Universidade de Aveiro
Departamento de Eletrónica, Telecomunicações e Informática
2020

Guilherme Moura Reis Correia Gil

Modelação e simulação de equipamentos de rede para Indústria 4.0

Modelation and simulation of network equipment for Industry 4.0

Universidade de Aveiro
2020

Departamento de Eletrónica,
Telecomunicações e Informática

Guilherme Moura Reis Correia Gil

Modelação e simulação de equipamentos de rede para Indústria 4.0

Modelation and simulation of network equipment for Industry 4.0

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Eletrónica e Telecomunicações, realizada sob a orientação científica do Doutor Paulo Bacelar Reis Pedreiras, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Luís Emanuel Moutinho da Silva, Professor Adjunto Convidado da Escola Superior de Tecnologia e Gestão de Águeda.

Dedico esta dissertação à minha família, namorada e amigos pelo apoio que me deram.

o júri / the jury

presidente / president

Prof. Doutor Manuel Bernardo Salvador Cunha
Professor Auxiliar, Universidade de Aveiro

vogais / examiners committee

Prof. Doutor Luís Miguel Pinho de Almeida
Professor Associado, Faculdade de Engenharia da Universidade do Porto

Prof Doutor Luís Emanuel Moutinho da Silva
Professor Adjunto Convidado, Universidade de Aveiro

agradecimentos /
acknowledgements

Gostaria de agradecer, em primeiro lugar, aos meus orientadores, Professor Doutor Paulo Pedreiras e Professor Doutor Luís Moutinho, pela excelente orientação e dedicação que proporcionaram ao longo deste percurso. Obrigado por estarem disponíveis sempre que precisasse, pela paciência em tolerar os meus erros e pela forma como me trataram. Espero que tenham tido tanto gosto em trabalhar comigo como eu tive com ambos desejando que, num futuro próximo, continuaremos a trabalhar juntos.

obrigado à minha namorada, Andreia Andrade, por ser aquela pessoa que me consegue relaxar até nos momentos mais difíceis. Obrigado por todo o amor que me deste e por todos os momentos que passámos juntos nestes últimos cinco anos. Agora vais ter de ficar comigo para o resto da vida :p.

obrigado aos meus amigos por todo o apoio e ajuda que me deram nestes últimos cinco anos. Em particular gostaria de agradecer ao Ivo Oliveira, Marco Sousa e Miguel Tavares por todos os bons momentos que despendemos a jogar às copas no aquário do DETI enquanto deveria estar a trabalhar na dissertação.

por último, mas não com menor importância, obrigado à minha família por me aturar nestes últimos meses, nos quais estivemos de ficar confinados. Eu sei que às vezes posso ser um bocado chato. Obrigado aos meus pais, Zé Toninho e Geninha, por todo o apoio, carinho e dedicação que me deram ao longo da minha vida e por me fazerem a pessoa que sou hoje, o filho favorito ;).

Palavras Chave

Redes de Comunicação Tempo-Real, Simuladores de Redes, Ethernet Tempo-Real, Indústria 4.0.

Resumo

Atualmente o setor industrial tem vindo cada vez mais a optar por tecnologias digitais de forma a automatizar todos os seus processos. Este desenvolvimento surge de noções como Indústria 4.0, que redefine o modo de como estes sistemas são projetados. Estruturalmente, todos os componentes destes sistemas encontram-se conectados numa rede complexa conhecida como Internet Industrial das Coisas. Certos requisitos advêm deste conceito, no que toca às redes de comunicação industriais, entre os quais se destacam a necessidade de garantir comunicações tempo-real bem como suporte a uma gestão dinâmica dos recursos, os quais são de extrema importância. Várias linhas de investigação procuraram desenvolver tecnologias de rede capazes de satisfazer tais exigências. Uma destas soluções é o "Hard Real-Time Ethernet Switch" (HaRTES), um switch Ethernet com suporte a comunicações de tempo-real e gestão dinâmica de Qualidade-de-Serviço (QoS), requisitos impostos pela Indústria 4.0.

O processo de projeto e implementação de redes industriais pode, no entanto, ser bastante moroso e dispendioso. Tais aspetos impõem limitações no teste de redes de largas dimensões, cujo nível de complexidade é mais elevado e requer o uso de mais hardware. Os simuladores de redes permitem atenuar o impacto de tais limitações, disponibilizando ferramentas que facilitam o desenvolvimento de novos protocolos e a avaliação de redes de comunicações.

No âmbito desta dissertação desenvolveu-se um modelo do switch HaRTES no ambiente de simulação OMNeT++. Com um objetivo de demonstrar uma solução que possa ser utilizada em redes de tempo-real industriais, esta dissertação apresenta os aspetos fundamentais do modelo implementado bem como um conjunto de experiências que o comparam com um protótipo laboratorial já existente, no âmbito da sua validação.

Keywords

Real-Time Communication Networks, Network simulators, Real-Time Ethernet, Industry 4.0.

Abstract

Currently, the industrial sector has increasingly opted for digital technologies in order to automate all its processes. This development comes from notions like Industry 4.0 that redefines the way these systems are designed. Structurally, all the components of these systems are connected in a complex network known as the Industrial Internet of Things. Certain requirements arise from this concept regarding industrial communication networks. Among them, the need to ensure real-time communications, as well as support for dynamic resource management, are extremely relevant. Several research lines pursued to develop network technologies capable of meeting such requirements. One of these protocols is the Hard Real-Time Ethernet Switch (HaRTES), an Ethernet switch with support for real-time communications and dynamic resource management, requirements imposed by Industry 4.0.

The process of designing and implementing industrial networks can, however, be quite time consuming and costly. These aspects impose limitations on testing large networks, whose level of complexity is higher and requires the usage of more hardware. The utilization of network simulators stems from the necessity to overcome such restrictions and provide tools to facilitate the development of new protocols and evaluation of communications networks.

In the scope of this dissertation a HaRTES switch model was developed in the OMNeT++ simulation environment. In order to demonstrate a solution that can be employed in industrial real-time networks, this dissertation presents the fundamental aspects of the implemented model as well as a set of experiments that compare it with an existing laboratory prototype, with the objective of validating its implementation.

Contents

List of Figures

List of Tables

Glossary

ART	Asynchronous Requirements Table
AW	Asynchronous Window
CAN	Controller Area Network
COTS	Commercial-Of-The-Shelf
CSMA/CD	Carrier-Sense Multiple-Access with Collision Detection
DES	Destributed Embedded Systems
DM	Deadline Monotonic
DS	Deferrable Server
DRTS	Distributed Real-Time Systems
EC	Elementary Cycle
EDF	Earliest Deadline First
EDP	Explicit Deadline Periodic
ET	Event-Triggered
FCFS	First-Come First-Served
FTT	Flexible Time-Triggered
FTTRS	Flexible-Time-Triggered Replicated Star
FTT-SE	Flexible Time-Triggered Switched Ethernet
HaRTES	Hard Real-Time Ethernet Switch
HSF	Hierarchical Scheduling Framework
IDE	Integrated Development Environment
LAN	Local Area Networks
MAC	Media Access Control
mit	Minimum Interarrival Time
NED	Network Description
NRT	Non real-time
OSI	Open Systems Interconnection
QoS	Quality-of-Service
RM	Rate Monotonic
SLA	Service Level Agreement
SRDB	System Requirements Database
SRT	Synchronous Requirements Table
SW	Synchronous Window
TM	Trigger Message
TT	Time-Triggered
TS	Timeline Scheduling
TSN	Time-Sensitive Networking
VID	Virtual Local Area Network (VLAN) Identifier
VLAN	Virtual Local Area Network
WCET	Worst-Case Execution Time
WCRT	Worst-Case Response Time

CHAPTER 1

Introduction

Contents

*T*he concept of Industry 4.0 is used to describe the current industrial revolution which proceeds the three previous ones. In the 18th century, the shift from manual labor to mechanical production, through steam and water power, led to the first industrial revolution. The second industrial revolution occurred during the late 19th century, with the introduction of electricity in the manufacturing process which improved the production by creating assembly lines. Throughout the 1970s, the third revolution stemmed from the introduction of digital electronics and communication technologies in the industrial sector. Industry 4.0 forecasts the digitalization of the industrial process using three major technologies: Internet- of-Things, Cyber-Physical Systems and Cloud Computing. In this transformation, computer systems, sensors and intelligent machines are integrated into a single network, also known as Industrial Internet-of-Things (IIoT). As several connected systems can interact with each other, exchanging information grants more flexibility and autonomy to factories with machines autonomously analyzing data, predicting failures and self-reconfigure when necessary. Besides gains in flexibility, Industry 4.0 has significant improvements in efficiency and cost reduction, productivity, human-machine interaction and product quality.

1.1 Problem Statement

Although the notion of Industrial Internet of Things offers considerable advantages for industrial sector, the need to interconnect the factory different components makes the requirements of industrial communication networks increasingly demanding. As the exchange of real-time information between the several applications is essential for the accuracy and correctness of the system, the implementation of a communication system that guarantees their timeliness requirements, is a crucial process in the development of this kind of networks.

Among the different industrial communication infrastructures employed over the years, Ethernet quickly became the most widely used due to several advantages that the protocol

provided. As the non-deterministic arbitration mechanisms of standard Ethernet did not guarantee the capability of meeting the, often strict, real-time requirements of industrial applications, several researchers strived to add support for real-time traffic over switched Ethernet. Such technologies include Time-Sensitive Networking (TSN) and Hard Real-Time Ethernet Switch (HaRTES) protocols. However, the process of developing and testing industrial networks using physical hardware can be complex, expensive and, for certain case studies, considerably limited. The development of network simulators allows to circumvent these issues. The creation of software models of the prototypes used in the industrial sector makes the process of designing industrial networks significantly easier, cheaper and faster. This work aims to integrate technologies from Industry 4.0 and network simulators with the development of a HaRTES switch simulation model for industrial real-time networks.

1.2 Objectives

The main objectives of this dissertation are:

- Study of industrial real-time Ethernet protocols, focusing on technologies that use the FTT paradigm;
- Survey of network simulators used in the scope of industrial applications and selection of one of them;
- Implementation of simulation models for real-time Ethernet switches and their subsequent validation;
- Analysis of the retrieved results.

1.3 Document Outline

This dissertation is organized as follows:

- ***Chapter 2***: starts by introducing some concepts regarding real-time systems and scheduling techniques for both synchronous and asynchronous real-time traffic. Then, it addresses the Ethernet protocol and presents some existing real-time technologies based on Ethernet switching.
- ***Chapter 3***: is dedicated to the FTT-enable switch, HaRTES. It starts by introducing the FTT paradigm and its implementation over switched Ethernet, the FTT-SE protocol. Then discusses the objectives and architecture of the new switch and compares it to its previous technology. Lastly, some improvements performed over the HaRTES are presented.
- ***Chapter 4***: presents some of the most commonly used network simulators and compares them to find the best candidate to implement the proposed switch. The chapter finishes with a detailed description of the selected simulator.

- ***Chapter 5***: describes the created simulation models and their components.
- ***Chapter 6***: is dedicated to the experiments conducted to validate the developed models and the result analysis.
- ***Chapter 7***: presents the conclusions of the dissertation and some possible lines for future research.

CHAPTER 2

Theoretical Real-Time and Ethernet concepts

Contents

*T*his chapter presents some basic concepts regarding real-time systems and the Ethernet protocol, which are fundamental for understanding concepts introduced throughout this dissertation. It initiates with the discussion of real-time scheduling algorithms and communications. The chapter closes with the presentation of Ethernet technologies and real-time communication protocols over the Ethernet switch.

2.1 Real-Time Systems

Sectors such as chemical, nuclear and flight control, automotive applications and military systems potentially experience extreme consequences when the timeliness constraints of their systems are not met [1]. Therefore, when designing these systems, i.e., ***real-time systems***, besides having to consider the correctness of the computed results, also required to guarantee

that computations, as well as communications, for the case of distributed real-time systems, satisfy specific timing requirements, being ***deadline*** one of the more common ones.

2.1.1 Classification of real-time systems

In real-time systems, computational activities are described as a set of real-time tasks. Each task is commonly characterized by its individual *deadline*, a constraint that restricts the task response time, and can be classified depending on the consequences of a missed deadline [1]:

- **Soft:** When a task misses a deadline produces, it might produce useful results for the system, although the delay may cause degradation to its quality and performance.
- **Firm:** The tasks results after a missed deadline have no utility for the system. Despite there may be some performance degradation, these are tolerable depending on their frequency, but there are no catastrophic consequences that result from such events.
- **Hard:** The tasks results after a missed deadline are useless. Furthermore, catastrophic consequences, such as human losses or significant costs, may emerge.

Depending on the type of task handled, real-time systems can then be classified as [1]:

- **Soft Real-Time:** Systems that only handle firm and/or soft real-time tasks. These are usually used in applications where the miss of a deadline is less critical such as video/audio encoding and decoding, video streaming and image processing.
- **Hard Real-Time:** Systems that execute at least one hard real-time task. Also known as **safety-critical systems**, these are used in several activities: missile control, airplane control and nuclear plant control.

Real-time systems are generally associated with a set of tasks. Tasks can normally have three different constraints [1]: **temporal constraints, precedence constraints** and **resources constraints**.

- **Temporal constraints**: typically associated with each task's *deadline*. However, it can also be correlated to the ***window*** (Upper and lower bounds in which a task must finish its execution), ***synchronization*** (Temporal boundary of the difference between two generated events), or the ***distance*** of a task (time limit between completion and consequent activation).
- **Precedence constraints**: associated with the order that the tasks are executed. If a task (T_a) can only execute after another task (T_b) finishes, T_b is specified as the *predecessor* of T_a. If a task has no predecessors, it is nominated as *beginning task* whereas tasks with no *successors* are called *ending tasks*.
- **Resources constraints**: applied to a set of tasks whenever they must access the same resource (*shared resource*) to continue their execution. For such cases it is necessary to ensure that it is not accessed simultaneously by multiple tasks (***mutual exclusion***).

The previously explained constraints will influence the moment when tasks are executed. Real-time systems have an entity responsible for selecting the following task to execute from a current set which is designated ***scheduler***. It is possible to perform a *scheduling analysis* to assess, if a given task set is *schedulable* (there is at least one feasible schedule), and then arrange the task order by applying *scheduling algorithms*. Besides the scheduler, a generic real-time system/kernel has the subsequent components [2]:

- **Task Manager:** responsible for creating, deleting, setting the initial activation and updating each task state.
- **Time Manager:** accountable for activating the tasks, verifying time constraints and measure time intervals.
- **Resource Manager:** grants mutual exclusion of a shared resource to the tasks (mutexes, semaphores, etc.).
- **Task Dispatching:** picks the selected task from the scheduling process and puts it in execution.

Even though a real-time system is formed by multiple entities, the scheduler is the one that requires more attention because it is the responsible structure for implementing complex algorithms that arrange the execution order of a task set. The following sections briefly explain several real-time scheduling concepts.

2.1.2 Task Model

A task [1] is a computational process that performs a specific instruction executed by the CPU. It can be classified as *periodic*, *aperiodic* and *sporadic* depending on its activation. Periodic tasks arrive regularly with a fixed time interval between two consecutive activations (*inter arrival-time*). Sporadic tasks are also activated regularly but with a minimum inter arrival-time instead of a fixed one. Finally, for aperiodic tasks, their arrival cannot be predicted nor constrained and can only be characterized by probabilistic means.
A set of periodic tasks can be characterized by the following model:

$$\Gamma = \{\tau_i(C_i, T_i, \phi_i, D_i, Pr_i), \;\; i = 1, 2, ..., n\} \tag{2.1}$$

where:

- C_i is the worst case computation time required to compute task τ_i, also designated as *Worst-Case Execution Time (WCET)*;
- T_i is the period of task τ_i;
- ϕ_i is the initial phase of task τ_i (the task first activation offset time);
- D_i is the relative deadline of task τ_i;

- Pr_i is the priority of task τ_i;

The release/activation time ($r_{i,k}/a_{i,k}$) and absolute deadline ($d_{i,k}$) of the k^{th} instance of a task τ_i can be computed as:

$$r_{i,k} = \phi_i + (k-1)T_i, \quad i \in \mathbb{N} \tag{2.2}$$

$$d_{i,k} = r_{i,k} + T_i = \phi_i + kT_i, \quad i \in \mathbb{N} \tag{2.3}$$

The model (2.1) can be applied to sporadic tasks if the period parameter (T_i) is changed to the *minimum inter-arrival activation time* ($Tmit_i$) and the phase parameter (ϕ_i) is excluded:

$$\Gamma = \{\tau_i(C_i, Tmit_i, D_i, Pr_i), \;\; i = 1, 2, ..., n\} \tag{2.4}$$

For aperiodic tasks, the release instant and absolute deadline value can be computed as follows:

$$r_{i,k} \geq r_{i,k-1} + Tmit_i, \quad i \in \mathbb{N} \tag{2.5}$$

$$d_{i,k} = r_{i,k} + D_i, \quad i \in \mathbb{N} \tag{2.6}$$

2.1.3 Scheduling

In real-time systems, the scheduler is responsible for assigning each task execution (*scheduling*) by applying different algorithms and rules (*scheduling algorithms*). According to [1], the scheduler requires three sets to create a schedule: a set of n tasks $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, a set of m processors $P = \{P_1, P_2, ..., P_n\}$ and a set of s types of resources $R = \{R_1, R_2, ..., R_n\}$. Besides these, precedence relations between tasks can be specified through cyclic graphs. The scheduler ensures the execution of all tasks under the imposed constraints by assigning processors from P and resources from R to tasks from Γ. Real-time scheduling algorithms can be classified as [1]:

- **Preemptive** *vs* **Non Preemptive**:
  - – In preemptive algorithms, a task execution can be interrupted by assigning the processor to another one..
  - In non-preemptive algorithms, a task always completes its execution when it is assigned to the processor.
- **Static** *vs* **Dynamic (priorities)**:
  - In Static priority algorithms, the scheduler selects the task order based on their fixed priorities. These are assigned to each task before their release.
  - In Dynamic priority algorithms, the task's priorities may vary through the system execution, and the scheduling is based on these changeable parameters.

- **Offline** *vs* **Online**:
  - In Offline scheduling, the scheduling algorithm is applied to the task set before their activation. The generated scheduler is then stored and executed during run-time.
  - In Online scheduling, all the scheduling decisions are made in run-time, whenever a task completes its execution or a new one is added to the task set.
- **Optimal** *vs* **Sub-Optimal/Heuristic**:
  - A scheduling algorithm is optimal if it minimizes the cost function assigned to the task set. However, if the cost function was not defined, an optimal scheduling algorithm is one able to find a feasible schedule for the task set, provided that one exists, computed with an algorithm of the same class.
  - A heuristic algorithm is one that tends to find the optimal schedule using heuristic functions.

A common taxonomy used for real-time scheduling is illustrated in Figure 2.1. The algorithms are generally divided into *online* and *offline* scheduling, with offline being more suitable for complex algorithms with high computational requirements. However, it is less flexible as, unlike online scheduling, it does not support changes that may occur in the system. Regarding online scheduling, it can be further classified depending on the priorities assigned to the tasks: *static* or *dynamic*. Both these categories can be *preemptive* or *non-preemptive*.

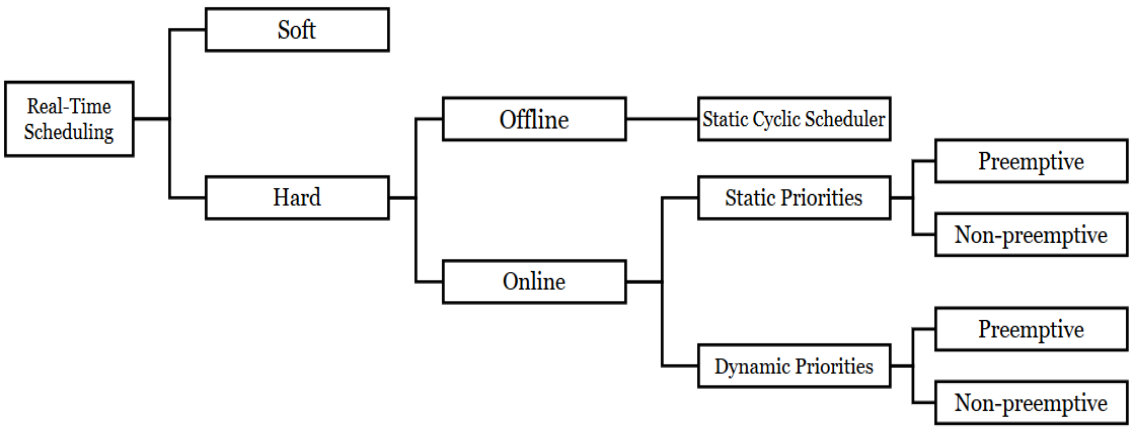


Figure 2.1: Scheduling algorithms.

2.1.3.1 Periodic Task Scheduling

The following segments present some of the most used techniques [1]: Timeline Scheduling (TS), Rate Monotonic (RM), Deadline Monotonic (DM) and Earliest Deadline First (EDF).

2.1.3.1.1 Offline Scheduling

Timeline Scheduling

Timeline Scheduling (TS) [1], also known as *Cyclic Executive*, is a scheduling algorithm that splits the time interval assigned to the tasks execution in equal segments of time (time slots). Each task is allocated to a given time slot in a way that all their requirements (frequency and deadline) are respected. In the TS algorithm, a time slot is denominated *Minor Cycle*, whereas the *hyperperiod* (minimal time interval from which the scheduler repeats itself) is designated *Major Cycle*. Prior to the system's execution, a schedule, in which all the tasks are associated with Minor Cycles for a whole Major Cycle, is created and stored in a scheduling table. The dispatcher then uses it and executes each task in their predetermined Minor cycle. Even though this type of scheduling algorithm has a low complexity level, being an offline technique makes it limited in terms of flexibility.

2.1.3.1.2 Online Scheduling

Fixed Priorities - Rate Monotonic

Rate Monotonic (RM) [3] is an online preemptive algorithm based on fixed priorities where task's priorities (P_i) are assigned monotonically based on their periods, with lower periods corresponding to higher priorities:

$$\forall_{\tau_i,\tau_j} \in \Gamma : T_i < T_j \rightarrow P_i > P_j \tag{2.7}$$

Since it is an online technique, whenever a new task is activated or a running one completed, the scheduler must select, at run-time, the task with the shortest period in the set for execution. Furthermore, as the RM is a preemptive algorithm, running tasks can be interrupted if a new one with a shorter period is activated. The Rate Monotonic was proven to be the optimal algorithm among all the fixed-priorities techniques by Liu and Layland [3], for $D_i = T_i$. They demonstrated that any fixed-priority scheduling algorithm can only schedule a task set if it is schedulable with RM.

Fixed Priorities - Deadline Monotonic

The *Deadline Monotonic* (DM) algorithm was created as an extension of Rate Monotonic by Leung and Whitehead [4]. It is an online technique with fixed priorities in which tasks have relative deadlines shorter or equal to their periods, i.e., $C_i \leq D_i \leq T_i$. Following the DM scheduling, priorities are monotonically assigned according to the relative deadlines (D_i), with tasks with lower deadlines being assigned higher priorities. Similar to Rate Monotonic, the Deadline Monotonic is also a preemptive algorithm, meaning that an executing task can be interrupted when a new one with a lower deadline is activated. Furthermore, the DM algorithm was also proven to be an optimal fixed-priority algorithm by Liu and Layland [3], meaning that the fixed-priority scheduling conditions applied to RM are also used in this algorithm.

Dynamic Priorities - Earliest Deadline First

The *Earliest Deadline First* (EDF) [3] is a preemptive technique based on dynamic priorities in which the tasks relative deadlines are equal to their periods: $\forall_{\tau_i \in \Gamma} : D_i = T_i$. In EDF, the tasks priorities are assigned according to their absolute deadlines (d_i), i.e., tasks with earlier deadlines relatively to the current execution time assigned higher priorities:

$$\forall_{\tau_i,\tau_j} \in \Gamma_{T_a} : d_i < d_j \rightarrow P_i > P_j, \quad Ta \in \mathbb{R} \tag{2.8}$$

where Γ_{T_a} is subset of Γ, composed by ready tasks at the time instant T_a, and (d_i, d_j) the absolute deadlines at T_a for both tasks τ_i and τ_j. Comparing to the Rate Monotonic scheduling algorithm, the EDF is capable of achieving higher utilization factors while reducing the preemption levels of the overall system and guaranteeing timeliness. Nonetheless, it is a more complex technique, which can be problematic for systems with low processing power due to the algorithm's high run-time overhead.

2.1.3.2 Aperiodic Task Scheduling

Asynchronous tasks [1], commonly external to the system, are event-triggered based and cannot be controlled by the scheduler. Hence, in order to maintain the real-time system requirements, with the integration of both synchronously and asynchronously task sets, some mechanisms are applied to the latter so that their overall interference can become predictable (deterministic) or, at least, bounded.

For that purpose, two types of techniques were created, one based on fixed priorities and the other on dynamic priorities. Some of these algorithms will be presented in the following segments.

2.1.3.2.1 Background Scheduling

In *Background Scheduling* [1], the aperiodic traffic is only processed when there are no periodic tasks ready to execute. Although this procedure makes this algorithm very simple, there are no guarantees that the time constraints of the aperiodic jobs will be satisfied, particularly in high periodic load systems. Therefore, this method should only be used for aperiodic task sets with limited timeless constraints and low periodic loads.

2.1.3.2.2 Server-based Scheduling

Server-based scheduling [1] is a technique that makes use of a periodic task (server) to act as a proxy of associated sporadic or aperiodic tasks. A server can be described by a given period T_s, and an execution time C_i (server capacity or budget). The server is scheduled just like a regular periodic task by the scheduler and, when active, allows requests execution made by aperiodic or sporadic tasks within its budget.

The server-based scheduling mechanisms are divided into static and dynamic priority scheduling. Some of the static-priority most well-known algorithms are [1]:

- **Polling Server (PS):** The polling server activates every T_s, replenishes its capacity, and executes any aperiodic requests within its budget. If the server capacity is exhausted or there are no requests during its activation, it suspends and loses all its budget (if it has any). Aperiodic requests that arrive during the suspended state must wait for the replenishment in the next activation state. In a polling server, whenever a task execution time surpasses the server capacity, it is suspended. Thus, the interference caused by both aperiodic and periodic tasks is the same. As requests that arrive after the server activation must wait to be processed, even if there are no other higher priority active tasks, the general response time of this server is not satisfactory.

- **Deferrable Server (DS):** The deferrable server is an algorithm also characterized by a periodic replenishment with period T_s and execution time C_i. Similar to a Polling Server, a DS executes aperiodic requests within its budget, which is replenished periodically. However, in this case, instead of depleting its capacity in the absence of requests, the server preserves its budget until each period completion or its exhaustion. This improvement allows better response times compared to the Polling Server. Nevertheless, the possibility for aperiodic tasks to execute at any point within the server period causes a drawback in the schedulability of the periodic traffic.

- **Sporadic Server (SS):** Similar to the deferrable server, the sporadic server preserves its capacity until it is exhausted by aperiodic requests. The main difference is that, in this server type, the server capacity budget is only replenished when it is used. These procedures reduce the sporadic server impact when scheduling periodic tasks. The main disadvantage is the complexity of its implementation.

- **Priority Exchange (PE):** Akin to the polling server, at the beginning of each period (T_s), the priority exchange server replenishes its capacity and checks for aperiodic requests. These servers are modeled as high priority tasks and can exchange their budget with low priority ones. If the server has the set highest priority, it executes all pending aperiodic requests within its budget. On the other hand, in the presence of higher priority periodic tasks, the server grants its budget for the task execution. This type of mechanism improves the schedulability. However, it increases the complexity and worsens the response time.

Dynamic priority server scheduling is not implemented in the *HaRTES* architecture. Nevertheless, for the sake of completeness, two types of schedulers will be explained [1]:

- **Total Bandwidth Server (TBS):** The total bandwidth server is a dynamic priority-based algorithm implemented to overcome the Sporadic Server limitation regarding the requests delays, which result from long server periods. The main concept of the TBS is that, while handling the aperiodic workload, the processor utilization never exceeds a defined bound U_s. In this algorithm, whenever a new aperiodic task is added to the

system at $t = r_k$, it receives the total server bandwidth, if possible. The deadline d_k assigned to the k_{th} request that arrived at r_k is computed as:

$$d_k = max(r_k, d_{k-1}) + \frac{C_k}{U_s} \tag{2.9}$$

where C_k is the request execution time and U_s the server utilization factor (bandwidth). Upon receiving a deadline, the tasks are inserted into a queue where they are treated as normal periodic tasks to be scheduled by the EDF algorithm. Although the average response time for aperiodic tasks is smaller, when comparing to other dynamic/static priority servers, Spuri and Buttazzo [1] proved that a set of n periodic tasks with a utilization rate of U_p is only schedulable with the EDF by the TBS with a utilization factor of U_s when:

$$U_p + U_s \leq 1 \tag{2.10}$$

The main advantages of this server are its simplicity and low overhead. On the other hand, it requires a priority knowledge of the server capacity and is vulnerable to overruns.

- **Constant Bandwidth Server (CBS):** The constant bandwidth server is a dynamic priority-based server. The CBS algorithms guarantee that a server contribution to the total utilization factor is always smaller than the total possessed bandwidth (U_s), even in the occurrence of overheads. A constant bandwidth server is characterized by a budget c_s and a pair of values (T_s, Q_s), where Q_s is the maximum budget and T_s the server period. The server bandwidth (U_s) is computed as: $U_s = \frac{Q_s}{T_s}$. When a request is added to the system, a mew scheduling deadline is assigned and afterwards stored in an EDF *queue*. While executing, whenever a task demands more bandwidth than the reserved (executes more than expected), its deadline is postponed, i.e., the tasks priority decreases due to the EDF rules. By doing so, the task interference on the overall workload is reduced. The CBS algorithm is defined as follows:
  - At each instant, a deadline $d_{s,k}$ is associated to the server and, at the beginning, $d_{s,0} = 0$.
  - All of the servers jobs $J_{i,j}$ receive dynamic deadlines $d_{i,j}$, which are equal to the servers deadline, i.e., $d_{i,j} = d_{s,k}$. Whenever a job is executed, the server budget (c_s) is decreased by the same amount.
  - When the server total budget is depleted ($c_s = 0$), it is recharged with the maximum valued pre-defined $c_s = Q_s$, and a new server deadline is generated: $d_{s,k+1} = d_{s,k} + T_s$.
  - A constant bandwidth server is defined as *active* at a time instant t if there is at least one pending job $J_{i,j}$ so that $r_{i,j} \leq t < f_{i,j}$, otherwise is said to be *idle*. Note that $r_{i,j}$ is the instant the job $J_{i,j}$ arrives and $f_{i,j}$, the worst-case finish time of the respective job.

- Whenever the server is active and a job $J_{i,j}$ arrives, it is stored in a *queue* of pending requests.
- At any time the server is idle and a job $J_{i,j}$ arrives, the server's deadline $d_{s,k}$ is computed as follows:

$$\text{if } c_s \geq (d_{s,k} - r_{i,j})U_s, \text{the deadline } d_{s,k} \text{ stays equal}$$
$$\text{otherwise a new one is generated } d_{s,k+1} = r_{i,j} + T_s \wedge c_s = Q_s$$

- After a job is finished, the next pending request is assigned to the current budget and deadline. If there are no requests, the server becomes idle.

Compared to a TBS, CBS has higher complexity due to the dynamic capacity management with similar response times for aperiodic requests. However, it has better performance whenever the tasks WCET has a high variance. A set of n periodic hard tasks, with a utilization factor U_p only can be schedulable with the EDF algorithm, using a set of m constant bandwidth servers with a processing factor $U_s = \sum_{i=1}^{m} U_{si}$, if condition (2.10) is verified.

2.1.4 Schedulability Analysis

Schedulability analysis is a computation performed by the scheduler and allows to determine, a priori, if a task set is feasible, i.e., all the time constraints of each task will be met. In [1], the most distinguished approaches are based on the *utilization rate* and *response time*. The former has generally lower computational complexity but is more pessimistic when compared to other approaches. Response time analysis tests are more complex, thus require systems with higher computational power. Nonetheless, they tend to be less pessimistic, providing more precise results.

2.1.4.1 Utilization based tests

Utilization tests are based on the fraction of time a processor spends to execute a task set $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$. This factor, generally represented as U, is calculated by the sum of each tasks contribution (u_{τ_i}):

$$u_{\tau_i} = \frac{C_i}{T_i} \tag{2.11}$$

For a uniprocessor system with n tasks, the utilization factor can be computed as:

$$U = \sum_{i=1}^{n} \left(\frac{C_i}{T_i} \right) \tag{2.12}$$

By calculating this value, and comparing it to a certain threshold, it is possible to confirm if the task set is schedulable. This bound depends on the system characteristics (task set deadlines, periods and applied scheduling algorithm).

In their study, Liu and Layland [3], proved that, for uniprocessor systems, any task set Γ of n periodic and independent tasks can be schedulable with the RM policy if the following inequality is verified:

$$U \leq n(2^{\frac{1}{n}} - 1) \tag{2.13}$$

2.1.4.2 Response Time based tests

Response time based tests are associated with the *Worst-Case Response Time (WCRT)*, i.e., the maximum interval between arrival and finish time of a task. For each task, the scheduler verifies if the WCRT is higher than their respective deadline and, if it occurs at least once, the entire set is not schedulable.

For fixed-priority preemptive systems, Joseph and Pandya [5], showed that, for a periodic task τ_i, its longest response time R_i is obtained by the sum of its computation time C_i and the amount of interference that it can suffer from higher priority tasks in the system I_i:

$$R_i = C_i + I_i \tag{2.14}$$

The maximum amount of interference occurs when a task τ_i and the remaining higher priority tasks (hp(i)) are released at the same time, also know as *critical instant*:

$$I_i = \sum_{\forall j \in hp(i)} \left(\left\lceil \frac{R_i}{T_i} \right\rceil \times C_j \right) \tag{2.15}$$

Computing equations (2.14) and (2.15) results:

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left(\left\lceil \frac{R_i}{T_i} \right\rceil \times C_j \right) \tag{2.16}$$

Equation (2.16) can be solved by the following iteration method:

$$R_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left(\left\lceil \frac{R_i^n}{T_i} \right\rceil \times C_j \right) \tag{2.17}$$

The iteration starts at R_i^0 and stops when $R_i^{n+1} = R_i^n$ or when $R_i^n > D_i$, i.e., when the longest response time exceeds the task deadline. Audsley *et.al* [6] improved this analysis in order to address the non-preemption effects of resource sharing. In this analysis, equation (2.14) is reformulated by adding a blocking factor caused by lower priority tasks B_i:

$$R_i = C_i + I_i + B_i \tag{2.18}$$

The blocking factor can be solved as:

$$B_i = \begin{cases} 0, & lep(i) = \emptyset \\ max_{j \in lep(i)}\{C_j\} & lep(i) \neq \emptyset \end{cases} \tag{2.19}$$

where *lep* represents the set of tasks with lower priority than τ_i.

It is important to highlight that, for this analysis, the *critical instant* is different due to the blocking factor. Now it happens when both τ_i and the remaining higher priority tasks are released, immediately afterward a lower priority task that blocks τ_i starts its execution.

2.1.5 Hierarchical Scheduling

Hierarchical scheduling or *Hierarchical Scheduling Framework (HSF)* [7], [8], are techniques implemented to simplify issues imposed by systems with global scheduling analysis. Apart from their complexity, these systems are more suitable for situations when the information about the systems applications established are *a priori* (*closed* systems). Nonetheless, there are multiple sectors where closed systems are not fitting as they are unable to execute applications during run-time that have not been scheduled together with the current set. Thus, different methodologies such as HSF were developed to overcome these problems.

Hierarchical scheduling allows fragmentation of global and complex systems into a set of subsystems whose individual analysis simplifies the scheduling problem. A hierarchical scheduling framework is typically represented as a tree, similar to the ones used in data structures (binary trees). Each component/subsystem is represented by nodes. These are independent structures hence, they all have individual schedulers to schedule their local task set. Each component also includes an interface that simplifies the integration of different nodes in the system by modeling their individual requirements into a single real-time requirement. Furthermore, it is also used to allocate resources between different hierarchy levels (parent $\rightarrow$ child).

According to Shin and Lee [7], [8], the following properties must be verified in any hierarchical scheduling framework:

- **Independence:** each node schedulability analysis must be performed independently to all other components of the same hierarchical level.
- **Separation:** the interaction between parent and child nodes is made through an interface that abstracts their inner complexity.
- **Universality:** all the components are independent. Thus any scheduling algorithm can be applied to them.
- **Composability:** the scheduling analysis of a parent component considers all its children's requirements. As such, these are only schedulable if all their children's time requirements are satisfied.

These properties are feasible using techniques based on server architectures as they have efficient methodologies to control resource distribution. Such scheduling techniques provide transparency when allocating virtual resources (fraction of the capacity of the corresponding hardware resources) to the components of the hierarchy. Consequently, server-based structures can be considered suitable for component-based scheduling interfaces.

A model for the construction of HSF was presented by Shin and Lee [7], [8]. In this model, a component C is defined as a triplet *(W, R, A)*, where W represents the workload (task set), R the resources available and A the scheduling algorithm that defines how the resources are shared by the workload. Using this nomenclature, two functions were established: (i) the

demand bound function and (ii) supply bound function. The former, $dbf_A(W,t)$, quantifies the maximum workload W that can be submitted to resource R under the scheduling policy A during a time interval t. The later, $sbf_R(t)$, computes the minimum possible resources that R can provide through a certain time interval t. Resource R satisfies the submitted workload W if:

$$dbf_A(W,t) \leq sbf_R(t), \quad \forall t \in \mathbb{R}^+ \tag{2.20}$$

Furthermore, the authors also defined a real-time model, $\Gamma(\Pi,\Theta)$, which characterizes the periodic allocation of the resource R by workload W. This periodic model represents a resource supply that provides Θ resource units periodically, every Π time units. Consequently, the supply bound function, $sbf_\Pi(t)$, can be defined as:

$$sbf_\Pi(t) = \begin{cases} b\Theta + max\{0, t-a-b\Pi\}, & t \geq \Pi - \Theta \\ 0, & t < \Pi - \Theta \end{cases} \tag{2.21}$$

where:

$$a = 2(\Pi - \Theta) \quad \wedge \quad b = \left\lfloor \frac{t-(\Pi-\Theta)}{\Pi} \right\rfloor \tag{2.22}$$

The model proposed by Shin and Lee, $\Gamma(\Pi,\Theta)$, was generalized for hierarchical frameworks by Arvind *et al.* [9] and named *Explicit Deadline Periodic (EDP)*. This model is also characterized by a triplet, $\Omega = (\Pi,\Theta,\Delta)$, which represents a resource supply that periodically provides, every Π time units, Θ resources units during Δ time units, where $\Delta \leq \Pi$. Thus, the periodic model $\Gamma(\Pi,\Theta)$ can be defined as (Π,Θ,Π) using the EDP structure. The supply bound function, $sbf_\Omega(t)$, is now computed as:

$$sbf_\Omega(t) = \begin{cases} b\Theta + max\{0, t-a-b\Pi\}, & t \geq \Delta - \Theta \\ 0, & t < \Delta - \Theta \end{cases} \tag{2.23}$$

where:

$$a = (\Pi + \Delta - 2\Theta) \quad \wedge \quad b = \left\lfloor \frac{t-(\Delta-\Theta)}{\Pi} \right\rfloor \tag{2.24}$$

2.2 Real Time communications

As previously described, the task's execution on real-time systems is highly conditioned by their temporal constraints. A set of real-time systems interconnected by communication systems (networks) is generally known as *Distributed Real-Time Systems (DRTS)*. Contrarily to standard real-time systems, which only impose time constraints on the tasks execution, DRTS also enforce these restrictions on exchanging messages within the network. Therefore, these systems behavior depends on both timeliness execution of the real-time nodes and communication systems capability of transmitting messages while satisfying timeliness

requirements. Communication systems that can provide such services are known as ***real-time communication systems*** [10].

Equally to real-time systems, which must properly schedule the task set to satisfy the time requirements, communication networks must also apply scheduling techniques when multiple nodes use a shared medium to exchange messages. Its purpose is to control the access of each device to the network so that the overall system time specifications can be maintained. Similar to task scheduling, techniques used in communication networks can also be classified into: (i) soft, hard and non-real-time, when referring to their time constraints, or (ii) periodic, aperiodic and sporadic, according to their activation. Although this allows the use of scheduling methods applied to tasks, such as scheduling analysis, to be used in the network domain, there are several network-specific issues in distributed real-time systems, such as, e.g., the lack of preemption, that require suitable adaptations.

2.2.1 Communication paradigms

To initiate communications, i.e., message exchanging between network nodes, real-time communication systems can use two distinct approaches, ***time-triggering*** and ***event-triggering*** [10]:

- **Time-Triggered (TT):** In systems following the time-trigger paradigm, message transmissions within the network only occurs at specific temporal instants. Those moments are defined in a scheduling table that the network nodes must follow. Therefore, all the nodes must be synchronized and have a common time reference so that the triggering of messages happens at the defined instants. The scheduling table can be built prior to the system activation, i.e., offline, or online, while the system is operating.

- **Event-Triggered (ET):** Regarding systems that use the event-trigger paradigm, the transmission of application messages is triggered upon the occurrence of an asynchronous event, meaning that they can occur at arbitrary time instants, and cannot be controlled by the network. However, using the minimum inter-arrival time between consecutive events, it is possible to determine the upper bounds for the response time of these systems.

