Alma Mater Studiorum - Università di Bologna

Dottorato di Ricerca in

COMPUTER SCIENCE AND ENGINEERING

Ciclo XXXV

Settore concorsuale:
09/H1 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

Settore scientifico disciplinare:
ING-INF/05 SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

QoS-aware Architectures, Technologies, and Middleware for the Cloud Continuum

Presentata da: Andrea Garbugli

Coordinatore Dottorato:
Prof.ssa Ilaria Bartolini

Supervisore:
Prof. Paolo Bellavista

Esame finale anno 2023

Abstract

The recent trend of moving Cloud Computing capabilities to the Edge of the network is reshaping how applications and their middleware supports are designed, deployed, and operated. This new model envisions a continuum of virtual resources between the traditional cloud and the network edge, which is potentially more suitable to meet the heterogeneous Quality of Service (QoS) requirements of diverse application domains and next-generation applications. Several classes of advanced Internet of Things (IoT) applications, e.g., in the industrial manufacturing domain, are expected to serve a wide range of applications with heterogeneous QoS requirements and call for QoS management systems to guarantee/-control performance indicators, even in the presence of real-world factors such as limited bandwidth and concurrent virtual resource utilization. The present dissertation proposes a comprehensive QoS-aware architecture that addresses the challenges of integrating cloud infrastructure with edge nodes in IoT applications. The architecture provides end-to-end QoS support by incorporating several components for managing physical and virtual resources. The proposed architecture features: i) a multilevel middleware for resolving the convergence between Operational Technology (OT) and Information Technology (IT), ii) an end-to-end QoS management approach compliant with the Time-Sensitive Networking (TSN) standard, iii) new approaches for virtualized network environments, such as running TSN-based applications under Ultra-low Latency (ULL) constraints in virtual and 5G environments, and iv) an accelerated and deterministic container overlay network architecture. Additionally, the QoS-aware architecture includes two novel middlewares: i) a middleware that transparently integrates multiple acceleration technologies in heterogeneous Edge contexts and ii) a QoS-aware middleware for Serverless platforms that leverages coordination of various QoS mechanisms and virtualized Function-as-a-Service (FaaS) invocation stack to manage end-to-end QoS metrics. Finally, all architecture components were tested and evaluated by leveraging realistic testbeds, demonstrating the efficacy of the proposed solutions.

Contents

List of Figures

List of Tables

Acronyms

BMCA Best Master Clock Algorithm

CNC Centralized Network Configuration

CPS Cyber-Physical Systems

CUC Centralized User Configuration

DMA Direct Memory Access

DPDK Data Plane Development Kit

FaaS Function as a Service

gPTP generic Precision Time Protocol

HPC High Performance Computing

I4.0 Industry 4.0

IEEE Institute of Electronics and Electrical Engineers

IETF Internet Engineering Task Force

IoT Internet of Things

IIoT Industrial Internet of Things

IT Information Technology

KVM Kernel-based Virtual Machine

MOM Message-oriented Middleware

M2M Machine-to-machine

NIC Network Interface Controller

NTP Network Time Protocol

OS Operating System

OT Operational Technology

OVS Open vSwitch

PTP Precision Time Protocol

QoS Quality of Service

RDMA Remote Direct Memory Access

SDN Software-Defined Networking

TCP Transmission Control Protocol

TSN Time-Sensitive Networking

UDP User Datagram Protocol

ULL Ultra-low Latency

VM Virtual Machine

XDP eXpress Data Path

Introduction

The widespread adoption of the Internet of Things (IoT) concept has driven an unprecedented digitalization process in various domains, such as Automotive, Industry 4.0 (I4.0), Healthcare, and Smart cities [82]. This trend is propelled by the exponential growth in connected devices, which collect vast volumes of raw data that should be transformed into insightful information by the next generation of IoT applications. This transformation enables the creation of innovative processes, services, and products in various industrial and societal sectors.

However, this transformation is characterized by high heterogeneity, stemming from devices with varying computing power, battery life, mobility, and more [17]. As a result, IoT applications with very different objectives and Quality of Service (QoS) must coexist, using heterogeneous technologies and resources such as communication protocols, storage, computing capacity, energy requirements, and security. The traditional cloud-centric model, which involves a few data centers collecting and processing all data generated by distant IoT devices, is inadequate in handling these heterogeneous requirements.

In recent years, there has been a notable paradigm shift in the integration of traditional Cloud infrastructures with diverse virtual and physical resources, leading to the emergence of the *Edge Cloud* or *Cloud Continuum* model. This model enables the efficient partitioning of available resources to meet the specific requirements of different applications, ensuring workload isolation and distribution across all layers of the infrastructure. Additionally, the Cloud Continuum model facilitates the establishment of end-to-end networks capable of meeting stringent Quality of Service (QoS) demands, while also supporting the development of decentralized and hierarchical programmable network architectures.

A crucial element in an infrastructure based on the Cloud Continuum model is the autonomous configuration of end devices and the network itself. It is essential for applications on end devices to effectively communicate their unique needs to the network control

plane, which in turn must guarantee QoS for all types of data traffic. Furthermore, the network should possess the ability to actively monitor and reconfigure itself in response to changing configurations within a reasonable timeframe.

Despite the considerable potential benefits offered by the Cloud Continuum model, its adoption in real-world production environments remains limited. For example, industries such as automation and healthcare have particularly stringent performance constraints that prove challenging to fulfill using existing resource virtualization technologies. These applications require the utilization of Ultra-low Latency (ULL) mechanisms for efficient communication between software components, necessitating the network infrastructure to ensure sub-millisecond communication between services. Recent advancements in Cloud Continuum technologies, such as Deterministic Networking (DetNet), Time-Sensitive Networking (TSN), and 5G, show promise in overcoming these strict constraints within Local Area Networks (LANs) and over larger distances, respectively.

However, the design, implementation, and deployment of comprehensive systems capable of meeting these stringent timing requirements present a significant challenge for researchers, especially with the increasing adoption of virtualization technologies in computing and networking. Achieving the desired level of service and seamless integration across all layers of the infrastructure calls for the development of novel management approaches that can effectively orchestrate computing, storage, and networking resources, ensuring QoS guarantees for future Internet of Things (IoT) applications. Furthermore, the Cloud Continuum model opens up possibilities for next-generation applications that leverage heterogeneous resources in terms of both hardware and software, thereby necessitating the creation of new system abstractions that transparently provide access to such a heterogeneous environment.

Given these challenges, this thesis proposes a comprehensive and multi-layered architecture that aims to ensure adherence to the QoS requirements of next-generation Cloud Continuum applications in IoT and Industrial IoT systems. The proposed architecture builds upon prior research and advancements in the field, encompassing three primary layers: the *Infrastructure Layer*, the *Virtualization Layer*, and the *Application Layer*.

The Infrastructure Layer assumes responsibility for the efficient management and allocation of physical resources, such as network and computing capabilities. The Virtualization Layer extends current network and computing virtualization techniques to support ULL network communications, leveraging modern network acceleration techniques while offer-

ing a transparent interface to the upper layers. Lastly, the Application Layer encompasses platforms and middlewares that leverage the underlying layers' mechanisms to facilitate the development of QoS-aware applications within IoT contexts.

The thesis is organized clearly and comprehensively, with each chapter building upon the previous one to provide a thorough understanding of the proposed architecture and its capabilities.

Chapter 1 presents an overview of the defining characteristics of Cloud Computing and its limitations, leading to the concept of the Cloud Continuum. The The chapter also introduces the fundamental principles of the Internet of Things and Industrial IoT as a representative example of a system that demands the integration of various resources and the management of data streams with varying Quality of Service (QoS) requirements.

Chapter 2 delves into the concept of Ultra-low Latency (ULL) networks and communication, outlining their defining characteristics and the performance criteria that must be met. The chapter examines the Time-Sensitive Networking (TSN) standard as a representative standard for ULL communication in IoT and Industrial IoT applications. Additionally, the chapter evaluates the role of modern network acceleration technologies and network virtualization techniques in both classical and containerized virtualization settings in realizing ULL communications.

Chapter 3 provides an in-depth illustration of the proposed architecture, examining its objectives, characteristics, and sub-components. The current state-of-the-art in each layer is also analyzed. A comparison is made between the proposed architecture and previous works, highlighting the improvements and advancements made in each aspect.

Chapter 4 focuses on the Infrastructure Layer, providing a detailed examination of middleware for converging Operational Technology (OT) and Information Technology (IT) to support applications with diverse QoS requirements in Edge Cloud environments. The chapter also introduces an end-to-end Quality of Service management system for Time-Sensitive Networking (TSN) networks that aims to ensure consistent QoS while handling network reconfiguration events and different traffic flows.

Chapter 5 delves into the Virtualization Layer, discussing three distinct components of the proposed architecture. The first component focuses on a new approach to run TSN-based applications with ULL constraints in virtual machines by combining a practical clock synchronization approach with high-performance network virtualization techniques. The second component presents a system for an accelerated and deterministic container

overlay network defining new userspace TSN packet scheduler. The third component is represented by an Edge-oriented general-purpose middleware that offers a uniform access interface to a wide range of network acceleration technologies and provides a minimal set of communication primitives to allow the definition of domain-specific abstractions.

Chapter 6 focuses on the Application Layer. It begins by presenting a QoS-aware middleware for Serverless platforms, which leverages the coordination of various QoS mechanisms and virtualized FaaS invocation stack to manage end-to-end QoS parameters such as jitter, latency, and enqueuing time. The chapter then highlights two applications developed based on the uniform acceleration access middleware presented in Chapter 5: a Message-oriented Middleware (MOM) and a high-quality image streaming application.

In this thesis's Conclusion and Future Work chapter, we summarize the main objectives and outcomes of the work presented. Additionally, we highlight the strengths and limitations of our proposed architecture and provide recommendations for future research and development in the field. This conclusion serves as a summary of our contributions, a platform for discussion, and suggestions for future work that can build upon and enhance our existing proposal.

Published Papers

Before proceeding with the substantive examination of this thesis, it is appropriate to briefly outline the research activities conducted during the doctoral period and the resulting peer-reviewed publications or works currently undergoing publication review.

- **Conference Paper - *Accepted***. Garbugli, A., Rosa, L., Bujari, A., & Foschini, L. *KuberneTSN: a Deterministic Overlay Network for Time-Sensitive Containerized Environments*. In ICC 2023-IEEE International Conference on Communications. IEEE.

- **Conference Poster - *Published***. Mazzuca, L., Garbugli, A., Sabbioni, A., Bujari, A., & Corradi, A. (2022, September). *Towards a Resource-aware Middleware Support for Distributed Game Engine Design*. In Proceedings of the 2022 ACM Conference on Information Technology for Social Good (pp. 409-413).

- **Conference Paper - *Published***. Rosa, L., & Garbugli, A. (2022, July). *Poster: INSANE–A Uniform Middleware API for Differentiated Quality using Heterogeneous*

Acceleration Techniques at the Network Edge. In 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS) (pp. 1282-1283). IEEE.

- **Conference Paper - *Published***. Garbugli, A., Rosa, L., Foschini, L., Corradi, A., & Bellavista, P. (2022, May). *A Framework for TSN-enabled Virtual Environments for Ultra-Low Latency 5G Scenarios*. In ICC 2022-IEEE International Conference on Communications (pp. 5023-5028). IEEE.

- **Journal Article - *Published***. Garbugli, A., Sabbioni, A., Corradi, A., & Bellavista, P. (2022). *TEMPOS: QoS Management Middleware for Edge Cloud Computing FaaS in the Internet of Things*. IEEE Access, 10, 49114-49127.

- **Journal Article - *Published***. Patera, L., Garbugli, A., Bujari, A., Scotece, D., & Corradi, A. (2021). *A Layered Middleware for OT/IT Convergence to Empower Industry 5.0 Applications*, Sensors, 22(1), 190.

- **Conference Paper - *Published***. Garbugli, A. (2021, August). *PhD Forum Abstract: Ultra-low Latency Communication in TSN-based Virtual Environments*. In 2021 IEEE International Conference on Smart Computing (SMARTCOMP) (pp. 414-415). IEEE.

- **Conference Paper - *Published***. Garbugli, A., Bujari, A., & Bellavista, P. (2021, June). *End-to-end QoS management in self-configuring TSN networks*. In 2021 17th IEEE International Conference on Factory Communication Systems (WFCS) (pp. 131-134). IEEE.

1 From the Cloud to a Continuum of Virtual Resources

Contents

In this chapter, we will briefly review the evolution of the Cloud toward a more distributed model called *Edge Cloud* or *Cloud Continuum*. In particular, we will elaborate on how this model is better suited in the context of the Internet of Things and Industrial Internet of Things (IIoT) as it enables next-generation applications exploiting heterogeneous resources and multiple data flow with diverse QoS requirements.

1.1 Introduction to Cloud computing

In the last decade, the *Cloud* has emerged as a disruptive and rapidly evolving technology that has significantly transformed how organizations operate and deliver value to their customers. The term "cloud" refers to the delivery of computing services, such as storage, computing, and applications, via the Internet, without the need to own or manage the underlying infrastructure leading to a paradigm shift in the traditional role of service providers in the IT industry [83]. Specifically, the role has been divided into two categories: *infrastructure providers* and *service providers*. Infrastructure providers operating cloud platforms and using a usage-based pricing model are responsible for leasing resources. On the other hand, service providers who lease resources from one or more infrastructure providers are responsible for serving end users. The emergence of Cloud Computing has significantly impacted the Information Technology industry, with major players such

as Google, Amazon, and Microsoft competing to provide more robust, reliable, and cost-effective cloud platforms.

Cloud computing has become a popular choice for businesses due to its numerous benefits. One of the main benefits is the lack of upfront investment required. With a *pay-as-you-go* pricing model, service providers can avoid significant upfront investment in infrastructure and rent resources from the Cloud as needed, resulting in significant cost savings. Another advantage of Cloud Computing is the reduced operational costs. By rapidly allocating and deallocating resources on demand, companies can eliminate the need for provisioning capacity based on peak loads and achieve significant savings. The Cloud also offers high scalability, as infrastructure providers aggregate significant resources from data centers, making them readily available for service providers to rapidly expand their services to large scales to accommodate rapid increases in demand.

Additionally, Cloud-hosted services typically feature web-based accessibility, allowing easy access through a wide range of Internet-enabled devices. Outsourcing service infrastructure to the Cloud can also help reduce service providers' business risks and maintenance expenses by shifting responsibilities for managing hardware failures and other risks to infrastructure providers, who often have specialized expertise. This can save costs through reduced hardware maintenance and staff training expenses.

The architecture of Cloud Computing is modular and can be divided into four layers: the *hardware/datacenter layer*, the *infrastructure layer*, the *platform layer*, and the *application layer*. The hardware/datacenter layer manages physical resources such as servers, routers, switches, power, and cooling systems and is responsible for hardware configuration, fault tolerance, traffic management, and power and cooling resource management. The infrastructure layer, also known as the *virtualization layer*, creates a pool of storage and compute resources by partitioning physical resources using virtualization technologies, enabling dynamic resource allocation. The platform layer, built on top of the infrastructure layer, consists of operating systems and application frameworks and aims to reduce the burden of deploying applications directly into Virtual Machines (VMs). The application layer is the highest level of the hierarchy. It consists of the actual cloud applications, which can take advantage of the automatic scaling feature to achieve better performance, availability, and lower operational costs.

Cloud Computing employs a service-driven business model, where hardware and platform-level resources are provided as services on an on-demand basis. In practice,

clouds offer services that can be grouped into three categories: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). IaaS refers to the on-demand provisioning of infrastructural resources, usually in terms of VMs. PaaS refers to providing platform layer resources, including OS support and software development frameworks. SaaS refers to providing on-demand applications over the Internet.

When moving an enterprise application to the Cloud environment, there are different types of clouds to consider, each with its benefits and drawbacks. *Public clouds* are available to the general public but lack fine-grained control over data, network, and security settings. *Private clouds*, also known as Internal clouds, are designed for exclusive use by a single organization and offer the highest degree of control over performance, reliability, and security. *Hybrid clouds* combine public and private Cloud models, offering more flexibility than public and private clouds.

1.2 From Fog and Edge Computing to the Cloud Continuum

The ongoing advancement of computing and network infrastructure has resulted in significant digital transformation and evolution across various industrial sectors. The increased distribution and heterogeneity of next-generation applications and their diverse Quality of Service requirements, including low-latency constraints in the hundreds of microseconds and high levels of reliability, have rendered the traditional Cloud Computing model inadequate.

In response to this need, various models have emerged in recent years that extend the Cloud Computing model to a multi-level hierarchical structure encompassing nodes at the edge of the network infrastructure, referred to as *Fog Computing* and *Edge Computing*. These concepts will be briefly described, with a discussion on how they have led to the development of a more general term, the *Cloud Continuum*.

Edge computing is an emerging paradigm aiming to bring computation closer to the data sources rather than relying on centralized data centers. This paradigm shift is driven by the need to support the growing Internet of Things devices and the increasing demand for low latency and high bandwidth applications. The core idea of Edge Computing is to

perform computation and storage at the edge of the network, close to the source of data, to reduce the amount of data that needs to be transmitted over the network [67].

Researchers have different definitions of Edge Computing, but they all agree on bringing computation closer to the data sources. For example, Shi et al. [67] define edge computing as "a new computing mode of network edge execution. The downlink data of edge computing represents cloud service, and the uplink data represents the Internet of Everything. The edge of edge computing refers to the arbitrary computing and network resources between the data source and the path of the cloud computing center". Similarly, Satyanarayanan, in [64], defines Edge Computing as "a new computing model that deploys computing and storage resources (such as cloudlets, micro data centers, or fog nodes, etc.) at the edge of the network closer to mobile devices or sensors".

Generally speaking, the critical advantage of Edge Computing is that it enables the processing of data closer to the source, which reduces the amount of data that needs to be transmitted over the network. This can significantly improve response times, energy consumption, and security. For example, researchers have shown that using cloudlets to offload computing tasks for wearable cognitive-assistance systems improves response times by between 80 and $200\,\mathrm{ms}$ and reduces energy consumption by 30 to $40\,\%$. Edge Computing can also better protect data by processing it closer to the source. However, supporting security and privacy can be more challenging in edge computing due to the network topology, the many inexpensive personal mobile devices in the system, and sensor unreliability.

In conclusion, Edge Computing is an emerging paradigm aiming to bring computation closer to the data sources. It enables the processing of data closer to the source, which reduces the amount of data that needs to be transmitted over the network, leading to improvements in response times, energy consumption, and security. However, supporting security and privacy can be more challenging in edge computing due to the network topology, the many inexpensive personal mobile devices in the system, and sensor unreliability. Edge computing is an active research area and will continue evolving as new technologies, and use cases emerge.

Internet of Things is a computing concept in which many intelligent objects, such as sensors, actuators, and mobile devices, can sense their surroundings, transmit and process the acquired data, and provide feedback using wired and wireless Internet standards-based connections. This concept's introduction has improved applications' quality, reducing

costs and enhancing functionality while offering more accessible access to resources and greater automation. The application of IoT and Cyber-Physical Systems (CPS) [21] ideas to the domain of industrial automation led to the definition of Industry 4.0 [79].

As a subset of IoT, the Industrial Internet of Things is the core concept of Industry 4.0. IIoT is where sensors, software, Machine-to-machine (M2M) collaboration, and various technologies gather and analyze data from the physical and virtual worlds for optimized operations and provide better services. For this reason, IoT, IIoT, and Cloud Computing are crucial for the manufacturing sector.

However, it is essential to note that while Cloud Computing is often advertised as a solution for IoT-related technological problems, it can be challenging to utilize in safety-critical systems due to its long end-to-end latency [59]. Instead, the Cloud Continuum is being proposed as an architectural means to achieve IT/OT convergence by bringing Cloud computing Closer to the Edge, reducing latency, and providing a more efficient and secure way to manage the data generated by IoT devices. The Cloud Continuum encompasses all computing environments, including edge devices. It offers a more comprehensive approach to Cloud Computing, enabling real-time data processing, decision-making on Edge, and improved security, privacy, and compliance.

The integration of these concepts will inevitably lead to an evolution of the industrial and production sectors, especially regarding the size of the networks in these sectors. An ecosystem will therefore be created, within which information sharing will be essential to improve decision-making processes and obtain efficient production concerning the resources used. Also, since systems of this type often exploit rapid communications between devices, factories, and suppliers, greater flexibility will be required to meet customer needs in terms of quantity, quality, design, and configuration.

2 Ultra-low Latency Communications and Networks

Contents

The need for Ultra-low Latency communications is rapidly increasing due to the growing demand for near real-time connectivity and high data rates in various fields, such as critical health care, transportation, industrial automation, autonomous vehicles, Augmented Reality (AR) and Virtual Reality (VR), and robotics. Traditional networks have successfully reduced end-to-end operational latencies by tens of milliseconds, but current and future applications require latencies of a few microseconds or milliseconds [52]. This chapter delves into the key concepts and technologies driving the progress of ULL networks, particularly by discussing the Time-Sensitive Networking standard and some innovative network acceleration technologies.

2.1 Terminology

This section provides an overview of the terminology used to define Ultra-low Latency communications.

We define *latency* as the total end-to-end packet delay from the time of initiation of transmission by the sender to the completion of reception by the receiver. Latency is a critical parameter in ULL networks because, as anticipated, many ULL applications require deterministic latency that is often less than a millisecond. Specifically, deterministic latency means that all frames of a given application traffic flow must not exceed a specific limit to ensure proper systems operation. This type of specification often results in industrial automation applications. In other cases, latency must have probabilistic behavior. The specified delay limit must be met with high probability, but failing to meet it does not lead to critical problems. For example, this type of latency is often required in multimedia streaming systems, where rare violations of the delay limit have a negligible impact on the perceived quality of the multimedia.

The second term that defines ULL networks is latency jitter or *jitter* for short. Jitter refers to variations in packet latency that, as in the case of latency, must be very small. Latency and jitter are the two main metrics that define QoS for ULL networks.

Finally, a secondary requirement, however present in ULL applications, is the *throughput*. The latter logically depends on the application's needs, which can vary widely from small amounts of IoT data to large exchanges of multimedia data to and from the cloud or edge in case latency is to be reduced. In particular, autonomous automotive vehicles, augmented and virtual reality, and robotic applications, all essential elements in modern Industrial Internet of Things applications, may require high data rates and low end-to-end latency requirements. For example, high data rates may be needed to carry video feeds from cameras that monitor vehicles and robots. Therefore, Given the wide range of ULL applications, which often have very different QoS requirements, a reliable mechanism to meet these different requirements universally is critical to the success of ULL networks.

2.2 Time-Sensitive Networking

This section will discuss the Time-Sensitive Networking standard and the protocols it introduces to support ULL applications. This standard builds upon the foundation of

Ethernet, which has become a popular choice for networking due to its simplicity and low cost. Ethernet has evolved over the years, and today it supports connections at speeds of up to 400 Gbps.

Despite the widespread success and adoption of Ethernet, it must be noted that the basic definition of Ethernet fundamentally lacks deterministic end-to-end flow properties. This lack of Quality of Service properties makes it difficult to support ULL applications, such as industrial communications, that require deterministic, real-time packet delivery.

Before the development of Time-Sensitive Networking standards, ULL applications were often addressed through alternative means, such as point-to-point communications, circuit switching, or through the use of specialized and semi-proprietary specifications. These included Fieldbus communication and various variants of Ethernet for use in the OT domain, such as PROFINET, EtherCAT, Ethernet/IP, and TTEthernet.

These variants of Ethernet, encapsulated under the generic term Industrial Ethernet, present a significant challenge in the form of compatibility issues among themselves [79]. This has resulted in a fragmented market with multiple incompatible solutions, making it challenging for users to select the most appropriate solution for their specific needs.

The advent of TSN standards aims to address these limitations and provide a unified, standardized solution for ULL applications in industrial and OT domains. By incorporating TSN capabilities, Ethernet can now offer the deterministic and real-time packet delivery required for mission-critical applications, enabling more widespread adoption of Ethernet in these domains.

In general, the Ethernet definitions lack the following aspects for supporting ULL applications:

- Lack of QoS mechanisms to deliver packets in real-time for demanding applications, such as real-time audio and video delivery.
- Lack of global timing information and synchronization in network elements.
- Lack of network management mechanisms, such as bandwidth reservation mechanisms.
- Lack of policy enforcement mechanisms, such as packet filtering, to ensure a guaranteed QoS level for an end-user.

Motivated by these Ethernet shortcomings, the Institute of Electronics and Electrical Engineers (IEEE) and the Internet Engineering Task Force (IETF) have proposed new definitions to introduce deterministic network packet flow concepts. The IEEE has pursued the TSN standardization, focusing mainly on the physical layer (layer one, L1) and link layer (layer two, L2) techniques within the TSN task group in the IEEE 802.1 working group (WG). The IETF has formed the Deterministic Network (DetNet) working group focusing on the network layer (L3) and higher layer techniques [52]. These new definitions aim to bring Ethernet to new heights of performance and efficiency, making it a more versatile and reliable option for ULL applications.

2.2.1 Time Synchronization

Time synchronization is a crucial aspect of networks, particularly in the context of TSN applications. These applications require a network-wide precise time synchronization, establishing a standard time reference shared by all TSN network entities. This time synchronization is used to determine the opportune moment for data and control signaling scheduling, ensuring that all devices in the network are working in harmony. To achieve this, the IEEE 802.1AS stand-alone standard is employed, which uses a specialized profile of the IEEE 1588 Precision Time Protocol standard called the generic Precision Time Protocol (gPTP) [39, 72].

gPTP is a specialized protocol that synchronizes clocks between network devices by passing relevant time event messages. It envisages two main entities, a Clock Master (CM) and Clock Slaves (CSs), deployed and provisioned on the networked devices. The CM, also referred to as the Precision Time Protocol (PTP) grandmaster, sends time information to each of the CSs connected using multicast communication. Each CSs, also called gPTP instance, must correct the synchronized time received by factoring in the time delay due to message propagation along the gPTP communication path from the grandmaster to the PTP instance.

Once all devices are synchronized, we have what is, in effect, a time-aware network also referred to as a gPTP domain (Figure 2.1). This time-aware network utilizes the peer-path delay mechanism to compute the residence time, i.e., the ingress-to-egress processing, queuing, and transmission time within a bridge, and the link latency, i.e., the single hop propagation delay between adjacent bridges within the time-aware network hierarchy.

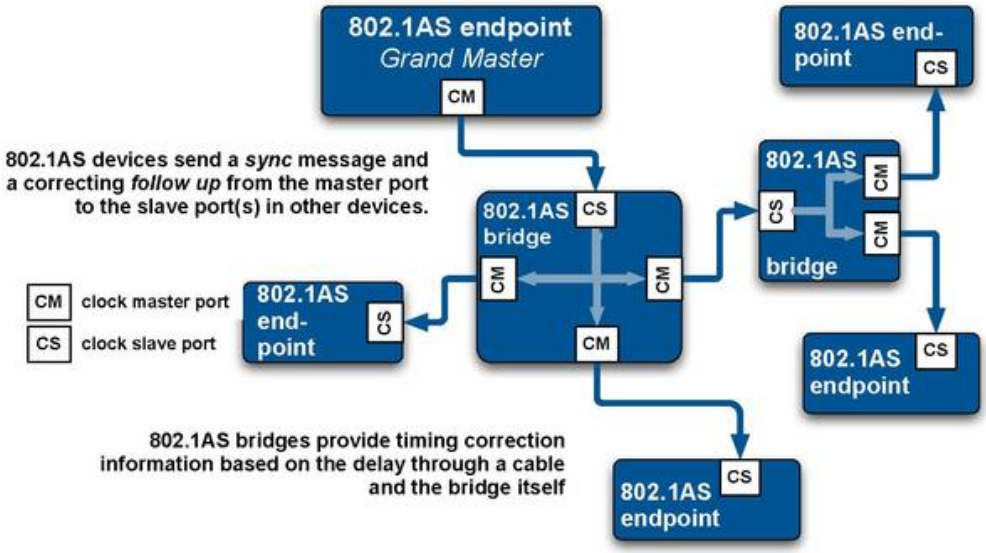


Figure 2.1: Example of a gPTP domain.

The GrandMaster (GM) clock, located at the root of the hierarchy, is used as the reference for the entire network. The GM clock is the bridge with the most accurate clock source, selected by the Best Master Clock Algorithm (BMCA).

gPTP systems consist of distributed and interconnected gPTP and non-gPTP devices. Time-aware bridges and endpoints are gPTP devices, while non-gPTP devices include passive and active devices that do not contribute to time synchronization in the distributed network. gPTP uses a master-slave architecture to synchronize the real-time clocks in all devices in the gPTP domain with the root reference (GM) clock. Synchronization is accomplished through a two-phase process: gPTP devices establish a master-slave hierarchy and then apply clock synchronization operations. In particular, gPTP establishes a master-slave hierarchy using the BMCA, consisting of two separate algorithms: data set comparison and state decision. Each gPTP device operates a gPTP engine, i.e., a gPTP state machine, and employs several gPTP User Datagram Protocol (UDP) IPv4 or IPv6 multicast and unicast messages to establish the appropriate hierarchy correctly synchronize time.

2.2.2 Flow Scheduling

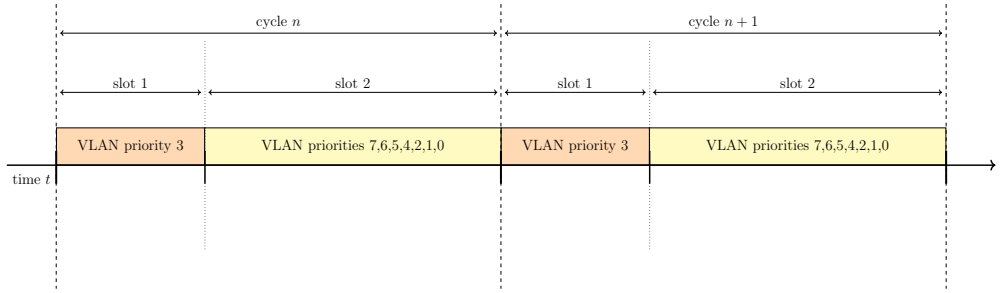
The TSN standard provides a primary protocol, called IEEE 802.1Qbv, for supporting traffic flows with diverse QoS requirements.

