999966 0.000481587180693
1.0000000000000000 2.0000000000000000 4.0000000000000000 0.000403584464402 0.0005779000000040 0.0000580000000040 0.000427784464403
1.0000000000000000 2.0000000000000000 4.0000000000000000 0.000412701929401 0.0005873000000040 0.0000094000000000 0.000436501929401

```

Figure 4.9: Collected data

Figure 4.9 shows the format of the data which, as usual, are stored in a

big matrix. Similarly to the case of the schedule we are not going to make comments here on the specific data, but we are just going to describe how they are displayed. Each row represent a received packet by the host; the columns contain the following information, in order:

- Id of the receiving host
- Source of the packet
- Class
- Id of the flow
- Queueing time
- Arrival time
- Interpacket latency
- Latency
- Counter of the total number of packets received by the host (this column is not shown in Fig. 4.9)

Latency is computed using a Simulink function which registers the exact time when the packet arrives at the host, and then by subtracting to that value the starting time of the packet, which had been written in the latency attribute of the packet when it was generated.

4.1.7 Network

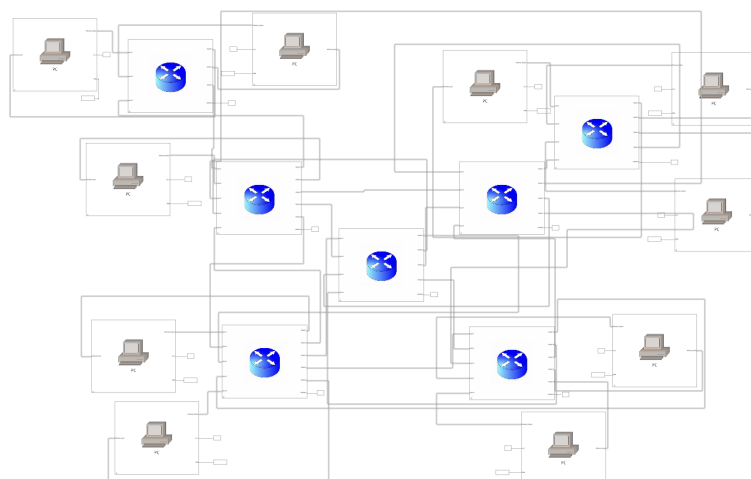


Figure 4.10: Generic network

Figure 4.10 shows a generic network built using the described simulation blocks for time sensitive networks, which have been created starting from the basic blocks of the SimEvent library.

It is clear that the icons and the connections required to make the simulation work become confused for large networks, and make it difficult to understand the topology at a first look. Therefore, in the section dedicated to the simulations, we will resort to other ways to properly describe the network and the problem that we are trying to solve.

Every Simulink file, such as the one shown in Fig. 4.10, is accompanied by a MATLAB script by means of which the user defines all the necessary variables and constants in the workspace. Some examples are the switching delay, namely the computation time required to the switches to transfer a packet from one input port to one output port, the duration and step size of the simulation, the maximum number of packets to be recorder by the hosts.

In addition, the cofiguration script contains the function implementing the routing algorithm; as we have seen we need to specify the adjacency matrix describing the topology of the network, and we get as a result a three-dimensional matrix storing all the relevant routes.

Eventually we need to specify a suitable schedule for every port of the system. This can either be computed by a scheduling algorithm or evaluated by hand and written in the appropriate variable, paying attention to the particular format required by the simulation and shown in Fig. 4.7.

4.1.8 Validation

After having completed the description of our custom library for the simulation of Time Sensitive Networks, we are going to provide in this subsection some sort of validation for such a library. In order to do so, we are going to compare the results of real tests with the results that the simulation environment gives, if we set it up to have the same configuration as the experimental benchmark. To this end, a natural set of tests that can be taken into account are the ones performed in first person, and described in chapter three, implementing the TSN tutorials from Intel. The focus on these tutorials, however, was not on the complexity of the topolgy, since the primary objective was to show the implementation details and the behavior of a time aware shaper; therefore the test itself, as a validation tool for our library, may be a little bit more simplistic. For this reason we are also going to consider another set of tests that have been conducted with a more complex topology and a more heterogeneous taskset; this test is described in [18]. In particular, this paper had indeed the purpose of supporting the simulation on OMNET++ of time sensitive networks, by comparing the results of the simulation with the ones of the experimental test. As a consequence it is particularly suitable to be adopted as a model to follow even in our case.

Tests of chapter 3

In the third chapter we presented this test from an implementative point of view, describing how to correctly set up the time aware shaper on Linux-based systems and analyzing not only the results but also all the implications of the new output interface. As far as the topology and the taskset are concerned, the test that we want to simulate is quite simple. We have a network made by two nodes, in particular by two hosts directly connected; one of them is a talker and the other is a listener, meaning that the talker device produces packets and sends them to the listener, which in turn has the duty of registering them.

Next we show the setup implemented in Simulink.

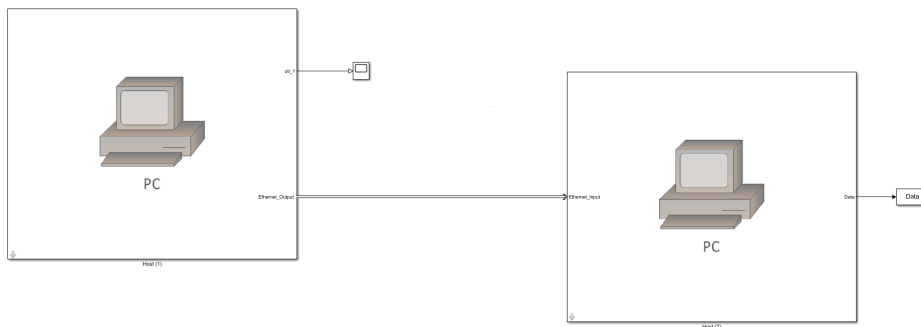


Figure 4.11: Generic network

Recall from the third chapter that the taskset was made by two different TSN traffic classes, namely class 3 and class 5; as usual, we also had the class of best effort data. We assigned to each one of these three classes a dedicated queue in the output interface of the talker device: in particular we used queues 1 and 2 for the TSN flows and queue 4 for the best effort packets. The third queue would have been used if the test had featured three TSN flows, but, since it does not, it remains idle at all times.

The application required to send, in nominal conditions, 2 instances of every TSN flow per cycle, which in the test lasted one millisecond. Therefore a total of four TSN packets had to be produced by the sending host every millisecond. In addition, the interfering traffic was made by large packets with size of 1500 bytes approximately, that had to be generated as much as possible. The dimension of the best effort packets was chosen to put ourselves in the worst possible scenario, as that is the largest ethernet frame that can flow in a network.

Just like in the actual test we put manually the schedule for the only relevant output port in the system, expressing the same windows using our format, which is slightly different with respect to the one required by the

actual time aware shaper.

We simulated the setup for one second, and the following are the results.

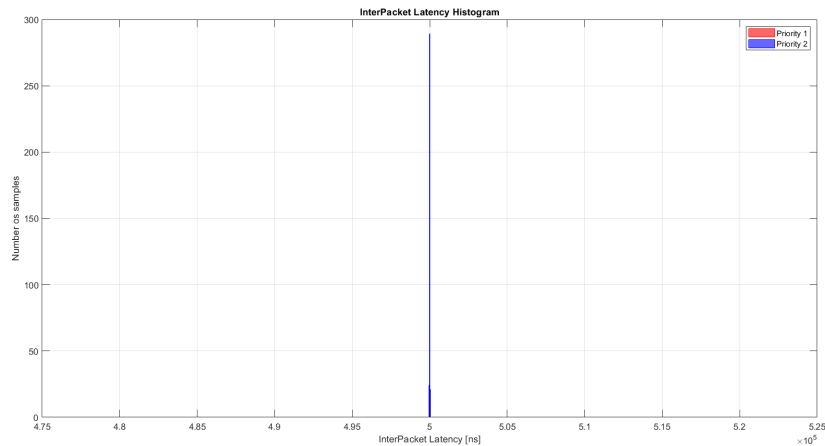


Figure 4.12: Measured interpacket latency

The first plot that we show is a histogram representing the interpacket latency, namely the time that passes between two consecutive arrivals of packets belonging to the same class. In the next plot we are going to show the same result but with a zoom on the horizontal axis.

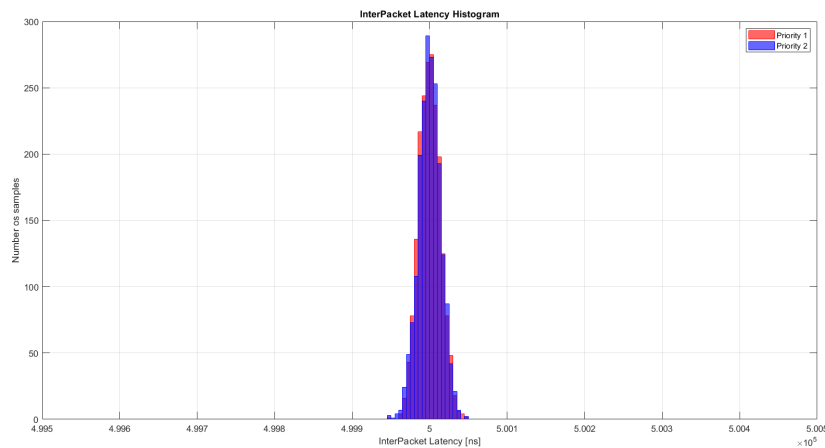


Figure 4.13: Zoom of the interpacket latency

By a quick comparison with figs. 3.18 and 3.18, it can easily be noticed that the shape of the data is pretty much the same. In almost every case we get values of interpacket latency very close to 500 microseconds, which was the expected value as it is the distance in the schedule between two

consecutive TSN windows.

We must also note that we had to add a randomly generated disturbance to the latency of packets, in order to simulate all the little losses of time that naturally and inevitably take place along the transmission. Without this disturbance we would get an exact 500 microseconds interpacket latency for every instance received.

We conclude the discussion about TSN data with the plot of the latency as a function of time.

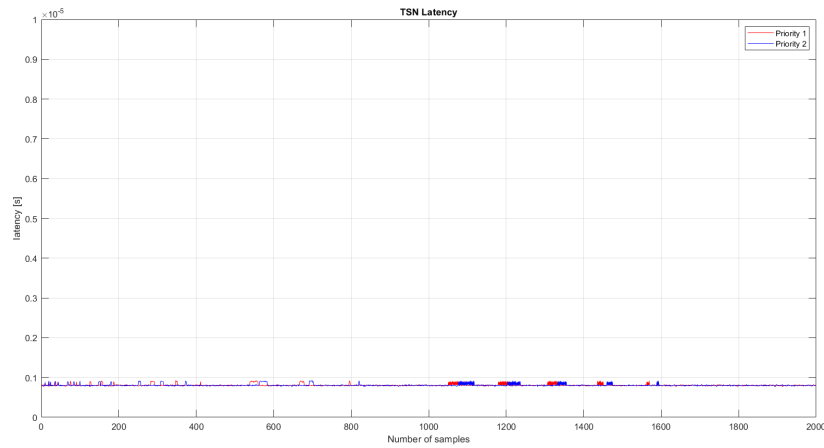


Figure 4.14: Measured TSN latencies

Then we show the measured latencies for best effort packets.

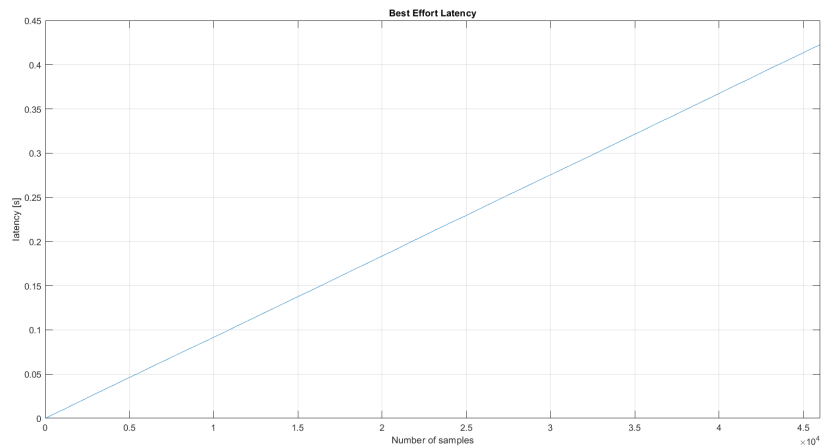


Figure 4.15: Measured best effort latencies

The two previous plots show that the latency of TSN packets remains

pretty much constant for the whole simulation, while the best effort latencies keep increasing their values. This means that the network is congested with packets, as the rate of best effort data does not fit the cycle; nonetheless this congestion does not affect the TSN data, that reach their destination with the intended frequency.

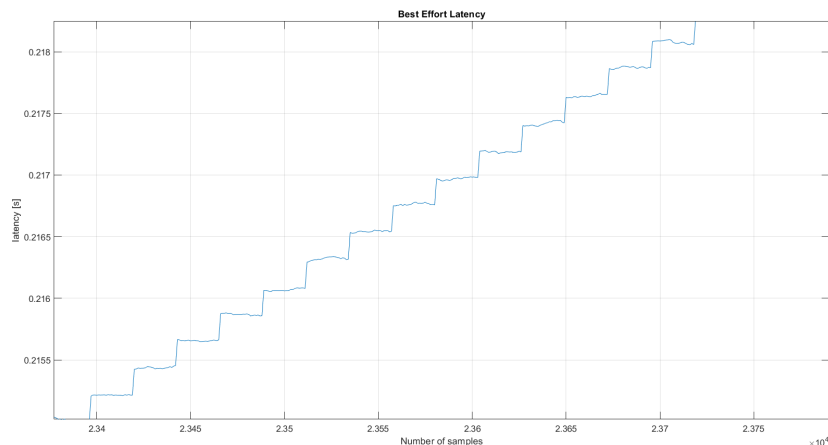


Figure 4.16: Zoom of the best effort latencies

Eventually we show a zoom of the latencies of best effort packets, as it is useful to make some additional considerations. We can see that this quantity increases in steps: it is practically constant for most of the time but has little intervals where it becomes way bigger. This behavior makes sense if we consider the cycle divided in two parts: the windows that allow best effort packets to pass and the windows that do not. During the windows belonging to the first type the best effort queue is emptied and it is filled at the same time with approximately the same rate; in the other windows this queue keeps being filled but cannot be emptied. This causes latencies to remain constant when best effort packets are transmitted and to increase drastically when TSN data are flowing instead.

In conclusion we can state that the results obtained with the simulation are very close with respect to the ones obtained experimentally. This shouldn't come as a surprise, since all the mechanisms that we implemented in simulation are conceptually rather simple: it is just a system of gates that can be opened or closed to let information pass or constrain it to wait. Since, as we already highlighted, the configuration and the topology of the test were this simplistic, we are now going to consider another set of tests in order to further validate our simulation environment.

A Simulation Model for Time-sensitive Networking (TSN) with Experimental Validation

The second set of tests that we take into account is described in [18], and consists in a network with 10 hosts and 2 switches. The details on the configuration and on the taskset are provided in the following picture.

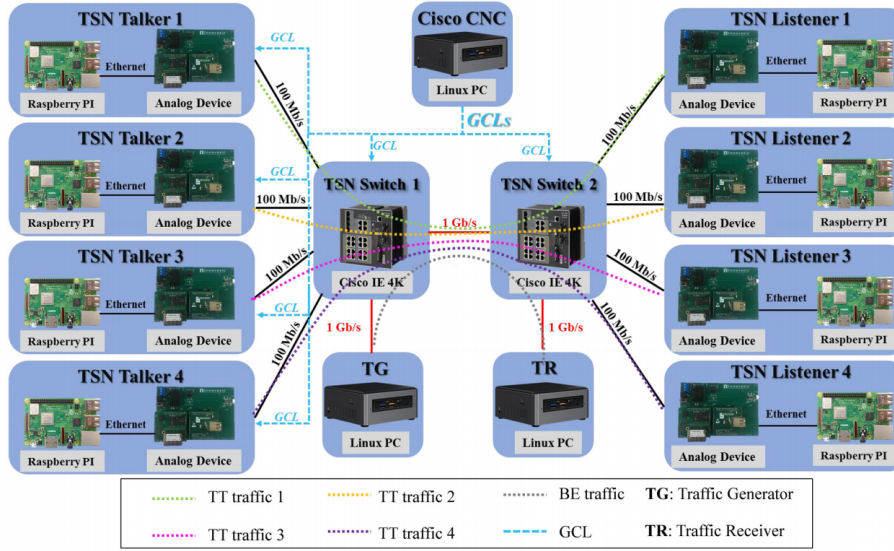


Figure 4.17: Experimental setup

We have five talking devices and five listening devices. The two groups are separated by a couple of switches in series. Of the talking devices, four are dedicated to the transmission of TSN data, while the last one has to produce the interfering traffic; the same goes for the listening devices. The links in the network do not have the same rates as the ones connecting the TSN hosts to the switches have 100 Mb/second, while the other links, namely the one between the switches and the ones for the best effort nodes, have 1Gb/second.

The traffic pattern is made by four TSN flows, each one generated by a different source and directed to a different destination; the application requires one instance for each flow per cycle. In addition, as we said, the traffic generator host continuously sends best effort packets to the traffic receiver host.

This particular test is meaningful because it creates multiple conflicts on the first switch, due to the fact that several packets need to use it in order to get to their destination; this scenario needs a suitable schedule to help said switch to handle all the conflicts on its bottleneck downstream cable. The test also presents some additional variability thanks to the fact that the rates of transmission are not the same; this feature requires the specific

configuration of each node within the simulation environment, by setting the downstream transmission rate for each port of each block.

We do not actually have to focus on the implementation details for our purposes, but we can notice that the switches employed in the network are produced by Cisco, which is a known vendor of TSN technology. Without entering in the details, in the paper it is explained that the system configuration tool provided within the switches allowed them to compute a suitable schedule just by inserting the characteristics of the traffic patterns; then, as Fig. 4.17 shows, the system configuration deals with the transmission of the gate control lists to each port. This is an example of the fact (explained in section 2.3.3) that, from this point of view, progresses still need to be made in the direction of procedure standardization: the methods adopted by Cisco both for the scheduling computation and for its distribution are proprietary and particular, therefore not accessible from the outside.

However, even though we do not get to know how it is been computed, the resulting schedule is provided in the paper to help the reader understand how it works, so we programmed our simulation to behave according to that schedule.

We implemented the network on Simulink and simulated the setup for a tenth of second; these were the results:

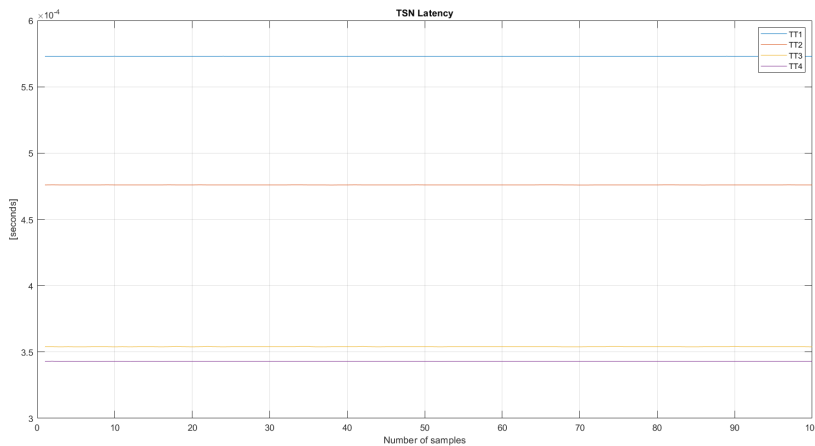


Figure 4.18: Latencies for TSN packets

The above graph shows a comparison between the latencies of the different time triggered flows. In the next graph we are going to show the latencies of the best effort packets. Both the plots have on the horizontal axis the number of samples, namely the ordered sequence of received packets.

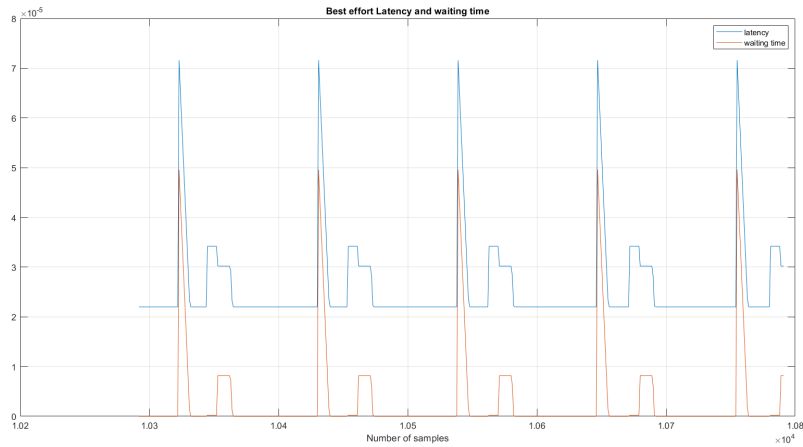


Figure 4.19: Latencies for BE packets

From figures 4.18 and 4.19 we can see that the latency of any kind of TSN packet is kept constant for the whole duration of the simulation, while the latencies of the best effort packets are affected by a high degree of variability. The reasons of this behavior are pretty much the same as in the previously presented example. In this case, though, the rate of generation of best effort packets is "affordable" by the network, which does not get congested as in the other scenario. That's why we do not see these latencies continuously increasing. They are, however, subject to the fact that TSN flows have a higher priority and dedicated windows; therefore, occasionally, the latency of some packets is affected by delays due to the fact that they had to wait the TSN windows to finish. In order to save memory, we registered only the last fifty samples of best effort traffic, but this behavior is clearly detectable; furthermore, from the values of the horizontal axis we can see that the total number of best effort packets overcomes the ten thousands samples.

The schedule provided from the paper is clearly not based on the No-wait packet scheduling algorithm, which has been described in section 2.4.2 and which will be actually implemented in the next section. If that were the case, since all the flows have the same path's length, we would have found exactly equal latencies. Instead, this schedule requires TSN packets to wait in their dedicated queues in the switches, namely they arrive a little while before the opening of their time window. Since we do not know the algorithm that generated the schedule we cannot comment further about it, but this is the reason why we have different values for the TSN latencies: clearly the flow named TT1 is the one which waits the most, while TT4 waits the least.