When comparing both paradigms, time-trigger communications are more suitable for synchronized systems, where they can provide low latency and jitter when handling messages. Regarding event-trigger strategies, due to the unpredictability of asynchronous traffic dispatching, these approaches are more fitting to handle sporadic messages with low latency. Nonetheless, upon receiving an overload of asynchronous messages, these procedures may suffer from high jitter and latency. Thus, to solve these issues, systems may ignore less important requests, only executing higher priority tasks, or switch to a time-trigger approach.

Several real-time systems use time and event-trigger approaches when handling both synchronous and asynchronous messages. The main advantages of employing both topologies are

the increase in flexibility on the overall system and costs reduction. However, the two distinct traffics must be temporally isolated so that the asynchronous nature of event-trigger messages can not interfere with the time-trigger traffic. This can be done by assigning bandwidth restricted sections to different traffic classes.

2.3 Ethernet

Ethernet is a communication technology for connecting devices, being the most commonly used to construct Local Area Networks (LAN). It was approved and released by the IEEE as IEEE 802.3 [11] in 1983, with CSMA/CD mechanisms. Carrier-Sense Multiple-Access with Collision Detection (CSMA/CD) is a collision-based protocol where multiple nodes can access the medium (Multiple Access - MA). However, each one must first confirm if the medium is idle before sending data (Carrier Sense - CS). Even with these measures, there still is the possibility that two distinct nodes start their transmissions at the same instant, thus resulting in a collision. This is detected as they keep sensing the medium even after sending the data, which stops the transmission (Collision Detection - CD). Both devices must then wait a random time interval and retry. The process is repeated a maximum number of times or until the transmission process is completed.

2.3.1 Ethernet Frame

The standard Ethernet IEEE 802.3 data frame is illustrated in Figure 2.2 [11].

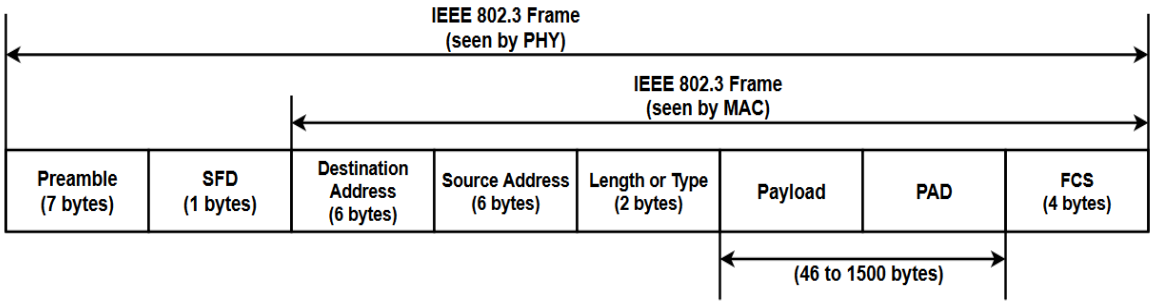


Figure 2.2: Standard Ethernet IEEE 802.3 Data frame.

The several fields present on the frame have the following functions:

- ***Preamble***: used to establish the synchronization of the receiver devices.
- ***Start of Frame Delimiter (SFD)***: sequence of 8 bits with the value "10101011". It indicates the termination of the preamble and the start of the Ethernet frame.
- ***Destination and Source Media Access Control (MAC) Addresses***: used to identify the destination and source network interfaces, respectively.

- ***Length or Type***: value between 0 to $2^{16} - 1$ that indicates the length of the payload or the protocol encapsulated in the data payload (e.g., 0x800 for IPv4 datagrams, 0x86DD for IPv6 datagrams).

- ***Payload***: field that contains the data to be sent/received. It can range from 46 to 1500 bytes.

- ***PAD***: field used for padding, whenever the minimum imposed size of 64 bytes for the frame is not reached.

- ***Frame Check Sequence (FCS)***: value generated by the transmission device, used for error detection by other nodes upon receiving the MAC frame.

2.3.2 Ethernet Switch

Ethernet switches emerged in 1990 as a mean to overcome the existing issues in Ethernet networks, specifically the lack of non-determinism from the CSMA/CD arbitrary methods, misusage of the accessible bandwidth, and absence of traffic isolation. A switch or bridge is an interconnection device that operates at the data link layer of the Open Systems Interconnection (OSI) reference model (second layer). Contrarily to Ethernet hubs, these devices have mechanisms that reduce the bandwidth wastage which result from broadcasting receiving frames to unnecessary network nodes. Instead, the switch forwards the data only through specific destination ports, thus increasing the overall throughput.

The general structure for the internal architecture of an Ethernet switch is illustrated in Figure 2.3. Upon the arrival of a new message, the frame is first buffered at the input ports, where its destination addresses are analyzed, and then moved towards the buffers of the appropriate destination port. However, if that output port is busy, the frame is stored in a queue for later transmission. Most current switches can handle the incoming traffic rate at the reception. Therefore, the input queues are generally smaller and do not build-up. The output queues, however, may rapidly fill-up if several frames arrive in short time intervals. As the frames are processed according to a First-Come First-Served (FCFS) scheduling policy, higher priority messages can be delayed by the transmission of lower priority ones. To overcome these issues, the IEEE 802.1D [12] standard was proposed.

Following the IEEE 802.1D standard, switches may have a limit of eight parallel queues for the distinct priority levels. It is important to highlight that, for these types of switches, the scheduling policy used to process received messages will strongly impact the timing behavior on the network.

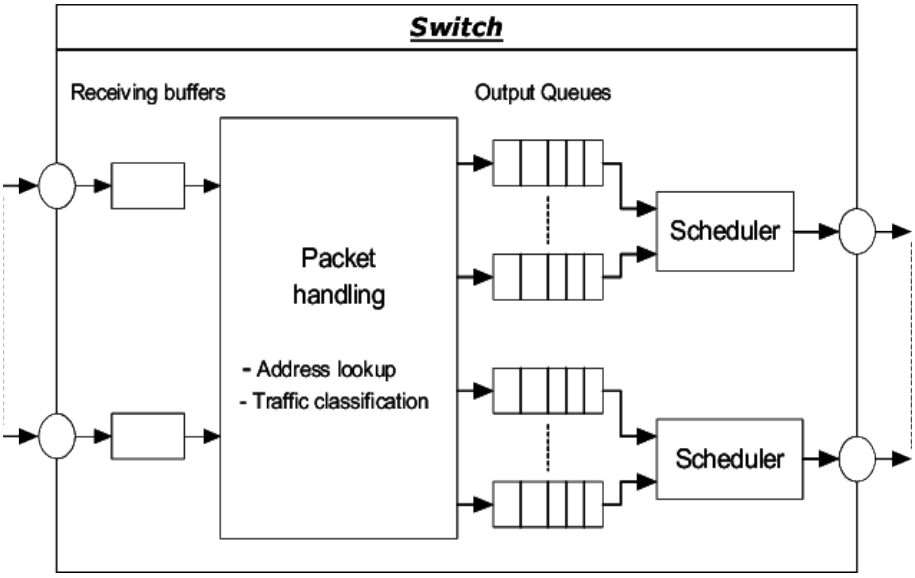


Figure 2.3: Internal structure of an Ethernet switch (*from R. Marau, L. Almeida and P. Pedreiras [13], 2006).*

Some of the Ethernet switches, following the IEEE 802.1D standard, had issues separating the incoming traffic into classes with different Quality-of-Service (QoS) requirements. To solve these problems, the original IEEE 802.3 data frame was extended to a set of standards that improve the traffic segregation. These included the IEEE 802.1Q for VLAN tagging and IEEE 802.1p and IEEE 802.1AC for priority identifier. A standard Ethernet frame with VLAN-tagging is illustrated in Figure 2.4 [14].

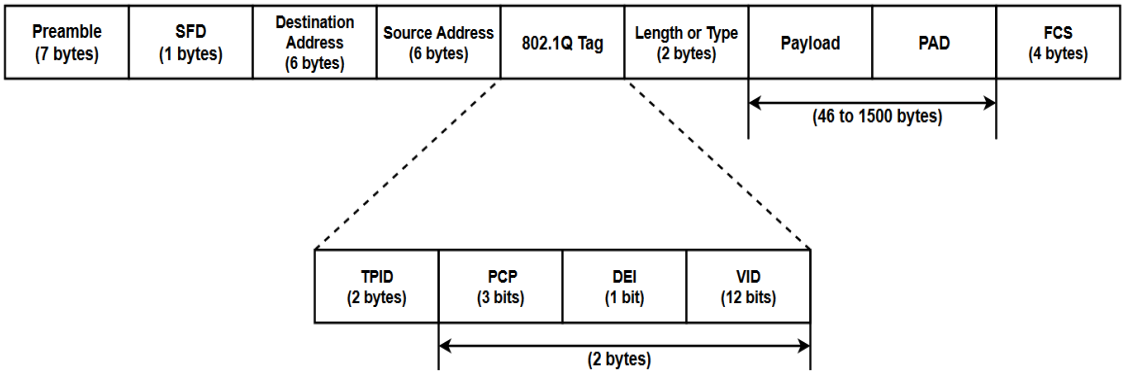


Figure 2.4: Standard IEEE 802.1Q Ethernet VLAN-tag frame.

The fields of the ***802.1Q*** tag have the following functions:

- ***Tag Protocol Identifier (TPID)***: 16-bit field with a fixed value of 0x8100 that identifies the frame as an Ethernet 802.1Q-tagged frame. It is located in the same position as the standard Ethernet Type/Length field for untagged frames.
- ***Priority Code Point (PCP)***: indicates the frame priority level, which can go from 0 to 7.

- ***Drop eligible indicator (DEI)***: 1-bit field that may be used in conjunction with PCP that indicates if a frame can be dropped in the event of an obstruction.

- ***VLAN Identifier (VID)***: specifies the VLAN to which the frame belongs. There are two reserved values: 0x000 and 0xFFF. The former indicates that the frame does not belong to a VLAN, and the 802.1Q tag only specifies its priority. The later is reserved for future implementation.

The forwarding mechanism for switches that do not support traffic segregation, i.e., do not use the IEEE 802.1Q VLAN tags, rely on MAC addresses. These devices have a forwarding table that maps the addresses of network nodes to the switch port, to which they are physically connected. The table can be static, with pre-configured addresses, or dynamically updated whenever a new frame arrives. However, if the MAC destination address of a given message is not registered on the forwarding table, the switch forwards the data to all the connected devices through broadcasting. Otherwise, frames are transmitted exclusively to their corresponding end-nodes. Ethernet switches support three types of forwarding addresses [14]:

- ***Unicast***: forwarding a frame from the source node directly to the destination device thought the association of the MAC address to the connected output port.

- ***Multicast***: forwarding a frame from a source device to a group of nodes. For this process, the IEEE 802.3 specifies distinct MAC addresses to define multicast groups. The forwarded frame is then received by all the nodes that belong to that group. Multicast protocols operate at the Network layer (layer 3) of the OSI model. Some Ethernet switches handle traffic at layer 3, supporting multicast addresses. However, pure layer 2 switches process multicasting as standard broadcasting.

- ***Broadcast***: implies the forwarding of a frame from the source device to the remaining nodes connected to the switch. The specific address 0xFFFFFFFFFFFF is set as the frame destination address. The switch recognizes it as a broadcasting address and forwards the frame to all output ports except the one that sent the frame.

Lastly, regarding switching methodologies, it is also important to distinguish the two different methods used:

- ***Store-and-forward***: The frame is firstly stored by the switch before being dispatched. This procedure allows the error verification inspecting the FCS field. Frames that pass the verification process are dispatched while others are discarded.

- ***Cut-through***: The frame is forwarded by the switch while being received. The switch starts the forwarding process after processing the MAC destination address or VID for VLAN-tagged frames.

Although the former method is more reliable, since it prevents error propagation on the overall network, the latter mechanism is generally faster, which decreases the switch latency.

2.4 Real-time protocols over Ethernet

As previously mentioned, standard Ethernet lacks the capabilities and determinism required for real-time networks. One of the reasons is the CSMA/CD mechanisms that may cause indefinite delays when collisions occur. Although this problem has been partially solved by the Ethernet switches development, these devices introduced other issues such as packet drops in overload situations and switching delays. In order to enforce real-time behavior in Ethernet switches, several solutions have been proposed. These can be categorized into two major groups: (i) relying on Commercial-Of-The-Shelf (COTS) Ethernet Switches; (ii) using customized Ethernet hardware.

2.4.1 Real-time protocols on COTS switches

The first group includes techniques as traffic shaping, master-slave protocols that can improve the efficiency of how the device handles incoming messages, e.g., QoS management, admission control, scheduling. These are used by different protocols such as *Flexible Time-Triggered Switched Ethernet (FTT-SE)* [13], and *ETHERNET Powerlink* [15].

2.4.1.1 Flexible Time Triggered - Switch Ethernet (FTT-SE)

The FTT-SE [13] is a COTS-based protocol to obtain real-time communications over switched Ethernet networks. It is an adaptation of the FTT-Ethernet [16] that follows the Flexible Time-Triggered (FTT) paradigm [17] and uses a master/multi-slave architecture. On the other hand, the FTT-SE master addresses multiple slaves with a single poll, thus reducing the protocol overhead. Since it follows the FTT paradigm, the communications are organized in fixed duration slots, known as Elementary Cycle (EC). Starts with the FTT master broadcasting a specific message, the Trigger Message (TM), which contains the periodic schedule for the current EC. The protocol supports both synchronous and asynchronous traffic. The first is scheduled and polled directly by the FTT master whereas the other is managed in the background, in the remaining EC time after handling the periodic traffic.

A standard architecture for systems based on the FTT-SE can be depicted in Figure 2.5. In Section 3.2, a more detailed explanation regarding the internal architecture of the FTT-SE and the management of synchronous and asynchronous traffic is presented.

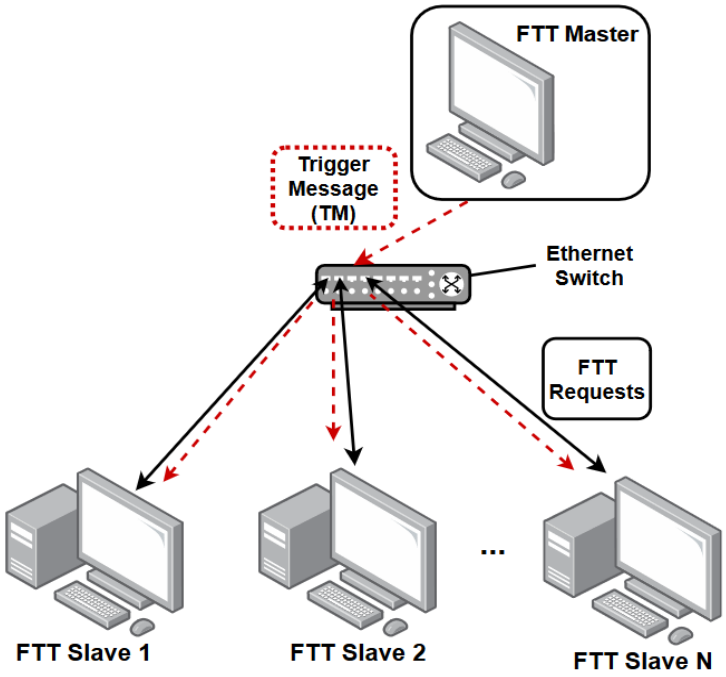


Figure 2.5: Typical FTT-SE system architecture.

2.4.1.2 Ethernet PowerLink (EPL)

ETHERNET PowerLink [15] is a real-time protocol based on standard IEEE 802.3 Ethernet that allows deterministic data transmission. The EPL has a master-slave control mechanism operating at the data link layer of the OSI model. This mechanism allows the master controlling the medium access through explicit messages. Besides preventing collisions, this technique also grants high determinism to the network, as non-deterministic methods used in the standard Ethernet such as CSMA/CD, are not activated.

The communications are divided into a sequence of cycles. Each has phases (time intervals) where the periodic (isochronous) and aperiodic (asynchronous) traffic is processed. In the EPL protocol, the master is known as Managing node. This node controls the communications by sending message requests at specific instants to a particular node. The slaves (Controlled nodes) react to those messages by sending a response containing the data. As such, it is the managing node that schedules both periodic and aperiodic traffic by triggering each of them every cycle phase. A cycle of the EPL protocol can be divided into four distinct phases (Figure 2.6):

- ***Start Phase***: All nodes are synchronized to the master node's clock. This is achieved by transmitting, at the beginning of each cycle, a specific message designated *Start of Cycle* (SoC).

- ***Isochronous Phase***: The managing node triggers the periodic traffic and assigns a time slot for each node to transfer their critical data by sending a poll request frame.

Addressed nodes answer with a poll response. As the data transmitted is broadcasted, the communications are based on a producer-consumer model.

- ***Asynchronous Phase***: The managing node allows a single particular node to send an aperiodic message. The master sends a Start of Asynchronous frame (SoA), and the replying node answers with a message containing the data. Standard IP-based protocols such as TCP/IP can be used during this phase.
- ***Idle Phase***: Time interval without communications. Provides low jitter between consecutive cycles.

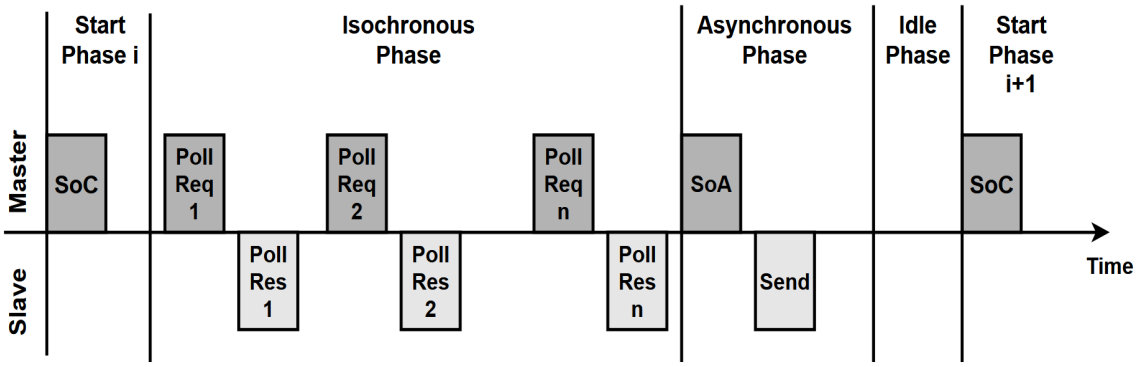


Figure 2.6: ETHERNET PowerLink communication cycle (*based from [2]*).

2.4.2 Real-time protocols on customized hardware

The use of COTS hardware includes a major limitation: only nodes that comply with the associated protocols can be integrated into the network. This implies that: (i) standard Ethernet nodes cannot be used since they can jeopardize the real-time services, and (ii) malfunctioning nodes can risk the system timeliness. Both problems can be solved by introducing management and control mechanisms in the Ethernet switch. This way, it is still possible to use COTS hardware and software, while real-time services can be integrated into specific layers.

2.4.2.1 Profinet-IRT

PROFINET is an industrial Ethernet standard developed by PROFIBUS & PROFINET International. Real-time communications can be achieved by PROFINET IO or PROFINET Component Based Automation (CBA) protocols. PROFINET CBA operates the component-based communications using TCPI/IP for data exchanging between machines and real-time (RT) communications to achieve the time requirements. PROFINET IO uses exclusively real-time and isochronous real-time communications (IRT) for distributed I/O designed for fast data exchange.

PROFINET-IRT [18] is part of the PROFINET protocol for high deterministic networks and fast cycle times, with values reaching 250 μs. To achieve strict and tight deadlines, PROFINET

introduced the Enhanced Real-Time Ethernet Controller (ERTEC) for isochronous Ethernet communications. The principle behind PROFINET-IRT consists of splitting each cycle into phases, in which a specific type of traffic is sent (Figure 2.7).

One phase is for IRT traffic, i.e., all the non-IRT messages are buffered and only IRT frames are transmitted. The other is used for both real-time and non-real-time traffic for standard address-based Ethernet communications. Although this method provides traffic isolation, it also requires efficient planning to obtain the communication schedule. The scheduler, responsible for this procedure, plans the IRT time slots. The remaining time slots will depend on the number of frames sent in each phase.

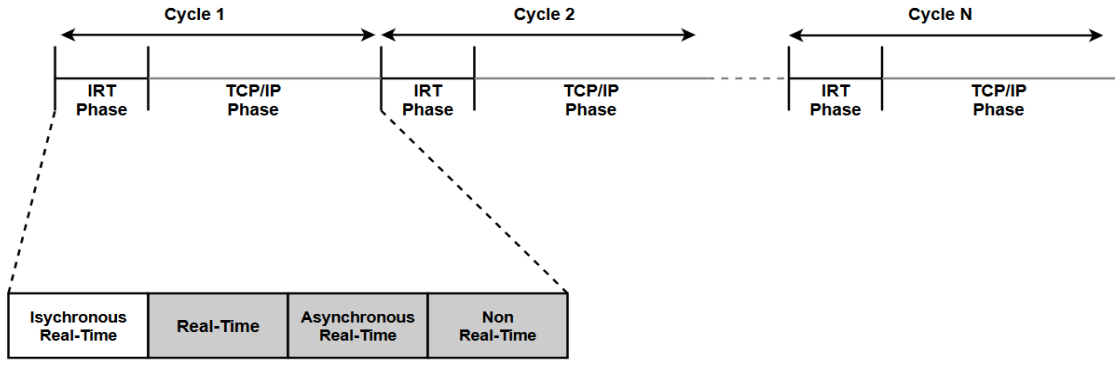


Figure 2.7: PROFINET-IRT communication cycle (*based from [19]*).

Each schedule is calculated for a system specification and includes information regarding network topology, producer's and consumer's data amount, and connection characteristics (cable length, type of medium, cable delay, etc.). As a result, whenever there is a change in the system, a new schedule must be created.

2.4.2.2 Avionics full duplex switched Ethernet (ADFX)

Avionics Full Duplex Switched Ethernet (AFDX) [20] is a network communication specification based on the IEEE 802.3 and ARINC 664, part 7 standards. The latter one defines the electrical and protocol specifications for high deterministic data exchange between avionic subsystems. As depicted in Figure 2.8, an AFDX network is formed by Avionic Subsystems and AFDX switches. The subsystems, e.g., flight computer control and global positioning system contains an *AFDX End system* that provides an interface to transmit data via Ethernet frames between the network nodes. Furthermore, AFDX networks implement two redundant networks using two independent switches to increase the system robustness. As such, every Avionic Subsystem has two pairs of Ethernet ports to send and receive redundant frames from both switches.

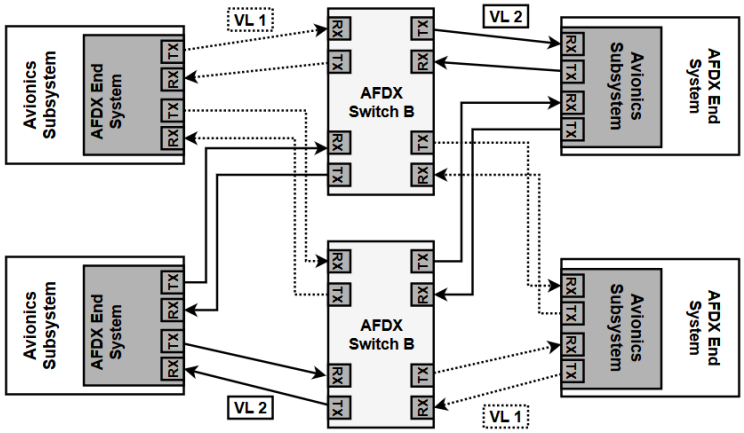


Figure 2.8: Typical AFDX architecture (*based from [2]*).

The central feature of AFDX networks is their communication channels, virtual links (VL). These channels allow communications between one source and multiple destination end-systems through unidirectional logical connections. Each virtual link has a dedicated bandwidth, controlled by traffic shapers at the end systems, which impose a minimum time interval between consecutive messages defined as bandwidth allocation gap (BAG).

Regarding standard switches that route incoming frames according to Ethernet MAC addresses, AFDX network messages are routed according to a 16-bit value named Virtual Link ID. Moreover, AFDX switches also introduced frame filtering and traffic policing to control VL requirements and prevent the interference of misbehavior subsystems.

CHAPTER 3

An FTT-Enabled Switch - HaRTES

Contents

As discussed in the previous chapter, DRTS are highly dependent on the communication subsystem capability to accurately exchange messages between nodes without jeopardizing the system timeliness. As time passed, the DRTS requirements evolved, with higher flexibility demands to support online modifications to the system configuration, which were solved with the creation of the FTT paradigm. As such, this chapter starts by introducing this protocol, followed by the description of an Ethernet-based switched technology to which it was applied, the FTT-SE. Lastly, the chapter closes with the presentation of HaRTES, a modified Ethernet switch based on the FTT paradigm.

3.1 The Flexible Time Triggered paradigm

The Flexible Time Triggered paradigm [17] is a communication model that enables DRTS to exchange real-time messages while maintaining both flexibility and timeliness guarantees. Contrarily to other communication protocols, the FTT paradigm supports online communication changes to the message set that arrive, e.g., from dynamic QoS management. This paradigm operates at several layers in the OSI model, however it requires an existing network protocol, implemented on the physical layer, to be deployed.

The FTT paradigm is based on a master-slave architecture. The master node, responsible for handling the communications, contains all the information regarding traffic requirements,

scheduling policy, QoS management, and admission control. To inform the slaves about the scheduling arrangements, the FTT master uses a master/multi-slave transmission control where it periodically broadcasts a specific message named *trigger message (TM)* with the information about the current schedule. The slave nodes, on the other hand, receive the TM and decode them in order to verify whether they are or are not producers of any scheduled message. This procedure has two consequences: (i) the system overhead decreases as the traffic triggering can be done using a single TM, and (ii) the whole network timeliness will depend on the master node timeliness.

There are several instances of the FTT paradigm being applied to different network technologies, such as the FTT-CAN, that operates in Controller Area Network (CAN), the FTT-Ethernet, employed in bus network topologies, and based on shared Ethernet, the FTT-SE and HaRTES, both based on COTS switched Ethernet and applied to star network topologies. However, the latter is a custom Ethernet switch, whereas the former utilizes legacy switches. Nonetheless, despite being implemented in different technologies, the following FTT properties are still present:

- High flexibility for handling synchronous traffic.
- Online admission control for real-time traffic.
- Support dynamic modifications for both the traffic properties and scheduling policies.
- Support different classes of traffic (synchronous and asynchronous real-time and non-real-time traffic) with temporal isolation.

3.1.1 FTT Elementary cycle

In the FTT paradigm, the communications are organized in a sequence of time-slots with a fixed duration designated *Elementary Cycles (EC)*. As depicted in Figure 3.1, each EC begins with the transmission of a TM by the FTT master. It synchronizes the slave nodes and informs them about the cycle scheduled messages. The remainder of each EC is divided into two distinct windows, Synchronous Window (SW) and Asynchronous Window (AW), that handle the synchronous and asynchronous traffic, respectively.

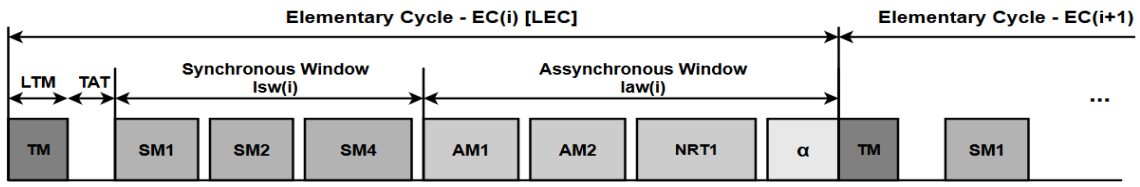


Figure 3.1: FTT paradigm Elementary Cycle structure (*based from [2]*).

The duration of each elementary cycle can be configured, and it is defined as *LEC* time units. This interval specifies the communication resolution since all traffic characteristics, i.e., deadlines, periods, phases, are multiples of this parameter. Within each EC, it is possible to distinguish four intervals that, overall, define each cycle length:

- ***LTM***: time interval that includes the broadcasting of a TM by the FTT master plus a guard window for propagation differences between nodes.
- ***TAT (turn-around time)***: period required by the FTT slaves to process and decode the TM.
- ***lsw(i)***: asynchronous window time length. It is a variable parameter that depends on the number of synchronous messages scheduled in the current EC (EC(i)). However, it is possible to define a maximum value for this window (LSW), thus always guaranteeing minimum bandwidth for handling the asynchronous traffic.
- ***law(i)***: duration of the asynchronous window. It can be calculated as the current EC remaining time:

$$law(i) = LEC - LTM - TAT - lsw(i) \tag{3.1}$$

  Equation 3.1 can be reformulated using the the synchronous window maximum size, thus also guaranteeing a minimum length for this window:

$$LAW_{min} = LEC - LTM - TAT - LSW \tag{3.2}$$

There is also temporal isolation within each EC that prevents the asynchronous traffic from interfering with the synchronous one. This is done by guaranteeing that transmissions only begin if they are completed within each EC window. Lastly, an idle internal (α), may also be used at the end of the asynchronous window, before the next EC, whenever a message does not fit in the window, and the cycle must be delayed until the following one.

3.2 Flexible Time Triggered Switch Ethernet

The FTT-SE [13] is a real-time communication protocol for COTS Ethernet switches based on the FTT paradigm. It was designed for master/multi-slave network architectures where the master node (FTT master) is responsible for the whole network traffic management. As such, each slave node transmits its communication requirements to the master, which then creates a schedule and broadcasts it using Trigger Messages. This schedule is based on the FTT paradigm Elementary Cycles, which provides separate time windows for synchronous and asynchronous transmissions.

Slave nodes transmit FTT packets through communication channels designated streams. Each stream is characterized by a specific traffic class (synchronous or asynchronous) and has specific time requirements. Furthermore, a single stream can have multiple subscribers. For messages to be scheduled, FTT slaves must register their streams in FTT master. Therefore, the master node is responsible for managing the streams, whereas, by sending particular control requests to the master, the slaves, handle their operations (e.g., creation, modification). With the communication requirements registered, the master can then broadcast the TM and, after decoding it, slaves transmit their produced packets if instructed. This mechanism

provides control of FTT communications to the master, making it responsible for maintaining the timeliness and the system integrity.

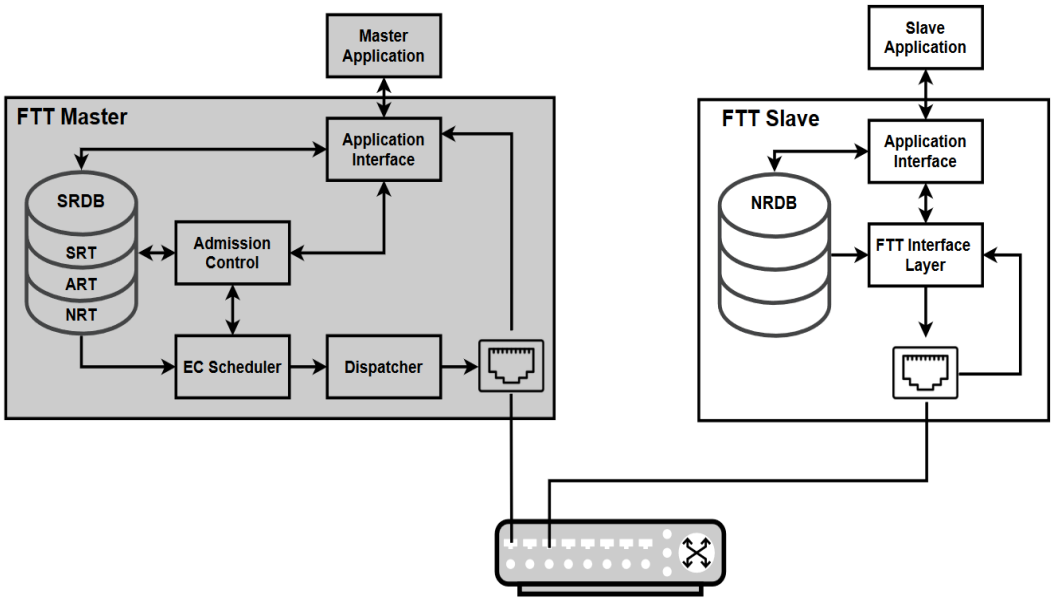


Figure 3.2: FTT-SE master/slave architecture (*based from [20]*).

3.2.1 FTT-SE master architecture

The internal architecture of the FTT master, illustrated in Figure 3.2 by the shaded area, contains five essential structures.

- ***Application Interface:*** allows access to services for system management. While the FTT master can access these locally with the appropriate software, slave nodes need to transmit specific asynchronous control messages created by their applications. Accessible services with this interface include: system configuration (e.g., set the EC duration), message management and information acquisition (e.g., jitter or latency values).

- ***System Requirements Database (SRDB):*** a central repository that stores all the information regarding the messages' characteristics (real and non-real-time) and parameters for the system configuration. Within the SRDB it is possible to distinguish three different tables: (i) Synchronous Requirements Table (SRT), (ii) Asynchronous Requirements Table (ART), and (iii) Non-Real-Time Requirements Table (NRT) for the three distinct traffic classes.

  The **SRT** contains the properties of N_s synchronous messages processed in the system.

$$SRT \equiv \{SM_i(DLC_i, C_i, Ph_i, P_i, D_i, Pr_i, *Xf_i), i = 1..N_s\} \tag{3.3}$$

  where SM_i represents a message, DLC_i its data length in bytes, C_i the transmission time, including overheads, Ph_i the initial phase (if defined), P_i the period for periodic

messages or Minimum Interarrival Time (mit) for aperiodic, D_i the deadline, Pr_i the messages' fixed priority and lastly $*Xf_i$ is a specific data structure defined in the SRT to enhance the system functionalities.

The **ART** contains the properties of N_a asynchronous messages including both, the ones with and without timeliness constraints:

$$ART \equiv \{AM_i(DLC_i, C_i, mit_i, D_i, Pr_i), i = 1..N_a\} \tag{3.4}$$

where AM_i is used to represent each message. The rest of the table is similar to 3.3 except for the absence of both the initial phase and the additional data structure. The *Period* parameter is replaced by the minimum inter-arrival time (*mit*).

Lastly, the non-real-time traffic, which is handled with a best-effort policy, has no timeliness constraints. Therefore, the FTT master only needs to know the producer streams and the size of their largest message. The **NRT** contains these properties for N_n streams that produce non-real-time messages.

$$NRT \equiv \{NM_i(SID_i, MAX\_DLC_i, MAX\_C_i, Pr_i), i = 1..N_a\} \tag{3.5}$$

where each message is represented with NM_i. The SID_i property identifies the message sender node, MAX_DLC_i the data length of the largest transmitted message in bytes, MAX_C_i the maximum transmission time, and Pr_i the node non-real-time priority.

- ***EC Scheduler:*** operates online and, according to the information stored in the SRDB, creates an EC-based schedule according to a specific algorithm (EDF, RM, and DM currently supported). It uses the content from both synchronous and asynchronous real-time tables as well as the system configuration to determine which synchronous messages should be transmitted.

- ***Admission Control:*** invoked whenever the real-time message set is altered. Uses different schedulability tests based on the scheduling algorithm employed and the SRDB content to assess the timeliness constraints of real-time traffic.

- ***Dispatcher:*** builds the trigger message with the created EC-schedule. It is posteriorly broadcasted to every device connected to the switch.

3.2.2 FTT-SE slave architecture

The FTT slaves, also known as stations, have a simpler architecture compared to the FTT master, having three relevant structures, as illustrated in Figure 3.2.

- ***Application Interface***: structure with similar functionalities as the one presented in the FTT master. As such, it provides a set of services for traffic management.

Regarding the real-time traffic, which is triggered by the master node, applications may request modifications to the stream, i.e., filter the produced and received messages or set call-back for specific events (e.g., occurrence of a missed deadline). Lastly, the asynchronous traffic is generated by the slave application and must be signalled to the master so it can be triggered and posteriorly processed.

- ***Node Requirements Database (NRDB)***: central repository identical to the SRDB for slave nodes. It contains information regarding the streams' properties and requirements.

- ***FTT Interface Layer***: responsible for receiving and decoding the TM and transmit the node messages according to the EC schedule. Upon receiving a new frame, this interface scans the NRDB and checks if the message ID belongs to the set that should be locally received and processed. If so, the frame is stored in a queue with a FCFS priority policy otherwise, it is discarded.

3.2.3 FTT-SE communication model

Regarding the scheduling of **synchronous traffic**, the FTT-SE master node uses an online scheduler that may consider the messages' priorities, either dynamic or static, when creating the schedule. The FTT-SE synchronous scheduling model can be represented by a set of N_s periodic streams (SM_i) stored in the SRT.

$$STR = \{SM_i : SM_i(C_i, D_i, T_i, O_i, S_i, \{R_i^1 ... R_i^{ki})\}, i = 1...N_s\} \tag{3.6}$$

where is C_i each stream message transmission time, D_i the stream deadline, T_i its period and O_i the offset, being the latter three parameters expressed in integer multiples of ECs. Lastly, S_i represents the sender node and R_i the stream receiving nodes. It is important to highlight that large messages are fragmented and sent as a set of packets sequentially scheduled by the FTT master.

The EC-scheduler uses the information present in the SRT, builds the schedule, and encodes it in trigger messages, which are posteriorly broadcasted by the Ethernet switch. The slaves receive and process them and, afterward, transmit their scheduled packets to the switch, which queues them locally until they are forwarded. The FTT master also holds information regarding the type of address of a given message: unicast, multicast[1], and broadcast. As such, it can build specific schedules that improve the throughput of the system by parallelizing the communications.

One significant downfall of this protocol is that it depends on the FTT slaves to have specific network drives (FTT compliant) to communicate following the protocol rules, otherwise the system timeliness is jeopardized. As these drives are entities that transmit and receive packets,

[1]Some switches do not support multicast addressing and, in those cases, multicast is treated as standard broadcast, as explained in Section 2.3

they allow nodes to register their streams in the FTT master, decode trigger messages and provide isolation between the local software application and the FTT structure.

When it comes to the **asynchronous traffic**, this protocol implements a time-triggered approach similar to the one used for the synchronous type. If not properly constrained, asynchronous messages could jeopardize the FTT communication paradigm, as their transmissions could be made outside the appropriate window. To prevent such situations, the FTT-SE uses the master node to control transmissions of both synchronous and asynchronous messages. This is accomplished by the master through the use of trigger messages, which poll asynchronous traffic so it can be transmitted in the appropriate time window. This type of traffic can be modelled by a set of N_a aperiodic streams (AM_i) stored in the ART:

$$ART = \{AM_i : AM_i(C_i, D_i, Tmit_i, S_i, \{R_i^1 ... R_i^{ki}\}), i = 1...N_a\} \tag{3.7}$$

Comparatively to synchronous stream structure (3.6), this model only differs in the absence of the offset parameter and the messages period is replaced by the inter-arrival time $Tmit_i$. Concerning to the non-real-time traffic, this class is treated as asynchronous traffic but only handled in the background with a best-effort policy within the asynchronous window.

The technique used by the FTT master to poll asynchronous traffic as it does for the synchronous type is advantageous when it comes to controlling access to the communications medium. However, the poll of sporadic messages does not always result in transmission. Consequently, this method can be very inefficient bandwidth efficiency terms. A signaling mechanism for master/multi-slave architectures with full-duplex switches was proposed to solve such problem. This scheme takes advantage of the parallelism available for full-duplex structures to send the status of slaves' asynchronous queues while the TM is being broadcasted. Readers are referred to [21] for more detailed information.

Lastly, significant measures have been implemented in regard to traffic isolation and asynchronous traffic signalling for master-slave Ethernet protocols. In order to ensure that asynchronous messages do not overlap the synchronous window of the next elementary cycle, FTT master only allows their transmissions if they finish within the respective EC window, thus ensuring empty queues at the switch ports for broadcasting the next TM.

3.3 Hard Real Time Ethernet Switch

HaRTES [2] is an Ethernet switch based on the FTT paradigm that provides real-time communication services. It was developed as a means to overcome the FTT-SE limitations, in particular, the constraint that all slave nodes need to be FTT-compliant, i.e., they require a specific network device driver to respect the EC-schedules and its respective timings. Since these drivers may not be available to different operating systems, several network nodes

will not respect the protocol schedules, resulting in failures on the timeliness constraints. To overcome this problem, the FTT master and Ethernet switch, which were originally separated on the FTT-SE protocol (Figure 2.5), were combined in a single device (Figure 3.3). Therefore HaRTES maintains a master-slave architecture for the network and applies the flexible time-triggered paradigm for the communications.

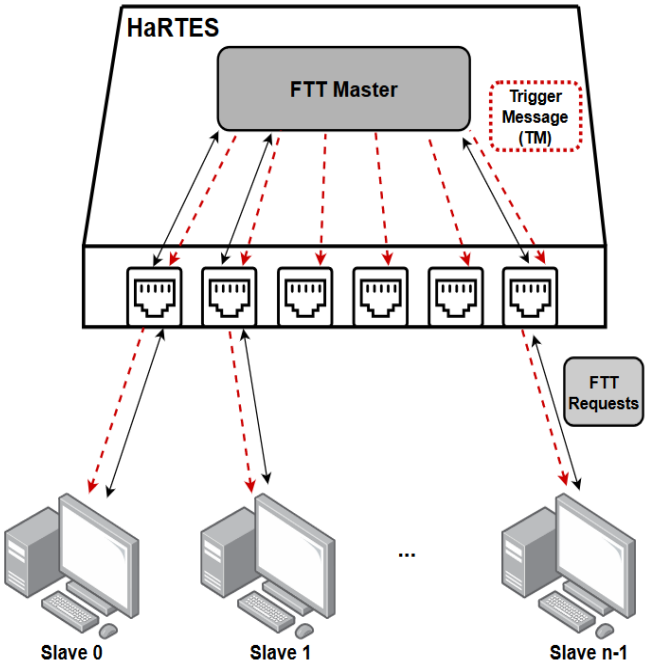


Figure 3.3: HaRTES architecture.