To ensure traffic-friendly QoS, such as those in critical control systems with sub-microsecond latency requirements, the IEEE 802.1Qbv Enhancements to Traffic Scheduling Time-Aware Shaper (TAS) standard has been created [52]. This standard specifies the planning of critical traffic flows within time-triggered communication windows (Figure 2.2). These windows are often referred to as secure traffic windows or time-aware traffic windows, and each of them is divided into multiple time slots that repeat cyclically, see Figure 2.2a. We can select a series of traffic classes within each time slot so that this slot is used only to transmit these traffic classes. This way, we can prevent lower-priority traffic, such as best-effort traffic, from interfering with real-time or scheduled traffic transmissions. A so-called guard band precedes those scheduled traffic windows (Figure 2.2b), and packets belonging to other traffic classes remain in their buffers until their traffic class can be transmitted. According to a known and time-aware schedule, the locking mechanism of these queues is based on open or closed gates. Transmission is therefore controlled through a Gate Control List (GCL), consisting of several scheduling items [52].

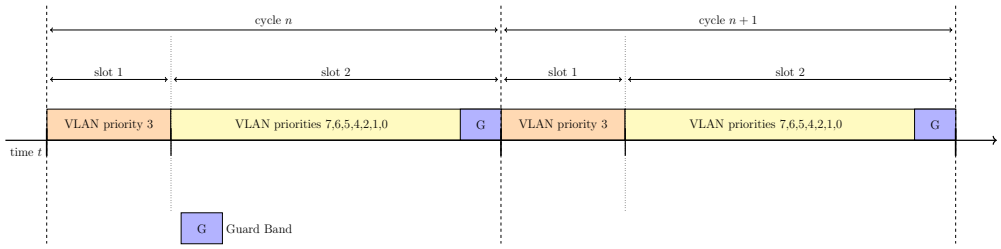
2.2.3 Network Management

Traditionally, one of the essential aspects of the IEEE 802.1 network is using some plug-and-play mechanisms to add more devices. The standard for providing this type of tool is IEEE 802.1Qcp YANG Data Model [18]. The YANG data model provides a framework for periodic status reporting and configuring 802.1 bridges and bridge components. In particular, YANG is a data modeling language for configuration data, state data, remote procedure calls, and notifications for network management protocols, e.g., NETCONF [29] and RESTCONF [16]. The latter represents the Network Configuration Protocols that provide mechanisms to install, manage, and delete network devices' configurations. In addition to a standard model representing a network device's properties, it is also essential to define an architecture to connect the different devices and manage and configure them.

For this reason, IEEE 802.1Qcc [47] provides global tools to manage and control the network. IEEE 802.1Qcc enhances the existing SRP with a User-Network Interface (UNI) supplemented by a Centralized Network Configuration (CNC) node. The UNI provides



(a) Time-aware traffic window with the different time slots.



(b) Time-aware traffic window with the presence of guard bands.

Figure 2.2: Schemes of time-aware traffic window.

a standard method of requesting Layer 2 services. Furthermore, the CNC interacts with the UNI to offer a centralized means for performing resource reservation, scheduling, and other types of configuration via a remote management protocol, such as NETCONF or RESTCONF; hence, 802.1Qcc is compatible with the IETF YANG/NETCONF data modeling language. An optional Centralized User Configuration (CUC) node communicates with the CNC via a standard Application Programming Interface (API) to create a fully centralized network architecture. This CUC can be used to discover end stations, the talkers, and listeners, to retrieve their capabilities and user requirements, and to configure delay-optimized TSN features in end stations. In Figure 2.3, we can see a diagram representing the centralized model we have just introduced.

2.2.4 Integration with 5G

Wireless communication's general use was limited to open-loop control and Manufacturing Execution System (MES) applications due to lack of availability, reliability, and real-time capabilities [7]. However, several standardization organizations today con-

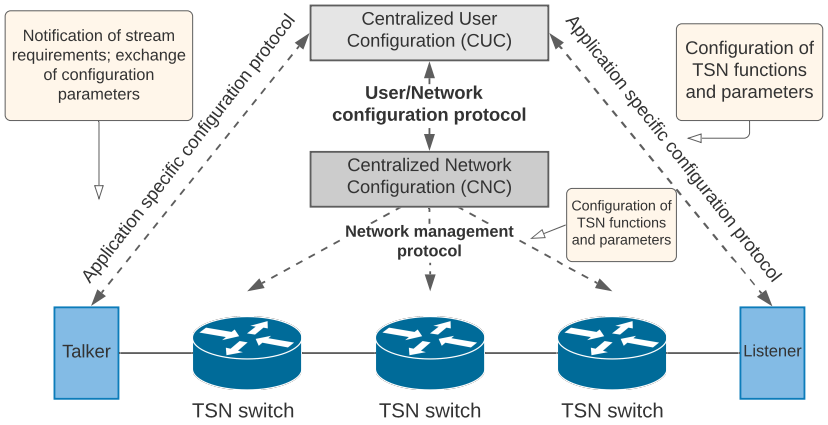


Figure 2.3: Fully centralized configuration model for TSN networks.

tribute to 5G standards, including IEEE and IETF, the Third Generation Partnership Project (3GPP), and the European Telecommunications Standards Institute (ETSI). Fifth-generation cellular technology (5G) represents a paradigm shift in network connectivity as 5G is expected to completely overhaul the network infrastructure by establishing an ULL end-to-end connection. The 5G, therefore, aims to meet the requirements for a wide range of field-level applications. It will enable the design of easily reconfigurable factories, reducing cable installation and maintenance and enabling innovative use cases such as automated guided vehicles (AGVs) with robotic arms.

Additionally, wireless communication systems result in lower installation costs, enabling large-scale manufacturing facility upgrades. The integration of 5G will improve the network's efficiency by improving its utilization and limiting the overhead of the control plane, thus leading to more significant energy savings.

The benefits of 5G and the corresponding industrial use cases are described in more detail in [8]. As a result, in combination, TSN and 5G offer wireless and wired solutions to create the sizeable real-time network needed for Industry 4.0 applications. To ensure the control application's deterministic behavior, the network must meet the corresponding QoS requirements. A solution should use standard Ethernet (IEEE 802.3), WLAN (IEEE 802.11), and 5G (3GPP) technologies, combined with time-sensitive communication enhancements such as the TSN standards specified by IEEE 802.1.

2.3 Acceleration Technologies

A key component in achieving the desired QoS in ULL communications is using acceleration technologies. These technologies play a crucial role in facilitating the achievement of the specified QoS level, thereby improving the overall performance of the communication system.

Communication links are evolving rapidly to support higher bandwidth and lower latency, outpacing the evolution rate of other host resources such as core speeds and cache sizes. This has led to an issue where the Operating System kernel-level networking stack, designed under the assumption of slower I/O operations, can no longer keep up with the available access link bandwidths and latencies [22]. This problem is particularly relevant for Edge Cloud scenarios, as datacenter-like resources are available at the network edge and latency requirements become extremely demanding [63, 70]. The primary sources of network overhead in the OS kernel include data copies, inefficient cache usage, protocol processing delays, and context switches [22, 35, 57].

To fully exploit the communication capabilities of modern hardware, new forms of highly efficient end-host networking have emerged, such as the Linux eXpress Data Path, which provides fast in-kernel packet processing [74], the Data Plane Development Kit and Remote Direct Memory Access, which bypass the kernel and allow direct interaction between userspace and Network Interface Controllers (NICs)[5, 6]. Despite their differences, all these techniques follow a similar approach to reduce the various sources of overhead. For example, they remove data copies by giving the user application access to a memory area where the NIC directly writes incoming data. This way, data is always in the same physical memory area, and the application and the NIC driver only need to exchange the address of the relevant data in the shared area. However, the mechanisms that enable these approaches substantially differ across the different technologies.

The Linux kernel introduced eXpress Data Path (XDP) as the lowest layer of its network stack, located within the driver of network devices. At this stage, XDP can execute user-provided code (eBPF programs) for each packet, forwarding it to and from a userspace socket. This way, XDP allows one to send and receive packages without involving the other network stack components, thus avoiding expensive operations such as memory allocation for incoming packets. The price to pay is that some amount of CPU is spent to forward each packet between the driver and the socket. To use XDP, developers have

first to open a socket of type `AF_XDP` and a shared memory area to allow the zero-copy packet writes/reads (directly or through higher-level libraries such as `libxdp` [3]). Then, users send packets by placing data into the memory area and writing a packet descriptor to the socket. Once received the descriptor, the eBPF program will send the packet on the network without copies. Packet reception works in the same way, but roles are reversed. If the network card supports it, it is possible to offload the eBPF program execution to the hardware. Therefore, this approach bypasses the kernel TCP/IP network stack, achieving efficient zero-copy and low-overhead data transfers. In turn, however, the user has to provide its userspace network and transport protocols (e.g., mTCP [40]).

Data Plane Development Kit (DPDK) and Remote Direct Memory Access (RDMA) take a step further and bypass the OS kernel altogether. This approach reduces scheduling overhead because there is no context change between userspace and kernel processes on the critical datapath. DPDK, in particular, consists of a set of C libraries that let users directly interact with a userspace version of the network device drivers (Poll Mode Drivers, PMD). Hence, in this case, the user has to provide its protocol stack. The user and the userspace driver exchange packet data on a shared memory area called *mempool*. To send a packet, the user will provide the driver with a pointer to the appropriate memory area. On the receiving side, to minimize the communication overhead, DPDK dedicates one or more threads (*lcores*), each pinned to a separate core, to busy poll for new messages. The driver places detected packets into the shared memory, and the corresponding pointers are returned to the user. Although extremely fast, this high resource consumption significantly limits adopting DPDK in constrained environments. Overall, all these techniques provide efficient zero-copy and low-overhead data transfers. Still, they require the user to provide their protocol stack and may have high resource consumption in some cases.

RDMA is a networking technique that allows a process on one machine to directly access the memory of another process on a remote device. Unlike XDP and DPDK, this abstraction avoids the need for the user to provide userspace network and transport protocols. To achieve exceptional performance, including high throughput ($\sim$200 Gbps) and low latency ($<$1 µs), RDMA offloads the network operations directly to the network card (NIC). Thus, a compatible network card is required. After registering a memory area with the network card (*memory region*), users establish a remote connection by opening a Queue Pair (QP), which comprises a couple of *work queues* for send and receive operations. Indeed, RDMA operations are asynchronous by nature: a node can issue a series of

Table 2.1: A comparison between the main options for end-host networking in the edge cloud.

Technology	Kernel integration	API	Zero-copy	CPU consumption	Dedicated Hardware
Kernel TCP/IP	In-kernel	`AF_INET` Socket	No	Per-packet	No
XDP	In-kernel	`AF_XDP` Socket	Yes	Per-packet	No
DPDK	Kernel-bypassing	RTE	Yes	Busy polling	No
RDMA	Kernel-bypassing	Verbs	Yes	Hardware offloading	Yes

service requests to be executed by the hardware, pushing them to the proper queue. Those requests include the transfer of portions of local memory to remote memory regions or vice versa. The network card enforces these requests transparently by implementing hardware protocols such as RDMA over Converged Ethernet (RoCEv2) [12]. There are two possible kinds of transfers: *two-sided*, which requires the receiver to listen to incoming data actively, and *one-sided*, which allows a process on one machine to asynchronously access a region of application memory on a remote node. A significant advantage of the latter is that the remote CPU is not involved in the network operation: an advantageous property in the Edge context.

Table 2.1 presents a summary of the critical features of XDP, DPDK, and RDMA. These technologies were initially designed for specific purposes, such as fast packet processing in network core routers for XDP and DPDK and HPC networking for RDMA. However, as their usage expands to include general-purpose end-host networking, they become potent options for supporting the heterogeneous requirements of the Edge scenario. One significant challenge that developers may face is using custom APIs for accessing network operations, which can be low-level and complex, potentially overwhelming for non-experienced system programmers. Additionally, the frequent evolution of these interfaces can make it challenging to maintain older code.

2.4 Network Virtualization

Next-generation applications based on the Cloud Continuum model demand a seamless integration of physical and virtual resources. This requirement extends to the ULL communication technologies utilized by these applications, emphasizing the importance of providing Quality of Service mechanisms for virtualized networks. These mechanisms must be effective in both traditional virtualization environments and the more lightweight, container-based virtualization contexts.

2.4.1 Virtual Machines

A key challenge for virtual machine applications is obtaining efficient access to I/O devices, especially if network performance is critical. The two prominent I/O virtualization techniques are direct device assignment and paravirtualization. With direct device assignment, a dedicated device instance is assigned exclusively to a VM and becomes invisible to the host (*passthrough*). Each VM requires a dedicated physical network adapter if this instance corresponds to the physical device (physical function, PF). To mitigate this effect, recent devices support hardware-assisted virtualization (e.g., SR-IOV [4]) that makes them appear as multiple separate devices called virtual functions (VFs), which can be assigned to different VMs. Either way, this technique avoids any involvement of the hypervisor: VMs can access the network as if they were physical hosts and achieve optimal network performance. However, the direct assignment also tightly couples network devices and VMs, strongly limiting the flexibility properties of virtualization, e.g., live migration.

On the contrary, the paravirtualization technique trades performance for flexibility. For each VM, a traditional paravirtualized network stack splits the device driver into a *frontend driver* in the guest OS and a *backend driver* on the host (Figure 2.4), which exchange commands using a dedicated communication channel. This separation allows the hypervisor to have complete control of the network state, thus enabling a high degree of flexibility and introducing overhead on data path operations when crossing the guest/host boundaries. The backend driver is located in the host kernel space in the traditional approach. All the traffic should traverse the host network stack, which involves multiple data copies and context switches [22]. Since this overhead can be significant, it is possible to move the backend driver in the host userspace and use kernel-bypassing techniques such as the DPDK to access the physical device with a zero-copy semantic directly.

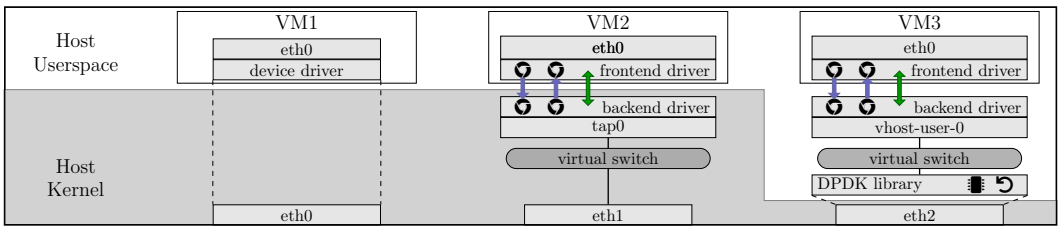


Figure 2.4: Network virtualization approaches.

The *de facto* standard framework for paravirtualization is virtio [61], which allows the hypervisor to expose paravirtualized devices to the guest. To reduce the network overhead, virtio separates the data plane, used for the actual transmission of network traffic between the host and the guest, and a control plane to exchange control messages about the data plane. The data plane consists of shared memory regions between the guest's frontend driver and the host's backend driver. Those memory areas, called *virtqueues*, are organized as couples of ring buffers that contain data to be received and transmitted, thus simulating the virtual queues of physical devices. Each virtual device can have zero or more associated queues, but, importantly, devices with more than one queue can only be used from virtual machines with two or more virtual CPUs (vCPUs) because each queue must have its associated thread. On the other hand, the control plane consists of a notification mechanism to detect and notify data in the queue between the frontend and the backend driver. This mechanism consists of a direct inter-process communication channel between the two drivers for network devices.

Once the VM traffic reaches the backend driver, it must be forwarded to a network. In Cloud environments, the common practice is to connect the VM belonging to the same tenant to a virtualized overlay network, regardless of the host they are running on. The critical component to achieve this is the virtual switch. This software application can isolate and manage traffic among VM on the same host and forward data to remote switch instances through point-to-point network tunnels. For example, Open vSwitch [58] is a widely used solution for virtual networking. It can operate at the kernel level and with DPDK to process traffic faster in the userspace. In the following, we evaluate the performance of both these options to investigate the cost of virtualization in ultra-low latency scenarios.

2.4.2 Containers

The popular choice for container networking in containerized environments is the overlay mode, which is frequently utilized in conjunction with Kubernetes. This mode offers improved isolation, ease of use, and security. In the overlay mode, containers are connected through an overlay network spanning multiple physical nodes across different networks. Each container has a virtual network interface, which can assign an IP address connected to the external world via a virtual switch in the host operating system's kernel. This virtual

switch acts as a bridge between co-located containers and tunnels network traffic to remote containers across the physical network. This setup results in containers on the same overlay network having an isolated address namespace and configuration separate from the host network or other overlays, as depicted in Figure 2.5.

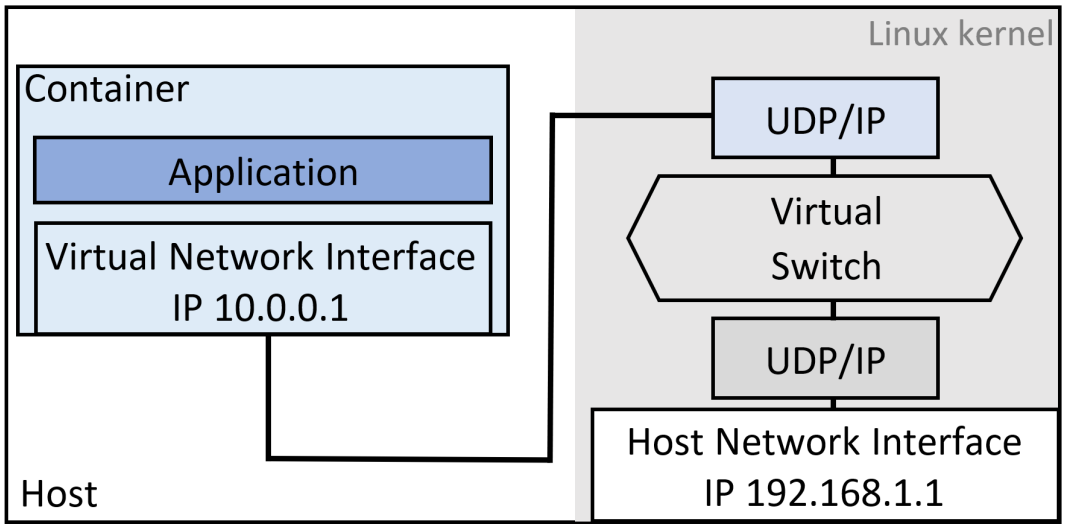


Figure 2.5: Container networking in overlay mode.

When utilizing Kubernetes, containers possess a single network interface for all network traffic, including management and control plane interactions with the Kubernetes master. The Multus plugin enables the attachment of additional interfaces to containers to differentiate between different traffic classes. Multus is a meta-plugin defining a container network interface (CNI) that other plugins can implement to configure a Layer 3 network fabric and provide advanced features. Several plugins exist, such as Flannel, Calico, or Weave, but none support the creation of an accelerated and deterministic communication channel among containers. For this reason, sending packets in a container overlay network incurs substantial overhead, as each outgoing packet must pass through multiple layers of the networking stack. The packet must traverse the isolated network namespace within the container, move to the host namespace, and pass through a virtual switch, as depicted in Figure 2.5. These multiple steps in sending packets in a container overlay network result in significant per-packet communication overhead, making the current form of overlay networks unsuitable for time-sensitive Edge applications that require low latency and real-time response. The extra layers of network traversal add considerable latency

to the communication process, making it challenging to meet the strict requirements of these applications. This highlights the need for further optimization and advancements in container overlay networks to ensure they can meet the demands of time-sensitive Edge applications.

3 A QoS-aware Architecture for the Cloud Continuum

Contents

In this Chapter, we present a comprehensive examination of a layered architecture that spans the Cloud Continuum, consisting of various components that work in conjunction to provide support for next-generation applications with varying QoS requirements. At the same time, we will delve into the current state of the art for each architectural component and elaborate on how our proposed solution enhances it.

The proposed QoS-aware Architecture for the Cloud Continuum, shown in Figure 3.1, comprises three main layers: the *Infrastructure layer*, the *Virtualization layer*, and the *Application layer*. Additionally, the architecture features two other components: an end-to-end QoS management and monitoring component that spans all layers and a horizontal component that transparently provides the services offered by the Virtualization layer. These components work together to support the architecture and the Edge Cloud applications deployed. It should be noted that each component of the architecture can be deployed across the Cloud Continuum, ranging from the data center to the network edge nodes.

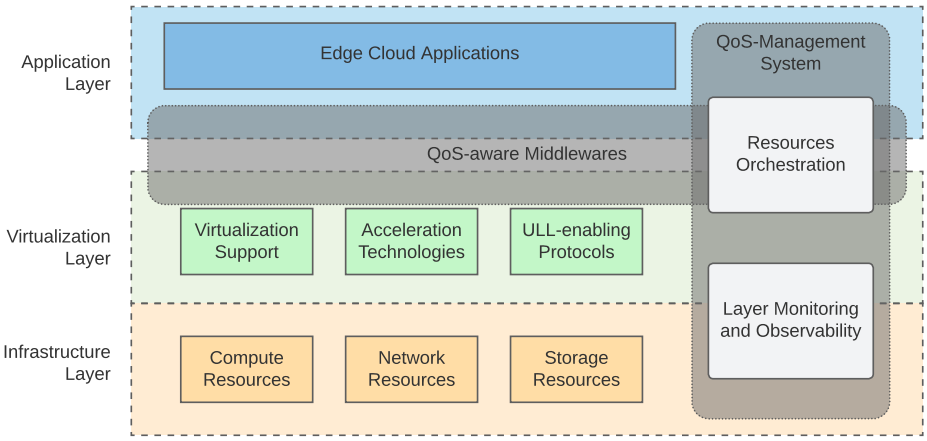


Figure 3.1: The proposed QoS-aware Architecture for the Cloud Continuum exposes the three main Layers.

3.1 The Infrastructure Layer

The foundational layer of the architecture is the *Infrastructure layer*, which encompasses the underlying physical resources such as computing, network, and storage capabilities. To cater to the varying resource requirements and Quality of Service demands of future applications, the physical resources within this layer must be managed unobtrusively. Additionally, these resources can be distributed across multiple levels of the Cloud Continuum, providing a flexible and scalable foundation for the architecture.

One key concept that plays a crucial role in managing the resources in the infrastructure layer is the concept of Slicing. Slicing refers to partitioning the physical resources into multiple isolated and customizable slices that can be allocated to different tenants, applications, or services based on their specific requirements. Each slice acts as a separate and isolated entity within the infrastructure layer, providing a dedicated set of resources that can be tailored to meet the requirements of a specific tenant or application.

Slicing enables the efficient and effective utilization of physical resources by allowing for the creation of multiple virtualized instances of the resources within the infrastructure layer. This enables the deployment of multiple services or applications with different QoS requirements and resource demands within the same physical infrastructure, maximizing resource utilization and reducing costs. Additionally, slicing provides the necessary level of

customization, allowing tenants and applications to control the resources they receive and enabling the deployment of innovative and highly customized services and applications.

In this scenario, integrating QoS-aware network management and configuration is imperative as it lays the foundation for more advanced service and application-aware capabilities. Our architecture primarily focuses on Ultra-low Latency communications, with a particular focus on the TSN protocol. Therefore, discussing some of the most relevant management solutions in the current literature is essential.

In [53], an architecture for real-time systems called Time-sensitive Software-defined Network (TSSDN) is presented. The objective is to schedule and route time-triggered traffic using commodity hardware via a software-defined networking approach. The authors present several scheduling algorithms, assigning time slots to time-triggered flows and minimizing in-network queuing while maximizing the number of co-existing flows.

Gutiérrez *et al.* propose a heuristic capable of the run-time configuration of fog-enabled TSN [34, 60]. The authors, in their works, take into account time-critical flows that can appear and disappear over time. To this end, they adopt a configuration agent architecture that reacts to network (traffic) changes.

Gerhard *et al.* in [31] presents an approach that combines SDN and TSN with an emphasis on network management and configuration. Their software-defined Flow Reservation (SDFR) architecture implements the IEEE 802.1Qcc standard and can be integrated with existing SDN solutions. The proposal allows for the configuration of time-triggered traffic flows exploiting a southbound interface protocol.

Some encountered literature work relies upon evaluating their proposals through simulation studies and reconfiguration events, which require some run-time capabilities and are often neglected and not provisioned. The mechanisms necessary to perform the same steps in end devices are often omitted on static monitoring and reconfiguring network devices.

Chapter 4 of the present thesis elaborates on a centralized architectural design encompassing different functional components. This design expands the notion of self-(re)configuration and monitoring to all elements in a Time-Sensitive Networking (TSN) network, thus including end devices. Furthermore, the proposed QoS management architecture has been subjected to experimental validation on a practical TSN testbed, showcasing its ability to adapt dynamically to changes in the network environment.

3.2 The Virtualization Layer

3.2.1 Network Virtualization

The virtualization layer of our architecture is a key component in providing network virtualization mechanisms capable of supporting ULL communications. This layer must support both classical virtualization techniques using virtual machines and container-based lightweight virtualization methods. In particular, ULL communication protocols, such as TSN, need two. This layer has slicing systems that divide virtual networks into slices to accommodate traffic flows with different QoS requirements, which are essential for ULL communications. In addition, this layer also addresses the challenge of propagating a global reference time across all virtualized layers, which is critical for synchronizing all elements participating in Time-Sensitive Networking communications.

In [9], Xen and KVM are proposed as suitable real-time hypervisors for industrial control systems; however, the evaluation of their characteristics does not consider the use of deterministic network communication protocols such as TSN. Another recent work, [33], proposed a container-based architecture for the flexible reconfiguration and redeployment of specific process control systems, but it does not apply to virtual machines. The authors evaluate their proposal through a PTP-synchronized testbed. They show that low-latency QoS requirements can be met. Still, they do not take advantage of the deterministic scheduling techniques defined in TSN and do not focus on the impact of network virtualization technique on latency overall. In [48], the feasibility of TSN virtualization was explored, and three different approaches to enhance hypervisors for time-triggered communication were discussed. However, the focus of the study was on architectural principles, and the evaluation was conducted through simulations without validation on an actual testbed. This study builds upon the previous related works by evaluating the use of TSN in virtualized industrial control systems using a kernel-bypassing network virtualization approach. The performance of the virtualized TSN application is compared against that of the same application running on bare-metal hosts, and the results are validated on an actual testbed. This provides a comprehensive evaluation of the use of TSN in virtualized industrial control systems and demonstrates the feasibility of meeting ULL constraints using a kernel-bypassing network virtualization approach.

The TSN virtualization approach presented in Section 5.1 of this thesis is based on several insights the works just discussed. A key aspect of our approach is the use of fast packet processing techniques, for which a detailed comparison is presented in [11]. The latter demonstrates the superior performance of kernel-bypassing approaches over traditional methods. However, this work compares containerized applications (not VMs) with a different traffic pattern than the one used in our study. Additionally, kernel-bypassing techniques for meeting the constraints of ULL applications have mainly been explored in the field of Software-Defined Networking (SDN), where network functions executing in VMs need to process packets at high speed [80]. Our work is complementary to this effort as it addresses the user-controlled infrastructure rather than the provider-controlled portion.

Moving to lightweight virtualization, previous research on the containerization of critical application components has mainly focused on orchestration strategies and CPU scheduling, such as in works like [36, 73]. These studies aim to find the best methods for placing components on appropriate resources and ensuring that those resources can schedule the execution of containerized applications according to their requirements. However, these studies do not take into account network and system-related aspects. Our proposal complements these studies as we believe that network and computing resources for edge applications should be orchestrated together.

Despite the importance of networking for edge applications, researchers have paid less attention to the networking requirements of critical applications. For example, Ara et al. [11] evaluate different kernel-bypass approaches for inter-container communications, highlighting the potential of DPDK as a network accelerator compared to the kernel-based approach. However, their contribution is limited to a framework for performance evaluation.

Slim [84] proposes a solution to reduce the processing overhead on container overlay networks by avoiding processing packets multiple times on the same host. Instead, it defines a component that intercepts calls to the socket API and directly translates network addresses from the overlay into the host namespace (and vice versa). This way, packets traverse the kernel networking stack only once. SocksDirect [49] uses the same interception technique to re-route packages on an accelerated kernel-bypassing datapath, but this is possible only with the *host* container networking mode. Both these works introduce the idea of accelerating container inter-networking, showing significant performance ad-

vantages for many applications built on top of them. However, these solutions are not integrated with standard production-ready technologies like Kubernetes, and they target data center environments and focus on reliable connection-oriented transport protocols (TCP). They do not support time-sensitive applications such as TSN, an essential requirement for edge applications. In this work, we adopt similar techniques (socket interception, kernel-bypassing) to accelerate network operations. Still, we also provide guarantees on connection determinism (through TSN) and implement our solution as a plugin for high-standard development and deployment technologies.

Furthermore, using TSN in virtual environments is a relatively new trend, as the standard was initially intended for bare-metal industrial applications. Leonardi et al.[48] first hypothesized this possibility, identifying three distinct architectural approaches to enhance hypervisor-based virtualization with time-triggered communication.

3.2.2 Transparent Network Acceleration Access

In addition to the network virtualization capabilities, due to increased overhead often results in utilizing virtualization techniques compared to bare-metal execution. The virtualization layer has to overcome this challenge and ensure optimal performance by exploiting multiple acceleration technologies. The goal is to manage and deliver these technologies to upper layers and applications transparently while ensuring optimal performance.

Outside the Edge Cloud, few works have at least partially attempted to provide a uniform interface for the emerging network acceleration options in data center environments. The first effort in this direction was `libfabric` [2], a library that enables RDMA applications to run even without the necessary supporting hardware. Instead of using the native API, developers code against a transparent set of communication primitives. If suitable support is available (e.g., RDMA NIC), then these primitives are efficiently translated to the native API; otherwise, their semantics are emulated using alternative transports (e.g., TCP/IP). However, the `libfabric` interface is still very low-level, mainly targeting High Performance Computing (HPC) applications, and non-RDMA transports are intended for debugging purposes. RocketBufs [37] proposes a buffer abstraction on top of different network acceleration technologies in the cloud computing domain. This system has several limitations. It does not allow the same application to specify different requirements for

different flows, making it unsuitable for the Edge environment. It also shows a significant performance overhead compared to raw RDMA.

The most significant work in this field is Demikernel [81], which defines a set of userspace libraries that replace the operating system datapath primitives intending to accelerate new or existing datacenter applications. Each library implements the same general-purpose interface for specific networking (DPDK and RDMA) or storage technology, integrating the missing OS functionalities (e.g., network stack) when needed. Furthermore, Demikernel supports the different data paths as shared libraries. Hence, it is not designed to allow the same application to access multiple network technology concurrently, a feature usually required for Edge applications. Even more importantly, if multiple Demikernel applications run concurrently on the same host, each will activate an instance of its datapath. For example, each DPDK application would require a separate network card and will spin on one or more cores.

