University of Macedonia
Department of Applied Informatics

Service Orchestration Architectures for the Network Edge

George Papathanail

Submitted in part fulfillment of the requirements for the degree of
Doctor of Philosophy in Applied Informatics of the University of Macedonia, January 2024

Abstract

The rapid evolution of modern networking has led to increased research efforts in the areas of edge computing and network slicing, fueled by the rise of the Internet of Things (IoT) and the advent of 5G (and beyond) networks. The growing popularity of mobile networks has resulted in a higher demand for both conventional cloud-based services and novel cloud services, such as mobile cloud games, remote control services for automobiles, and manufacturing process management services. Nevertheless, traditional cloud computing platforms that rely on large-scale datacenters have encountered difficulties in meeting the increasing computational requirements of emerging mobile and IoT-enabled applications. As a response to these difficulties, the notion of edge computing has emerged as a viable solution. Edge computing involves the positioning of computational resources in proximity to the edge of the network. This strategy provides numerous benefits, such as lower latency, faster response times, and the capability to offload computationally demanding tasks from mobile devices. Nevertheless, the efficient utilization and orchestration of edge computing resources, particularly in situations with limited resources, pose serious challenges that require drastic solutions.

This thesis undertakes a thorough investigation of the capabilities of edge computing and its potential to improve services. In this context, we identify numerous significant obstacles and propose innovative approaches and architectural frameworks in order to lower the barrier for service orchestration at the edge. Initially, we explore the practicality and benefits of offloading resource-intensive tasks to edge servers. We primarily focus on assessing the effect of this approach on latency and throughput, with the ultimate goal of enhancing the user experience. To this end, COSMOS comprises an advanced orchestration framework specifically designed to enable intelligent compute offloading for edge clouds. Our experimentation with COSMOS showcases the versatility and efficacy of the system in improving response times and customer satisfaction.

Network slicing is evolving as an indispensable feature of network infrastructures, as it provides the means for the deployment of next-generation services with performance and reliability guarantees. Based on the trends in evolving service ecosystems, services situated on separate slices inside the same edge data center can enhance their functionality through synergies. In this context, we have identified a novel aspect of network slicing, which we term as Cross-Slice Communication (CSC). To foster CSC, we introduce a new CSC orchestration framework,

namely optiMized Edge Slice OrchestratioN (MESON), which facilitates secure and efficient communication between different slices co-located on the same (edge) cloud infrastructure. This can empower service providers to capitalize the added value of cross-service interactions and, thereby, render their service offerings more appealing to their customers.

Resource allocation at the edge comprises a crucial challenge, given the resource-constrained nature of edge computing infrastructures. In this respect, we examine the issue of resource allocation in environments with limited resources, such as an edge data center. We circumvent this difficulty at the intra-server level, which pertains to the final stage of resource allocation (*i.e.,* the assignment of VNFs to CPU cores). In this respect, we assess the impact of different CPU core allocation strategies on the performance of service chains and identify situations at which resources can be wasted. Our findings can be of great value to resource schedulers, enhancing their ability to allocate resources for VNFs or cloud-native applications.

Furthermore, we advocate for the crucial need of low-latency communication in the scope of converged IoT-cloud environments, where digital twins or other forms of IoT middleware (*e.g.,* Virtual Objects - VOs) reside in edge clouds to augment the operation of IoT devices. Our main aim here is to sustain low latency in the communication between IoT-VO pairs. To this end, we seek to reap the benefits of Time-Sensitive Networking (TSN), by presenting the design and implementation of a TSN platform that couples a TSN bridge, compliant with Time-Aware Shaper (TAS), with Centralized Network Control (CNC). As CNC mainly deals with the computation of TSN schedules, we present a constrained programming formulation for the TSN scheduling problem at hand. In addition, we discuss the interaction between the CNC and the TSN data plane, especially for the configuration of TSN schedules based on the intervals computed by our schedule engine at the CNC. Our evaluation results corroborate the efficacy of the TSN platform in terms of delay/jitter bounds and communication overhead.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. Panagiotis Papadimitriou, for his unwavering support, invaluable guidance, and continuous encouragement throughout my doctoral journey. His expertise, dedication, and mentorship have been instrumental in shaping my research and personal growth.

I am also deeply thankful to the exam committee members, Prof. Eleftherios Mamatas and Prof. Sofia Petridou, for their insightful feedback, constructive criticism, and valuable contributions to my work. Also, I want to thank Prof. Illias Sakellariou for his useful guidance and collaboration.

I appreciate the stimulating conversations, shared insights, and helpful feedback from other researchers, especially Angelos and Makis from the Netcloud team, which have made my study better and helped me improve my ideas. And finally, I would like to thank my friend Stelios Hadjidimitriou, who has encouraged me to finish this trip.

I want to express my deepest gratitude to my mother and friends for their constant love, support, and encouragement during this difficult but ultimately fulfilling journey.

Dedication

This thesis is dedicated to my mother.

‘”Not everything that can be counted counts, and not everything that counts can be counted.” ’

Albert Einstein

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Challenges

The proliferation of Internet-connected devices poses significant stress on conventional cloud computing, due to the escalating volume of data generated by these devices. In the era of 5G networks, Internet-of-Things (IoT) plays a crucial role in driving this expansion. In essence, IoT encompasses an extensive network of sensors, intelligent gadgets, and other interconnected devices that generate data requiring processing and storage. IoT connections are projected to reach 35 billion by 2028, including machinery, sensors, wearables, connected cars, point-of-sale terminals, and consumer gadgets [4].

The communication of these devices encompasses both cellular and low-power unlicensed technologies. It is projected that by 2028, broadband IoT will account for 60 percent of cellular IoT connections. In the conventional cloud computing approach, data is transferred to remote cloud data centers for processing and storage. This can lead to network congestion, latency, and security vulnerabilities, especially with the proliferation of IoT devices. The emergence of 5G (and beyond) networks opens up opportunities to circumvent this difficulty by facilitating expedited and dependable communication among devices, thereby, meeting the requirements of vertical industries, such as healthcare, transportation, and smart cities. Nevertheless, the substantial volume of data generated by these devices poses difficulties with respect to data

processing, storage, and transmission.

The ever-increasing popularity of mobile networks has led to an increased demand for both traditional cloud-based services, such as video streaming, social networking, etc., as well as for emerging cloud services, such as mobile cloud games, remote control services for vehicles, and manufacturing process management services. Conventional cloud computing platforms that depend on datacenters have faced difficulties in supporting the increasing computational demands of mobile and IoT-enabled applications. These platforms have increased their server capabilities to enhance Quality of Service (QoS). However, there is a need for new cloud computing architectures in order to better accommodate the growing demands of mobile services.

Edge computing has been seen as a viable method for addressing the challenges of cloud-based services in mobile networks. This essentially entails the handling of data near the boundaries of the network, in closer proximity to its origin, rather than moving its computation to distant data centers [5]. Several studies confirm that edge computing plays a crucial role in enabling various future technologies, including IoT, augmented reality, and vehicle-to-vehicle communications, in the context of 5G and beyond. This is achieved by establishing a connection between cloud computing facilities, services, and end users [6]. Edge computing can better support for delay-sensitive applications by providing low latency, mobility, and location-awareness.

In this context, this thesis proposes a set of architectures and mechanisms for the orchestration of network services at the network edge, facilitating the execution of processing workloads on edge cloud servers, while alleviating mobile and IoT devices from this burden. These advancements can effectively pave the way for a convergence between IoT and edge computing infrastructures, where edge clouds can support the execution of IoT digital twins or other forms of IoT middleware.

In addition, the thesis reaps the benefits of network slicing by exploring the domain of cross-slice communication (CSC), which can empower synergies among services co-located on the same edge cloud infrastructures. This comprises an essential requirement of evolving service ecosystems, where a service can enhance the experience offered to its users through the consumption of other (potentially co-located) services. One crucial challenge entailed here is how

to foster such CSC in a secure and controlled manner, while adhering to CSC policies that can be mandated by service providers.

The placement and deployment of processing workloads on edge computing facilities is of paramount importance, given that edge clouds are commonly resource-constrained environments where resource allocation should be handled with particular care. In this respect, we stress on the need for fine-grained resource allocation in Network Function Virtualization (NFV) edge infrastructures. This essentially boils down to the assignment of processing workloads not merely onto servers but onto specific CPU cores. As such, the CPU core is deemed as the minimum resource allocation unit, as opposed to the majority of Virtualized Network Function (VNF) placement studies that seek to optimize the placement of VNFs onto servers. Our study on this aspect of the VNF placement problem sheds light into the impact of inter-core packet transfers and the co-existence of homogeneous and heterogeneous workloads on the same core, which could be of great value for resource schedulers on virtualized servers.

Next-generation cloud applications will be hyper-distributed and will need to be provisioned over resources that span IoT devices, IoT gateways, and compute nodes from edge computing infrastructures. The deployment and orchestration of such services benefits from types of IoT middleware, known as Virtual Objects (VOs). However, in order to reap the benefits of VOs, low-latency communication is required between the IoT devices and their associated VOs. To this end, we utilize Time-Sensitive Networking (TSN), which is paving the way for the timely delivery of delay-sensitive traffic. More specifically, we propose an integrated TSN switch architecture that couples a Time-Aware Shaper (TAS) based TSN bridge with a Centralized Network Controller (CNC). One salient feature of this TSN platform is a TAS-compliant scheduler, which is capable of computing TSN schedules on short timescales using constrained programming. This can facilitate the adaptation of traffic scheduling to dynamic conditions, where the association of IoT devices and their virtual counterparts is subject to change.

1.2 Contributions

The thesis proposes orchestration frameworks tailored to the needs of applications whose provisioning spans IoT/mobile devices and edge computing infrastructures. More specifically, the main contributions of the thesis are the following:

- **Orchestration Framework for Efficient Computation Offloading.** Considering the ever-increasing number of IoT and mobile devices, computation offloading is emerging as a cutting-edge and significant research area with enormous potential and practical applications. In this respect, we present the architecture design and experimental evaluation of an orchestration framework for smart computation offloading from IoT or mobile devices to edge cloud servers. The proposed orchestration platform, namely COSMOS, encompasses control-plane components for workload prediction, load balancing, and admission control. The efficiency of COSMOS is experimentally assessed in relation to an object identification application that is offloaded from mobile devices to edge cloud servers.

- **Optimized Cross-Slice Communication.** In the dawn of the 6G era, evolving service ecosystems raise the need for cross-service interactions. Existing resource/service orchestration frameworks are oblivious to such cross-service interactions, since they tend to orchestrate services independent to each other. In this respect, we stress on the need for optimized cross-service communication, and more specifically, for optimized cross-slice communication (CSC). To this end, we present a CSC orchestration platform, namely MESON, which fulfills all main CSC requirements, such as the discovery of CSC-enabled services, the selection of the most suitable (edge) computing infrastructure, and also the establishment and configuration of CSC in a secure and policy-compliant manner.

- **Fine-Grained Resource Allocation for NFV Infrastructures.** The exploration towards efficient resource/service orchestration extends to the realm of NFV infrastructures, where different resource allocation strategies are investigated in order to improve the performance of VNFs. The primary focus here is on intra-server resource allocation,

an aspect often neglected by related studies. In particular, we study the implications of CPU cache hierarchy, memory locality, and varying NFV resource profiles on packet forwarding performance, and demonstrate that the assignment of processing workloads onto the CPU cores of a virtualized server is not a trivial matter. Our findings are crucial for the minimization of resource wastage in resource-constrained infrastructures, such as edge clouds.

- **TSN Platform and TAS scheduling model.** We present a TSN architectural framework that can sustain bounded latency in the communication between IoT/mobile devices and compute nodes on edge clouds that may host application components or VOs that augment the operation of IoT nodes. To this end, we introduce a TSN scheduler that complies with TAS standard. In this respect, we present a constraint programming formulation for the TSN scheduling problem at hand. The proposed TSN scheduler achieves the effective and accurate computation of schedule periods at short time-scales. The TSN platfrom couples the TSN schedule engine with a TAPRIO switch datapath can be configured to prioritize certain classes of traffic based on the schedules computed by the schedule engine. These TSN platform components (*i.e.,* TSN switch datapath and the TSN scheduler) are coupled using NETCONF, which enables the population of computed schedules into the switch in the form of GCL configurations.