The integration of the FTT master within the Ethernet switch provides traffic confinement, maintains the most significant FTT-SE characteristics and improves the following aspects:

- The asynchronous traffic is now autonomously triggered by network nodes instead of being polled by the master, thus simplifying the handling of this traffic class.
- Blocking at the switch input ports unauthorized real-time transmissions preventing them from interfering with the rest of the system, improving the system integrity, and, simultaneously, the integration of non-FTT-compliant nodes.
- Diminishes the jitter and latency of the TM transmissions, enhancing the overall network synchronization.

3.3.1 HaRTES internal architecture

The general architecture of the previously explained Ethernet switch is illustrated in Figure 3.4. This structure can be divided into four functional sections: **FTT master module**, **Input ports module**, **Output ports module**, **Memory Pool**.

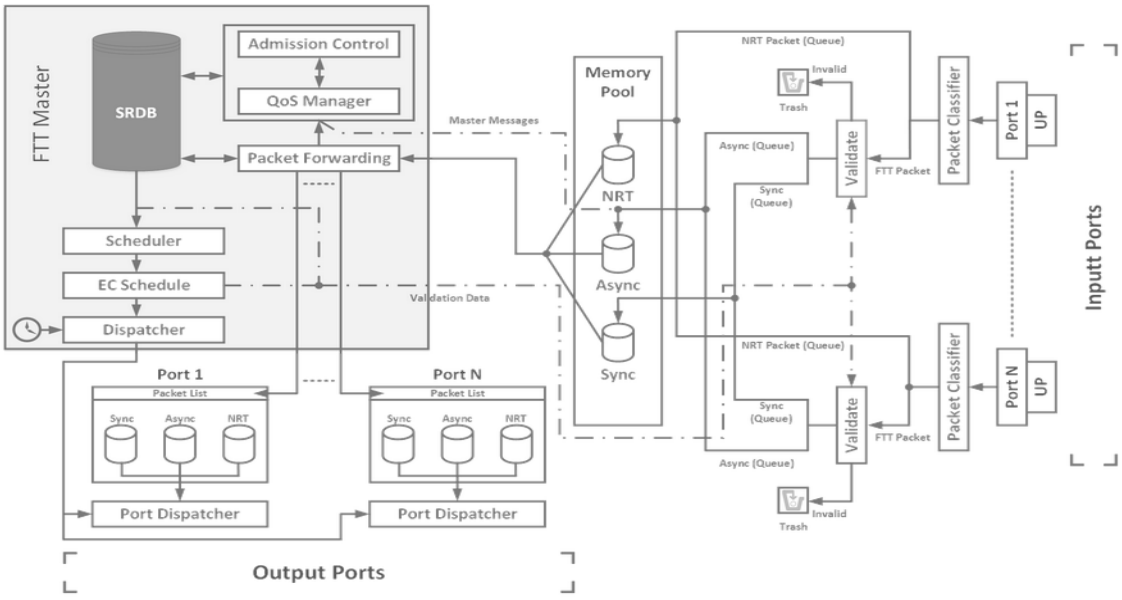


Figure 3.4: HaRTES internal architecture (*from [22]*).

3.3.1.1 FTT master module

The FTT master module, represented in Figure 3.4 by the shaded area, is responsible for the overall management and decision making logic of the switch. It can be subdivided into different blocks that perform specific tasks, namely Admission Control, QoS manager, Scheduler, Packet Forwarding and Dispatcher. The SRDB is the central repository for all information regarding traffic management, just as in the FTT-SE protocol. It contains properties that can be divided into three different groups:

- Message attributes for synchronous and asynchronous traffic: transmission times, periods/minimum inter-arrival times, offsets, sender nodes, priorities, deadlines.
- Global configuration information: elementary cycle duration, maximum synchronous window duration, time required to transmit the TM to all nodes, turn-around nodes time, asynchronous window duration.
- Resources information allocated to each traffic class: phases duration, buffer memory.

In the event of occurring a modification in the message set, the Admission Control, together with the QoS Manager and the SRDB, assert the timing guarantees of ongoing communications. The Scheduler is responsible for periodically scan the SRDB and build a list of synchronous messages to be transmitted in the following EC. This new EC-schedule is broadcasted within the TM by the dispatcher.

3.3.1.2 Input ports module

To integrate both FTT and non-FTT compliant nodes in the network, the HaRTES first operation, at the input ports module, is to classify packets into synchronous, asynchronous

and non-real time. This operation consists of inspecting the packets' header and stored them in the dedicated memory pool queues. Non-real time packets are appended to the non-real time queue, while real-time messages are first subjected to a validation process. If valid, these are stored in the designated memory pool (synchronous or asynchronous), otherwise they are discarded. Lastly, FTT request packets targeted to the FTT master are handled by the Admission Control and QoS manager. If the request is not feasible, the system keeps unchanged. Otherwise, the SRDB is modified with the processed request content.

The synchronous packets validation process consists of analyzing the EC-schedule and detecting if any failures occur. For asynchronous messages, this process involves examining the inter-arrival time and size. Whenever a message is stored in the memory pool module, specific pointers that point to the data packet are generated and forwarded to the target output ports. Regarding non-real-time packets, the forward mechanism employed is the standard Ethernet MAC address procedure. On the other hand, for FTT real-time traffic, the switch employs a producer-consumer model. Therefore, when a new FTT message is received, the *Packet forwarding* module inquires the SRDB to determine which ports have consumers attached and updates the output queues of the designated ports.

3.3.1.3 Output ports module

Each output port is composed by three pointer queues, one for each traffic type, and a dispatcher. Each queue stores pointers for packets stored in the memory pool which will be transmitted in that port. Connected to these queues is the dispatcher, the responsible module for packet transmissions. It is in charge for keeping the temporal information about the elementary cycle and, throughout each cycle, send packets pointed by each queue in the appropriate EC phase.

3.3.1.4 Memory Pool

Validated packets are stored in designated memory queues, keeping each traffic class independent and, at the same time, avoiding memory exhaustion for real-time packets. Since real-time traffic is subject to a registration process, it is possible to pre-allocate the required memory amount to guarantee enough resources to them all.

3.3.2 HaRTES communication description

As a result of being a derivative of FTT-SE, HaRTES shares many of the characteristics presented in the FTT paradigm. The HaRTES master node organizes the communications by transmitting successive ECs divided into two windows: synchronous and asynchronous (Figure 3.5).

The synchronous traffic is scheduled and triggered by the FTT master by broadcasting, at the beginning of each EC, the trigger messages containing the current EC-schedule. The FTT-compliant nodes receive and decode the TMs and transmit their scheduled packets. The

scheduling of the synchronous traffic is made online, based on a specific scheduling policy such as RM or EDF.

Regarding the asynchronous traffic, the FTT master applies server-based scheduling techniques [23]. Furthermore, this traffic type is triggered by the slave nodes and handled by the master without interfering with the synchronous class. Upon receiving the slave messages, the switch stores them in dedicated memory pools and transmits them when suitable.

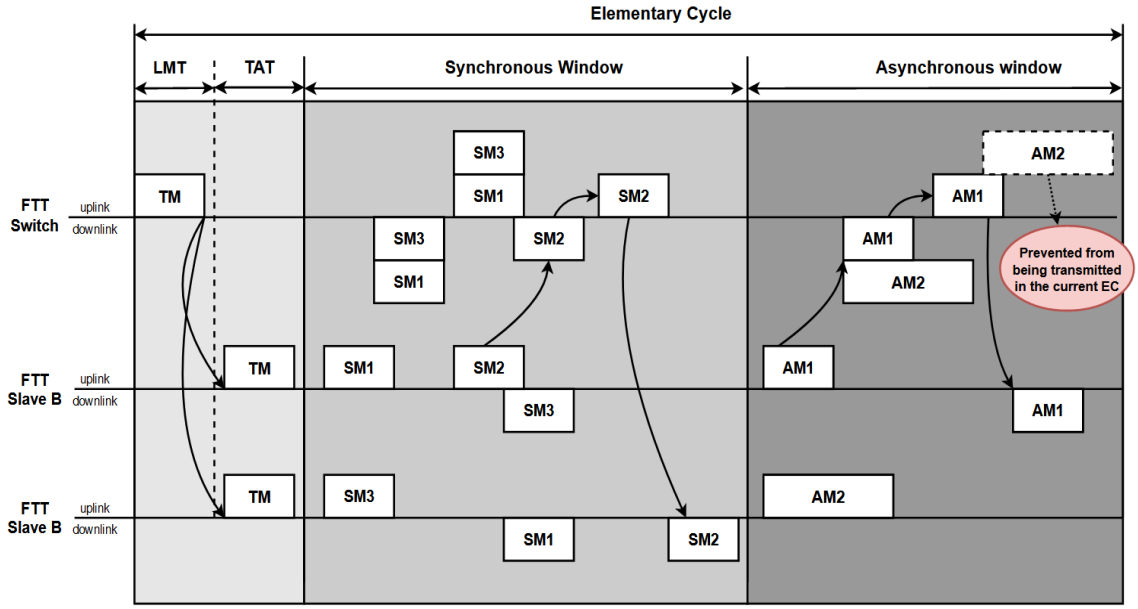


Figure 3.5: HaRTES communication model (*based from [2]*).

3.3.2.1 HaRTES synchronous subsystem

Compared to FTT-SE, HaRTES has significant gains when handling synchronous traffic. First, as TMs are generated within the switch, synchronous messages can be polled directly by the FTT master. Consequently, the aggregation of the FTT master and Ethernet switch is transparent to the rest of the nodes. When compared to the FTT-SE protocol in which the TM must go through the protocol stack before reaching the switch itself, HaRTES TM transmissions have less jitter and latency. The other improvement is the capability of blocking unauthorized real-time communications, both synchronous and asynchronous, at the switch input ports, thus not interfering with the remainder system. However, if the slave nodes require a message to be transmitted synchronously, they still need the FTT compliant driver to perform synchronous hard-real time communications. Without the drivers, TMs sent by the FTT master cannot be processed.

3.3.2.2 HaRTES asynchronous subsystem

Regarding the asynchronous traffic, HaRTES provides significant simplifications. Compared to the FTT-SE, HaRTES does not require this specific traffic to be polled, it is triggered by the slave nodes and queued inside the switch in dedicated memory when received. To

transmit the asynchronous traffic, HaRTES applies hierarchical, server-based mechanisms. Therefore, producer nodes must perform an explicit registration in which they declare the stream properties (e.g., minimum inter-arrival time, transmission time). These allow the switch to allocate the necessary resources and assign a server to each stream.

Both asynchronous and non-real-time traffic is transmitted in the asynchronous windows. The former is sent until the server depletes its capacity. On the other hand, the latter type is processed solely when there are no queued asynchronous messages or when all the streams associated servers are budget depleted.

3.3.3 Advances over the HaRTES implementation

To enhance the properties of HaRTES, several solutions have been proposed. This section focuses on implementations made to improve traffic management in the switch. The first one focuses on total order broadcast and multicast mechanisms for synchronous messages [24] while the other addresses hierarchical server-based scheduling for asynchronous traffic [25].

3.3.3.1 Real-Time and Consistent Multicast in *HaRTES*

Total order broadcast is a specific type of broadcasting in which a set of processes receive the same sequence of messages, i.e., if a correct process receives two distinct messages, n followed by m, then all the remaining correct processes must receive n before m. It is associated with fault-tolerance mechanisms, which assume that almost no failures occur in the system. Several works have been made to combine the FTT-paradigm with such mechanisms [26], [27].

Total order broadcast is accomplished if the following three properties are fulfilled:

- ***Agreement***: if a participant process receives a message n, then all other participants will eventually receive that message.
- ***Integrity***: for a message n, every participant process receives n just once, and only if n was previously broadcasted.
- ***Validity***: if a participant process correctly broadcasts a message n, then all the participants will receive n.

In order to simplify the implementation of the protocol in HaRTES, the authors considered three fault tolerance assumptions proposed in the scope of the Flexible-Time-Triggered Replicated Star (FTTRS) [28], a mechanism for TM replication in star topology networks: (i) the device has services to prevent error propagation, e.g., transmissions occurring out of the appropriate windows, (ii) the switch successfully transmits consecutive TM without errors, and (iii) the probability of failure in HaRTES is negligible. These assumptions, together with HaRTES mechanisms, specifically a property in which an EC-schedule implicitly provides the order of synchronous messages, lead to the following conclusion: if the slaves process the delivered messages in the same order as the one administered by the EC-schedule, in an EC,

Agreement implies total order.Therefore, as long as the nodes receive the same messages, the HaRTES scheduler guarantees the order in which they are delivered is always identical.

To achieve the first two properties, the proposed protocol associates sections of the EC structure, between the transmission of two consecutive TM, into four phases, as illustrated in Figure 3.6. Figure 3.7 shows the behavior of the network components during each phase.

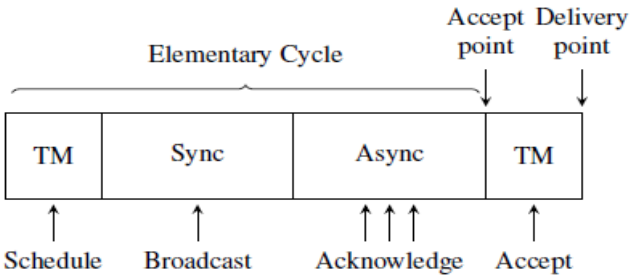


Figure 3.6: Distinct four phases of the proposed protocol over the FTT elementary cycle (*from Guillermo Rodriguez-Navas, Julián Proenza [24], 2013).*

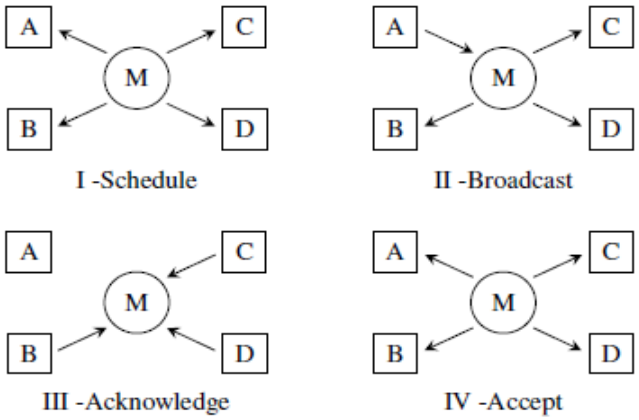


Figure 3.7: Network behavior throughout the different four phases of the protocol (*from Guillermo Rodriguez-Navas, Julián Proenza [24], 2013).*

The first phase, i.e., Schedule Phase, is similar to the standard FTT paradigm, in which the master node broadcasts the TM with the EC schedule. Upon decoding the message, the slaves are informed about the producer streams of that cycle. In the following phase, the Broadcast phase, all the producers broadcast their scheduled messages within the synchronous window. In the third phase, the Acknowledge phase, each subscriber of the broadcasted messages informs the master about the transmissions outcome. They send, through the asynchronous window, a positive notification (ACK) if that message was received or a negative notification (NAK) if otherwise.

The end of the asynchronous window, also named as Accept point, marks the initiation of the last phase, the Accept. During this phase, the master uses the information received

from the subscribers to decide whether the broadcasted message is accepted. Based on the SRDB content, the master knows how many ACK must be transmitted for a given message. Consequently, if the expected number is not reached the message is aborted, otherwise it can be processed. At the Accept point, the master also builds a specific vector, designated EC-status vector (EC-SV), which indicates what messages are accepted and can be delivered. This vector has similar functionalities as an EC-schedule, and it is also transmitted in the trigger message. Upon receiving and decoding the TM, the subscribers process the messages. This instant is designated as Delivery Point.

To attain *Validity*, the authors present two different alternatives. The first consists of implementing an automatic re-scheduling algorithm identical to the one used in the FTT-CAN to re-transmit erroneous messages by the master. Although it can be easily implemented, this method can delay the messages retransmission for a full EC duration as the last phase of this protocol is located at the end of each cycle and there is no time to recreate a new schedule. In the second method, the master reserves some bandwidth specifically for retransmissions. Assuming that the number of aborted messages is low, the master reserves bandwidth in the EC for any retransmission required, which are scheduled in the EC-SV of the broadcasted TMs. If no retransmissions occur, the slaves use the unused bandwidth during the asynchronous window. Compared to the first approach, this method has higher complexity but does not delay retransmissions.

3.3.3.2 Hierarchical served-based traffic scheduling over *HaRTES*

As explained in the previous chapter, hierarchical scheduling is used to simplify complex systems scheduling by fragmenting them into multiple subsystems, which can then be managed individually. When applied to complex DRTS, together with server-based architectures, these structures provide composability, i.e., a parent node schedulability depends only on its requirements as well as its children requirements. The following section focuses on a multi-level hierarchy architecture designed to handle asynchronous traffic in Ethernet switches, that was applied to *HaRTES*. It also describes a schedulability analysis based on the response time of such structures for Ethernet switches.

The proposed structure, depicted in figure 3.8, illustrates a tree in which a node represents a server. The nodes at the lowest level of the hierarchy are designated as leaves while the others are known as branches. The purpose of this architecture is to divide the system resources through multiple servers. Therefore, a parent node handles a section of the bandwidth and share it among their children. To prevent the children from excessive bandwidth usage, parents have a local scheduler that manages their access to the resource. The entities that consume the bandwidth, i.e., streams, are connected to the leaf nodes, with a single stream being commonly associated with only one server. This architecture was designed to support dynamic reconfigurations within the structure for both the servers' organization (e.g., add/remove serves) and individual characteristics. These are made through specific requests transmitted by the nodes to the switch. These reconfigurations can, however, jeopardize the timeliness of

the system. Thus, to prevent possible issues, the switch assesses if the temporal requirements of all servers and streams, after the request outcome, are met.

Using the authors' notation, each component (servers and streams) is represented by Γ_{y_x}, where y identifies the hierarchy level and x the component within that level, as illustrated in Figure 3.8. The following model defines a set of asynchronous streams:

$$AS_{y_x} = (C_{y_x}, Tmit_{y_x}, Mmax_{y_x}, Mmin_{y_x}, P_{y_x}, RT_{y_x}, D_{y_x}) \tag{3.8}$$

where C_{y_x} is the transmission time of the stream messages, $Tmit_{y_x}$ the minimum interarrival time, D_{y_x} the stream deadline, P_{y_x} the associated leaf server, RT_{y_x} the computed response time, and $Mmin_{y_x}$ and $Mmax_{y_x}$, the minimum and maximum size of the streams' packets respectively.

The model which characterizes the servers (3.9), uses similar notation as the one used in the streams (3.8). However, it this case, C_{y_x} represents the server capacity, $Tmit_{y_x}$ its replenishing time, D_{y_x} the deadline, P_{y_x} its parent, RT_{y_x} a computed upper bound of the server response time, and $Mmin_{y_x}$ and $Mmax_{y_x}$, the minimum and maximum packet transmission times. This nomenclature will be used when explaining the schedulability tests proposed by the authors.

$$Srv_{y_x} = (C_{y_x}, T_{y_x}, Mmax_{y_x}, Mmin_{y_x}, P_{y_x}, RT_{y_x}, D_{y_x}) \tag{3.9}$$

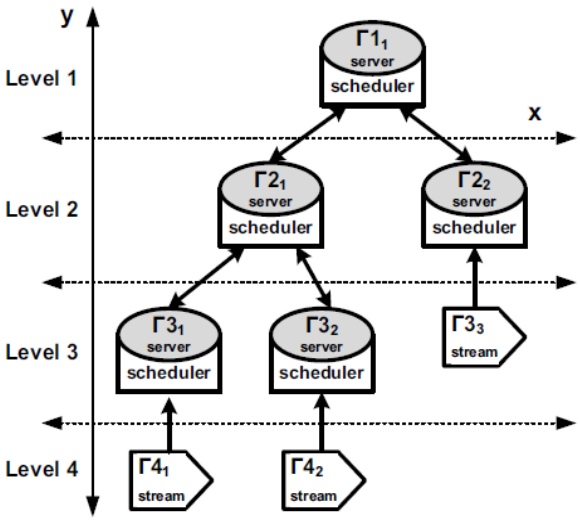


Figure 3.8: A general architecture of the proposed server-based hierarchy (*from Rui Santos et al [25], 2011).*

As previously described, the schedulability analysis verifies if the results of a request may cause failures in the components' deadlines. The proposed algorithm is based on DM and considers the following two constraints: (i) preemptions can occur, but not during packet transmission and (ii) exceeding the server capacity is not allowed. To prevent overruns (constraint (ii)), this algorithm inserts an idle-time whenever a server cannot process a full packet with the

current capacity it possesses. As shown in Figure 3.9, this procedure delays packet executions until the capacity budget is replenished. The idle-time is strictly associated with message transmissions and thus impacts the response time. Therefore, it is important to know the maximum value that this interval can influence on a server, which is equal to its maximum packet transmission time ($Mmax$). When a modification request surges, the algorithm verifies the impact caused by the $Mmax$ and $Mmin$ changes on the hierarchy and evaluates the results. This is made through two distinct phases.

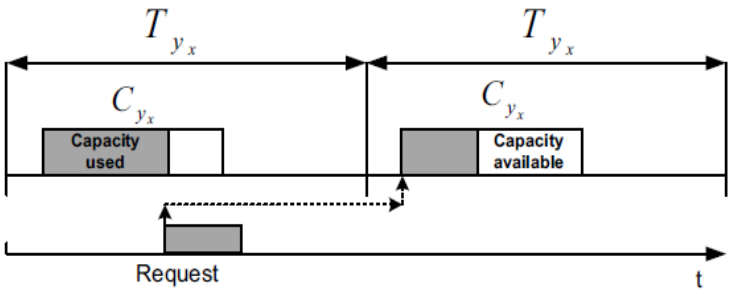


Figure 3.9: Example of idle-time insertion (*from Rui Santos et al [25], 2011).*

In the first phase, the impact of the Mmax and Mmin modifications on all the hierarchy levels is assessed. Starting from the lower levels, parent servers, inherit the Mmax and Mmin values of their children. Therefore, at the end of this phase, the top-level component will have the maximum and minimum packet transmission times among all streams. Lastly, it is also verified if every server has enough capacity to process the largest packet they can receive:

$$\forall \Gamma_{y_x, y=1...N_{levels}, x=1...N_{components}}, C_{y_x} \geq Mmax_{y_x} \tag{3.10}$$

If this condition fails, the request is denied, the structure configuration stays the same, and the analysis terminates, otherwise it continues to the next phase.

The second phase consists of analyzing the hierarchy schedulability by computing the worst-case response-time (RT_{y_x}) of each component, and compare those results to their respective deadlines. For this, the authors use a technique based on two specific functions used in hierarchical scheduling analysis, the request bound function, and the supply bound function. The latter, $\mathbf{rbf}_{y_x}(t)$, quantifies the maximum load received by a parent component ($\Gamma_{P_{y_x}}$) of a given server (Γ_{y_x}) until the instant t. The submitted load comes from the server itself together with both interference and blocking off higher and lower priority components, respectively. The former function, $\mathbf{sbf}_{P_{y_x}}(t)$, computes the minimum bandwidth supply provided by a parent component of Γ_{y_x} to its children at the instant t. The response time is then computed using the following equation:

$$\begin{cases} RT_{y_x} = \omega_{y_x} + M_{y_x}^{last} \\ \omega_{y_x} = earliest \;\; t > 0 : \mathbf{rbf}_{y_x}(t) = \mathbf{sbf}_{P_{y_x}} \end{cases} \tag{3.11}$$

where $M_{y_x}^{last} = Mmin_{y_x}$ and ω_{y_x}, the lapse from when the server becomes ready until its last packet starts being transmitted. The request bound function can then be obtained from:

$$\begin{cases} \mathbf{rbf}_{y_x}(t) = IH_{y_x}(t) + BL_{y_x} + C_{y_x} - M_{y_x}^{last} \\ IH_{y_x} = \sum_{\Gamma_{y_j} \in hp(\Gamma_{yx})} \left\lceil \frac{t}{T_{y_j}} \right\rceil \times C_{y_j} \\ BL_{y_x} = \max_{\Gamma_{y_j} \in lpe(\Gamma_{y_x})} Mmax_{y_x} \end{cases} \tag{3.12}$$

where IH_{y_x} is the higher priority load generated and submitted by the parent component of Γ_{y_x} until the instant t and BL_{y_x}, is a blocking term associated with the non-preemptive transmissions between messages. This factor is maximized by the maximum transmission time of all lower priority components ($lpe(\Gamma_{y_x})$).

The supply bound function, on the other hand, can be calculated using the EDP model, previously explained in Section 2.1.5. Using this model a server component can be defined as: $\Gamma_{y_x} = (\Pi_{y_x}, \Theta_{y_x}, \Delta_{y_x}) = (T_{y_x}, C_{y_x} - Mmax_{y_x}, RT_{y_x} - Mmax_{y_x})$ and the function itself can be obtained using (2.23). Lastly, by assessing if the worst-case response of every component Γ_{y_x} (obtained from 3.11) is equal or lesser than their respective deadlines, it is verified if the hierarchy is feasible :

$$\forall \Gamma_{y_x, y=1...N_{levels}, x=1...N_{components}}, RT_{y_x} \geq Mmax_{y_x} \tag{3.13}$$

Similar to the previous phase, a failed test for a single component means that the hierarchy stays unchanged, and the schedulability analysis terminates.

CHAPTER 4

Network Simulators

Contents

*N*etwork simulators play an essential role in the development and testing of communication networks. These allow engineers and researchers to create experiments and evaluate networks' performance while reducing costs and time associated with setting up physical setups, particularly for large and sophisticated systems. Additionally, simulators facilitate the test and validation of new protocols and technologies in a controlled environment. This chapter reviews some of the current network simulators and compares them to find the best candidate to implement the previously explained switch model (HaRTES). It starts with a summary description of six different simulators and their principal features, followed by their comparison. The chapter concludes with a presentation of the main aspects and components of the chosen simulation framework (OMNeT++).

4.1 Overview of different network simulators

Generally speaking, the central purpose behind network simulators is to create virtual models that capture the behavior of physical networks and their devices. These models can then be

easily used and modified, thus allowing the analysis of different experimental scenarios whose real-world implementation could be more complex, expensive, and time-consuming. Figure 4.1 depicts an abstraction architecture of network simulators. The simulation models, e.g., switches, hubs, routers, are implemented through a kit of several algorithms and structures, depending on the programming language employed. The simulations are then based on a set of parameters determined by the user, e.g., traffic rate, scheduling algorithm, number of nodes, and so forth. When completed, the simulator returns a set of metrics whose purpose is first to validate the implemented model and, subsequently, analyze the performance of different networks using the previously verified models [29].

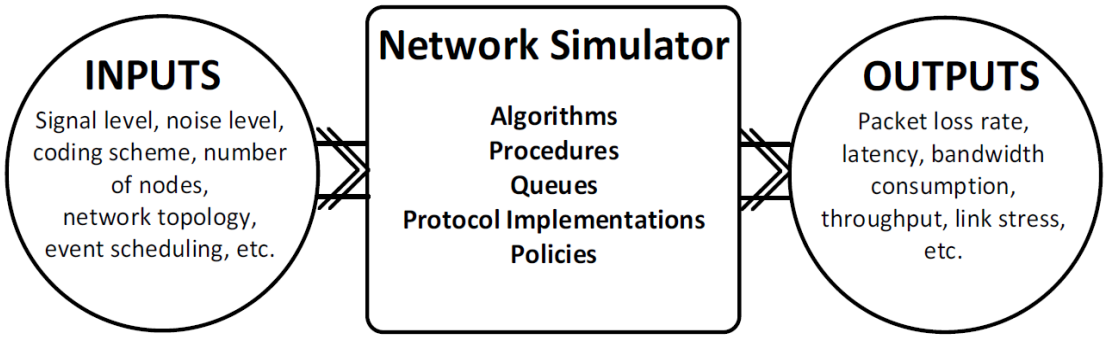


Figure 4.1: Abstract structure of network simulators (*from J.Suárez et al. [29], 2015*).

Most of the existing network simulators are discrete event-based, i.e., both the system and the nodes' operation are modeled by a (discrete) sequence of time events. The simulator then stores pending events by the order in which they are triggered. As such, the simulations themselves are just the execution of the successively stored events. Computer networks are mostly simulated using discrete-event software since the behavior of their protocols can be modeled by a finite state machine. Consequently, simulations are generally faster as the simulators can jump from between consecutive states [29]. Furthermore, discrete-event simulators have better performance in terms of flexibility and computer overhead when compared to other types (examples of non discrete-based simulators include GrooveNet[30] and NCTUns[31]).

Selecting the best simulator is not a straightforward task. Depending on the application, some network simulators may be more appropriate than others. For example, when designing wired based networks (e.g., Ethernet protocol), it is better to use simulators that already have such models implemented. Other relevant traits when studying such tools include their scalability, availability, data manipulation (analysis software), user interface, and so forth. The following section presents six different discrete-event network simulators, ns-3, OMNeT++, QualNeT, NetSim, OPNET and TrueTime, and compares them based on some of their generic technical and user characteristics. Readers are referred to [32],[33] for more information regarding the those simulators and others not examined.

4.1.1 ns-3

Network Simulators (NS), is a series of three discrete-event simulators, ns-1, ns-2, and the latest one, ns-3 [34]. Regarding the latter one, ns-3 is an open-source and free software licensed under the GNU GPLv2 license for development and use. It was developed as an improvement of ns-2, thus allowing compatibility between the models of its predecessor to be used. The main language used to write the simulations and core models is C++, with bindings available in Python.

The ns-3 supports both simulations and emulations using sockets with animators to visualize the results. One key feature of ns-3 is the possibility of integrating physical nodes into the network through emulation. The simulator includes specific net devices that, when associated with the host system, provide emulation capabilities. Examples include sending data from an ns-3 simulation to a real physical network or transmit data from a physical node to an ns-3 simulation.

The simulator also includes a real-time scheduler that allows interactions with external real-time systems. This scheduler synchronizes the simulation clock with the external time base. Thus, between consecutive events, the simulator compares the next event execution time with the external clock to keep the system synchronization. If that event is scheduled a given instant t in the future, the simulator stays idle until the "real" time reaches that moment. It then executes the event and repeats the process.

The simulation results are stored in generated pcap trace files for debugging. To maintain a large number of high-quality validated models, ns-3 relies on the vast community of developers and users to update, debug and develop new models. It includes models for, for example, wired and wireless communications, device-to-device communication protocols, and Software-Defined Networking (SDN) devices.

4.1.2 OMNET++

OMNET++ [35] is a modular and extensible library and framework based on C++, fundamentally used for building computer networks and network simulations. Although being often quoted as a network simulator, OMNET++ includes the primary tools to write simulations, but itself does not provide any components specifically for computer networks; these application areas are supported by independent simulation models (e.g., INET [36] for Ethernet, Internet protocols, etc., SimuLTE [37] for LTE (User-Plane) models).

OMNET++ implements a component-based architecture for networks, which are built with specific structures designated *simple modules*, programmed in *C++*. It then uses an infrastructure to assemble simulations from these components using a specific high-level language, Network Description (NED). Within the ned files, users can define several parameters for their implemented networks. These can be static (e.g., link speed, number of nodes) or dynamic (e.g., scheduling policy utilized). Dynamic parameters are configured during the network initialization. Lastly, OMNeT++ includes an Eclipsed-based Integrated Development

Environment (IDE) (Figure 4.2), graphical run-time, extensions for real time-simulations, network emulation, database integration and other functions.

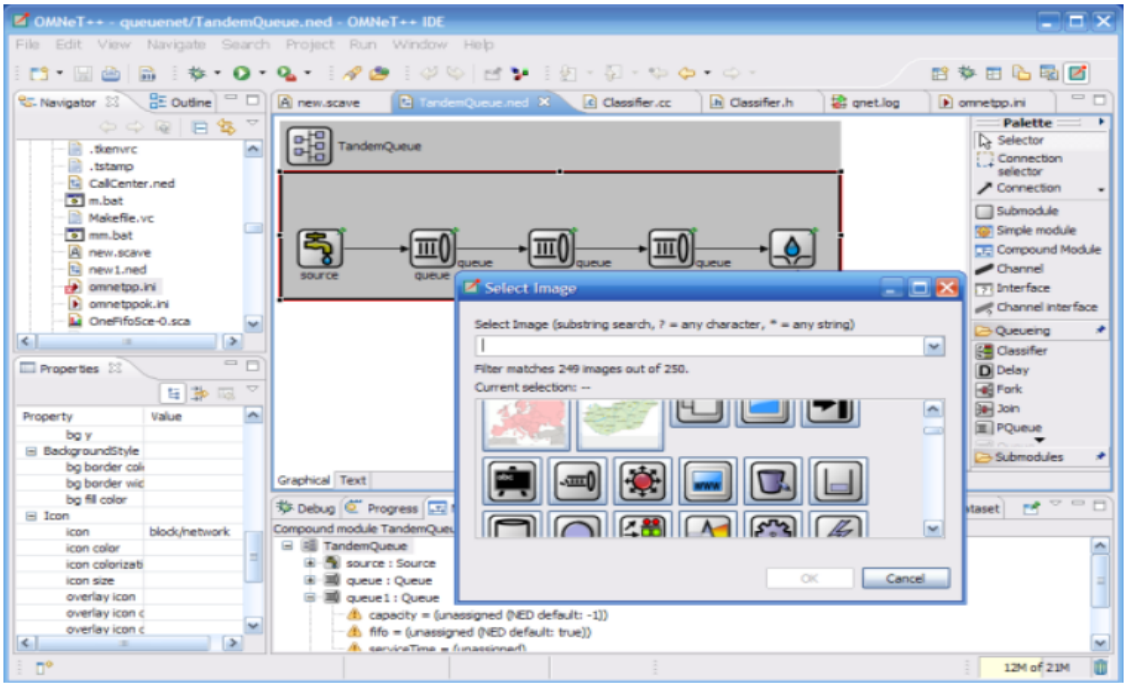


Figure 4.2: OMNeT++ IDE (*from [35]*).

4.1.3 QualNet

Quality Networking (QualNet) [38] is a network simulation software developed for planning, testing, and training network models with high precision of their real behavior. It is a commercial version of GloMosim [39],[40], with exclusive licenses for academic use, written purely in *C++*. The models used by the simulator are divided into nodes and links. The former ones are entities that represent network elements and endpoints (routers, switches, satellites, mobile phones, radios, sensors, PCs, servers, firewalls, etc.). The latter type serves as connections that interconnect nodes (LAN segments, internet circuits, radio transmissions, Wi-Fi signals, and so on). QualNet also includes an extensive range of libraries with models for wired and wireless networks, sensor networks, mobile ad hoc network (MANET), and WiMAX (IEEE 802.16). Some of the features provided by the simulator are [38], [32]:

- *Processing Speed*: allows users to run multiple analyses while changing the model/simulation parameters in a short-time.
- *Scalability*: simulation of large networks with high fidelity.
- *Model Fidelity*: high fidelity protocol models for accurate simulation behaviors.
- *Extensibility*: the simulation tools can connect to other hardware or software applications.

4.1.4 NetSim

NetSim [41] is a discrete event-based simulator for network simulation and protocol modeling. NetSim has three distinct versions, Academic, Standard and Pro, which require a specific license for their usage. It comes with a GUI (Figure 4.3) for easy network creation (click and drop services) and testing (animated simulations that execute autonomously or can be controlled by the user). Netsim also includes support for emulations where physical components can be connected to the created networks.

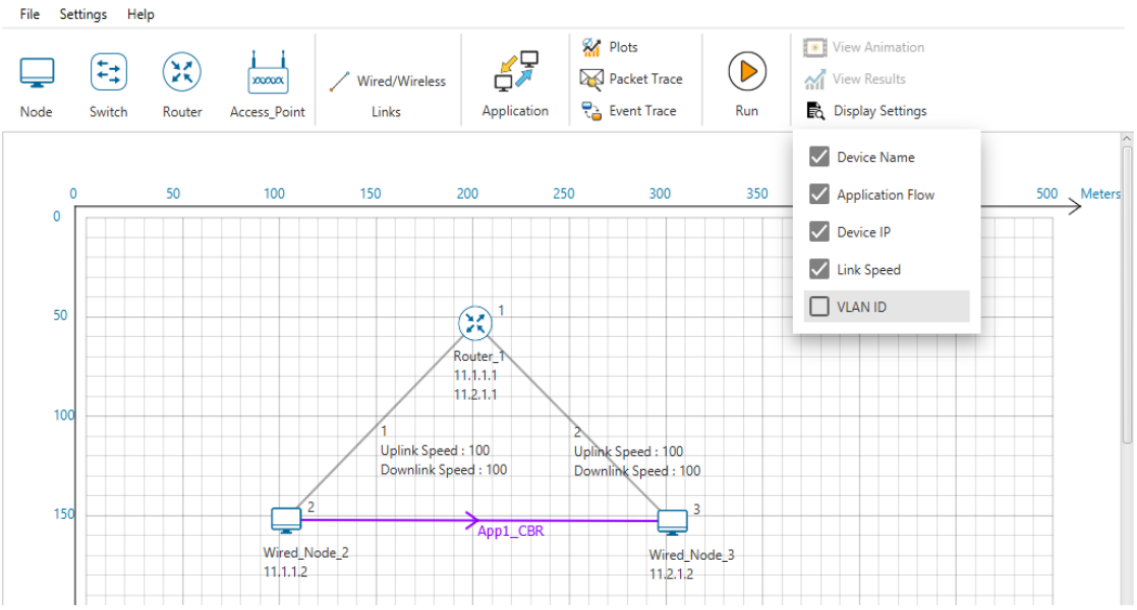


Figure 4.3: NetSim GUI (*from [42]*).

NetSim provides performance metrics of the different components (e.g., throughput, simulation time, generated, and dropped packets), from the network itself to its nodes or packets. It includes an in-built environment that allows the user to create specific models using C-code or extended some of the implemented algorithms provided by the simulator. NetSim includes in-built models for wired and wireless networks, IoT networks, cognitive radio networks, Vehicular Adhoc Networks, etc. Furthermore, it provides interfaces for external software such as MATLAB, SUMO and WIRESHARK, depending on the version used. Besides being a commercial simulator, a negative point of NetSim is being a single processer event simulator. A single event queue is used to store the events [32].

4.1.5 OPNET

OPNET [43] stands for Optimized Network Engineering Tool. It is a high level, event-based commercial network simulator useful for testing large and complex networks. OPNET provides a powerful graphical interface so users can implement the networks and create the components with an object-oriented programming language (*C++*). The simulation configurations (e.g., network topology) are initialized using a provided GUI, through specific XML files or C library calls.

In OPNET, the simulation stores all the events in a global list. These are scheduled based on the timing order of the list and, when completed, are removed. The event list is managed by the simulation kernel. This entity requests events from modules, inserting them in the list and delivers them when they reach the top of the list. Upon reaching the head of the list, the event becomes an interrupt which is then delivered by the kernel to the appropriate module, that processes it (Figure 4.4).

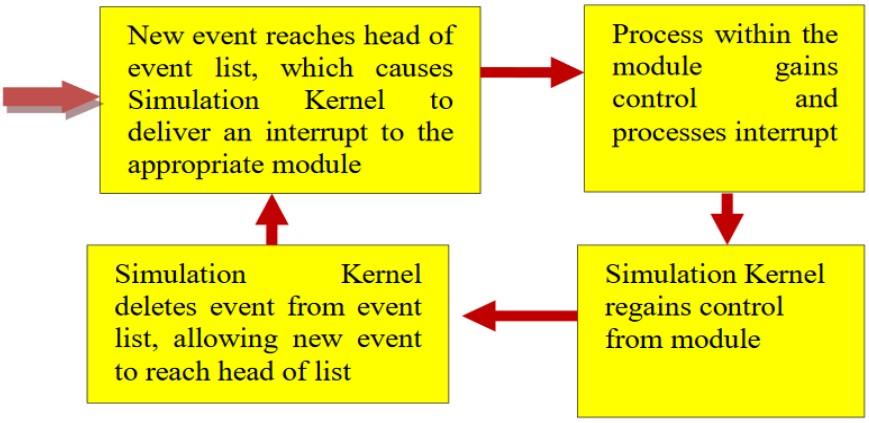


Figure 4.4: OPNET architecture (*from Saba Siraj et al. [44], 2012*).

The purpose of the OPNET is to optimize the networks costs, efficiency, performance, viability, and scalability. As such, simulations can be easily reconfigured and repeated. It supports multiple network configurations, protocols, traffic, and user applications with the combination of nodes (fixed, mobile, or satellite), links (simplex, duplex, wired or wireless) and subnets. This simulator also includes a comprehensive development environment for fast and easy simulation results analysis.

4.1.6 TrueTime

TrueTime [45] is a Matlab/Simulink based simulator for network embedded real-time systems. The software consists of models of real-time kernels and networks, as Simulink [46] library blocks (Figure 4.5) [47]:

- **Kernel block**: simulates an event-based real-time kernel, executes real-time tasks and interrupts handlers based on a scheduling policy selected by the user. It also includes other features such as data logging, task synchronization, and task activation graphs. The TrueTime simulator kernel is structured as a real-time kernel, with several implemented queues for the different objects (tasks, interrupts handlers, timers). It includes n ready queues for objects ready to execute, a time queue for scheduled objects, and different waiting queues for tasks that try to access monitors, mailboxes, and semaphores. These are managed by the kernel or calls to the kernel.

- **Network blocks**: simulate the behavior of several link-layer MAC protocols (Ethernet, CAN, Round Robin, PROFINET, FDMA, and TDMA for wired communications and

WLAN and ZigBee for wireless). However, it does not support higher-level applications. Despite the communication type, the TrueTime network blocks always generate the network nodes transmission schedule. For wired communications specifically, the scheduling policy employed will depend on the network parameters (data rate, minimum frame size, loss probability, etc.). It supports direct address transmissions or broadcasting. Regarding wireless communications, devices cannot send and receive packets at the same instant. It is also required to take into account packet loss and attenuation for radio signals as well as interference from other nodes for shared medium networks. The wireless network block provides multi-path propagation, reflection and shadowing. Network parameters include transmission power, receiver sensibility, data rate and others.