3.3 The Application Layer

Next-generation IoT applications with diverse QoS requirements are implemented in the application layer. This layer is closely integrated with the various middleware components, enabling applications to benefit from the discussed technologies without requiring in-depth knowledge of their implementation. An example of a middleware implemented in this layer will be discussed in Section 6.1. In contrast, two examples of edge applications that seamlessly exploit the underlying layers are analyzed in Section 6.2. The middleware discussed provides a comprehensive, IoT-focused approach to QoS monitoring, control, and management for different virtualized resources (such as networking and processing). It specifically targets deployment environments that utilize edge cloud resources to enable the Serverless paradigm in the Cloud Continuum.

To the best of our knowledge, the design and implementation of a middleware able to exploit and coordinate different QoS mechanisms across the stack of virtualized Function as a Service (FaaS) invocations for the Cloud Continuum are entirely novel in the existing literature. However, several works have proposed solutions to some of the challenges addressed by our proposal. Many of them not only paved the way for the development of TEMPOS but also inspired some architectural and technological choices that we adopted.

The following concise section aims only to be a representative excerpt of the most influential published research papers related to that.

The opportunistic usage of Edge Cloud resources to improve latency and jitter has been extensively discussed in [13, 56] and also represents one of the key factors pushing for wide adoption of this computing model [38].

The coordination and coupling of different prioritization mechanisms is not a recent issue, but with the recent advent of next-generation networking, it has gained an increasing research interest. Since the earliest distributed systems, the need for concatenation of mechanisms at different stack levels has been a primary problem. To tackle resource orchestration and partitioning while guaranteeing QoS levels at the Edge, [24] proposes DRAGON: that paper describes some implementation insights about DRAGON. It evaluates its performance benefits if compared with traditional orchestration approaches.

The introduction of middleware for concatenating QoS-aware composition mechanisms is a frequent design pattern applied in the literature to reduce complexity. In [65], the authors propose a technique to couple priority and reservation-based OS and network QoS management mechanisms through Distributed Object Computing middleware with adequate performance results. In [69], the authors present a middleware built on CORBA for providing distributed soft real-time applications with a uniform API to reserve heterogeneous resources with real-time scheduling capabilities in a distributed environment: that solution introduced uniform interfaces to support the reservation of CPU, disk, and network bandwidth on Linux systems.

Even if Serverless computing and, in particular, FaaS platforms are relatively novel, some platform improvements have already been proposed in the literature to achieve better FaaS performance and, in particular, latency reduction. Some papers have proposed deploying serverless platforms on edge nodes to achieve better QoS [14]. Using different invocation methods to speed up function startup has been proposed as the exploitation of cross-compiling to achieve faster executables. For example, in [68], the authors propose *Faaslets*, an isolation abstraction that exploits WebAssembly to achieve good isolation and fast function startup; they also propose an additional optimization with a mechanism to restore from already initialized snapshots, thus improving platform throughput and tail latency. In the proposed project Catalyzer [27], the authors propose a serverless sandbox system to enhance function startup and isolation. To provide fast startup, Catalyzer exploits a checkpoint mechanism to skip initialization and a new OS primitive to reuse

the state of the running sandbox; this results in a relevant reduction of the startup time of function invocations, up to less than 1 ms in the best cases.

4 The Infrastructure Layer

Contents

This chapter explores the concept of layered middleware for OT/IT convergence, which serves as a case study for examining the challenges and opportunities presented by applications that expose traffic flows with different Quality of Service requirements and extend across the Cloud Continuum. The chapter then introduces an end-to-end QoS management system for TSN networks, which aims to address the challenge of ensuring consistent QoS in the presence of diverse traffic flows and changing network conditions.

4.1 A Layered Middleware for OT/IT Convergence

In this first study, we outline a strategy to effectively blur the boundary between the OT and IT domains, enabling fast, secure, and reliable exchange of operational OT data to the IT domain. The approach involves a Gateway component positioned on the OT/IT boundary and a two-layered middleware solution designed to fulfill each domain's functional and non-functional requirements.

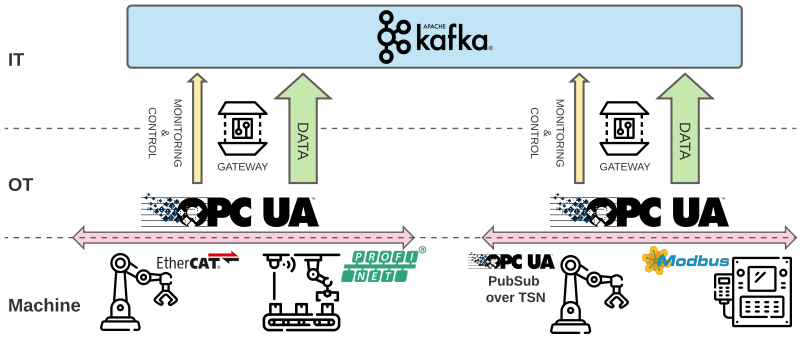


Figure 4.1: Architecture overview diagram.

4.1.1 System Components and Integration

Our solution leverages OPC UA Pub/Sub for Machine-to-machine communication at the OT level while using Apache Kafka, a high-speed, low-latency MOM, for data collection from multiple OT sites to the IT domain. The OPC UA standard enables data transfer to higher layers and serves as a low-level interoperability protocol for fast data transmission. In the IT domain, Kafka enables secure and reliable management of large volumes of data while providing an extensible framework with a rich set of tools for IT.

The system architecture is presented in Figure 4.1. The dotted line separates the OT, IT, and Machine layers. The Machine layer is made up of assets that use a variety of low-level, heterogeneous protocols, with some adhering to standard specifications with open-source implementation (e.g., OPC UA over TSN), and others being proprietary, closed-source, and not interoperable (e.g., EtherCAT, PROFINET, Modbus-TCP) [23]. Our proposal employs the OT layer as a homogenization layer, abstracting the technical details of specific protocols from the upper layers. The design of the OT layer is pluggable, enabling users to add components to the infrastructure through specific adapters dynamically.

The adapter, after configuration, collects data using the machine-specific language and exposes the machine information model through the common OPC UA information standard. This ensures a consistent representation of information between the machine and the upper layers. Our design can accommodate various adapter deployment strategies based on the available computational resources on the industrial asset. If the machine has enough resources, the adapter can be deployed directly on it; otherwise, it can be deployed elsewhere and connected to the machine through the network.

On top of the OPC UA protocol, we use the OPC UA Pub/Sub specification for message exchange within the single shop floor, where heterogeneous traffic coexists and varies from safety-critical control traffic to best-effort ones. Data are gathered by the Gateway component, which listens to OPC UA Pub/Sub endpoints and sends data to the Kafka MOM. Gateways can be customized using configuration files that specify machine addresses and registers that must be manipulated and re-exposed on Kafka topics. An example of the configuration file can be found in Section 4.1.2.

One of the primary goals of the Gateway is to differentiate between heterogeneous flows, including raw sensor data and data derived from monitoring processes on the shop floor. Raw sensor data represent information that industrial machines expose, containing data regarding their internal state. At the same time, the monitoring flow consists of data and metrics related to networks, industrial processes, and more.

On the producer side, prioritization of monitoring and controlling data traffic is crucial for efficient system operation. Different topics and partitioning levels are utilized for different data types to ensure this. Monitoring and control topics are configured with a single partition and higher replication degree to guarantee the total ordering of sent messages and increased fault tolerance. Meanwhile, raw sensor data topics are configured with multiple partitions and lower replication degrees to achieve higher input and throughput rates and reduced memory usage.

At the consumer end, Apache Kafka provides differentiated semantics for commit management, specifically At-Most-Once, At-Least-Once, and Exactly-Once [26]. The Exactly-Once semantic is utilized for monitoring and controll data, while for data traffic, the At-Least-Once semantic is utilized to ensure faster reading.

Regarding security, Apache Kafka offers Access Control Lists (ACLs) through the use of ACL Authorizers [25]. This feature can play a crucial role in maintaining the confidentiality of industrial data by allowing for the application of fine-grained access policies on topics through the definition of authorized reader and writer groups.

When considering the stringent latency requirements at the OT layer, direct exposure to the external world is not assumed, and as such, significant security mechanisms are not in place. Instead, the software in this domain is certified not to pose any threat. Despite this, it is still important to investigate potential solutions for improving security, and the adoption of lightweight security mechanisms is currently being considered as a solution to this issue.

4.1.2 Bootstrapping the System

The system can be bootstrapped by providing configuration parameters that bind the components together and facilitate the exchange of structured information with the IT layer. The bootstrapping process consists of the following steps:

- *Configuration*. The Gateway component is provided with a structured configuration file that contains information about OPC UA-enabled assets, such as IP addresses and multicast network groups, for registering the internal state of industrial assets. The configuration file also includes information about the Kafka endpoints and topics for publishing messages, QoS level mappings, and publication frequency. For clarity, an example configuration file is included in Listing 4.2.

- *Discovery*. The Gateway verifies data representation by querying the OPC UA server(s). During this phase, the Gateway also checks the consistency of OPC UA-reported registers with the configuration file.

- *Operation*. Once the discovery phase is successful, the Gateway subscribes to the multicast network groups and begins to flow messages. Upon receipt, the messages are unmarshaled to JSON in a specific protocol dialect. The messages can be sent on different channels based on their data type. For example, messages classified as control flows are sent with a high-quality and ordered level, guaranteeing fast and reliable delivery. In contrast, sensor messages can be sent with a non-ordered semantic based on customer-specific policies.

After the bootstrapping process is complete, the data can be fetched from the Kafka topics, and multiple consumers can access the data based on specific access policies. The decoupling of the OT and IT layers through the use of a lightweight and configurable Gateway enables advanced control features to address reliability and scalability in scenarios of high ingress traffic.

4.1.3 Experimental Analysis

The experiment's objective is to demonstrate our proposed architecture's efficacy in fulfilling diverse constraints while maintaining effectiveness. The experiment aims to exhibit

```
{
  "machines": [
    {
      "name": "MACHINE_1",
      "ip_address": "192.168.0.3",
      "transport_profile": "http://opcfoundation.org/UA-Profile/Transport/
          pubsub-udp-uadp",
      "network_address_url": "224.0.0.18:4840"
    }
  ],
  "kafka": {
    "cluster_ip_addresses": [
      "192.168.1.2"
    ],
    "topic": "myTopic"
  },
  "publishers": [
    {
      "data_group_name": "datagroup-1",
      "writer_group_id": "1",
      "registers": [
        "PRESSURE_1",
        "OVEN_TEMPERATURE_1"
      ],
      "interval": "100",
      "QoS": "data"
    }
  ]
}
```

Figure 4.2: Example of JSON configuration file used by the Gateway.

the capacity of the architecture to meet the QoS requirements of low-latency flows at the OT layer while concurrently evaluating its capability to provide high-throughput and quality data to the IT layer. For this purpose, we have established a testbed as depicted in Figure 4.1.

Table 4.1: Testbed deployment: components, OS, and hardware characteristics.

Name	Component	Operating System	CPU	RAM	Network
Node 1	Machine Simulator 1	Ubuntu 20.04.3 LTS	Intel Core i5-2400 CPU @ 3.10GHz	8 GB	1 Gpbs
Node 2	Machine Simulator 2				
Node 3	Gateway				
Node 4	Kafka Consumer				
Node 5	Apache Kafka	Ubuntu 20.04.3 LTS	Intel Core i5-3470 CPU @ 3.20GHz	16 GB	1 Gpbs

Experimental Settings

To evaluate the total functional potential of our proposal, we have established a real testbed comprising five nodes, each with various hosting functionalities related to the OT and IT layer. The nodes are interconnected through a dedicated network with a 1 GB switch. While this network configuration may not be representative of a typical deployment scenario, it serves the purpose of this study, which is to test and assess the functional components of the architecture in an operational setting. The deployed nodes' characteristics are listed in Table 4.1 for completeness.

The infrastructure comprises two nodes dedicated to traffic simulation. The simulation relies on custom-built software packages that emulate industrial machinery traffic based on actual machinery specifications. The simulation process is described in detail in a previous [20].

Node 1 simulates an industrial asset by exposing its internal operational state via the Modbus/TCP protocol. A Modbus adapter at the machine layer reads and extracts the information in a protocol-agnostic format, which is then structured using the OPC UA data model. Depending on the configuration and purpose, the adapter acts as both a subscriber and publisher of data. The structured data is transmitted through the OPC UA Pub/Sub protocol and made available to all other entities in the network. Node 2, acting as an OPC UA Subscriber, receives the data emitted by Node 1 and simulates a typical sensors-to-controller scenario.

Node 3 hosts the Gateway component and subscribes to the messages sent by the simulator in Node 1, the same messages received by the simulator in Node 2. Node 4 is a Docker-based Kafka deployment that receives messages produced by the Gateway acting as a producer. Finally, Node 5 hosts a Kafka consumer, consisting of a custom program that receives messages from specific Kafka topics. This consumer enables us to estimate the transit time for a message from Node 1 to reach the IT department of the factory or the Cloud.

PTP (Section 2.2.1) synchronizes nodes for accurate time measurement. The node hosting the Gateway is configured as the controller and serves as a reference clock for all other participants in the PTP domain while the others act as responders.

Experimental Results

The proposed architecture results were evaluated by measuring the latency from the OT-to-IT layer under varying traffic regimes. Figures 4.3 and 4.4 show the latency measurements, respectively, in the OT layer (Node 1 to Node 2) and the end-to-end latency from the OT layer to the Kafka consumer in the IT layer (Node 5). Latency was calculated as the time interval between receiving and transmitting messages at the application layer, with message rates ranging from 400 to 1500 messages per second.

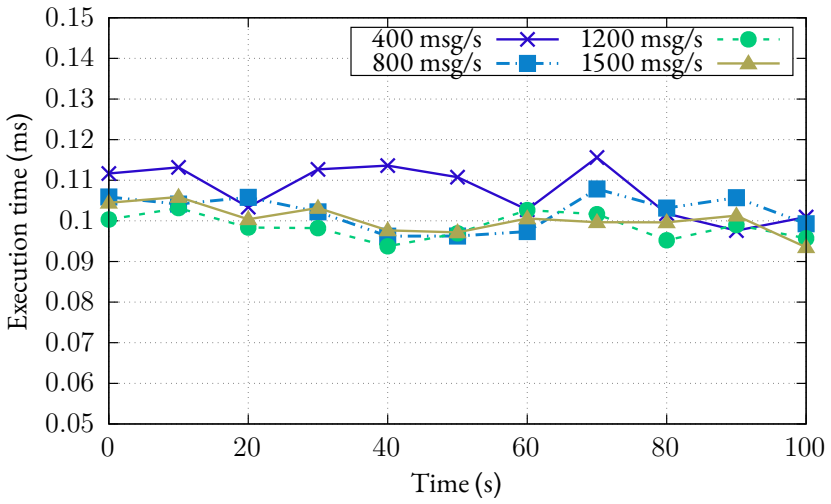


Figure 4.3: Machine-to-machine communication latency under varying message load of the OT layer.

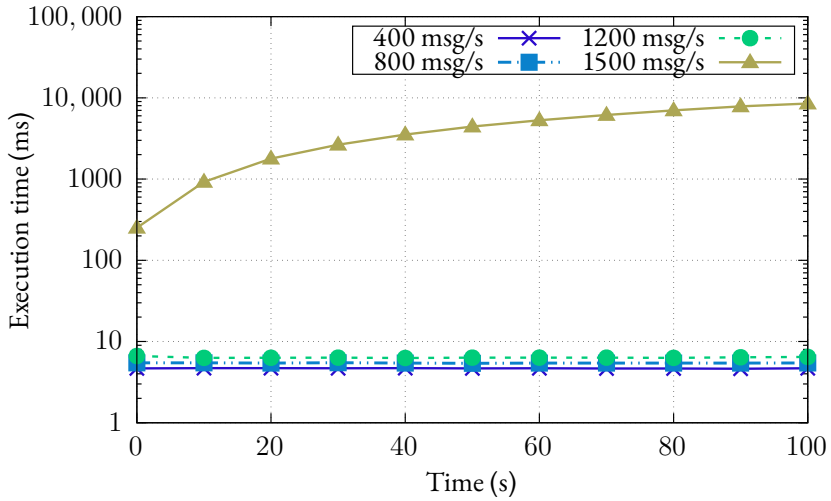


Figure 4.4: Machine-to-consumer communication latency under varying message load of the IT layer.

The results of our experiment are shown in Figures 4.3 and 4.4. Figure 4.3 shows that the latency between the two simulated machines remains stable while increasing the number of messages/second up to 1500 per second. This is a critical result, as it demonstrates that the proposed system can maintain low latency communication between the OT layer machines, a requirement in industrial control systems. The observed latency is always below 1 ms, which is the typical latency expected in the OT layer, particularly for the communication between different machines or Programmable Logic Controllers (PLCs).

Figure 4.4 shows the end-to-end latency measured at the IT level for the same message rate range, exhibiting a latency that is an order of magnitude higher than the one observed in the OT level. This increase is expected when considering the additional components and processing the message must go through before reaching the IT layer and the latency introduced by the Kafka MOM features. The Kafka MOM has been configured to manage the forwarding of the messages to the consumer by imposing a total ordering and ensuring exactly-once semantics, which is particularly important when conveying safety-critical information from the OT.

The effects of the aforementioned semantics are visible in the 1500 message/second configuration, causing a significant increase in latency. In this setting, the rate mismatch of servicing the incoming data, marshaling messages to the IT-layer compliant format,

and emitting the messages to the output queue creates an increasing backlog of messages over time. This trend highlights the importance of implementing selective pre-processing in the OT layer, such as filtering and aggregation, to coordinate the different layers better and alleviate the burden at the OT/IT bridging point.

This conclusion is further confirmed by the data shown in Figure 4.5, which shows the Gateway CPU usage, evidencing an increase in the CPU usage trend, augmenting with the increase of the message arrival rate. Despite this increase, plenty of resources can still be devoted to other computational tasks.

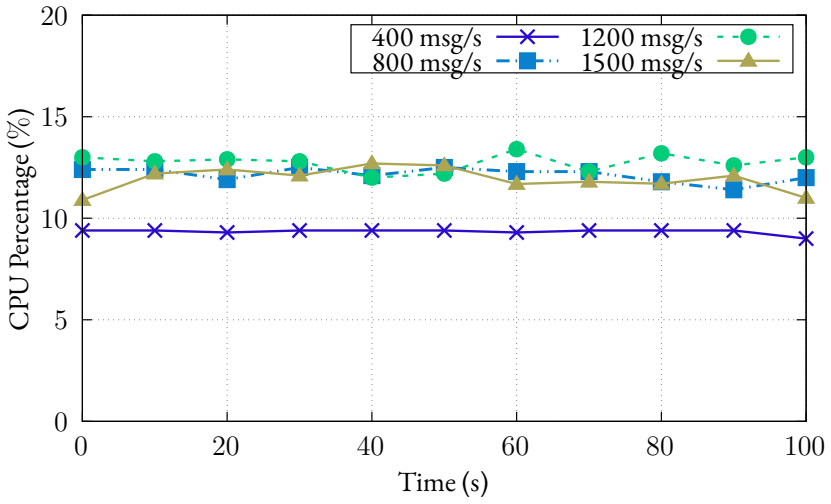


Figure 4.5: Gateway CPU usage under varying message loads.

The experiment's findings validate our hypothesis of implementing customized solutions for the Operational Technology and Information Technology layers while utilizing an Edge Cloud infrastructure that houses specific functions connecting the two domains. Analysis of the OT layer reveals latency values within sub-millisecond ranges, meeting the stringent latency demands for critical OT traffic. Furthermore, the utilization of Kafka affords increased versatility in the IT layer, enabling traffic categorization through a comprehensive set of configurable options.

4.2 End-to-end QoS Management in TSN Networks

One of the key factors of the Cloud Continuum model and the middleware just presented is their flexibility. Thus, an essential aspect of systems of this type is the self-configuration of end devices and the network. Applications on end devices must be able to announce their communication needs to the network control plane, ensuring QoS for all types of data traffic. In addition, the network must perform rapid reconfiguration, that is, react to changed configurations, with a reasonably short delay during execution.

For these reasons, we propose a centralized architectural solution and functional building blocks, extending the concept of self-(re)configuration and monitoring to all network elements, including end devices. Then, the proposed solution is validated on a real TSN testbed, showcasing its capability to adapt to network changes dynamically.

4.2.1 System Configuration

Since the end devices considered in this thesis project will be mainly based on Linux systems, we have identified the perfect candidate for their configuration in the Netlink protocol. Netlink is used to transfer information between the kernel and user-space processes [54]. It consists of a standard sockets-based interface for user space processes and an internal kernel API for kernel modules. Netlink is made up of several subsystems and protocols called Netlink families. These families are used to select the kernel module or Netlink group we want to communicate with. We will use the `NETLINK_ROUTE` family, often referred to as `rtnetlink`. This subsystem allows receiving routing and link updates. It may also modify the routing tables (IPv4 and IPv6), IP addresses, link parameters, neighbor setups, queuing disciplines, traffic classes, and packet classifiers.

To support a stream of TSN packets, we will use two new queuing disciplines (qdiscs) built into the Linux kernel. The first, called `taprio` qdisc, implements a simplified version of the scheduling defined by IEEE 802.1Qbv, which allows the configuration of a sequence of gate states. Each gate state permits outbound traffic from a specific subset of traffic classes. This first qdisc supports what we have called Time-Aware Scheduler. The second discipline is called `etf` (Earliest TxTime First) qdisc. This queuing discipline allows applications to control when a packet must be removed from the network device's queue. If offload is configured and supported by the Network Interface Controller (NIC),

it will also check when packets leave the NIC. This last property allows forwarding the packets within the scheduling windows and, above all, in the time slots set through TSN.

As already pointed out several times, all the network devices must be synchronized for communication with ultra-low latency constraints. In the case of end devices, the programs used for this synchronization are part of the `linuxptp` package and are, respectively:

- `ptp4l` represents an implementation of the Precision Time Protocol (PTP) according to IEEE standard 1588 for Linux.
- `phc2sys`, the program that synchronizes two clocks in the system. Typically, it synchronizes the system clock to a PTP hardware clock (PHC), synchronized by the `ptp4l` program.

4.2.2 System Architecture

Fig. 4.6 provides a high-level overview of our proposal. Architecturally, we follow the centralized management and configuration model comprising the CUC, CNC, and UNI entities. The solution comprises several components, each of which plays a particular role in managing and servicing the different actors of a TSN network. The following sections will explore these components' roles in further detail.

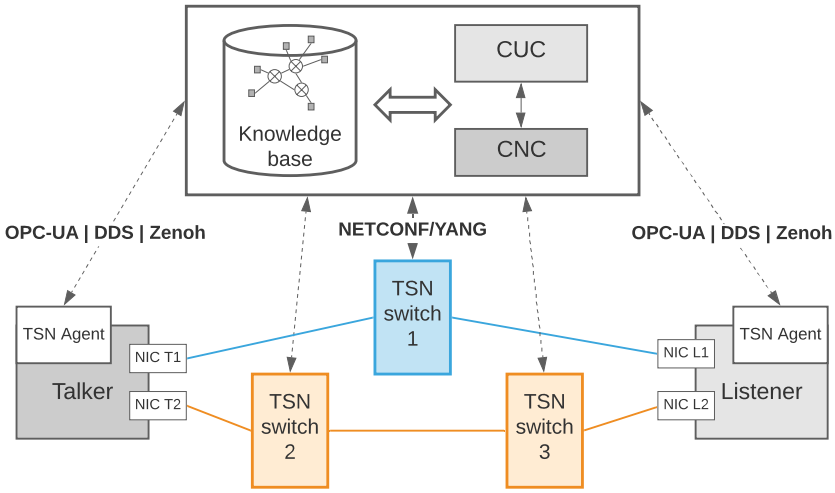


Figure 4.6: Proposed QoS Management Architecture.

Centralized Network and User Configuration

We designed two modules for infrastructure configuration, assuming that network devices and end devices utilize distinct configuration methods.

The Centralized Network Configuration component engages in direct communication with network devices through the NETCONF protocol, gathering information on their capabilities and configurations, which is then stored in the Knowledge Base (as described in Section 4.2.2. Specifically, communication is established through NETCONF over SSH [78]. Additionally, the Centralized Network Configuration (CNC) module is responsible for managing network devices and ensuring adherence to the QoS requirements stipulated by TSN endpoints for communication.

The CNC implements the following mechanisms to achieve its goals:

- Management of PTP-based synchronization service by sending commands to initiate the PTP service directly to switches.
- Establishment of VLANs to distinguish between communication flows with varying QoS needs.
- Distribution of network schedule, including the configuration of communication window parameters and gate control lists for ports connected to end devices.

From a monitoring perspective, the module leverages the NETCONF Event Notifications protocol to monitor network devices by subscribing to relevant notification events. This allows real-time event notifications to be received by the CNC from the switch, providing information on the switch's current state. Through this mechanism, it becomes possible to monitor various metrics and events, such as configuration changes, errors, and performance metrics related to the data plane.

The second module, denoted as the Centralized User Configuration, works in conjunction with the TSN Agent (as described in Section 4.2.2) to orchestrate the configuration of TSN streams, ensuring that they meet the specified QoS requirements as imposed by end devices, while also managing the PTP service running on them. Unlike the CNC, the Centralized User Configuration (CUC) module is not built on the NETCONF protocol. Still, instead, it has been designed to be compatible with various communication protocols such as OPC UA PubSub, DDS, and gRPC. The module communicates with the end

device through messages containing configuration parameters, which can be sent in either direction between the CUC and the device. Upon receipt of a message, the TSN Agent software module installed in the end device interfaces directly with the operating system and uses the parameters to configure all TSN-related services.

Knowledge Base

The Knowledge Base serves as the repository for information regarding all managed elements in the network, including end devices. The information stored here encompasses a wide range of data, including device configurations, physical and overlay network topologies, and active communication flows. The knowledge base is constructed through periodic updates sent by all participants in the TSN network through the CUC-CNC interface. This data can be easily accessed and queried, for example, in the event of reconfiguration activities within the network. A centralized knowledge base facilitates effective management and orchestration of TSN communication, ensuring that network devices are configured optimally to meet the specified QoS requirements.

TSN Agent and Fault Detection

The TSN Agent is a crucial component in the TSN architecture and represents the module deployed on the end device tasked with communication with the CUC. The main role of the TSN Agent is to request the setup and configuration of viable QoS-aware TSN streams from the CUC.

To act upon the request, the CUC first queries the Knowledge Base to retrieve all relevant information about the network and its components. Based on this information, the CUC computes a viable schedule and sends the configuration parameters to the TSN Agent. Furthermore, the CUC communicates with the CNC to manage the network path configuration and set up the end-to-end TSN stream.

The TSN Agent exploits Netlink, a standard socket-based interface that allows user space applications to communicate and modify the settings of kernel modules. Specifically, the TSN Agent utilizes the `NETLINK_ROUTE` subsystem, often referred to as `rtnetlink`, to receive routing and link updates. This interface enables the TSN agent to modify routing tables (IPv4 and IPv6), link parameters, neighbor settings, queuing disciplines, and more. All of these features are essential to support and enact the concept of a TSN stream.

To implement the TSN stream in end devices, the TSN Agent utilizes two new queuing disciplines (`qdiscs`) built into the Linux kernel:

- `taprio` (Time Aware Priority Shaper) implements a simplified version of the scheduling defined by IEEE 802.1Qbv (Section 2.2.2).
- `etf` (Earliest TxTime First) qdisc, which allows applications to set a transmission time for each packet, information the scheduler uses to dequeue and forward it over the TSN network.

Regarding the synchronization of the end devices, the TSN Agent implements this feature is exploiting the `linuxptp` package.

In terms of monitoring, the TSN Agent exploits the capability of Netlink to subscribe to one or more multicast groups to receive networking events. The TSN Agent subscribes to the group on `RTMGRP_LINK` of the `NETLINK_ROUTE` family, which allows it to receive events related to network interface configuration and status updates.

An essential aspect of the TSN infrastructure is to monitor the system and react if a fault is detected. In case of a fault, the priority is to activate an alarm mechanism. However, a dynamic reconfiguration of the communication flows can also be performed if possible. For example, two communication channels connect the end devices in a reference system. If one channel is unavailable, it is possible to reconfigure the TSN stream to continue functioning using the available channel.

The system's architecture presents various reconfiguration mechanisms, each operating distinctly. One key aspect of the architecture is the utilization of the Netlink protocol, which enables real-time detection of changes in the state of network interfaces. The system can trigger reconfiguration processes by receiving events related to these changes, i.e., link creation, deletion, or status changes.

One of the reconfiguration mechanisms, exclusive to the talker side, involves monitoring error messages generated during message transmission. The Linux kernel provides a mechanism for indicating the success or failure of a message transmission request at the end of the send request. In the event of an error, such as an attempt to forward a packet outside the communication window set via `taprio` qdisc or if the send time, which we can see as a deadline, set via `etf` qdisc is not met, the error message queue is read and a reconfiguration process may be triggered.

The final reconfiguration mechanism leverages information from both the talker and listener sides of the communication. This mechanism is based on the fact that communication time slots are configured through periodic publication cycles with a specific deadline. The listener can determine if a packet has been delivered within the time-aware window by knowing the publication cycle and the deadline. However, this information may be incomplete, as the packet may still be delivered in the next communication window along with the next packet. To address this issue, a unique identifier, such as a sequence number, may be included in the message to provide insight into any late or lost packets in the communication path. This information is critical for evaluating essential metrics in ULL communications, such as latency and jitter.

4.2.3 In-the-field Experimental Validation

Experimental Settings

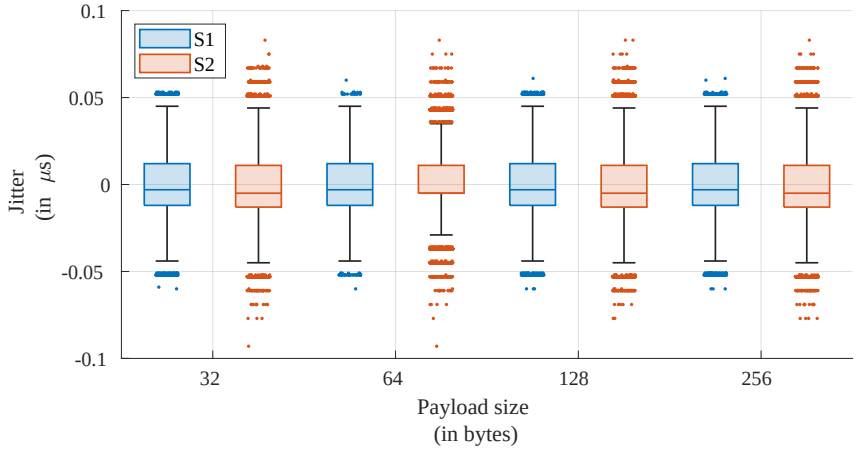
We now report experiments demonstrating the proposed approach's effectiveness in handling reconfiguration events at runtime. To achieve this, a real testbed was constructed based on the architecture depicted in Fig 4.6. The testbed consists of two main components: a talker (T) and a listener (L), each represented by a UP Xtreme board. The boards have 4 TSN NICs (Intel I210), an Intel Core i3-8145UE CPU with 2/4 cores, and 8GB of RAM.