Another consideration about latencies is the reason why the latency of TSN packets is actually ten times higher with respect to the latency of the best effort packets. This behavior is due to the configuration of the network,

which provides 1Gb/second cables for the whole path of best effort packets while for $\frac{2}{3}$ of the TSN paths it only provides 100 Mb/second cables. Clearly that's a feature which is not under our direct control so we just have to accept that, on average, TSN packets will have higher transmission times with respect to the best effort packets.

Other than intrinsically making sense, these results are identical to the ones reported in the paper, both from the simulation on OMNET++ and, most importantly, from the experimental test. Therefore we can consider as successful the validation process for our library, which is conceptually simple but suitable to simulate with sufficient accuracy Time Sensitive Networks.

4.2 Implementation of No-wait packet scheduling on MATLAB

This section is dedicated to the description of the scripts and the functions that in MATLAB implement the No-wait packet scheduling. It is a heuristic algorithm that allows the computation of a suitable schedule in a generic network for a generic set of tasks. In section 2.4.2 we have given a brief and general overview of the main concepts, supported by the versions in pseudocode of the most important parts. Here we are going to focus mainly on the implementative details of the algorithm.

Recall that the No-wait packet scheduling is based on the assumptions that each packet has to be scheduled as soon as possible within the period and it never has to wait. Furthermore, we want to find the combination of tasks that minimizes the span of the solution, namely the arrival time of the task that finishes last. We are going to see that it is not possible to find what is properly called the optimal solution, so we will settle for the best among a limited set of possible solutions. Once we have our combination of tasks we need to produce a schedule that meets the requirements and the format of the specific time aware shaper which is supposed to implement the just computed schedule. So other functions are needed to integrate the result in the simulation or, in general, with the test network.

We implemented the calculations necessary to perform this work in a series of functions, which also define the structure of this section: first of all we have the timetabling function, which deals with the computation of the span for a single sequence; then we have the general NWPS_Heuristic function whose duty is continuously change the input of the timetabling function in order to find the minimum span solution; inside the NWPS_Heuristic, there are also other functions which deal with the actual creation of the schedule useful in the simulation.

The final objective is not only to actually implement one complete scheduling algorithm, but also to test and analyze its behavior by means of the simulation libraries described in the previous section.

4.2.1 TimeTabling

The core strategies of the No-wait packet scheduling algorithm are implemented in the `timetabling` function. We created a MATLAB function that, starting from a given sequence of the tasks to be scheduled, places the tasks in the network and computes span and critical flow. Recall that the critical flow is the task that finishes last, and whose arrival time is the span; recall also that we use the word "place" with the meaning of "occupying the nodes of the path of the task with TSN windows".

The file `TimeTabling.m` contains our function, which has to be called inside other programs in the following way:

$$\begin{aligned} [\text{generation, span, TimeTable, critical_flow, success}] = & \text{TimeTabling}(\text{Adjacency_matrix, Routes, TaskSequence, map, TaskSet, cycle_time,} \\ & \text{propagation_time, computation_time, accuracy}) \end{aligned} \tag{4.1}$$

In order for it to be the most general and configurable possible, several different inputs are necessary, allowing the user to tune the function in the most appropriate way. So we are going to consider the main steps of the function where the inputs are used to generate the outputs.

The input variable `TaskSequence` is supposed to contain the ordered sequence of tasks that the timetabling function has to place into the network. So one of the first things coded in the function is the call of a *for* cycle which starts from the beginning of such a variable, taking into account one task at a time.

Each iteration of the cycle manages, if possible, to place one task in the network. Therefore at the start we use the Id of the task to access the variable `TaskSet` to extract the information related to the task's source, destination and size. Then we use source and destination to access the variable `Routes` and extract the nodes that are going to be involved in the placement of the task, namely the components of its path. We also define and initialize the variable named `TimeTable`, which at the end will be given out as an output. This variable is a matrix with several rows and two columns, and it is used to indicate which time windows are currently occupied in the port it is referring to: so we are going to have a timetable for each port in the system. At the beginning all the elements of any timetable are -1, while during the execution of the algorithm any new window will be indicated by a non-zero row, where the first column will report the start time and the second column will report the finish time of that window. In order to accomplish its goal, in each iteration, we first need to check whether or not, with the current starting time, the considered task satisfies the constraint by accessing the timetable of each node along its path (without modifying it); then, if the task fits, we have to retrace the path again creating, in the

timetables, the new windows related to the current task. If, instead, we find that the path is somewhere occupied, we finish the iteration and increment of a small quantity the starting time.

Inside the for cycle we implement a *while* cycle wich stops when the current start time of the task exceeds the *cycle_time* provided from the outside. As we said, the objective of the inner cycle is to see if, with the current starting time, the task finds a clear path and fits the cycle time. So we consider every node of its route (by means of another *while* cycle) and access its timetable to see if it is occupied or not. In particular we store, inside of a variable that we call *delay*, the passing of time during the transmission of the packets, namely we take into account *propagation_time*, *computation_time* and transmission time that affect the motion between a node and the next one. The transmission time is computed by dividing the size of the packet by the transmsission rate; the transmission rate is in turn extracted starting from the *Adjacency_matrix* and *map* variables: the matrix is supposed to contain the transmission rates of every connection, while map expresses the mapping between the neighbor nodes and the port of a node . In this way we are able to know when exactly packets are going to transit to each node, and to check their timetable accordingly. We do not, however, modify the timetable because we first need to check the whole route. As a matter of fact, if we find that the timetable of at least one node is occupied when the packet is supposed to transit to that node, we restart the computations over the route, but assuming a start time incremented by *accuracy*. An important detail is that we initialize the value of the start time not to zero, but to the duration of a guardband: this is done because every TSN windows needs a guardband before, for the reasons widely explained, therefore in any case we would first need to wait for the guardband to finish. As a result, the earliest possible start time is equal to the duration of the complete transmission of a maximum-sized ethernet packet.

If we successfully terminated the first while cycle and we found a suitable starting time for our task, we first register the start time in the output variable *generation*; then we proceed, by means of another, and similar to the first one, *while* cycle, to consider each step of the route. This time, though, since we are sure that the nodes are not occupied, we modify the timetables and add the rows related to the current packet. At the end of this cycle we compare the *delay* variable, which at this point is the arrival time, with the maximum value of this variable found so far, in order to compute *span* and identify the *critical_flow*.

This concludes the series of operations performed in the big for cycle, and also concludes the function, as all the output variables have been computed. The binary output variable *success* is simply written with a one if the whole task sequence fits the cycle and with zero otherwise.

4.2.2 NWPS_Heuristic

This is the main function addressing the No-wait packet scheduling algorithm, namely the one function that has to be called in a generic program and that contains all the others, timetabling included.

The function has the following features:

$$\begin{aligned} &[\text{Schedules, generation, span, max_span, critical_flow, success}] = \\ &\text{NWPS_Heuristic}(\text{Routes, Adjacency_matrix, TaskSet, map, cycle_time,} \\ &\text{prop_time, comp_time, accuracy, margin, x, y, outputs}) \end{aligned} \tag{4.2}$$

The first thing that the function does is compute an initial solution, namely an ordered sequence containing all the tasks in *TaskSet*. It can be random or it can be based on some criteria; we chose to generate our initial solution according to the Ids of the tasks, so as to put as first the task with the lowest Id and as last the one with the highest Id. In this way we try to assign a sort of prioritizing role to the Ids.

Then we apply the timetabling algorithm (using 4.1) to the initial solution; the timetabling in turn gives the span and the critical flow of the initial solution. Since it is the only value known so far, we register the span as the best span, other than as the current solution.

Our search is based on neighborhoods. At each iteration we produce a neighborhood, namely a restricted set of solution, that depends on the current solution; then we search the whole neighborhood for its best solution and use that as the current solution. Every current solution is also compared with the best one in order to update the best one, if it needs to be; we stop when we reach a certain number of consecutive neighborhoods visited without improving the solution.

So we build a *while* cycle, and we put as the condition when the number of unsuccessful iterations, which is a local variable, is higher than the input *y*; by means of this input it is possible to tune the duration of the algorithm.

Inside the while cycle we create the neighborhood. Recall from section 2.4.2 that a neighborhood is generated by using two operations, swapping and insertion, on all the flows preceding the current critical flow in the current solution: swapping consists in just switching the places of the critical flow with the target flow, while insertion requires to put the critical flow before the target flow. In this way we produce $2n-2$ solutions, where *n* is the position of the critical flow in the current solution. So, by means of a *for* cycle, we analyze each component of the neighborhood with the timetabling function. Another feature of the search is that it is based on a tabu list: we keep track of the last *x* values of the critical flow in a vector that we call *tabu_list*, and we are not allowed to select a solution as the current solution if its critical flow figures in the tabu list. With the input *x* it is then

possible to tune the length of the tabu list, whose purpose is to continuously change the critical flow of the current solutions. The exception is when the solution has a better span with respect to the best one encountered so far: in this case, even if its critical flow belongs to the tabu list, it is eligible to be selected as the next solution.

Once we have our current solution, before repeating again the cycle for another neighborhood, we compare the current solution with the best one encountered so far: if it is better then we register the current solution as the best one, and we reset the counter for unsuccessful iterations; otherwise we increment the counter.

Eventually, when all the necessary iterations have been performed, the timetable (recall from the previous subsection that it is an output variable of the timetabling algorithm which reports the occupied windows for each port in the system) variable is used by another function to create a proper schedule. As we already highlighted, the simple sequence solution is not enough to drive the time aware shaper, and neither it is the timetable; we programmed a function specifically to build the schedule, and we are going to describe it in the next subsection.

Eventually, after the computation of the actual schedule, which is stored in the output variable *Schedules*, the function analyzes the input flag *outputs*: if it is one, some useful quantities are computed and then printed in the command window, to provide the user with useful information about the solution just computed; with *outputs* enabled, the function also prints periodically the number of solutions analyzed while it is visiting neighborhoods in the solutions' space.

There are still a couple of input and output variables that haven't been described. *max_span*, for instance, is the span of the worse solution encountered in the search. *margin*, instead, is a safety margin, expressed in bits, for the computation of the windows: it is added to the size of every task in the taskset in order to consider bigger windows and bigger transmission times than the ones actually required by the tasks. As the name suggests, it is a safety margin which allows to compensate for inaccuracies in the model of the network. The other input and output variables that have not been mentioned here are needed for the timetabling function.

4.2.3 Schedules

As we already mentioned, the solution of the timetabling algorithm is not sufficient to drive the time aware shaper. As we built it, our time aware shaper needs a schedule in the form of a matrix, with the same format shown in Figure 4.7. Therefore we created a function, named *create_schedule*, which deals with these computations. It needs to be called in the *NWPS_Heuristic* function in the following way:

```
[Schedules, TSN_time, idle_time, total_time] = create_schedule( (4.3)
TimeTable, Adjacency_matrix, map, cycle_time)
```

As the first thing, we extract from the input *TimeTable* the information about the TSN windows and add them to the output variable *Schedules*. *Schedules* is a matrix with four dimensions, where the first indicates the node's Id and the second its port. If we access this matrix with a node's Id and the number of one of its ports, we get a bidimensional matrix similar to the one of Figure 4.7. TSN windows are characterized by an offset, a duration and a specific combination of the gates. From the values of the *TimeTable* we have to compute the first two parameters and put the known TSN sequence in its intended position.

Then we have to place guardbands before every TSN window. By means of a *while* cycle we analyze every schedule built so far and place the guardbands: if there isn't room for a whole guardband it means that two TSN flows are scheduled so close that no best effort packet will ever be allowed to pass in between; therefore we just have to fill the space between the TSN windows with a window characterized by all the gates closed, but which is not technically a guardband.

Eventually we have to consider every hole left by the scheduling of TSN windows and guardbands, and fill them with best effort windows: again we have to compute, from the neighbor windows, duration and offset of the new windows, and also put the best effort's combination in its place.

These operations give us proper matrices that can be used to drive our time aware shapers. There is, however, another schedule that we need, which is not a standardized schedule but it is needed for the simulation: it is the generation schedule, which contains the information needed by the packets generators to know when and which packets to produce.

Outside the function *NWPS_Heuristic*, namely in the main program, the function *create_generation_schedule* has to be called, in the following way:

```
generation_schedule = create_generation_schedule( (4.4)
generation, TaskSet, number_of_hosts, cycle_time)
```

This function takes the *generation* variable, which has the start times of every flow in the best solution of the No-wait packet scheduling algorithm, along with their associated tasks' Ids. The objective is to create, for each node, a sequence of the tasks that it needs to generate with all the information about them: their Id, their destination, their size, their intergeneration time and so on and so forth. So we exploit the *TaskSet* input variable to access the data about the tasks and we create the output *generation_schedule* so that it can easily be read and understood by the packets generators in the simulation.

4.2.4 Multi-objective optimization

So far we have described the No-wait packet scheduling algorithm as it is presented in [14]. As we have seen, it is a heuristic solution which is inspired by iterative algorithms for classical optimization problems: we start from an initial solution and, based on that solution, we generate another solution to evaluate, and so on and so forth. In our case we do not have the gradient of the cost function to indicate us the search direction: instead, we generate a whole neighborhood based on the critical flow and search there. Our "direction" is the solution which minimizes the span in its neighborhood.

In this subsection we propose a variant of that algorithm, based on a multi-objective optimization problem. The main reason that led us to develop this alternative version is that minimizing the span isn't always indicative of a better solution. As a matter of fact, we can consider the final goal of a good scheduling algorithm, other than correctly placing the TSN tasks within the period, maximizing the overall throughput: we can achieve this by minimizing the guardbands in the schedules.

By trying to schedule tasks as soon as possible, also minimizing the span, the No-wait packet scheduling is already an excellent solution as it concentrates the tasks in a restricted section of the cycle; as a result, the number of guardbands will decrease since one guardband will be used by several TSN windows and not by just one. We can, however, directly consider the actual amount of idle time of the system as a cost, and use it to find the search direction of our algorithm.

We still kept the span as a cost, but added another cost, which is indeed the idle time of the system. Recall from 4.3 that our scheduling function gives as output three variables: *total_time*, *TSN_time* and *idle_time*. These three values represent the total time available in one cycle for all the ports, the time spent in TSN windows and the time in which the system has to remain idle due to guardbands. We used the *idle_time* variable as our second cost.

What we just created is a sort of multi-objective optimization problem, where we are trying to minimize at the same time both the costs. In general it does not exist a solution minimizing both. What does exist is a set of solutions, each one minimizing a single cost or differentiating by the others because neither one of the others has a better value for at least one of the costs without worsening the other. In classical optimization problems, this set is named *Pareto* set. Ours is not a classical optimization problem, therefore it wouldn't be correct to use the term Pareto set, but we were in fact inspired by this concept in the development of our algorithm.

In order to solve the presented problem, we used a scalar method, namely we turned our two costs in a scalar variable which can be used to make comparisons. As a matter of fact we defined our global cost as:

$$cost = \sqrt{span^2 + idle_time^2} \quad (4.5)$$

which is the euclidean distance from the origin in the plane with `span` and `idle_time` as dimensions.

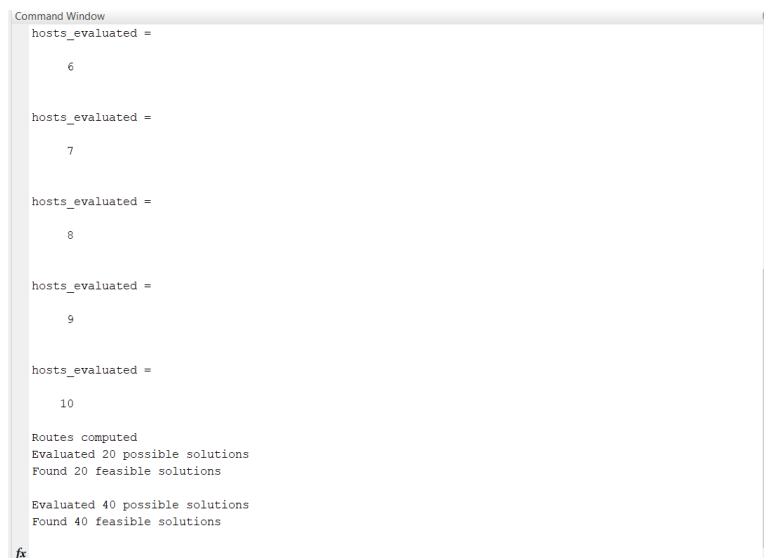
So we created another function named *NWPS_Heuristic_mo*, very similar to the one described in section 4.2.2, but with the cost of equation 4.5 instead of the mere `span`. In this way our search direction is the one of the solution that minimizes the new cost, which takes into account both the `span` and the actual idle time.

From an implementative point of view, the code is in this case heavier, because we need to compute the actual schedules for every analyzed solution, while in the previous case we computed the schedules only for the final solution. Afterwards we use the two costs to compute the global cost, and then we proceed exactly as we used to do in *NWPS_Heuristic*.

4.2.5 Performance

As the final note about the implementation of the No-wait packet scheduling algorithm, we provide some information about its performances, in terms of execution times. The considerations about its effectiveness will be presented in the next section, when the produced schedules will actually be tested using the simulation environment; also, comparisons with the Joint Routing and Scheduling problem and our custom version will be provided, highlighting its advantages and drawbacks.

First of all we show what the function prints on the command window of MATLAB if we enable it to do it.