- **Battery blocks**: simulate battery-powered devices (charged or discharged). It supports dynamic consumption, i.e., users can increase or decrease the kernel CPU speed to increase or decrease power usage, respectively.

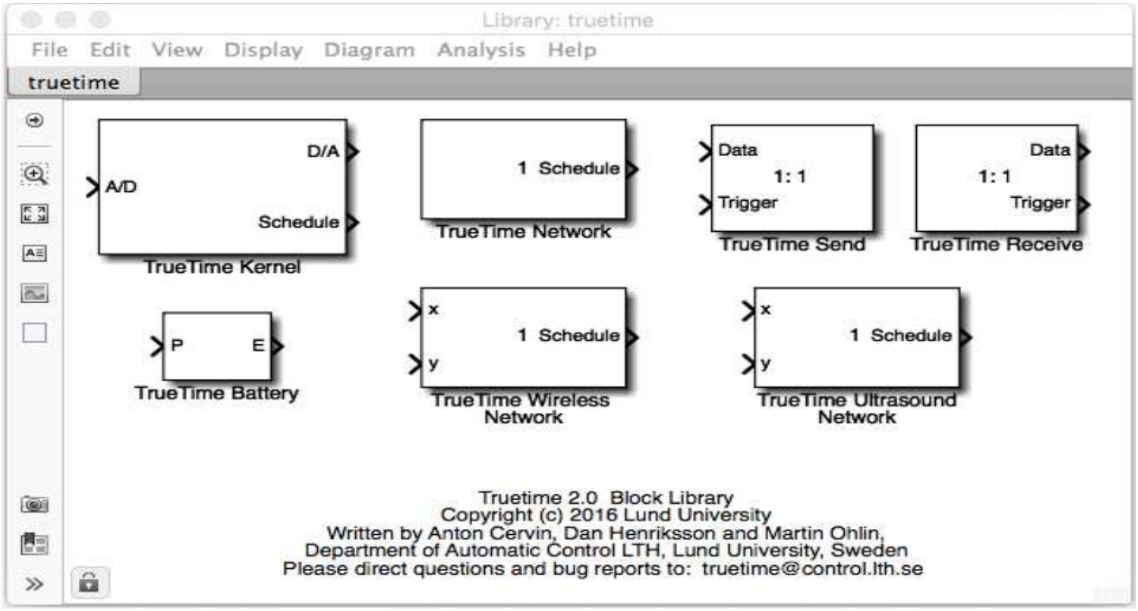


Figure 4.5: TrueTime Simulink blocks (*from [47]*).

In the TrueTime simulator, the several network components are structured into tasks and interrupt handlers and implemented by users as M-files or in C++ functions. Tasks can be periodic or aperiodic. For the former type, these are created by the local periodic timer. The aperiodic tasks, however, can be created in response to an external event. Tasks contain several properties, some of which are dynamically updated by the kernel throughout the simulation (e.g., absolute deadline, release time) while others are static and only adjusted by the user (e.g., WCET, period). The drawbacks of this simulation software include the impossibility to access the simulation models for further analysis, lack of support for higher-level network protocols and slow simulations for larger models.

4.2 A comparative analysis of network simulators

When comparing the different network simulators, there are several approaches one could take. If the software tools performance were the main requirement, a series of tests would be performed to evaluate which simulator is the most efficient. This procedure, however, is not possible since not all the previously described simulators can be acquired. Furthermore, even if the simulator has, for example, good computation time or low memory usage, if it does not possess the required models to implement the HaRTES model (e.g., models for Ethernet-based communications), its utility, in the scope of this dissertation, is worse than other simulators with inferior performance. Thus, before comparing the most relevant existing simulators, a series of necessary requirements to implement the HaRTES switch model were established:

- **Availability:** The simulator must be open-source. Although several organizations provide academic licenses, the overall process of inquiring the license can be very timing consuming. By selecting an open-source simulator, the software learning process can commence immediately after choosing it.

- **Easiness:** The selected simulator must be easy to use and fast to learn. As the available time to implement the switch model is limited, picking a simulator that includes such features, facilitates considerably the overall process. Other important properties include easiness in: (i) changing the simulation parameters, (ii) code implementation and (iii) model creation, i.e., the simulator should include a user-friendly GUI.

- **User Community:** By default, a good simulator is one that is widely used. If the software tool is regularly utilized, it receives several updates with new protocols models and other features. Furthermore, a big user community can be useful when certain implementations issues appear, which can be answered via forums.

- **Available Models:** For the implementation of the HaRTES switch model, several computer network models are required. These include the Ethernet protocol (interfaces for receiving and dispatching frames between nodes) and the OSI layers for transmitting messages between the network nodes. These models are essential so that the switch implementation is the focus of this study.
  Documentation is also a crucial focus when learning how to use the different provided simulation models. Besides providing the models, the simulator must properly describe how to use and employ them.

After conducting a general analyzation of the previously described simulators, it was determined which one would be the best choice to implement the HaRTES switch model. Firstly, a table containing each of the simulators' characteristics was created, Table 4.1. This table is divided into technical (programming language, integrated applications, available networks, etc.) and user characteristics (easy to use, free license and so forth).

Based on the table characteristics and the requisites previously presented, QualNet, NetSim and OPNET are immediately discarded as they required a paid license for their utilization.

From the remaining three open-source simulators, TrueTime appears to be the one that least fits the profile of the intended simulator. Its low utilization by the research community is demonstrated in the software version, with the last update being made in 2016. For those reasons, the simulator was discarded.

For the remaining two simulators, ns-3 and OMNeT++, they both have high academic utilization (more than a thousand papers published in ieeexplore), thus resulting in being regularly updated. A study conducted by Zarrad and Alsmadi [48] demonstrated that OMNeT++ has a better performance than ns-3 for certain properties such as CPU usage and simulation run time while it is more inefficient in others. Overall, both simulators could be used to implement the switch simulation model however, considering the timeline of the dissertation, OMNeT++ would be more beneficial since it is easier to work and faster to learn.

Table 4.1: Table with generic characteristics of different Network Simulators [32], [44], [46],[49], [48] ,[50].

		NS-3	OMNeT++	QualNet	NetSim	OPNET	TrueTime
Technical Characteristic	Interaction with real-time systems	Possible	Possible	Possible	Possible	Possible	Possible
	Analysis Tool	Yes	Yes	Yes	Yes	Yes	No
	Scalability	Yes	Yes	Yes	Yes	Yes	No
	Generates Trace Files	Yes	Yes	Yes	Yes	Yes	Yes
	Available Networks	Wired, Wireless, ADHOC, MANET, SDN, VANET, etc.	Wired, Wireless, ADHOC, MANET, SDN, WBAN, etc.	Wired, Wireless, SDN, ADHOC, MANET, etc.	Wired, Wireless, IoT, BGP networks, Cellular networks, etc.	Wired, Wireless, ADHOC, MANET, Radio networks, etc.	Wired and wireless networks
	Processing Speed	Moderate	Good	Excellent	Excellent	Excellent	-
	GUI support	Good	Good	Excellent	Excellent	Excellent	Good
	Language	C++, Python	C++	Parsec C++	C, Java	C (C++)	Matlab, C++
	Environment OS	Windows, Linux, unix	Window, Linux, unix, MAC OS	Windows, Linux, DOS	Windows 7 (SP1 or higher), Win 8 or Win 10.	Windows, Linux, Solar	Window, Linux, MAC OS (requires MatlabR2012a with Simulink 7.9 or later)
User Characteristics	Easy to Use	Hard	Easy	Moderate	Easy	Easy	Easy
	Learning curve	Moderate	Short	Short	Short	Moderate	Moderate
	Documentation	Excellent	Good	Excellent	Excellent	Good	Good
	Updates	Recent (18/09/2019)	Recent (13/01/2020)	Recent (19/02/2020)	Recent (20/01/2020) "Beta"	Dated (4/02/2019)	Old (6/4/2016)
	License	Open-Source	Open-Source	Commercial	Commercial	Commercial	Open-Source
	Academic Utilization	Very High	Very High	Moderate	Poor	Very High	Poor

4.3 The OMNeT++ framework

OMNeT++ [51] is an open-source, C++ based, discrete-event framework design for modeling and simulating computer networks, multiprocessors and distributed systems. As previously mentioned, OMNeT++ is a framework, i.e., it does not provide the simulation components for the various network areas, e.g., computer networks. Instead, it presents the essential tools to write the simulations while specific application models (e.g., Ethernet, PPP, Wi-fi, etc.) are created independently by the user community and employed in OMNeT++ as libraries. This framework was designed to support the creation of large-scale networks. For this, it uses a hierarchical architecture which allows the re-utilization of components.

4.3.1 Model Structure

The OMNeT++ networks, also known as models, are built with reusable structures named modules, which communicate with each other via message passing. In OMNeT++, the network itself is a module (the top-level one), which may contain submodules and themselves may contain other submodules, thus a hierarchical structure is formulated, whose levels are unlimited. Modules that contain submodules are designated compound modules. On the other hand, the ones at the hierarchy lowest level are termed simple modules. These are active modules that contain the algorithms that model the the different components behavior. Simple modules are written in as a set of *C++* functions supported by the OMNeT++ simulation class library, thus providing flexibility when implementing the network components. To write these structures code, OMNeT++ provides an Integrated Development *C++* Environment, so there is the possibility to create, test, and debug the code without the need for additional software. Simple modules can be combined to create compound modules.

Simple modules communicate with each other by transmitting messages directly to the destination module or through a series of *connections* and *gates*, i.e., input/output ports. (Figure 4.6). Indirect messaging is normally employed for wired communications when different network components gates are connected via *connections*, i.e., the communication path is known and created in advance. On the other hand, direct messaging is generally used for wireless communications, where the communication path is not known in advance. For the latter type, instead of using the connections, a destination module pointer is firstly obtained, which then allows transmitting the data to the desired module.

Messages can be actual frames or packets from communication networks, costumers for queueing networks, and other types, depending on the simulated application. These can be transmitted between different modules or directly within the module itself, i.e., self-messages, which can be used to implement, for example, timers. For wired communications, messages travel through a series of *connections* that always start and end in simple modules. These paths have three different properties: (i) propagation delay, (ii) data rate, and (iii) bit rate. These can be individually specified for each connection created or defined as a unique link for re-utilization with the *channel* type (Figure 4.7). All these components and connections are then assembled into networks using a specific type of language known as NED.

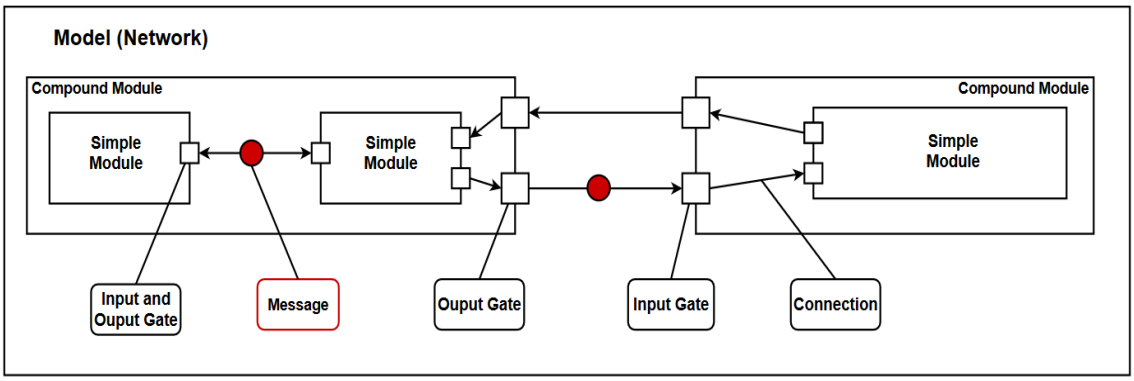


Figure 4.6: OMNeT++ module structure.

```
network ConnectionExample{
    types:
        channel Connection_1 extends ned.IdealChannel{
        }
        channel Connection_2 extends ned.DelayChannel{
            delay = 100ms;
        }
        channel Connection_3 extends ned.DatarateChannel{
            delay = 100ms;
            datarate = 100Mbps;
        }
    submodules:
        Module1: SimpleModule;
        Module2: SimpleModule2;

    connections:
        Module1.In      <--  Connection_1 <-- Module2.Out;
        Module1.Out     -->  Connection_2 --> Module2.In;
        Module1.InOut   <--> Connection_3 <--> Module2.InOut;
        Module1.InOut$i <--  {datarate=100Mbps;} <-- Module2.InOut$o;
        Module1.InOut$o -->  {delay=100ms;} --> Module2.InOut$i;
```

Figure 4.7: OMNeT++ connection types.

4.3.2 NED Language

In OMNeT++, network structures and topologies are described in a specific file that uses NED, a high-level description language. When writing such entities, OMNeT++ provides both code and graphical-based approaches that allow a more accessible environment for editing. Usually, within a NED file, the simple modules are declared, and the compound modules and networks are defined, including creation of their respective gates, connections, and internal *parameters* (Figure 4.8). Parameters are variables used to pass configuration properties for the simple modules. These can be different types, e.g., int or boolean, and can be accessed when initiating the simulation, thus granting higher flexibility for the users.

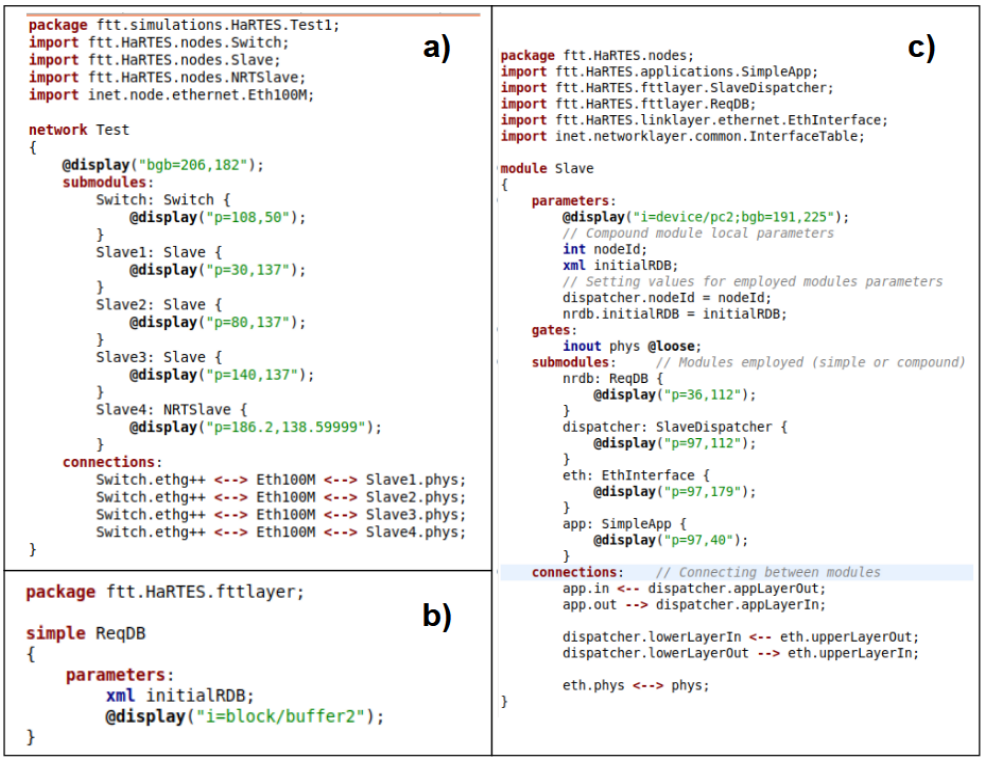


Figure 4.8: Example of a NED file structure: a) network; b) simple module; c) compound module.

Regarding Figure 4.8, some details should be highlighted. Firstly, every structure (simple, compound modules, and networks) have a defined package at the beginning of the file. This package must be included when using the module in other NED files. Another important aspect is the modules parameters. Simple and compound modules can declare local parameters (e.g., *int nodeId*, *xml initialRDB*) and define their values in their individual NED files. However, these can also be defined by other compound modules that use them as submodules as illustrated in Figure 4.8 c). In this case, the *initialRDB* parameter of the *reqRDB* simple module is defined in the *Slave* compound module. Lastly, when employing different submodules, it is possible to declare them with a different name (e.g., in the *Slave* compound module, the *reqRDB* simple module is termed *nrdb*). This is useful when the same module is employed several times in a compound module (Figure 4.8, a)).

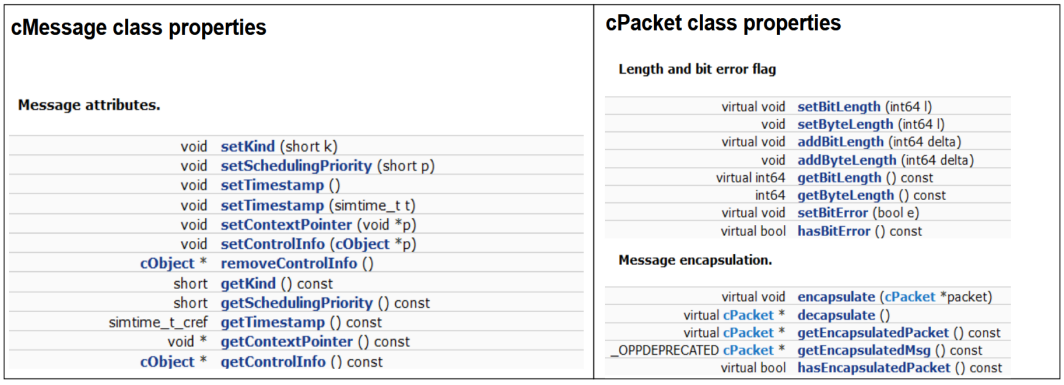
Besides the previous characteristics, the NED language also includes the following features [52]:

- ***Hierarchical structure***: decreases the complexity of projects by disassembling complex modules into simpler ones.
- ***Component-based***: simple and compound modules can be reusable, thus allowing the usage of component-based libraries, e.g., INET.
- ***Interfaces***: Module and channels interface placeholders can be used instead of concrete modules and channels. The latter ones can then be determined during the network setup by specific parameters. Considering, for example, a compound module with a submodule named etherApp of the type IetherApp, which is a module interface, etherApp may be one of the different types of IetherApp (etherApp8021Q, etherApp8023, etc.), depending on a parameter chosen by the user.
- ***Inheritance***: modules and channels can be subclassed. The derived components can employ new parameters, gates, submodules, and connections.
- ***Packages***: a Java-like package structure was implemented in the NED language to reduce the risk of name clashing between models.
- ***Inner-Types***: channels and module types used by a compound module can be created within that module to reduce namespace pollution.
- ***Metadata***: it is possible to annotate the different structures within the NED file (modules, gates, connections) in the parameters. They carry extra information that can be used in several applications in OMNeT++.

Upon designing the desired network, this can be initiated in specific INI files, which are used to set up the configurations and input data for the simulation Therefore, OMNeT++ separates the different aspects of the simulation throughout different types of files. The modules' behavior is implemented in C++ files, the network structure and the components parameters are defined in NED files, and the simulation desired variables and initiation are set in INI files.

4.3.3 Messages and Packets

As previously explained, messages are essential for communications between the created modules. These can be represented by the *cMessage* and *cPakcet* classes, provided by OMNeT++, and used in the simple modules source files. The former class is used to represent events, messages, jobs, and other simulation entities. It includes different properties such as name, message kind, and time stamp. The latter type represents packets, frames, application messages, and so forth. It adds length (bits or bytes), bit error flags, and encapsulation capabilities to the cMessage class. Nonetheless, both classes have predefined parameters that cannot be modified (Figure 4.9), which may restrict some applications.



cMessage class properties

Message attributes.

void	**setKind** (short k)
void	**setSchedulingPriority** (short p)
void	**setTimestamp** ()
void	**setTimestamp** (simtime_t t)
void	**setContextPointer** (void *p)
void	**setControlInfo** (cObject *p)
cObject *	**removeControlInfo** ()
short	**getKind** () const
short	**getSchedulingPriority** () const
simtime_t_cref	**getTimestamp** () const
void *	**getContextPointer** () const
cObject *	**getControlInfo** () const

cPacket class properties

Length and bit error flag

virtual void	**setBitLength** (int64 l)
void	**setByteLength** (int64 l)
virtual void	**addBitLength** (int64 delta)
void	**addByteLength** (int64 delta)
virtual int64	**getBitLength** () const
int64	**getByteLength** () const
virtual void	**setBitError** (bool e)
virtual bool	**hasBitError** () const

Message encapsulation.

virtual void	**encapsulate** (cPacket *packet)
virtual cPacket *	**decapsulate** ()
virtual cPacket *	**getEncapsulatedPacket** () const
_OPPDEPRECATED cPacket *	**getEncapsulatedMsg** () const
virtual bool	**hasEncapsulatedPacket** () const

Figure 4.9: OMNeT++ cMessage and cPacket class properties.

In order to solve this issue, the framework allows the creation of unique messages and packets that are subclasses of cMessage or cPacket, respectively. These are designed in specific MSG files (.msg) with all their associated variables defined by the user. After completing the code, during the compilation, the message compiler is invoked and generates two files a ***.h*** and ***.cc***. The header file will contain the class declaration and must be included in the source file of a simple module that uses the created message, similar to a *C++* library. The source file will contain the header subclass implementation, as well as the code, that allows the message variables manipulation.

4.3.4 OMNeT++ Architecture

The local architecture of OMNeT++ is depicted in Figure 4.10. It comprises the Model Component Library, the simulation kernel (*Sim*), and the user interfaces libraries (*Envir, Cmdev, and Tkenv)*. The Model Component Library comprises the different modules (compound and simple) compiled code. The simulated network (simulation model) is built by the simulation kernel when the simulation is executed. The different user interface libraries then execute the simulation. They defined how the simulation is visualized (animated or not), control the overall simulation execution (e.g., start, stop, repeat the simulation, record, advance events), where the results go, and so forth.

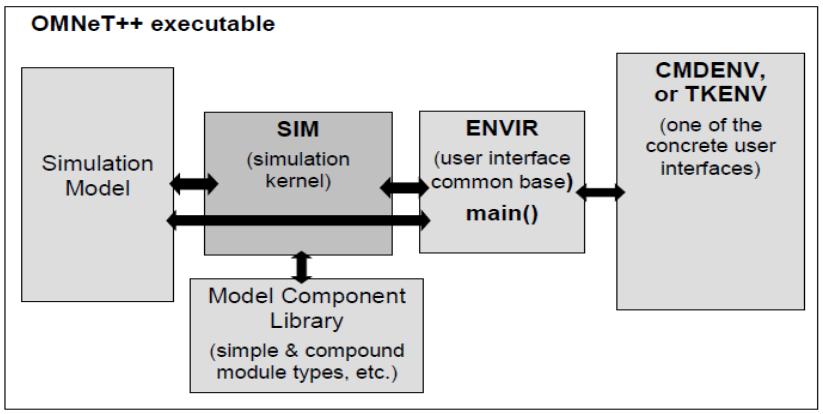


Figure 4.10: OMNeT++ architecture (*from András Varga and Rudolf Horning [51], 2008*)

An important feature of OMNeT++ is the possibility of replacing the existing user interfaces or embed the OMNeT++ environment into other applications (Figure 4.11). This procedure is possible because: (i) the simulator includes generic interfaces between the class library (*Sim*) and user libraries (*Envir, Tkenv, Cmdev*) and (ii) all the libraries are physically separated.

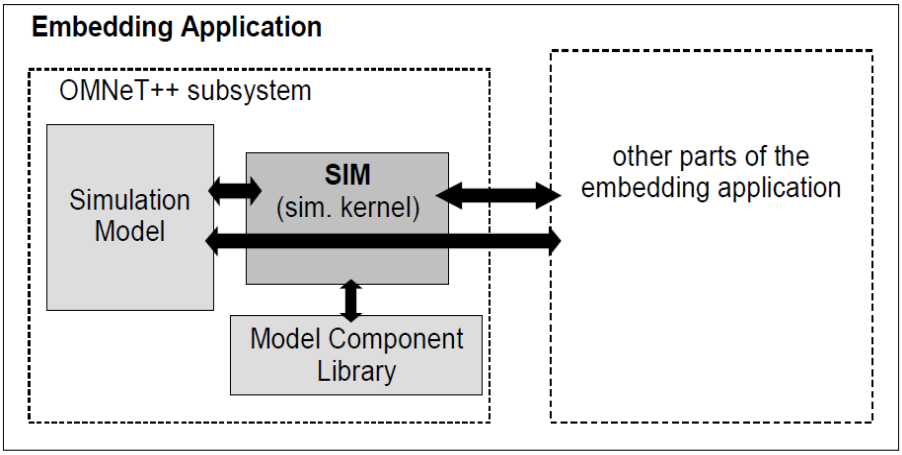


Figure 4.11: Embedded OMNeT++ architecture (*from András Varga and Rudolf Horning [51], 2008*).

4.3.4.1 User Interfacers

As previously mentioned, in OMNeT++, the simulations are executed in the provided user interfaces, *Envir, Tkenv, and Cmdev* [51]. *Cmdev* is mostly used to run simulation via command line (batch execution), making it a fast procedure. For a more user-friendly and simple debugging approach, the GUI of *Tkenv* can be utilized (Figure 4.12). It has three essential methods:

- *Automatic Animation*: this feature provides message flow animation between the simulated network different modules.

- *Module Output Window*: It is a unique window used to display the information output of individual modules or groups of them. Allows for printf()-style debugging.

- *Object Inspector*: It is a specific GUI window within the Tkenv with the properties of the inspected module. It can be used to verify the simulation object content in the desired way (histogram, graphs, individual values) or change it during any point of the simulation.

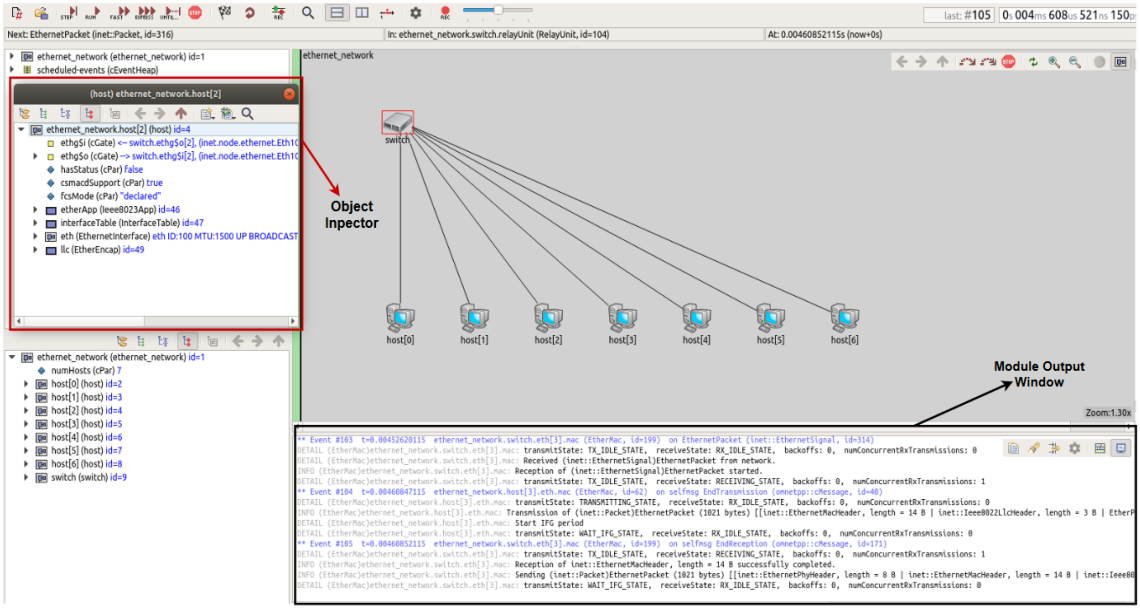


Figure 4.12: OMNeT++ Tkenv.

Lastly, the *Envir* interface is the main interface of OMNeT++. It is used to design the different networks in the NED files, implement the simple models' behavior via C++ functions, analyze the simulation results, create and initiate simulations, and so on. (Figure 4.13). All the previous features are provided by OMNeT++ without the necessity of additional programming.

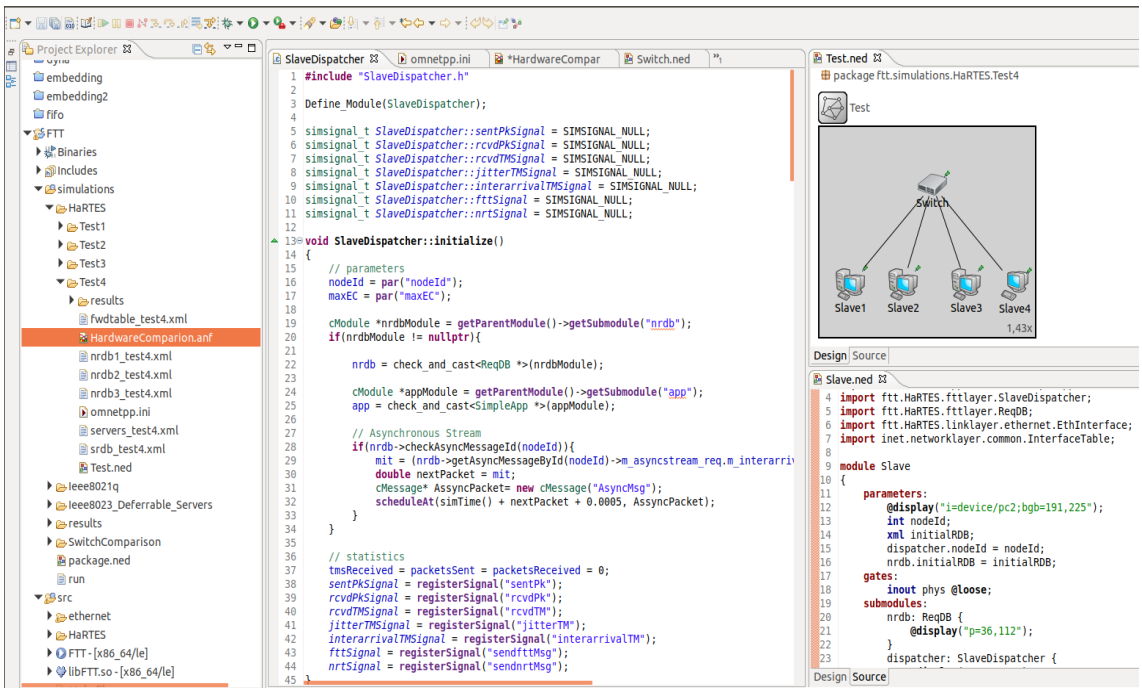


Figure 4.13: OMNeT++ main environment.

4.3.5 Analysis facilities

Analyzing the simulation results is a crucial task for validating the implemented models. Depending on the application, such results can be better represented in the form of histograms (e.g., transmission jitter) or, for other situations, standard line graphs (e.g., switch throughput). The process of analyzing this data can be, however, a lengthy process. Furthermore, users tend to repeat the same simulations multiple times, only changing some of its parameters, meaning that, depending on the software, the same plot must be created every time the simulation is completed. As such, it is important that analysis tools include simple approaches for searching the results as well as providing some data and graph automation so that the overall process can be easier and less time consuming.

The result analysis in OMNeT++ can be done within the framework with its simulation tools or using other software programs. To use the analysis tool, users must first select either a scalar (.sca) or vector (.vec) file. These are automatically generated after the simulation ended. Afterward, the Simulation IDE creates the Analysis Files (AFN) with the simulation results in form of scalars, vectors, and histograms. The user can then scan and plot the obtained data (Figure 4.14). They can also create specific patterns termed datasets with the desired data (Figure 4.15). Whenever the selected files are altered by a re-simulation, the dataset contents (variables, graphs) are automatically updated. Besides the previous procedure, other programs such as Python, Matlab, and Octave can be used to analyze the information. When employing these tools, the results must be first exported to the programs required formats using the *scavetool* of OMNeT++.

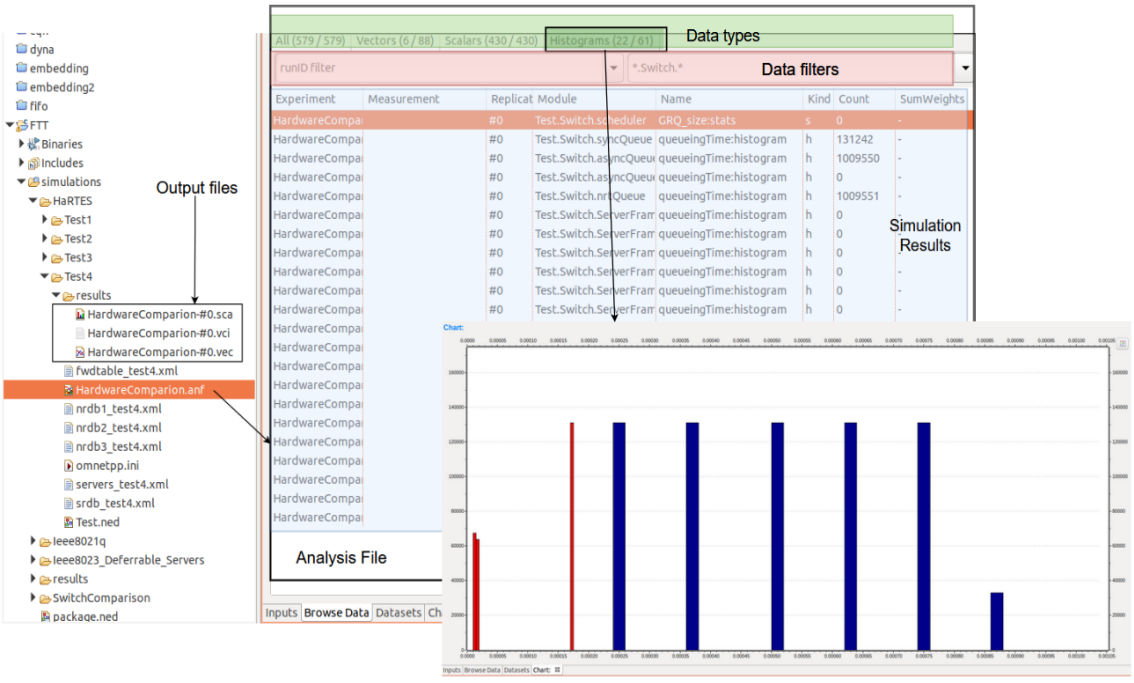


Figure 4.14: OMNeT++ Analysis tool.

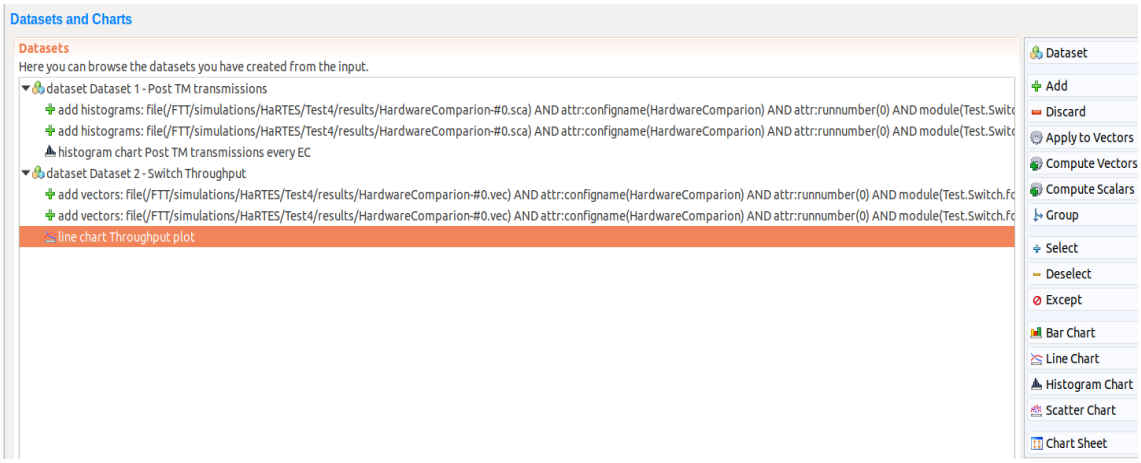


Figure 4.15: OMNeT++ dataset example.

4.3.6 Third party libraries

The majority of the different computer network models, e.g., Ethernet, CAN, Wi-fi, Bluetooth, Cyber-Physical Systems, etc., are developed by the OMNeT++ community as independent libraries and employed in the main framework. Examples include:

- **NeSTiNg [53]:** Provides simulation models for TSN.
- **CloudNetSim++ [54]:** A simulation toolkit for the simulation of distributed datacenter architectures, energy models, and high-speed data centers communication networks. It includes support for several Service Level Agreement (SLA) policies, scheduling algorithms, and modules for the datacenter components and selected by the users.
- **FLoRA [55]:** A simulation framework to perform LoRA based networks. Networks are constructed using different LoRA components, gateways, and a network server. It accurately models the LoRA physical layer with the incorporation of both collisions and capture effects. It also includes statistics of every node energy consumption.
- **NETA [56]**: A simulation framework developed for OMNeT++ to the network security field. Its purpose is to simulate attacks in heterogeneous networks. Furthermore, it tests and validates the effectiveness of security techniques/solutions and compares different employed defense techniques.

Readers are referred to [35] for a list of all the different community libraries developed for OMNeT++. Note that most of them require the utilization of both OMNeT++ and INET, an extensive library that includes most computer communication network models. As this framework was required for the implementation of the HaRTES switch model, it will be presented in more detail.

4.3.6.1 INET framework

INET [36] is an open-source library and provides an extension to OMNeT++. It contains several simple and compound modules for validating new protocols. It also includes several of the standard protocols' models for computer communication networks simulation. Just like OMNeT++, this framework is built around message passing between the assembled modules. The INET framework includes several components that, when combined, form different network devices such as switches or hubs. Users can create new ones or modify the already programmed depending on the application. An important property of INET is that this framework benefits from OMNeT++ interfaces and kernel. As such, the creation, parameterization, debugging and models testing can all be done within the Simulation IDE. The INET includes simulation models for several layers of the OSI model as well as other applications [36]:

- **Application layer models:** Models for the OSI model last layer (layer 7). Includes several traffic generator models such a constant and variable bit-rate traffic generator, HTTP traffic generator, DCHP protocol and others.
- **Transport layer models:** Includes models for transferring variable-length data sequences between the network nodes with QoS. INET as its own TCP simulation models, UDP, SCTP, and RTP with extensions (RTP Profile for Audio and Video Conferences with Minimal Control, and MPEG video payload).
- **Network layer models:** Includes models for transferring variable-length data sequences from a source to destination nodes across networks. Models implemented include IPv4, ICMPv4, ARP, etc.
- **Routing models:** Provides several modules for routing protocols in computer networks, i.e., how routers distribute the data to the destination node. These include link-state routing, OSPFv2, BGPv4, and others.
- **MANET Routing models:** Models for Mobile Ad hoc networks. Includes table-driven routing protocols (DSDV), on-demand routing protocol (DSR).
- **MPLS models:** Provides simulation models for multi-protocol label switching networks, including LDP modules for the LDP protocol (does not support the CR-LDP protocol) and RSVP modules for implementing the RSVP-TE control protocol.
- **Wired and Wireless communication models:** Models for wired and wireless-based protocol communications. As these modules are the ones mostly employed to create computer networks, they are detailed in Table 4.2.

Table 4.2: Table with the INET models for wired and wireless communications [36].

Communication Type	Protocol	Description
Wired Communications	PPP	Includes modules for simulating simple PPP links with encapsulation/decapsulation and queueing services. Features such as link configuration and maintenance from this protocol are not implemented.
	Ethernet	INET provides several modules for the Ethernet protocol. It includes models for classic Ethernet (10Mbps), Fast Ethernet (100Mbps), and Gigabit Ethernet (1000Mbps). It comprises two MAC implementations with and without the CSDM/CD mechanisms. The Ethernet frames supported are raw Ethernet, Ethernet-II, and Ethernet SNAP frames. The INET already has a functional switch and hub models.
	STP	Simulation modules for spanning tree protocol networks.
	RSTP	Simulation modules for rapid spanning tree protocol networks.
	TTE	Contains modules from CoRE4INET [57], an extension of INET for real-time Ethernet, for Time-Triggered Ethernet simulations.
Wireless Communications	Wi-Fi (802.11)	Includes modules for 802.11b and 802.11g protocols (ad hoc and infrastructure modes). Does not support fragmentations, power consumption, and polling.
	WAVE (802.11p)	Modules for the 802.11p protocol based on the MiXiM [58], a modeling framework for OMNeT++ for wireless networks (mobile and fixed).
	LR-WPANs (802.15.4)	Several modules based on the MiXiM implementations for UDB, NarrowBand, and CSDA.
	LTE	Implemented modules for both LTE Control-Plane and User Plane protocols. Regarding the latter one, includes several features such as buffering, PDU concatenation, CQI reception at the MAC layer, UM (Unacknowledged Mode) and AM (Acknowledge Mode) segmentation for the RLC layer, several eNodeB models (Macro, micro, pico eNodeBs).

In INET, the different protocols and applications communicate using a specific message class termed *Packet*. This class was purposely developed to facilitate the protocols communications, e.g., construction, encapsulation/decapsulation, fragmentation, aggregation, and overall packet manipulation. It can represent Ethernet frames, TCP segments, IP datagrams, and other type of data.