The boards are connected via two distinct physical paths, as shown in Fig 4.6. The first path (represented in blue) connects T's NIC (T1) to L's NIC (L1) through Switch 1 (SW1), which we refer to as Stream 1 (S1). The second path (represented in orange) connects T's NIC (T2) to L's NIC (L2) through Stream 2 (S2), which crosses Switch 2 (SW2) and Switch 3 (SW3). The two streams are configured with different VLANs.

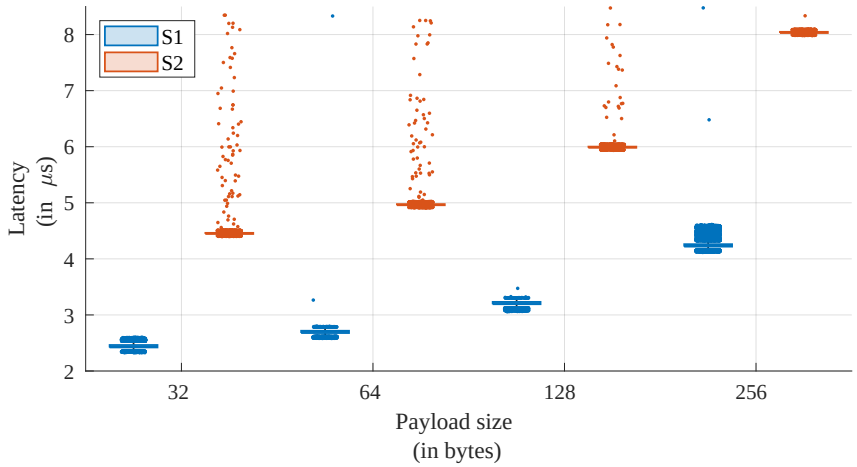
The experiment uses UDP traffic with a payload size of 32, 64, 128, or 256 bytes. A total of 10^5 packets are sent for each configuration, with a regular interval of 1 ms. The first set of experiments measures the latency and jitter of the time-critical flows.

The second experiment demonstrates the ability of the TSN Agents to identify a link-drop event. In this scenario, the talker communicates with the listener using the S1 stream. After a drop occurs between T and SW1, the agent stops the transmission of the talker through NIC (T1), requests a new configuration from the CUC, and reschedules it via the alternative S2. The TSN Agent then updates the network scheduling parameters

and initiates the PTP functionality to resynchronize the clocks before restarting the transmission via NIC (T2). Upon completion, packets can flow through the alternative S2.



(a) Observed jitter for streams S1 and S2.



(b) Public cloud (Cloudlab)

(c) Observed latency for streams S1 and S2.

Figure 4.7: Jitter and latency of the received UDP packets; metrics are expressed for each packet size and stream.

Experimental Results

The experiment results are depicted in Figure 4.7, which displays the jitter and latency values for both the S1 and S2 streams. The graph indicates that the flow QoS meets the

specified 1 ms requirement. It is worth noting that the jitter remains relatively constant across different payload sizes and communication streams. However, the latency increases when communicating via S2 compared to S1. This can be attributed to the fact that S2 passes through two additional switches (SW2 and SW3), leading to an increase in latency by a few microseconds. Despite this, the proposed approach can fulfill the specified QoS constraints for all configurations.

Fig 4.8 shows the result of the reconfiguration scenario, where the communication is initially serviced via S1 and, following a link-drop event, goes through S2. The reconfiguration process is seamless, and the flow can quickly be rerouted to the viable alternative S2 with a downtime period due to the reconfiguration of about 150 ms. This downtime period is minimal, considering the complex nature of the reconfiguration process, which involves updating the network scheduling parameters and communicating with the CUC to request a new configuration.

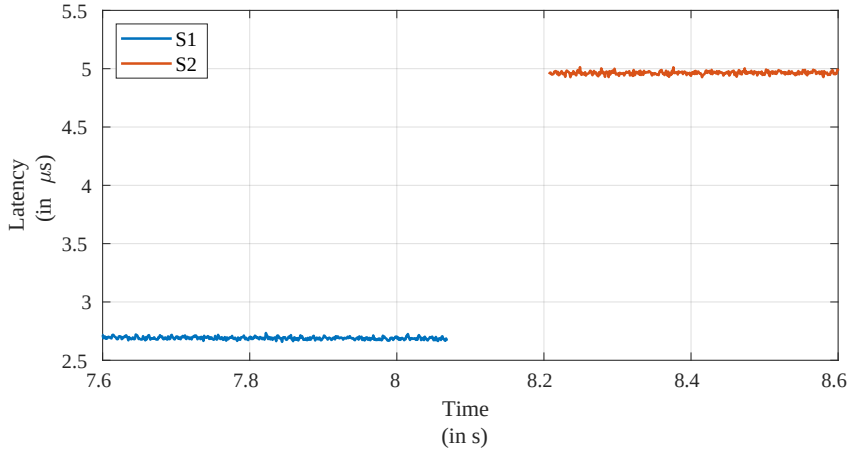


Figure 4.8: Observed latencies before and after the reconfiguration phase.

5 The virtualization layer

Contents

This chapter discusses several virtualization solutions that enable critical TSN-based applications to leverage virtualized networks and processing resources. Next, we present a middleware solution that can transparently unify access to different network communication acceleration technologies.

5.1 TSN-enabled Virtual Environments

As a first solution, we developed a framework that exploits TSN protocols in virtual environments targetting Ultra-low Latency communications in 5G scenarios. Specifically,

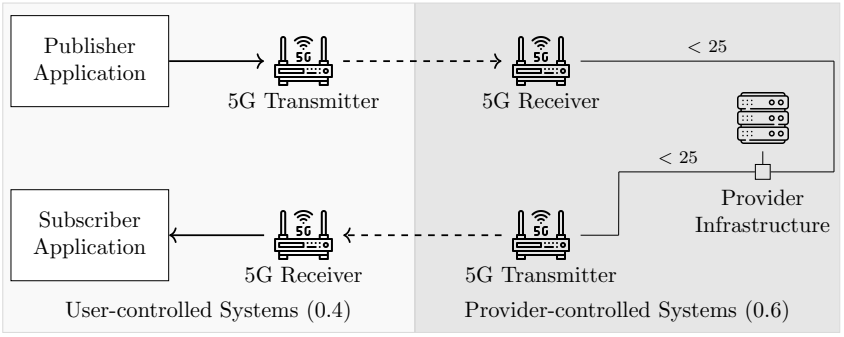


Figure 5.1: A typical latency budget distribution for ULL applications.

as a potential operational scenario in which this solution would be advantageous, we take that of industrial applications in which a robotic device, called a *subscriber*, periodically exchanges messages with a remote software component, called a *publisher*, located in a separate facility and connected via a network managed only partially by the user. An essential requirement in this scenario is that the end-to-end transmission of each message occurs within a minimum and specific time interval, often on the order of 1 ms. To meet this strict deadline, developers typically have only a limited portion of the overall latency budget available. Therefore, a significant amount of the latency must be allocated to the transmission provider responsible for wide-area propagation, as illustrated in Figure 5.1. Since this type of industrial communication is often periodic, a standard method of ensuring low latencies is to reserve specific time slots for each transmission through TSN protocols, effectively making the network and all the communication deterministic.

Currently, two main barriers impede the adoption of virtualization solutions and limit the usage of TSN protocols to bare-metal applications. Firstly, current virtualization techniques do not provide virtual network devices with a paravirtualized hardware clock, which results in the use of a software mode for PTP-based synchronization, leading to a lack of precise time synchronization. Secondly, Virtual Machines require virtual network devices that are efficient in their operations and support multiple queues, as the packet I/O and processing operations between the host and guest machines can cause significant overhead and negatively impact the effort to achieve bounded end-to-end latency. The following sections describe how we address these critical issues for mission-critical applications.

5.1.1 Paravirtualized PTP Clock and TSN-capable NIC

Our first contribution is designing a virtualization approach for TSN networks, as illustrated in Figure 5.2. To support ULL communications, the network frame scheduler standardized in the IEEE 802.1Qbv necessitates the synchronization of clocks among all participants in the TSN network (as discussed in Section 2.2.1). To achieve this in virtual machines, we first utilize a PTP service to synchronize all physical hosts. Then, we provide the guest operating system with a paravirtualized clock that is transparently synchronized with the host's real-time clock by the hypervisor. This approach effectively creates a virtualized PTP clock (referred to as vPTP) that each VM can use as a reference to synchronize its system clock through an Network Time Protocol (NTP) daemon. However, due to the high number of indirection levels involved, the real-time clocks (i.e., `CLOCK_REALTIME`) of the host and guest may be affected by NTP adjustments and diverge over time. To address this, Linux-4.11 introduced specific PTP devices for Kernel-based Virtual Machine (KVM) and Hyper-V. These devices are not connected to the PTP protocol and do not work with network devices, but they are exposed as PTP devices (`/dev/ptp*`), making them usable by existing time synchronization software. Specifically, the type of KVM clock we use is called `kvm-clock`[1] and implements the `pvclock` protocol[2]. This type of clock, as mentioned, allows guest machines to read and consequently track the host wall clock time, automatically synchronizing their system clocks. In addition, this approach also allows us to have a PTP service, e.g., `linuxptp`, active only in the host machine.

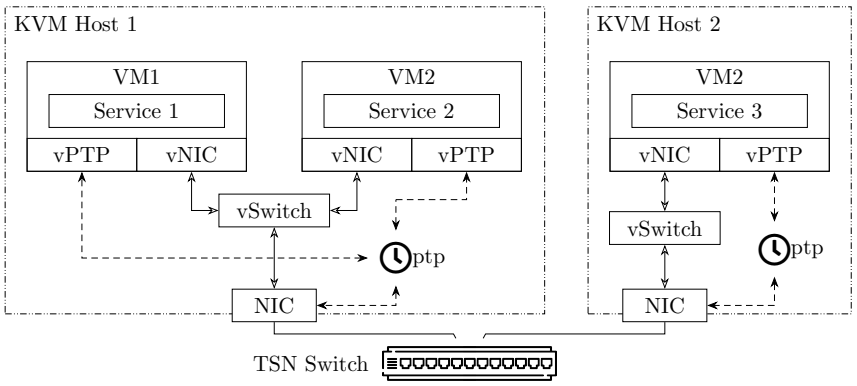


Figure 5.2: Virtualization architecture for TSN-based networks.

[1] https://www.linux-kvm.org/page/KVMClock
[2] https://wiki.osdev.org/Timekeeping_in_virtual_machines

Once the synchronization mechanism is in place, effectively implementing the time-aware traffic windows defined by the IEEE 802.1Qbv standard requires network interfaces to support multiple broadcast queues. Each queue can be associated with a different class of traffic. Within VMs, paravirtualized devices emulate multi-queue transmission support across multiple *virtqueues*. However, an important constraint is that a virtual device with multiple queues must be associated with a VM with two or more virtual CPUs. This is because each queue must have its associated thread, even if the VM is provided with a physical TSN-enabled network card through a passthrough mechanism. In this case, it is possible to synchronize the VM directly using the physical network card and, if available, use offload mechanisms for traffic scheduling to improve performance.

5.1.2 Network Virtualization

The efficiency of how the host processes packets to and from Virtual Machines is a vital aspect of virtual networking. Inefficiencies in this area can impede meeting tight latency deadlines and make guest-level time-sensitive scheduling ineffective. Our framework examines two network paravirtualization techniques (Figure 5.3, which distinguish how packets are processed (datapath operations). One method is to process packets within the host's kernel, while the other is to process them directly within the user space. The kernel-based approach is currently the most mature as vendors widely support it and enable integration with other tools for monitoring and control (e.g., Conntrack, BPF). However, this option may cause poor network performance due to multiple data copies and context switches. On the other hand, kernel-bypassing techniques provide better performance by interacting directly with the device driver but have higher CPU utilization (one or more host cores dedicated to poll incoming messages) and less integration with kernel-based tools. In the context of our work, both options support the TSN protocol within a VM, and developers can choose the most appropriate based on their performance and resource usage constraints.

Overall, our TSN virtualization proposal enables existing TSN-based applications to operate seamlessly within virtual machines without modifying the source code. Depending on the network virtualization approach chosen, developers can optimize TSN traffic in the VMs for either the best possible network performance or the lowest resource usage, depending on their specific requirements.

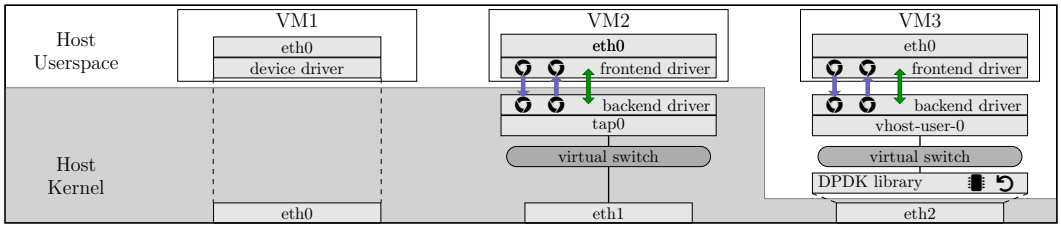


Figure 5.3: Network virtualization approaches. On the left, direct device assignment. On the right, paravirtualization with kernel-level (center) or user-level (right) network virtualization.

5.1.3 Experimental evaluation

In this section, we evaluate the performance of our virtualization framework. To do this, we created a simple TSN application consisting of one *publisher* and one *subscriber*, both running on virtual machines located on two separate hosts. We conducted a latency test where the publisher sent UDP packets with a $1\ \mathrm{ms}$ publishing cycle. This type of traffic, common in real TSN applications, put the entire network pipeline under pressure and highlights any sources of latency overhead. Specifically, the test measures two key indicators for TSN communications: *end-to-end latency* and *jitter*. The end-to-end latency is the time between when a message is scheduled for transmission by the publisher and when the subscriber receives it. Jitter measures how much the actual arrival time of each message differs from its expected arrival time. More precisely, if t_i is the arrival time of the i-th message, its jitter is defined as $Jitter(i) = t_i - (t_{i-1} + T)$, where T is the transmission period, i.e., $T = 1ms$. Finally, the clocks of the two virtual machines were synchronized using the mechanism outlined in Section 5.1.1.

Experimental Settings

To demonstrate that this communication meets ULL requirements, we set a threshold of $1\ \mathrm{ms}$ as the maximum acceptable end-to-end latency for each message. As said before, we must consider that in real-world production environments, about $60\ \%$ of this time will be used by the wide-area network provider to propagate data over 5G networks, which is out of the control of end-system developers (Figure 5.2). We adopted an end-system developer perspective to address this constraint and designed our testbed as two physical hosts directly interconnected by an Ethernet cable, assuming negligible propagation time between them. Consequently, our ULL deadline becomes $0.4\ \mathrm{ms}$ of end-to-end latency between the two

TSN application components. In particular, our test environment consists of two UP Xtreme boards, each equipped with 4 TSN NICs (Intel I210), an Intel i3-8145UE 2/4 CPU, and 8GB of RAM. The two hosts run Ubuntu 20.04 with Linux kernel 5.4.0 and are connected directly through an Ethernet cable. The TSN application components run in VMs that use the same OS as the host and are managed by QEMU/KVM (v4.2.1). Network virtualization is implemented through the virtio framework [61] and OVS [58] (v2.13.3) in two variations: using the kernel-based datapath and the kernel-bypassing DPDK library [6] (v19.11.7).

Under those assumptions, we conducted latency and jitter tests using typical message payload sizes in TSN applications (16, 64, and 256 bytes). We repeated the tests using a virtual switch with a kernel-based datapath and a kernel-bypassing approach, and each test lasted for 100 s. We also tested the communication properties on bare-metal hosts to evaluate the overall effect of the virtualization framework.

Performance Results

The results of the latency tests are shown in Figure 5.5 and Figure 5.4. Let us first consider the behavior of the virtualized applications. We note that the option with the kernel-based datapath struggles to meet our target deadline: in particular, Figure 5.4 shows that the average message latency, computed every 10 s on all the messages exchanged since the previous measurement, is just below the threshold. We observe the same if we consider the median values reported in Figure 5.5 for all the considered payload sides, which means that about half of the measures exceed the ULL constraints. Furthermore, despite using the TSN protocol to reduce latency variability, the jitter remains relatively high (Figure 5.6).

On the other hand, when considering the kernel-bypassing approach, the opposite behavior is observed. The average latency remains just above 100 µs during all the experiments, and the overall median value is around 120 µs for all message sizes. This median value is about 3.25 times lower than the kernel-based alternative and represents just 30 % of the total available latency budget. Additionally, the jitter is minimal for all cases, so this option can effectively preserve the determinism provided by TSN. While it would be interesting to investigate whether this approach can maintain this behavior even on a saturated network, this aspect is not discussed in this work. However, our experiments show that latency does not change significantly when the network between the two VMs

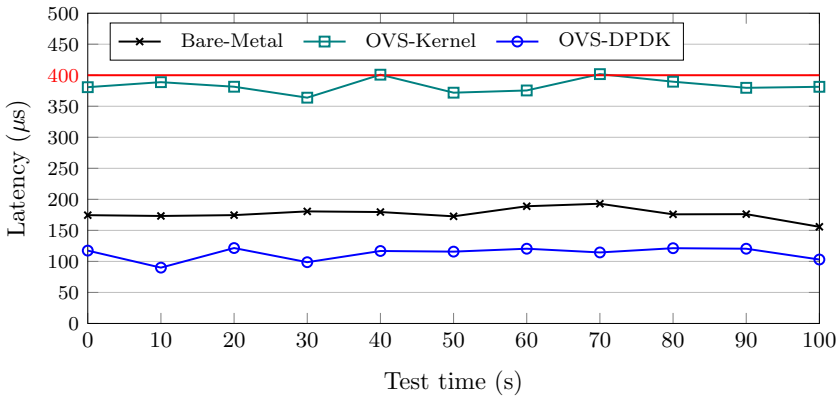


Figure 5.4: End-to-end latency averaged every 10 s for 64 bytes payload size. The red line is the latency threshold.

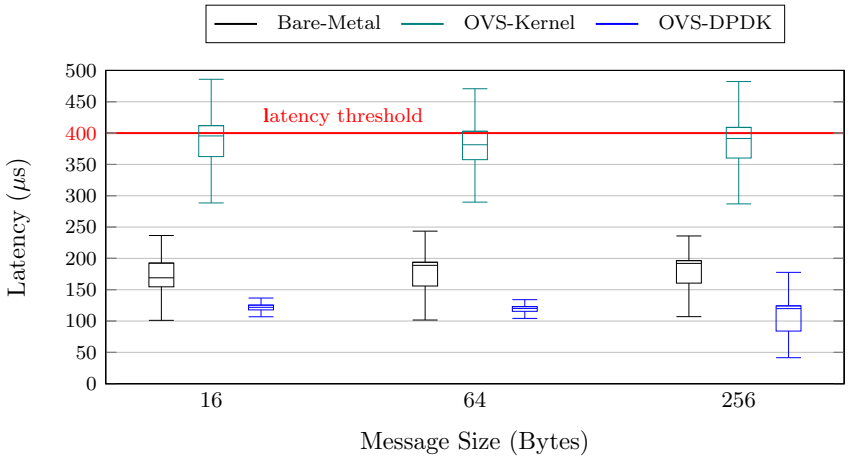


Figure 5.5: End-to-end latency for different payload sizes and network virtualization techniques.

is saturated. Therefore, we conclude that the kernel-bypassing network virtualization approach can effectively allow virtualized TSN applications to respect the ULL constraints and preserve a reduced latency variability.

The primary distinction between the two approaches under consideration is their method of handling packets between the external network and the virtio backend driver, as depicted in Figure 5.3. The traditional kernel-based approach necessitates the forwarding of packets by the virtual switch dataplane to traverse the Linux kernel networking stack, a process that is known to be relatively slow due to scheduling, interrupts, data copies, and context switches. As such, it is not unexpected that network performance in terms of

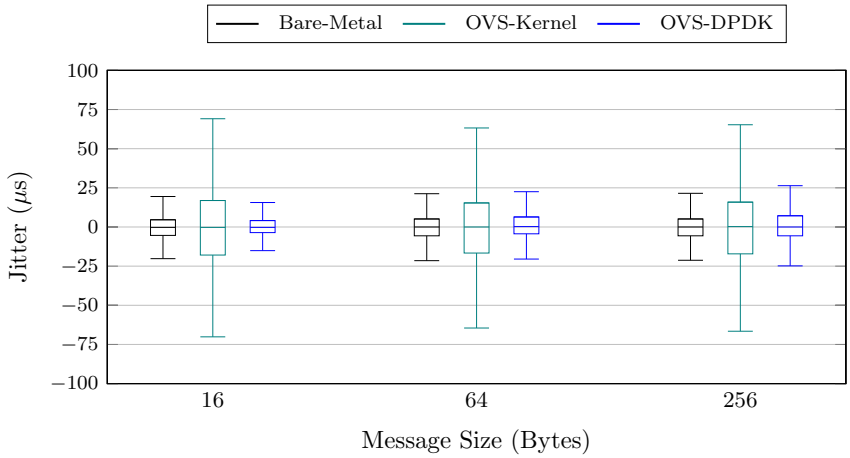


Figure 5.6: End-to-end jitter for different payload sizes and network virtualization techniques.

latency and jitter is significantly improved when bypassing this stack entirely. While this speed improvement is achieved at the cost of a significant proportion of CPU cores being dedicated to packet processing, usually around 100 % of the cores used, and an increase in complexity in network setup and management, the kernel-bypassing approach on the host can fully satisfy our ULL constraints.

Finally, the performance of our virtualized TSN application was compared against that of the same application running on bare-metal hosts. For the 64 bytes case, the average latency of the bare-metal application was consistently around 175 µs, with a median of 190 µs, and a slight jitter. These values are approximately two times lower than the kernel-based virtualization approach. This difference can be attributed to the fact that in the former, each UDP packet only traverses the host kernel, whereas, in the latter, the packets are also managed by the guest kernel.

Interestingly, the kernel-bypassing virtualization approach performed even better than the bare-metal alternative. This behavior can be explained by the fact that the kernel-bypassing technique is more efficient than the operations in the host kernel, as it avoids data copies. Additionally, the network operations in the host kernel require a context switch to a kernel thread, whereas the guest kernel executes in the same process as the VM. Thus, on our testbed, once a single UDP packet with a payload of 64 bytes is received by the host network device, it takes 20 µs to be delivered to the application on the guest, whereas the same operation on the same packet takes 70 µs through the host kernel. These factors, combined with a traffic pattern that magnifies any network overhead, explain

this performance effect. Our virtualized TSN application was even faster than the bare-metal equivalent (36 % lower latency, considering the median value for 64 B packets) and provided almost the same jitter. To obtain a more appropriate comparison, it would be necessary to create a kernel-bypassing TSN host application and compare it against our current best option. However, the TSN scheduler is currently only implemented in the kernel. In the subsequent section, we partially address this issue by demonstrating a prototype of a user-space TSN scheduler operating in containerized environments.

In summary, the performance results presented in this study indicate that latency-sensitive TSN applications can adhere to the ULL constraints even when executed in virtual machines. Specifically, it was shown that kernel-based network virtualization solutions result in a high degree of latency variability and cannot meet the target deadline. On the other hand, kernel-bypassing techniques yield exceptional results, consuming only 30 % of the available latency budget.

5.2 Overlay Network for Time-Sensitive Containerized Environments

In the preceding section, we presented a framework designed to address the overhead associated with virtualization and capable of meeting stringent performance requirements. However, virtual machines are known to have scalability issues. As such, the next step is to shift towards a more lightweight virtualization technique, such as containerization. Containers offered reduced overhead and increased scalability compared to hypervisor-based virtual machines, making them a more suitable option for running services and applications on edge devices with limited resources. Additionally, containerization allows for a unified service provisioning platform that can adhere to applications' Quality of Service specifications.

Kubernetes, in its whole or reduced forms, is the established industry standard for resource management and orchestration. It provides automatic deployment, monitoring, and migration of containerized application components across a shared infrastructure while enforcing applications' QoS specifications. However, it should be noted that containerization alone is not sufficient to support the highly distributed nature of Edge Cloud applications. Network and system-level considerations are also of paramount importance.

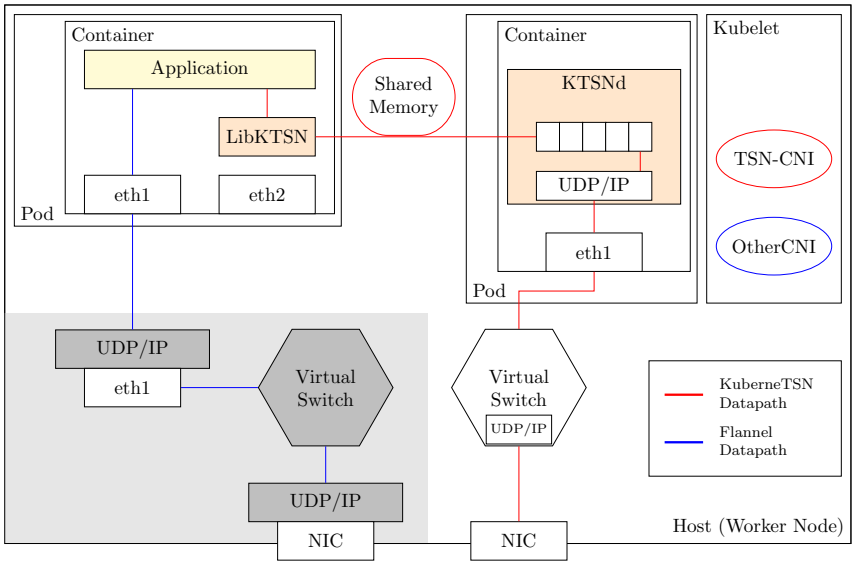


Figure 5.7: The architecture for an accelerated and deterministic overlay network, implemented as a Kubernetes CNI plugin.

In this section, we introduce a cost-efficient solution that enables accelerated and deterministic communication among containerized applications by proposing a novel architecture for a container overlay network that integrates two techniques for ULL communications.

5.2.1 KuberneTSN

The architecture of KuberneTSN is devised to establish a unique container overlay network that is both accelerated and deterministic, accommodating the time-sensitive demands of containerized business or control logic. To attain this, modifications are made to the outgoing container traffic's packet processing pipeline by incorporating two architectural elements: a user library referred to as *LibKTSN* and a daemon referred to as *KTSNd*. Figure 5.7 depicts the role of these components in forming a new data path for time-sensitive traffic.

LibKTSN offers a standard POSIX socket interface to application binaries, allowing for the interception of send operations on a datagram socket and forwarding the packets to a memory region shared with the KTSNd daemon. Our emphasis is on providing time-sensitive traffic services. Hence, we capture only outgoing transmissions with a precise transmission time, i.e., TSN traffic, using the `SO_TXTIME` socket option, where other packets

are redirected onto the standard data path. This strategy enables TSN networking without the need to modify container images, unlike current alternatives (refer to Section 2.4). LibKTSN is the only component of our solution that needs to be present within the application container. It is supplied as a shared library that can be loaded using the flag `LD_PRELOAD` with no modifications to the application code.

The *KTSNd* daemon is a crucial component of our proposed solution, as it functions as both a packet scheduler and a network accelerator. Upon detecting a new packet from an application, KTSNd schedules its transmission based on the application-provided transmission time. While the design of the daemon is agnostic to the specific scheduling strategy, by default, it utilizes a Time-Aware Shaper (TAS) compliant with the IEEE 802.1Qbv standard (as outlined in Section 2.2.2). This packet scheduling option is currently unavailable for containerized applications, as popular virtual switches (e.g., Linux bridge, Open vSwitch, etc.) do not support it. Our solution allows for deterministic packet scheduling for unmodified application binaries running in containers.

When it is time to transmit a scheduled packet, the scheduler must send it on the network on behalf of the original application, preserving the source MAC, IP addresses, and UDP ports while minimizing packet processing delays to meet the user-required transmission time as precisely as possible. We adopt a kernel-bypassing approach to satisfy these requirements and move the entire transmission pipeline to userspace. This allows us to avoid the expensive double-crossing of the kernel networking stack and unnecessary user/kernel thread context switches (as discussed in Section 2.4). Instead, we provide our own simple and efficient UDP/IP stack implementation directly within KTSNd and use the DPDK library to forward packets on the virtual L2 link. This choice enables us to preserve the original packet metadata, as we can manipulate protocol headers directly and significantly reduce the processing overhead.

As shown in Figure 5.7, packets are then handled by a userspace virtual switch that, in turn, should provide its UDP/IP userspace stack to forward them on the physical network. In our implementation, we adopt a widely-used and state-of-the-art userspace virtual switch, Open vSwitch [58], which also uses DPDK for kernel-bypassing.

The simple yet powerful design makes KuberneTSN easy to integrate into standard platforms such as the Kubernetes orchestrator in its various distributions, making it suitable for use in critical networked applications with stringent requirements. To facilitate this, we have developed a Kubernetes network plugin, *tsn-cni*, that implements our proposed

architecture. Specifically, *tsn-cni* implements the Multus CNI interface [46] to create a Layer 3 network fabric that incorporates our accelerated and deterministic data path. The plugin requires applications to include LibKTSN in their execution environment, encapsulating the KTSNd daemon in a separate container. This approach is beneficial for supporting time-sensitive edge applications: by allowing for the use of multiple network plugins at the same time, developers can choose standard ones (e.g., Flannel, Calico) for best-effort traffic, and *tsn-cni* for time-sensitive networking, as shown in Fig. 5.7. As a result, KuberneTSN and its *tsn-cni* implementation enhance Edge Cloud's capabilities by supporting deterministic networking and integrating this option into a familiar ecosystem for application designers. By identifying application components as time-sensitive, developers can instruct Kubernetes to automatically deploy KTSNd alongside the application containers, thus transparently obtaining support for performance-sensitive workloads.