1.3 Outline

The remainder of the thesis consists of a background chapter, followed by four technical chapters and a chapter with the concluding remarks. In the following, we discuss the content of each chapter.

Chapter 2 is a background section that provides an overview of the fundamental technologies and concepts that are employed throughout the thesis. Initially, we introduce the notion of Network Softwarization and its crucial role in the evolution of networks. In the following, we delve into Edge Computing and stress on the benefits stemming from moving the computation

towards the network edge. Subsequently, we clarify the underlying principles and structural framework of Network Function Virtualization (NFV), accompanied by a brief description of Software-Defined Networking (SDN). Edge computing plays a crucial role in facilitating the deployment of Network Slicing. In addition, we introduce the concept of Time-Sensitive Networking (TSN), emphasizing its significant importance for next-generation network applications that tend to exhibit stringent latency, and/or reliability requirements.

Chapter 3 presents the architecture and the experimental evaluation of an orchestration framework tailored to the offloading of computation from IoT and mobile devices to edge clouds. Our framework, namely COSMOS, ingeniously integrates edge computing with cutting-edge technologies, such as object identification and NFV. The evaluation carried out on a substantial experimental platform yields promising results. We also demonstrate that mobile computational offloading comprises a viable solution for alleviating the burden of computational-intensive tasks from client devices.

In Chapter 4, we introduce a new aspect of network slicing, *i.e.,* optimized cross-slice communication (CSC). We argue that multi-tenancy and service co-location open up unique opportunities for B2B interactions, cross-service communications and service composition, especially in the domain of edge computing and location-based services. In this context, we present optiMized Edge Slice OrchestratioN (MESON), a MANO-based architecture for optimized CSC in edge clouds. We elaborate on the functionality of each MESON component and explain the workflows for the establishment of CSC through the MESON architecture. Based on a proof-of-concept implementation of MESON, we present evaluation results from various performance and scalability tests, and further uncover significant gains from the deployment of optimized CSC as promoted by MESON.

Chapter 5 advocates for fine-grained resource allocation in NFV infrastructures. The problem of VNF placement commonly entails the selection of the most appropriate server within a single or among multiple Points-of-Presence (PoPs). Nevertheless, CPU cache hierarchy and memory locality in NUMA multi-core servers along with the diversity in NFV resource profiles introduce significant challenges in terms of intra-server resource allocation; a problem that is

often overlooked. As such, we stress on the need for fine-grained resource allocation in NFV infrastructures, and, to this end, we study various aspects of CPU allocation for VNF chains. We deem this intra-server resource allocation problem as complementary to the large body of literature that seeks to optimize VNF placement onto virtualized infrastructures.

In Chapter 6, we stress on the need for bounded latency for applications that are deployed at the edge of the network, since they often exhibit strict requirements in terms of latency and jitter. To this end, we present the design and implementation of a Time-Sensitive Networking (TSN) platform that couples a Time-Aware Shaper (TAS) based TSN bridge with a control plane architecture, which is aligned with the notion of Centralized Network Controller (CNC). The NETCONF management protocol is used as a glue between the TSN data and control plane, facilitating the population of Gate Control List (GLC), which designate the periods over which packets can be transmitted from each queue of an egress switch port. These TSN schedules are computed at the control plane using constrained programming. Our experimental results with Mininet corroborate the efficacy of this TSN platform in terms of delay/jitter bounds and communication overhead. Additional simulation results on OMNeT++ indicate the superiority of the TSN scheduler in comparison with static schedules.

Finally, Chapter 7 concludes the thesis by laying out the main concluding remarks and discussing the practical applications of the architectural frameworks proposed in the context of this thesis. In addition, this chapter provides directions for future work.

1.4 Publications

Technical parts of this thesis are based on the following papers that have already been published or are under review.

1. **Papathanail, G.**, Fotoglou, I., Demertzis, C., Pentelas, A., Sgouromitis, K., Papadimitriou, P., and Papavassiliou, S. (2020, April). COSMOS: An orchestration framework for smart computation offloading in edge clouds. In NOMS 2020-2020 IEEE/IFIP Network

Operations and Management Symposium (pp. 1-6). IEEE.

2. Spatharakis, D., Dimolitsas, I., Dechouniotis, D., **Papathanail, G.**, Fotoglou, I., Papadimitriou, P.,and Papavassiliou, S. (2020). A scalable edge computing architecture enabling smart offloading for location-based services. Pervasive and Mobile Computing, 67, 101217.

3. **Papathanail, G.**, Pentelas, A., Fotoglou, I., Papadimitriou, P., Katsaros, K. V., Theodorou, V., and Papavassiliou, S. (2020). Meson: Optimized cross-slice communication for edge computing. IEEE Communications Magazine, 58(10), 23-28.

4. Fotoglou, I., **Papathanail, G.**, Pentelas, A., Papadimitriou, P., Theodorou, V., Dechouniotis, D., and Papavassiliou, S. (2020, June). Towards cross-slice communication for enhanced service delivery at the network edge. In 2020 6th IEEE Conference on Network Softwarization (NetSoft) (pp. 22-28). IEEE.

5. **Papathanail, G.**, Pentelas, A., and Papadimitriou, P. (2021, December). Towards Fine-grained Resource Allocation in NFV Infrastructures. In 2021 IEEE Global Communications Conference (GLOBECOM) (pp. 1-6). IEEE.

6. Laskaratos, D., Dimolitsas, I., **Papathanail, G.**, Xezonaki, M. E., Pentelas, A., Theodorou, V., and Papavassiliou, S. (2022). MESON: A Platform for Optimized Cross-Slice Communication on Edge Computing Infrastructures. IEEE Access, 10, 49322-49336.

7. **Papathanail, G.**, Dimolitsas, I., Fotoglou, I., Dechouniotis, D., Papavassiliou, S., and Papadimitriou, P. (2022, June). Towards Secure and Optimized Cross-Slice Communication Establishment. In 2022 IEEE 8th International Conference on Network Softwarization (NetSoft) (pp. 127-132). IEEE.

8. **Papathanail, G.**, Mamatas, L., and Papadimitriou, P. (2023, June). Towards the Integration of TAPRIO-based Scheduling with Centralized TSN Control. In 2023 IFIP Networking Conference (IFIP Networking) (pp. 1-6). IEEE.

9. **Papathanail, G**, Sakellariou, I., Mamatas, L., and Papadimitriou, P. (Accepted). Dynamic Schedule Computation for Time-Aware Shaper in Converged IoT-Cloud Environments. ICIN 2024, IEEE

Chapter 2

Background

In the dynamic and ever-changing realm of networking, a significant change in perspective has emerged, introducing revolutionary ideas that are fundamentally altering the fundamental structure, functionalities, and potentialities of networking. The aforementioned innovative ideas, such as softwarization, Network Function Virtualization (NFV), Software-Defined Networking (SDN), Edge Computing, Network Slicing, and Time-Sensitive Networking (TSN), are collectively reshaping the operational, interactive, and adaptive aspects of networks within a progressively interconnected global landscape.

At the heart of this evolution lies the concept of **softwarization**, a departure from the conventional hardware-centric approach to networking. This paradigm shift represents a pivotal moment where the lines between hardware and software blur, enabling networks to transcend the limitations imposed by physical infrastructure. Through this, networks achieve newfound agility and adaptability, becoming dynamic entities that can evolve in response to changing demands. When utilized in conjunction with Network Function Virtualization (NFV), the process of softwarization enables networks to rapidly adapt, resulting in enhanced resource usage and decreased operating expenses related to hardware maintenance and deployment.

NFV, standing as a cornerstone in this transformative journey, embodies the notion of abstraction and virtualization. This concept liberates network functions from their traditional dependence on proprietary hardware, enabling them to be implemented and orchestrated as

flexible software instances. This decoupling of network functions from specific hardware platforms facilitates efficient scalability, simplified management, and the potential for rapid innovation. The synergy between softwarization and NFV becomes particularly pronounced in their shared pursuit of freeing networks from the constraints of hardware, fostering an environment where functions can be dynamically deployed, adjusted, and scaled to meet the ever-evolving demands of applications and services.

Edge computing serves as a vital orchestration, introducing a level of immediacy and proximity previously unrealized in traditional network architectures. By positioning computation and data storage closer to the point of data generation, edge computing minimizes latency and enhances real-time interactions. When aligned with the tenets of NFV, edge computing further accelerates the responsiveness of networks by enabling the efficient deployment of virtualized network functions to the edge. This powerful collaboration drives an enhanced user experience by ensuring that data processing occurs in close proximity to users and devices, enabling seamless interactions and significantly reducing the latency that can hinder the delivery of critical services.

One notable development of comparable importance is the rise of Network Slicing. The concept of **Network Slicing** exemplifies the necessary flexibility needed to accommodate a wide range of use cases and applications. The concept of network slicing involves the division of the network infrastructure into separate virtual segments, each designed to meet specific requirements. The utilization of this slicing mechanism enables networks to effectively accommodate multiple applications simultaneously, each possessing unique performance attributes, security prerequisites, and resource demands. The incorporation of network slicing alongside NFV and other revolutionary concepts highlights the adaptability of contemporary networking. This integration allows for the efficient allocation of network functions to specific slices, enabling the optimization of their performance according to the distinct requirements of each application.

Time-sensitive networking (TSN) plays a crucial role in facilitating the synchronization of many innovative components. In a context where accurate timing is crucial for uninterrupted data transmission, TSN implements methods that guarantee the arrival of data at the

appropriate moment. The incorporation of Time-Sensitive Networking (TSN) into the wider networking architecture ensures the accurate coordination of data streams, which is essential for applications that necessitate prompt communication and real-time engagements. When integrated with the principles of Software-Defined Networking (SDN), Time-Sensitive Networking (TSN) facilitates the dynamic coordination of network resources, ensuring that data streams with time-sensitive requirements are efficiently transmitted via the network, minimizing any potential delays.

2.1 Network Softwarization

Network softwarization is a paradigm within the field of networking that involves the conversion of conventional network infrastructure into a software-driven architecture that is characterized by enhanced flexibility and dynamism. This approach separates network services from the physical infrastructure, enabling their virtualization and deployment as software instances. Key enablers for Network Softwarization are the: NFV, SDN, network slicing, and edge computing.

There are numerous advantages associated with network softwarization:

1. **Agility and Flexibility:** Traditional networks often struggled to keep pace with the agility required to deploy new services and adapt to changing conditions. Network softwarization addresses this by enabling the dynamic configuration and reconfiguration of network elements. SDN, for instance, centralizes network control, allowing for swift adjustments and policy changes across the network.

2. **Resource Efficiency:** NFV allows for the consolidation of multiple network functions onto commodity hardware, resulting in efficient resource utilization. This minimizes the need for specialized hardware devices and reduces power consumption and operational costs.

3. **Rapid Service Provisioning:** The virtualized nature of network functions in NFV significantly accelerates service deployment. New services can be spun up as virtual

instances, leading to faster time-to-market and improved service delivery.

4. **Elasticity:** With SDN and NFV, network resources and functions can be scaled up or down based on demand. This elasticity ensures optimal resource utilization and responsiveness to traffic spikes without over-provisioning.

5. **Reduce Capital Expenditure:** By relying on software-driven solutions and virtualized functions, organizations can avoid the upfront costs associated with purchasing and maintaining specialized hardware appliances.