The *Packet* structure is built on top of another data structure known as *chunks*. INET comes with several C++ classes for chunks such as *BytesChunk and BitsChunk* for raw bytes and bit chunks, respectively or *SequenceChunk* for ordered sequence chunks. However, just as in OMNeT++, user can define their chunks in MSG files with the *FieldsChunk* subclass (Figure 4.16) for specific applications or protocols. Packets may also contain several chunks. To access them, the chunk API provides several functions (e.g., fragmentation and merging) (Figure 4.17).

```
class UdpHeader extends FieldsChunk
{
  chunkLength = B(8); // UDP header length is always 8 bytes
  int sourcePort = -1; // source port field is undefined by default
  int destinationPort = -1; // destination port field is undefined by default
  B lengthField = B(-1); // length field is undefined by default
  uint16_t crc = 0; // checksum field is 0 by default
  CrcMode crcMode = CRC_DISABLED; // checksum mode is disabled by default
}
```

Figure 4.16: Example of a user-defined chunk for a .msg file (*from [36]*).

```
auto sequence = makeShared<SequenceChunk>(); // create empty sequence
sequence->insertAtBack(makeShared<UdpHeader>()); // append UDP header
sequence->insertAtBack(makeShared<ByteCountChunk>(B(10), 0)); // 10 bytes
```

```
auto udpHeader = makeShared<UdpHeader>(); // create 8 bytes UDP header
auto firstHalf = udpHeader->peek(B(0), B(4)); // first 4 bytes of header
auto secondHalf = udpHeader->peek(B(4), B(4)); // second 4 bytes of header
```

Figure 4.17: Example of chunk manipulation (*from [36]*).

When communicating between different protocols, packets may require specific metadata, e.g., IEEE 802.1Q header for priority-based Ethernet communications. The INET packets

can include such information in the form of tags. The tags are attached to the whole packet (packet tags) or to a specific region (region tags). Figure 4.18 depicts an example of a packet tagging procedure. Tags can be classified as: (i) requests tags (e.g., *MacAddressReq*) that carry information from higher to lower protocol layers, (ii) indication tags (e.g., *InterfaceInd*) that provide information from lower to higher layers, and (iii) plain tags (e.g., *PacketProtocolTag*) that contain some metadata information.

```
void Ipv4::sendDown(Packet *packet, Ipv4Address nextHopAddr, int interfaceId)
{
  auto macAddressReq = packet->addTag<MacAddressReq>(); // add new tag for MAC
  macAddressReq->setSrcAddress(selfAddress); // source is our MAC address
  auto nextHopMacAddress = resolveMacAddress(nextHopAddr); // simplified ARP
  macAddressReq->setDestAddress(nextHopMacAddress); // destination is next hop
  auto interfaceReq = packet->addTag<InterfaceReq>(); // add tag for dispatch
  interfaceReq->setInterfaceId(interfaceId); // set designated interface
  auto packetProtocolTag = packet->addTagIfAbsent<PacketProtocolTag>();
  packetProtocolTag->setProtocol(&Protocol::ipv4); // set protocol of packet
  send(packet, "out"); // send to MAC protocol module of designated interface
}
```

Figure 4.18: Example of packet tagging in INET (*from [36]*).

CHAPTER 5

Implementing HaRTES switch model on OMNeT++

Contents

*A*s discussed in chapter 4, OMNeT++ provides a modular-based structure for the implementation of its simulation models. This feature is specifically helpful when the simulated system is highly complex. As sections of the system architecture can be individually validated, and then put together in a single simulation model, this type of approach makes the implementation process easier. Thus, the development process of the created simulation models was based on such features.

Even though the main focus of this dissertation is a HaRTES switch simulation model, an IEEE 802.1Q model was first implemented. As this was the standard that introduced traffic isolation capabilities on Ethernet switches, there was some interest in comparing its performance with other real-time protocols. Afterwards, a server-based scheduling framework, whose purpose is to handle asynchronous communications in HaRTES, was developed. Since the management of the asynchronous traffic is not made in the FTT master itself, but on a separate structure [2], this framework can be implemented and tested without being included in the HaRTES switch. Furthermore, it can be incorporated in other Ethernet switches. Subsequently, the focus was on completing the HaRTES simulation model.

This chapter addresses the development of the several simulation models implemented in the simulator environment Omnet+. It starts with the description of an IEEE 802.1Q switch module, followed by a hierarchical server-based scheduling module designed for Ethernet switches. The chapter closes with the presentation of the HaRTES simulation model with the server-based framework incorporated, to handle the asynchronous traffic.

5.1 Traffic isolation with IEEE 802.1Q in a Ethernet switch model

The IEEE 802.1Q protocol was originally developed to solve issues in standard Ethernet, in particular, the lack of support for traffic segregation and prioritization. Even though the introduction of distinct queues to separate the traffic classes, with different QoS requirements, provided some levels of traffic isolation, it was not enough for the implementation of priority-based networks with large sets of real-time streams. Nonetheless, there is still some interest in comparing the performance of standard IEEE 802.1Q Ethernet switches and other Ethernet real-time protocols.

In OMNeT++, the majority of the protocols' simulation models, including the standard Ethernet protocol, are included in the INET library. However, the implemented Ethernet switch model does not include the eight priority queues defined by the IEEE 802.1Q standard. In this scope, extensions to the INET Ethernet switch model were conceived in order to add support for traffic segregation, provided by the IEEE 802.1Q protocol.

The the developed framework primary components operate at the output ports of an Ethernet switch. Figure 5.1 depicts an overview of the switch model implemented in INET, an OMNeT++ library. It comprises two essential modules, the *SwitchingUnit*, and the *Ethernet Ports (eth[n])*, where n is the number of ports. The former is responsible for selecting the destination ports of incoming messages based on the information stored in the *macTable* and on the destination address field of received Ethernet frames. The latter one, which functions as both an ingress and egress port, contains the MAC components. It models the reception and transmission of Ethernet frames between network nodes.

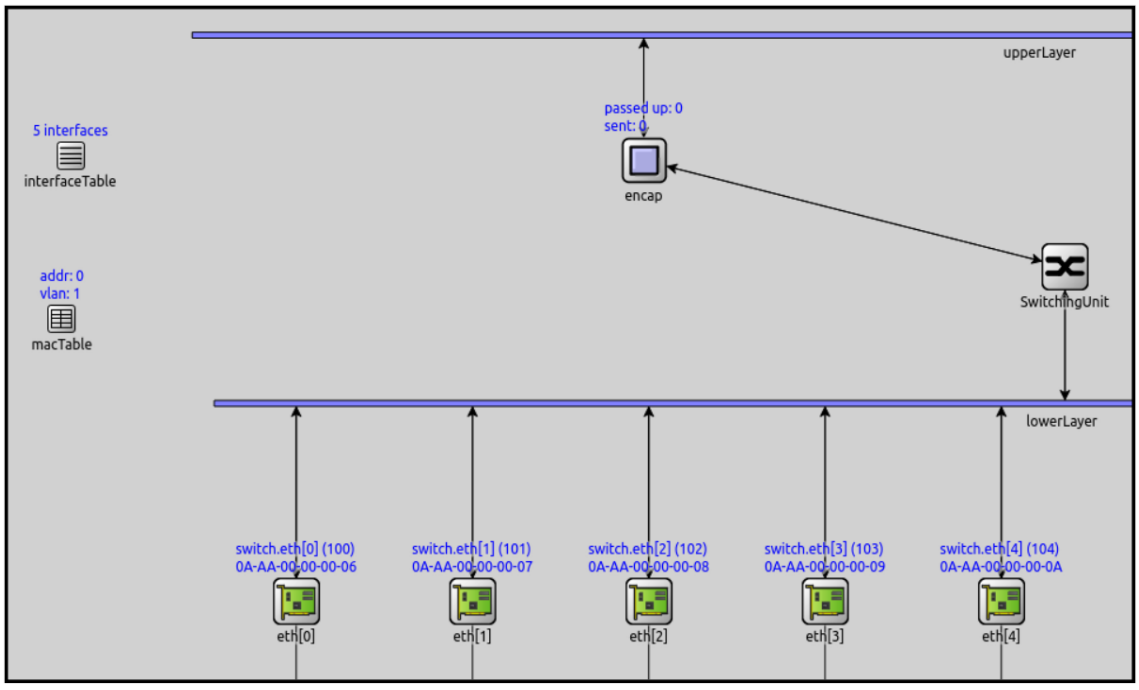


Figure 5.1: Ethernet switch architecture developed by INET.

The implemented queuing structure is imported at the switch output ports. It comprises a frame classifier module (*Processor*) and eight FIFO queues (Figure 5.2). Processed packets reach the *classifier* before being transmitted and are stored in the appropriate queue. For this, the component analyzes the IEEE 802.1Q header, more precisely the priority indicator (PCP field), and enqueues the frame according to this value. The mapping between the PCP value and the different priority queues follows the default IEEE 802.1Q mapping table [12]. In this table, the different traffic classes are numbered from 0 to 7, which represents the lowest and the highest dispatching priority levels, respectively. Priority level 7 is typically associated with the highest priority traffic class, class 7. However, traffic class 0 is associated with priority level 1, while priority level 0 is translated into traffic class 1 due to legacy reasons [14]. As such, priority level 1 represents the lowest priority level of this framework.

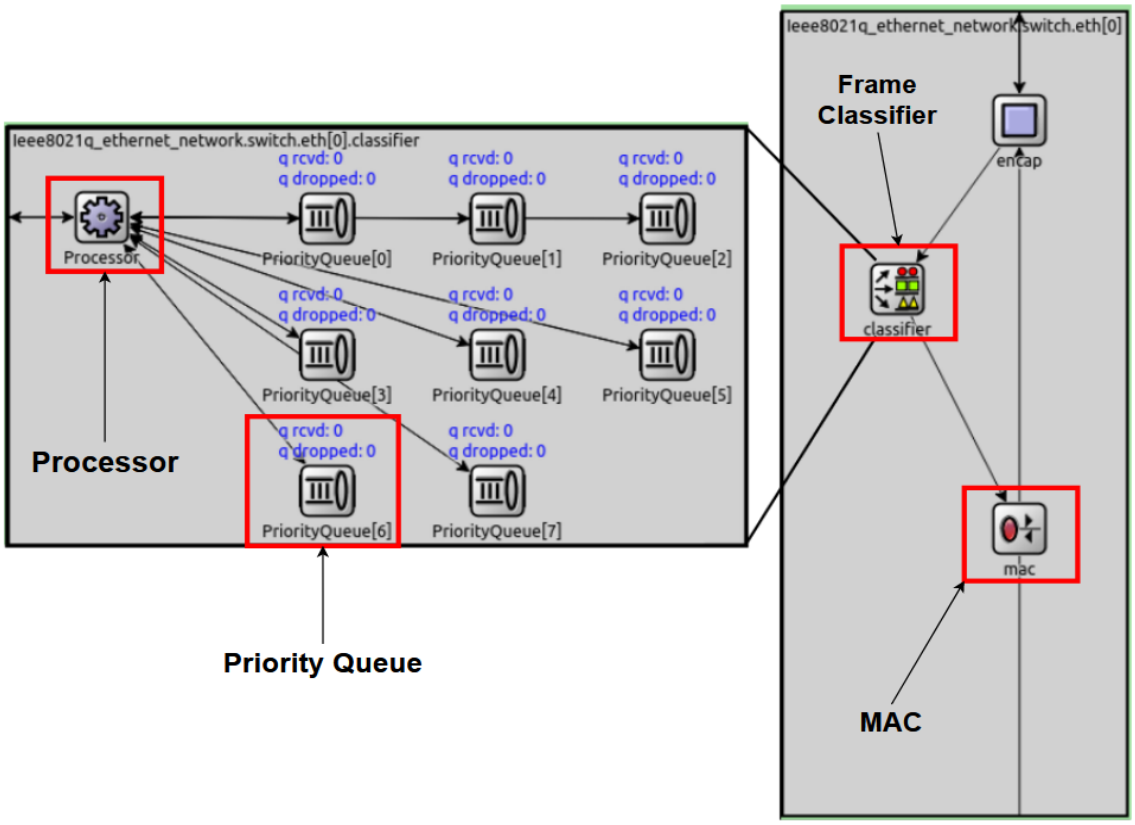


Figure 5.2: Ethernet switch ports with the implemented framework in OMNeT++.

The MAC module, responsible for dispatching the messages, informs the classifier whenever the port can transmit a new packet. As such, when a new request is made, the module searches for the highest priority stored frame and forwards it to the MAC component. As the switching delay for processing a message is almost negligible for the INET components, users can define a processing latency for the classifier to simulate a real word switch. Figure 5.3 illustrates a frame flow through the different modules from ingress to egress for the INET switch with and without the developed framework.

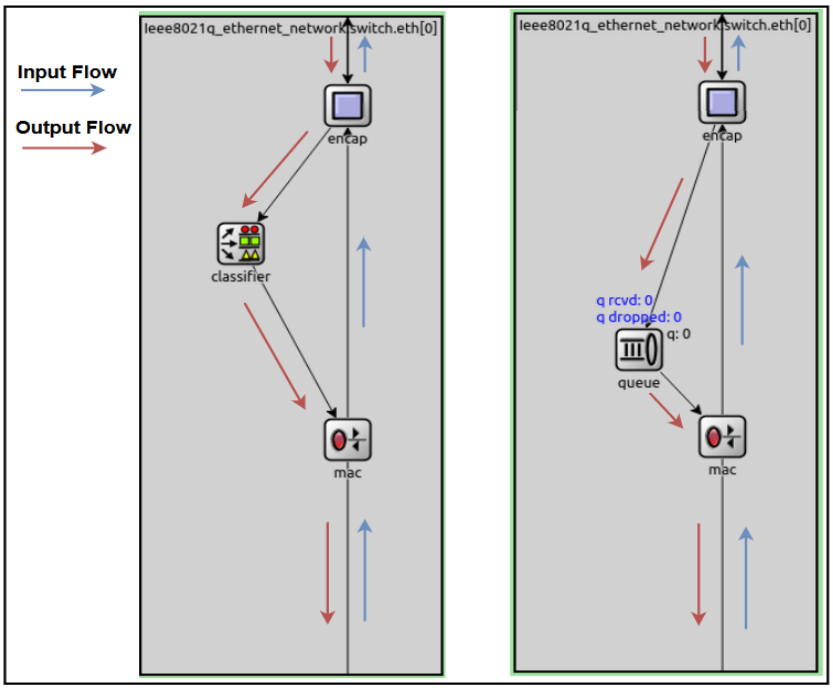


Figure 5.3: Ethernet switch ports: a) with the implemented framework; b) without the framework in OMNeT++.

The methods and variables employed by the module that performs all the logic in the implemented framework, e.g., the *Processor*, are shown in Figure 5.4. This C++ class has two variables: *SWLatency*, which is defined by the user and models the switch latency, and *portFree*, a boolean variable that indicates if the egress port can transmit a packet. Regarding its methods, this class comprises six different functions:

- *initialized*: Called at the start of each simulation. Sets the value for *SWLatency*, i.e., acquires the user-defined latency;

- *handleMessage*: Invoked whenever a new Ethernet message is received or a timer, i.e., self-message, is triggered. The former goes through a classification process and, based on the outcome, is stored in the appropriate priority queue. The latter are used to simulate the switching latency. Whenever a timer is triggered, the *ForwardPacket()* method is called, so that the next highest priority packet can be dispatched;

- *StorePacket()*: Distinguishes priority-based Ethernet frames from the standard ones. Standard Ethernet frames are automatically assigned the lowest priority. On the other hand, for priority-based frames, the method acquires the PCP value from the IEEE 802.1Q header. It then calls the PCP mapping method to store the messages according to their priority levels;

- *PCPmapping*: Maps the PCP value according to the previously described table. It then stores the packet in the appropriate priority queue;

- *ForwardPacket()*: Searches for the highest priority packet and sends it to the MAC module;
- *getNext()*: A public method called by the MAC module informing that the port is free, thus a new transmission can occur.

```
class ProcessUnit : public cSimpleModule
{
    protected:
        // --------------- Variables -------------------- //
        double SWLatency;               // For modeling switch latency
        bool portFree = true;    // Indicates transmissions are occurring
        // --------------- Methods -------------------- //
        // Initialize the model
        virtual void initialize(void) override;
        // Called when a message arrives at the module
        virtual void handleMessage(cMessage *msg) override;
        // Identifies priority-based and standard Ethernet frames
        virtual void StorePacket(inet::Packet *msg);
        // Maps the priorities to the different queues
        virtual void PCPmapping(inet::Packet *packet, int priority);
        // Sends the stored packets to the MAC module
        virtual void ForwardPacket();
    public:
        virtual void getNext(void); // Called by the MAC moduled. To process a stored packet
};
```

Figure 5.4: *Processor* class implemented in OMNeT++.

With the module-based architecture developed in OMNeT++, users can effortlessly remove the implemented structure from the INET Ethernet switch or add it to their own models.

5.2 A server-based scheduling framework for Ethernet switches

Although switched Ethernet networks include several advantages such as low-cost hardware and large bandwidth, standard COTS Ethernet switches were not designed with the required features to support real-time communications. To address these issues, several Real-time Ethernet (RTE) protocols have been proposed. However, despite the mechanisms employed, server-based scheduling in RTE protocols is rather limited, as there is no support for arbitrary server policies or hierarchical composition. In this scope, a server-based framework for Ethernet switches was implemented in the OMNeT++ simulation environment. The implemented architecture focuses on features not available in RTE protocols, specifically the lack of support for hierarchical structuring.

Server-based scheduling is a technique used to control the interference caused by asynchronous messages. As their activation cannot be synchronized and controlled by a scheduler, it can

jeopardize the timeliness of the periodic ones. Thus, such mechanisms are employed to make asynchronous traffic more deterministic. Despite guaranteeing the associated traffic temporal requirements, a significant disadvantage arising from this type of technique is the scheduling analysis, which is done globally. This can be particularly difficult for complex systems. To address these issues, methodologies based on hierarchical scheduling have been proposed. As described in Section 2.1.5, HSF simplifies the systems scheduling analysis by segmenting them into more simplistic subsystems that can be examined individually.

These frameworks are commonly represented by a *tree*, i.e., an association of different components interconnected. When combining multi-level hierarchical scheduling with server-based techniques, each individual component is a server that can be represented as Γ_{y_x}, where y is the hierarchy level and x the component identifier (Figure 5.5). The components at the hierarchy lowest level are designated Leaf servers. They process incoming messages from the streams and consume the system bandwidth. The remaining, known as Branch servers, handle a system bandwidth section and share it through their children.

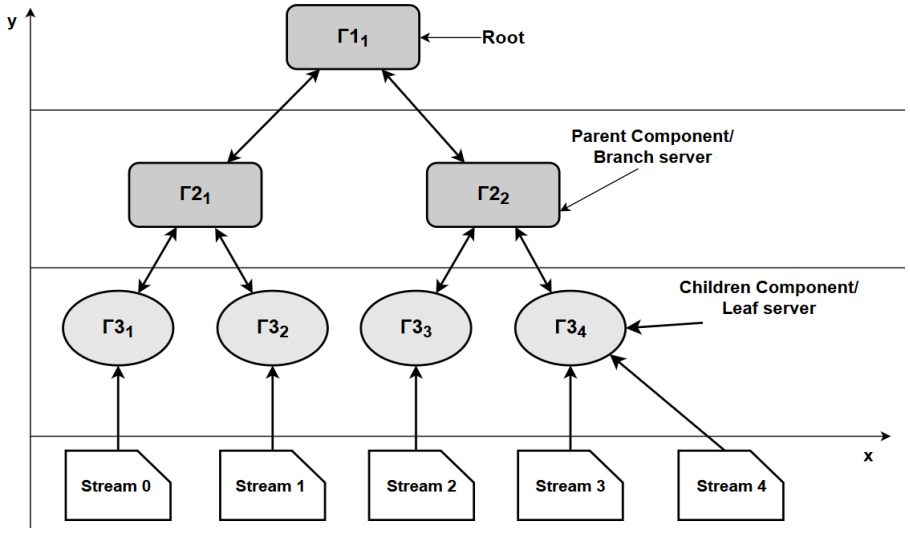


Figure 5.5: Multi-level server-based hierarchy representation.

5.2.1 Switch Architecture

The proposed simulation model makes use of the standard Ethernet switch developed by INET, a library of OMNeT++, (represented by the shaded area), and incorporates the hierarchical server-based structure (represented by the clearer area), as illustrated in Figure 5.6. The implemented framework is entirely isolated from the central switching block created by INET (Figure 5.7),which is mainly responsible, for example, for performing forwarding decisions based on MAC addresses. On the other hand, the hierarchical unit is responsible for dispatching messages to the associated egress port according to the policies defined by server-based framework.

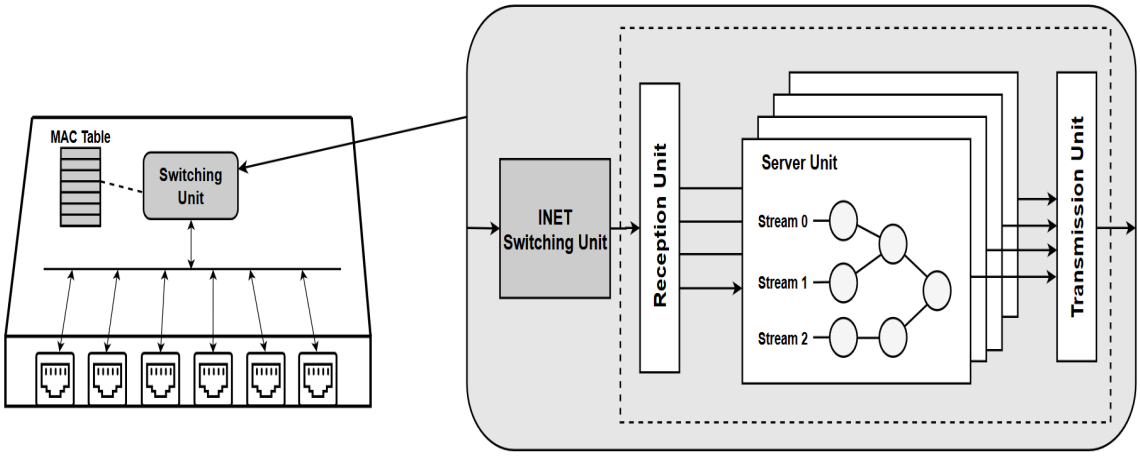


Figure 5.6: Hierarchical server-based framework architecture.

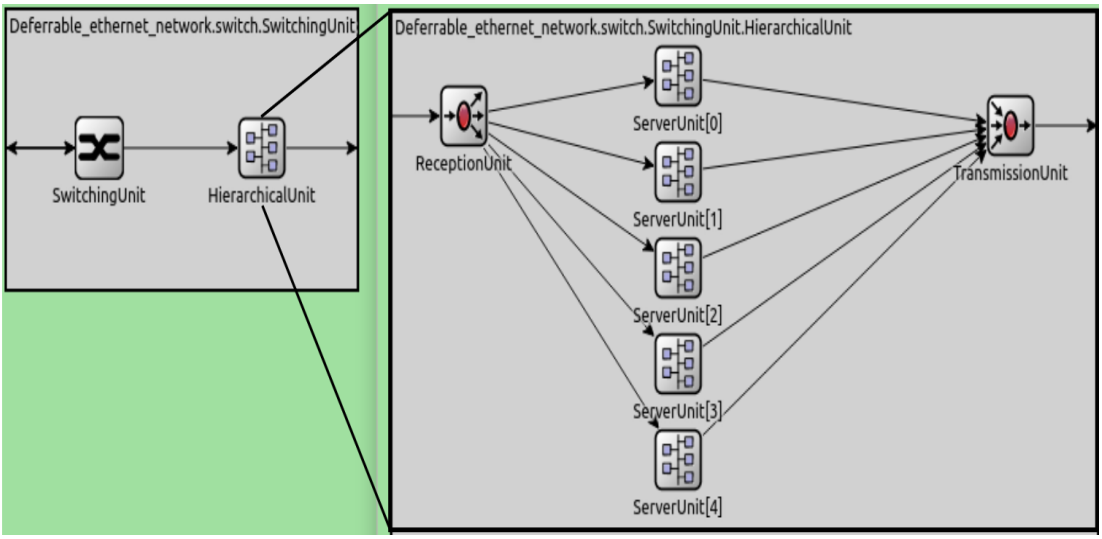


Figure 5.7: The server-based framework employed in the INET Ethernet switch.

As the server-based framework was purposely implemented not to interfere with the switching block, users can easily remove the structure to simulate the standard Ethernet switch behavior or add their own modules to handle the synchronous traffic. However, a classifier module would be required to direct the TT and ET traffic to the correct structures.

5.2.2 Implementation of the hierarchical server-based framework

The *ReceptionUnit* module, directly connected to the switching block, functions as a demultiplexer. It receives packets from the *SwitchingUnit* and, based on results from the switching process, sends them to the appropriate ServerUnit. Within the *ServerUnit*, packets are subsequently stored on different servers, which control the resources available in the network. The *TransmissionUnit*, which acts as a multiplexer, forwards the packets to the respective egress ports. This module also notifies when a given egress port is currently free. Based on the size of a message that is going to be dispatched, the TransmissionUnit calculates the time

required to send the frame together with headers and overheads, and creates a timer. When triggered, the module requests a new packet from the ServerUnit and repeats the process. Note that this component can have several timers, depending on the output ports number.

Figure 5.8 illustrates a sequence chart example of the several transactions that occur between the different blocks. Initially, all egress ports are available hence, the TransmissionUnit automatically forwards the processed packet. While the switch is transmitting the message, two new ones arrive. As the port is currently busy, these are held in the ServerUnit queues. When the transmission is completed, the timer created by the module is triggered, which makes it request a new frame. The ServerUnit scans for the highest priority packet stored and sends it, thus repeating the process.

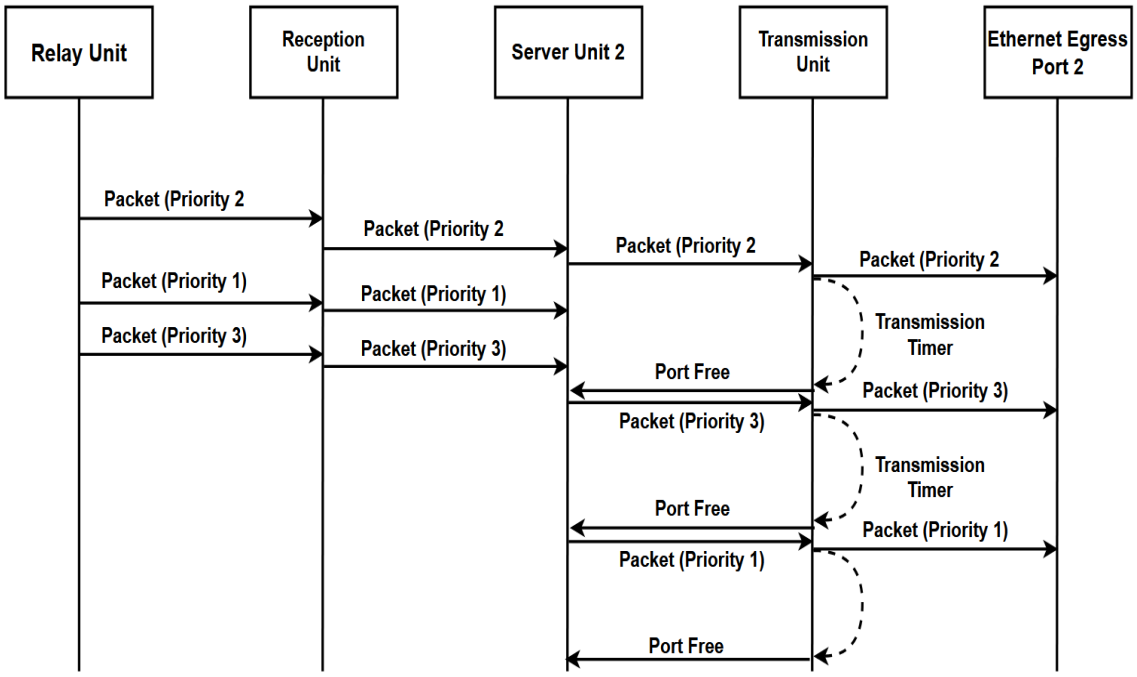


Figure 5.8: Sequence Chart of the transmissions within the switch.

The ServerUnit, specific for each output port, is a compound module that contains the various components of the multi-level server hierarchy. Figure 5.9 describes its internal blocks. It includes the different servers (leaves and branches), the *StreamManagementUnit* and the local scheduler, composed by both the *VerificationUnit* and *SelectionUnit*. Incoming packets are received by the StreamManagementUnit, which then sends them to the appropriate leaf server, thus acting as a demultiplexer. The leaf and branch servers are responsible for controlling the network resources. The former type stores the packets while the latter provides the necessary resources for their processing. Whenever the egress port is free, the scheduler, based on the employed scheduling algorithm, selects which leaf server should operate.

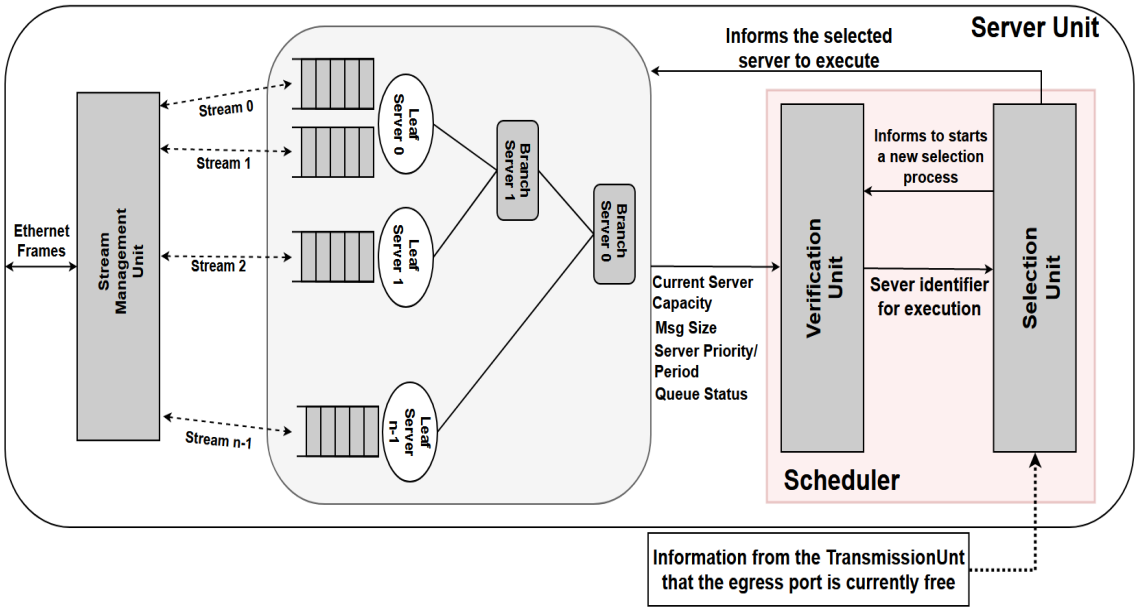


Figure 5.9: Server Unit architecture.

To build the server hierarchy in OMNeT++, the user must specify the number of branch and leaf components used as well as connect them according to the desired structure in the NED file. Figure 5.10 depicts the ServerUnit implementation in the NED file. This compound module has several parameters, some of which define the number of leaf and branch servers employed in the structure (e.g., *LeafComponents* and *BranchComponents*, respectively). When using these parameters, the hierarchy can be built using a vector-based approach, i.e., instead of creating each leaf and branch server individually, these are generated as a standard vector (*branch[BranchComponents], leaf[LeafComponents]*), which can be accessed individually. This approach is specifically helpful when the structure has multiple levels with several components. Note that both parameters have default values. However, these can be changed in the initialization file, when setting up the simulation properties.

The creation of the server-based hierarchy is made when connecting the different components. Similar to other programming languages, the NED language enables the user to use both logical conditions and loops (e.g., *if...else*, *while*), when connecting the modules. By taking advantage of such features, several hierarchies can be implemented and saved in a single file. Then, during the simulation parameterization, when defining the number of leaf and branch servers employed, one of them is utilized. In Figure 5.10, the hierarchy illustrated is used in the simulation when the number of branch servers is one (*BranchComponents == 1*). However, by adopting this sort of approach, dynamic modifications within the structure, e.g., number of servers and their type, are not possible during run-time. Nonetheless, reconfigurations of the existing components, e.g., period or capacity, can be made at any time during the simulations.

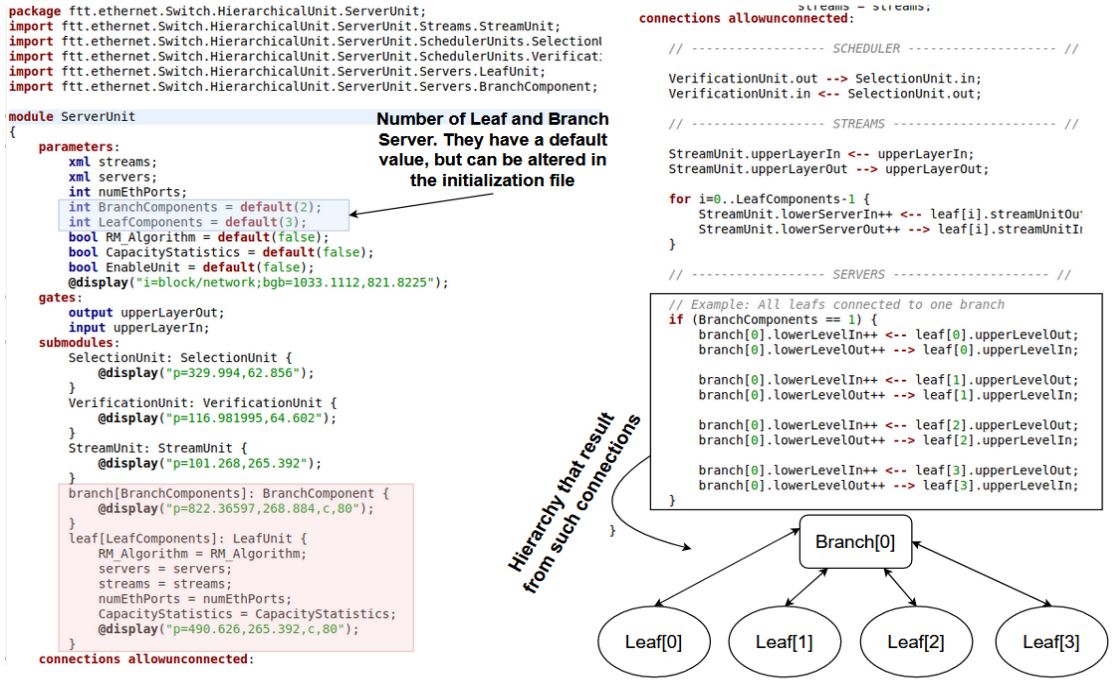


Figure 5.10: *ServerUnit* implementation in OMNeT++ NED file.

5.2.2.1 Stream Management Unit

Within each ServerUnit exists the *StreamManagementUnit*. It is a simple module that associates packets from the different streams to the appropriate leaf servers. Likewise, it dispatches processed packets from those components to the TransmissionUnit so they can be forwarded to the respective end-nodes. This module can also process reconfiguration requests sent by streams to alter some hierarchy parameters, e.g., change stream and server association. The C++ class of this module is depicted in Figure 5.11. The module main variable is the C+ vector *Streams*. It contains the association between streams and servers. The StreamManagementUnit methode are the following:

- *intialize()*: Called at the start of each simulation. Based on the simulation properties (number of nodes, number of each server type), sets the values for the different variables;

- *handleMessage()*: Received packets can have two sources: the leaf servers or the ReceptionUnit. The former is ready for being dispatched by the switch thus are sent to the TransmissionUnit. The latter are stored in the servers;

- *forwardToSever()*: Sends the packet to the appropriate leaf server, based on the mapping between the streams and servers;

- *processStreamReconfigurationPacket()*: Processes a reconfiguration packet that involves the streams (e.g., add, remove, and modify an existing association). Any alteration involves changing the *Stream* vector;

- *processServerReconfigurationPacket()*: Acts as a validator method for the servers' reconfiguration packets. Packets whose destination server does not exist in the hierarchy are discarded. Calls the *informLeafComponent()/informBranchComponent()* methods for non discarded packets;

- *informLeafComponent()/informBranchComponent*: Invoked whenever reconfiguration packets pass the validation process. Sends the content of the packet to the server;

- *parseXMLFile()*: Reads an XML file with the association between the servers and the streams. Fills the *Stream* vector based on the file content (Table 5.1).

Table 5.1: Properties stored in the *StreamManagementUnit*.

Parameter	Description	Value
Stream Id	Stream Identifier	$[0, 2^{32} - 1]$
Server Id	Leaf server NED identifier	$[0, 2^{32} - 1]$

```
1  class StreamUnit : public cSimpleModule
2  {
3      int BranchComponents;   // Number of Branch Servers
4      int LeafComponents;     // Number of Leaf Servers
5      int EthernetStreams;    // Number of Ethernet stream
6
7      // Vector with Streams informations
8      vector<vector<int>> Streams; // nx1 vector, where n is the number of streams
9                                   // Ex: Streams[3][0] -> gives associated server for stream 4
10     protected:
11         // Initialize module
12         virtual void initialize() override;
13         // Module received a message
14         virtual void handleMessage(cMessage *msg) override;
15         // Send packet to server
16         virtual void forwardToServer(inet::Packet* msg);
17         // Process reconfiguration packet with stream alteration
18         virtual void processStreamReconfigurationPacket(inet::Packet *msg);
19         // Process server reconfiguration packet (check if destination server exists)
20         virtual void processServerReconfigurationPacket(inet::Packet *msg);
21         // Send information from reconfiguration packet to the branch server
22         virtual void informBranchComponent(int serverId, int capacity, int priority, double period);
23         // Send information from reconfiguration packet to the leaf server
24         virtual void informLeafComponent(int serverId, int capacity, int priority, double period);
25         // Get association between servers and streams from an XML file
26         virtual void parseXMLFile(cXMLElement *root);
27 };
```

Figure 5.11: *StreamManagementUnit* C++ class.

5.2.2.2 Leaf Component

The leaf components are compound modules that process incoming messages whenever they have enough resources. These servers parameters are initialized using information from a specific XML file (Figure 5.12) that contains their period, capacity, priority (for static priority scheduling), and hierarchy parameters (x and y, Γ_{yx}). As the XML file holds the properties for several leaf and branch servers, the first two parameters (e.g., *Type* and *ID*) are used as filters. This way, each server only gets the content referred to their type and ID. Each leaf component has an associated FIFO queue to store the messages sent by streams (Figure 5.13), which are transmitted whenever the ServerUnit scheduler assigns the server to execute.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <ServerStats>
3    <Server>
4      <Type>Leaf</Type>
5      <ID>0</ID>
6      <y>3</y>
7      <x>1</x>
8      <Capacity>3000</Capacity>
9      <Priority>2</Priority>
10     <Period>0.0001</Period>
11   </Server>
12 </ServerStats>
```

Figure 5.12: Example of an XML file with the properties of a leaf server.

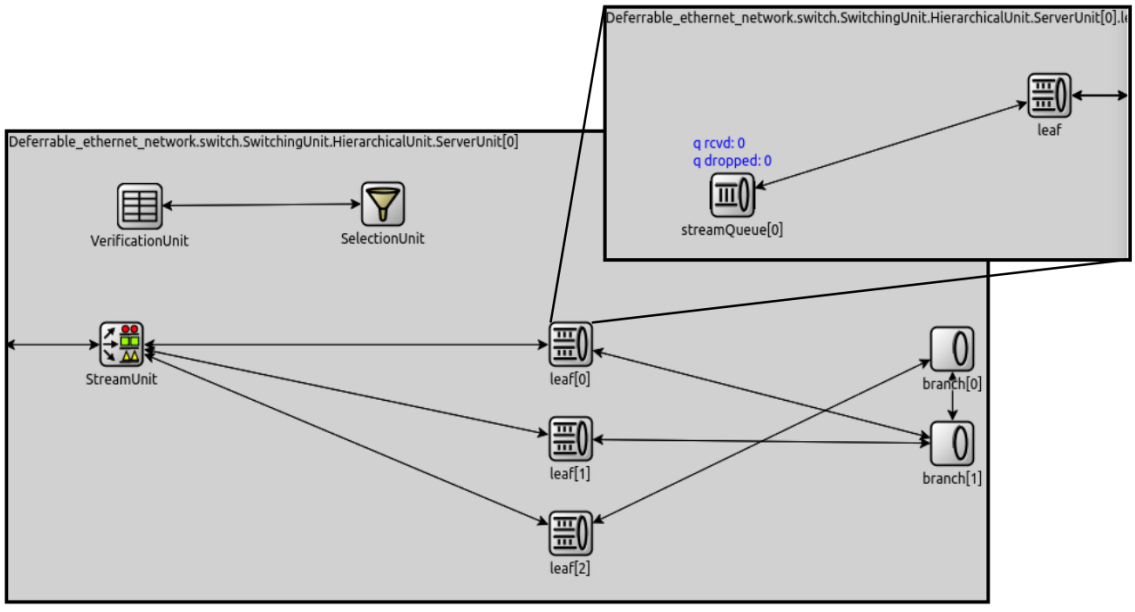


Figure 5.13: ServerUnit implementation in OMNeT++.

As the components are based on the Deferrable Server (DS) scheduling algorithm, their capacity budget is replenished periodically. Thus, each server has a local timer that, when triggered, restores the capacity of the component. As previously mentioned, streams can transmit specific requests to modify some properties within the hierarchy. If the new configuration involves the server parameters exclusively, the leaf components can directly process these requests.

The leaf component C++ class includes several methods and variables, some of which are only used to print useful information to the user or store data for analysis. Therefore, instead of presenting all the module methods, only the most important are described (Figure 5.14):

- *intialize()*: Invoked at the start of each simulation. Initializes the module variables, i.e., sets their values based on the simulation parameters. It also calls a specific method to read the XML file and get the server properties;

- *handleMessage()*: Received packets can be self-messages, which are handled by the *handleSelfMessage()* method, or Ethernet frames. For the latter type, this function calls the *storePacket()* method;

- *handleSelfMessage*: Called whenever the replenishing timer is triggered. The server capacity budget is recharged with the maximum value defined, and a new timer is created. The *updateVerificationUnit()* method is invoked afterwards;

- *updateVerificationUnit()*: This method updates the VerificationUnit with the server current parameters(current capacity, stored packets, size of the largest stored packet). The information is then used to select the next server to operate;

- *storePacket()*: Whenever an Ethernet packet is received, this method is invoked. It stores the packet in the queue. If the FIFO queue is full, i.e., $storedPackets = queueSize$, the received packet is discarded;

- *processPacket()*: This method is invoked by the SelectionUnit whenever the server is selected to operate. It gets the next stored packet (i.e., top of the FIFO queue) and sends it to the StreamManagementUnit so it can be dispatched by the switch. Afterward, the server properties are updated (e.g., capacity decreased, number of stored packets decreased, size of the largest stored packet updated), thus the *updateVerificationUnit()* is invoked. The server also sends a message to the parent with the resources spent to process the packet.

- *reconfigureParameters()*: Invoked by the StreamManagementUnit whenever a server reconfiguration packet passes the initial validation process. This packet content(e.g., priority, capacity, period) is sent to the server. Based on this information, the server assesses if the modifications are feasible (e.g., $capacity \geq maxPacketSize$, $period > 0$).