```
Command Window
hosts_evaluated =
    6

hosts_evaluated =
    7

hosts_evaluated =
    8

hosts_evaluated =
    9

hosts_evaluated =
   10

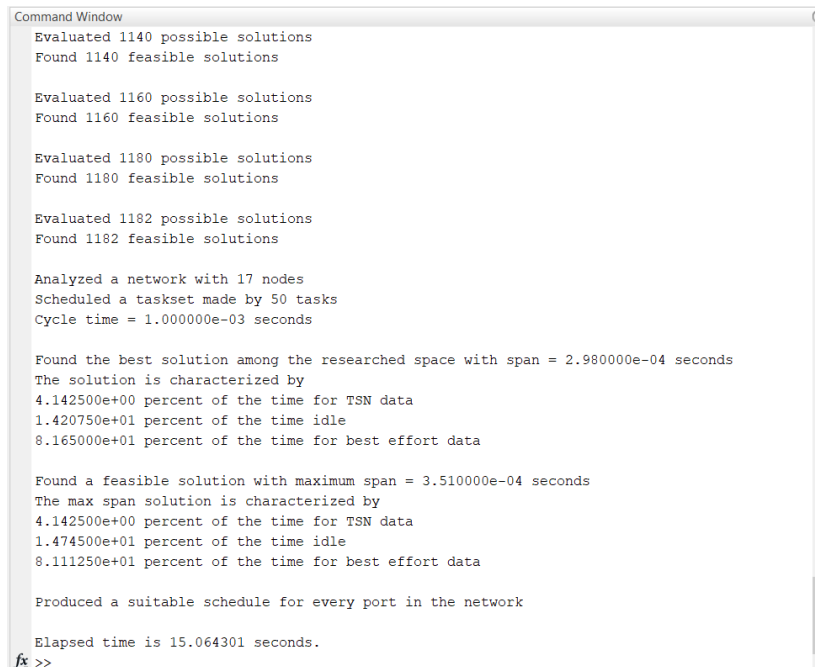
Routes computed
Evaluated 20 possible solutions
Found 20 feasible solutions

Evaluated 40 possible solutions
Found 40 feasible solutions
```

Figure 4.20: Outputs of the function: 1

The computations about the routing need to be performed before the scheduling: we can see that the routing function also prints its progression, as the number of hosts for which all the possible routes have been computed. For the simple network in the example the routing execution is almost instantaneous.

Then the scheduling function is called. If the outputs flag is activated it periodically shows the number of solutions that have been evaluated up to the moment, namely the number of times that the timetabling algorithm has been applied. In addition it shows the number of feasible solutions found, i.e. the number of sequences whose span is lower than the cycle time. This parameter is also useful to immediately see if the problem is easy to solve or not.



```
Command Window
Evaluated 1140 possible solutions
Found 1140 feasible solutions

Evaluated 1160 possible solutions
Found 1160 feasible solutions

Evaluated 1180 possible solutions
Found 1180 feasible solutions

Evaluated 1182 possible solutions
Found 1182 feasible solutions

Analyzed a network with 17 nodes
Scheduled a taskset made by 50 tasks
Cycle time = 1.000000e-03 seconds

Found the best solution among the researched space with span = 2.980000e-04 seconds
The solution is characterized by
4.142500e+00 percent of the time for TSN data
1.420750e+01 percent of the time idle
8.165000e+01 percent of the time for best effort data

Found a feasible solution with maximum span = 3.510000e-04 seconds
The max span solution is characterized by
4.142500e+00 percent of the time for TSN data
1.474500e+01 percent of the time idle
8.111250e+01 percent of the time for best effort data

Produced a suitable schedule for every port in the network

Elapsed time is 15.064301 seconds.
fx >>
```

Figure 4.21: Outputs of the function: 2

Figure 4.21 shows the command window after some time with respect to Figure 4.20. We can see that the data concerning the numbers of evaluated and feasible solutions keep being shown throughout the whole execution; then, when the algorithm terminates, a summary of the computations just performed and of the obtained results is provided to the user. We can see the dimension of the network and of the taskset, the cycle time and the final number of solutions analyzed. The the best solution found is presented to us, along with its span and the percentages of idle time, time for TSN packets and best effort packets.

In addition, the worst solution's details are presented as well, in order to

appreciate the enhancements provided by the long search; clearly it is not even close to the worst possible solution since, as we said, it is the worst solution encountered while searching for the best one.

Eventually we can see that a suitable schedule has been crafted for all the ports in the network, and we can read the execution time of the whole algorithm which, in the example, was of 15 seconds.

Next we show the output of our alternative function named *NWPS_Heuristic_mo*.

```

Command Window
Evaluated 1960 possible solutions
Found 1960 feasible solutions

Evaluated 1980 possible solutions
Found 1980 feasible solutions

Evaluated 1996 possible solutions
Found 1996 feasible solutions

Analyzed a network with 4 nodes
Scheduled a taskset made by 50 tasks
Cycle time = 1.000000e-03 seconds

The research is multiobjective, and it is aimed at minimizing both the span and the idle time
The cost function is the cartesian distance from the origin computed with the two values
Found the best solution among the researched space with cost = 3.305893e-04
The solution is characterized by
Span: 3.250000e-04 seconds
1.460000e+01 percent of the time for TSN data
1.513333e+01 percent of the time idle
7.026667e+01 percent of the time for best effort data

Found a solution with the best span: 3.250000e-04 seconds
But with idle time: 1.520000e+01 percent

Found a solution with the best idle time: 1.510000e+01 percent
But with span: 3.260000e-04 seconds