Nevertheless, the process of network softwarization causes several problems. A significant area of concern pertains to security, given that the virtualized environment presents novel attack paths and possible vulnerabilities. It is imperative to implement appropriate security protocols in order to protect both the virtualized network operations and the underlying infrastructure. The utilization of techniques such as micro-segmentation and end-to-end encryption plays a vital role in upholding the integrity and secrecy of data that traverses the software-defined network. Moreover, the intricacy associated with the administration of a software-based network may escalate, necessitating the utilization of sophisticated orchestration and management tools in order to guarantee optimal functionality and effective troubleshooting. Automation is a crucial process in various tasks such as service provisioning, load balancing, and fault detection. In essence, automation seeks to alleviate the workload of network administrators and reduce the occurrence of human errors. The adoption of this paradigm shift also entails a learning curve, necessitating network engineers to acquire additional skills and knowledge in areas such as software development, orchestration frameworks, and software-defined networking (SDN) ideas.

Additionally, the migration from traditional legacy systems to completely software-defined networks could result in substantial initial expenses, encompassing both hardware enhancements and the need for people retraining. Organizations are required to do a meticulous assessment of the prospective return on investment (ROI) in the long run, taking into consideration the reduction in operational expenses, enhanced flexibility, and enhanced service provision. The

use of open standards and interoperability is crucial in order to prevent vendor lock-in and foster a robust ecosystem of software-defined network components that are interoperable with one another. The role of standardization initiatives is crucial in facilitating the seamless integration of diverse software-defined components within the evolving industry. This, in turn, contributes to the establishment of a more cohesive and efficient network environment. Notwithstanding these problems, the significant benefits of network softwarization render it a crucial trend in contemporary networking, offering the potential for revolutionary alterations in the design, operation, and user experience of networks. The continuous research and innovation in this domain are propelling the advancement of original approaches to tackle obstacles, rendering network softwarization a captivating and dynamic sector with the capacity to revolutionize the complete networking terrain.

2.2 Edge Computing

Edge computing is a distributed computing paradigm that brings computation and data storage closer to the location where it is needed, which is often referred to as the "edge" of the network and aims to enable low-latency, high-bandwidth and high-quality service delivery for emerging applications at the network edge [7, 6]. In traditional cloud computing, data and computation are primarily handled in centralized data centers located far away from the end-users or devices generating the data. Edge computing, on the other hand, shifts some of this processing closer to the source of data generation, reducing latency and improving overall performance [8].

The adoption of Edge Computing is driven by the increasing demand for emerging applications that require real-time processing and low latency, such as augmented reality, virtual reality, autonomous vehicles, and industrial automation. Edge Computing provides a distributed infrastructure that supports these applications by enabling edge devices to offload computation-intensive tasks to nearby edge servers, reducing latency and improving application performance. This approach also provides several other benefits, such as improved network scalability, reliability, and security, among others [9].

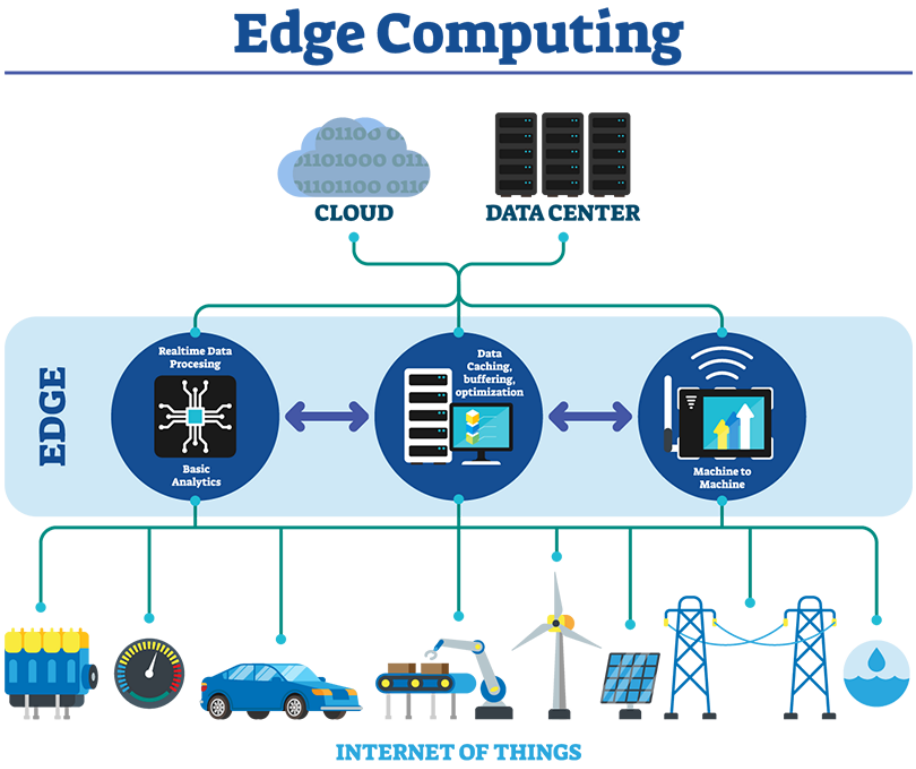


Figure 2.1: Edge Computing Real Life Use Cases[1]

2.2.1 Key Characteristics and Advantages

The main idea behind edge computing is to process data locally, at or near the point of origin, rather than sending all data to a remote data center for processing. This is particularly useful in scenarios where real-time processing is critical, as it reduces the time it takes to transmit data back and forth between devices and centralized servers [10]. Some key characteristics and benefits of edge computing include:

- **Low Latency:** Since data does not need to travel as far, the processing time is reduced, leading to lower latency and faster response times [11]. This is crucial for applications that require real-time or near-real-time interactions.
- **Bandwidth Efficiency:** Edge computing reduces the need to transmit large amounts of data to centralized data centers, which can help in conserving network bandwidth and lowering costs.

[1]Source: `https://innovationatwork.ieee.org/real-life-edge-computing-use-cases/`

- **Privacy and Security:** Processing sensitive data locally can enhance privacy and security by minimizing the exposure of data to external networks. This is especially important for applications dealing with personal or confidential information.

- **Offline Capabilities:** Edge computing enables devices to continue processing data and performing tasks even when they are disconnected from the central network or the Internet. This can be valuable in remote or intermittently connected environments.

- **Scalability:** Edge computing can distribute processing across multiple devices, helping distribute the computational load and scale efficiently.

Edge computing has found traction across a wide range of vertical sectors, such as industrial automation, healthcare, autonomous vehicles, retail, and more [12]. As devices and sensors become more intelligent and generate vast amounts of data, the need for localized and efficient processing has grown, leading to the increasing adoption of edge computing solutions.

2.3 Network Function Virtualization (NFV)

Network Function Virtualization (NFV) [13, 14, 15] represents a significant departure from traditional network architectures by enabling the consolidation of software-based Network Functions (NFs), such as firewalls, proxies, NATs, load balancers, *etc*, on commodity servers, switches, and storage. By running NFs on virtualized infrastructures, NFV replaces the need for specialized, hardware-based middleboxes, affording increased flexibility, programmability, and scalability. This allows for the rapid deployment of new services and features and supports a more dynamic and responsive network environment that can better meet the changing needs of users.

NFV offers a range of benefits for network operators, service providers, and end-users. For example, NFV allows for the migration, instantiation, and cloning of NFs across different locations within the network, without requiring new hardware deployments. This provides a

level of flexibility and agility that was previously unattainable and enables the rapid scaling of network services to meet changing demands.

Another key advantage of NFV is its potential for increased innovation and collaboration within the industry. By consolidating NFs on virtualized infrastructures, NFV allows for the participation of a wider range of players in the development of network services. This includes small businesses, academia, and other organizations that may not have had the resources or expertise to participate in the past. This opens up new opportunities for collaboration and innovation and supports the development of new, customized network services tailored to clients' geographic locations and preferences.

Furthermore, NFV enables the deployment of new service models such as Network Function as a Service (NFaaS) [16]. With NFaaS, network operators can rent their compute resources to host clients' NFs in a pay-per-use fashion, leading to significant cost savings and allowing for dynamic, on-demand-based resource provisioning. This approach also makes it easier for service providers to offer a wider range of network services to their customers, as they can easily scale up or down as needed.

2.3.0.1 ETSI-NFV Framework

To implement Network Function Virtualization (NVF) paradigm, the European Telecommunications Standards Institute (ETSI) NFV has proposed a hierarchical Management and orchestration Architecture (MANO) which is illustrated in Fig.2.2. The MANO architecture is comprised of three main architectural components: i) the **orchestration layer** ii) the VNF Manager Layer, and iii) the Virtual infrastructure (VIM) layer.

The NFV Orchestrator (NFVO). The NFVO layer is an essential component that provides a unified and cohesive view of the underlying network infrastructure, services, and resources. The orchestration layer is composed of two distinct layers - service orchestration and resource orchestration - that work together to control the integration of new network services and Virtualized Network Functions (VNFs) into a virtual framework.

[2]Source: `https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/`

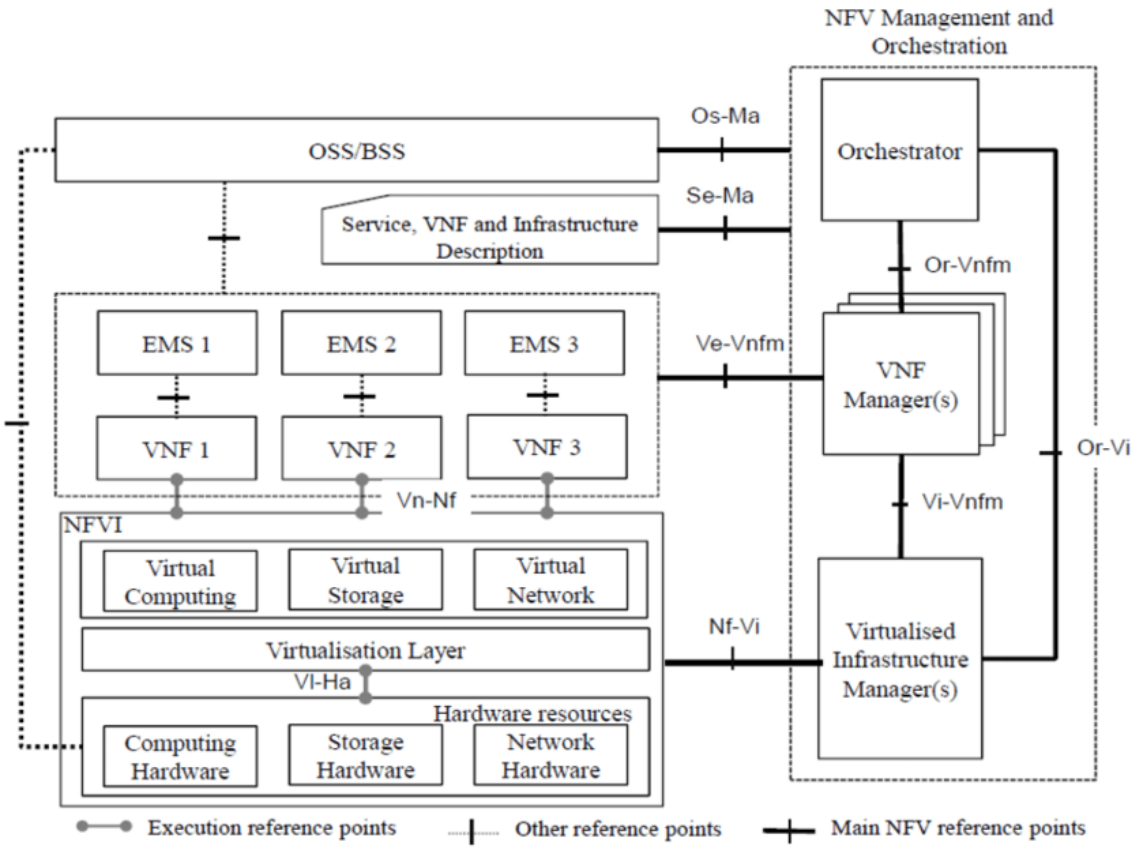


Figure 2.2: ETSI NFV-MANO Framework[2]

- **Service orchestration** is responsible for managing the end-to-end lifecycle of network services, including the deployment, configuration, and optimization of VNFs to meet specific service requirements. It interacts closely with the Virtual Network Function Manager (VNFM) component to ensure that VNFs are deployed in the right order and sequence and that the service chain is properly configured and optimized for performance.