```
class LeafComponent : public cSimpleModule
{
    VerificationUnit *verificationUnit; // Pointer to the verificationUnit module
    cMessage *ReplenishTimer;           // Replenishing Timer
    struct ServerParameters{            // Parameters of the server
        int y;
        int x;
        int capacity;
        int priority;
        double period;
    };
    ServerParameters self;
    int queueSize;                  // Maximum number of packets stored in queue at the same time
    int currentCapacity;            // Server Capacity at the moment
    int storedPackets = 0;          // Number of Packets Stored
    int maxPacketSize = 0;          // Size of the largest stored Packet
    (...)
    protected:
        virtual void initialize() override;     // Initialize the module
        virtual void handleMessage(cMessage *msg) override; // Module received a message
        virtual void handleSelfMessage(cMessage *msg);
        virtual void updateVerificationUnit();
        virtual void storePacket(inet::Packet* msg);
        (...)
    public:
        virtual void processPacket(); // Called by the SelectionUnit. Processes a stored packet.
        virtual void reconfigureParameters(int capacity, int priority, double period); // Called by the
      StreamManagementUnit. Send the reconfiguration packet contents to the server.
};
```

Figure 5.14: Section of the leaf components C++ class.

5.2.2.3 Branch Component

Each branch component provides a section of the network resources (i.e., bandwidth) to their children so they can process messages. Therefore, whenever a connected leaf server processes a packet, the branch capacity decreases by the same amount. Note that it is not only the immediate branch that has to reduce its budget. All the servers in the *path* from the *root* to the leaves must decrease their capacity. Hence, starting from the lowest level of the hierarchy, the components send specific messages informing the following level the quantity to decrease.

Similar to leaf components, branch parameters are also initialized using an XML file. Furthermore, as they are also based on Deferrable Server scheduling algorithm, their capacity is limited and can be depleted. Thus, within each component exists a periodic timer to replenish the capacity budget of the server.

As the branch component implementation is very similar to leaf servers, only some of its main methods are explained:

- *handleSelfMessage()*: Function used for replenish the capacity budget of the server;

- *updateCapacity()*: Invoked whenever a child leaf server processes a packet. The component sends a message with the capacity spent in that process. The branch server then decreases the same amount from its own capacity;

- *updateVerificationUnit()*: Called whenever the server parameters are altered (e.g., capacity replenishment). Informs the VerificationUnit about the current server parameters (e.g., current capacity).

5.2.2.4 Scheduler

The scheduler, composed by both *VerificationUnit* and *SelectionUnit* simple modules, is responsible for selecting the appropriate server to execute based on the system current parameters. These parameters are stored in two tables that exist within the VerificationUnit, one for each type (Table 5.2 and Table 5.3). Both tables are updated whenever occurs a process that modifies the components' current properties, e.g., replenishing of their capacities, processing a packet.

Table 5.2: VerificationUnit table with all the simulation branch servers' properties.

Branch Component	Capacity (Bytes)	Period (s)	Priority	Parent Component
$\Gamma 1_0$	5000	0.001	-	-
$\Gamma 2_0$	3000	0.005	1	$\Gamma 1_0$
...	...	...	...	...
Γy_x	...	...	...	...

Table 5.3: VerificationUnit table with all the simulation leaf servers' properties.

Leaf Component	Capacity (Bytes)	Period (s)	Priority	Size of the largest stored Packet	Number of stored Packets	Parent Component
$\Gamma 2_1$	20	0.01	2	0	0	$\Gamma 1_0$
$\Gamma 3_0$	1000	0.02	2	1000	1	$\Gamma 2_0$
$\Gamma 3_1$	1400	0.03	1	700	2	$\Gamma 2_0$
...	...	...	...	...	...	...
Γy_x	...	...	...	...	...	...

Based on both tables, and the algorithm illustrated in Figure 5.15, the VerificatioUnit assesses which current leaf components can process their stored messages. As such, starting from the first entry of Table 5.2, the module analyses if any leaf components satisfy the following conditions:

1. Leaf component current budget $\geq$ Largest stored packet size.

2. Number of stored packets > 0.

Note that the first condition implies that, for certain situations, resources may not be correctly used. As the scheduler verifies the largest stored packet, instead of the next in the queue,

servers, with enough capacity budget to process smaller packets, can be prevented from operating. Nonetheless, this type of approach makes the scheduling analysis easier.

If any of the two conditions fail, the module advances to the next entry. Otherwise, it starts examining the successful component parent. The VerificationUnit knows the associations between servers through the *Parent Component* (Γy_x) column in both tables. The y section of this parameter (hierarchy level) can be either null, for the root component, or bigger than zero, thus pointing to the parent level. Hence, the module analyses if the following server has enough resources:

- Branch component current capacity $\geq$ Leaf largest stored packet.

If this condition is satisfied, the following branch in the *path* is tested, otherwise, the process is stopped, and the module repeats the previous steps for the next leaf table entry. If all the branch servers, including the root, satisfy the preceding condition, all the previously examined components are added to a vector that holds the available servers for execution. Note that the leaf table entries are not ordered by priorities thus, the VerificationUnit must always repeat the complete process for all leaf components.

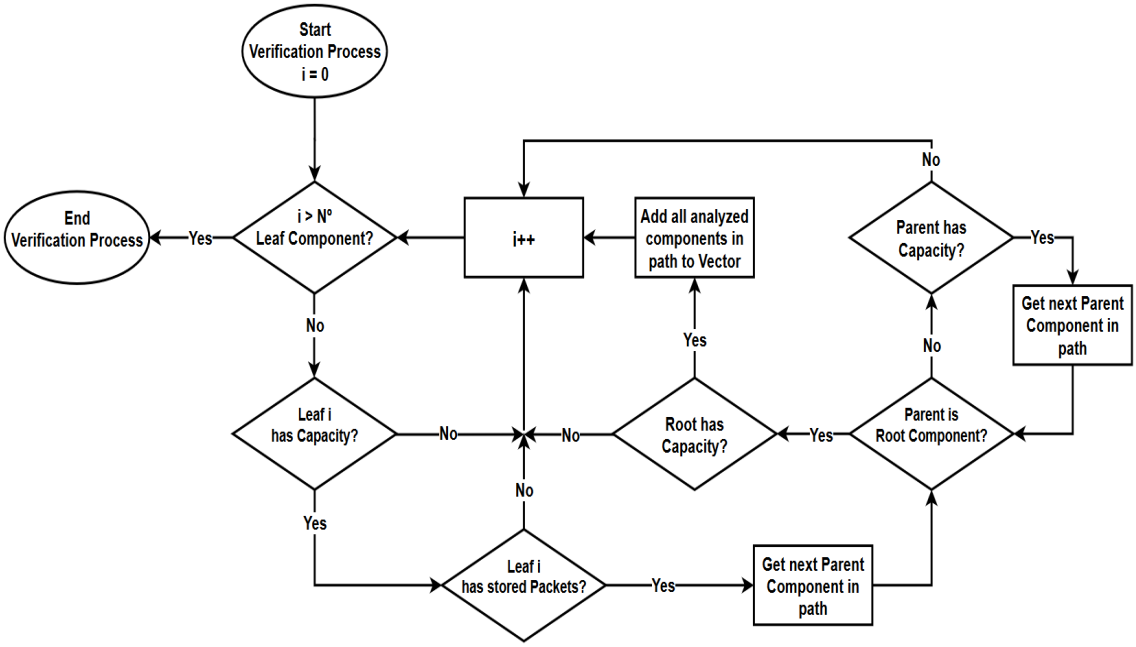


Figure 5.15: VerificationUnit processing algorithm.

Using the vector with all available servers, the VerificationUnit selects the one with the highest priority to execute using the algorithm depicted in Figure 5.16. Starting from the hierarchy second level, the module selects, for each level, the highest priority component stored in the vector. This process is based on the RM algorithm or static priorities. Upon reaching the first leaf component, the module informs the SelectionUnit which server should operate.

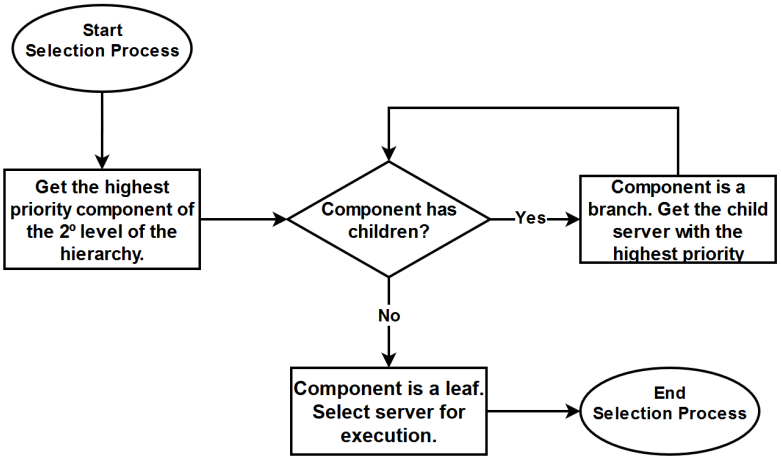


Figure 5.16: VerificationUnit selection algorithm.

Based on the information from the VerificationUnit, the SelectionUnit requests a server to process one of its stored packets (i.e., invokes the previously described leaf component method, *processPacket()*). This structure is also responsible for starting the scheduling process. Whenever a transmission is completed, the TransmissionUnit informs this module that the port is currently free, which then informs the SelectionUnit to start the scheduling process.

In summary, the VerificationUnit is responsible for the scheduling algorithm itself while the SelectionUnit starts the scheduling process and administers its results. Note that all the previously described processes could have been made in a single scheduler module. However, because OMNeT++ supports a modular-based structure, by creating two distinct modules with different functions, the system becomes simpler. To better understand the previous mechanisms, a simple example is shown (Figure 5.17).

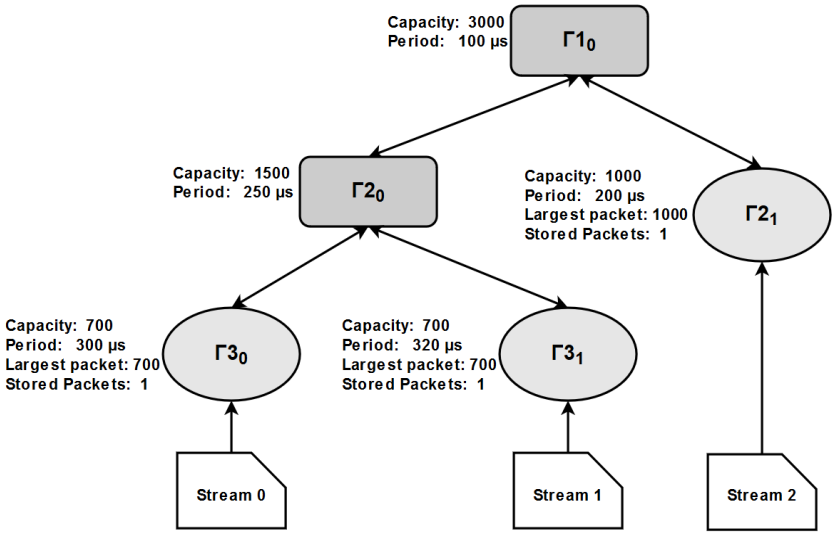


Figure 5.17: Example of multi-level server Hierarchy.

Figure 5.17 presents a multi-level server hierarchy example with each component current properties. In this example, the leaves' budget is equal to the stream frame maximum size.

Thus, only one frame can be sent in each period. Based on the properties displayed in the figure, and considering that all the egress ports are currently available, all three leaf servers can presently process their packets. Consequently, the VerificationUnit creates a 3D vector with every component, as depicted in Figure 5.18.

Starting from the *Hierarchy Level* column second position and, based on the RM algorithm, the VerificationUnit determines that the server with the highest priority is $\Gamma 2_1$. As that server has no children, it denotes that it is a leaf thus, it is selected to execute. Following the sending completion, the TransmissionUnit requests another packet. A new 3D vector is created, except this time, component $\Gamma 2_1$ is not added. Consequently, for the hierarchy second level, component $\Gamma 2_0$ is selected as the one with the highest priority, followed by $\Gamma 3_0$ for the third. This server can immediately execute as it has no children. The process is later repeated, and the last server, $\Gamma 3_1$, is selected for execution.

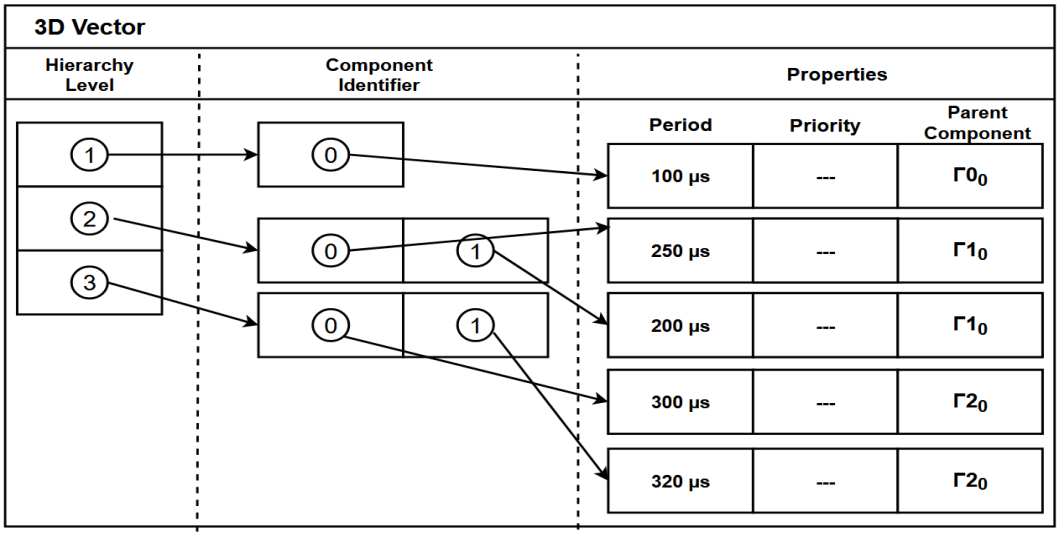


Figure 5.18: Resulting 3D Vector from the first verification process.

5.3 HaRTES simulation model

In the scope of the FT4FTT [59] project, which aims to improve reliability and flexibility in Destributed Embedded Systems (DES) via fault tolerance mechanisms, a simulation model of HaRTES for OMNeT++ was implemented, whose purpose is to evaluate the efficiency of such mechanisms in a convenient and faster environment. Both the HaRTES switch and FTT-slaves simulation models (Figure 5.19), designing by Knezic et al. [60], were specifically created to study a procedure that improves the FTT-compliant nodes synchronization by replicating the trigger messages and transmitting them multiple times. Therefore, when originally designed, the authors focused specifically on the switch synchronous subsystem. Consequently, both asynchronous and Non real-time (NRT) communications were not supported. The HaRTES simulation model, implemented in the scope of this dissertation, is based on the original Knezic et al. model and completes it by introducing the server-based framework presented in the previous section (Section 5.2) to handle both the asynchronous and non-real-time traffic.

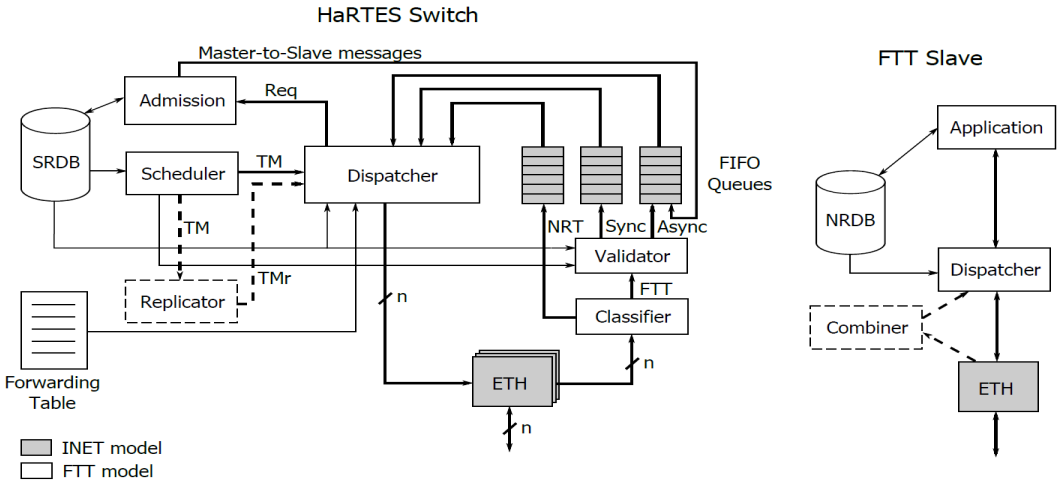


Figure 5.19: HaRTES switch and FTT-slave architectures for OMNeT++ (*from Knezic et al. [60], 2014).*

The general architecture of the proposed model for OMNeT++ is illustrated in Figure 5.20. The HaRTES switch model comprises three distinct types of modules. The ones that were developed by INET, the FT4TT modules that remained untouched, and lastly, the newly added components to process the asynchronous traffic as well as an updated FT4TT modules version. As the original HaRTES model was implemented in 2014, with the current version of OMNeT++ and INET, it was required to update several of its original components to recent software versions. Furthermore, several modifications were also necessary to incorporate asynchronous communication subsystem.

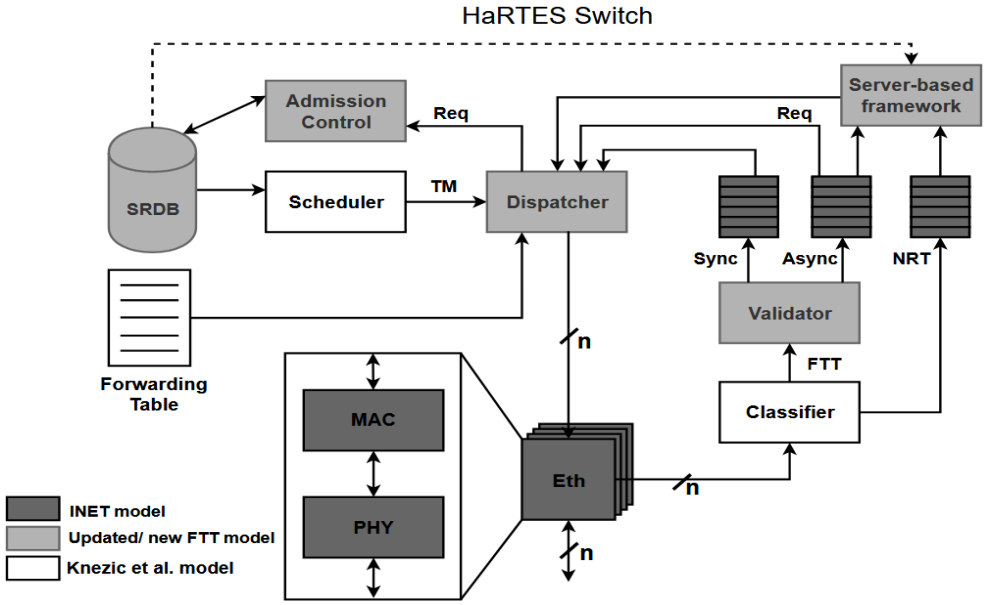


Figure 5.20: Implemented HaRTES switch architecture for OMNeT++

The INET modules include the FIFO Queues employed to store the different messages processed by the switch. They also comprise an Ethernet Interface that contains a MAC and

EtherEncap components responsible for transmitting Ethernet messages between the physical and data link layers. The remaining modules, based on the blocks described in Section 3.3.1, are presented in the following segments.

5.3.1 HaRTES switch

An overview of the HaRTES switch simulation model in the OMNeT++ IDE is depicted in Figure 5.21.

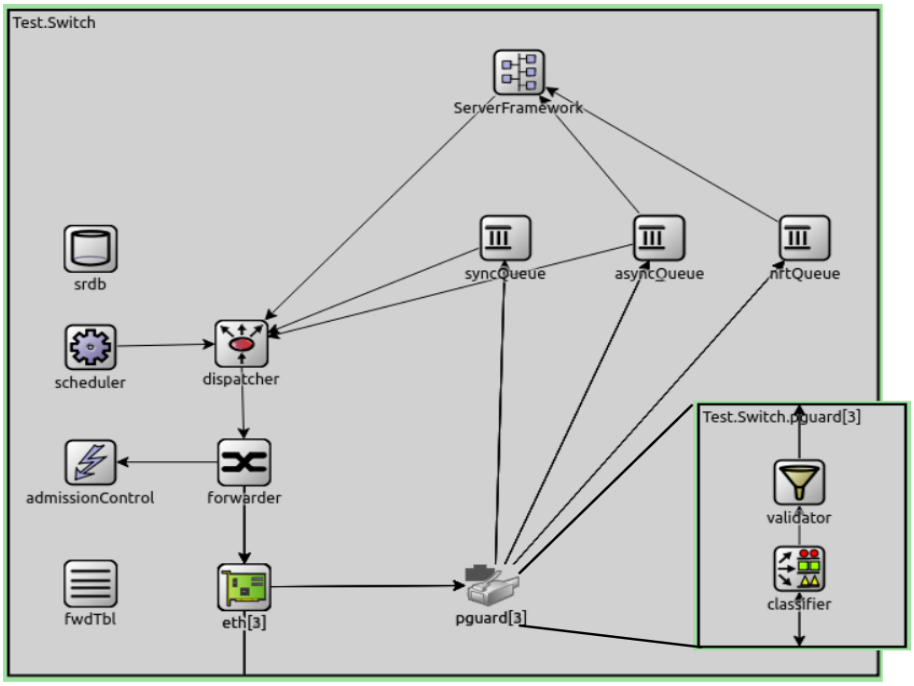


Figure 5.21: HaRTES switch in OMNeT++.

5.3.1.1 FTT Master

The FTT Master, responsible for the management and decision making within the switch, comprises the *srdb*, *scheduler*, *admissionControl*, *dispatcher* and *forwarder* simple modules.

The SRDB module contains all the properties regarding the system synchronous and asynchronous streams. The information stored in this component is essential for building the scheduler, validating the packets as well as dispatching them to the appropriate egress ports, and distribute the resources throughout the server-based framework. At the beginning of every simulation, this module is initialized using the information from a specific XML file. Table 5.4 contains the kind of data stored in the file that is registered during the initialization.

Table 5.4: Properties stored in the SRDB/NRDB module.

Parameter		Description	Value
General	StreamType	Real-time traffic class	0 (TT), 1 (ET)
	StreamReq Id	Stream Identifier	$]0, 2^{32} - 1]$
	Size	Frame length, in bytes. Excludes all the 802.3 headers and FCS as well as other FTT overheads	$]46, 1500]$
	Deadline	Frame absolute deadline	$[0, 2^{32} - 1]$ in ECs
	Priority	Frame absolute priority. Higher values equals to higher priority levels	$]1, 2^{32} - 1]$
	Publisher	List of streams' ingress ports	—
	Subscribers	List of streams' egress ports	—
TT	Period	Periodicity of frames	$]0, 2^{32} - 1]$ in ECs
ET	Mit	Minimal inter arrival time between frames	$]0, 2^{32} - 1]$ μs
	Server	Associated Deferrable server identifier	$[0, 2^{32} - 1]$

The Scheduler is responsible for periodically construct the EC schedule. As such, within the module, the trigger messages are created and subsequently sent to the *dispatcher*. The the elementary cycle duration is defined by the user in the initiation file. This value can range between $[1, 2^{32} - 1]$ ms, with a default value of 1ms. Note that the LTM and TAT duration (Figure 3.1) is, by default, 10 μs each, thus giving enough time for the FTT nodes to receive the TM messages and process them. As it was implemented, this module uses the attributes in the SRDB to constructs the EC schedules based on the RM or EDF algorithm (selected by the user in the initialization file).

The *admissionControl* module comprises both QoS Manager and Admission Control blocks. This component is responsible for managing possible changes in the SRDB. These modifications are requested by the FTT slaves through Slave-to-Master asynchronous requests named *update slave requests (Req)*. As these are asynchronous packets, their transmission is not limited to any window. Reconfiguration requests include the modification of a registered synchronous or asynchronous stream (e.g., add or remove subscribers, alter the period/tmit, message size, and priority) or the server reservation modification (e.g., alter period, capacity budget, and priority). Note that these are only basic reconfiguration services as no assessment algorithm was implemented to validate the reconfiguration requests.

The FTT master dispatcher comprises three distinct simple modules: (i) the *fwdTbl*, (ii) the *dispatcher* and (iii) the *forwarder*. The first one, the forwarding table, is a module that contains the association between the egress ports and the streams subscribers. Instead of standard MAC addresses, the forwarding rule is based on *Stream Ids* with a publisher/subscriber model. Consequently, the table content stems from a specific XML file that is scanned through the simulation initialization. Figure 5.22 presents an XML file example used to initialize the table. The first value corresponds to the Ethernet port (it can range between $[0, 2^{32} - 1]$) whereas, the second is the stream identifier.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <!--Egress Port :: Stream Id-->
3 <Ports>
4     <Port No="0">1</Port>
5     <Port No="1">2</Port>
6     <Port No="2">3</Port>
7     <Port No="3">4</Port>
8 </Ports>
```

Figure 5.22: Example of an XML file with the forwarding table contents.

The *dispatcher* and *forwarder* are responsible for dispatching messages to the end-nodes of the network. The latter module uses the content from both the SRDB and the forwarding table to transmit packets to the streams' publishers. It also broadcasts the TM at the start of every EC, upon constructed by the scheduler, and sends the Slave requests to the *admission Control*. The former simple module requests the creation of a new EC schedule, gets the packets from the Sync FIFO queue, and notifies the server-based framework the initiation and duration of the asynchronous window. As these distinct processes can only occur in specific time slots, the dispatcher contains two periodic timers, which, when triggered, notify the initiation of a new elementary cycle and a new asynchronous window, respectively.

5.3.1.2 Switching structures

The remaining switching blocks include both **Input** and **Output Ports**, **Memory Pool**, and the **Server-based framework** that handles the asynchronous and non-real-time traffic.

Both ingress and egress ports of the Ethernet switch are from the INET library, whose function is to transmit and receive packets between the network components as described in Section 5.1. Upon the reception of a new frame, the *classifier* module classifies the packet into synchronous, asynchronous and non-real-time. Non-real-time packets are directly stored in non-real-time FIFO queue, whereas the remaining ones go through a validation process. Based on the contents from the SRDB, and the current EC schedule, the *validator* simple module, assesses if the synchronous messages are scheduled in the current elementary cycle. Regarding asynchronous packets, this module examines if the publisher stream is registered in the SRDB. Synchronous and asynchronous messages that pass the validation process are enqueued in the appropriate FIFO queues, while others are discarded.

In order to incorporate the server-based framework described in Section 5.2 in the HaRTES switch, some minor adjustments were required. As this structure processes both real and non-real time traffic, a new Background server was introduced in all the ServerUnits. This component has the lowest priority of the hierarchy and is directly connected to the root component (Figure 5.23). It has infinite bandwidth but is limited to the resources available throughout the asynchronous window.

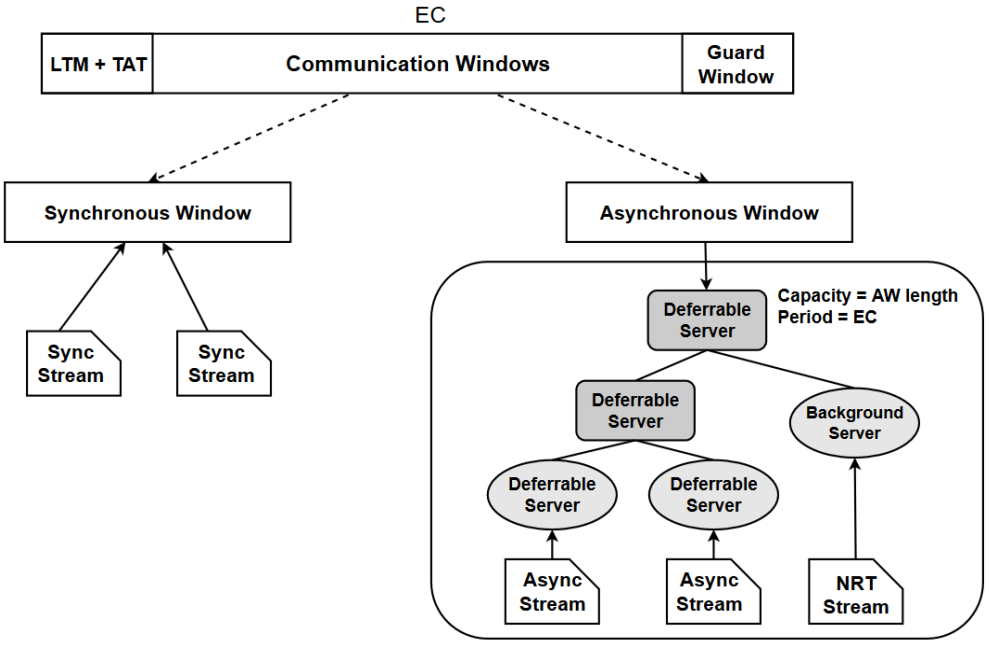


Figure 5.23: HaRTES elementary cycle structure.

The asynchronous and non-real-time traffic can only be transmitted through the duration of the asynchronous window. To guarantee traffic isolation and prevent these messages from interfering with the broadcasting of the trigger messages, some measures were introduced. The first one was to limit the resources available for the servers. Consequently, the hierarchy top component has a capacity budget equal to the asynchronous window duration, which is replenished at the start of every elementary cycle.

The second measure guarantees that asynchronous or non-real-time transmissions can only initiate if they are completed before the start of the next elementary cycle. It is important to highlight that the FTT protocol already implements mechanisms that prevent overruns between the traffic classes. These include a Guard Window introduction at the end of the asynchronous window, in which no new transmissions can start. This window is normally as large as the largest packet in the network, thus ensuring that all the egress ports are available for transmitting the TM. However, as the previous statement is not always guaranteed (i.e., the size of the Guard Window can be smaller than the network packets), the framework verifies the following conditions:

1. Transmissions can only start during the asynchronous window;

2. Transmissions must cease before the start of the next EC.

The Guard Window usage has, however, a significant inconvenience as an available bandwidth portion is not used. As such, since the framework continuously checks the shown conditions before any transmission, the Guard Window can be completely removed, improving the switch efficiency.

To achieve this procedure, at the start of the asynchronous window, the *dispatcher* module notifies the ServerUnits the moment it starts, its length (LAW), and the guard window (GW) duration. Therefore, when selecting the next server to execute, the framework scheduler first verifies if the transmission starts within the asynchronous window. Then, if so, assesses if the time required to transmit the largest and highest priority stored frame together with the maximum switching latency (ϵ) is shorter than the available time in the window (Equation 5.1):

$$t_{sch} + C_i + \epsilon \leq t_{AW} + LAW + GW \tag{5.1}$$

where t_{sch} is the instant the scheduling process starts, C_i the frame transmission time, and t_{AW} the time instant when the asynchronous window started. Note that, when the guard window is removed, GW = 0.

If the condition is verified, the packet is transmitted. Otherwise, no other messages are forwarded throughout the rest of the window. This means that, for certain circumstances, the resources are not optimally used, as lower priority messages could still be transmitted at these intervals if they are large enough. However, this type of approach eases the schedulability analysis.

The initialization of leaf servers' properties (period, capacity, priority) was modified to acquire the information from the SRDB or from an XML file as it was formerly implemented, based on the user decision. By using the SRDB format, the switch assigns the necessary resources to each reservation. The servers' parameters can also be altered by Slave-to-Master modification requests.

5.3.2 FTT Compliant nodes

The architecture of FTT compliant nodes is depicted in Figure 5.24. Alike the HaRTES switch model, the INET modules to transmit and receive Ethernet frames (*Eth*) are reused.

The *nrdb* module is the FTT slave equivalent of SRDB for the HaRTES switch. Consequently, its data is referred exclusively to the node itself. Similar to the switch procedure, this module content is initialized using an XML file whose parameters are shown in Table 5.4.

The *dispatcher* module receives the TM transmitted by the switch, decodes it, and, based on the EC schedule, generates the synchronous messages. Regarding the asynchronous subsystem, this module has a local timer for the creation of asynchronous packets. As this class of traffic is non-deterministic, the timer can range between [mit, 3×mit]. The structure of both messages is constructed by the dispatcher using the NRDB contents (destination, source, size, and so forth) whereas, the data is provided by the Application (*app*). The dispatcher delivers both real-time and non-real-time frames to the application module, targeted by other network nodes. The application can also create Slave-to-Master requests to alter some NRDB contents and, consequently, the SRDB if feasible. These are forwarded directly to the switch by the dispatcher.

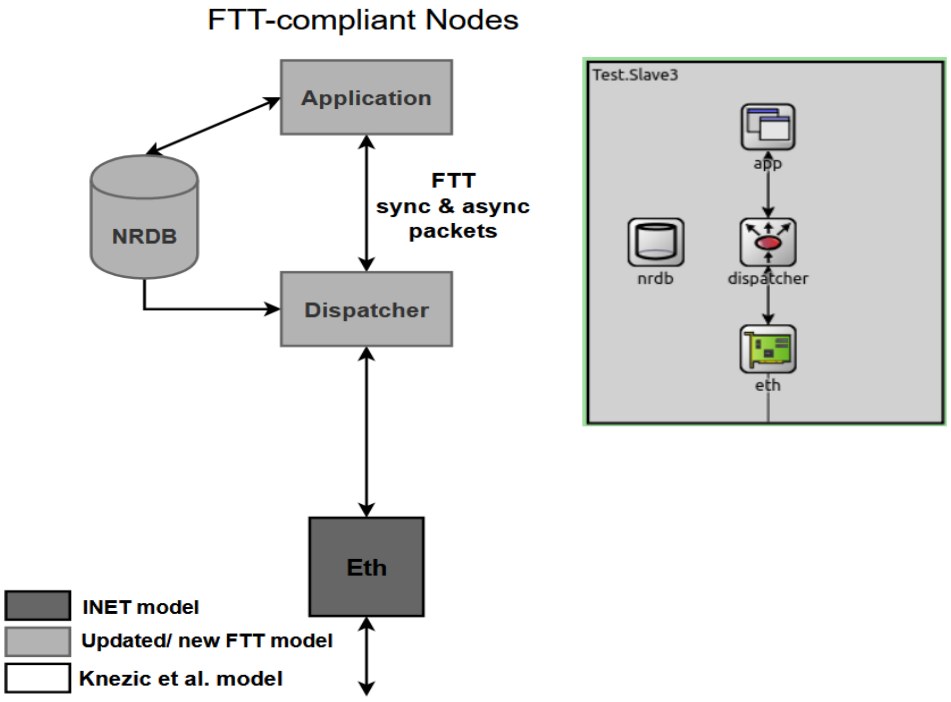


Figure 5.24: FTT compliant nodes architecture.

CHAPTER 6

Validation of the implemented OMNeT++ simulation models

Contents

*T*his chapter presents the experiments conducted to validate the correctness of the simulation models described in Chapter 5. It starts with the individual validation of the IEEE 802.1Q protocol model, followed by the evaluation of a hierarchical server-based framework simulation model. The chapter concludes with the assessment of the HaRTES OMNeT++ model with the server-based framework incorporated.

6.1 Validation of the IEEE 802.1Q simulation model

As addressed in Section 2.3, standard Ethernet did not provide the capabilities to isolate traffic different classes with different QoS requirements. These features were introduced with the IEEE 802.1Q standard, which extended the IEEE 802.3 Ethernet frame with a specific priority identifier field, and introduced different priority queues at the switch output ports. In Section 5.1, an OMNeT++ simulation model that provides such features to standard Ethernet switches was described.

The following segment presents the experimental results from the implemented simulation model as well as their analysis. The experiments focus is the correctness of the control algorithm and the model traffic segregation capabilities, which classify the messages according to different priorities. The experimental setup used to conduct the different experimental scenarios is illustrated in Figure 6.1. It consists of an Ethernet switch and five nodes. The switch links were configured for 100 Mbit/s and the internal latency ϵ is about 5 μs[1].

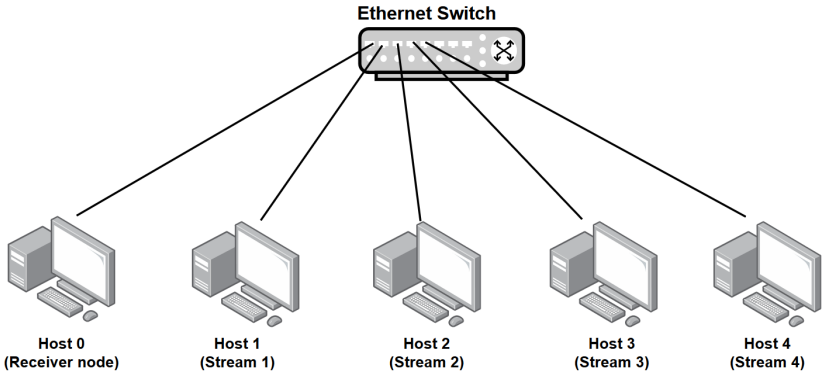


Figure 6.1: Experimental setup used to validate the IEEE 802.1Q modeled switch.

In order to validate the implemented IEEE802.1Q simulation model, two experiments were conducted: Exp1, in which a homogeneous set of synchronous messages was employed, to assess if the control algorithm is correctly processing the frames according to their priorities, and Exp2, in which the model was tested using a heterogeneous traffic set, to evaluate the model performance in a more realistic situation. The the generated traffic properties are presented in Table[2] 6.1.

Table 6.1: IEEE 802.1Q: Generated traffic properties for Experiment 1.

Experiment	Node	Traffic Type	Priority (PCP)	Payload (Bytes)	$T_i/T_{mit_i}(\mu s)$	Destination Node
Exp1	1	Synchronous (S1)	4	500	350	0
	2	Synchronous (S2)	3	500	350	0
	3	Synchronous (S3)	2	500	350	0
	4	Synchronous (S4)	1	500	350	0
Exp2	1	Synchronous (S1)	4	1200	400	0
	2	Synchronous (S2)	3	1200	400	0
	3	Asynchronous Sporadic (A3)	2	900	350	0
	4	Asynchronous Aperiodic (A4)	1	1000	-	0

Note: The payload size does not include the frame headers and other overheads from the OSI Data link layer.

6.1.1 Experiment 1 - Homogeneous traffic set

In the first experiment, all nodes are perfectly synchronized in time. Consequently, all streams' packets reach the switch at the same instant, thus being easier to verify the order in which

[1]This value is based on several Ethernet switches datasheets ([61], [62], [63]), which, for 100 Mbit/s links, have an local latency ranging from 3 to 7 μs.

[2]For this analysis scope, the streams' deadlines and offsets (for the synchronous type), were not considered.

they are dispatched. All streams generate synchronous packets with a transmission time (C_i) of 42 μs (Table 6.1). Figure 6.2 illustrates the latency values measured for each stream and Table 6.2 presents the statistical breakdown of the observed data.