Produced a suitable schedule for every port in the network

Elapsed time is 29.727694 seconds.
fx >>

```

Figure 4.22: Outputs of *NWPS_Heuristic_mo*

At the end of the execution of this function we are again shown the details of the problem that we are trying to solve. Besides the optimal solution, though, the output presents two other solutions. As a matter of fact, the first solution is the one minimizing the global cost described by equation 4.5, while the second minimizes only the span and the third only the idle time. Therefore we can consider the first as a tradeoff, while the others are aim at minimizing just one cost at the expense of the other.

We conclude this section with an analysis of the average execution time of the implemented instance of the No-wait packet scheduling. We have seen that several cycles are required, usually nested one within the other. Even though the number of lines does not exceed four or five hundreds, the structure of the code turns out to be quite heavy, since the number of iterations increases very rapidly. This behavior is immediately detectable for non-trivial scheduling problems, where it takes some time to the algorithm

to finish. At the end of the day, among all the data concerning the optimal solution and network utilization, the execution time is a parameter to rate the validity of a scheduling algorithm as well. One positive feature of this implementation is that its memory consumption is limited and does not depend on the execution time: by construction, the necessary amount of memory at the start of the operations gets continuously updated without the need to ask the system for more during the execution. Therefore the algorithm could run virtually for a very long time, even in systems with small amounts of RAM available.

So we have made some tests, measuring the execution time of the *NW-PS_Heuristic* via a simple MATLAB command. The tests were carried out using the 2018a version of MATLAB, running on a PC with 16 gigabytes of RAM and an 8th generation i7 Intel processor.

As we have seen, there are several parameters that can change from within the behavior of the algorithm, and thus affect its execution times: the accuracy, namely the offset between one test start time and the next in the timetabling algorithm, the number of unsuccessful iterations, the dimension of the tabu list, of the network and of the taskset. Among all these parameters, probably the most meaningful dependency to test is the one of the execution time with respect to the number of tasks; therefore we fixed the other conditions and we run the algorithm several times, progressively increasing the number of tasks. In particular we considered a network with 10 hosts and 7 switches, 10 maximum unsuccessful iterations, 2 values maximum in the tabu list and an accuracy of 1 μ s. Then, by means of a simple script, we generated a suitable set of tasks by randomly choosing source, destination and size of every flow; the series of tasksets used in the tests was obtained by progressively adding some elements to the already existing taskset.

Figure 4.23 shows the evolution of the execution time and of the number of solutions considered, as functions of the number of tasks.

As we can see, the execution time increases faster than the number of tasks, with an exponential shape. This is a reasonable behavior, especially if we consider how the algorithm works. For the same network, if we increase the number of task, we need to allocate more windows to the transmission of TSN data, therefore the last tasks to be placed will have to start later and later, increasing the number of iterations needed to find the appropriate start time. As a matter of fact, the $\frac{time}{evaluated\ solutions}$ ratio constantly increases with the number of tasks. The specific duration, both in terms of execution time and of solutions, depends also on the initial solution considered: if, for instance, it has already a good span, which is hardly improvable, the execution will stop sooner with respect to a "bad" first solution.

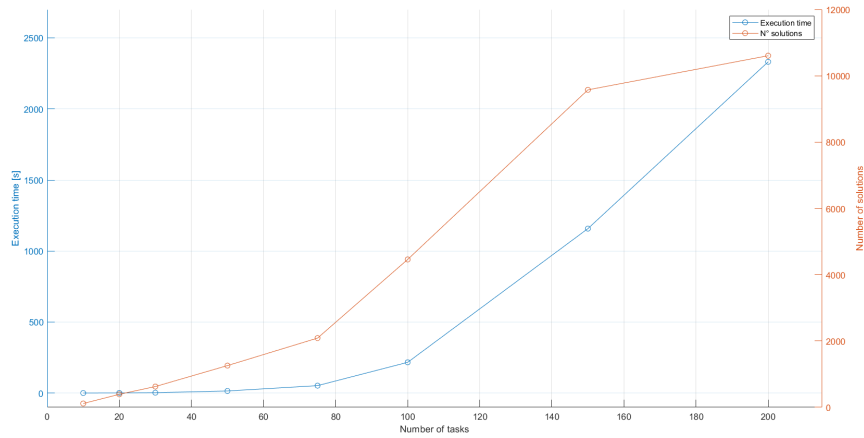


Figure 4.23: Latencies for TSN packets

4.3 Test simulations

Finally in this concluding section of the chapter we exploit the topics of both the previous sections to perform simulations of time sensitive networks, whose schedules are computed by means of the No-wait packet scheduling algorithm.

We are going to show the results of two simulations, carried out on two different networks: the first network is very simple, and its purpose is to provide an example to better understand how the scheduling algorithm works and to directly see the consequences of the No-wait policy; the second network, instead, is more complex and representative of any possible network compliant with the TSN standards.

Eventually we are going to end the chapter with a subsection dedicated to the conclusions about the simulations and, in particular, about the scheduling algorithm's features.

4.3.1 Simple network

As we said, this simulation is quite trivial as it aims to show the properties of the No-wait packet scheduling algorithm. We consider a simple network made by three hosts and one switch, with a star connection. The hosts have Ids from 1 to 3, while the switch as Id 4. Next we show the implementation on Simulink of such a network.

Since the topology is simple, it is still possible to unsterstand the connections from Simulink.

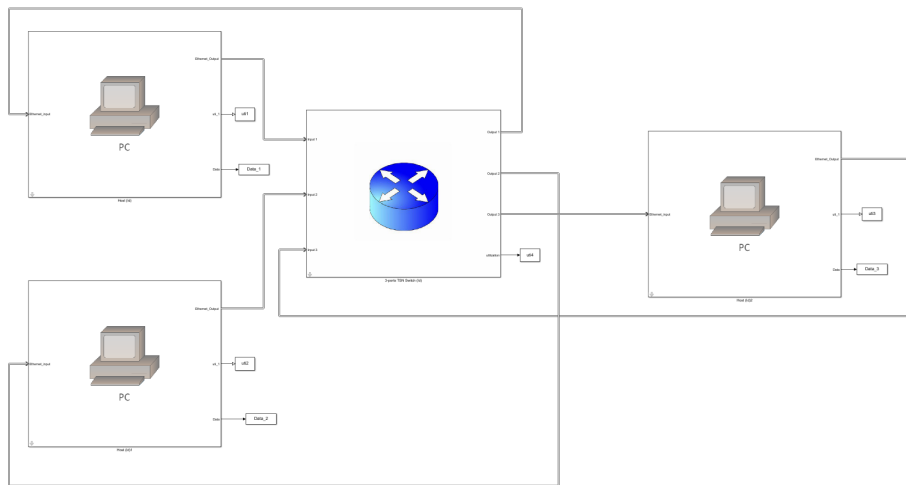


Figure 4.24: Simple network

The taskset was very simple as well, since it had only five tasks; we used Excel to write the taskset in a more comprehensible way and then imported the data on MATLAB.

	A	B	C	D	E
1	Task_Name	Task_Id	Task_Source	Task_Destination	Task_Size (bits)
2	Task_1	1	1	3	100
3	Task_2	2	1	3	200
4	Task_3	3	2	3	300
5	Task_4	4	2	3	400
6	Task_5	5	2	3	500

Figure 4.25: Taskset

For the purposes of the test, we have assigned to the tasks different sizes in order for it to be easier to tell them apart in the schedules. Furthermore they are all directed to host number 3 and generated in hosts 1 or 2. In this way we create conflicts on the port of the switch connected to host 3, since it has to receive and redirect the incoming packets from host 1 and 2.

So we launched the configuration script on MATLAB, which in turn called the routing and scheduling functions; at the end we had all the parameters needed by Simulink stored in the workspace, and we were able to correctly start the simulation. Once the simulation had finished, we launched another script which analyzed the results of the simulation, namely the big data matrix such as the one in Figure 4.9. These were the results of the analysis, printed on the command window of MATLAB.


```
>> Results
Evaluation of the results of the simulation...

Simulation time: 1.000000e-02 seconds
Cycle time for TSN packets: 1.000000e-03 seconds
Number of cycles simulated: 10
Number of TSN packets per cycle: 5

Received a grand total of 2933 packets, of which:
50 TSN packets
2883 BE packets

Number of TSN packets that missed their deadline: 0

Average queueing time for TSN packets: 8.180000e-07 seconds
Average queueing time for BE packets: 3.420747e-03 seconds

The graph shows a comparison of the waiting times
for BE and TSN per each cycle

Actual utilization rate of the network: 8.423990e+01 percent
fx >>
```

Figure 4.26: Results of the simulation

The results of the simulation initially summarize the characteristics of the problem such as simulated time, cycle time and number of TSN packets. Then we can read the total number of received packets, in this case by host 3: correctly, the TSN packets are the exact product between the number of cycles and the design number of packets per cycle, while the number of best effort packets does not have a specific meaning. Afterwards the counter of TSN packets that missed their deadline is shown: this counter is increased whenever a packet is received in a different millisecond with respect to the millisecond when it was generated and, in this case, we can appreciate that all the packets were delivered on time. The average waiting time is also meaningful, in particular since the scheduling algorithm is based on it: as a matter of fact, TSN packets are constrained to wait for fractions of microsecond, which is practically nothing, while best effort packets have to wait, on average, more than three complete cycles during their path.

Even though these results are expected and make sense, the actual focus of this simulation is on the schedule. As we can gather from the taskset and from the network's topology, this scheduling problem is easily solvable by hand, but we used our algorithm to highlight some of its features. In order to support our considerations, in the next picture we show the computed schedule for the port of the switch connected with host 3 and the generation variable, which reports the best sequence of task found along with their, generation time.

```

Command Window
>> reshape(Schedules(4, 3, :, :), [16 6])

ans =

    0    0    0    1.0000000000000000    0    0.0000070000000000
    0    0    0    0    0    0.0000070000000000    0.0001200000000000
  1.0000000000000000    0    0    0    0    0.0001270000000000    0.0000010000000000
    0    0    0    0    0    0.0001280000000000    0.0000010000000000
  1.0000000000000000    0    0    0    0    0.0001290000000000    0.0000030000000000
    0    0    0    0    0    0.0001320000000000    0.0000010000000000
  1.0000000000000000    0    0    0    0    0.0001330000000000    0.0000020000000000
  1.0000000000000000    0    0    0    0    0.0001350000000000    0.0000050000000000
    0    0    0    0    0    0.0001400000000000    0.0000010000000000
  1.0000000000000000    0    0    0    0    0.0001410000000000    0.0000040000000000
    0    0    0    1.0000000000000000    0.0001450000000000    0.0008550000000000
    0    0    0    0    0    0.0010000000000000    0
    0    0    0    0    0    0
    0    0    0    0    0    0
    0    0    0    0    0    0
    0    0    0    0    0    0