5.2.2 Experimental evaluation

In this section, we evaluate the performance of the *tsn-cni* plugin, which implements the KuberneTSN architecture. This experimental assessment aims twofold: to demonstrate that the proposed *accelerated* datapath is indeed faster than current state-of-the-art networking options and to show that our solution can provide *deterministic* guarantees. Specifically, we compare *tsn-cni* against two alternatives. The first is a bare-metal setting that reproduces the way typical TSN applications are deployed to assess the overhead introduced by the virtualization layer. The second is *Flannel*, a popular CNI plugin for Kubernetes. Flannel uses a Linux bridge combined with VXLAN encapsulation in its recommended configuration to implement the virtual switch, thus building an overlay network corresponding to the *regular data path* of Figure 5.7. By comparing *tsn-cni* and Flannel, we assess whether KuberneTSN meets its design goal of providing additional performance benefits and deterministic properties to inter-container networking.

For this evaluation, we constructed a simple TSN application consisting of two processes, a *talker* and a *listener*, each running inside a container on two remote hosts. We then set up a latency test in which the talker sends UDP packets with a cycle of 1 ms. The test measures two critical indicators of time-sensitive communications: end-to-end latency and jitter, defined as in Section 5.1.3. It's worth noting that the bare-metal and the *tsn-cni* test suites are implemented as actual TSN applications, which associate a desired transmis-

sion time to each packet. However, this option is not available for the test using Flannel as a communication choice, as TSN scheduling would not be enforced (as discussed in Section 2.4). Instead, the only alternative is to send one message and then sleep, repeating this behavior every time T.

Experimental Settings

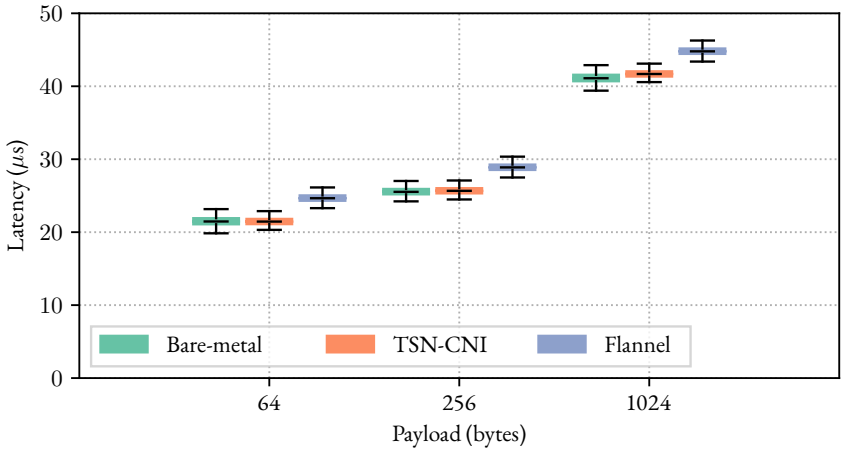
The evaluation and analysis of the performance of the proposed architecture are conducted on a real testbed that simulates an Edge deployment scenario. The testbed consists of two Dell Workstations equipped with an Intel I225 NIC, an Intel i9-10980XE 18/36 CPU, and 64 GB of RAM. The two hosts are interconnected via a physical TSN-compliant switch. The hosts run Ubuntu 22.04 with Linux kernel 5.16. When using Open vSwitch [58], we utilize its two variants: kernel-bypassing on the sender side and kernel-based on the receiver side. In KTSNd, we use the same version of DPDK as OVS, i.e., v21.11. As TSN requires, the two hosts' clocks are synchronized using two PTP daemons. Additionally, we pin the processes to dedicated cores to avoid bias in the measurements caused by the CPU scheduling policy.

End-to-end Latency

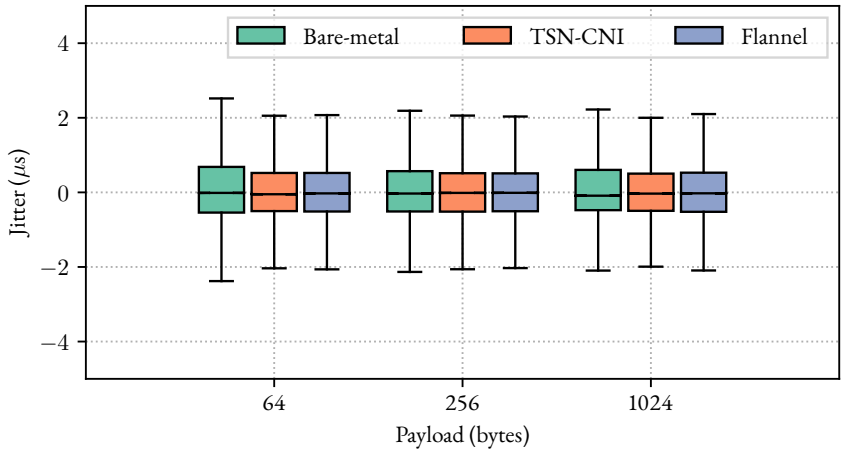
As illustrated in Figure 5.8a, the end-to-end latency and jitter were measured for three typical data sizes (64 B, 256 B, 1024 B) for each of the considered deployment scenarios: bare-metal and containerized applications with *tsn-cni* or Flannel as network plugin. A first observation is that the performance of *tsn-cni* is consistently excellent, with median latency values ranging from 21.5 µs in the case of small packets (64 B) to 41.7 µs for 1024 B. These values are almost identical to those registered for the bare metal deployment, with a small variation in the ns scale starting to appear for the 1KB packet size. Latency variability is negligible in both cases.

If we consider Flannel, we note a slight but evident latency increase (12 % on average). This results from the expensive in-kernel packet processing, which we avoid thanks to the kernel bypassing technique embodied in our solution. The same trend observed for latency is confirmed by the analysis of the jitter metric reported in Fig. 5.8b: the median value is zero in almost all cases, and the variability is negligible. Therefore, we can conclude that

KuberneTSN and its *tsn-cni* implementation minimize the packet processing overhead for containerized applications, achieving the goal of an *accelerated* data path.



(a) Latency.



(b) Jitter.

Figure 5.8: Performance comparison among three deployment options for the latency test application: bare metal, containerized with *tsn-cni*, containerized with *Flannel*. The experiment is repeated for increasing payload sizes: 64 B, 256 B, 1024 B.

Our experiments show that both *tsn-cni* and Flannel exhibit low latency numbers. However, our kernel-bypassing solution using *tsn-cni* demonstrates lower median latency values. In principle, one could expect even better performance from *tsn-cni*, as raw DPDK is known for its high performance [11]. However, the implementation of OVS-DPDK introduces a non-negligible overhead on our userspace datapath, accounting for at least

23 % of the total reported latency. Despite this, we decided to retain it in our system as it is a widely-used tool supported by an active community and offers a wide range of additional features for virtual networking, such as OpenFlow programmability, compared to the Linux Bridge used by Flannel.

Determinism Support

To evaluate the ability of KuberneTSN to provide deterministic guarantees to time-sensitive flows, we analyzed the latency test results. We plotted the respective Cumulative Distribution Function (CDF) in Figure 5.9. Ideally, the curve should be as vertical as possible, indicating a highly predictable packet reception time. As shown in the figure, the bare-metal application and the containerized application using *tsn-cni* exhibit similar performance and are very close to the ideal behavior. Specifically, for *tsn-cni* the 90 % and the 99 % probability correspond to 26.4 µs and 28.1 µs respectively. On the other hand, for Flannel, these thresholds correspond to 29.6 µs and 30.7 µs respectively, indicating a less precise arrival time interval.

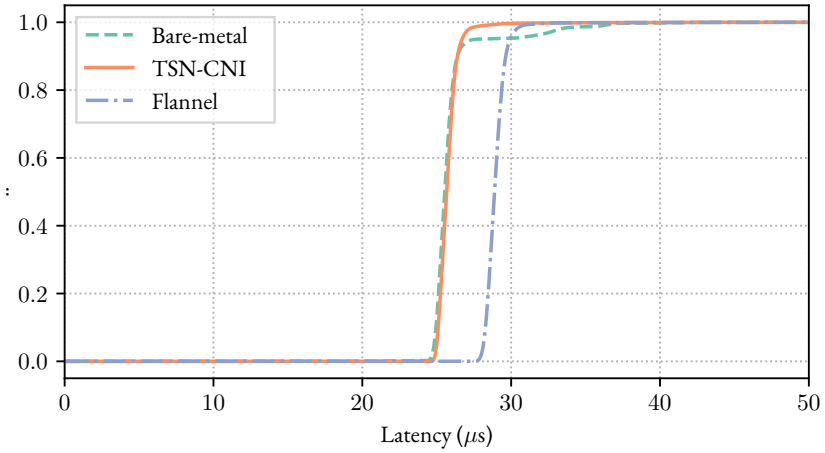


Figure 5.9: CDF with packets of 256 bytes.

This difference highlights the advantage of using KuberneTSN for time-sensitive traffic. The main reason for this behavior is how the test application sends messages. It is impossible to explicitly set a transmission time when using Flannel, as this feature is not supported in containerized environments. Therefore, we are limited to using a classic send-and-sleep loop, mimicking a periodic send operation. This difference has minimal impact

on our experiment as we do not have other competing flows; however, in Section 6.1, we will demonstrate that time-sensitive flows require dedicated support. *tsn-cni* serves this purpose by providing dedicated support to containerized applications, enabling them to meet the requirements of heterogeneous flows in mixed-criticality scenarios.

In conclusion, our implementation and evaluation of KuberneTSN as a network plugin for Kubernetes, *tsn-cni*, has demonstrated its effectiveness in providing accelerated and deterministic performance for containerized networks. The kernel-bypassing approach and novel userspace TSN packet scheduler used in KuberneTSN allows for minimal packet processing delays, making it suitable for time-sensitive Edge applications. Our experiments have confirmed that containerized applications using *tsn-cni* exhibit the same performance and determinism as bare-metal applications, outperforming the widely used Flannel network plugin.

5.3 SELENE: SElective acceLEration at the Network Edge

In this section, we introduce SELENE. A new communication middleware designed and optimized for Edge Cloud applications with intelligent logic, strict performance requirements, and various resource constraints. SELENE is the first middleware that enables developers to select the most appropriate transport technology through high-level policies, thus separating application code from the specific transport technology. This capability of SELENE makes modern network acceleration techniques transparent to the final user, resulting in increased code portability, improved network performance, and preserving domain independence. SELENE has two main components: a *runtime*, which is the core of the middleware and must be running on each participating host, and a *client library* that provides the application's API, allowing them to interact with the runtime.

In the following sections, we will provide more detailed information about the SELENE API and how it supports the diversity of edge applications (Section 5.3.1 and Section 5.3.2). Additionally, we will give an overview of the runtime architecture to understand how the SELENE primitives are implemented using underlying technologies (Section 5.3.3).

5.3.1 SELENE API

The SELENE client library provides a minimal interface that serves three main purposes:

1. It is designed to be easy for developers, in contrast to the current interfaces of network acceleration techniques which require knowledge of complex and low-level details.

2. The interface is expressive enough to allow for efficient implementation of diverse domain-specific abstractions on top of SELENE.

3. The interface is agnostic to underlying transport protocols and only exposes high-level policies to inform the middleware about the quality requirements of different data flows.

The SELENE API defines a few basic concepts to keep the interface as simple as possible. A communication *channel* represents a unidirectional data flow between endpoints, which can interact locally or through the network. A channel exists within a *stream*, an abstract concept that associates quality requirements to one or more channels. In the context of a stream, a communication channel is established between endpoints called *sources*, which produce data, and *sinks*, which consume data. Each channel is uniquely identified by an application-provided *channel id*, which developers must choose based on their higher-level business logic. For example, a SELENE-based Message-oriented Middleware would typically assign channel ids according to topic names. Figure 5.10 illustrates an example of a SELENE channel, where sources and sinks opened within the same stream and with the same channel id will communicate on the same channel.

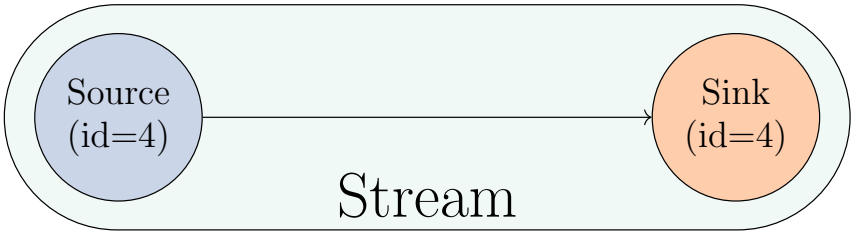


Figure 5.10: A SELENE channel is created between sources and sinks with the same channel id within the same stream.

The concept of the stream is crucial in the interface. Only sources and sinks belonging to the same stream can exchange data because the stream defines the quality requirements for

communication. Depending on these requirements, SELENE will transparently map the channel to a technology-specific concept, for example, a kernel-based socket. Direct data forwarding using shared memory is also enabled when sinks and sources are co-located.

```
1  /* Open and close a session */
2  int init_session();
3  int close_session();
4
5  /* Stream */
6  stream_t create_stream(options_t opts);
7  void close_stream(stream_t stream);
8
9  /* Source APIs */
10 source_t create_source(stream_t stream, int channel);
11 void close_source(source_t source);
12 buffer_t get_buffer(source_t src, size_t size, int flags);
13 int emit_data(source_t src, buffer_t buffer);
14 int check_emit_outcome(source_t source, int id);
15
16 /* Sink APIs */
17 sink_t create_sink(stream_t stream, int channel, data_cb cb);
18 void close_sink(sink_t sink);
19 int data_available(sink_t sink, int flags);
20 buffer_t consume_data(sink_t sink, int flags);
21 void release_buffer(sink_t sink, buffer_t buffer);
```

Figure 5.11: The SELENE API.

Figure 5.11 illustrates the complete SELENE APIs. An application must open a communication session with the local runtime to use the API. Then, one or more streams can be opened by specifying a set of quality options, which are covered in detail in Section 5.3.2. Once a stream is open, communication channels can be created by creating sinks and sources and defining the desired channels using the channel id mechanism previously described.

All the available operations on sinks and sources are asynchronous to facilitate zero-copy communication. To send a new message from a source, developers must first request a memory area (*buffer*) from the runtime. Then, the application can write the message into that buffer and *emit* it, signaling to the middleware that the data is ready to be sent. This

operation returns a token that can later be used to retrieve the outcome of the operation. Similarly to Demikernel [81], the API does not offer *after-write protection*: developers must not modify the buffer content once it has been emitted. On the sink side, there are three different ways to receive data, similar to the DDS middleware [32]. Users can register a callback that will be called every time a new message is received for that sink. Alternatively, users can directly call the *consume* operation, which can be configured to either return immediately, regardless of the presence of new data or to block until new data is available. New data is returned as a pointer to a memory area borrowed from the runtime to preserve the zero-copy semantic. Once the user has finished processing the data, the memory should be returned to the middleware by explicitly *releasing* the buffer.

We believe this set of primitives meets our design goals of simplicity, flexibility, and transparency to multiple network acceleration options. At the same time, this API is expressive enough to define very different higher-level interfaces. To demonstrate this, in Section 6.2, we will present our experience implementing and deploying two very different applications, a Message-oriented Middleware and an image streaming framework. Both applications were easy to develop and demonstrated a significant performance advantage from the selective acceleration capabilities guaranteed by SELENE.

5.3.2 SELENE QoS policies

An essential aspect of this work is associating a set of *quality requirements* to each communication channel through the stream concept. These requirements are defined in terms of high-level Quality of Service policies, making the SELENE interface transparent to low-level network details. In line with our goal of maximum simplicity, we have reduced the number of available options to the essential. Currently, SELENE defines three possible *quality options* that can be associated with a stream: the degree of *datapath acceleration*, the level of tolerable *resource consumption*, and the *time-sensitive* constraints of a data stream.

The *datapath acceleration* policy informs the middleware whether a specific data flow requires any network acceleration or if regular kernel-based networking is sufficient. In case acceleration is needed, edge developers must have control over the associated cost. To achieve this, users can use the *resource consumption* policy to specify whether resource usage is a concern when mapping data flows to specific technologies. For example, DPDK requires high CPU consumption, which may be unacceptable in some contexts. Finally, a

third policy allows users to characterize data flows based on their *time sensitiveness*. This policy specifies the packet scheduling strategy for the packets of that flow. By default, a FIFO scheduler handles all the packets and sends them to the network as soon as the user code emits them. However, if the stream is labeled as time-sensitive, a TSN-compliant scheduling strategy [30] can be used to provide deterministic network behavior (see Section 5.3.3).

Upon initiating a new stream, SELENE dynamically maps the stream's quality requirements to the most appropriate network technologies available within the given deployment environment based on a user-configured mapping strategy. Without a custom strategy, SELENE employs the following approach: if no acceleration is necessary, the kernel-based UDP is utilized. Otherwise, Remote Direct Memory Access is employed as it offers superior network performance with minimal resource usage, as network operations are offloaded to the Network Interface Card (NIC) [44]. However, RDMA is currently not widely adopted in Edge contexts and is generally limited to small-scale data centers close to data sources. As an alternative, SELENE maps user code to the Data Plane Development Kit if resource usage is not a concern or to eXpress Data Path if it is. XDP may result in slower performance but does not necessitate a dedicated set of CPU cores for detecting incoming packets. The mapping process is executed by SELENE at runtime upon the creation of a stream, and the user code remains unchanged regardless of the actual deployment execution.

It should be noted that SELENE only considers these policies as suggestions for application performance requirements and makes a best-effort attempt to align quality and existing technologies. If acceleration is required, but no acceleration technology is available, SELENE will revert to the standard kernel-based network stack and provide a warning to the user.

SELENE, by design, does not offer additional policies to control other aspects of communication. As such, there are no means to control network operation semantics. Currently, SELENE only provides uniform access to the underlying network technologies without introducing any enhancing mechanisms such as flow or congestion control. It is the responsibility of the user to design such mechanisms as part of their custom logic, thus allowing for easy implementation of existing solutions on top of SELENE, such as OMG DDS [32] and its reliable solution RTPS [71].

Similarly, SELENE does not impose fault tolerance semantics or protocols, as developers are responsible for implementing the most appropriate solution for their specific use case.

5.3.3 SELENE runtime

The SELENE runtime architecture consists of three main components, illustrated in Figure 5.12. These components include a memory manager, a packet scheduler, and a set of polling threads. The memory manager is crucial as it effectively implements the abstraction that decouples the homogeneous interface presented to the applications from the highly heterogeneous details of each transport technology. As discussed in Section 2.3, all the considered technologies adopt a similar approach to achieve zero-copy data transfers, which is to place data to send or receive in a shared memory area registered with the NIC for Direct Memory Access.

Based on this insight, the SELENE runtime has been designed with a technology-agnostic mechanism for zero-copy communication, implemented using shared memory. This abstraction is then implemented differently for different transport options. At system startup, the memory manager reserves two memory areas, referred to as memory pools, to contain outgoing and incoming application data. These areas are divided into memory slots, uniquely identified within the collection by a slot id. When a new application connects to the runtime, it maps parts of the memory pools to its address space. From then on, the application and the memory manager communicate by exchanging slot ids that refer to relevant parts of the memory pools.

Figure 5.13 illustrates the communication flow between a sink and a source in the SELENE runtime. As a preliminary operation, each application must connect to the runtime by calling the `init_session` function. After connecting, the application requests the manager to send a new packet to a memory slot ①. If a free slot is available, the manager sends the corresponding *slot id* to the client library, which provides the application with a pointer to the associated memory area. This lets the user write the packet content in the shared memory directly. Once the writing is completed, the application *emits* the packet ②, and the SELENE client library communicates the corresponding *slot id* to the runtime. The *packet scheduler* then schedules the packets for sending according to the *time sensitiveness* policy. By default, the scheduler adopts a First-In-First-Out (FIFO) strategy. The scheduler supports the TSN standard for time-sensitive data, implementing

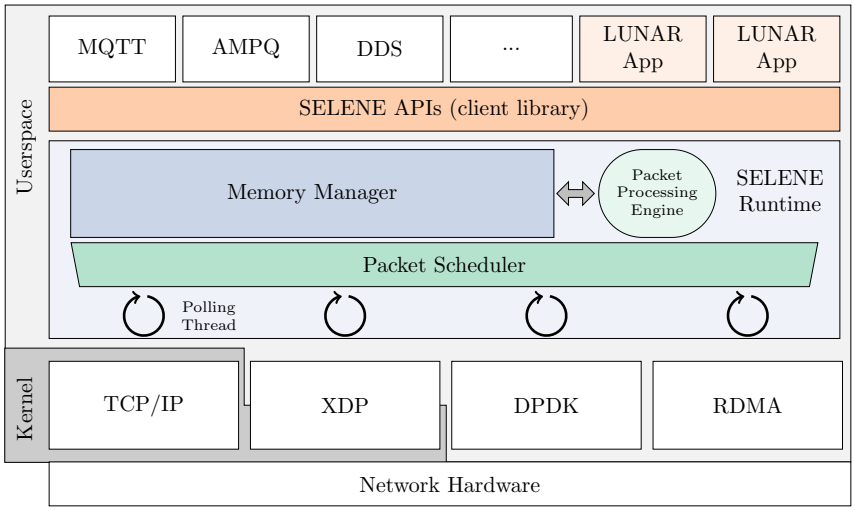


Figure 5.12: The SELENE Architecture.

the IEEE 802.1Qbv time-aware scheduler [52], which is designed explicitly for real-time applications.

On the receiving side, the mechanism works symmetrically. The NIC places the newly arrived packets in a designated memory area. When the manager detects them, it sends the relevant *slot id*s to the client library, which offers the applications a pointer to the same memory where the data has been previously placed ③. Once the application has processed the data, it must return the token to the runtime to make it available for subsequent operations ④.

This design allows SELENE to avoid data copies on the data path by only exchanging a *slot id* between the runtime and the client library. However, as these components reside in separate processes, this interaction may become a performance bottleneck. To mitigate this risk, we handle this communication through state-of-the-art lock-free queues [1, 75].

Implementing this general mechanism for the different available network technologies is the responsibility of the *core polling threads*. These threads, one per available transport technology, continuously execute a send and a receive operation. The send operation sends the scheduled packets on the existing network using the low-level API of each specific technology. Before this, in the case of DPDK and XDP, the *packet processing engine* processes the scheduled packets through our custom userspace implementation of the transport (UDP) and network (IP) protocols. On the reception side, the polling threads use technology-specific APIs to check for newly arrived packets. If necessary, new packets

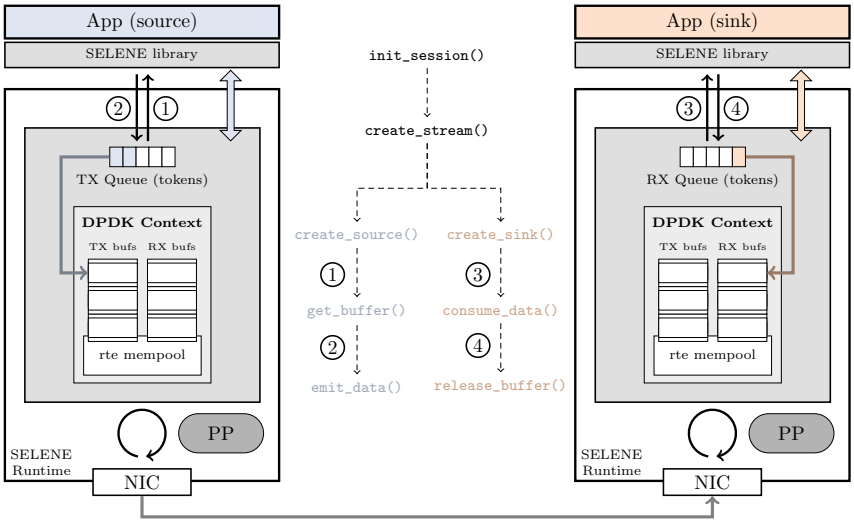


Figure 5.13: SELENE communication flow.

are first processed by the packet processing engine and then dispatched to the relevant applications according to the previously described mechanisms.

To conserve resources, we provide a mechanism that dynamically pauses the thread operations when they are not needed. For instance, if no application requires accelerated endpoints, the corresponding threads are paused. This allows for efficient resource usage and improved performance in the edge context.

5.3.4 Evaluation over Real Testbeds

Our evaluation demonstrates that our abstraction layer introduces minimal overhead compared to each native communication technology. We also compare SELENE to Demikernel [81], the most comprehensive and state-of-the-art alternative for transparently accessing kernel-bypassing technologies, and show that the additional dynamic capabilities provided by SELENE come with comparable or even better performance.

For this evaluation, we developed a C prototype of the SELENE runtime that supports two network technologies: kernel-based UDP and DPDK. Integrating RDMA and XDP is still ongoing work. However, we prioritized the two former options as they are the most commonly adopted in the Edge Cloud ecosystem. Unlike RDMA, they do not require special hardware and are easy to use from virtual environments (VMs, containers). They

also represent the differences between the kernel-based and kernel-bypassing networking approaches.

Experimental setup

We evaluate SELENE in two testbeds to assess its performance under different conditions. The first testbed is a local setup, where two nodes are directly interconnected to minimize network operations' overhead and highlight the impact of SELENE on the measured metrics. The second testbed is a public cloud infrastructure (CloudLab [28]), where we reserved two interconnected nodes by a switch. The hardware and OS specifications of the nodes in both the testbeds are reported in Table 5.1. For DPDK, we used version 21.11 [3].

Testbed	OS	CPU	RAM	NIC	Switch
Local	Ubuntu 22.04	18-core Intel i9-10980XE @ 3.00GHz	64GB	Mellanox DX-6 100Gbps	—
Public cloud	Ubuntu 22.04	32-core AMD 7452 @ 2.35GHz	128GB	Mellanox DX-5 100Gbps	Dell Z9264F-ON

Table 5.1: Setup of the local and public testbed for SELENE evaluation.

To minimize OS-induced latency, we pinned processes to cores (one core per network technology, one to test applications). We also increased the Linux socket buffers to allow receivers to keep up with the highest possible send rate. Unless otherwise specified, we did not adopt further optimizations.

Latency and throughput benchmarks

To demonstrate that SELENE introduces minimal overhead compared to using each native technology directly, we built a benchmarking application for latency and throughput. For latency, we used a simple *ping-pong* application designed to highlight any overhead in the send and receive pipeline. It measures the round-trip time (RTT) of a single message sent from one host and immediately echoed by a remote receiver. We repeated this test for 1 million messages. The throughput benchmark is a *stress test* application that evaluates how much of the available network bandwidth is practically achievable when a sender continuously sends 1 million messages at full speed to a remote receiver. We measured throughput as the amount of payload data (*goodput*) received in the time unit. We ran every throughput experiment 10 times. We implemented the benchmarking application in three

[3] https://core.dpdk.org/download/

versions: one that uses UDP sockets, one that uses native DPDK, and one that uses the SELENE API. We found that even for such a simple benchmarking application, SELENE minimizes the amount of code necessary for networking, as Table 5.2 summarizes, without requiring developers to understand the details of each technology.

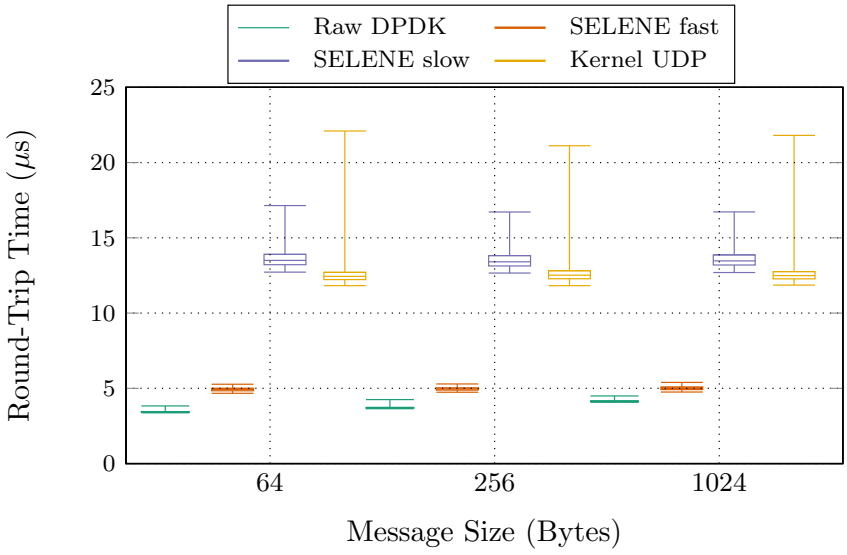
Interface	Lines of Code (LoC)	Increase
SELENE	189	—
UDP socket	227	+20%
DPDK	384	+103%

Table 5.2: LoC to implement the benchmarking application.

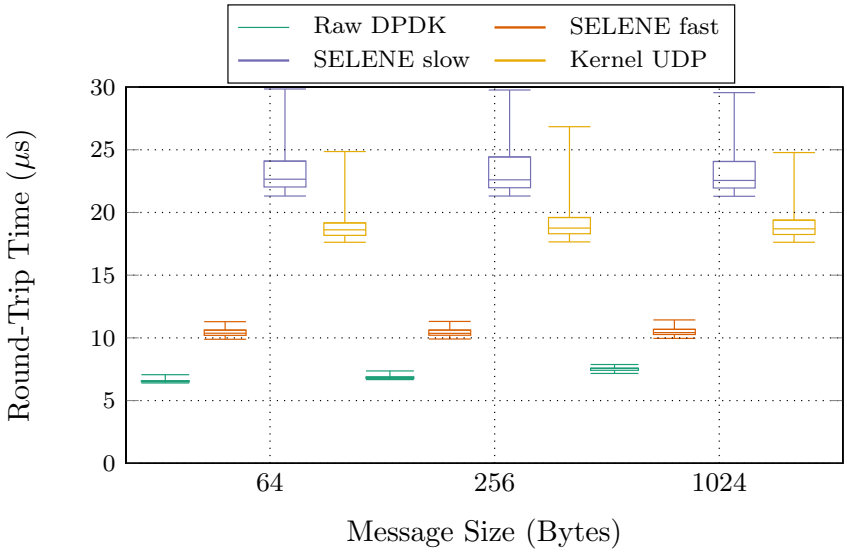
Figure 5.14a and Figure 5.14b report the latency of SELENE for increasing payload sizes when using two different datapath acceleration QoS: *slow*, which maps network operations to UDP sockets, and *fast*, which maps to DPDK. Overall, we noted no significant difference among different payload sizes. In the local testbed, we observed that *SELENE fast* keeps very close to raw DPDK, with an increase of the median RTT values of at most 1 µs. The same gap separates *SELENE slow* from the pure kernel-based UDP benchmark. Hence, we can conclude that SELENE introduces on average a 500 ns overhead on each UDP packet both in fast and slow mode. In the public cloud setup, we noted a general increase in RTT values, as expected, because of the introduction of a switch between the two hosts. According to our measurements, the switch added, on average 1.7 µs, and packets had to traverse it twice. However, SELENE's latency increased more than expected, adding around 1.7 µs to the raw DPDK median values. We investigated this increase by breaking the latency value into its main components in Figure 5.15. In addition to the expected growth of the network latency, we also observed a significantly higher time spent by SELENE in the send and receive operations.