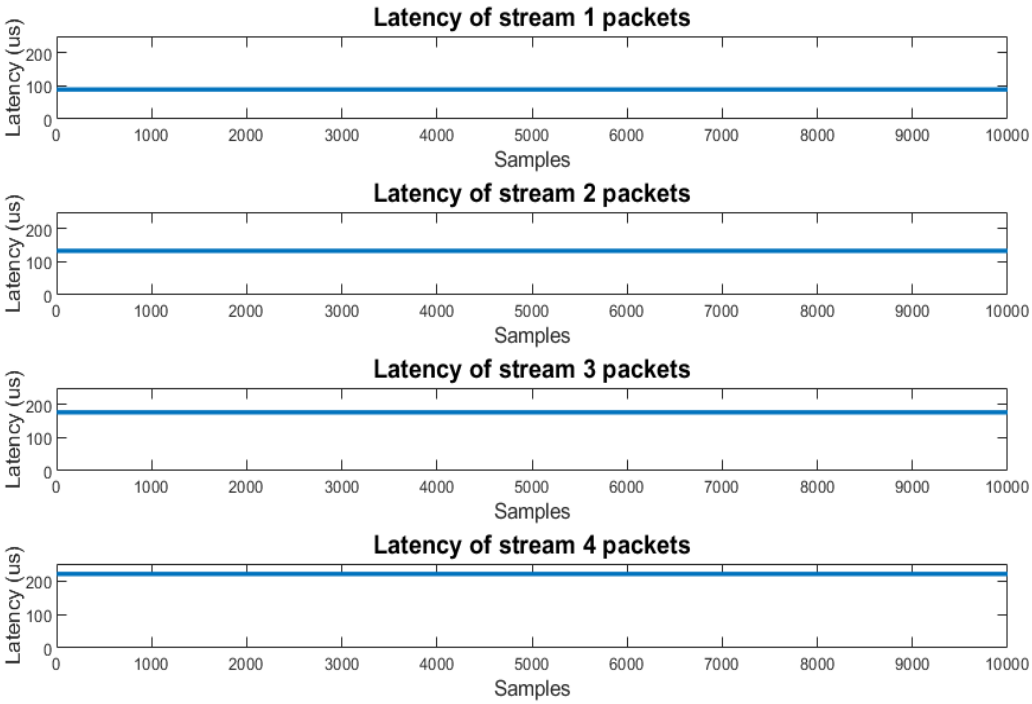


Figure 6.2: IEEE 802.1Q: Latency measurements for Experiment 1.

Table 6.2: IEEE 802.1Q: Statistical values for Experiment 1.

Stream	Latency (μs)		
	Min	Max	Mean
S1	90.07	90.07	90.07
S2	138.34	138.34	138.34
S3	185.97	185.97	185.97
S4	233.61	233.61	233.61

Total number of samples: 10000

The obtained results demonstrate the correct traffic prioritization by the implemented model as the streams' latency values increase the lower their priorities, with the observed minimum of 90 μs for the highest priority stream (S1) and the maximum latency of 233 μs for the lowest priority stream (S4). As the streams are perfectly synchronized, there is always a 47 μs interval between the reception of consecutive frames. This value accounts for the transmission duration (42 μs), together with the switch local latency (5 μs). As the sum of all the transmission times is always lesser than the streams' periods, there is never interference between two consecutive sets of created messages.

6.1.2 Experiment 2 - Heterogeneous message set

In this experiment, the generated traffic set now includes both TT and ET messages. Consequently, frames can be received at different time instants and thus, blocking can now occur, and distinct interference patterns can emerge on the egress link. Note that both circumstances only happen because frames cannot be preempted while being transmitted. Table6.1 describes the generated message set.

Frames from stream 1, the higher priority stream, can be immediately dispatched upon reaching the switch or, for the worst-case situation, are delayed because of blocking caused by the largest frame full transmission from a lower priority stream ($B_{A4} = C_{A4}$). It is important to highlight that theoretically, messages from S2 would result in the highest interference. However, for this case, as this stream is synchronized with S1, such a situation never occurs. Thus, the latency of stream 1, L_{S1}, can range between:

$$L_{S1} = \begin{cases} min = 2 \times C_{S1} + \epsilon \\ max = (2 \times C_{S1} + \epsilon) + B_{A4} \end{cases}$$

For stream 2, all messages experience interference from the highest priority stream (I_{S1}) because they are synchronized in time. They may also suffer blocking from lower priority frames if the same conditions applied to stream 1 occur:

$$L_{S2} = \begin{cases} min = (2 \times C_{S2} + \epsilon) + I_{S1} & , I_{S1} = C_{S1} + \epsilon \\ max = (2 \times C_{S2} + \epsilon) + I_{S1} + B_{A4} \end{cases}$$

For sporadic messages, the maximum latency results from blocking caused by the complete transmission of the lower priority aperiodic message, and interference from the two synchronous ones stored at the switch output queues. On the other hand, the minimum value occurs if the packet reaches the switch and all the queues are empty and the ports free:

$$L_{A3} = \begin{cases} min = 2 \times C_{A3} + \epsilon \\ max = (2 \times C_{A3} + \epsilon) + I_{S1} + I_{S2} + B_{A4} \end{cases}$$

Lastly, for the aperiodic stream, the worst-case situation occurs if the frames of all four streams reach the switch at the same instant. In such circumstances, the latency is considerably increased by the duration of three higher priority complete transmissions:

$$L_{A4} = \begin{cases} min = 2 \times C_{A4} + \epsilon \\ max = (2 \times C_{A4} + \epsilon) + I_{S1} + I_{S2} + I_{A3} \end{cases}$$

The subsequent table, Table 6.3, presents the theoretical ranges expected for the messages' latency values as well as the experimental measurements of 100000 generated packets for each stream.

Table 6.3: IEEE 802.1Q: Latency values for Experiment 2.

Stream	L_{theo} (μs)		L_{exp} (μs)		
	Min	Max	Min	Max	Mean
S1	201.64	283.96	201.36	283.96	195.86
S2	304.96	387.28	303.97	386.60	212.97
A3	153.64	442.60	153.33	441.23	313.33
A4	169.64	455.60	169.36	453.25	210.69

The results demonstrate a small discrepancy between some theoretical and experimental values, which accounts for 1 μs for both maximum and minimum ranges. Nonetheless, all the measurements are within the expected theoretical ranges, with the highest latency with the highest latency (453 us) observed for the lowest priority stream, and the lowest latency value (283 us) for the highest priority stream.

6.2 Validation of the server-based scheduling model

This section details the conducted experiments that evaluate the implemented server-based scheduling framework, described in Section 5.2, and the results analysis. When evaluating the model, three central features were considered: (i) the control algorithm correctness, (ii) the possibility to make several online reconfigurations to the hierarchy and (iii) the capability to handle multiple streams with different requirements. The experimental setup, used throughout the different scenarios, is illustrated in Figure 6.3. It comprises four nodes and an Ethernet switch with the configured hierarchical structure depicted. Each stream priority is set according to the associated servers' static priorities. For both experiments, the switch links operate at 100 Mbit/s, and the device latency ϵ is considered to be 5 μs.

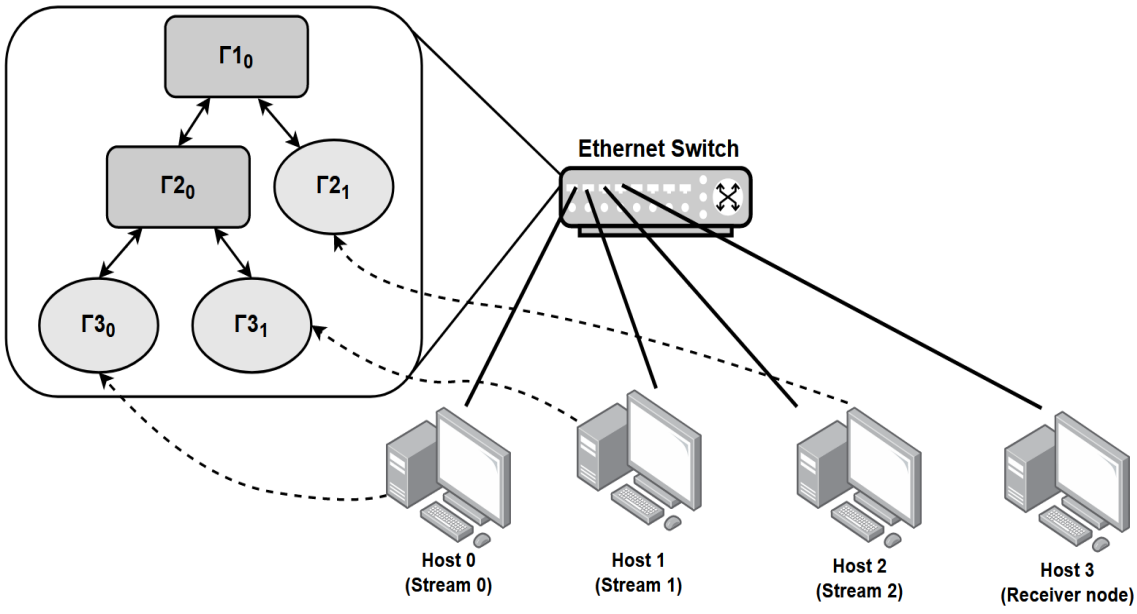


Figure 6.3: Server-based scheduling: Experimental setup.

Altogether, two experiments were made: Exp1, in which the model was tested using a homogeneous synchronous traffic set, to assess the server-based scheduling algorithm, i.e., if the servers' execution is based on hierarchy priorities and if the structure limits the streams' bandwidth, and Exp2, that simulates a typical real-time network with several asynchronous streams, whose latency values are measured.

6.2.1 Experiment 1 - Assessing the model control capabilities

In the first experimental scenario (Exp1), three streams generate synchronous messages with a transmission time (Ci) of 114 μs. As the streams are perfectly synchronized in time, their frames reach the switch at the same instant. The servers, employed to manage the streams' bandwidth, have capacity budgets that allow only one frame transmission per replenishing period (Table 6.5). Each flow is handled by a single component at the hierarchy second or third level. In order to assess how the resources are distributed within the framework, the servers' periods were configured to operate in three distinct modes. Table 6.4 and table 6.5 lists the streams and servers properties for each mode, respectively.

Table 6.4: Server-based scheduling: Generated traffic properties for all three modes of Experiment 1.

Source Node	Traffic Type	Associated Server	Priority	Payload (Bytes)	$T_i(\mu s)$	Destination Node
0	Synchronous (S0)	$\Gamma 3_0$	2	1400	1000	3
1	Synchronous (S1)	$\Gamma 3_1$	1	1400	1000	3
2	Synchronous (S2)	$\Gamma 2_1$	3	1400	1000	3

Table 6.5: Server-based scheduling: Server properties for Experiment 1.

Server	Priority	Capacity Budget (Bytes)	Period (μs)		
			Mode 1	Mode 2	Mode 3
$\Gamma 1_0$	-	4500	300	300	4000
$\Gamma 2_0$	1	3000	600	600	600
$\Gamma 2_1$	2	1400	1000	2000	2000
$\Gamma 3_0$	2	1400	1000	2000	2000
$\Gamma 3_1$	1	1400	1000	2000	2000

Mode changes were performed online, during simulation, with nodes issuing a reconfiguration request to the framework. The parameters of these configuration requests are presented in Table 6.6. Figure 6.4, illustrates the observed inter-arrival values for each stream. Figure 6.5 depicts the servers' capacity throughout the experiment for Mode 1. Lastly, Table 6.7 presents a statistical breakdown of the observed latency and inter-arrival timings for each stream.

Table 6.6: Server-based scheduling: Requests to the server-based framework.

Reconfiguration requests				
Server	Operation	Producer	Capacity (Bytes)	Period (μs)
$\Gamma 2_1$	Mod. Period	S2	1400	2000
$\Gamma 3_1$	Mod. Period	S0	1400	2000
$\Gamma 3_1$	Mod. Period	S1	1400	2000
$\Gamma 1_0$	Mod. Period	S0	4500	4000

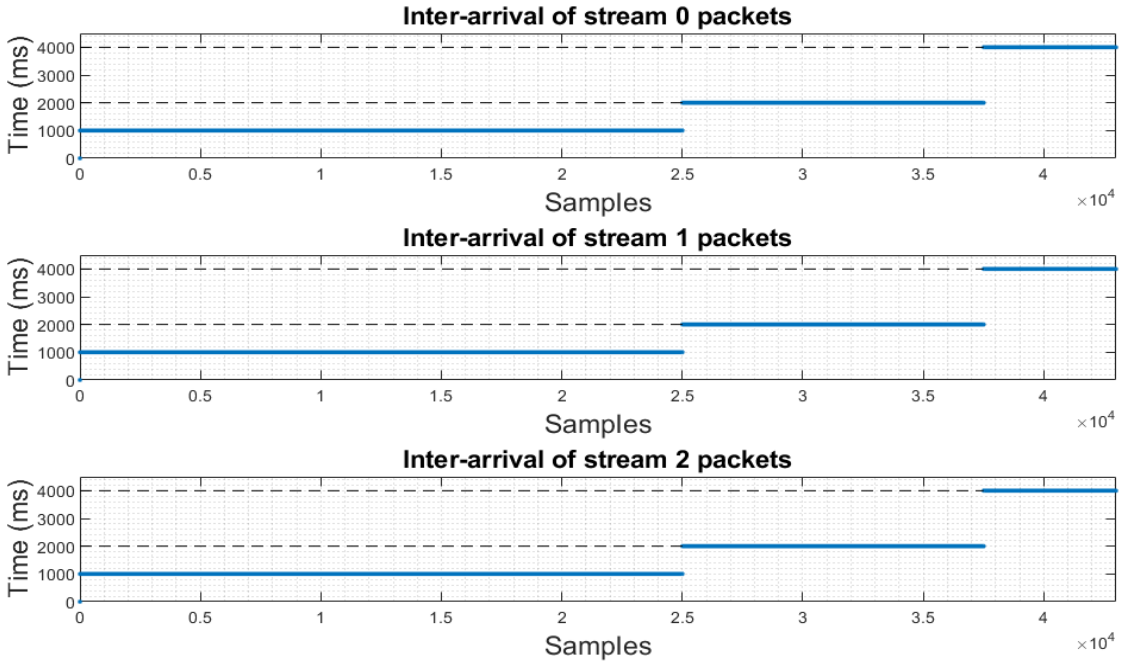


Figure 6.4: Server-based scheduling: Inter-arrival timings for Experiment 1.

Table 6.7: Server-based scheduling: Statistical values for Experiment 1.

Stream	Latency (μs) (Mode 1)			Mean Inter-arrival (μs)		
	Min	Max	Mean	Mode 1	Mode 2	Mode 3
S0	351.01	351.01	351.01	999.97	1999.92	3999.84
S1	469.33	469.33	469.33	999.97	1999.93	3999.84
S2	232.69	232.69	232.69	999.96	1999.92	3999.85

Total number of samples: 75000

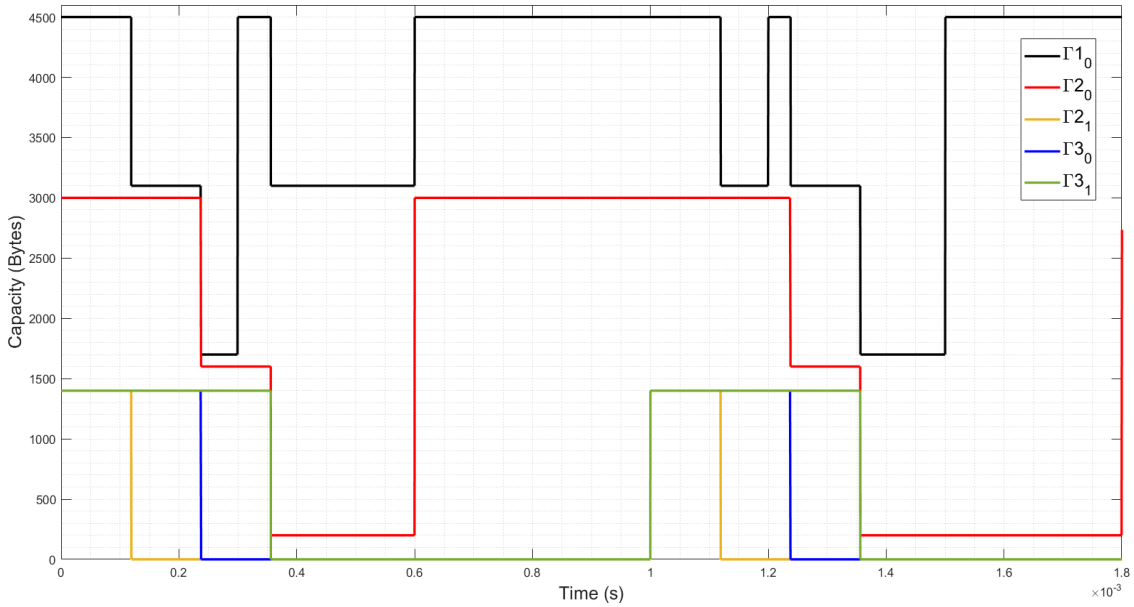


Figure 6.5: Server-based scheduling: Servers' capacity over time for Experiment 1 (Mode 1).

The results from this experiment show that the framework is processing the packets according to the desired algorithm, with higher priority frames being dispatched first. As all packets reach the switch at the same time, the lower priority are always delayed because of interference, with a 119 μs value for stream 0 and 237 μs for stream 1 (Table 6.7). This situation is enhanced by the results of Figure 6.5, which confirms that $\Gamma 2_1$ is the first to execute, followed by $\Gamma 3_0$ and then $\Gamma 3_1$. Note that the presented latency values are only regarding Mode 1 because, for the remaining two modes, the frames' latency values are influenced by interference and blocking between messages and the servers' replenishing periods.

The experimental results also demonstrate that the servers' capacity budget limit the streams' bandwidth consumption. Note that such restrictions are not only imposed by the associated components but also by higher level components. As the servers can only process one frame per period, whenever their periods increased, the inter-arrival timings followed this growth, always coinciding with the servers' periods, as shown in Figure 6.4 and Table 6.7.

6.2.2 Experiment 2 - Realistic network simulation

For the second experiment, all flows are asynchronous with their properties presented in Table 6.8. Similar to Exp1, the servers were configured to process only one frame with the available capacity budget. Figure 6.6 illustrates the latency histogram for each stream, and Table 6.9 presents a statistical counterpart of the obtained values.

Table 6.8: Server-based scheduling: Generated traffic properties for Experiment 2.

Source Node	Traffic Type	Associated Server	Priority	Payload (Bytes)	$Tmit_i(\mu s)$	Destination Node
0	Asynchronous Sporadic (A0)	$\Gamma 3_0$	2	1100	650	3
1	Asynchronous Sporadic (A1)	$\Gamma 3_1$	1	1000	700	3
2	Asynchronous Sporadic (A2)	$\Gamma 2_1$	3	1200	650	3

Table 6.9: Server-based scheduling: Latency values for Experiment 2.

Stream	L_{theo}[3](μs)		$L_{exp}(\mu s)$		
	Min	Max	Min	Max	Mean
A0	185.64	371.28	184.69	365.32	198.66
A1	169.32	368.28	168.69	316.67	303.97
A2	201.64	291.96	200.69	287.01	218.77

Total number of samples: 100000

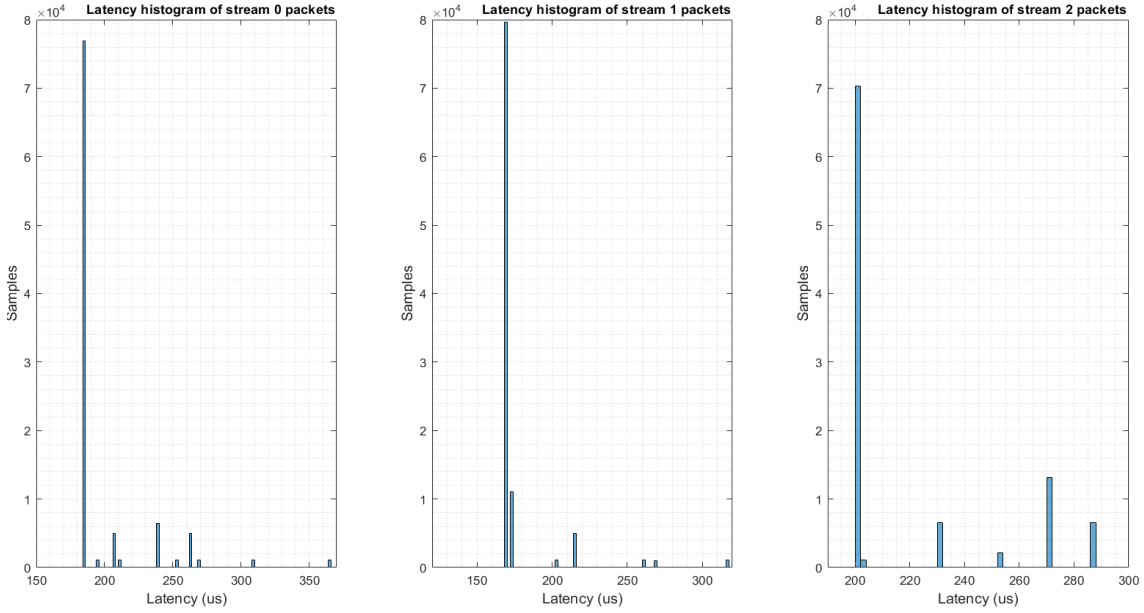


Figure 6.6: Server-based scheduling: Latency Histograms in μs.

The results show that the maximum observed values were all below the analysis upper bounds, with a maximum latency of 365 μs for stream 0. This difference can result from the worst-case situation not being generated. Nevertheless, the minimum latency measured for all streams is moderately close to the expected results, with less than 1 μs of difference between values.

[3]The theoretical analysis was based on the same methodology described in section 6.1.2.

6.3 Comparing the performance of the frameworks in Ethernet switches

In the previous sections, two distinct structures to manage real-time traffic in Ethernet switches were individually validated. The former assigns distinct priorities to the traffic classes, thus improving their separation at the switch ports. The latter, employed to handle the asynchronous traffic, provides resource control as well as dynamic reconfigurability to the system. The following segment presents several experiments in which those frameworks, together with a standard Ethernet switch model, implemented by INET, were compared.

Altogether, two different scenarios were examined: the first, in which all the nodes respect the established stream properties, the second contains some "badly behaved nodes" that do not respect the negotiated reservations. The experiments purpose is to evaluate the different simulation models performance for resource distribution and traffic segregation. As such, two different metrics are measured and compared between experiments: (i) the number of packets dropped by the switch relative to each stream and (ii) the minimum and maximum latency values for each stream.

The experimental setup employed to assess the models is illustrated in Figure 6.7. It comprises an Ethernet switch connected to seven different nodes, with six distinct flows and one receiver node. The purpose of having only one receiver node is to force both blocking and interference between messages so that increases in latency values are easily discerned. Table 6.10 holds the streams' parameters. Figure 6.8, illustrates the hierarchical structure employed in the server-based switch. These are based on Deferrable servers, and their budgets are configured to allow only one frame transmission per period (Table 6.11). For the different experiments, the switch links are configured to 100 Mbit/s, and its internal latency is approximately 5 μs. Furthermore, to simulate a real switch with limited memory, the queues, at the output ports, were parameterized to store at most 100 packets at a time.

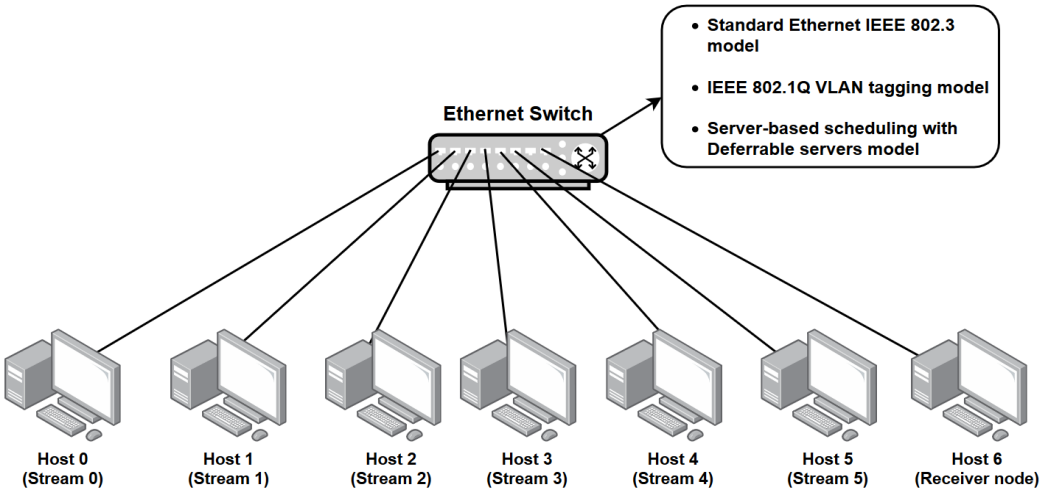


Figure 6.7: Experimental setup employed to compare switch models.

Table 6.10: Switch Comparison: Generated traffic properties in the experimental scenarios.

Source Node	Traffic Type	Associated Server	Payload (Bytes)	$T_i/Tmit_i(\mu s)$	Priority	Destination Node
0	Synchronous (S0)	$\Gamma 3_0$	1400	1500	6	6
1	Synchronous (S1)	$\Gamma 3_2$	1100	2500	4	6
2	Synchronous (S2)	$\Gamma 3_4$	1200	3000	2	6
3	Asynchronous Sporadic (A3)	$\Gamma 3_1$	1000	2300	5	6
4	Asynchronous Sporadic (A4)	$\Gamma 3_3$	700	6000	3	6
5	Asynchronous Aperiodic (A5)	$\Gamma 3_5$	1400	-	1	6

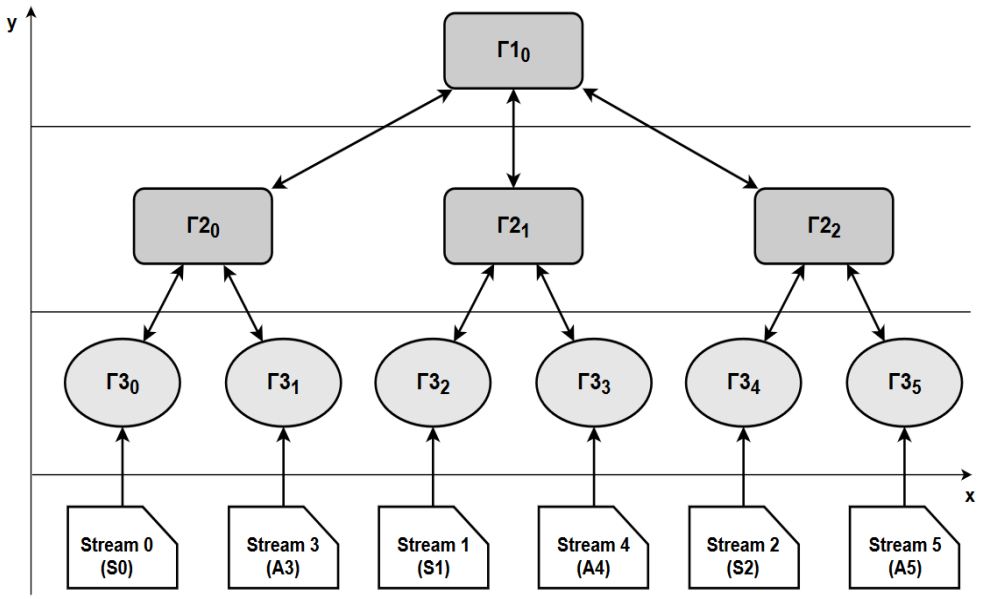


Figure 6.8: Switch Comparison: Server-based structure employed throughout the experiments.

Table 6.11: Switch Comparison: Servers' properties throughout all four experiments.

Server	Priority	Capacity Budget (Bytes)	Period (μs)
$\Gamma 1_0$	-	15000	800
$\Gamma 2_0$	3	5000	1000
$\Gamma 2_1$	2	3000	1000
$\Gamma 2_2$	1	5000	1000
$\Gamma 3_0$	2	1400	1500
$\Gamma 3_1$	1	1000	2000
$\Gamma 3_2$	2	1100	2500
$\Gamma 3_3$	1	700	6000
$\Gamma 3_4$	2	1200	3000
$\Gamma 3_5$	1	1400	5000

6.3.1 Experiment 1 - Normal operation

As it was previously mentioned, the first scenario was structured to simulate a network where all the nodes operate normally. Table 6.12 exhibits the results for 100000 single packet instances for each stream.

Table 6.12: Switch Comparison: Experiment 1 results.

			S0	S1	S2	A3	A4	A5
Inet Switch	Packets Dropped		0	0	0	0	0	0
	Latency (μs)	Min	228.69	180.69	196.69	164.69	116.69	228.69
		Max	576.67	362.21	461.39	524.47	398.42	550.58
		Mean	288.73	183.24	212.89	187.79	120.09	230.95
Hierarchical server-based switch	Packets Dropped		0	0	0	0	0	0
	Latency (μs)	Min	228.69	180.69	196.69	164.69	116.69	228.69
		Max	343.37	494.12	589.66	388.17	505.93	599.46
		Mean	275.35	184.46	227.83	184.91	123.42	231.22
Ethernet with IEEE 802.1Q switch	Packets Dropped		0	0	0	0	0	0
	Latency (μs)	Min	229.34	181.13	197.34	165.33	117.34	229.34
		Max	344.74	495.72	591.35	389.45	507.85	601.38
		Mean	276.18	185.13	228.60	185.70	124.11	231.88

The experiment results demonstrate that, for this specific situation, the three switch models have similar performances. As all nodes operate has it was supposed to, there are no dropped packets. Regarding the latency values, there are some minor deviations for both minimum and mean values between the three models, with less than 2 μs differences. For the maximum latency, however, there are some noticeable variances between the INET and the priority-based switches. This difference results from the switching rule employed by the devices, FIFO policy for the INET switch, and priority-based for the remaining two.

6.3.2 Experiment 2 - Abnormal node operation

In the second experiment, the streams were purposely altered so that their generated load would use close to 100% of the corresponding link, thus not respecting the associated reservations. This behavior was tested for three distinct situations: (i) the altered stream has the highest priority (S0), (ii) the modified stream as an intermediate priority (S1) and (iii) the modified stream has a lower priority (S2). For all three circumstances, upon transmitting 50000 packets, the streams revert to their normal behavior (identical to Exp1). Table 6.13 presents the streams' modifications. The results of the three described situations are shown in Table 6.14, Table 6.15 and Table 6.16, respectively.

Table 6.13: Switch Comparison: Streams' modifications.

Stream	Priority	Normal behavior		Abnormal behavior	
		Period (μs)	Load (Mbit/s)	Period (μs)	Load (Mbit/s)
S0	6	1500	7.5	100	100
S1	4	2500	3.5	100	89
S2	2	3000	3.2	100	97

Table 6.14: Switch Comparison: Experiment 2 results with the highest priority stream (S0) altered.

			S0	S1	S2	A3	A4	A5
Inet Switch	Packets Dropped		3227	1029	1004	1228	330	178
	Latency	Min	228.69	180.69	196.69	164.69	116.69	228.69
		Max	- - -	- - -	- - -	- - -	- - -	- - -
		Mean	5688.28	329.65	317.95	335.85	178.40	256.98
Hierarchical server-based switch	Packets Dropped		46566	0	0	0	0	0
	Latency	Min	228.69	180.69	196.69	164.69	116.69	228.69
		Max	- - -	404.03	474.37	393.03	513.36	565.35
		Mean	- - -	186.57	213.67	180.83	121.46	230.43
Ethernet with IEEE 802.1Q switch	Packets Dropped		0	2241	1858	2483	764	322
	Latency	Min	229.34	181.13	197.34	165.33	117.34	229.34
		Max	344.74	- - -	- - -	- - -	- - -	- - -
		Mean	270.94	6049.93	6053.01	6064.57	5693.98	5445.66

- - - (Indication indication that the deadlines were infringed)

Relatively to Exp1, the results show that the first created scenario has a significant impact on the switch performance, particularly for the IEEE 802.1Q model. For the two switch models with priority-based policies, this situation will force them to prioritize the messages of S0 that are continuously being generated. However, in the server-based switch, the bandwidth utilization is restricted by the framework. Therefore, if stream 0 uses the associated server resources, other servers can start processing their stored messages. The difference between the Exp1 and Exp2 results regarding S0 stems from the server replenishing period. As stream 0 period was decreased to 100 μs while the associated server stayed the same, the server reaches saturation faster. Consequently, accumulated packets have to wait for the replenishment instant for being transmitted, thus increasing their latency. Furthermore, due to insufficient storage, newly arriving packets are continuously being discarded.

Regarding the IEEE 802.1Q switch, because it has "unlimited" bandwidth, it always dispatches messages from stream 0 first. Thus, there is nearly no difference between the Exp1 results and this situation for that stream. On the other hand, all the remaining packets were delayed to the point that their latency values reached *5s*. Lastly, for the INET switch, the results exhibit an increase in all the streams' maximum and mean latency as well as the number of dropped packets. As messages priority is irrelevant, only the order in which they arrive at the

ports, lowering the streams' period resulted in more packets being stored than the ones being dispatched.

Table 6.15: Switch Comparison: Experiment 2 results with an intermediate highest stream (S1) altered.

			S0	S1	S2	A3	A4	A5
Inet Switch	Packets Dropped		1248	1355	283	756	99	80
	Latency	Min	228.69	180.69	196.69	164.69	116.69	228.69
		Max	9506.21	9470.93	9450.37	9433.52	9389.50	9476.95
		Mean	475.16	4618.23	331.03	317.74	177.76	255.74
Hierarchical server-based switch	Packets Dropped		0	47901	0	0	0	0
	Latency	Min	228.69	180.69	196.69	164.69	116.69	228.69
		Max	343.77	- - -	541.95	388.16	485.52	670.65
		Mean	270.67	- - -	202.62	184.04	122.80	230.07
Ethernet with IEEE 802.1Q switch	Packets Dropped		0	1539	1573	0	638	261
	Latency	Min	229.34	181.13	197.34	165.33	117.34	229.34
		Max	345.33	- - -	- - -	389.44	- - -	- - -
		Mean	270.94	5070.10	5173.01	185.85	4828.01	4584.99

Table 6.16: Switch Comparison: Experiment 2 results with a lower priority stream (S2) altered.

			S0	S1	S2	A3	A4	A5
Inet Switch	Packets Dropped		2443	78	3638	1204	346	167
	Latency	Min	228.69	180.69	196.69	164.69	116.69	228.69
		Max	- - -	- - -	- - -	- - -	- - -	- - -
		Mean	333.27	371.10	4898.39	283.42	157.68	249.00
Hierarchical server-based switch	Packets Dropped		0	0	48234	0	0	0
	Latency	Min	228.69	180.69	196.69	164.69	116.69	228.69
		Max	345.69	494.11	- - -	388.16	505.92	570.71
		Mean	243.674	185.61	- - -	182.74	122.21	230.53
Ethernet with IEEE 802.1Q switch	Packets Dropped		0	0	7703	0	0	260
	Latency	Min	229.34	181.13	197.34	165.33	117.34	229.34
		Max	344.73	495.71	- - -	389.44	507.84	- - -
		Mean	245.62	186.73	5575.60	184.29	123.60	4567.65

The remaining two scenarios results show that the server-based switch performance did not diverge from the first situation because the hierarchical structure limits the bandwidth utilization throughout the various components. The results confirm that the stream with the highest latency and packets dropped is always the one whose behavior is purposely changed, stream 1 and stream 2. As those streams are rapidly depleting the associated server capacity budget, more packets are being discarded while waiting for the replenishment instant.

The INET switch also has a similar performance in all three experimental scenarios. Nevertheless, there is a small difference between the last two regarding the number of discarded

packets that stems from the streams' loads. For the former, stream 1 generates a load of 89 Mbit/s, whereas, in the latter, stream 2 uses 97% of the respective uplink. Consequently, there are fewer packets dropped for the former scenario.

In IEEE 802.1Q switch, the difference in the messages' priorities considerably changes the way whole traffic is processed. For the second scenario, in which the modified stream has a medium priority (S1), higher priority frames, from stream 0 and stream 3, are always processed first. Thus their latency values did not differ from the results of Exp1. Regarding the lower priority streams latency, these reached values above *5s* because of interference caused by all the higher priority packets stored. For the last experimental scenario, when the altered stream has a lower priority (S2), all the remaining traffic, except from stream 5, is prioritized first. Therefore, the stream behavior does not influence the majority of the system.

The results from the second experiment show that the server-based switch has the best traffic confinement capabilities because the hierarchy restrains the streams' bandwidth. Thus, when one of them starts using more resources, there is no impact on the others. It is important to highlight that the hierarchy structure, as well as the servers' parameters, will considerably influence the switch performance. Regarding the IEEE 802.1Q switch, although this device has mechanisms to handle priority-based traffic, it has a poor performance when restricting the bandwidth consumption of higher priority streams, thus preventing lower priority packets from being processed. Lastly, the INET switch does not possess any capability to handle messages with different priorities. Therefore, both higher and lower priority streams have their latency values increased by a significant amount.

6.4 Experimental validation of HaRTES simulation model

The final section exhibits the results obtained from a HaRTES simulation model with the server-based scheduling framework integrated to manage the asynchronous traffic. Different experiments have been carried out to assess: (i) the switch traffic isolation capabilities, and (ii) the classification and confinement within the asynchronous window when managed by a server-based framework. Figure 6.9 depicts the experimental setup. It comprises the HaRTES switch and four nodes, from which two generate TT and ET real-traffic, one transmits NRT packets, and the remaining is the network receiver end-node. Throughout the experiments, the links are configured to 100 Mbit/s, and the switch internal latency (ϵ) can range from 2.0 to 2.4[4] μs.

[4]These values were based on experiments conducted by Rui Santos [2], in a HaRTES switch laboratory prototype.

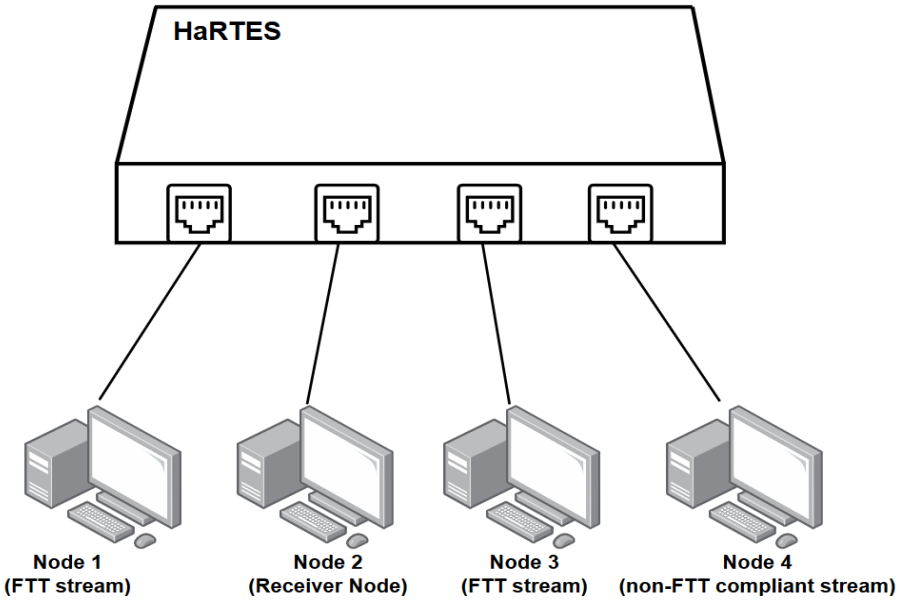


Figure 6.9: HaRTES: Experimental setup.

6.4.1 Temporal isolation between the traffic classes

The following segment assesses the temporal isolation between the different traffic classes in the HaRTES switch model. The different flows properties throughout the experiments are shown in Table 6.17.

Table 6.17: HaRTES Temporal isolation: Generated traffic properties for both experiments.

Experiment	Node	Traffic Type	Frame (Bytes)	$T_i/Tmit_i(\mu s)$	Destination Node
Exp1	1	Synchronous	1029	1000	2
	3	Synchronous	1029	2000	2
	4	Non Real-Time	829	67	2
Exp2	1	Synchronous	64	1000	2
	3	Asynchronous	1014	130	2
	4	Non Real-Time	1514	130	2

(Note: The frame size includes the Ethernet headers and overheads)

6.4.1.1 Experiment 1 - Synchronization assessment

For the first experiment, the FTT-master, responsible for managing the traffic within the switch, implements EC of 1 ms, which 45% is assigned to the synchronous window, 53% for the asynchronous window and 2% for the TM transmission (LTM) and respective processing by the FTT nodes (TAT). Note that the guard window was purposely disabled to demonstrate that the model prevents the NRT traffic from blocking the TM in the following elementary cycle. For this scenario, Nodes 1 and 3 generate 1kB synchronous real-time packets to Node 2. Node 4, the non-FTT compliant, sends 800B messages to Node 2 separated by the minimum

inter-frame gap, generating a load close to 100% of the uplink (Table 6.17). Packets that exceed the 45% load provided by the switch to the asynchronous window are automatically discarded. Figure 6.10 depicts a jitter histogram that affected the TM receptions. Figure 6.11 presents the inter-arrival of the different synchronous and non-real-time messages.

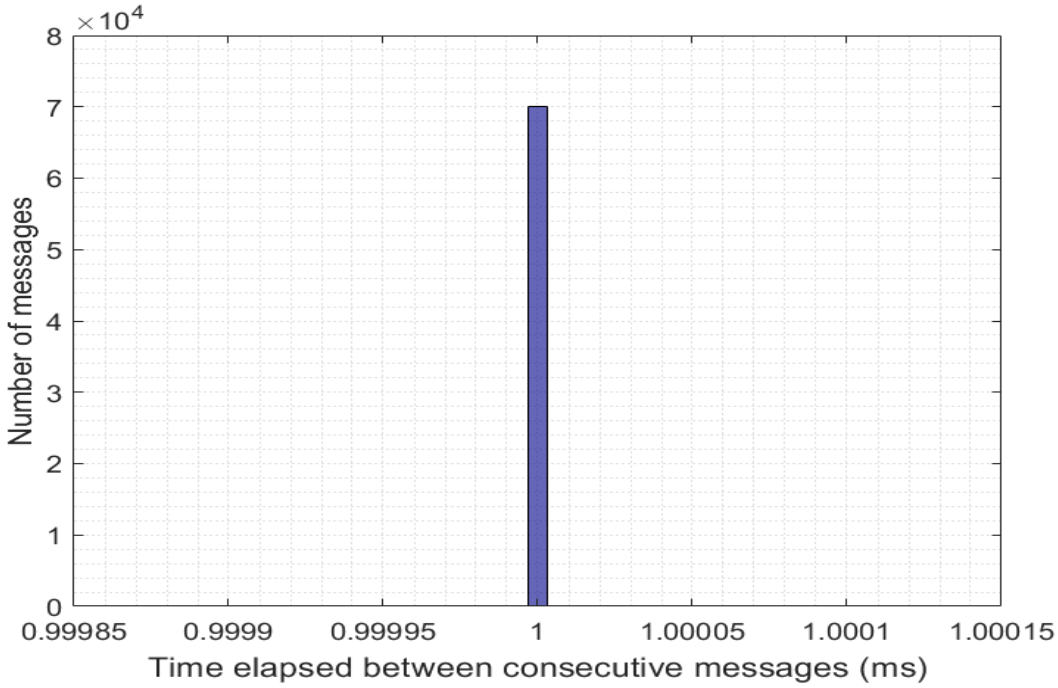


Figure 6.10: HaRTES Temporal isolation: Jitter affecting the Trigger messages.