>> generation

generation =

  3.0000000000000000    0.0001200000000000
  2.0000000000000000    0.0001250000000000
  5.0000000000000000    0.0001240000000000
  4.0000000000000000    0.0001310000000000
  1.0000000000000000    0.0001200000000000

```

Figure 4.27: Schedule and generation

Starting from the schedule, we can see that the first window is dedicated to best effort traffic (refer to subsection 4.1.6 for the format of the schedule); this happens because TSN packets have to be first generated in their sources and travel to this port, so there is room to let some best effort packets flow. After $7 \mu\text{s}$ from the start of the cycle, a guardband is placed and, subsequently, the series of TSN windows. The duration of the guardband of $120 \mu\text{s}$ is obtained by multiplying 1500 bytes times 8 bits and dividing by the rate of 100 millions of bits per second, namely the time it takes to transmit the largest ethernet frame. A part from one case, due to the needs of the schedule, the TSN windows are not completely attached, but they are still close. So close that in between there is no space to put a whole guardband and another best effort window; therefore we just have to put a sort of shorter guardband to fill the gap. This explains why the throughput of best effort packets is intrinsically augmented with this procedure.

From the durations of the TSN windows we can figure out which flow they represent: the first one to transit is flow number 1, then numbers 3, 2, 5 and 4. Eventually we have one last best effort window which ends at the end of the cycle; due to the particular structure of this schedule, it resumes right at the beginning of the new cycle.

By taking into account the computation and propagation delays that we provided to the algorithm, along with the specific transmission times of packets, it is possible to verify that each one of them is scheduled so as it starts as soon as possible and it does not have to wait. In particular we considered $5 \mu\text{s}$ of computation time and $1 \mu\text{s}$ of propagation time. These features are supported by the results of the simulation.

Let's consider now the generation variable: on the first column we have the ordered sequence which has been found to have the best span by the algorithm. It is therefore the order followed placing the tasks on the system. The second column reports the generation times associated to each task.

We can see that the earliest generation times are of $120 \mu\text{s}$, which is the duration of a guardband: this means that we want to produce flows 3 and 5 as soon as possible but, since we have to place guardbands before any TSN window, we have to wait $120 \mu\text{s}$ from the start of every cycle. According to the taskset, their sources are different, hence it is physically possible to send them at the same time.

We can also notice that we do not produce flows as soon as the sources could: if this were the case, we would have that host 1 produces flow 3 at $120 \mu\text{s}$, flow 5 at $123 \mu\text{s}$ and flow 4 at $128 \mu\text{s}$, while host 2 would produce flow 2 at $120 \mu\text{s}$ and flow 1 at $122 \mu\text{s}$. This hypothetical schedule can be computed by hand just by looking at the taskset and the optimal sequence of tasks. However, since we have to meet the No-wait policy, we need to postpone the production of some packets because, otherwise, they would be constrained to wait, specifically, in the switch. So, for instance, in host 2, flow number 4 is not produced at $128 \mu\text{s}$ but instead it is produced at $131 \mu\text{s}$.

Another interesting feature of this schedule is that flow number 1 is actually the first to reach the destination, even though it is the last one to be placed. We can see that it is the last value on the sequence in the generation variable, but it is supposed to be produced at the beginning of the cycle in host 1 and, also, the first TSN window in the host has the same duration as flow 1. This happens because the first flow to be placed, namely number 3, is so big that a small flow such as the number 1 fits in the schedule before flow 3. As a matter of fact, if we take into account propagation, transmission and computation delays, flow 3 arrives at the switch at $129 \mu\text{s}$ ($120 \mu\text{s}$ plus $3 \mu\text{s}$ of transmission delay plus $1 \mu\text{s}$ of propagation and $5 \mu\text{s}$ of computation), while flow 1, by $128 \mu\text{s}$, has already finished using the switch. Since the other tasks are not small enough, they do not fit and so they have to pass after flow 3; instead, flow 1 does not get in the way of any of the previously placed task, therefore it actually gets to pass as first.

This simple example is useful to understand how the No-wait packet scheduling works and, in particular, how the final solution's sequence does not imply that that's the actual ordering of the packets. Clearly, with a generic network and a more complex taskset, these kinds of patterns, while still guiding the structure of any schedule, become almost impossible to highlight in a comprehensible way; for this reason we have adopted this simple and nice test scenario.

4.3.2 Generic network

The second simulation that we present in this section concerns a larger network with a more extended set of tasks, and it is supposed to simulate all the configuration steps in a generic scenario.

The network that we consider for the test has actually been provided as an example of generic network in Figure 4.10; however, we are going to show its principle scheme, drawn by means of Excel, so that it is more comprehensible than the actual Simulink file.

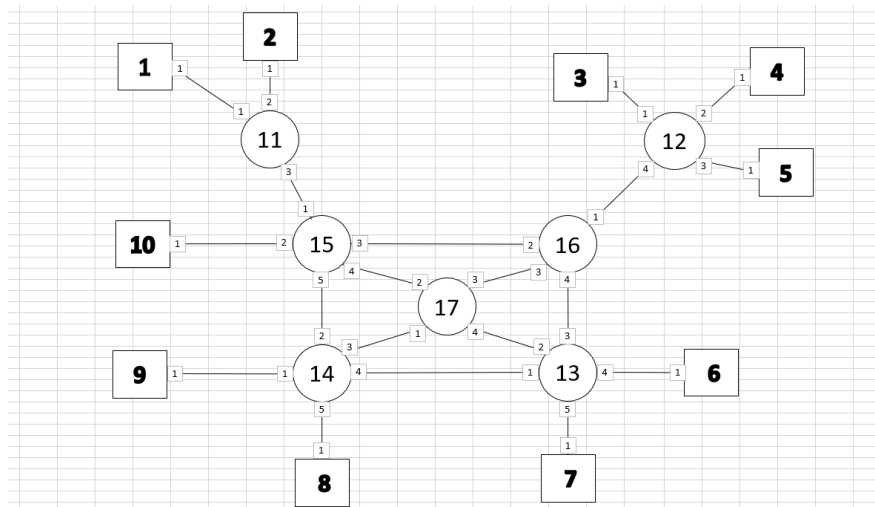


Figure 4.28: Generic network

We indicated hosts with squares and switches with circles. The number inside every node is its Id and every line represents a bidirectional connection; at the ending points of the connection the port number of the node is written.

The taskset is made by 100 TSN flows, randomly generated; we used the same script described in the last section which computes random sources, destination and sizes for every task. Then we saved it in another script in order for it to be available for any simulation.

As usual, the first thing to do is launch the configuration script, which computes the necessary routes from all the hosts to all the hosts and then computes the schedules for the specific taskset. In this case, after 170 seconds and 4004 possible solutions evaluated, the algorithm found a solution with span 0.356 milliseconds and 10.3% of time idle; the worst solution encountered had a span of 0.377 milliseconds and the 10.6% of the time idle.

Then we simulated the network for 0.01 seconds and, eventually, we evaluated the results.

```

>> Results
Evaluation of the results of the simulation...

Simulation time: 1.000000e-02 seconds
Cycle time for TSN packets: 1.000000e-03 seconds
Number of cycles simulated: 10
Number of TSN packets per cycle: 100

Received a grand total of 3354 packets, of which:
1000 TSN packets
2354 BE packets

Number of TSN packets that missed their deadline: 0

Average queueing time for TSN packets: 4.566400e-06 seconds
Average queueing time for BE packets: 3.147973e-03 seconds

The graph shows a comparison of the waiting times
for BE and TSN per each cycle

Actual utilization rate of the network: 7.371917e+01 percent
fx >>

```

Figure 4.29: Results about the generic network

The graph it is referring to in the second-last sentence is the following.

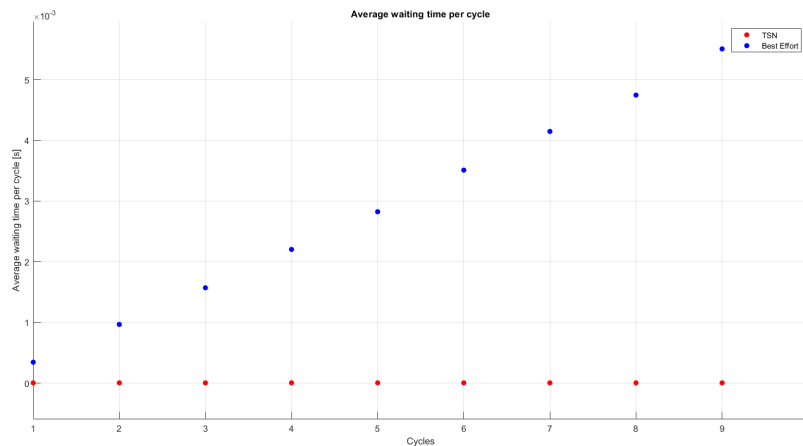


Figure 4.30: Average waiting times

As we expected, the 10 cycles in 0.01 seconds produced a total of 1000 packets, all of them delivered within the millisecond in which they were sent. The number of best effort packets is more than twice as the number of TSN packets, and that's reasonable since the TSN packets all finish before the $\frac{2}{5}$ of the period. Furthermore we can notice that the No-wait property is respected as the average waiting time for TSN packets is of 4 μ s, while for best effort packets it is a thousand times higher. Not only that: as a matter of fact, we can see from the graph in Figure 4.30 that the average waiting time per cycle of best effort data linearly increases with the number of cycles, while the average waiting time per cycle for TSN data remains

constant. This means that the best effort network is congested and does not manage to satisfy all the requests it receives in a period, without, though, affecting the behavior of the TSN part, which delivers its flows accordingly to the schedule.

The actual waiting time for TSN packets is not zero because of the safety margins of the model; namely we considered worst case scenario values for the transmission time and propagation time of packets while crafting the schedule, whereas in the simulation we used the "real" values. In any case, 4 μ s is a very low value, especially if compared with the duration of the cycle which is 1000 μ s.

We also report the results of a simulation performed with a taskset of 200 elements, namely 20 tasks per host, on average.

```
>> Results
Evaluation of the results of the simulation...

Simulation time: 5.000000e-03 seconds
Cycle time for TSN packets: 1.000000e-03 seconds
Number of cycles simulated: 5
Number of TSN packets per cycle: 200

Received a grand total of 2039 packets, of which:
1000 TSN packets
1039 BE packets

Number of TSN packets that missed their deadline: 0

Average queueing time for TSN packets: 4.751100e-06 seconds
Average queueing time for BE packets: 1.625065e-03 seconds