- **Resource orchestration**, on the other hand, is responsible for managing the underlying infrastructure resources required to support network services and VNFs. This includes managing the allocation of compute, storage, and network resources, as well as monitoring and maintaining the performance and availability of these resources. Resource orchestration also works closely with the Virtualized Infrastructure Manager (VIM) to ensure that VNFs are running on virtualized infrastructure that meets their performance requirements.

In addition to these critical functions, NFV orchestrators are also responsible for validating

and authorizing NFV Infrastructure (NFVI) resource requests. This involves verifying that the requested resources are available and properly configured, and ensuring that the request is coming from an authorized entity. By performing these important functions, NFV orchestrators ensure that the NFV infrastructure is running optimally and that all resources are being used efficiently.

VNF Manager. The VNF manager is responsible for overseeing the entire lifecycle of VNF instances, including instantiation, configuration, scaling, migration, and termination. The VNF manager interacts closely with the NFV Infrastructure (NFVI) and Network Function Virtualization Infrastructure (NFVI) Manager to ensure that VNF instances are running on virtualized infrastructure that meets their performance requirements. The VNF manager also works in close collaboration with the NFV Orchestrator (NFVO) to ensure that VNF instances are properly orchestrated and managed throughout their lifecycle. This involves managing the allocation of resources to VNF instances, configuring network connectivity, and ensuring that VNF instances are properly monitored and maintained.

Virtual Infrastructure Manager (VIM). Virtualized Infrastructure Managers (VIMs) play a critical role in controlling and managing the NFV infrastructure. This infrastructure encompasses a range of resources, including compute, storage, and network resources, that are required to support the deployment and operation of Virtualized Network Functions (VNFs).

VIMs are responsible for managing the virtualization layer of the NFV infrastructure, including the management and orchestration of virtual machines and other virtual resources. This involves managing the allocation and assignment of compute, storage, and network resources to VNFs, as well as ensuring that these resources are properly configured and optimized for performance.

There are multiple implementations of NFVOs, the most popular among them are the Open Source Mano (OSM) [17], the Openbaton [18], and the OpenMano [19] while two popular VIMs are the Openstack [20] and the Openvim [21].

2.4 Software-Defined Networking (SDN)

Software-Defined Networking (SDN) represents a novel approach to network architecture that decouples the network control plane from the data plane. This shift moves network state and intelligence to a programmable, logically centralized controller, and provides an abstract view of the network infrastructure to the applications [22, 23]. By doing so, SDN provides a range of compelling benefits to the network, including increased flexibility, programmability, and agility.

One of the key benefits of SDN is its ability to enable network operators and administrators to program the network control logic in real-time. This leads to a shorter innovation cycle and a faster, more dynamic response to users' requirements and business needs. By separating the control plane from the data plane, SDN allows network operators to dynamically reconfigure the network, adjust routing policies, and optimize network performance in response to changing demand and traffic patterns.

Moreover, SDN enables the network to be more responsive to user needs by providing a flexible and programmable framework that can be adapted to specific applications and use cases. By providing an abstract view of the network infrastructure, SDN makes it possible for applications to interact with the network in a more meaningful and context-aware way. This leads to improved application performance, reduced latency, and better overall user experience.

In addition, SDN also allows network operators to automate many network management tasks [24], which reduces the risk of human error and frees up valuable IT resources. This results in a more efficient and cost-effective network infrastructure that can be more easily managed and optimized.

The SDN centralized controller enables network-wide visibility, abstracting the network as a single logic switch that applications and services can interact with. This simplifies the process of configuring the network as SDN provides a standard, vendor-independent interface to access network devices. This eliminates the need for a complicated configuration process that targets diverse types of devices and vendors, resulting in more consistent policy enforcement.

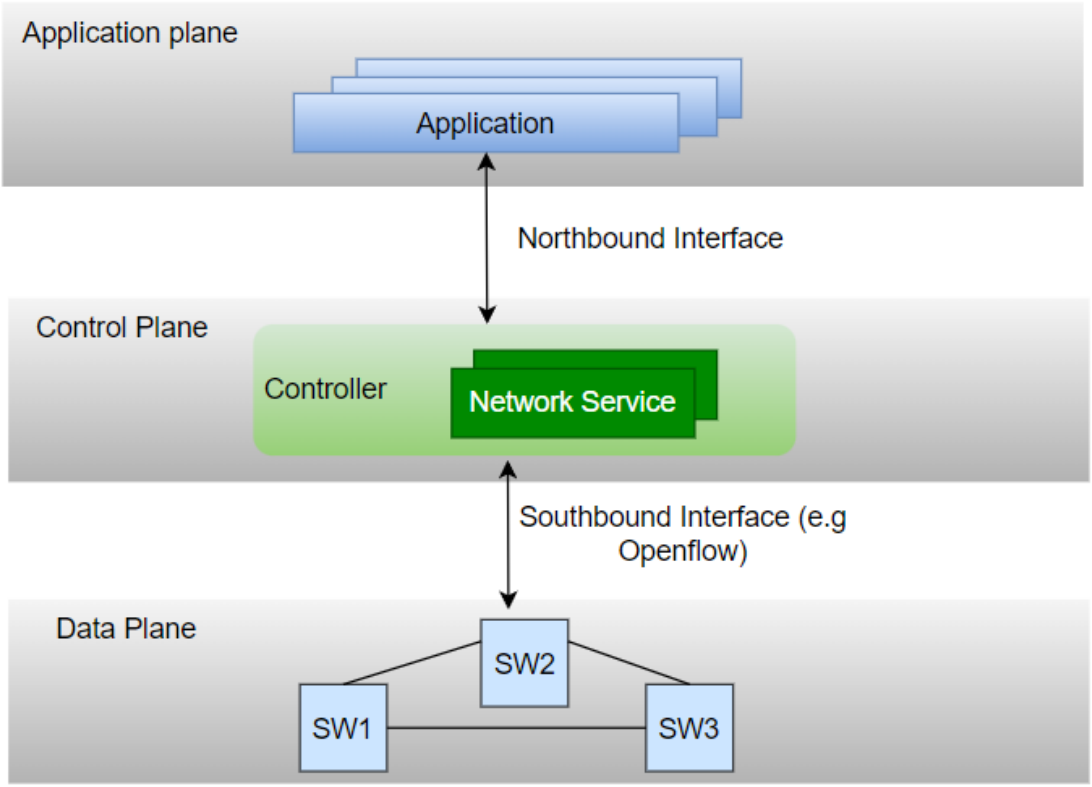


Figure 2.3: SDN-Architecture Overview.

In Figure. 2.3, the SDN architecture is depicted as consisting of three main layers:

Data Plane: The data plane is composed of a collection of simple switches that do not execute complex algorithms or protocols such as routing algorithms, unlike traditional network devices. These switches are considered "dumb" because their primary function is to forward packets based on instructions provided by the control plane.

Control Plane: The control plane is the central component of SDN. It provides a platform, known as the SDN controller, for executing software modules that implement various network control logic, such as routing, load balancing, and access control.

Application Plane: The application plane is responsible for representing the business or user applications that require network services support. For instance, a video application may require routing through a network path with low delay.

In an SDN architecture, the communication between the control plane and data plane layers is facilitated by an SDN-enabled southbound interface. This interface is responsible for critical

tasks such as network configuration, advertising, and flow insertion. OpenFlow is a widely used and popular southbound interface in SDN.

The southbound interface plays a crucial role in SDN, as it enables the control plane to interact with the data plane and execute the network control logic implemented by the software modules. This interface is responsible for the configuration and management of the underlying network devices, such as switches, routers, and access points. By providing a standardized, vendor-independent interface, the southbound interface streamlines the process of managing and configuring network devices, leading to greater consistency in policy enforcement.

OpenFlow is a popular southbound interface that has gained significant traction in the SDN community due to its versatility and widespread adoption. It allows for granular control over the flow of network traffic, enabling network administrators to configure and manage the network in a more precise and efficient manner. The use of OpenFlow as a southbound interface has also been linked to improved network performance, reduced latency, and better scalability, making it a popular choice for SDN deployments in various settings, including data centers, cloud computing environments, and enterprise networks.

2.5 Network-Slicing

With the advent of network function virtualization (NFV) [13] and software-defined networking (SDN) [23], network slicing has emerged as a promising approach for enabling network providers to offer customized network services and lease bundles of computing, storage, and network resources to service providers operating in different commercial domains, or verticals. This paradigm provides a unique opportunity to integrate services more deeply into the network infrastructure (including the edge), facilitating the development of innovative applications that require stringent performance requirements across various domains of social and economic activity, such as automotive, media, e-health, and manufacturing and ensuring the optimal performance, security and QoS.

Moreover, network slicing has the potential to accelerate innovation and enable new business

models, such as pay-as-you-go, on-demand, and subscription-based pricing models, which can significantly improve the overall economics of network services. There have been a number of definitions provided regarding network slicing.

According to Ericcson:[3] *"Network slicing is a powerful virtualization capability and one of the key capabilities that will enable flexibility, as it allows multiple logical networks to be created on top of a common shared physical infrastructure. The greater elasticity brought about by network slicing will help to address the cost, efficiency, and flexibility requirements imposed by future demands."*

Another definition given: *Network slicing allows multiple virtual networks to be created on top of a common shared physical infrastructure.*[4]

However, the deployment of network slicing also entails several challenges, such as the need for efficient slice management, orchestration, and coordination, as well as the need for adequate security, privacy, and trust mechanisms. Furthermore, network slicing requires the integration of various technologies, including NFV, SDN, edge computing, and cloud computing, which may introduce additional complexities and interoperability issues.

2.5.0.1 Network-Slicing in 5G era

In the context of 5G, network slicing has emerged as a novel concept that enables the creation of multiple logical, self-contained networks on a common physical infrastructure platform, thereby allowing for a flexible stakeholder ecosystem that fosters technical and business innovation by integrating physical and/or logical network and cloud resources into a programmable, open, and software-oriented multi-tenant network environment. NGMN (Next Generation Mobile Network) [25] introduced this concept in their report, while 3GPP [26] defined network slicing as a technology that "enables the operator to create networks customized to provide optimized solutions for different market scenarios that demand diverse requirements, in terms of functionality, performance, and isolation". Meanwhile, ITU-T perceives network slicing as Logical

[3]https://www.ericsson.com/en/digital-services/trending/network-slicing
[4]https://5g.co.uk/guides

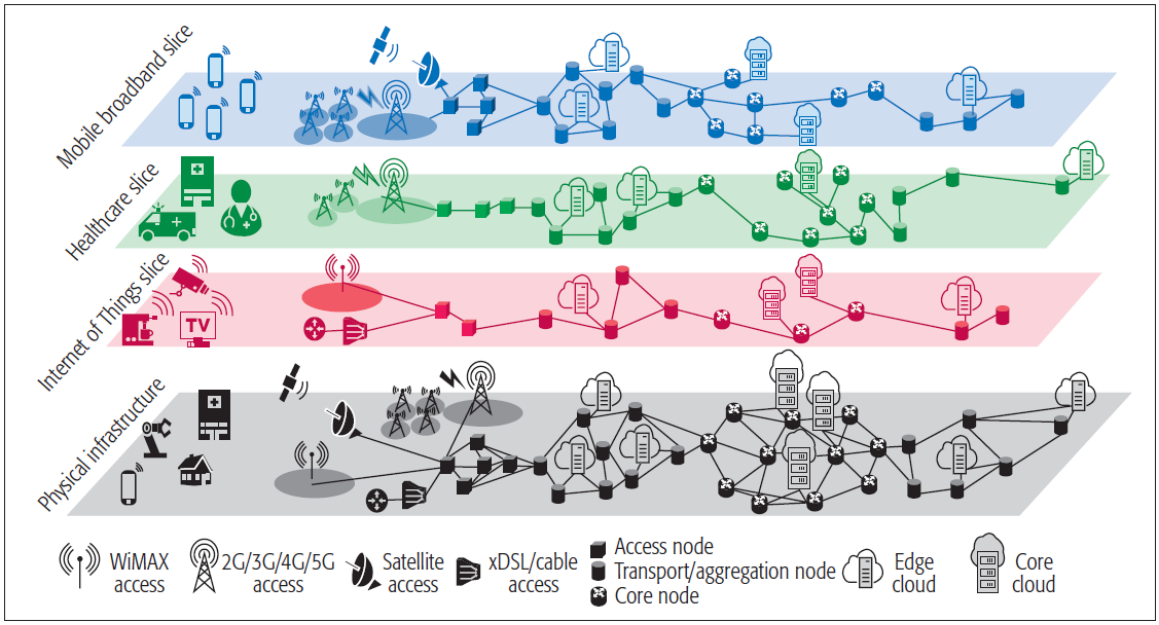


Figure 2.4: Network-Slicing in 5G[2].