The cause of this behavior is that the processor on the cloud servers is significantly slower than in our local testbed[4]. Although SELENE tries to minimize the processor intervention on the critical path, the requirement to support multiple applications running as separate processes makes it hard to reduce further the number of CPU cycles required for internal operations. A possible direction to reduce this overhead is to parallelize our send-and-

[4] https://www.cpubenchmark.net/high_end_cpus.html



(a) Local testbed.



(b) Public cloud (Cloudlab)

(c) Round-Trip Time (RTT) for increasing payload sizes.

receive routines over multiple cores. In our prototype, we decided not to implement this feature, considering the usually limited resources of edge hosts.

To provide a comprehensive understanding of the performance of SELENE, we present an expansion of our latency experiments in Figure 5.16 by including a more comprehensive range of systems and reporting the average RTT for a 64 B payload size. Specifically, we

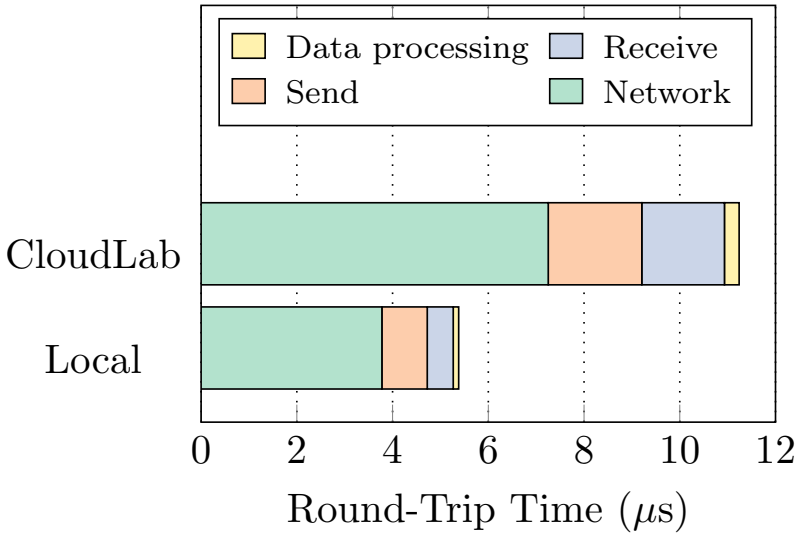
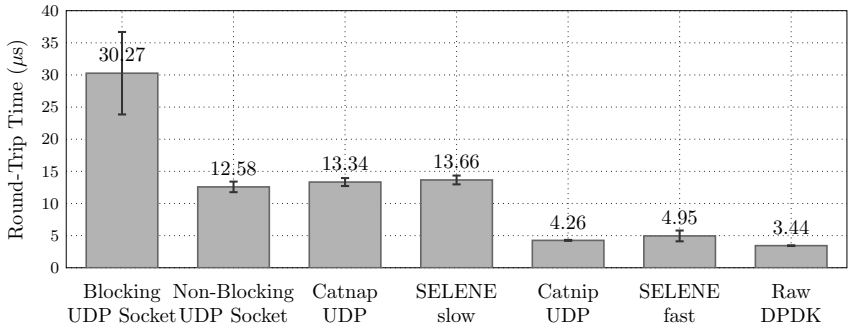


Figure 5.15: *SELENE fast* latency breakdown (64 B)

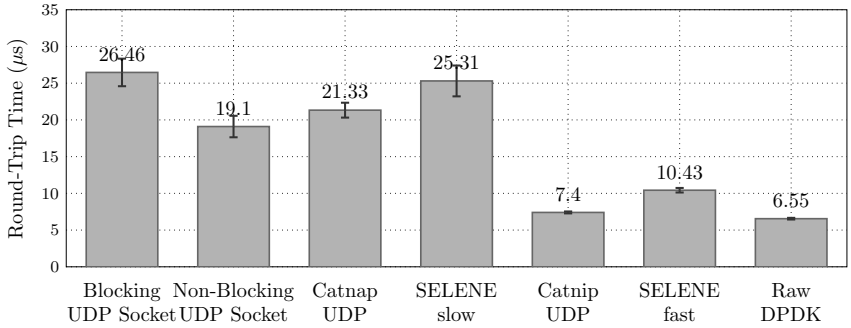
have implemented two versions of the benchmark utilizing pure UDP sockets, one using a blocking receive and one utilizing a continuously polling non-blocking socket. As expected, the former exhibits a slower performance than the latter, as process wake-ups introduce significant latency overhead. Additionally, we have evaluated the performance of Demikernel by binding it to two of its offered libraries, namely *Catnap* and *Catnip*. These libraries correspond to SELENE with *slow* and *fast* datapath QoS, respectively. Our results indicate that the Catnap library performs slightly slower than the native socket application in both testbeds. The slow datapath of SELENE displays a performance comparable to that of Catnap in our local setup, with an average latency of 1.9 µs in the cloud setting.

Furthermore, we observe a similar trend as previously discussed when utilizing the DPDK library. On the local testbed, the fast datapath of SELENE exhibits an additional latency of 690 ns compared to the raw DPDK performance. In the cloud setting, all the latencies increase. However, the Catnip library preserves a similar gap to raw DPDK. The Demikernel exhibits a more straightforward logic for payload delivery, as it is a library compiled with the application. The fast datapath of SELENE may suffer from a slower processor. However, it still demonstrates a competitive latency performance, despite the added dynamicity it offers to multiple concurrent applications.

In addition to latency, network bandwidth is a crucial metric in Edge Cloud applications, as they often require the rapid transfer of large data payloads, such as camera images, for



(a) Local testbed.

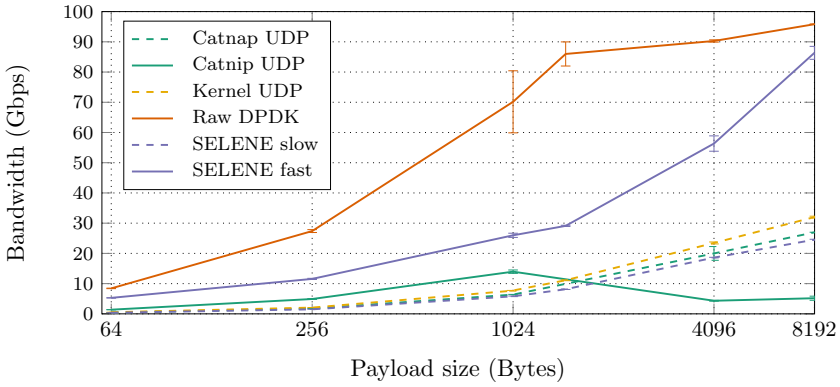


(b) Public cloud (Cloudlab)

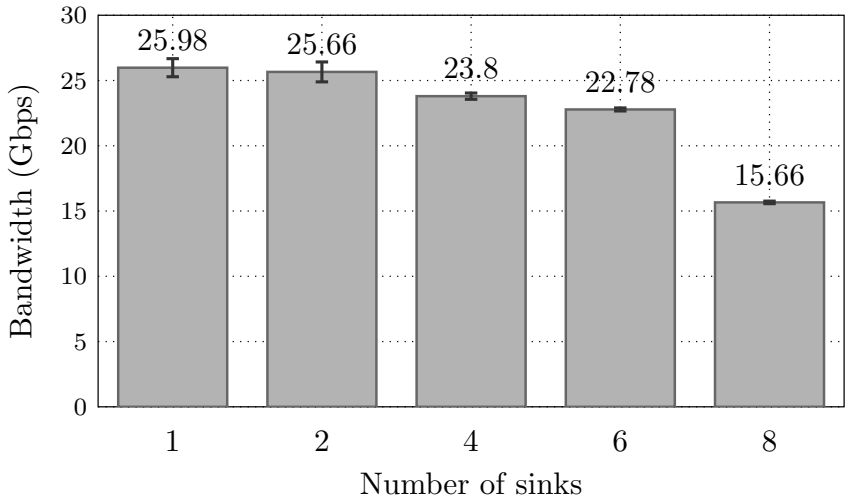
Figure 5.16: Average RTT of raw network technologies, SELENE, and Demikernel for 64B payload size.

remote analysis. Our findings indicate no significant performance difference between the two testbeds regarding bandwidth utilization. We only present data for the local setup in Figure 5.17a. The figure illustrates the throughput of *SELENE fast* and *SELENE slow* compared to the corresponding Demikernel libraries, kernel-based UDP sockets, and raw DPDK for increasing payload sizes. To avoid fragmentation overhead, jumbo frames are enabled for payloads larger than 1.5 kB. Our results show that raw DPDK can quickly saturate the network interface card (NIC) as it does not perform any data processing. Despite the need for inter-process communication, *SELENE fast* demonstrates the second-best performance, reaching peaks of 90 Gbps for the largest payload size, whereas Catnip exhibits a significantly lower throughput. This difference reflects the distinct utilization of the underlying DPDK library: Catnip is optimized for latency [81] and sends one packet at a time on the network, whereas SELENE employs a form of *opportunistic batching* [41, 43] at the sender side. This technique sends messages ready to be sent in a batch, but

the system never waits for a fixed-size batch to fill up. As previously demonstrated, this approach allows for the highest throughput under heavy traffic without significantly harming latency. When this technique is not employed, such as in *SELENE slow*, we observe that Demikernel and SELENE perform similarly.



(a) Throughput for increasing payload size.



(b) SELENE throughput for an increasing number of sinks (1KB)

Figure 5.17: Throughput benchmark for SELENE and the other reference systems.

In our experimentation, we have demonstrated that SELENE can support multiple applications on the same host simultaneously. Figure 5.17b illustrates this by conducting a throughput test with an increasing number of sinks connected to the runtime on the receiver host, all listening on the same *channel id*, but from separate applications. The plot depicts the average throughput received by all sinks for a payload size of 1 kB. It can be

observed that for up to 6 concurrent sinks, the average received throughput only drops by 8 %, while a significant degradation is observed with eight sinks at (−39 %). This is noteworthy as multiple concurrent applications in an Edge context are not typical.

Our experiments demonstrate that SELENE can achieve µs-scale latencies and tens of Gbps bandwidth utilization, which is competitive or even better performance than other kernel-bypassing systems across different environments. Additionally, SELENE offers added dynamicity, portability, and flexibility to developers without significant performance degradation.

6 The application layer

Contents

6.1 TEMPOS: a Time–Effective Middleware for Priority Oriented Serverless

The proposed TEMPOS middleware is a novel solution designed to address the advanced Quality of Service management requirements in Function-as-a-Service (Faas) infrastructures. The main objective of TEMPOS is to provide a unified and simplified approach to managing the heterogeneity and complexity of Edge deployment environments while ensuring the separation of quality of service levels among the workflows being executed. To accomplish this, TEMPOS uses an orchestrator that coordinates and combines various technologies for prioritization and reservation available across the entire stack of virtualization layers involved in FaaS infrastructures.

TEMPOS's ability to abstract away complexity makes it suitable for many applications, such as Smart Tourism, Industry 4.0, and Smart Agriculture. The TEMPOS abstractions

only require the definition of business logic in the form of a workflow and an associated QoS level. TEMPOS continuously monitors the QoS support of targeted resources in the deployment environment components and updates the configuration of the FaaS components accordingly, ensuring that service level agreements are met.

The current implementation of TEMPOS assumes that the necessary middleware and FaaS components are already installed and configured. However, we are exploring the integration of TEMPOS with existing orchestration features for QoS-aware dynamic deployment, which can further enhance the capabilities of the TEMPOS middleware.

TEMPOS's architecture is designed with extensibility and flexibility in mind to accommodate different application scenarios, deployment sites, and technologies. The architecture comprises three functional slices: *Bridging*, *Delivery*, and *Processing*, and a TEMPOS component called the *Controller*, which is responsible for orchestrating these slices. The slices are designed to interact with each other only through a set of predefined interfaces, such as *UDP* ports, which creates a contract among the slices and allows each one to be independent of the specific technologies or protocols used by the other slices.

6.1.1 The TEMPOS Architecture

The next sections introduce the different slices that compose the TEMPOS architecture.

Controller

The *TEMPOS Controller* is the central control point for configuring and managing the TEMPOS slices and customer-defined workflows. In particular, the Controller represents the endpoint facilitating the process for application developers by providing a simple interface for configuring the infrastructure. For this purpose, the Controller is responsible for mapping and matching the desired QoS levels to the different slices and handling the deployment and modification of workflows at runtime.

The process of configuring the TEMPOS infrastructure involves the following steps:

- Receive and remap the configuration for all components of the TEMPOS infrastructure from the developer/deployer, which includes configuring the Channels and Topics with the different QoS levels offered by the middleware.

- Receive the set of workflows and associated QoS levels from the developer/deployer and map them to the underlying infrastructure.

The Controller handles these reconfiguration events by interacting synchronously with the TEMPOS slices. This ensures that the QoS required for the whole application is preserved. The developer/deployer can also request the configuration of workflows at runtime, as the Controller exposes APIs for deploying new workflows or modifying existing ones. The workflows can be defined with per-flow granularity, which means that the QoS levels are specified for each flow and not just for a single invocation.

Delivery slice

The *Delivery* slice of TEMPOS is responsible for realizing QoS-aware event distribution among TEMPOS components. The TEMPOS event distribution process is achieved through interworking different communication technologies and protocols, along with services in the duty of orchestrating and composing them. Then, a series of abstractions are introduced to easily extend the set of supported technologies and provide developers/deployers with a simplified view.

The core component of the Delivery slice is a novel Message-oriented Middleware that can dynamically exploit different mechanisms and technologies to achieve QoS differentiation. This MOM decouples interactions among TEMPOS components and enables advanced features such as load balancing and automatic fault tolerance. It also has a transparency feature that allows for dynamically adding and scaling TEMPOS middleware components. The synergy between the TEMPOS MOM and Controller completely hides the internal complexity of our middleware from the application developers' perspective, thus achieving an essential feature of Serverless computational models.

Applications and middleware components can connect to the MOM to send or receive messages by creating a Channel. The *TEMPOS Channel* is an abstraction that defines a connection between any TEMPOS components. Since the Delivery slice potentially covers several communication environments, a Channel is characterized by a specific communication protocol and, if supported, a prioritization or reservation technique. To support a wide range of environments, the MOM uses a mechanism based on the concept of an Adaptor. Adapters allow for supporting many Channels and interacting with them seamlessly and simultaneously.

Messages received by a specific Channel are processed in priority order through *priority queues*. The TEMPOS MOM processes events in parallel, prioritizing those associated with higher QoS. This allows for high flexibility and adaptability in the TEMPOS system, making it well-suited for use in highly heterogeneous contexts.

To provide our middleware with a consistent end-to-end quality abstraction, we introduced the concept of *QoS-aware Topic*, a logical grouping of channels that share common QoS requirements and characteristics, formally defined as:

$$T = \left(\begin{bmatrix} C_{in1} \\ C_{in2} \\ \vdots \\ C_{inN} \end{bmatrix}, Q, \begin{bmatrix} C_{eg1} \\ C_{eg2} \\ \vdots \\ C_{egN} \end{bmatrix} \right)$$

where

$$T = \text{Topic},\ Q = \text{Priority Queue},$$
$$C_{in} = \text{Channel Ingress},\ C_{eg} = \text{Channel Egress}$$

The TEMPOS system utilizes the concept of a QoS-aware Topic to coordinate and abstract the different QoS levels available through the channels and associated priority queues of the MOM. Each Topic is associated with a specific QoS derived from the associated channels' performance and the processing slice's processing performance. This abstraction allows application developers/deployers to express their QoS requirements at a higher level of granularity rather than having to configure individual channels. The QoS-aware Topic is a key concept in the TEMPOS system, as it allows for a consistent end-to-end quality abstraction across the entire middleware, providing developers with a single and transparent view, even when leveraging different QoS-sensitive technologies such as TSN, 5G slicing, or Wi-Fi 6 prioritization.

To allow for differentiated message distribution policies among the subscribers of a TEMPOS Topic, we also inherited, from classical MOM solutions, the concept of *Subscription Groups*. In the classical pub-sub model, each subscriber of a topic receives all messages sent through it. Through the mechanism of Subscription Groups, each message

is sent to only one of the subscribers in each group, thus realizing a load-balancing feature among the subscribers. Indeed, load balancing is an essential capability of a MOM in the context of FaaS platforms, as it enables the distribution of workflow requests across multiple executor nodes.

To manage the distribution of messages among subscribers of a TEMPOS Topic, TEMPOS utilizes the concept of Subscription Groups. This concept, inherited from traditional MOM solutions, allows for differentiated message distribution policies. In a classical Pub/Sub model, all topic subscribers receive all messages sent through it. However, with Subscription Groups, each message is only sent to one subscriber in each group, providing a load-balancing feature. This capability is crucial in the context of FaaS platforms as it enables the distribution of workflow requests across multiple executor nodes, ensuring efficient and effective processing.

Bridging slice

The *Bridge* slice is a fundamental abstraction in the TEMPOS platform, responsible for providing a consistent and unified interface for external entities to access and utilize the services distributed through the platform. The Bridge slice employs components and mechanisms that transform external events into an internal representation that can be managed by other slices and transparently processed by functions. The central component of the Bridge slice is the *Trigger*, which serves as the main entry point for incoming requests from external sources.

The main responsibility of the Trigger is to forward every piece of information sensed or received to the MOM after adapting and encapsulating it in the form of events. The Trigger acts as a bridge between the external world and TEMPOS, adapting external protocols, representations, and QoS levels to internal ones. This allows for seamless integration of external entities with the TEMPOS platform, regardless of the specific communication technology or protocol being used.

In addition, the Trigger is the first TEMPOS component to differentiate and characterize event quality by exposing a different endpoint for each supported QoS level. This allows for fine-grained control over the quality of service provided to external entities, ensuring that the platform can meet the diverse and dynamic requirements of different use cases.

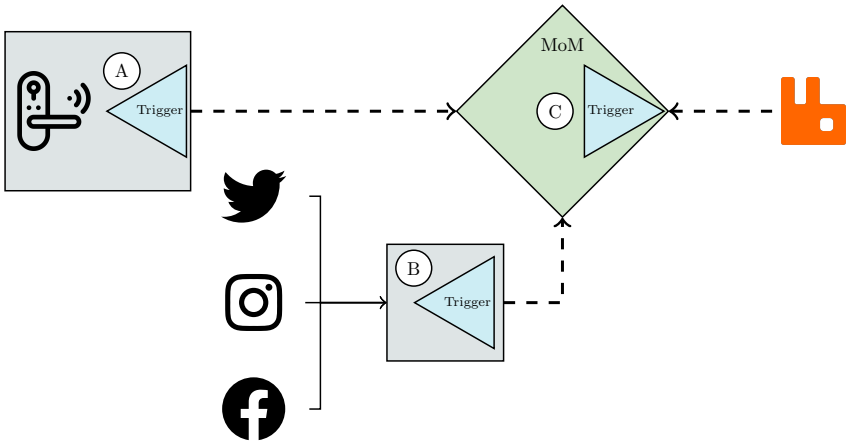


Figure 6.1: Different deployment options of the Trigger.

The location transparency introduced by the MOM also enables the deployment of triggers to adapt to different scenarios and needs. We have designed and implemented three deployment options for the Trigger, depending on its closeness to either the MOM or the external source.

In the first deployment scenario (Figure 6.1 case (A)), the Trigger is situated alongside the event source. This configuration allows for easy support of delivery quality between the source and the trigger, as they are located on the same host. However, this pattern limits the ability to use the trigger for multiple sources and necessitates that the source device has sufficient resources to host the trigger execution.

In the second scenario, i.e., Figure 6.1 case (B), the Trigger is positioned between external sources and the MOM. In this scenario, we can ensure that the delivery of information between the source and the trigger is of adequate quality. The Trigger can be located on any node accessible by both MOM and sources and can also serve as a gateway between different networks. In this configuration, multiple sources can address the Trigger, maximizing resource usage and potentially causing conflicts. However, concurrent transmission of incoming events belonging to the same quality class can result in conflicts; thus, a fine-grained distribution and allocation of triggers are advisable to prevent quality degradation.

In the final case, Figure 6.1 case (C), the external source already implements QoS concepts and exchanges information in events. This scenario encompasses use cases where TEMPOS is integrated into an existing infrastructure that already utilizes some form of event exchange, such as an Enterprise Service Bus (ESB) infrastructure. In this context, the

Trigger is placed within the TEMPOS MOM and acts as a connector to external sources. The Trigger is still responsible for mapping arguments, queues, and qualities of an external system to internal ones.

Processing slice

The final abstraction in the TEMPOS pipeline is the *Processing* slice, which is responsible for executing user-defined business logic and ensuring that the Quality of Service requirements are met.

The Processing layer is designed to take the burden off the customer of knowing both the characteristics of the processing environment and the computing resources used to execute a specific workflow. This allows the customer to define both the business logic and QoS requirements without worrying about how the platform implements the support that can satisfy them. Specifically, the processing is done through user-defined business code loaded in advance.

The main component responsible for the Processing layer behavior is the *Invoker*. The Invoker is the terminal part of each output channel and waits for the arrival of events to be processed by functions. Invoker instantiates the associated function ahead of time at each event arrival through the user-provided configuration and then forwards the event to the function.

No TEMPOS components, except the Controller, are aware of how a specific Invoker will process an event; Invoker is, therefore, the component in charge of managing the life cycle, the execution environment, and the invocation of the functions in such a way as to reach the target QoS for that workflow. TEMPOS Invokers can be specialized to exploit different function invocation methods and execution environments depending on deployment scenarios and achieving better performance or resource-saving.

This specialization can take advantage of the opportunistic composition of different technologies available in the environment of execution of the Invoker, e.g., Operating System, Hypervisor, or realized by the Invoker itself. So, for its execution, the same workflow can exploit different technologies and optimizations at the same time, e.g., concurrent usage of an execution environment for two different functions or re-usage of the same function instance for subsequent requests.

Regarding QoS-aware processing, Invoker can employ its internal techniques and the ones possibly present at the Executor Node, e.g., Operating System prioritization. This allows the Invoker to manage the execution of functions in a way that guarantees the target QoS for that workflow. Thanks to the Invoker abstraction, TEMPOS can execute heterogeneous functions while employing different QoS mechanisms and policies without causing side effects on other components or executor nodes.

6.1.2 The TEMPOS prototype

This section provides insight into the primary implementation of the TEMPOS architecture, starting with the QoS Level and then moving on to the System Level (as shown in Figure 6.2).

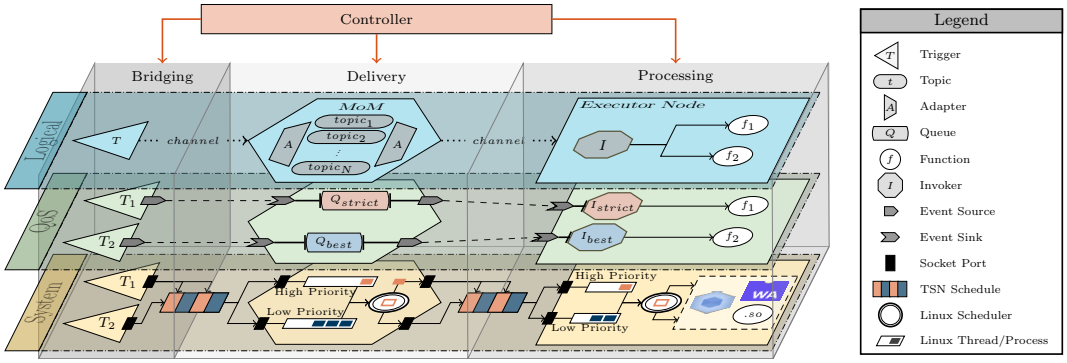


Figure 6.2: Multilevel representation of TEMPOS architecture foreseeing the three slices and the three conceptual layers.

Quality of Service Levels

The TEMPOS middleware currently offers application developers two distinct Quality of Service levels: the *Best-effort Quality (BQ)* and the *Strict Quality (SQ)*. The BQ level is intended when communication and function invocation do not have strict latency and jitter requirements. The SQ level, on the other hand, is designed for the execution of functions that require more stringent and soft real-time QoS, such as those that need to meet tight deadlines.

Controller

The TEMPOS middleware includes a Controller module that manages all other TEMPOS entities. The Controller is implemented as a Linux daemon process and is responsible for initializing and configuring the entire TEMPOS middleware. For the initialization to be successful, the application developer/deployer must provide a configuration file, currently in the TOML format, which contains all the information needed for the Controller to interact with other entities such as the MOM, Triggers, and Invokers, as well as the specification of the requested QoS levels for connections between components and function execution at each targeted node.

Once the initialization phase is complete, the Controller waits for reconfiguration/-management requests from the developer/deployer, making the Controller and the entire middleware reconfigurable and modifiable at run-time. The current implementation exposes the Controller functionality through REST APIs.

When the Controller receives a request, it performs the necessary reconfiguration by interacting synchronously with the entities involved in each slice. These entities, in turn, expose specific management interfaces and handle the configuration requests in an ad-hoc process outside the interactions of the TEMPOS workflows defined by the developer/deployer. In future work, we plan to implement this configuration mechanism through a special configuration topic on the MOM, where different TEMPOS components can subscribe to receive updated configurations.

Additionally, the Controller maintains an internal representation of all TEMPOS components, which is updated with each request, allowing for a centralized and updated view of the entire middleware deployment environment. This means that during the configuration phase, there is no need for direct interaction between the different TEMPOS components as the MOM mediates communication to guarantee strong decoupling between the infrastructure entities.

Message-oriented Middleware

The TEMPOS Message-oriented Middleware is a crucial component of the TEMPOS architecture. It is designed to provide transparent and flexible management of different QoS levels for real-time and best-effort traffic. The MOM comprises two *queues* for SQ and BQ traffic. These queues are implemented using two different network sockets and

two threads. The sockets are used to separate messages into two separate queues. At the same time, each thread is a priority queue processor, as they are scheduled according to the Linux real-time scheduler. The first thread handles all messages labeled with strict quality and runs with a higher priority than the best-effort thread. By default, the lowest priority (0) is used for BQ, while the highest priority (99) is associated with SQ. However, the application developer can use the Controller to specify the type of Linux real-time scheduler and set different values for the priorities of the threads associated with the queues, making the MOM more flexible and able to introduce additional queues with intermediate quality in the future.

One of the critical features of the TEMPOS MOM is its transparency of the protocols used by the underlying network. This is achieved through TEMPOS middleware elements called *Adapters*, which are implemented using a plugin-based mechanism within the MOM. Each plugin represents a set of well-defined interfaces that specify how to: open a connection, configure the QoS level of a newly created connection, send messages through the connection, and safely close the connection. The association between one or more channels connected to the MOM and one of the queue processors realizes the concept of a *TEMPOS Topic*. The TEMPOS components, including the MOM, are implemented using the Rust programming language, and the plugin system is based on the dynamic library loading mechanism. This allows the MOM to load plugins at runtime according to the configuration received from the Controller.

We have implemented a TSN-based plugin exploiting the IEEE 802.1Qbv standard. As discussed in Section 2.2.2, this standard aims to support the best-effort and real-time traffic within TSN networks and defines a mechanism to support different time-critical flows. It uses the concept of time-triggered communication windows, divided into multiple time slots associated with selected traffic classes and repeated cyclically. This allows for minimizing the interference of best-effort traffic with priority traffic, referred to as strict communication QoS level. Overall, the TEMPOS MOM provides a flexible and transparent way of managing different QoS levels for real-time and best-effort traffic within TSN networks.

Trigger

The *Trigger* is the TEMPOS component responsible for forwarding events issued by one or more sources to the MOM, thus defining the core part of the Bridging slice. We developed the Trigger as an always-running Linux process listening on a network socket to provide a unique implementation for the different deployment scenarios presented in Section 6.1.1. Thus, an event issued by a source (e.g., a sensor) is received by the Trigger via the network or, if possible, taking advantage of an IPC mechanism. This can be applied to optimize communication in the case of the co-located deployment scenario. Once executed, the Trigger receives the configuration containing the QoS level to be used from the Controller and then opens a second connection, i.e., the Channel used to communicate with the MOM. Different from the MOM, we consider the implementation of each Trigger as limited to a single protocol, be it TSN, Wi-Fi 6, or any other protocol providing a priority-based communication mechanism. At the moment, we have completed the implementation of the co-located trigger model by exploiting TSN-based communication.

Invoker

The *Invoker* is implemented as a multi-threaded TEMPOS component, which spawns two main threads. The first manages configuration requests from Controller, while the second handles actual invocations. Once started, Invoker receives its configuration from Controller and sets its QoS level based on the application developer's specification. Moreover, we developed three distinct invocation mechanisms, and the developer must express which one to use in the configuration request. The three invocation methods, namely DLF (Dynamically Loaded Function), WASMF (WASM Function), and FSpawn (Function Spawn), are designed to support different use cases. In the next sub-sections, we will introduce these methods in detail.

DLF (Dynamically Loaded Function) The DLF mechanism is based on *dynamic library loading* technique, which is commonly used to combine multiple functions into a single unit that can be shared by multiple processes at run-time, thus saving disk space and RAM. Although multiple processes can use the library code simultaneously, its variables remain isolated. In this method, the Invoker uses POSIX standard APIs to handle the dynamic loading of the library by calling `dlopen` to open the requested shared object file,

`dlsym` to load the symbol related to the main library entry point, and `dlclose` to unload the function after execution [45]. The developer must ensure the function is exposed within the library with the name and arguments expected by the Invoker. This invocation method is suitable for executing functions with high performance and strict requirements and is primarily aimed at our strict QoS level [42].

WASMF (WASM Function) The WASMF invocation method is similar to DLF, adopting the exact underlying loading mechanism. However, it uses the Wasmer library [76], a complete WASM engine, to handle the loading and execution of the function. The engine is initialized in the startup phase of the Invoker, and when a function request is received, the engine dynamically loads the shared library containing the requested function. To ensure correct loading, the library must be compiled using a WASM code generator, such as Cranelift [77], which converts a target-independent intermediate representation into executable machine code. Once the function is executed, the engine removes the WASM code from its internal store. This invocation method allows developers to implement functions in their preferred programming language and achieve good performance and quality results.