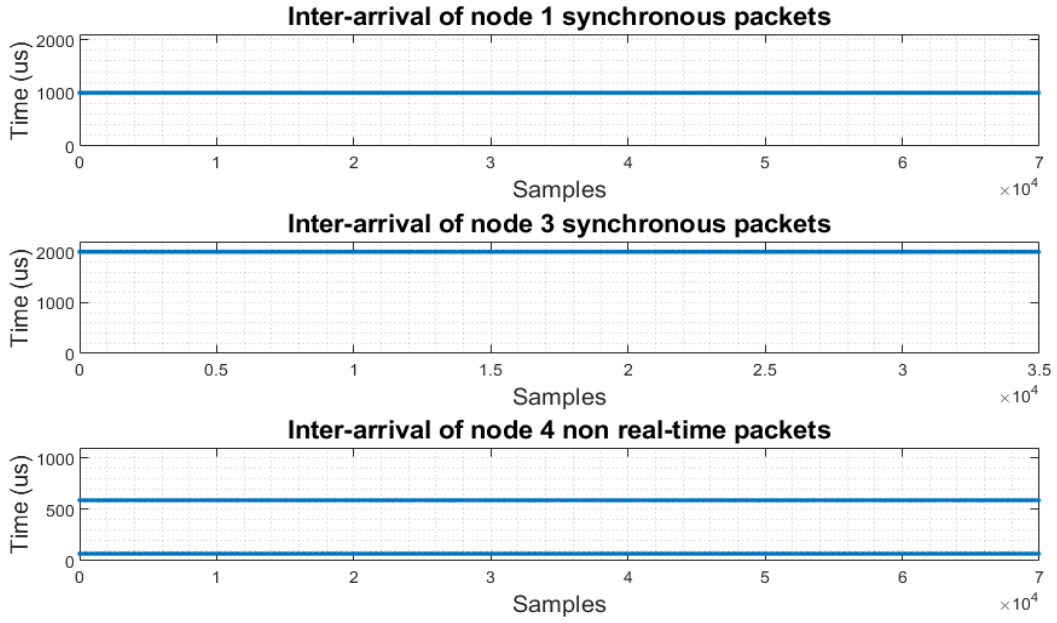


Figure 6.11: HaRTES Temporal isolation: Inter-arrival values for Experiment 1.

The results show that the synchronization of the FTT compliant nodes is never affected by the

NRT traffic. Figure 6.10 illustrates the high precision of TM transmissions by the switch with jitter reaching values inferior to 1 ns, even when one of the links was used close to 100% of its capability. Furthermore, the inter-arrival measurements confirm that the temporal isolation provided by HaRTES as the interference between the non real-time and time-triggered traffic is eliminated by the device. As such, the frames inter-arrival timings is always their respective periods, one EC for Node 1, and two ECs for Node 3. Regarding the non real-time packets, the minimum value of 70 μs corresponds to the transmission of a frame together with the switch latency. Whereas, the maximum value corresponds to part of the asynchronous window not utilized, the full duration of the synchronous window, and the transmission of a frame.

6.4.1.2 Experiment 2 - Comparison between the HaRTES simulation model and hardware prototype isolation capabilities

The procedures for the second scenario were taken directly from an experiment conducted by Luis Silva et al. in a hardware prototype of HaRTES extended for OpenFlow [64]. The FTT-master creates elementary cycles with 1 ms duration, with the following parametrization: 150 μs assigned to the synchronous window, 700 μs to the asynchronous window, 130 μs to the guard window, and the remaining time to the transmission and decoding of trigger messages. The properties of each traffic flow are presented in Table 6.17. With the minimal frame-gap of 130μs, the NRT and ET flows generate an approximate load of 94 Mbit/s and 64 Mbit/s, respectively. Table 6.18 compiles the properties of the employed reservations to handle the different traffic classes.

To evaluate the system performance, the authors chose as metrics the messages inter-arrival times. Figure 6.12 and Figure 6.13 depict the inter-arrival timings of each traffic flow for the simulation and the hardware results, respectively. Table 6.19 shows a statistical counterpart of the measurements.

Table 6.18: HaRTES Temporal isolation: Assigned reservations properties for Experiment 2.

Traffic Type	Reservation	Properties
Synchronous	Asynchronous Window	1 packet per EC
Asynchronous	Deferrable Server	Capacity Budget (Bytes): 1014 Period: 1 EC
Non Real-Time	Background Server	Capacity Budget (Bytes): 8750 Period: 1 EC

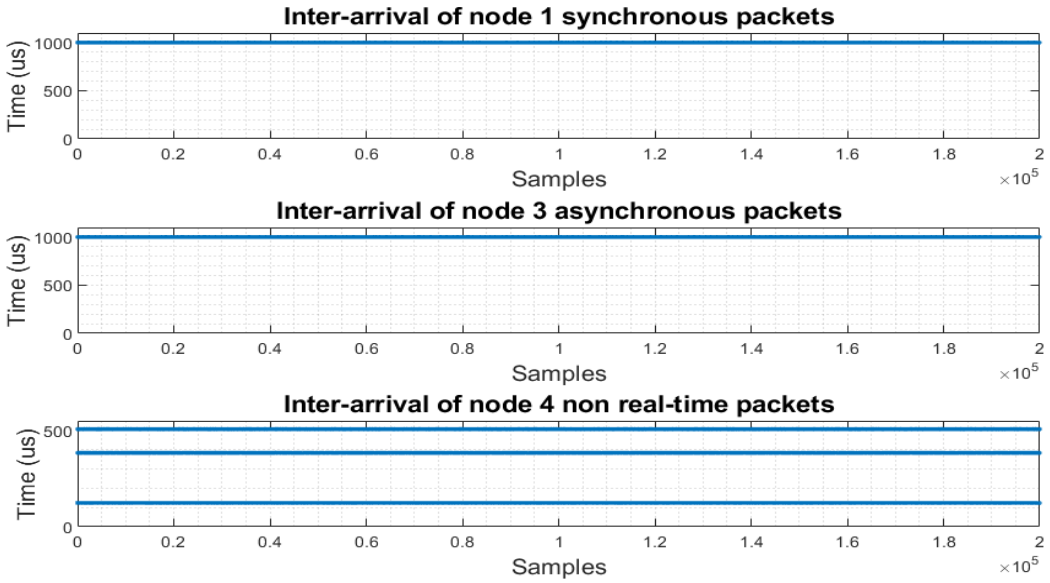


Figure 6.12: HaRTES Temporal isolation: Inter-arrival timings (simulator).

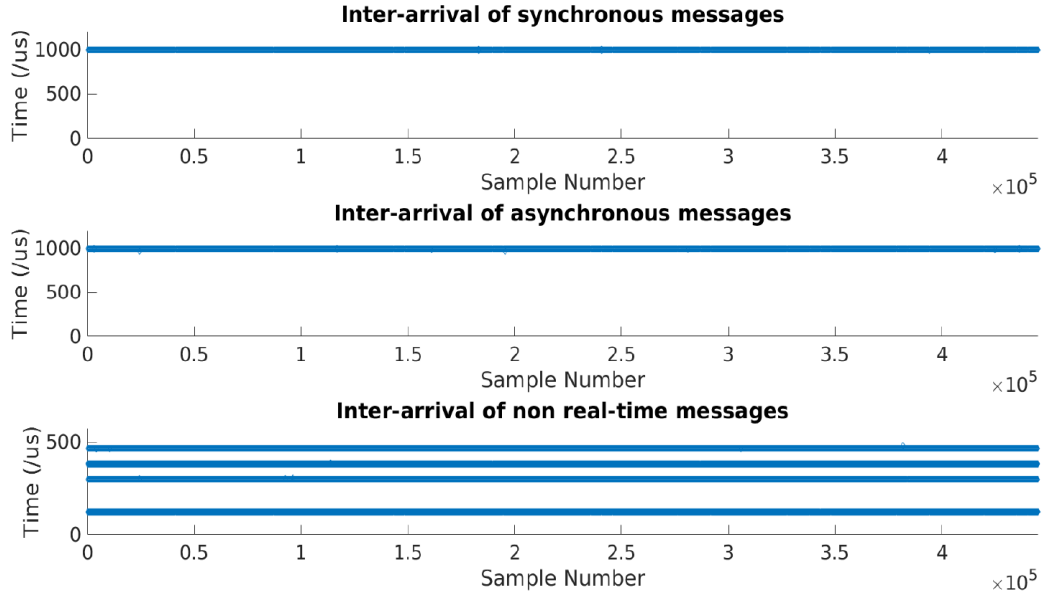


Figure 6.13: HaRTES Temporal isolation: Inter-arrival timings (hardware) (*from Luis Silva et al. [64]*).

Table 6.19: HaRTES Temporal isolation: Observed values for Experiment 2.

Traffic Flow	Inter-arrival in μs (simulator)			Inter-arrival in μs (hardware)		
	Min	Max	Mean	Min	Max	Mean
Synchronous	999.65	1000.33	1000.00	986.00	999.00	992.05
Asynchronous	999.65	1000.33	1000.00	986.30	999.00	992.04
Non Real-Time	123.20	506.99	190.43	121.00	479.00	160.63

The results demonstrate that the simulation model and the laboratory prototype have similar behaviors. The synchronous and asynchronous inter-arrival timings correspond to the streams' period and the reservations' maximum load, respectively, just as in the HaRTES prototype. However, regarding the non-real-time packets, there is a notable deviation between the measurements. This difference probably stems from the switch processing latency. For the simulation, the considered switch latency ranges from [2.0, 2.4] μs based on measurements made by Rui Santos [2]. However, these may no longer correspond to the currently existing prototype values. This metric is relevant as it affects when the packets are forwarded, specifically the non-real-time. Figure 6.14 illustrates the situations corresponding to the different inter-arrival timings measured. The minimum value of 123 μs is the transmission time of a frame together with the minimum switch latency. The maximum inter-arrival of 507 μs occurs whenever a frame finishes its transmission at the start of the guard window. Thus, the following packet is delayed the window remaining duration, the synchronous window full length, and the transmission of asynchronous and non real-time packets. Lastly, the other measured value of 383 μs happens when a transmission starts immediately before the guard window.

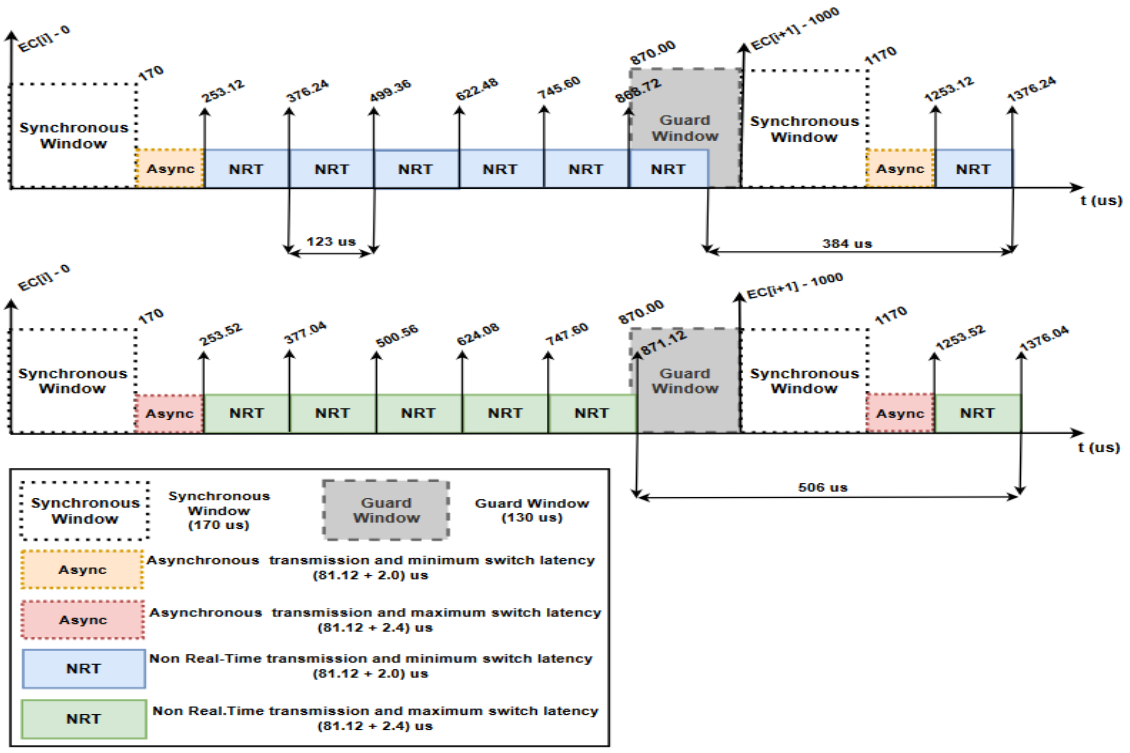


Figure 6.14: HaRTES Temporal isolation: Inter-arrival moments theoretical description.

The differences between the prototype and the simulation results could not be determined as these can also arise from issues in the HaRTES prototype implementation that will be investigated. An advantage of conducting experiments in network simulators and comparing the results with hardware-based tests is the possibility of encountering implementation issues in the prototype that could be difficult to identify.

6.4.2 Traffic confinement within the asynchronous subsystem

The second segment evaluates the switch capabilities to classify the different asynchronous traffic and confine its consumption to the asynchronous window duration through the use of a server-based scheduling framework. Table 6.20 shows the streams properties for the experimental scenarios.

Table 6.20: HaRTES Asynchronous Traffic confinement: Generated traffic properties for both experiments.

Experiment	Node	Traffic Type	Payload (Bytes)	$T_i/Tmit_i(\mu s)$	Destination Node
Exp1	3	Asynchronous	1500	123	2
	4	Non Real-Time	1500	123	2
Exp2	3	Asynchronous	1500	–	2
	4	Non Real-Time	1500	–	2

– (Indication that the inter-frame gap deviates throughout the experiment)

6.4.2.1 Experiment 1 - Assessment of the server-based traffic confinement capabilities

In the first experimental scenario, the nodes generate a mix of ET and NRT traffic. FTT-master implements 1 ms elementary cycles, with 28% assigned to the synchronous window, 54% to the asynchronous window, 16% to the guard window, and 2% to TM transmissions. Throughout the scenario, Node 3 transmits 1500B asynchronous real-time packets with a minimum inter-frame gap, thus using 100% of the uplink. Node 4 also generates a load close to 100% of the respective link with 1500B non-real-time packets (Table 6.20). Table 6.21 presents the reservations properties created to manage the asynchronous window.

The previously described procedure was performed in a laboratory prototype by Rui Santos et al.[65]. Figures 6.15 and 6.16 present a histogram of the time elapsed between TM transmissions and the remaining packets in each EC for the simulation model and HaRTES prototype, respectively.

Table 6.21: HaRTES Asynchronous Traffic confinement: Assigned reservations properties for Experiment 1.

Traffic Type	Reservation	Properties
Asynchronous	Deferrable Server	Capacity Budget (Bytes): 3000 Period: 2 EC
Non Real-Time	Background Server	Capacity Budget (Bytes): 6750 Period: 1 EC

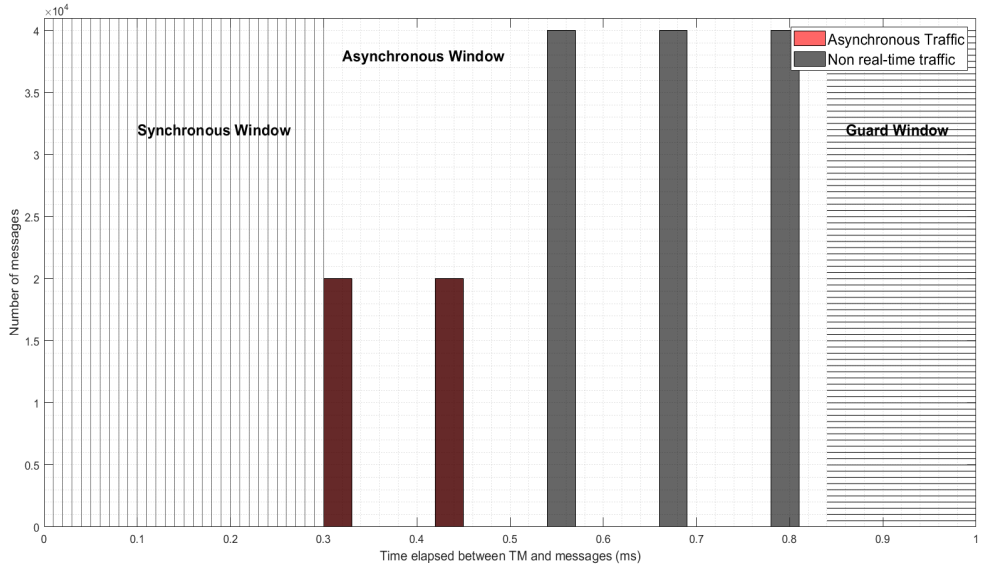


Figure 6.15: HaRTES Asynchronous Traffic confinement: Histogram of transmissions inside the EC (simulator).

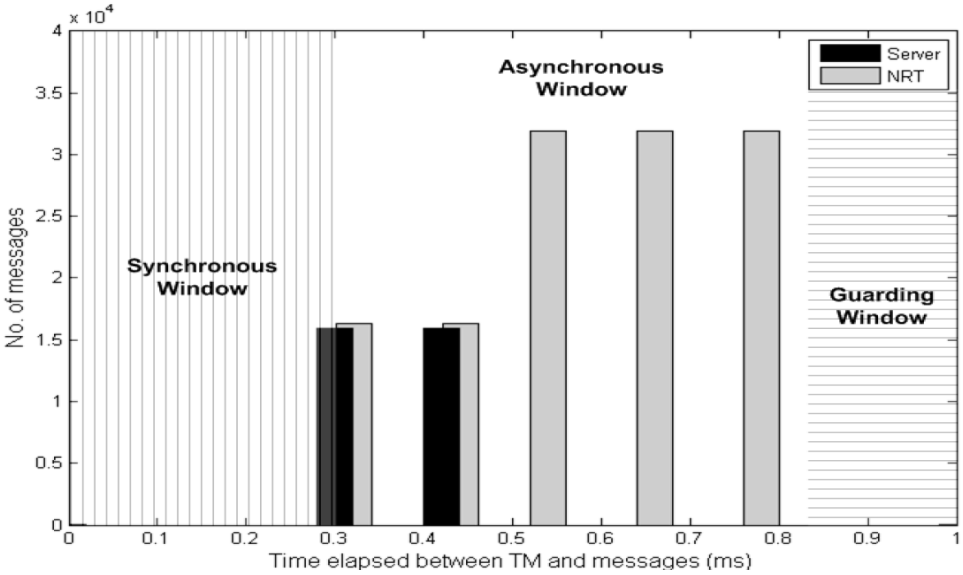


Figure 6.16: HaRTES Asynchronous Traffic confinement: Histogram of transmissions inside the EC (hardware) (*from Rui Santos et at [65]*).

Figure 6.15 confirms that the scheduling framework prioritizes the asynchronous real-time traffic, which is processed at the beginning of the asynchronous window, whereas, non-real-time packets are transmitted in the window remaining time. Note that, the ET and NRT bars overlap at the start of the asynchronous window. The deferrable server, with the capacity budget established, can only process two packets every two elementary cycles. Meanwhile, the background server uses the remaining time available in the window. Thus, at worst, it can transmit three packets after the transmission of the asynchronous traffic or use the complete duration of the AW to process five packets. The results show that, before the new a EC

initiation, during the guard window, no frames are transmitted. Although not mandatory, the guard window ensures that the ports are always ready to dispatch the TM in the following EC.

The results also show that the simulation model has a similar performance to the switch prototype. It is important to highlight that, in Figure 6.16, the histogram asynchronous bars were presumably shifted to demonstrate this traffic class prioritization as its transmission can never overlap the synchronous window. Another aspect that can explain some differences between the results is the switch latency. As the authors did not specify a value, the considered switch latency of [2.0,2.4] μs was based on measurements obtained in [2], from the same author.

6.4.2.2 Experiment 2 - Evaluating the switch throughput

Experiment 2 assesses the framework confinement capabilities within the asynchronous window. Throughout the last experiment, Node 3 transmits 150B real-time asynchronous messages to Node 2 with an inter-frame gap that ranges from 1 Mbit/s to 100 Mbit/s. Node 4 sends 600B non-real-time packets to the same destination, also with a variable load that reaches 100% of the uplink. The traffic is managed in HaRTES by the same server hierarchy employed in the previous experimental scenario. Furthermore, the EC parametrization is also identical to Experiment 1. Figure 6.17 illustrates the switch throughput for 60000 consecutive ECs, with and without the Guard Window implementation.

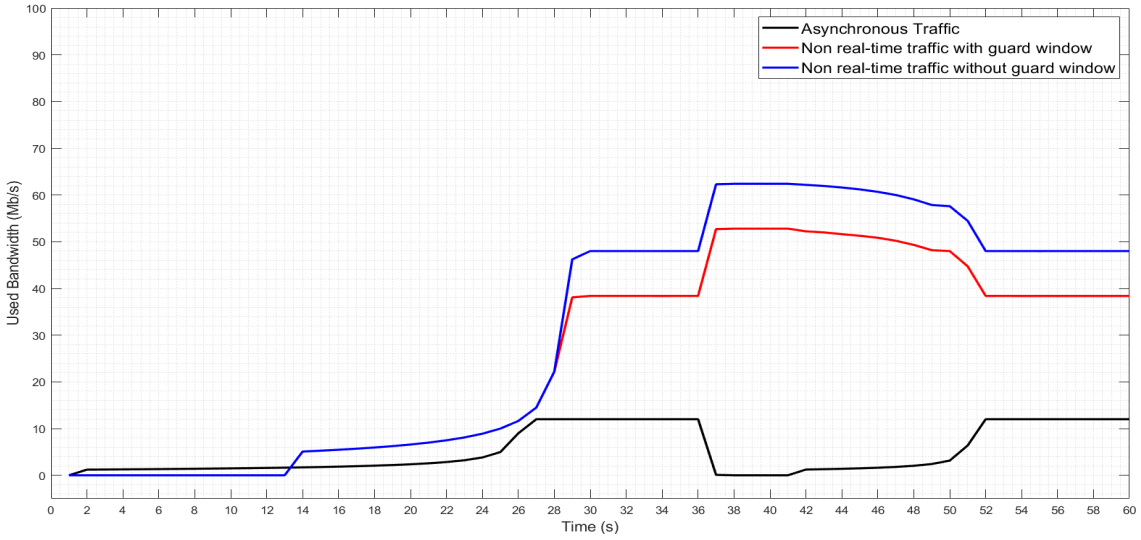


Figure 6.17: HaRTES Asynchronous Traffic confinement: Asynchronous and non-real-time traffic throughput.

Initially, the switch only dispatches asynchronous messages. These are sent by Node 3, with an inter-frame gap of 1 ms, decreasing over time until it reaches 100% of the uplink. The non-real-time traffic is triggered at t = 12s and has a similar behavior. At second 26, the Deferrable Server maximum load of 3000B is reached. As it can only process two frames per two EC, additional arriving packets are discarded within the switch. At second 30, the

background server also reaches saturation, with the load corresponding to the remaining of the asynchronous window. The asynchronous stream is deactivated at t = 35s, which enables the background server to use the respective window complete duration. Consequently, the NRT throughput increases until it reaches the window maximum load. At second 40, the stream is reactivated with the same behavior as before. This causes the non real-time throughput to decreased, and both return to the previous saturation values at second 52.

As expected, by completely remove the Guard Window, the non real-time traffic throughput increases whereas, the asynchronous traffic behavior stays identical since the deferrable server capacity budget limits the stream bandwidth utilization. As the switch can only process one packet at a time and, taking into account the switch latency, the top component capacity budget is never totally depleted. Consequently, the throughput of both traffic classes never reaches the available 70 Mbit/s.

The results of the experiments show the effectiveness of server-based scheduling in the HaRTES switch. The servers constrain the bandwidth usage in critical situations where the input ports are overloaded, thus solely allowing transmissions to occur through the elementary cycle respective window. The framework also has efficient capabilities to allocate the resources within the hierarchy without risking the requirements of the attached real-time streams. As the deferrable server stopped processing real-time packets, the background server was able to reclaim the remaining bandwidth available. Moreover, the asynchronous stream reactivation was not affected by the non real-time load.

CHAPTER 7

Closure

Contents

7.1 Conclusions

The advances in industrial infrastructures that emerged from the 4th Industrial Revolution impose strict timeliness requirements as well as high flexibility for their communication systems. Technologies based on real-time Ethernet such as the HaRTES switch provide such properties, with support for both real-time and non-real-time communications as well as a dynamic Quality-of-Service (QoS) management services.

As the limitations imposed by the development and testing of industrial networks have been growing with larger and complex systems, the utilization of network simulators provides several benefits in this sector. Among which, the possibility of examining specific circumstances that, with physical prototypes would be complicated, stands out. Recognizing the previous statement, a OMNeT++ Ethernet switch simulation model for industrial real-time networks was developed. The proposed model is based on HaRTES, an Ethernet switch based on the FTT paradigm, with a server-based framework to manage the asynchronous communications.

The individual assessment of the server-based framework took place before being employed in the HaRTES switch to guarantee the correctness of the structure. Both the control algorithm and the capabilities to perform online modifications were evaluated in different experimental scenarios. Lastly, the developed HaRTES model was validated throughout a set of experiments, some of which, directly compared with a physical prototype. The main aspects considered were the isolation between the different traffic classes and the confinement within the asynchronous window. The results demonstrate that the transmission of Trigger Messages is never interfered with by asynchronous and non-real time traffic, with low jitter values even in high load situations. Consequently, the FTT slaves are correctly synchronized as the inter-arrival of their messages is always equal to their periods. Furthermore, within the asynchronous window, the model prioritizes sporadic frames and transmits them first, as the non-real-time traffic is continuously handled by a background server with the remaining resources available.

The developed models achieved satisfactory results since the observed behaviors agreed with the technologies they intend to simulate. This type of tool has proven to be quite useful, specifically in the ease of changing the simulations parameters. As it was required to repeat the same experiments several times to try to simulate certain specific situations, the use of a simulator made this process fast and easy. Another simulator advantage is the possibility of discovering potential failures in the hardware implementations.

7.2 Future Work

Future works may include:

- Complete the switch and slave models to support all the different types of FTT requests.
- Address the worst-case response time scheduling analysis of the server-based framework for dynamic reconfigurations.
- Develop additional server-based scheduling algorithms for the asynchronous framework.
- Add support for multi-switch network topologies.

References

[1] G. C. Buttazzo, *Hard real-time computing systems: Predictable scheduling algorithms and applications*, 3. Springer, 1998, vol. 36, p. 126, ISBN: 0387231374. DOI: 10.1016/s0898-1221(98)90205-x.

[2] R. Santos, "Enhanced ethernet switching technology for adaptive hard real-time applications", PhD thesis, Universidade de Aveiro, 2007.

[3] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973, ISSN: 1557735X. DOI: 10.1145/321738.321743.

[4] J. Y. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks", *Performance Evaluation*, vol. 2, no. 4, pp. 237–250, 1982, ISSN: 01665316. DOI: 10.1016/0166-5316(82)90024-4.

[5] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System", *The Computer Journal*, vol. 29, no. 5, pp. 390–395, Jan. 1986, ISSN: 0010-4620. DOI: 10.1093/comjnl/29.5.390.

[6] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling", *Software Engineering Journal*, vol. 8, no. 5, p. 284, 1993, ISSN: 02686961. DOI: 10.1049/sej.1993.0034.

[7] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees", *Proceedings - Real-Time Systems Symposium*, no. December, pp. 2–13, 2003. DOI: 10.1109/real.2003.1253249.

[8] ——, "Compositional real-time scheduling framework with periodic model", *Transactions on Embedded Computing Systems*, vol. 7, no. 3, 2008, ISSN: 15399087. DOI: 10.1145/1347375.1347383.

[9] A. Easwaran, M. Anand, and I. Lee, "Compositional analysis framework using EDP resource models", *Proceedings - Real-Time Systems Symposium*, no. December, pp. 129–138, 2007, ISSN: 10528725. DOI: 10.1109/RTSS.2007.36.

[10] H. Kopetz, *Real-Time Systems*. Springer US, 2011. DOI: 10.1007/978-1-4419-8237-7. [Online]. Available: https://doi.org/10.1007%2F978-1-4419-8237-7.

[11] "Ieee standard for ethernet", *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, pp. 1–5600, 2018.

[12] "Ieee standard for local and metropolitan area network–bridges and bridged networks", *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014)*, pp. 1–1993, 2018.

[13] R. Marau, L. Almeida, and P. Pedreiras, "Enhancing real-time communication over COTS Ethernet switches", *IEEE International Workshop on Factory Communication Systems - Proceedings, WFCS*, no. June 2014, pp. 295–302, 2006. DOI: 10.1109/wfcs.2006.1704170.

[14] *802.1Q-2018 - IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks*. 2018, ISBN: 9781504449298. [Online]. Available: https://ieeexplore.ieee.org/document/8403927.

[15] *Ethernet Powerlink - EPSG Ethernet Powerlink*. [Online]. Available: https://www.ethernet-powerlink.org/.

[16] P. Pedreiras, P. Gai, L. Almeida, and G. C. Buttazzo, "FTT-Ethernet: A flexible real-time communication protocol that supports dynamic QoS management on Ethernet-based systems", *IEEE Transactions on Industrial Informatics*, vol. 1, no. 3, pp. 162–172, 2005, ISSN: 15513203. DOI: 10.1109/TII.2005.852068.

[17] P. Pedreiras and L. Almeida, "The flexible time-triggered (FTT) paradigm: An approach to QoS management in distributed real-time systems", *Proceedings - International Parallel and Distributed Processing Symposium, IPDPS 2003*, no. June, 2003. DOI: 10.1109/IPDPS.2003.1213243.

[18] *Isochronous Real-Time (IRT) Communication - PROFINET University*. [Online]. Available: https://profinetuniversity.com/profinet-basics/isochronous-real-time-irt-communication/ (visited on 06/24/2020).

[19] F. Scope, "PROFINET Real-Time Communication", *PROFIBUS International*, p. 17, 201. [Online]. Available: http://www.profibus.org.pl/index.php?option=com%7B%5C_%7Ddocman%7B%5C&%7Dtask=doc%7B%5C_%7Dview%7B%5C&%7Dgid=28.

[20] M. E. Yanik, "Full Duplex Switched Ethernet (Afdx) Data Bus", *Microelectronics, Guidance and Electro-Optics Division ASELSAN Inc.*, no. OCTOBER 2007, 2007.

[21] R. Marau, P. Pedreiras, and L. Almeida, "Signaling asynchronous traffic over a {M}aster-{S}lave {S}witched {E}thernet protocol", *Proc. on the 6th {I}nt. {W}orkshop on {R}eal {T}ime {N}etworks ({RTN}'07)*, no. June 2014, 2007.

[22] *Welcome to the FTT home page*. [Online]. Available: https://paginas.fe.up.pt/%7B~%7Dftt/sections/Flavours/index.html%7B%5C#%7Dftt%7B%5C_%7Dswitch (visited on 06/08/2020).

[23] R. Santos, P. Pedreiras, M. Behnam, T. Nolte, M. Mrtc, and L. Almeida, "Schedulability Analysis for Multi-level Hierarchical Server Composition in Ethernet Switches", *Proceedings of the 9th International Workshop on Real-Time Networks*, 2010. [Online]. Available: http://hartes.av.it.pt/files/papers/c-2010-rtn.pdf.

[24] G. Rodriguez-Navas and J. Proenza, "A proposal for flexible, real-time and consistent multicast in FTT/HaRTES Switched Ethernet", *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, pp. 1–4, 2013, ISSN: 19460740. DOI: 10.1109/ETFA.2013.6648153.

[25] R. Santos, M. Behnam, T. Nolte, P. Pedreiras, and L. Almeida, "Multi-level hierarchical scheduling in ethernet switches", *Embedded Systems Week 2011, ESWEEK 2011 - Proceedings of the 9th ACM International Conference on Embedded Software, EMSOFT'11*, pp. 185–193, 2011. DOI: 10.1145/2038642.2038671.

[26] J. Ferreira, P. Pedreiras, L. Almeida, and J. Fonseca, "Achieving fault tolerance in ftt-can", in *4th IEEE International Workshop on Factory Communication Systems*, 2002, pp. 125–132.

[27] S. Derasevic, J. Proenza, and M. Barranco, "Using ftt-ethernet for the coordinated dispatching of tasks and messages for node replication", in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, 2014, pp. 1–4.

[28] D. Gessner, J. Proenza, M. Barranco, and A. Ballesteros, "A Fault-Tolerant Ethernet for Hard Real-Time Adaptive Systems", *IEEE Transactions on Industrial Informatics*, vol. 15, no. 5, pp. 2980–2991, 2019, ISSN: 19410050. DOI: 10.1109/TII.2019.2895046.

[29] F. J. Suárez, P. Nuño, J. C. Granda, and D. F. García, *Computer networks performance modeling and simulation*, October 2017. 2015, pp. 187–223, ISBN: 9780128011584. DOI: 10.1016/B978-0-12-800887-4.00007-9.

[30] R. Mangharam, D. Weller, R. Rajkumar, P. Mudalige, and F. Bai, "GrooveNet: A hybrid simulator for vehicle-to-vehicle networks", in *2006 3rd Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services, MobiQuitous*, 2006, ISBN: 1424404991. DOI: 10.1109/MOBIQ.2006.340441.

[31] S. Y. Wang and Y. B. Lin, "NCTUns network simulation and emulation for wireless resource management", *Wireless Communications and Mobile Computing*, vol. 5, no. 8, pp. 899–916, 2005, ISSN: 15308669. DOI: 10.1002/wcm.354.

[32] M. Kabir, S. Islam, M. Hossain, and S. Hossain, "Detail comparison of network simulators", *International Journal of Scientific & Engineering Research*, vol. 5, no. May 2015, pp. 203–218, 2014. DOI: 10.13140/RG.2.1.3040.9128.

[33] R. L., M. J., and A. J., "Survey on Network Simulators", *International Journal of Computer Applications*, vol. 182, no. 21, pp. 23–30, 2018. DOI: 10.5120/ijca2018917974.

[34] Ns-3, *Ns-3 | a Discrete-Event Network Simulator for Internet Systems*, 2018. [Online]. Available: https://www.nsnam.org/ (visited on 05/04/2020).

[35] J. Heidemann and U. S. C. Isi, "OMNeT++ Discrete Event Simulator", *Audio*, no. March, pp. 1–9, 2002. [Online]. Available: https://omnetpp.org/%20https://omnetpp.org/%7B%5C%7D5Cnhttp://www.omnetpp.org.

[36] INET, *INET Framework - INET Framework*, 2019. [Online]. Available: https://inet.omnetpp.org/.

[37] *SimuLTE - LTE User Plane Simulator for OMNeT++ and INET*. [Online]. Available: https://simulte.com/%20http://simulte.com/ (visited on 06/29/2020).

[38] I. SCALABLE Network Technologies, *QualNet - Network Simulation*, 2019. [Online]. Available: https://www.scalable-networks.com/products/qualnet-network-simulation-software-tool/%7B%5C#%7D%20https://www.scalable-networks.com/qualnet-network-simulation.

[39] L. Bajaj, M. Takai, R. Ahuja, and K. Tang, "Glomosim: A Scalable Network Simulation Environment", *Compare A Journal Of Comparative Education*, vol. 28, no. 1, pp. 154–161, 1999. DOI: 10.1.1.45.7167. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.45.7167%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf.

[40] X. Zeng, R. Bagrodia, and M. Gerla, "GloMoSim: A library for parallel simulation of large-scale wireless networks", *Proceedings of the Workshop on Parallel and Distributed Simulation, PADS*, pp. 154–161, 1998. DOI: 10.1109/pads.1998.685281.

[41] *NetSim-Network Simulator & Emulator | Home*. [Online]. Available: https://www.tetcos.com/ (visited on 05/04/2020).

[42] T. Libraries and G. Started, "NetSim User Manual", vol. ver 11.1, pp. 1–244, 2019.

[43] OPNET, *OPNET Network Simulator - Opnet Projects*, 2009. [Online]. Available: http://opnetprojects.com/opnet-network-simulator/ (visited on 05/04/2020).

[44] S. Siraj, A. K. Gupta, and Badgujar-Rinku, "Network Simulation Tools Survey", *International Journal of Advanced Research in Computer and Communication Engineering Vol. 1, Issue 4, June 2012*, vol. 1, no. 4, pp. 201–210, 2012. [Online]. Available: http://www.nsnam.org/ns-3-13/download/.

[45] *TrueTime | Automatic Control*. [Online]. Available: http://www.control.lth.se/research/tools-and-software/truetime/ (visited on 05/04/2020).

[46] Mathworks, *Simulink - Simulation and Model-Based Design - MATLAB & Simulink*, 2015. [Online]. Available: https://www.mathworks.com/products/simulink.html%20https://uk.mathworks.com/products/simulink.html%7B%5C%7D0Ahttps://www.mathworks.com/products/simulink.html%7B%5C%7D0Ahttps://uk.mathworks.com/products/simulink.html (visited on 06/29/2020).

[47] A. Cervin, "TrueTime : Simulation of Networked and Embedded Control Systems", *Simulation*,

[48] A. Zarrad and I. Alsmadi, "Evaluating network test scenarios for network simulators systems", *International Journal of Distributed Sensor Networks*, vol. 13, no. 10, pp. 1–17, 2017, ISSN: 15501477. DOI: 10.1177/1550147717738216.

[49] R. L. Patel, "Survey on Network Simulators", Tech. Rep. 21, 2018, pp. 975–8887.

[50] M. C. Gayathri and D. R. Vadivel, "An Overview: Basic Concept of Network Simulation Tools", *International Journal of Advanced Research in Computer and Communication Engineering ICITCSA 2017 Pioneer College of Arts and Science, Coimbatore*, vol. 6, no. 1, 2017, ISSN: 2319-5940. DOI: 10.17148/IJARCCE.

[51] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment", *SIMUTools 2008 - 1st International ICST Conference on Simulation Tools and Techniques for Communications, Networks and Systems*, 2008. DOI: 10.4108/ICST.SIMUTOOLS2008.3027.

[52] S. Manual, "OMNeT++: Simulation Manual", *user's Manual Version 4.4.1*, pp. 1–511, 2001, ISSN: 1522-9653. DOI: 10.1016/j.joca.2010.07.004. [Online]. Available: https://doc.omnetpp.org/omnetpp/manual/%7B%5C#%7Dcha:overview%20https://omnetpp.org/doc/omnetpp/manual.

[53] *Protocol Simulations /*. [Online]. Available: https://1.ieee802.org/protocol-simulations/ (visited on 06/30/2020).

[54] *CloudNetSim++ - Toolkit for Distributed Data Center Simulations.* [Online]. Available: https://omnetpp.org/download-items/CloudNetSim.html (visited on 06/30/2020).

[55] M. Slabicki and G. Premsankar, *Home | FLoRa - A Framework for LoRa simulations.* [Online]. Available: https://flora.aalto.fi/ (visited on 06/30/2020).

[56] *NETA.* [Online]. Available: https://nesg.ugr.es/index.php/en/neta-2 (visited on 06/30/2020).

[57] *Simulation - CoRE Group.* [Online]. Available: https://core-researchgroup.de/projects/simulation.html (visited on 06/30/2020).

[58] *MiXiM.* [Online]. Available: http://mixim.sourceforge.net/ (visited on 06/30/2020).

[59] *FT4FTT | Systems, Robotics & Vision Group.* [Online]. Available: http://srv.uib.es/ft4ftt/ (visited on 07/04/2020).

[60] M. Knezic, A. Ballesteros, and J. Proenza, "Towards extending the OMNeT++ INET framework for simulating fault injection in ethernet-based Flexible Time-Triggered systems", *19th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2014*, pp. 1–4, 2014. DOI: 10.1109/ETFA.2014.7005319.

[61] D. Sheet, "PACSystems Industrial Ethernet Switches", no. April, 2020.

[62] P. Highlights, "7010T Gigabit Ethernet Data Center Switches 7010T Series | Platform Overview High Availability",

[63] T. Hpe, E. I. S. Series, F. Ethernet, E. I. S. Series, S.-b. G. Ethernet, and G. E. Sfp, "HPE 3600 EI Switch Series",

[64] L. Silva, P. Goncalves, R. Marau, and P. Pedreiras, "Extending OpenFlow with industrial grade communication services", *IEEE International Workshop on Factory Communication Systems - Proceedings, WFCS*, pp. 0–3, 2017. DOI: 10.1109/WFCS.2017.7991965.

[65] R. Santos, A. Vieira, R. Marau, P. Pedreiras, A. Oliveira, D. Ieeta, U. D. Aveiro, and L. Almeida, "Robust and Efficient Server-Based Communication over Switched Ethernet",