Isolated Network Partitions (LINP) composed of multiple virtual resources that are isolated and equipped with a programmable control and data plane [27].

The primary objective of network slicing is to offer a dynamic, flexible, and cost-effective approach to network service delivery, enabling network operators to cater to the diverse and often unique needs of different verticals and use cases, while achieving high levels of resource efficiency and utilization. By leveraging network slicing, service providers can offer customized, fine-grained control over network resources, ensuring optimal performance, security, and QoS for the applications running on each slice. This, in turn, can lead to improved user experience, increased revenue, and new business opportunities for both network operators and service providers.

Despite the potential benefits of network slicing, its deployment poses several challenges, such as the need for efficient slice management and orchestration, the integration of various technologies, including SDN, NFV, and cloud computing, and the development of adequate security, privacy, and trust mechanisms. Moreover, the lack of standardized interfaces, protocols, and architectures may introduce interoperability issues and hinder the adoption of network slicing by different stakeholders.

To address these challenges, extensive research is being carried out in the area of network

slicing, focusing on the development of new techniques, tools, and frameworks for efficient slice management, orchestration, and coordination, as well as the integration of different technologies and the provision of adequate security and privacy mechanisms. Overall, network slicing has the potential to revolutionize the way network services are delivered and consumed, enabling new levels of customization, flexibility, and performance for a wide range of applications and use cases in the 5G era and beyond.

2.5.1 Network-Slicing Principles

According to [28], Network Slicing it is held by seven main principles:

Automation makes it possible to configure network slicing on demand, eliminating the need for fixed contractual agreements and the need for manual intervention in the process. This convenient operation is made possible by signaling-based mechanisms, which enable third parties to place a slice creation request. This request must include, in addition to the conventional SLA, which would reflect the desired capacity, latency, jitter, etc., timing information that takes into consideration the starting and ending time, as well as the duration or periodicity of a network slice.

Isolation is a fundamental property of network slicing that ensures performance guarantees and security (to defend network openness to third parties) for each tenant, even when multiple tenants use network slices for services that have conflicting performance requirements. This is because isolation prevents unauthorized access to the network. Nevertheless, depending on the method of resource separation for explicit usage, isolating data may come at the expense of a reduced multiplexing gain. This may lead to inefficient use of the network's available resources. The idea of isolation encompasses not just the data plane but also the control plane, and the degree to which resources are kept apart is determined by how its implementation is carried out. It is possible to implement isolation in one of three ways: (i) by using a different physical resource; (ii) by separating via virtualization, which means using a shared resource; or (iii) by sharing a resource under the direction of a respective policy that defines the access rights for each tenant.

Customization refers to the process of ensuring that the resources that have been allotted to a certain tenant are utilized effectively in order to meet the corresponding service requirements in the best possible manner. Customization of slices can be accomplished: (i) on the control plane by introducing programmable policies, operations, and protocols; (ii) on the data plane by including service-tailored network functions and data forwarding mechanism; (iii) on the data plane by including service-tailored network functions and data forwarding mechanism; and (iv) through value-added services such as big data and context awareness.

Elasticity is an essential operation that is related to the resource that is allotted to a specific network slice. This is done in order to ensure the desired Service Level Agreement (SLA) in the face of varying radio and network conditions, (ii) the number of users that are being served, or (iii) the geographical serving area as a result of user mobility. Such resource elasticity can be achieved by reshaping the use of the allocated resources by scaling up or down, relocating value-added services and virtual network functions (VNFs), or adjusting the policy that is being applied and re-programming the functionality of certain data plane and control plane elements. Altering the number of resources that were initially allotted can also be a form of elasticity. This can be accomplished by modifying the physical and virtual network functions, such as by adding a different radio access network (RAN) technology or a new VNF, or by increasing the radio and network capacity. Nevertheless, in order to avoid negatively affecting the performance of other slices that share the same resources, this process needs to be preceded by an interslice negotiation.

Programmability permits third parties to manipulate the assigned slice resources, such as networking and cloud resources, using open APIs that expose network capabilities to provide on-demand service-oriented customization and resource elasticity.

End-to-end is a property that is inherently possessed by network slicing, and its purpose is to facilitate the delivery of service all the way from the service providers to the end-user or consumer (s). A property of this kind has two extensions: (i) it extends across different administrative domains, in other words, it is a slice that combines resources that belong to different infrastructure providers; and (ii) it unifies various network layers and heterogeneous

technologies, for example, taking into consideration RAN, core network, transport, and the cloud. Both of these extensions are referred to as extensions of the property's core functionality. For instance, an end-to-end network slicing consolidates a variety of resources, which makes it possible to have an overlaid service layer. This opens up new doors for effective networking and service convergence.

Hierarchical abstraction is a trait of network slicing that has its roots in recursive virtualization. In hierarchical abstraction, the resource abstraction technique is repeated on a hierarchical pattern with each successively higher level, delivering a stronger abstraction with a bigger scope. Hierarchical abstraction is achieved by recursive virtualization. To put it another way, the resources of a network slice that have been allotted to a certain tenant can thereafter be traded, either partially or completely, to yet another third player. This means that the network slice tenant can facilitate the provision of yet another network slice service on top of the one that was already present. For instance, a virtual mobile operator that has acquired a network slice from an infrastructure provider may offer a portion of such resources to a utility provider that makes use of its virtual network in order to form an Internet of Things slice.

2.6 Time-Sensitive Networking (TSN)

2.6.1 Deterministic Networking

Deterministic networking is a type of networking that makes it easy to predict how latency, throughput, and reliability [29] will affect communication. This is needed in many mission-critical and time-sensitive applications, such as industrial automation [30], self-driving cars, and remote surgery, where even small changes in communication performance can have significant effects. Using deterministic networking principles, networks can send data reliably and consistently, making sure that time-sensitive data packets get to their destination within strict parameters. This predictability is made possible by well-defined standards, protocols, and mechanisms that work together to create a strong and efficient networking environment

that meets the strict needs of time-sensitive applications.

2.6.1.1 Time-Sensitive Networking

Time-Sensitive Networking (TSN) constitutes a collection of standards established by the Time-Sensitive Networking task group within the IEEE 802.1 working group. These TSN standards delineate methodologies for transmitting data with high time sensitivity over deterministic Ethernet networks. A significant portion of TSN projects extends the IEEE 802.1Q – Bridges and Bridged Networks standards, which pertain to Virtual LANs and network switches. The primary objective of these extensions is to facilitate data transmission with minimal latency and maximum reliability.

TSN mechanisms hold particular significance in domains such as automotive and industrial control, where real-time Audio/Video Streaming and control streams are employed within converged networks. Various IEEE 802.1 implementations are available, including 802.1Qbv – Time Aware Shaper [31], IEEE 802.1 Qbu Preemption [32], and IEEE 802.1AS Timing and Synchronization [33]. These implementations provide a diverse range of features and functionalities for network communication. For example, 802.1Qbv offers the Time-Aware Shaper (TAS) mechanism to manage latency, while 802.1Qbu introduces the Preemption functionality to interrupt and resume frame transmission. Moreover, 802.1AS concentrates on timing and synchronization within the network. These IEEE 802.1 standards serve as vital components in contemporary networking, supplying essential tools for transmitting data over networks with varying performance requirements.

In Table 2.1, we present a classification of the pertinent IEEE TSN standards. Resource management aspects within TSN are addressed by amendments such as 802.1Qcc and 802.1Qdd. Synchronization in TSN is encompassed by IEEE 802.1AS, with ongoing work being conducted in 802.1AS-Rev. Standards that support delay assurances include Scheduled Traffic (IEEE 802.1Qbv) and Frame Preemption (IEEE 802.3br, IEEE 802.1Qbu). These standards delineate the manner in which TSN-enabled bridges manage frames belonging to specific traffic classes or those with particular priorities. In this thesis, we give more attention to 802.1Qbv standard.

Table 2.1: IEEE TSN Standards Overview [1]

Category	**Standards**
Time Synchronization Providing network wide precise synchronization of the clocks of all entities at Layer 2.	IEEE 802.1AS & IEEE 802.1AS-Rev (Network Timing & Synchronization)
Latency & Jitter Separating traffic into traffic classes and efficiently forwarding & queuing the frames in accordance to these traffic classes.	IEEE 802.1Qav (Credit Based Shaping) IEEE 802.1Qbv (Scheduled Traffic) IEEE 802.3br & IEEE 802.1Qbu (Frame Preemption) IEEE 802.1Qch (Cyclic Queuing) IEEE 802.1Qcr (Asynchronous Traffic Shaping)
Reliability & Redundancy Maintaining network wide integrity by ensuring path redundancy and ingress queue policing.	IEEE 802.1CB (Frame Replication & Elimination) IEEE 802.1Qca (Path Control & Reservation) IEEE 802.1Qci (Per-Stream Filtering)
Resource Management Providing dynamic discovery, configuration and monitoring of network in addition to resource allocation & registration.	IEEE 802.1Qat & IEEE 802.1Qcc (Stream Reservation) IEEE 802.1Qcp (YANG Models) IEEE 802.1CS (Link-Local Reservation)

2.6.1.2 TSN Data Plane

The IEEE 802.1 TSN standards delineate an array of methodologies for facilitating deterministic services over traditional Ethernet. These techniques encompass Scheduled Traffic (IEEE 802.1Qbv), Frame Preemption (IEEE 802.3br, IEEE 802.1Qbu), Asynchronous Traffic Shaping (802.1Qcr), and Cyclic Queuing and Forwarding (802.1Qch). This thesis primarily concentrates on IEEE 802.1Qbv, which introduces the concept of transmission gate operation for each traffic class queue, as depicted in Fig. 2.5. Outgoing frames traversing the egress port of a TSN switch pass through a Traffic Classification block, which is tasked with categorizing different streams into their corresponding traffic classes. This procedure aids in mitigating traffic congestion scenarios that could potentially impact the switches. Subsequently, packets are queued in distinct traffic classes based on the state of the transmission gates, which can be either open or closed, and are regulated by a Gate Control List (GCL). Each output port's GCL is composed of multiple schedule entries. For open gates, the designated traffic is permitted to traverse to the transmission selection block, which grants access to the medium. To ensure the defined behavior of a switch configured with IEEE 802.1Qbv schedules, it is imperative that the clocks of all switches are synchronized.