FSpawn (Function Spawn) The last invocation mechanism, FSpawn, uses the classic Unix idiom of `fork()` followed by `exec()` to execute a different program in a child process. However, if the Invoker is deployed on a node that supports the `posix_spawn` API, it is used instead of the `fork()` and `exec()` scheme to achieve better performance in case the parent process has a larger size or memory layout [15]. This invocation method is flexible and standard, allowing functions to be executed in arbitrary environments. As a first implementation, we leveraged FSpawn to execute the function as a Linux user-space process. Alternatively, a function can also be spawned inside an already-started Docker container, providing all the necessary dependencies for execution.

6.1.3 Experimental Evaluation

To conduct a quantitive evaluation of the efficacy of TEMPOS, we have developed a series of testbeds to analyze the behaviors of several of its primary components. The objectives of these testbeds are to demonstrate that the TEMPOS middleware can support

differentiated end-to-end Quality of Service levels while providing application developers with a simplified interaction and instrumentation interface. Special attention was given to demonstrating the ability of TEMPOS to orchestrate and compose different mechanisms to achieve highly differentiated QoS for different workflows. Furthermore, these testbeds also validate the implementation of the TEMPOS stack under a range of load conditions.

It should be noted that the performance results achievable by TEMPOS are dependent on the characteristics of resources in the targeted deployment environment. As such, the following series of testbeds can serve as a preliminary step toward calibrating resources in target deployment scenarios with similar technological stacks. Our testbeds are designed to simulate a worst-case scenario where the number of concurrent requests places the TEMPOS middleware under stress. In particular, we have organized our testbeds into three cases, each aimed at stressing the event-delivery process, the event processing, or the overall middleware.

In the first case, we test the behavior of TEMPOS event delivery under different load conditions, thus emulating diverse resource competition scenarios of workflows. The specific goal is to demonstrate the ability of our middleware to chain different QoS mechanisms while maintaining guarantees about latency and jitter.

The second case demonstrates the TEMPOS's ability to hide heterogeneity while still providing a strong differentiation of QoS. For this reason, in this testbed case, we trigger the execution of a complex and computationally heavy function, representative of many common workloads (as outlined in Algorithm 1) while employing all the different function invocation methods currently supported in our TEMPOS prototype.

In the final testbed case, we sought to evaluate TEMPOS's capability to compose mechanisms for QoS at different TEMPOS slices to achieve configurable and complete end-to-end QoS across various workflows. To accomplish this, we implemented and configured two distinct workflows, each invoking the same function (Algorithm 1), with one being configured at the BQ level and the other at the SQ level. All tests were designed to increase the number of requests for each workflow to examine TEMPOS's behavior under challenging dynamic changes in the supported service load. The results obtained were analyzed and discussed by presenting the overhead quotas introduced by individual TEMPOS components.

To thoroughly evaluate the feasibility and effectiveness of TEMPOS in Edge Cloud deployment environments, we have designed and executed a series of experiments on a

Algorithm 1 Pseudo code showing the operations performed by the function used in the tests: deserialization, count of occurrence in the text, and repetitions of operations of square root and power based on the index value.

```
1:  function MAIN(e : Event)                          ▷ The function entry point
2:      message, pattern ← deserialize(e)
3:      occur ← count_occurence(message, pattern)
4:      res ← 0
5:      for i ← 0, occur do
6:          if i mod 2 = 0 then
7:              res ← res+ pow(i)
8:          else
9:              res ← res+ sqrt(i)
10:         end if
11:     end for
12:     output(res)
13: end function
```

testbed consisting of three TSN-enabled nodes (Table 6.1). The nodes were chosen to have limited computational resources to simulate real-world scenarios where resources are constrained. The testbed was designed to investigate the ability of TEMPOS to compose mechanisms for Quality of Service at different TEMPOS slices to achieve configurable and complete end-to-end QoS over different workflows. As a small note, Node A and Node B are introduced and employed only in the second testbed case to examine the variation of invocation method performance.

Table 6.1: Specifications of the nodes used for the evaluation testbed.

Node Tag	Model	CPU	Memory	TSN driver
A	Custom Workstation	AMD Ryzen 3700X 8/16 CPU	32 GB	1 × Intel I211
B	Dell Optiplex 3010	Intel Core i5-3470 4/4 CPU	10 GB	-
C	UP Core Plus board	Intel Atom E3950 4/4 CPU	8 GB	4 × Intel I210
D,E	UP Xtreme board	Intel Core i3-8145UE 2/4 CPU	8 GB	4 × Intel I210

We selected to co-locate Triggers and the data Producer on Node E to simulate a practical scenario where two edge nodes cannot communicate through the utilization of differentiated QoS mechanisms. This decision to assign one of the two resource-rich nodes to these TEMPOS components is primarily driven by the requirement to generate high and precise loads to stress our middleware. Node D, the second most performant board, hosted an

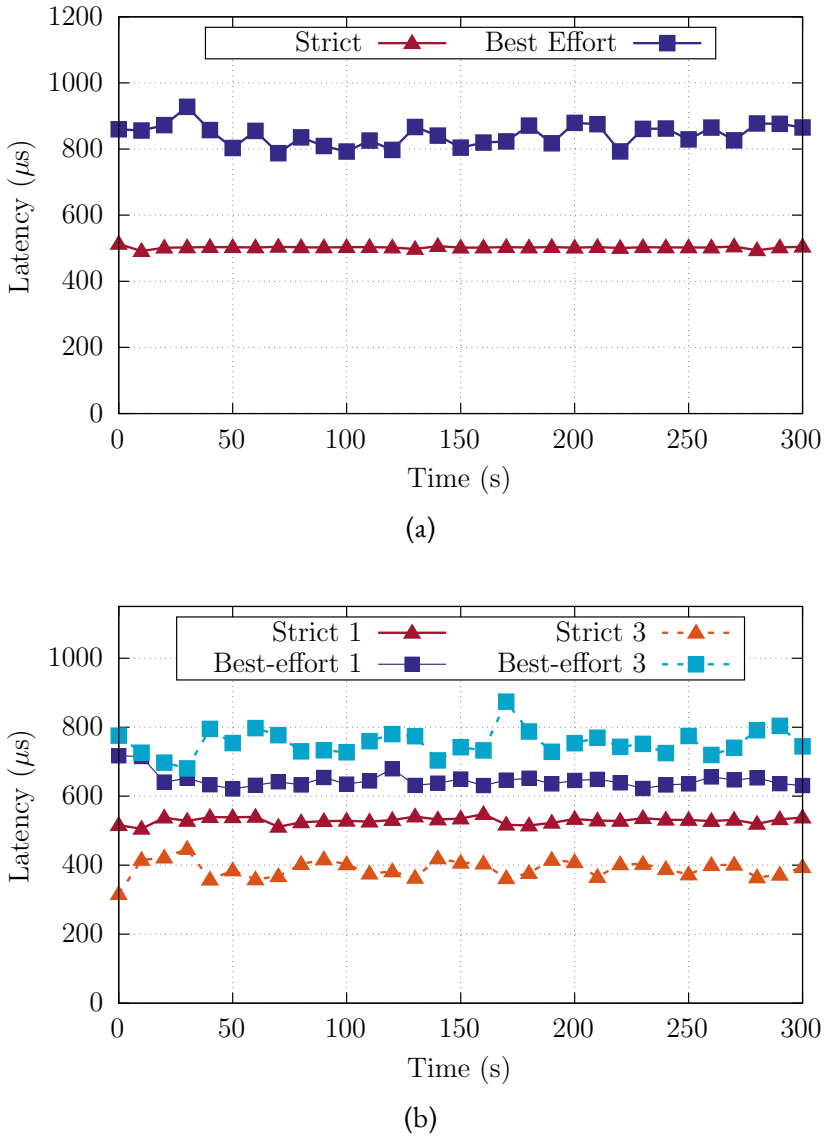


Figure 6.3: First testbed section showing performance of Delivery slice. (a) Average end-to-end latency of best effort and Strict effort traffic when executed in separate environments. (b) Average end-to-end latency of best effort and Strict effort traffic with 1 and 3 concurrent best effort producers and one strict effort.

instance of the TEMPOS MOM. Furthermore, we chose to deploy all invokers on the node with fewer resources to highlight concurrent resource requests and potential QoS conflicts in the processing slice, which is a common situation in practice.

The three nodes of our testbed were interconnected through a Relyum RELY-TSN-BRIDGE Ethernet switch, configured with a TSN setup that implements differentiated QoSchannels for the ingress/egress of topics. To establish the selected QoS mechanism for TEMPOS' best effort and strict effort levels, we utilized two *qdiscs* queuing disciplines built into the Linux kernel: *taprio* (Time-Aware Priority Shaper) implementing a simplified version of the scheduling defined by IEEE 802.1Qbv and *etf* (Earliest TxTime First) qdisc that allows applications to set a transmission time for each packet (this information is then used by the scheduler to de-queue the packet and forward it over the network). It should be noted that applications based on the IEEE 802.1Qbv standard must rely on a single-time reference. To this end, in our testbed, each TEMPOS node participates in electing a controlling entity, determined by the Best Master Clock Algorithm (BMCA): this controlling node, referred to as the PTP grandmaster, sends clock information to each of the Clock Slaves connected to it; once all TEMPOS devices are synchronized, we have what is effectively a time-aware network of nodes, i.e., a ready-to-use gPTP domain.

In our testbed, we created two time-aware TSN windows of 1 ms, i.e., between Trigger and the MOM and between the MOM and Invoker. Each window was divided into two time slots, one for SQ and one for BQ, each of 500 µs; the first SQ slot was scheduled in the first half of the first window, where the second SQ slot was skewed of 300 µs concerning the starting time of the MOM-Invoker window; this configuration enables strict TEMPOS traffic to find the gate open at each step, with no additional delays.

Event delivery

In the first testbed case, we aimed to demonstrate TEMPOS's ability to prioritize event delivery based on workflow QoS. Specifically, we submitted a constant rate of 1000 events per second to the Trigger for a time-lapse of 5 minutes. We then measured the difference between the timestamp corresponding to the event creation at the Trigger and the one reported at its delivery. We alternated the activation of SQ and BQ workflows to observe the behaviors of the two in a scenario with no perturbation due to concurrency. The results in Figure 6.3b show that lower end-to-end latency and jitter characterize the events belonging to the SQ workflow compared to those of the BQ workflow.

The end-to-end latency for SQ workflow events settled to 501 µs on average, demonstrating that TEMPOS is compatible with very challenging contexts that call for less than 1 ms response time, such as soft real-time ones.

It is worth noting that a clear differentiation between TEMPOS QoS levels is possible thanks to the combined exploitation of prioritization mechanisms acting at the network and event processing layers. In particular, the lower jitter is mainly due to the strict scheduling of events and the synchronization of TSN windows in the ingress and egress of the topic. The maximum latency that we measured throughout all tests for each hop is 223 µs for the delivery of one event to the TEMPOS MOM, 57 µs for event processing, and 299 µs for event delivery to Invoker. Overall, once transmitted by the Trigger, a packet reaches Invoker in no more than 700 µs, in full compliance with what is configured as the QoS request in the testbed setup.

Additionally, it is essential to note that the high priority assigned to the queue processor for SQ events prevents other applications in the user space running at the edge node (such as the MOM control thread) from stealing resources for event processing; this does not happen for BQ. TEMPOS considers these measurements the baselines for event delivery in the ideal case of the absence of perturbations.

In the second test of this testbed case, we investigate how the TEMPOS event-delivery mechanisms behave when multiple workflows are active and in competition for resources. This test consists of two rounds:

1. A constant rate of 1000 events per second per each active flow is submitted for 5 minutes, with one best QoS and one strict QoS workflow concurrently active.

2. The number of active best QoS workflows is increased to 3 while maintaining a constant rate of 1000 events per second per each active flow.

We decided to increase only the number of best QoS workflows in this test because the configuration of the current testbed makes the simultaneous sending of more than 1000 strict messages per second impossible. Moreover, in most practical scenarios, most events tend to belong to the Best-quality type.

The study results, as depicted in Figure 6.3a, demonstrate that the strict-quality latency is not significantly impacted by the concurrent execution of one or more best-effort workflows. In both rounds of experimentation, the latency remained under the threshold

of 600 µs, indicating that the concurrent execution of best-effort workflows does not result in significant penalties on strict-quality latency.

As a second-level observation, it was noted that the jitter was negligible in the first round of experimentation, despite the concurrent presence of two active flows of event delivery. However, a noticeable increase in jitter was observed in the second round of experimentation. This increase in jitter can be attributed to using a new API, NAPI (Network API), a device driver packet processing extension implemented to improve networking performance. In particular, NAPI implements an interrupt mitigation mechanism for network devices, which allows for the utilization of both interrupt request (IRQ) and polling-based packet reception modes. The utilization of the NAPI polling mode allows for substantial acceleration in terms of latency. It allows the kernel to periodically check incoming network packets without interruption, as explained in [55, 62]. However, this acceleration comes at the expense of increased jitter and CPU utilization.

In conclusion, the results indicate that the concurrent execution of BQ workflows does not result in significant penalties on SQ latency. Additionally, the utilization of the NAPI API allows for substantial acceleration in terms of latency but comes at the expense of increased jitter and CPU utilization. The ability to disable this feature through TEMPOS abstractions provides flexibility in the system's configuration, enabling performance optimization for a specific use case.

Processing

After validating the QoS-constrained delivery features of TEMPOS, we present a series of tests to show the TEMPOS performance in event processing. The following tests are therefore implemented by considering the Processing Layer only, with local-to-nodes function triggering.

The first test focuses on how different invocation and execution environments perform when run over heterogeneous hardware. To this purpose, we consecutively invoked the same function (Algorithm 1), programmed in a compiled language, for 2 minutes when invoked with the mechanism of *DLF*, *WASMF*, and *FSpawn*. We next repeated the test with the *FSpawn* mechanism but with two different versions of the same function implemented in two different interpreted languages, i.e., Python and JavaScript. These tests are repeated on nodes A, B, and C (Table 6.2) as representative of three very different

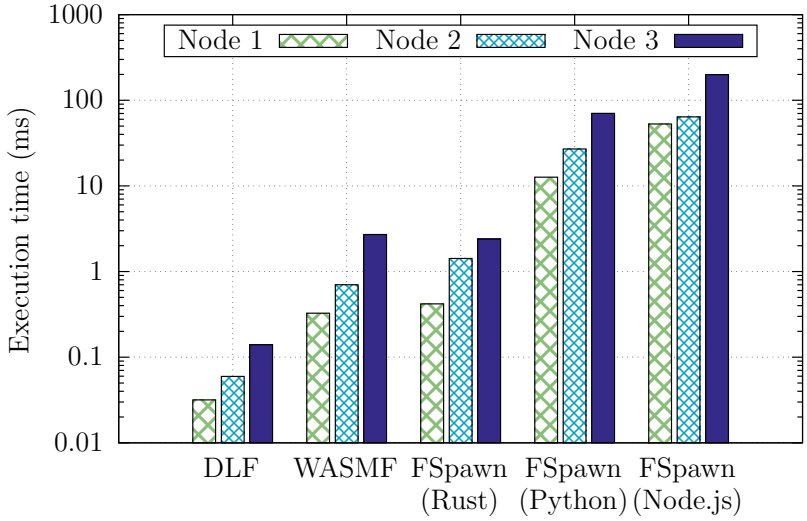


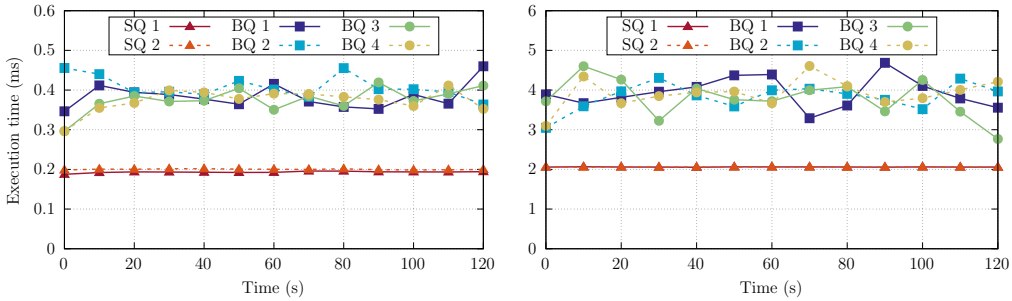
Figure 6.4: Mean execution times for the different invocation methods gathered in a run of 5 min. Each run repeated on nodes A, B, and C.

cases of resource availability on Edge hosts. All the results show that the employed hardware sensibly influences startup and execution times, with latency minor than 1 ms easily achievable on medium-top class hardware.

The *DLF* execution showed the best performance in terms of execution time, with latency near to 100 µs, making it the fastest mechanism to invoke functions. This opens up the application of TEMPOS in many challenging and latency-sensitive use cases where sub-millisecond end-to-end latency is needed. However, *DLF* restricts the usable programming languages to the only ones compatible with the generation of shared libraries.

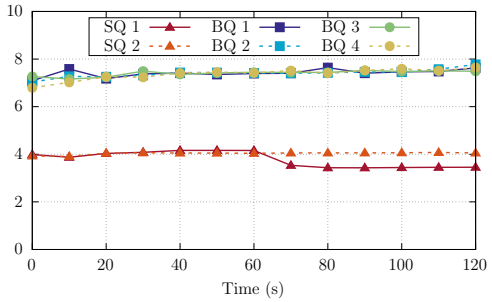
Table 6.2: Number of invocations executed by the different invocation methods during the processing test (5 min. run).

Invocation Mode	**Node A**	**Node B**	**Node D**
DLF	1884×10^3	1004×10^3	428×10^3
WASMF	183×10^3	85×10^3	22×10^3
FSpawn (Rust)	142×10^3	42×10^3	25×10^3
FSpawn (Python)	5×10^3	2×10^3	855
FSpawn (Node.js)	1×10^3	940	303



(a) Test results using the DLF invocation model.

(b) Test results using FSpawn invocation model.



(c) Test results using WASMF invocation model.

Figure 6.5: Testbed results of concurrent invocation of functions configured with different QoS.

The execution of the *FSpawn* mechanism demonstrated maximum flexibility, as it could run every language executable in a Linux environment. However, it exhibited poor performance in total execution time, with latencies reaching hundreds of milliseconds, particularly when running non-compiled languages (see Figure 6.4). Despite its flexibility, the measured performance of FSpawn renders it infeasible for deployment scenarios where end-to-end latency needs to be below the threshold of 1 ms.

The execution through *WASMF* performed one order of magnitude worse than *DLF* and only slightly better than the execution of a compiled function with *FSpawn*, with an execution time on the order of 1 ms. However, this mechanism demonstrated the potential to significantly reduce the execution and startup time of many non-compiled languages.

Table 6.2 illustrates that the choice of the appropriate invocation mechanism is a trade-off between the freedom of implementation language selection and the number of executable functions on a given hardware infrastructure. The initial test results also serve

as a baseline for subsequent results, as the first test was conducted without concurrent execution among workflows.

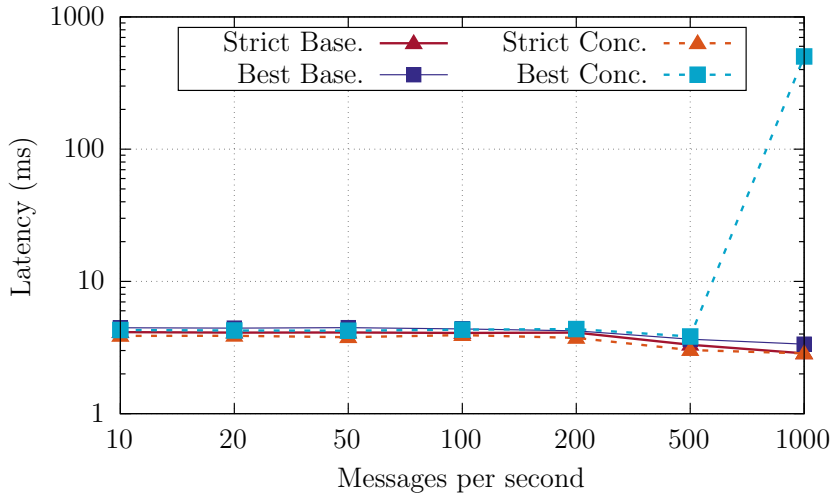
In the second test, we separately experimented again with the three invocation mechanisms, i.e., DLF, WASMF, FSpawn, but this time with concurrent invocations of 6 functions: 2 executed with SQ level and 4 with the BQ one. The level of parallelism was selected based on the number of cores available on the utilized nodes (Node C, see Table 6.1) and the need to create challenging resource conflicts among workloads in our tests. As illustrated in Figure 6.5, our queuing mechanism was able to prioritize strict quality when resource conflicts occurred effectively: the execution time of SQ functions was almost half that of BQ functions. In other words, the Invoker demonstrated the capability to correctly apply the requested prioritization even with heterogeneous mechanisms and in different execution environments.

Furthermore, the reported results revealed a negligible variability in execution time for SQ functions, in contrast to BQ functions. BQ functions displayed a significant variation in execution time on the order of hundreds to thousands of milliseconds, depending on the method used. Therefore, using SQ functions not only enables a reduction in latency but also allows for stricter predictability of processing time. The Invoker demonstrated the capability to execute heterogeneous workloads transparently while leveraging diverse technologies in infrastructure nodes.

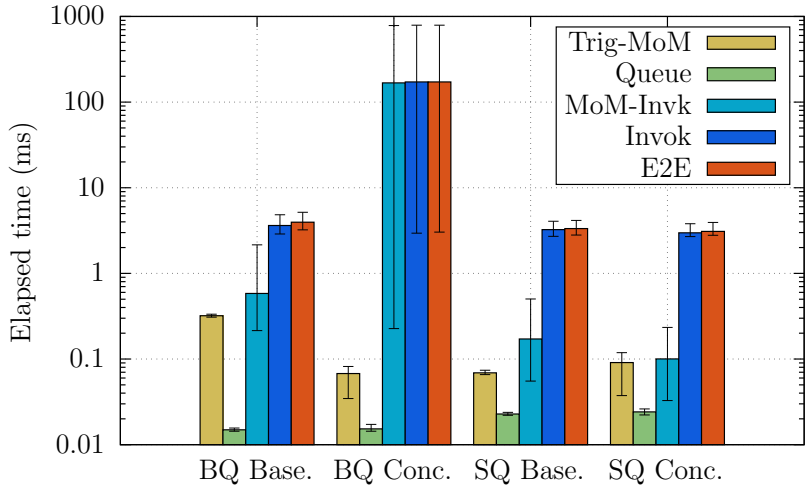
Full Stack

This section presents results about the TEMPOS system's ability to coordinate and concatenate different QoS mechanisms in each slice to achieve the targeted end-to-end quality for the workflows. We deployed on Node E two data producers and two Triggers configured with BQ and SQ levels, and on Node B, we deployed 3 Invokers with SQ configuration and 3 with BQ.

We then created and deployed two workflows executing the same function and triggered by the same event but configured one with BQ and one with SQ. Subsequently, we linearly increased the number of events submitted to the Triggers until reaching 1000 events per second for each workflow. The experiment was repeated firstly with only one active workflow, then with both workflows concurrently active.



(a) Comparison of end-to-end latency averages for BQ and SQ traffic executed both separately and simultaneously with an increasing number of messages.



(b) Zoom-in on end-to-end latency results showing single contributions of TEMPOS components (execution time) to the overall response times.

Figure 6.6: End-to-end test performance of the TEMPOS platform.

As predictable from the results of the previous sub-section, in an isolation case with only one workflow active per time, the SQ end-to-end latency is considerably better, with an average of 3.34 ms than BQ, which settled to an average of 3.96 ms, as also shown in Figure 6.6a. This difference in latency can be attributed to the QoS mechanisms implemented in the TEMPOS middleware, which prioritizes the execution of SQ workflows over BQ workflows.

It is also noteworthy that this behavior is maintained for the entire test duration, with different request rates, thus demonstrating the elasticity of the TEMPOS middleware. In the concurrent scenario, with both workflows active and competing for resources, the two workflows coexist and do not affect each other's performance until reaching the critical threshold of 500 messages per second. Until this threshold, we can also observe that both workflows behave similarly as in the previous experiments, where they were executed separately. However, over the critical threshold, we can observe that conflicts among workflows become critical, and the BQ workflows progressively degrade their performance. This clearly indicates the effectiveness of the implemented QoS mechanisms in managing resource allocation and preventing bottlenecks.

It is also worth mentioning that the latency performance of SQ workflows remains consistently approximately 3.1 ms despite the constrained hardware adopted and the concurrency with other workflows. This is a testament to the efficiency and scalability of the TEMPOS middleware in managing multiple workflows and ensuring the optimal performance of each workflow.

Zooming in on the performance behavior of some single TEMPOS components, we can observe (Figure 6.6b) how QoS mechanisms are correctly applied across all the hops of the technological stack. We can observe how, in each trait of the invocation stack, SQ performs almost identically when executed in concurrency with other workflows, while BQ workflows degrade their performance when competing with other active workflows. This further emphasizes the effectiveness of the implemented QoS mechanisms in managing resource allocation and ensuring the optimal performance of each workflow.

Let us finally note that in Figure 6.6b the "BQ Conc." MOM-Invk bar is almost the same as the Invok bar because the time is taken as the difference between the invoker function invocation instant and the sending message instant from the MOM. Given that the invoker reception is sync-blocking, that message waits in the invoker socket until the previous invocation is completed. This indicates that the MOM component can

handle the concurrency efficiently and does not introduce any significant overhead in the end-to-end latency.

In conclusion, the results of this experiment demonstrate the effectiveness of the TEMPOS middleware in managing multiple workflows and ensuring the optimal performance of each workflow, even in a concurrent scenario with constrained hardware. The implemented QoS mechanisms effectively manage resource allocation and prevent bottlenecks, resulting in the consistent and predictable performance of the system.

6.2 SELENE-based applications

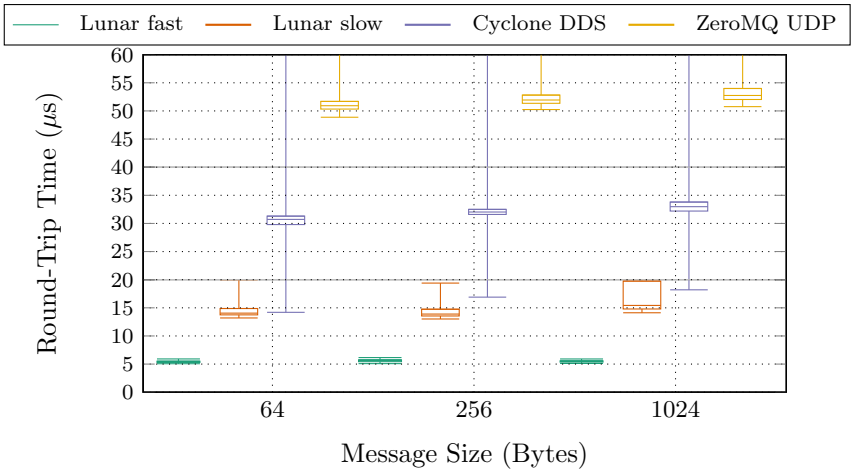
In Chapter 5, we discussed that a primary objective of the design of SELENE is to facilitate the development of a diverse range of applications with disparate requirements on Edge Cloud nodes (Section 5.3). To validate this design goal, we utilized the SELENE API to construct two representative Edge applications, a Message-oriented Middleware (*Lunar MoM*) and a data streaming framework (*Lunar Streaming*). Our results demonstrate that these simple applications are fully portable across various network technologies due to their SELENE-based implementation and can attain performance comparable to or even better than widely used systems of similar nature.

6.2.1 LUNAR MoM

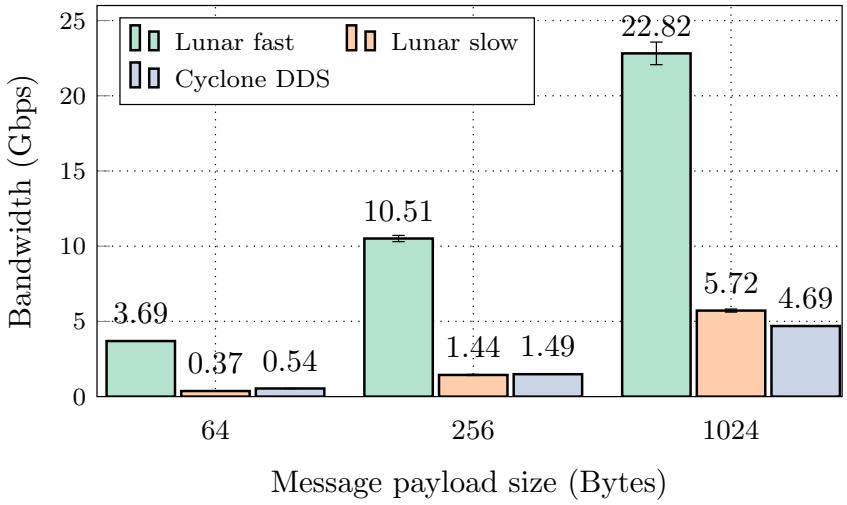
To demonstrate the ease of implementation and scalability of SELENE, we have developed a decentralized Message-oriented Middleware system, called *LunarMoM*, using the SELENE API. MOMs are commonly used in heterogeneous systems at the network edge for asynchronous, low-overhead communication. Depending on the deployment scenario and application needs, they can be either centralized or decentralized. They implement the publish-subscribe communication pattern, with the main concepts being *topics*, which are abstract named queues, and *publishers* and *subscribers* as producers and consumers of these queues.

Mapping the MOM abstractions to the SELENE primitives is straightforward. The LunarMoM application, which consists of only 135 lines of C code, defines two main primitives for publishing or subscribing to a topic: `lunar_publish` and `lunar_subscribe`. The publish function takes the topic name and a callback function as arguments. The

topic name is passed to a hashing function to obtain the topic id and to open a SELENE source if this is the first publication for that topic. Then, it obtains a buffer from SELENE, executes the user callback to fill it, and sends it. Under the hood, SELENE will deliver the messages to the reachable remote SELENE daemons and return them to the subscribed sinks. The subscriber function works symmetrically.



(a) Latency of MOMs for increasing payload sizes.



(b) Throughput of MOMs for increasing payload sizes

Figure 6.7: Performance benchmark for Lunar MoM and other reference systems.