The graph shows a comparison of the waiting times
for BE and TSN per each cycle

Actual utilization rate of the network: 6.844285e+01 percent
fx >>
```

Figure 4.31: Results about the generic network with 200 tasks

As the picture summarizes, we simulated five whole cycles and correctly delivered 200 TSN flows, none of which arrived in a different period with respect to the one where it was sent. We can appreciate how the space for best effort data is reduced in comparison with the previous case, since we have doubled the TSN flows. The details of the schedule's computation are the following: the algorithm terminated after 1569 seconds (almost half an hour), having evaluated 7044 possible solutions, all of which were feasible; the best solution had 0.545 milliseconds as span, while the worst had 0.608 milliseconds. In this case we can also see that the lowest time span does not always lead to the lowest waste in terms of bandwidth, because the best solution had the 16.42% of time idle, while the worst had 16.35%. The behavior of the waiting times was the same as in the previous case, reported in Fig. 4.30.

We performed the same simulation with different tasksets and every time we got the same results; we can therefore conclude that the No-wait packet

scheduling algorithm could be a valid strategy to be employed in the configuration step of a time sensitive network.

4.3.3 Final considerations

In section 2.4 we have introduced the scheduling problem for a time sensitive network and we presented two different ways to achieve a suitable schedule: the Joint Routing and Scheduling problem and, indeed, the No-wait packet scheduling. JRaS is an optimization problem which expresses the physical constraints of the network and of the application as mathematical equations and which is solved by means of ILP methods; NWPS, instead, is a heuristic method which tries to schedule packets as soon as possible and so as they are never constrained to wait. In this chapter we implemented and tested the latter; we conclude with some considerations about its features, with particular reference to the JRaS.

The first consideration concerns the heuristic nature of the No-wait packet scheduling algorithm, and in particular the limit of unsuccessful iterations after which it stops. We have a solution space that has a number of elements of $n!$, where n stands for the number of tasks: as a matter of fact, all the possible inputs for the timetabling function are all the permutations of the n tasks in the taskset. It is possible that some solutions, if they present "independent" tasks, will lead to the exact same schedules, therefore the number of actual different solutions could be a little smaller; but from a practical point of view there is no way of knowing them a priori, hence we can consider the solution space as made of $n!$ solutions. As a consequence, if we actually wanted to find the absolute best solution there exists, we would have to perform the timetabling algorithm $n!$ times. This would mean:

- 120 times for 5 tasks
- over 3 million times for 10 tasks
- over 2 billion billion times for 20 tasks

and so on and so forth. The behavior of the factorial sequence is very sudden and we can quickly find ourselves to handle a solution space which is too large even for the most powerful computer in the world in an acceptable time. Consider also that the timetabling itself requires more time to be executed if the number of input tasks is increased, as we have seen. That's why scheduling is classified as a NP-hard problem. Therefore we set a criterion to stop the search before having completed the whole solution space, and then settling for a solution which is not guaranteed to be the best.

On the other hand, even if the cost is different, the JRaS problem gives the best solution, which makes its execution time way heavier with its corresponding heuristic version. In fact, in [13], all the tests are carried out on

a powerful computing system with 128 gigabytes of RAM and with tasksets made by 25 tasks top; they also explain how, in order to obtain a solution in acceptable time, the number of tasks in the system shouldn't exceed 50/60. That's one drawback of optimal solutions and, in contrast, one advantage of heuristic methods: heuristic methods allow to obtain a solution, even if not the best, in a reasonable time with respect to the dimensions of the problem. We were in fact able, according to Figure 4.23, to solve the problem with a considerable number of tasks in an acceptable time: we did not explore the whole solution space, as the number of evaluated solutions attests, but managed to find a suitable schedule for up to 200 tasks. In the worst case it took us a couple of hours to execute the MATLAB's script, which is still acceptable, especially if we consider that this is a procedure which has to be performed just once, before starting the operations. Furthermore, 200 tasks is a reasonable maximum number of flows to be scheduled in a real environment; recall that the dimensions of the network, for the time being, are constrained not to exceed a maximum path's length between nodes of about 7/8 hops, due to the difficulty of keeping a stable and precise time synchronization between the devices. Also, since the services provided by the time aware shaper aren't free, there is a limit on the number of windows per cycle that a device can correctly handle; therefore, since both the number of nodes and the packets per node have hardware limits, we can safely assume that the total number of tasks to be scheduled in a real-life scenario does not exceed a few hundreds of flows.

The second consideration is that the schedule produced by the No-wait packet scheduling has some fault tolerant features, as we already mentioned in subsection 2.4.2. Since the resulting flows are scheduled to finish as soon as possible, there is still time in the cycle to compensate for possible inaccuracies in the model of the network and/or in the assumptions about the maximum values of delays and latencies. Clearly this is not an ideal situation, but the concept that we want to highlight is that, by reducing both the finishing time and the time packets spend around in the network, the amount of risks, for what concerns packet loss or delay, is reduced. The JRaS problem, for instance, does not include any constraint promoting early departures of packets. Thus there could exist a scenario where in the optimal solution the TSN flows are all scheduled at the end of the cycle: while this may work on paper, in practice it is always better to consider safety margins that are as large as possible, because if one unpredicted delay were to take place, it would be easier to miss some deadlines.

To this fault tolerant feature we have to add that the No-wait packet scheduling algorithm is more flexible as well, as it allows to add flows to the taskset, after an initial computation, more efficiently. Minimizing the span is not only useful to compress the flows and reduce the guardbands, but also to have more time available to let best effort packets flow or, even, schedule other TSN tasks. If, for instance, we needed to add a couple of flows

to an already existing and scheduled taskset, we would just have to execute the NWPS algorithm for the two additional flows, taking into account the already occupied time windows in the network. So, basically, we would be appending the new tasks at the end of the current best sequence of tasks for the timetabling, without modifying the first part; as we have seen, this wouldn't even imply that they would be going to be scheduled as last because, if they fit, they would be placed as soon as possible. Hence, the NWPS algorithm gives us the degree of freedom of being able to choose whether or not to recompute the schedule for the whole taskset: if we did, we would definitely have a larger solution space to explore, but at the cost of an increased time complexity; otherwise we could apply the algorithm just to the new packets and obtain a solution within a few seconds. With the JRaS, we do not have this possibility as, in any case, we would need to recompute all the schedules.

One possible drawback of the NWPS with respect to the JRaS is that we need to assume that all the TSN flows have the same cycle time. In a practical usecase it could happen that different time sensitive flows have different frequencies, and we wouldn't be able to handle similar tasksets very well. We would need to consider one big cycle, with duration equal to the least common multiple of the individual flows, and put in the taskset as many packets per flow as the number of periods in the big cycle. Then we could add some conditions within the timetabling so as to recognize if a solution meets the additional constraints, but we wouldn't in fact be able to search in the "right" direction. As a matter of fact, we base our next solution on the assumption that, if the current one isn't feasible, by minimizing the span there will be a higher probability that the next solution is feasible; that's not the case for such a kind of problems, where minimizing the span does not always imply finding a feasible solution. For instance, the minimum span solution could schedule all the packets belonging to one flow at the end of the cycle, therefore meeting only the last deadline. That's what we meant with right research direction: by just minimizing the span, we would be blindly exploring the solution space, recognizing the feasible solutions that we encounter but not really looking for them. On the other hand, the JRaS problem handles this scenario in a better way as it is possible to specify, for each flow, frequency and cycle time; the solution will then automatically consider them as constraints to be satisfied in an optimal way. However, if we consider our particular field, namely the industrial automation framework, it is not so common to have tasks with different cycles: an automatic machine is in fact driven by a Programmable Logic Controller which imposes the cycle time to all the elements of its network, that have to send their data in fixed windows. So, granted that this is a weakness of the algorithm, it is not that problematic in our line of work.

In the end, the last consideration is the answer to the following question: is it really necessary to obtain the best possible schedule? The answer, in our opinion, is no. There are several reasons why it is preferable, from a

practical point of view, to use the No-wait packet scheduling algorithm for the scheduling problem of a time sensitive network.

As we have seen, the best feature of Time Sensitive Networks is the possibility of integrating both the best effort traffic and the time sensitive traffic in one single network, with the side goal of connecting all the devices that, before, had to communicate using different means. Therefore it is important that both the data types flow in the system. If, for instance, we had a taskset that hardly fits in the cycle, for which would be then required to have the optimal schedule, the best effort packets would never get to flow in our network. As a consequence, there wouldn't be the need for a time sensitive network. We can thus safely assume that all the tasksets that we are going to have to schedule will be widely feasible. This is also reasonable if we consider that in an industrial plant what matters is the determinism, rather than the throughput and the actual amount of data. The problem is therefore not much finding a feasible solution as finding a solution which is "good" and does not require the computational effort to compute the best.

The No-wait packet scheduling is an excellent method in this sense, as its guiding principles are intrinsically "smart" and efficient. As we have seen, scheduling tasks as soon as possible and preventing them any kind of waiting provides to the scheduling not only with an automatic compression of the tasks, but also with some fault tolerant and flexible features. Furthermore, the tuneable search of the algorithm allows to reduce, on average, of another 5-15% the span and 3-4% the idle time of the system.

One final consideration that applies to every scheduling algorithm is that their effectiveness depends also on the accuracy of the model of the system that they exploit. If the behavior of the system is different from what is supposed to happen on paper, then some unwanted phenomena could take place, with the risk of interfering with the efficiency of the schedule. For instance, if the values for the transmission times or the propagation times for packets in the network are too high or too low, packets can arrive at the same time at the same node (which is forbidden by the algorithm) and even get swapped, without the system knowing; this loss of determinism could lead to delays and deadlines misses. We did actually experience this behavior, in particular concerning the safety margins that we added both to the transmission times and to the sizes of packets. If we chose values too different from the ones used in the simulation, non-determinism occurred. This in order to highlight the importance of a proper scheduling method and also of actual tests, even in a simulated environment.

In conclusion we can state that there exists a tradeoff of performances versus efficiency between No-wait packet scheduling and Joint Routing and Scheduling problem, but it is very biased in favor of the NWPS; so much that in the major part of cases it would be the better choice for the scheduling problem of an actual time sensitive network.

Chapter 5

Conclusions

The initial objective of this thesis project was to study the new topic of Time Sensitive Networking with the perspective of its future implementation in an industrial environment, in particular in the industrial automation field.

The advantages of a possible employment of a time sensitive network are that every type of data, namely the ones that are subjected to temporal constraints and the ones that are not, can flow in the same, big, corporate network.

The currently most widespread solution for this kind of problems is to use different networks to satisfy the needs of each type of data. For time sensitive data, such as the ones exchanged between the sensors and the PLC of an automatic machine, fieldbuses are employed: fieldbuses are special kinds of networks whose purpose is to reduce the efficiency and the actual amount of data circulating in the network in order to increase the determinism and the predictability of the working parameters. Instead, for the data that are not subject to any particular constraint, which are called best effort, a common switched network is used. We have seen in the first part of the second chapter how, in common switched network, it is virtually impossible to impose or even to predict a certain behavior of the data flows, as they depend on a high number of factors and events which are, most of the times, random.

Time Sensitive Networks are a series of standards which describe some mechanisms that allow a common switched network be able to support and meet the time constraints of a certain data flows, while letting the other circulate in the usual way. The concept and the potential of this technology are huge, as they are also compliant with the new paradigms of Industry 4.0 and Industrial Internet of Things.

In the first section of this final chapter we are going to briefly summarize the main points of this thesis project and the results obtained, both experimentally and in simulation. The second section is instead focused on the future developments, in particular the goals that are still far away from being reached and the topics that, even now, can be successfully explored.

5.1 Summary

As the title reports, the work has been divided in four parts. The analysis part is about the actual study of the material available on the internet, describing the main features and characteristics of Time Sensitive Networks. Since the final objective is the actual implementation of a time sensitive network, these sought features pertain also the implementation details, both on the hardware and software sides. As a matter of fact, the second part of this work was dedicated to the practical testing, on a special setup, of the main mechanisms defining TSN. The third and fourth parts, instead, are dedicated to the implementation and to the simulation, with validation purposes, of a suitable and efficient scheduling algorithm for a time sensitive network, which is needed to organize and configure the network in order for it to work properly.

5.1.1 Analysis

By definition, Time Sensitive Networks refers to a set of communication standards based on ethernet, which in turn describe a series of mechanisms that are supposed to be implemented in a switched network. These standards are still under direct control of the IEEE association, which has developed a part of them and aims at their completion in the next future.

We can divide the topics addressed in the standards in three areas: traffic shaping, time synchronization and system configuration.

In the traffic shaping area we can find the most important mechanism of TSN, which is the Time Aware Shaper. The TAS is a particular output interface that allows to drive, by means of an appropriate list of commands, a series of output queues which are filled with packets belonging to different traffic classes. The main idea is to build a system of gates that open and close to create "protected windows", namely intervals of time where only a specific traffic class gets to be transmitted. Every output port of every node in the system is supposed to have a time aware shaper which controls the outgoing traffic. Then, by accurately organizing the time windows on every time aware shaper, it is possible to guarantee the latencies of the data flows of interest.

The Time Aware Shaper, as the name suggests, relies on the accuracy of the time measure of every device in the network; the time synchronization area deals with the methods to provide to all the nodes a common and shared measure, in order to better synchronize all the gates.

Lastly, system configuration is the part of the standards that describes how to correctly setup a time sensitive network in an automatic way. This is a procedure which needs to be performed before the start of the operations and basically consists in producing suitable schedules for the ports in the system and delivering them to the related nodes. All of that in the most

flexible and interoperable way possible.

Among the three areas just mentioned, system configuration is the less developed, especially in terms of interoperability: the currently existing state of the art solutions feature vendor-dependent methods which are not compliant with the purposes of the standards.

5.1.2 Testing

There is no abundance of material on the internet concerning practical tests about Time Sensitive Networks, with particular focus on the implementation details necessary to personally build a setup and perform some tests. The most interesting set of tutorials was provided by Intel, which has been long active in the field of TSN. On its GitHub page there is a complete set of demos, accurately described and detailed in a comprehensive document, that implement some of the TSN features on Linux-based systems, in order to perform some trivial performance tests. Since this was in line with the objectives of this thesis, we decided to replicate the test and build our own time sensitive network.

The test involves one talking node which is supposed to send TSN messages to a listening node, by means of a direct ethernet connection. During our activity we discovered that the time aware features are implemented in Linux by means of a kernel module which is called TAPRIO. Via the traffic control utility it is possible to load said module and to install the TAPRIO queueing discipline, which acts as a time aware shaper on one of the output ports of the system. Of course it is also necessary to specify a suitable schedule and the selection criteria that have to be followed to divide packets in the queues; these tasks, which usually pertain to the system configuration area, were performed manually, due to the simple topology of the tests.

The demos also included a section dedicated to the time synchronization of devices, which was based on the Precision Time Protocol. We managed to configure the systems in such a way to have less than 1 μ s of delay between the system clocks of the two nodes, which is widely acceptable, in particular for the purposes of a time sensitive network.

Once system configuration and time synchronization had been performed, we would launch the nominal tests in which the talking device would send several TSN packets to the listening device every millisecond; furthermore, the network was flooded with a huge amount of interfering traffic, generated by the talking device as well. We verified that the time aware shaper correctly managed to handle all the packets in its output queues, allowing the TSN packets to leave the device with their intended frequency while leaving to the others the remaining bandwidth. We also checked that in normal conditions, namely with the default queueing discipline, the behavior of the traffic was way less deterministic and the deadlines of the TSN data were hardly met.

Besides the complete lack of any system configuration tool, we detected

the presence of some other anomalies and inconsistencies in the setting up procedure of the time aware shaper, which suggested that, even more than the standards themselves, the hardware and software support has yet to be fully developed. However, the obtained results were in fact encouraging and useful to better understand the working principles and dynamics of a time sensitive network, in addition to the implementation details.

5.1.3 Scheduling

Scheduling is actually the main idea behind a Time Sensitive Network, even though right now the attention is mostly focused on the time aware shaper and the other necessary mechanisms. As a matter of fact, what the standards describe is just a set of tools that provide a higher level of control over a switched network: scheduling represents, in a way, how we use such tools in order to achieve our goals. A proper schedule is supposed to exploit the working gating mechanism to control all the ports in the network and accurately plan every detail about the transmission of the time triggered flows.

However, as we said, since the hardware and software infrastructure supporting TSN is not fully completed, the scheduling problem is mostly treated in academic papers, without an actual implementation.

We studied several solutions concerning the scheduling problem, selecting two of them to be further analyzed and to be compared. Scheduling algorithms can be divided in optimization algorithms and heuristic algorithm, i.e. solutions that are guaranteed to be the best and solutions which are not the best but allow to save time and resources.

To the first category belongs the Joint Routing and Scheduling algorithm, which is an optimization problem that solves both the routing and the scheduling problems for a time sensitive network. This means that the solution will provide a suitable schedule for all the ports, containing the time windows associated to each packet, and also the routes that TSN flows will have to follow to get to their destination. We need to express the constraints derived by the physical, scheduling, routing and application domains in mathematical forms, in order for them to be met in the solution. Then, by exploiting a solver for ILP problems, which is a particularly common kind of optimization problem, a solution can be computed. The possibility to determine both the time windows and the route for each packet extends the solution space, allowing for a solution even in the hardest cases, but also increases the complexity of the algorithm.

Heuristic algorithms, instead, such as the No-wait packet scheduling, do not guarantee an optimal solution but are the most common solution in practical cases because they allow to compute a solution in acceptable time for problems otherwise unsolvable with the optimal approach. In particular, NWPS tries to schedule TSN flows so as they all start as soon as possible

within the cycle and so as they are never constrained to wait. There is also a research part which is aimed at the maximization, in a weaker sense, of the general throughput of the network. The main advantage of this approach is that, as we said, it is able to compute solutions for a larger set of tasks in more complex networks; furthermore, thanks to the "smart" guiding principles that we mentioned before, it has some interesting fault tolerant and flexible features which make it the better choice in a practical case for a scheduling problem.

We implemented a version of the NWPS algorithm on the software MATLAB/Simulink, in order to test its performances. The coding itself and the subsequent tests helped us to value all the positive features of this kind of solution, in contrast to an optimal one. We were in fact able to test the algorithm on a scheduling problem, which was simulated but as generic as possible, and to correctly prove the claims of the papers in terms of performances and effectiveness.

5.1.4 Simulation

Simulation is the fastest and cheapest way to test some features of complex systems, such as a Time Sensitive Network. Several simulation environments already existed which were suitable and fully supported TSN featured, but they all lacked the computing power and libraries useful to implement the scheduling algorithm. On the other hand, MATLAB/Simulink, while not fully equipped to simulate a time sensitive network, had all the potential to build the necessary simulation libraries and also to perform the complex matrix operations needed to compute the schedules.

Therefore we started from the simple blocks of the SimEvents library and built the main components of a switched network with TSN features: from the routing system to the time aware shaper, up to an effective and complete way to record the results and all the working parameters.

In the end we created a set of elements, basically consisting in hosts and various types of switches, which could be combined together to form a network of any topology. It was also necessary to perform a kind of configuration step of the network, providing it a structure with all the connections, along with the taskset and other details.

We performed some test simulations in order to validate the just built simulation environment, by comparing its results with the ones of other simulations or even with the experimental results. We started with the tests personally conducted about the time aware shaper, replicating in simulation the setup and the initial conditions and we found exactly identical results.

We also exploited a paper describing a more complex set of tests, performed both experimentally and on another simulation environment; the paper gave all the necessary details, in terms of schedule and working parameters, to correctly setup our simulation to be a close replica. Again, the

results were very similar to the ones reported in the paper, indicating that our simulation blocks were in fact suitable to approximate the behavior of a real-life time sensitive network. This conclusion is also supported by the fact that the mechanisms that we replicate are conceptually rather simple and straightforward; the complexity comes when we consider systems made by several nodes, which implies the presence of many ports, all of which need to be modelled and taken into account for the creation of the specific schedules.

Eventually, we used our simulation environment to test the scheduling algorithm that we implemented, namely the No-wait packet scheduling, fully exploiting the computing capabilities of MATLAB and the simulation tools of Simulink. As we expected, the computed schedules adapted to the type of network and the specific taskset assigned, allowing the TSN flows to correctly reach the destination within their deadlines.

5.2 Future developments

By definition, Time Sensitive Networking is an evolving topic, therefore the set of possible aspects that can be developed in the future is quite large. We have divided all the feasible goals in two groups, namely the ones that may still need some time to be achieved and the ones that can be explored in the next future.

5.2.1 Long-term objectives

As we said, the standards themselves describing the common and most important features of time sensitive networks are waiting to be finished by the IEEE association, therefore the primary objective is to be able to read and analyze their final versions, in order to actually manage to produce the supporting hardware and software technology.

This reasoning can be applied, first of all, to the time aware shaping mechanism which, as we have seen, still has some compatibility issues and, specifically the one we got to test, is available only on the latest versions of the Linux operating system.

The time synchronization area is already quite well developed, mostly due to the fact that it is not an original feature of TSN, as there already existed methods and protocols with the same exact purpose, even if they were meant for different applications.

On the other hand, the system configuration area is hardly complete, since there currently does not exist a common and standardized method of network organization adopted by vendors. As we have found in several occasions, at the moment the few vendors of TSN technology rely on proprietary systems and procedures of network's configuration, making it hard to employ devices provided by different producers in a flexible and interoperable way;

this, according to the standards, is one of the final and most important objectives. Remaining in this area, the scheduling problem is a topic which is not even mentioned in the standards but has a central role in a time sensitive network. This work starts to take into account these kinds of problems as it takes an already existing solution and provides implementation and testing results about its performances and effectiveness.

5.2.2 Short-term objectives

We conclude with a brief list of ideas and projects that can be put into practice in the next future.

As far as the simulation is concerned, it could be integrated with more features belonging to the pure network communication framework, such as the IP protocol, MAC addresses and so on and so forth. This in order to simulate the network as a whole and not just the features of interest of the TSN framework. As we already said, there isn't quite yet the infrastructure suitable to support a complete system configuration able to automatically compute the schedules, but nonetheless the scheduling topic could be explored as well, either with different proposals for new algorithms or improvements of the existing ones. In addition, an actual software implementation, not for simulation purposes, will be required, at some point.

Furthermore, TSN-compatible switches are already on the market, and the objective of another TSN-related work could be to actually analyze and test them. By a simple research on the internet, using the keywords "managed industrial ethernet switches", it is possible to find several models of TSN switches, manufactured by as many companies. Some examples are Moxa, Belden, Winsystems, Cisco and Phoenix Contact.

Eventually, the implementaion details discovered in the chapter dedicated to the tests can be exploited and employed in a more real setup (e.g. an actual machine-to-machine communication network), different from the one that we used, which was simplistic and illustrative. To this end, Beckhoff is working on the developement of a particular module (the EK1000) which is supposed to communicate directly with the EtherCAT fieldbus and to extract from it some TSN data flows, which in turn can be injected in a proper Time Sensitive Network. A detailed set of tests would then be required to verify its effectiveness and the possible use cases in practical applications.

Bibliography

- [1] Julius Pfrommer, Andreas Ebner, Siddharth Ravikuma, Bhagath Karunakaran, (2018), *Open Source OPC UA PubSub over TSN for Realtime Industrial Communication*.
- [2] James F. Kurose, Keith W. Ross, (2013), *Computer networking: a top-down approach*, Pearson
- [3] Andreja Rojko, (2017), *Industry 4.0 Concept: Background and Overview*
- [4] Hugh Boyes, , Bil Hallaq, Joe Cunningham, Tim Watson (2018), *The industrial internet of things (IIoT): An analysis framework*
- [5] Wolfgang Mahnke, Stefan-Helmut Leitner, Matthias Damm (2009), *OPC Unified Architecture*, Springer
- [6] Eric Gardiner (2017), *Theory of Operation for TSN-enabled Systems*
- [7] Cisco Systems Inc (2017), *Time-Sensitive Networking: A Technical Introduction*
- [8] Dorin Maxim, Ye-Qiong Song (2017), *Delay Analysis of AVB traffic in Time-Sensitive Networks (TSN)*
- [9] How to run OPC UA stack open62541 with Realtime PubSub on Realtime Linux and TSN using Intel i210 Ethernet card, <https://www.kalycito.com/how-to-run-opc-ua-open62541-with-realtime-pubsub-on-realtime-linux-and-tsn-from-source/>
- [10] Adopting Time-Sensitive Networking (TSN) for Automation Systems, <https://software.intel.com/content/www/us/en/develop/articles/adopting-time-sensitive-networking-tsn-for-automation-systems-0.html>
- [11] Intel's account on GitHub, https://github.com/intel/iotg_tsn_ref_sw/tree/apollolake-i
- [12] Carlos San Vicente Gutierrez, Lander Usategui San Juan, Irati Zamalloa Ugarte, Victor Mayoral Vilches, (2018), *Real-time Linux communications: an evaluation of the Linux communication stack for real-time robotic applications*

-
- [13] Eike Schweissguth, Peter Danielis, Dirk Timmermann, Helge Parzyjegl, Gero Muhl (2017), *ILP-Based Joint Routing and Scheduling for Time-Triggered Networks*
- [14] Frank Durr, Naresh Ganesh Nayak (2017), *No-wait Packet Scheduling for IEEE Time-sensitive Networks (TSN)*
- [15] Silviu S. Craciunas, Ramon Serna Oliver, Martin Chmelik, Wilfried Steiner (2016), *Scheduling Real-Time Communication in IEEE 802.1Qbv Time Sensitive Networks*
- [16] Dong Xie, Jiang Li, Huisheng Gao (2019), *Comparison and Analysis of Simulation methods for TSN Performance*
- [17] Jonathan Falk, David Hellmanns, Ben Carabelli, Naresh Nayak, Frank Durr, Stephan Kehrer, Kurt Rothermel (2019), *NeSTiNg: Simulating IEEE Time-sensitive Networking (TSN) in OMNeT++*
- [18] Junhui Jiang, Yuting Li, Seung Ho Hong, Mengmeng Yu, Aidong Xu, Min Wei (2019), *A Simulation Model for Time-sensitive Networking (TSN) with Experimental Validation*