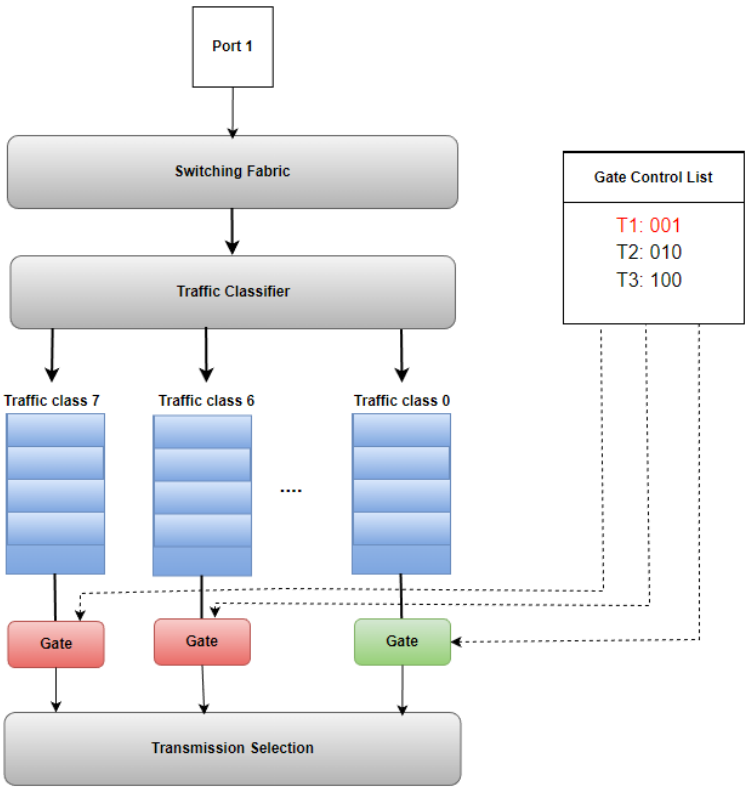


Figure 2.5: Illustration of 802.1Qbv.

This guarantees that all switches adhere to the same cycle base time with their schedules, a responsibility fulfilled by the Precision Time Protocol (PTP). In this context, the IEEE 802.1AS and IEEE 802.1AS-Rev amendments are designed to ensure time synchronization in TSN. Consequently, by employing suitable GCL schedules and time synchronization, it is possible to ascertain that the transmission of high-priority critical traffic remains unhampered by other best-effort traffic flows. This, in turn, guarantees deterministic delay and minimal jitter for the scheduled traffic flows.

2.6.1.3 TSN Control Plane

The TSN network is composed of several key components, including Bridges and Talker-Listener pairs. According to TSN terminology, the Talker and Listener denote the end stations that serve as the source and sink for data streams, while the Bridge refers to the L2 devices responsible for forwarding data streams between the Talker-Listener pairs. The User Network Interface (UNI) is the means by which the end stations interact with the TSN network.

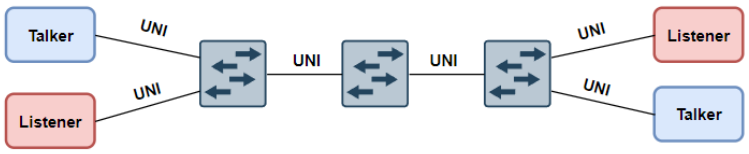


Figure 2.6: Fully distributed TSN control plane.

The TSN control plane model serves as the interface between the Talker-Listener pairs and the TSN network, allowing for the conveyance of stream requirements to the network without requiring knowledge of the network's specific characteristics. The control plane model also enables the configuration of TSN bridges based on topology-specific information and the capabilities of the TSN network, in combination with the stream requirements. This enables deterministic communication between Talker-Listener pairs, ensuring that latency, jitter, and other network-related factors do not negatively impact the delivery of data streams.

To this end, the IEEE 802.1Qcc amendment has proposed three TSN control plane models, each with its own unique characteristics and benefits. These models provide an effective means of managing the TSN network and ensuring the reliable and efficient delivery of data streams between Talker-Listener pairs.

1. **Fully Distributed:** The schematic diagram of a fully distributed control plane model based on the notion of distributed network and user model is depicted in Fig. 2.6. In this model, the user requirements from the end-stations are communicated throughout the entire network topology using a distributed protocol. The UNI is situated between the end-stations and the Bridge to which they are connected in the topology.

   The IEEE 802.1Qdd amendment investigates the protocols and procedures for a Resource Allocation Protocol (RAP) that utilizes Link Registration Protocol (LRP) based underlay transport to provide stream reservation and Quality of Service (QoS) guarantees in the fully distributed case. The RAP protocol operates in a fully distributed fashion, where the network is responsible for allocating resources based on the user requirements without the need for a centralized controller.

   This fully distributed control plane model offers several benefits, including increased scal-

ability and fault tolerance, as well as improved network performance. The ability to distribute control plane functionality across the entire network eliminates the need for a centralized controller, reducing the risk of single points of failure and enabling the network to operate more efficiently. Additionally, the use of a distributed protocol ensures that user requirements are communicated throughout the network, allowing for more effective resource allocation and QoS guarantees.

2. **Hybrid:** The hybrid case of the TSN network control plane model is based on the concept of centralized network control and a distributed user model. It is comprised of a centralized network controller, known as the Centralized Network Configurator (CNC), which possesses complete knowledge of the network topology and the stream specifications. The CNC is responsible for configuring TSN features and performing complex operations such as computation of TSN schedules, control of reservation mechanism and scheduling of streams, Frame Preemption, etc., at the TSN bridges.

   The CNC communicates with the TSN bridges via remote network management protocols such as NETCONF [34], RESTCONF [35], and IETF YANG data models [36]. In a client-server based network management protocol architecture, the TSN bridge serves as the management server, while the CNC serves as the management client.

   In this model, the TSN bridge located at the edge of the network, which is directly connected to an end station, is responsible for communicating the stream requirements to the CNC. This allows the CNC to have complete knowledge of the user requirements and network topology, enabling it to efficiently allocate resources and configure TSN features.

   The hybrid TSN control plane model offers several benefits, including improved network performance, scalability, and ease of management. The centralized network controller simplifies the management of the network by providing a single point of control, while the distributed user model ensures that user requirements are communicated throughout the network. This model also provides the ability to perform complex operations, such as scheduling and reservation control, at the TSN bridges, improving the overall efficiency and performance of the network.

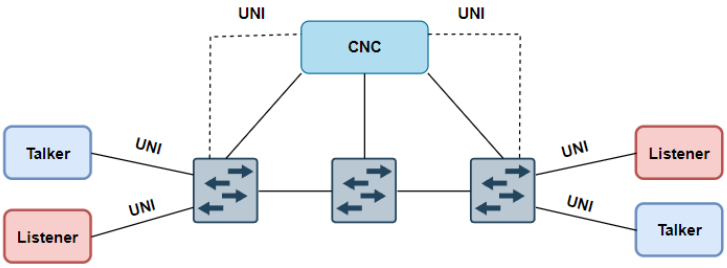


Figure 2.7: Hybrid TSN control plane.

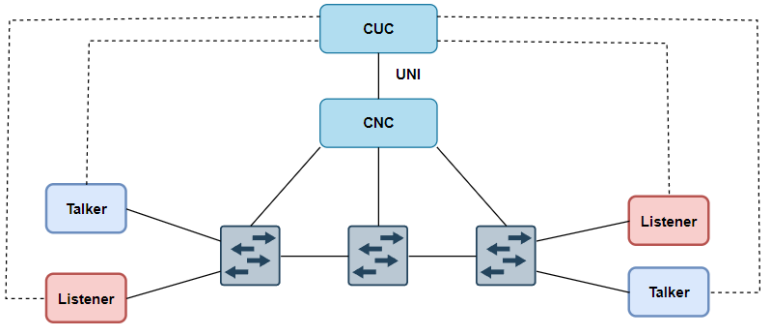


Figure 2.8: Fully centralized TSN control plane.

3. **Fully Centralized:** Figure. 2.8 illustrates the fully centralized TSN control plane model, which proposes a centralized network and user model. This model is comprised of a logical entity known as the Centralized User Configurator (CUC), which communicates with both the CNC and the end devices. The end stations communicate the flow-specific requirements to a centralized CNC. The CUC then requests the CNC for deterministic communication (TSN flows) with specific requirements for those flows. The flow specifications are conveyed from the CUC to the CNC using a User Network Interface (UNI), as described in 802.1Qdj. The CUC is also responsible for the discovery of end-stations, retrieval, and configuration of end-station capabilities. On the other hand, the CNC is responsible for the configuration and control of the TSN switch fabric. This model ensures that the CNC has complete knowledge of the network topology and user requirements, allowing it to allocate resources and configure the TSN switch fabric efficiently.

   The fully centralized TSN control plane model offers several benefits, including improved scalability, centralized control, and efficient resource allocation. The centralized network and user model ensure that user requirements are communicated to the CNC, allowing it to configure the TSN switch fabric accordingly. This model also provides the ability

to perform complex operations, such as scheduling and reservation control, at the CNC, further improving the overall efficiency and performance of the network.

Chapter 3

Computation Offloading for the Edge

3.1 Motivation

The widespread adoption of mobile devices and the increasing demand for mobile services have led to a surge in the amount of data that needs to be processed and transmitted. However, the limited processing capabilities and high power consumption of mobile devices have made it difficult to meet the requirements of modern mobile services. This has created a need for new computing paradigms that can address these challenges.

Edge computing has emerged as a promising solution for addressing the limitations of mobile devices. By moving computation closer to the edge of the network, edge computing provides faster response times and reduced latency, which can improve the performance of mobile services. In addition, edge computing can offload computationally intensive tasks from mobile devices, thereby reducing their power consumption and extending their battery life. By leveraging the rich data sources available at the edge of the network, edge computing can provide tailored services to individual users based on their preferences, location, and other contextual information. This can lead to a better user experience and increased engagement with mobile services.

However, the deployment and management of edge computing resources can be challenging,

especially in resource-constrained environments. To address this challenge, there is a growing need for efficient resource management, application lifecycle management, and performance guarantees in the edge. This requires developing novel approaches that can enable effective orchestration of edge computing resources and services.

Hence, the objective of this chapter is to investigate the capabilities of edge computing and devise novel methodologies for optimizing resource allocation, managing application lifecycles, and enhancing scalability within the edge computing paradigm. Our primary objective is to examine the efficiency and advantages of offloading computationally demanding tasks to edge servers, with a specific focus on evaluating the impact on latency and throughput.

3.2 Contributions

This chapter's primary objective is to introduce COSMOS, an orchestration framework created to facilitate effective smart compute offloading in edge clouds. The proposed framework has been developed with a particular emphasis on enhancing the implementation of a mobile item identification service for customers. However, it is worth noting that this framework is also applicable to other types of services, as it is designed to be versatile and adaptable. In the given context, individuals who are engaged in visiting a particular tourist location employ their mobile devices, which are equipped with cameras, to capture photographs or record brief video clips of a Point of Interest (PoI). This PoI may encompass a noteworthy architectural or sculptural element. The aim of this work is to employ an object recognition service for the purpose of retrieving crucial information.

The findings presented in this chapter demonstrate the effectiveness of shifting computing operations to the edge in order to facilitate object identification services in resource-constrained environments. The proposed architectural design presents a possible option for reducing the computational burden on mobile devices, hence, enhancing response time and the overall user experience.

3.3 Problem Description

The allocation and management of Edge Computing resources can pose difficulties, particularly in settings with limited resources. The effective coordination of Edge Computing resources plays a crucial role in maximizing resource usage and facilitating the provision of high-quality services. The necessity for Edge Computing orchestration emerges due to the subsequent requirements:

Performance Guarantees in a Resource-Constrained Environment. Edge computing, operates at the last part of the network, in closer proximity to the data sources and end-users. The close proximity between entities results in decreased latency and enhanced response times for applications that require immediate attention. Nevertheless, the inherent limitations of edge resources, such as restricted computational capabilities and bandwidth, require meticulous deployment of resources. Resource orchestration is a process that guarantees the allocation of essential resources to vital workloads in order to fulfill performance requirements, such as maintaining low latency and achieving high throughput, even when operating in environments with limited resources.

Efficient Resource Management. Edge devices, such as IoT sensors and edge servers, are deployed in diverse and dynamic environments. To optimize resource utilization, orchestration systems must dynamically allocate and reallocate resources based on changing workloads and network conditions. Efficient resource management enabled by orchestration minimizes wastage and ensures that resources are utilized to their fullest potential, reducing operational costs and environmental impact.

Application Lifecycle Management. In edge computing, applications often need to be deployed and managed across a distributed network of edge devices. Resource orchestration simplifies the deployment and lifecycle management of these applications. It automates tasks such as application provisioning, scaling, and updates, ensuring that applications are always available, up-to-date, and optimized for the available resources.