Our demonstration of LunarMoM, a decentralized messaging system built using the SELENE API, shows that it offers an efficient option for the Edge Cloud. To evaluate its

performance, we compared LunarMoM against two widely used decentralized messaging systems in that environment, OMG DDS and ZeroMQ. We configured these systems to use a UDP transport without additional semantic reliability. We conducted performance benchmarks using a ping-pong pattern to measure the round-trip time (RTT) between a publisher and a remote subscriber and a throughput test to evaluate effective bandwidth utilization. The tests were conducted on the local testbed described in Section 5.3.4

The results, as shown in Figure 6.7a, indicate that LunarMoM has the lowest latency in both fast (using DPDK) and slow (using UDP) modes. Compared to the raw SELENE performance (Figure 5.14a), we observed that LunarMoM adds ns-scale overhead to SELENE, resulting in stable low latency. The performance of Cyclone (+45 %) is comparable to that of systems that use blocking sockets in their receiver thread, although with higher variability. ZeroMQ's UDP support, on the other hand, adds additional 20 µs latency compared to Cyclone. Similar considerations apply to the throughput evaluation (Figure 6.7b), where DPDK allows LunarMoM to significantly increase bandwidth utilization, while Cyclone and LunarMoM slow have similar behavior. ZeroMQ showed unstable performance and was excluded from the graph.

In conclusion, our experimentation demonstrates that SELENE dramatically simplifies the development of a lightweight messaging system that outperforms currently available alternatives, with ns-scale latency overhead compared to the SELENE interface. Additionally, LunarMoM is portable across all supported kernel-based and kernel-bypassing technologies, making it a promising solution for data dissemination at the network edge. LunarMoM is still a prototype, but we believe it shows how existing messaging systems could significantly leverage SELENE to improve their performance and portability.

6.2.2 LUNAR Streaming framework

In edge cloud scenarios, we often need to deal with real-time streaming and analysis of large amounts of data, such as intelligent applications based on Machine Learning (ML) or image processing. Especially in an industrial setting, it is common to have applications where cameras take images of a product during different stages of production and transmit them in real time to a central computing node for processing. If defects are present in the semi-finished product, a control application interacts with the production line to handle the failure.

Resolution	*HD*	*Full HD*	*2K*	*4K*	*8K*
Size (MB)	2.76	6.22	11.6	24.88	99.53

Table 6.3: Size of the images sent in the streaming benchmark.

To support Quality of Service requirements, streaming applications frequently use data fragmentation and compression techniques. For our prototype, *Lunar Streaming*, we decided to use only fragmentation, leaving compression as future development, as it is outside the scope of our test framework. Lunar Streaming exposes a simple set of APIs, allowing clients to connect to the server application, which must implement a simple interface by exposing two methods: `get_frame` and `wait_next`. The first allows getting a new frame, while the second pause the server waiting for the next frame. To start streaming, the server application must invoke `lnr_s_loop`, which performs the following steps: requesting a new frame, fragmenting and sending the frame, and waiting for the next frame to restart the loop until the end of streaming.

We tested Lunar Streaming by implementing a simple application that streams raw images, i.e., for each image frame, we send RGB values for every pixel. We used sample images of different standard sizes (Table 6.3) and compared our SELENE-based implementation with one that uses the `sendfile` primitive. Since `sendfile` sends data directly from a file descriptor loaded into the kernel without involving user space, it implements a sender-side zero-copy technique. For this reason, we believe it can be a good reference for our framework.

To demonstrate the performance of our streaming prototype, Lunar Streaming, we evaluated the following:

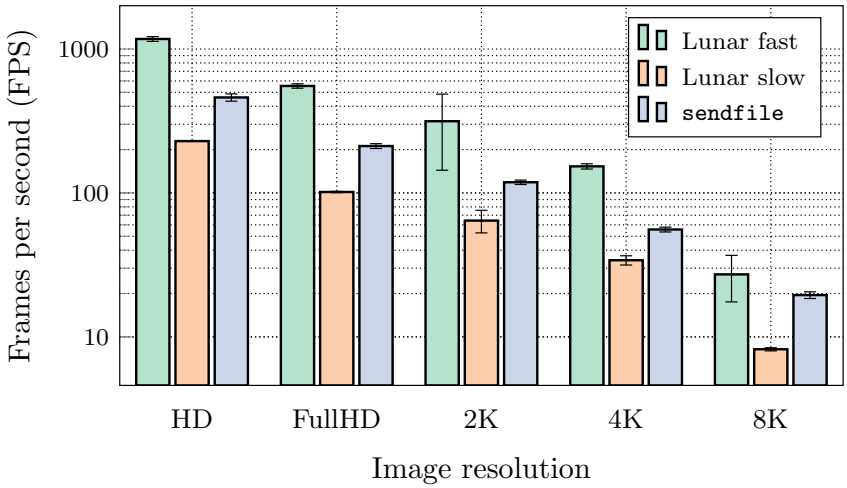
1. The number of frames per second (FPS) the client application can handle (Figure 6.8a)

2. The average end-to-end latency for frame transmission (Figure 6.8b), i.e., the time between the server application sending a frame (including fragmentation) and the client application receiving the reconstructed frame.

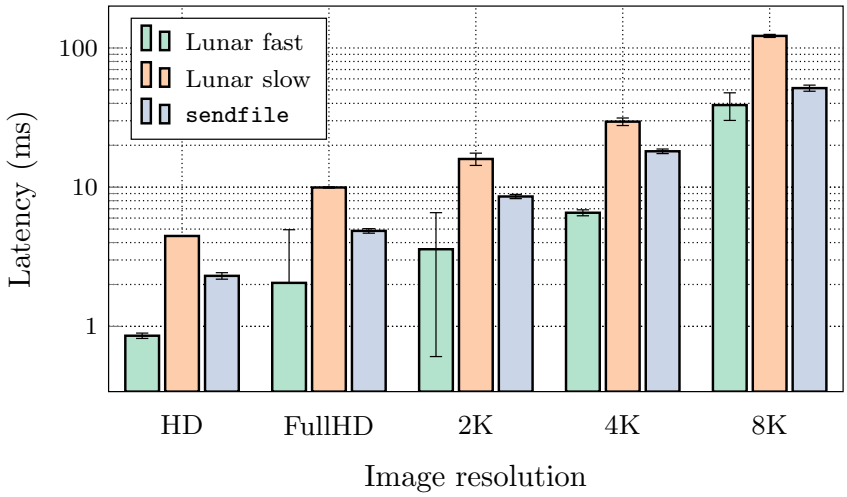
As seen from the results, Lunar Streaming achieves outstanding performance in both latency and FPS, particularly in the fast case. In particular, for images up to 4k, we can

support frame rates above 100 FPS and even above 1000 FPS for low-quality images. Latency never exceeds 10 ms for pictures up to a maximum resolution of 4k, making Lunar Streaming an excellent candidate for applications such as the tactile internet [66] or real-time simulations (e.g., Cloud Gaming [51]).

Finally, it is worth noting that streaming applications typically send various media, such as video and images, in a compressed format. For example, HVEC, VP9, or the newer VVC are commonly used as video CODECs [50] and are transmitted using protocols such as RTP or WebRTC [19]. While implementing a full stack of streaming protocols is beyond the scope of this work, it is essential to note that by sending raw images alone, we have already obtained excellent results. This highlights that SELENE can be effectively used to accelerate existing streaming frameworks [10].



(a) FPS for increasing image resolution.



(b) Latency per frame for increasing image resolution

Figure 6.8: Performance benchmark for Lunar Stream and `sendfile`.

7 Conclusions and Future Work

In conclusion, this thesis work has thoroughly investigated the evolution of IoT applications and the challenges posed by integrating cloud infrastructures with edge nodes. The traditional cloud-centric model has become insufficient to meet the growing demands of next-generation IoT applications, leading to the development of the Cloud Continuum model. This new model provides a more fluid approach to managing heterogeneous physical and virtualized resources while ensuring that different QoS requirements are met.

Despite the availability of various solutions to address the individual problems associated with network communications in such environments, there remains a lack of a comprehensive system architecture that can provide end-to-end QoS support for IoT applications. This gap in the literature prompted the research presented in this thesis, which proposed a new architecture designed to address this issue.

In the following paragraphs, we will revisit our proposed architecture's critical components highlighting its strengths and weaknesses. Additionally, we will discuss potential future developments aimed at addressing any limitations.

Chapter 4 presents a practical solution to the OT/IT convergence issue by utilizing the Edge Cloud paradigm to blur the OT/IT separation. The solution employs a two-layered MOM approach as an interoperability layer at the OT and facilitates the rapid transfer of large data volumes to the IT. The proposal's validity was demonstrated through a real-world testbed utilizing machine data and showing the capability of the components to handle increasing data volumes efficiently.

As ongoing research, we are examining the feasibility of implementing a pluggable processing mechanism at the Gateway for OT data filtering and aggregation. This mechanism aims to alleviate the pressure on the upward path by providing dynamic and selective processing at the edge, possibly adopting an event-centric serverless processing model, as discussed in Section 6.1. Additionally, we are investigating the implementation of a load-balancing mechanism at the gateway to enhance the reliability and scalability of the

proposal under increasing traffic pressure from the OT layer. Another area of investigation is enabling secure IT-to-OT data flow to facilitate the deployment of future cognitive agents at the IT layer.

Later in the same chapter, we presented a practical, end-to-end TSN-compliant QoS management approach that can handle reconfiguration events and was validated in a real testbed. We are exploring integrating SDN concepts to enhance the approach. SDN decouples the control plane from the data plane and centralizes control through a controller that sets forwarding tables for all switches, addressing traditional network management complexities and compatibility issues. Our system could utilize an SDN controller, such as OpenDaylight or ONOS, to configure switches with different capabilities or protocols via a connection to the CNC.

In Chapter 5, two novel approaches were presented for virtualized network environments. The first approach aimed to execute TSN-based applications with Ultra-low Latency constraints in virtual machines by combining a practical clock synchronization approach for remote VMs with high-performance network virtualization techniques. The solution was validated in a real testbed and demonstrated the ability to respect ULL constraints, allowing unmodified critical applications to benefit from virtualization advantages. Future work includes exploring other kernel-bypassing techniques, such as XDP, Open vSwitch (OVS) DPDK offload, RDMA, or SmartNICs, to support high-performance packet processing with fewer resources.

The second approach was KuberneTSN, an accelerated and deterministic container overlay network architecture that uses a novel userspace TSN packet scheduler and a kernel-bypassing approach to minimize packet processing delays. KuberneTSN was implemented as a network plugin for the Kubernetes orchestrator, *tsn-cni*, and evaluated on a real testbed, outperforming the widely used Flannel network plugin for containerized applications. Future work includes performance characterization under different traffic conditions and demonstration of *tsn-cni* with other network plugins. In the long term, as performance-critical AI/ML components move to the network Edge, a systematic performance study of the inter-container datapath will be conducted to identify optimization opportunities. Also, the full implementation of the TSN user-space scheduler may unlock these approaches' full potential.

In Chapter 5, the SELENE middleware was introduced as a solution for integrating heterogeneous communication technologies, including kernel UDP/IP, XDP, DPDK,

and RDMA, at the Network Edge. The SELENE middleware offers a simple yet flexible API, allowing for the development of a wide range of portable Edge applications with multiple data flows with varying requirements, as shown in Section 6.2. The user only needs to specify high-level requirements for these flows, and the SELENE runtime will efficiently allocate the most appropriate technology in the dynamically determined deployment environment.

Developing SELENE raised several important research questions in network Edge. The following are the significant open challenges in the deployment of SELENE:

- Receiver-side overhead: The current implementation of SELENE dedicates a single core to network operations, leading to pressure on the receive pipeline. Two possible solutions are offloading operations to hardware devices or parallelizing them across multiple cores.

- Zero-copy fragmentation: The current implementation does not support UDP/IP packet fragmentation and relies on jumbo frames. A technique for zero-copy data reconstruction remains an open challenge.

- Scheduling: A careful scheduling strategy is crucial for high-performance systems like SELENE. The prototype uses a FIFO strategy, but future work will introduce TSN-compliant scheduling for improved network latency for time-critical applications.

- Virtualization: Network acceleration techniques often contrast with virtualization principles, and virtualized network acceleration is a crucial research direction for solutions like SELENE.

- Security: Security implications of SELENE have not been studied yet and represent the biggest challenge, especially at the network Edge. Leveraging programmable network hardware to increase security is the most promising approach.

Finally, in Chapter 6 we presented TEMPOS, a QoS-aware middleware for serverless platforms. TEMPOS integrates different QoS mechanisms provided by individual technologies, such as Linux real-time scheduling and Time-Sensitive Networking protocols, to manage end-to-end QoS in terms of jitter, latency, and en-queuing time. The main idea

behind TEMPOS is to provide a flexible API that eases the development of portable Edge applications by allowing the user to specify high-level requirements for data flows. The TEMPOS runtime then maps these requirements to the most appropriate technology available in the deployment environment.

To evaluate the validity of TEMPOS, we conducted a series of real testbed experiments. The results showed that TEMPOS effectively differentiated workflows based on the assigned QoS level. QoS awareness was preserved throughout the entire invocation stack, with the delivery layer achieving nearly twice the performance for event delivery for SQ workflows compared to BQ workflows, even under concurrent execution. The TEMPOS processing slice also leveraged multiple invocation methods seamlessly, ensuring that higher priority (SQ) workflows executed twice as fast as lower priority (BQ) workflows.

In future work, we plan to integrate TEMPOS with a resource orchestrator for the full Cloud Continuum chain, including 5G micro-datacenters and traditional geographically distant cloud data centers. This integration will allow TEMPOS to handle both network and computing resources fully. Additionally, we aim to introduce new levels of QoS considering latency and jitter differentiation and the semantics of delivery and throughput while expanding support to resources not considered in this work, such as storage or hardware accelerators.

In conclusion, future developments in the field of architecture entail the integration of various works discussed in the thesis to realize layered architecture, as outlined in Chapter 3. This integration will involve exploring solutions that serve as connectors between different levels and modules. Moreover, the scope of these future directions extends beyond the network domain and encompasses computing aspects such as GPU, CPU, and acceleration technologies like FPGAs, as well as the storage component.

Expanding the studied approaches from the network domain to the computational domain is essential. This expansion involves applying the methodologies and insights gained from network-related investigations to the computing infrastructure. By doing so, overall system performance and efficiency can be enhanced through the optimization and offloading of computationally intensive tasks onto suitable computational resources.

Additionally, the exploration and development of monitoring and observability components are of utmost importance. These components play a crucial role in ensuring the reliability, performance, and security of the architecture. By improving and broadening the capabilities of the monitoring and observability component, a comprehensive

understanding of system behavior can be achieved, enabling proactive management and mitigation of potential issues.

The proposed future directions aim to fulfill the architectural requirements of a comprehensive system capable of effectively supporting and adapting to the evolving landscape of IoT applications. By incorporating the findings from various works and expanding their coverage to encompass a wider range of components and technologies, the architecture can provide end-to-end quality of service (QoS) support, ensuring efficient and reliable operation of IoT applications in dynamic and challenging environments.

Bibliography

1. [n.d.] *DPDK Ring Library*. [Online]. URL: https://doc.dpdk.org/guides/prog_guide/ring_lib.html.
2. [n.d.] *libfabric, Open Fabrics Interface*. [Online]. URL: https://github.com/ofiwg/libfabric.
3. [n.d.] *libxdp*. [Online]. URL: https://github.com/xdp-project/xdp-tools.
4. [n.d.] *PCI SIG. Single Root I/O Virtualization*. [Online]. URL: https://pcisig.com/specifications/iov/single_root/.
5. [n.d.] *RDMA Aware Networks Programming User Manual*. [Online]. URL: www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.
6. [n.d.] *The Data Plane Development Kit*. [Online]. URL: www.dpdk.org.
7. 5G-AICA. *Integration of 5G with Time-Sensitive Networking for Industrial Communications*. 2021. URL: https://www.5g-acia.org/whitepapers/integration-of-5g-with-time-sensitive-networking-for-industrial-communications/.
8. 5GACIA. "5G for Connected Industries and Automation". November, 2019. URL: https://www.5g-acia.org/index.php?id=5125.
9. L. Abeni and D. Faggioli. "Using Xen and KVM as real-time hypervisors". *Journal of Systems Architecture* 106, 2020, p. 101709.
10. A. Altonen, J. Räsänen, J. Laitinen, M. Viitanen, and J. Vanne. "Open-source RTP library for high-speed 4K HEVC video streaming". In: *2020 IEEE 22nd International Workshop on Multimedia Signal Processing (MMSP)*. IEEE. 2020, pp. 1–6.

11. G. Ara, T. Cucinotta, L. Abeni, and C. Vitucci. "Comparative Evaluation of Kernel Bypass Mechanisms for High-performance Inter-container Communications." In: *CLOSER*. 2020, pp. 44–55.

12. I. T. Association. *Supplement to InfiniBand Architecture Specification*. Release 1.2.2 Annex A16: RDMA over Converged Ethernet (RoCE). 2010.

13. C. Avasalcai, B. Zarrin, and S. Dustdar. "EdgeFlow—Developing and deploying latency-sensitive IoT edge applications". *IEEE Internet of Things Journal* 9:5, 2021, pp. 3877–3888.

14. L. Baresi, D. Filgueira Mendonça, and M. Garriga. "Empowering low-latency applications through a serverless edge computing architecture". In: *Service-Oriented and Cloud Computing: 6th IFIP WG 2.14 European Conference, ESOCC 2017, Oslo, Norway, September 27-29, 2017, Proceedings 6*. Springer. 2017, pp. 196–210.

15. A. Baumann, J. Appavoo, O. Krieger, and T. Roscoe. "A fork () in the road". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2019, pp. 14–22.

16. A. Bierman, M. Bjorklund, and K. Watsen. *RESTCONF protocol*. Technical report. 2017.

17. L. Bittencourt, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. DaSilva, C. Lee, and O. Rana. "The internet of things, fog and cloud continuum: Integration and challenges". *Internet of Things* 3, 2018, pp. 134–155.

18. M. Bjorklund. *YANG-a data modeling language for the network configuration protocol (NETCONF)*. Technical report. 2010.

19. N. Blum, S. Lachapelle, and H. Alvestrand. "WebRTC: Real-time communication for the open web platform". *Communications of the ACM* 64:8, 2021, pp. 50–54.

20. F. Bosi, A. Corradi, L. Foschini, S. Monti, L. Patera, L. Poli, and M. Solimando. "Cloud-enabled smart data collection in shop floor environments for industry 4.0". In: *2019 15th IEEE International Workshop on Factory Communication Systems (WFCS)*. IEEE. 2019, pp. 1–8.

21. H. Boyes, B. Hallaq, J. Cunningham, and T. Watson. "The industrial internet of things (IIoT): An analysis framework". *Computers in industry* 101, 2018, pp. 1–12.

22. Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal. "Understanding host network stack overheads". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 2021, pp. 65–77.

23. Carlsson, T. *Industrial network market shares 2020 according to HMS networks. Available online:* https://www.hms-networks.com/news-and-insights/news-from-hms/2020/05/29/industrial-network-market-shares-2020-according-to-hms-networks *(accessed on 1 Nov 2021).*

24. G. Castellano, F. Esposito, and F. Risso. "A distributed orchestration algorithm for edge computing resources with guarantees". In: *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE. 2019, pp. 2548–2556.

25. Confluent Inc. *Authorization using ACLs | Confluent Documentation. Available online:* https://docs.confluent.io/platform/current/kafka/authorization.html *(accessed on 1 Nov 2021).*

26. Confluent Inc. *Kafka Consumer – Offset Management | Confluent Documentation. Available online:* https://docs.confluent.io/platform/current/clients/consumer.html#offset-management *(accessed on 21 Nov 2022).*

27. D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen. "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 467–481.

28. D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, et al. "The Design and Operation of CloudLab." In: *USENIX Annual Technical Conference*. 2019, pp. 1–14.

29. R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. *Network configuration protocol (NETCONF)*. Technical report. 2011.

30. J. Farkas, L. L. Bello, and C. Gunther. "Time-sensitive networking standards". *IEEE Communications Standards Magazine* 2:2, 2018, pp. 20–21.

31. T. Gerhard, T. Kobzan, I. Blöcher, and M. Hendel. "Software-defined flow reservation: Configuring IEEE 802.1 Q time-sensitive networks by the use of software-defined networking". In: *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2019, pp. 216–223.

32. O. M. Group. *OMG Data Distribution Standard*. [Online]. URL: https://www.dds-foundation.org/omg-dds-standard.

33. M. Gundall, C. Glas, and H. D. Schotten. "Introduction of an architecture for flexible future process control systems as enabler for industry 4.0". In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. IEEE. 2020, pp. 1047–1050.

34. M. Gutiérrez, A. Ademaj, W. Steiner, R. Dobrin, and S. Punnekkat. "Self-configuration of IEEE 802.1 TSN networks". In: *2017 22nd IEEE international conference on emerging technologies and factory automation (ETFA)*. IEEE. 2017, pp. 1–8.

35. S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. "MegaPipe: a new programming interface for scalable network I/O". In: *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 2012, pp. 135–148.

36. J. Harmatos and M. Maliosz. "Architecture Integration of 5G Networks and Time-Sensitive Networking with Edge Computing for Smart Manufacturing". *Electronics* 10:24, 2021, p. 3085.

37. H. Hoang, B. Cassell, T. Brecht, and S. Al-Kiswany. "RocketBufs: a framework for building efficient, in-memory, message-oriented middleware". In: *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*. 2020, pp. 121–132.

38. Y. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young. "ETSI White Paper # 11 Mobile Edge Computing-a key technology towards 5G-etsi_wp11_mec_a_key_technology_towards_5g.pdf". *ETSI White Pap. No. 11 Mob* 11, 2015, pp. 1–16.

39. IEEE Computer Society. *802.1AS-2020 - IEEE Standard for Local and Metropolitan Area Networks -Timing and Synchronization for Time-Sensitive Applications*. Vol. 2020. 2020. ISBN: 9781504464307.

40. E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. "mtcp: a highly scalable user-level {TCP} stack for multicore systems". In: *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 2014, pp. 489–502.

41. S. Jha, L. Rosa, and K. Birman. “Spindle: Techniques for optimizing atomic multicast on rdma”. In: *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2022, pp. 1085–1097.

42. C. Jung, D.-K. Woo, K. Kim, and S.-S. Lim. “Performance characterization of prelinking and preloadingfor embedded systems”. In: *Proceedings of the 7th ACM & IEEE international conference on Embedded software*. 2007, pp. 213–220.

43. A. Kalia, M. Kaminsky, and D. G. Andersen. “Design guidelines for high performance {RDMA} systems”. In: *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*. 2016, pp. 437–450.

44. M. Karlsson and B. Töpel. “The path to DPDK speeds for AF XDP”. In: *Linux Plumbers Conference*. 2018.

45. M. Kerrisk. *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010.

46. Kubernetes Network Plumbing Working Group. *Multus CNI*. URL: https://github.com/k8snetworkplumbingwg/multus-cni.

47. LAN/MAN Standards Committee of the IEEE Computer Society. *IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks – Amendment 31: Stream Reservation Protocol (SRP) Enhancements and Performance Improvements*. Vol. 2018. 2018, pp. 1–208. ISBN: VO -.

48. L. Leonardi, L. L. Bello, and G. Patti. “Towards time-sensitive networking in heterogeneous platforms with virtualization”. In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. IEEE. 2020, pp. 1155–1158.

49. B. Li, T. Cui, Z. Wang, W. Bai, and L. Zhang. “SocksDirect: Datacenter sockets can be fast and compatible”. In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 90–103.

50. I. Mansri, N. Doghmane, N. Kouadria, S. Harize, and A. Bekhouch. “Comparative evaluation of VVC, HEVC, H. 264, AV1, and VP9 encoders for low-delay video applications”. In: *2020 Fourth International Conference on Multimedia Computing, Networking and Applications (MCNA)*. IEEE. 2020, pp. 38–43.

51. L. Mazzuca, A. Garbugli, A. Sabbioni, A. Bujari, and A. Corradi. "Towards a Resource-aware Middleware Support for Distributed Game Engine Design". In: *Proceedings of the 2022 ACM Conference on Information Technology for Social Good*. 2022, pp. 409–413.

52. A. Nasrallah, A. S. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein, and H. ElBakoury. "Ultra-low latency (ULL) networks: The IEEE TSN and IETF DetNet standards and related 5G ULL research". *IEEE Communications Surveys & Tutorials* 21:1, 2018, pp. 88–145.

53. N. G. Nayak, F. Dürr, and K. Rothermel. "Time-sensitive software-defined network (TSSDN) for real-time applications". In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. 2016, pp. 193–202.

54. P. Neira-Ayuso, R. M. Gasca, and L. Lefevre. "Communicating between the kernel and user-space in Linux using Netlink sockets". *Software: Practice and Experience* 40:9, 2010, pp. 797–810.

55. *networking:napi [Wiki]*. URL: https://wiki.linuxfoundation.org/networking/napi.

56. J. Pan and J. McElhannon. "Future edge cloud and edge computing for internet of things applications". *IEEE Internet of Things Journal* 5:1, 2017, pp. 439–449.

57. A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. "Improving network connection locality on multicore systems". In: *Proceedings of the 7th ACM european conference on Computer Systems*. 2012, pp. 337–350.

58. B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, et al. "The design and implementation of open vswitch". In: *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 2015, pp. 117–130.

59. P. Pop, M. L. Raagaard, M. Gutierrez, and W. Steiner. "Enabling fog computing for industrial automation through time-sensitive networking (TSN)". *IEEE Communications Standards Magazine* 2:2, 2018, pp. 55–61.

60. M. L. Raagaard, P. Pop, M. Gutiérrez, and W. Steiner. "Runtime reconfiguration of time-sensitive networking (TSN) schedules for fog computing". In: *2017 IEEE Fog World Congress (FWC)*. IEEE. 2017, pp. 1–6.

61. R. Russell. "virtio: towards a de-facto standard for virtual I/O devices". *ACM SIGOPS Operating Systems Review* 42:5, 2008, pp. 95–103.

62. J. H. Salim, R. Olsson, and A. Kuznetsov. "Beyond Softnet." In: *Annual Linux Showcase & Conference*. Vol. 5. 2001, pp. 18–18.

63. J. Santos, T. Wauters, B. Volckaert, and F. De Turck. "Towards low-latency service delivery in a continuum of virtual resources: State-of-the-art and research directions". *IEEE Communications Surveys & Tutorials* 23:4, 2021, pp. 2557–2589.

64. M. Satyanarayanan. "The emergence of edge computing". *Computer* 50:1, 2017, pp. 30–39.

65. R. E. Schantz, J. P. Loyall, C. Rodrigues, D. C. Schmidt, Y. Krishnamurthy, and I. Pyarali. "Flexible and adaptive qos control for distributed real-time and embedded middleware". In: *Middleware 2003: ACM/IFIP/USENIX International Middleware Conference Rio de Janeiro, Brazil, June 16–20, 2003 Proceedings 4*. Springer. 2003, pp. 374–393.

66. S. K. Sharma, I. Woungang, A. Anpalagan, and S. Chatzinotas. "Toward tactile internet in beyond 5G era: Recent advances, current issues, and future directions". *Ieee Access* 8, 2020, pp. 56948–56991.

67. W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. "Edge computing: Vision and challenges". *IEEE internet of things journal* 3:5, 2016, pp. 637–646.

68. S. Shillaker and P. Pietzuch. "Faasm: Lightweight isolation for efficient stateful serverless computing". *arXiv preprint arXiv:2002.09344*, 2020.

69. M. Sojka, P. Pıša, D. Faggioli, T. Cucinotta, F. Checconi, Z. Hanzálek, and G. Lipari. "Modular software architecture for flexible reservation mechanisms on heterogeneous resources". *Journal of Systems Architecture* 57:4, 2011, pp. 366–382.

70. W. Song, Y. Yang, T. Liu, A. Merlina, T. Garrett, R. Vitenberg, L. Rosa, A. Awatramani, Z. Wang, and K. Birman. "Cascade: An Edge Computing Platform for Real-time Machine Intelligence". In: *Proceedings of the 2022 Workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems*. 2022, pp. 2–6.

71. O. Specification. "The real-time publish-subscribe protocol (RTPS) DDS interoperability wire protocol specification". *Object Management Group Pct07-08-04*, 2007.

72. K. B. Stanton. "Distributing deterministic, accurate time for tightly coordinated network and software applications: IEEE 802.1 AS, the TSN profile of PTP". *IEEE Communications Standards Magazine* 2:2, 2018, pp. 34–40.

73. L. Toka. "Ultra-reliable and low-latency computing in the edge with kubernetes". *Journal of Grid Computing* 19:3, 2021, pp. 1–23.

74. M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira. "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications". *ACM Computing Surveys (CSUR)* 53:1, 2020, pp. 1–36.

75. J. Wang, D. Behrens, M. Fu, L. Oberhauser, J. Oberhauser, J. Lei, G. Chen, H. Härtig, and H. Chen. "{BBQ}: A Block-based Bounded Queue for Exchanging Data and Profiling". In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022, pp. 249–262.

76. *wasmer_engine_dylib – Rust*. URL: https://docs.rs/wasmer-engine-dylib/2.0.0/wasmer%5C_engine%5C_dylib/.

77. *wasmtime/cranelift at main.bytecodealliance/wasmtime*. URL: https://github.com/bytecodealliance/wasmtime/tree/main/cranelift.

78. M. Wasserman. *Using the netconf protocol over secure shell (ssh)*. Technical report. 2011.

79. M. Wollschlaeger, T. Sauter, and J. Jasperneite. "The future of industrial communication: Automation networks in the era of the internet of things and industry 4.0". *IEEE industrial electronics magazine* 11:1, 2017, pp. 17–27.

80. Z. Xiang, F. Gabriel, E. Urbano, G. T. Nguyen, M. Reisslein, and F. H. Fitzek. "Reducing latency in virtual machines: Enabling tactile Internet for human-machine coworking". *IEEE Journal on Selected Areas in Communications* 37:5, 2019, pp. 1098–1116.

81. I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, et al. "The demikernel datapath os architecture for microsecond-scale datacenter systems". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 195–211.

82. K. Zhang, S. Leng, Y. He, S. Maharjan, and Y. Zhang. "Mobile edge computing and networking for green and low-latency Internet of Things". *IEEE Communications Magazine* 56:5, 2018, pp. 39–45.

83. Q. Zhang, L. Cheng, and R. Boutaba. "Cloud computing: state-of-the-art and research challenges". *Journal of internet services and applications* 1, 2010, pp. 7–18.

84. D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson. "Slim:{OS} Kernel Support for a {Low-Overhead} Container Overlay Network". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019, pp. 331–344.