Scalability. Scalability is a key requirement in edge computing, especially as the number of

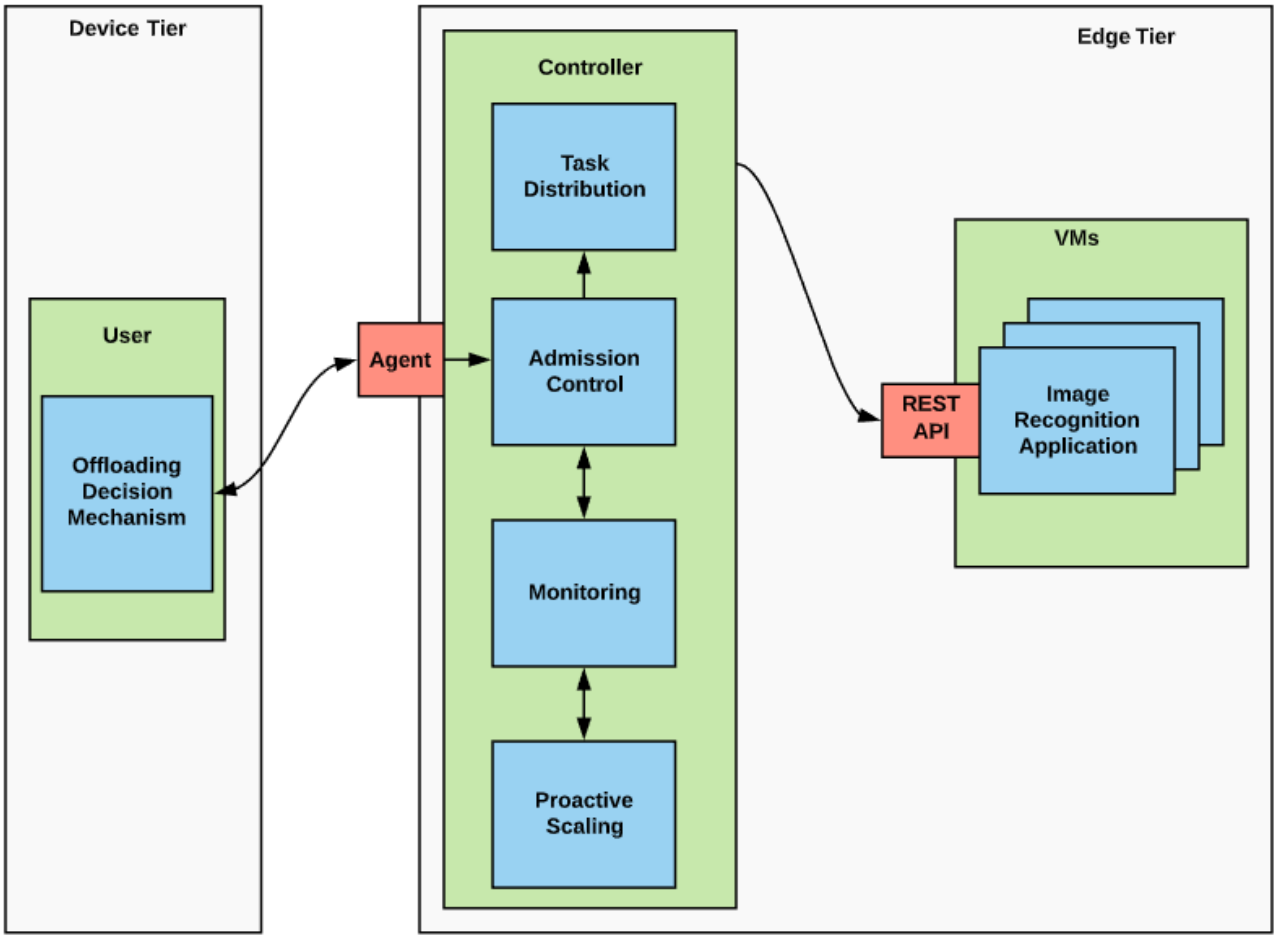


Figure 3.1: COSMOS Architecture.

edge devices and the diversity of applications continue to grow. Resource orchestration enables seamless scaling by intelligently distributing workloads across available resources. It ensures that the system can handle both the increasing number of edge devices and the surges in demand for various applications.

To effectively handle resource orchestration in an Edge Computing environments, we propose an orchestration framework named COSMOS. This framework is designed to address the aforementioned problems and ensure efficient management of resources.

3.4 COSMOS Architecture

The COSMOS framework which is depicted in Fig. 3 has been designed with several objectives in mind, including reducing response time to avoid any negative impact on the user experi-

ence due to the computational offload, admission control of offloading requests based on the residual computational capacity of the edge cloud, and dynamic load balancing of the object identification instances among the allocated Edge Computing MEC servers.

The COSMOS framework is a two-tier architecture based on a centralized controller deployed in the **Edge Tier**, which is responsible for handling requests from mobile users seeking to offload object identification into the edge. This system is designed to handle multiple concurrent requests from various Wi-Fi access points, enabling efficient management of resources. The COSMOS controller is composed of several key components, including i) proactive scaling, ii) task distribution, iii) admission control, and iv) monitoring, which work together to ensure the smooth operation of the system.

At the **Device Tier**, we have portable IoT devices, e.g., Raspberry Pi and mobile phones, equipped with sensors and camera modules that provide location information and media content in order to identify PoIs (e.g., museum exhibits) and retrieve information about them through an image recognition service.

Also at this tier, we implement the **Offloading Decision Mechanisms** which acts as a first-level admission control mechanism. Users are required to send an initial request to the controller to inform their position and current signal strength. Subsequently, the users are notified whether their requests are accepted or rejected, based on a predefined threshold of signal strength correlated with their position. If the request is rejected, the object identification is executed locally on the user's device, reducing the burden on the system. This offloading decision mechanism enables the COSMOS controller to anticipate the number of incoming requests and facilitate the efficient operation of admission control.

Below we analyze the core modules of our architecture:

Admission Control. The primary responsibility of this module is to reject requests that cannot be handled by the deployed objective identification instances implemented using TensorFlow in the edge cloud. This is achieved through a stringent admission control process, which is designed to admit incoming requests based on the level estimated by Kalman filtering.

Any request exceeding this level is rejected, along with requests initiated from user devices with a signal strength below a fixed threshold. The admission control mechanism, thus, plays a critical role in maintaining the QoE for mobile users. Furthermore, the primary objective of the admission control module is to provide high-quality service to mobile users by sustaining a high Quality of Experience (QoE).

To facilitate the admission control process, a database resides within the controller, which maintains monitoring data for the requests and the processing load of the TensorFlow VNF instances. This enables the controller to handle multiple concurrent requests from various Wi-Fi access points and ensures that the resources are utilized efficiently.

Task Distribution. After the admission control process has been executed, the offloaded traffic generated by mobile users must be distributed among the running TensorFlow instances responsible for object identification. To ensure an efficient and effective distribution of the workload, time is quantized into discrete time intervals within our proposed framework. During each time interval, the controller actively monitors all incoming requests and computes the distribution of the workload for the subsequent interval.

To accomplish this task, the proposed system employs an online and proactive load balancing process that utilizes an internal prediction mechanism known as the Workload Predictor. This mechanism provides an estimation of the expected number of requests within the time interval based on Kalman Filter technique [37]. It is further assumed that the requests that are offloaded can be represented as a system [10] with uncertainty in both the process and the measurements.

$$L(t+1) = L(t) + w(t) \tag{3.1}$$

$$Z(t) = L(t) + v(t) \tag{3.2}$$

L represents the quantity of offloaded requests for the image recognition service throughout the time interval t. At time $t+1$, the workload value is determined by adding the current value to the process noise w, which represents the production of user requests. The matrix Q represents

the covariance of the process noise. The variable Z represents the measurement of the offloaded request, which is defined as the sum of the actual number of requests L and a measurement noise value $v(t)$. The noise value $v(t)$ is assumed to follow a normal distribution with a mean of zero, and its covariance matrix is denoted by R. The computation of the estimated request at the next time interval, denoted as $\hat{L}^-$, can be achieved by utilizing the following equations of the Kalman filter.

$$\hat{L}^-(k) = \hat{L}(k-1) \tag{3.3a}$$

$$P^-(k) = P^-(k-1) + Q \tag{3.3b}$$

$$K(k) = \frac{P^-(k)}{P^-(k) + R} \tag{3.3c}$$

$$\hat{L}(k) = \hat{L}^-(k) + K(k)(Z(k) - \hat{L}^-(k)) \tag{3.3d}$$

$$P(k) = (1 - K(k))P^-(k) \tag{3.3e}$$

The variable $\hat{L}^-$ represents the updated state value, which is determined by the measured value of Z and the extrapolated state value from the previous step. The term "K" refers to the Kalman gain, a numerical value ranging from zero to one that quantifies the significance of the measurement. The error covariance, denoted as P, refers to a concept used in statistical analysis and modeling to quantify the covariance of errors or residuals. The initial equation is utilized at each stage to predict the future requests for the subsequent interval, relying on the extrapolated values from the previous iterations.

Finally, a load balancer considers the processing load of each running TensorFlow instance and aims to direct requests to instances with lower utilization levels, thus balancing the workload more efficiently.

Monitoring. The monitoring service operates as a core component within the COSMOS architecture. Its primary function is to continuously gather and analyze critical data points related to the system's performance, incoming requests, and the utilization of resources, with a particular focus on the active virtual machines (VMs) deployed in the edge tier. This data collection process involves the systematic retrieval of various metrics and statistics that are

essential for maintaining optimal system functionality and responsiveness.

Some metrics among others are briefly described below:

- **Incoming Requests.** The monitoring service tracks and records the incoming requests to the edge computing infrastructure. This includes details such as request frequency, type, source, and destination. This information is vital for understanding the workload placed on the edge tier and identifying potential traffic patterns.

- **VM Performance Metrics.** For each active VM running in the edge environment, the monitoring service gathers performance metrics such as CPU utilization, memory consumption, network throughput, and disk I/O. These metrics offer insights into the health of individual VMs, helping detect any anomalies or resource bottlenecks.

- **Resource Utilization.** It also monitors the utilization of physical and virtual resources within the edge tier. This entails tracking the availability and usage of CPU cores, RAM, storage capacity, and network bandwidth. By assessing resource consumption in real-time, the monitoring service ensures that resources are allocated efficiently and identifies situations where resource scaling may be necessary.

3.4.1 Workflows and Component Interactions

Based on the previously demonstrated description of the constituent components of the COSMOS architecture, we hereby describe the interactions among them. A high-level illustration of the workflows and interactions between the core components of the COSMOS controller is depicted in Figure. 3.2.

The COSMOS architecture efficiently manages object recognition/identification offloading requests. The procedure begins when a user initializes a request for object identification. This request is evaluated by the Offloading Decision Mechanism at the Device Tier *(step 1)*, which takes into account criteria, such as position and signal strength. After the request has been

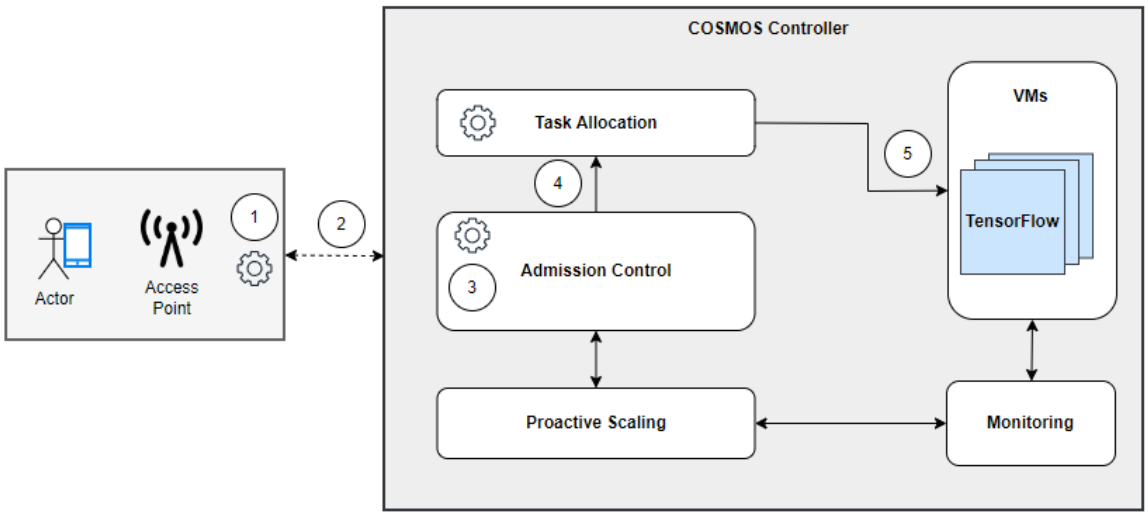


Figure 3.2: COSMOS Workflow.

checked to ensure that it satisfies the predetermined requirements, it is subsequently sent to the COSMOS controller which is located at the Edge Tier *(step 2)*.

When the request is received, the COSMOS controller initiates the Admission Control process. This operation involves the application of Kalman filtering and the monitoring of data in order to determine the load that is being placed on the system. Requests that exceed the predetermined load levels or yield low signal strength are rejected, whereas requests that satisfy the criteria are then submitted to further processing *(step 3)* in the following phase. The requests that have been granted approval are sent in an effective manner to the Task allocation module, which then breaks down the available time into intervals in order to determine an appropriate distribution of the workload *(step 4)*. The Workload Predictor is a computational tool that is developed to forecast the expected quantity of queries or jobs that will be received during a particular future timeframe. This is achieved by looking at historical data and projecting forward into the foreseeable future. The aforementioned data is then utilized in the subsequent phase, *(step 5)*, to distribute the requests across the operating TensorFlow instances that are tasked with the responsibility of object identification. The utilization of this integrated technique ensures a streamlined and efficient procedure for the recognition of objects, while concurrently regulating the workload resulting in improved overall system performance and better user experience.

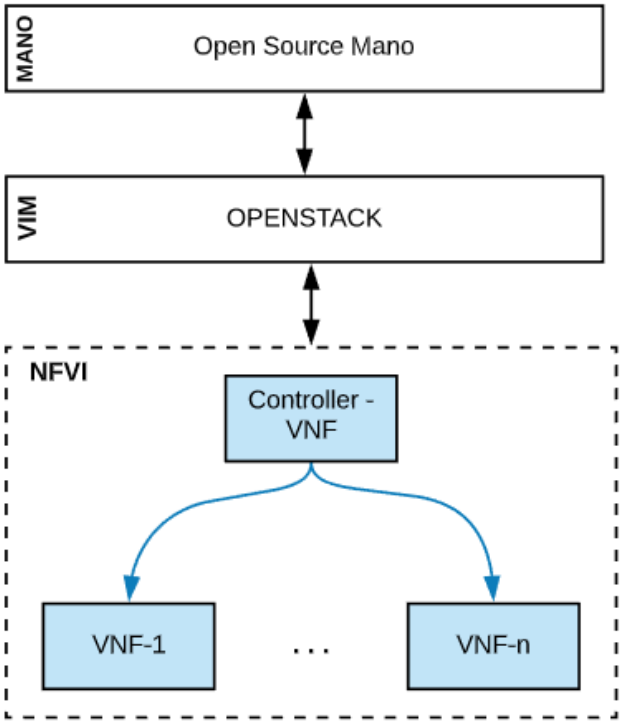


Figure 3.3: Experimental deployment of COSMOS.

3.5 Evaluation

3.5.1 Experimental Setup

In accordance with the COSMOS orchestration framework described in the section above, a service chain consisting of four Virtual Network Functions (VNFs) has been specified and deployed in the experimental facility. The primary responsibility of the controller VNF is to handle admission control and load balancing among the remaining three TensorFlow VNFs. Each VNF has been deployed, managed, and orchestrated using OpenStack and OSM, based on the principles of Management and Orchestration (MANO).

The controller VNF has been associated with 4 CPU cores, 8 GB of RAM, and 20 GB of storage space to ensure optimal performance. For the TensorFlow VNFs, three different flavors have been utilized, each with varying resource specifications. The resource specifications for each flavor are detailed in Table 3.1. As discussed in the previous Section, incoming user requests are distributed among these TensorFlow flavors to achieve load balancing.

Table 3.1: TensorFlow flavor specifications

Flavors	vCPUs (cores)	RAM (GB)	Storage (GB)
Small	2	2	10
Medium	4	4	20
Large	8	8	20

It is important to note that while the proposed architecture has been designed with a single Multi-access Edge Computing (MEC) domain in mind, it can be easily expanded for use in federated cloud infrastructures.

For experimentation purposes, we have created appropriate software images that correspond to the required functions for our evaluation. The functions of the controller, along with their corresponding Application Programming Interface (API), have been installed in the Linux-based Ubuntu Bionic 18.04 LTS operating system as the first image. The second image features the TensorFlow-based object recognition functions. The VNFs have been deployed, managed, and orchestrated based on the MANO principles using OpenStack and OSM. Suitable VNF and Network Service (NS) descriptors have been defined to describe all required functions and their chaining as a VNF-graph.

3.5.1.1 TensorFlow Model

The open-source TensorFlow object detection framework has emerged as a valuable tool for constructing, training, and deploying object detection models. In particular, the object detection API, which is utilized during the experiment, provides the capability to run inference jobs on pre-trained object detection models, thereby eliminating the need for training models from scratch. Among the various models available, we employ the faster RCNN Inception V2 COCO mode [38]l, given its high speed predictions (58ms) and decent mean Average Precision (28 mAP). This model has been pre-trained on the Common Objects in Context (COCO) dataset, which is a large-scale object detection, segmentation, and captioning dataset.

Figure 3.4: Model Retraining Process

The COCO dataset comprises over 330K images, with more than 200 of them being labelled, 1.5 million object instances, 80 object categories, and 250K people with key points. We leverage the faster RCNN Inception V2 COCO model for our experiment, as it has demonstrated high performance and accuracy in object detection tasks.

To prepare our model for re-training, we store the pre-trained model's weights as initial weights and subsequently readjust them after adding our dataset. Our training data comprises approximately 5 minutes of high-resolution (1920x1080), 30 frames/second video captured from the Millennium Square of Bristol Smart City testbed. We use the FFmpeg software to extract 100 images per class, each with a resolution of 800x600, resulting in a total of 500 images. In Figure. 3.4 we can see an instance of the re-training process. It is worth noting that the images are carefully selected to ensure diversity, encompassing a wide range of angles and lighting conditions. To augment our dataset further, we create a mirror transformation of the 500 selected images, ultimately resulting in a total of 1000 images.

Figure 3.5: Bristol's Infrastructure.

3.5.2 Experimental Results

To evaluate the effectiveness of the COSMOS framework, we deployed it in a large-scale experimental facility of the 5GinFIRE project, utilizing OpenSourceMANO (OSM) for Network Function Virtualization (NFV) orchestration. The experimental evaluation of COSMOS was conducted in a realistic environment with various PoIs located in Bristol's square and is depicted in Fig. 3.5 within the facility. Real mobile users were recruited and connected via Wi-Fi access points to the edge cloud infrastructure. The measured response times were used to validate the efficiency of the COSMOS orchestration functions, such as load balancing and admission control.

The primary objective of our experimental evaluation is to assess the feasibility of offloading object identification tasks into an edge cloud infrastructure. To this end, we have utilized the metric of total response time, which is further divided into individual constituent metrics, namely computation and transmission time. Our approach involves offloading the most demanding portion of the identification task to the mobile edge cloud infrastructure, leveraging

the superior hardware of edge servers compared to consumer-grade hardware of client devices.

To conduct our experiment, we have generated a dataset consisting of 15224 object detection requests triggered within 30-second time intervals. This dataset is classified into records based on the number of requests, with corresponding values averaged for each record. Ultimately, 653 such time intervals have been recorded and analyzed.

Our experimental results, as depicted in Fig. 3.6, indicate that the application's execution, i.e., response time, steadily remains under 6.3 seconds, with the minimum time observed being 3.2 seconds. The most significant insight that can be drawn from this plot is the non-exponential nature of the increase in the average response, computation, and transmission times. This can be attributed to the efficient utilization of the three distinct flavors of TensorFlow VNFs deployed, as discussed above.

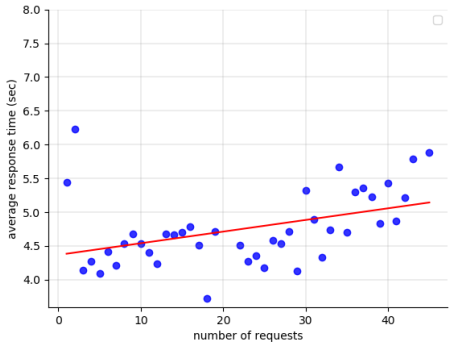


Figure 3.6: Average response time (grouped observations).

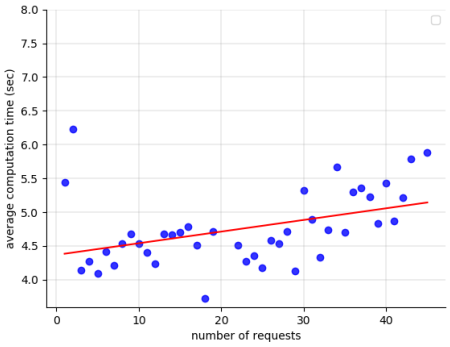


Figure 3.7: Average computation time (grouped observations).

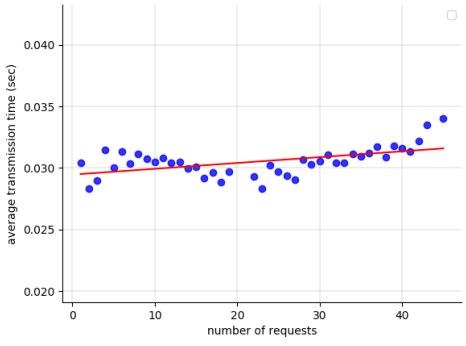


Figure 3.8: Average transmission time (grouped observations).

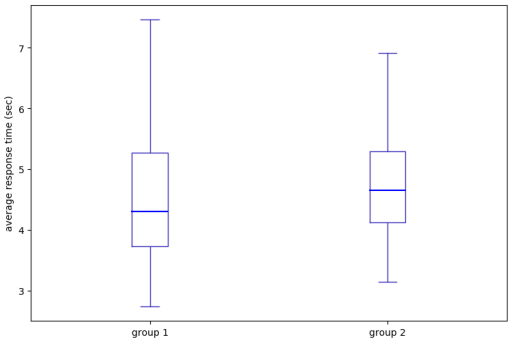


Figure 3.9: Boxplot of response time (split into groups by the median of requests).

In our analysis, the selection of the appropriate TensorFlow VNF flavor is determined by the volume of incoming requests, with a near-linear increase in key metrics deemed satisfactory.

However, we observe a significant variation in the standard deviation of our observations, which we attribute to the varying complexity of calculations based on individual images sent to the application.

Our analysis also reveals a strong correlation between response (Fig. 3.6) and computation time, indicating that computation time (Fig. 3.7) is the dominant factor in the overall application execution time. The negligible contribution of transmission time to the overall response time is highlighted by the near-order of magnitude lower scale of average transmission time and is depicted in (Fig. 3.8). Furthermore, the admission control mechanism ensures low deviation from Fig. 3.8 is the mean for all transmissions, with a negligible near-linear increase observed when the number of requests increases.

To further analyze the application's behavior in terms of response time, we divide our observations into two separate groups. This separation is driven from the median of our dataset, *i.e.*, the value that segregates records in two. The first half comprises time intervals where less than the *median* number of requests are triggered, while the second includes the ones exceeding it. Fig. 3.9 illustrates the correspondence of the two variables' distributions, with the second group demonstrating a slightly narrower deviation. We point out that the effectiveness of utilizing a range of different flavors becomes apparent. That is, the mean of the second group, *i.e.*, the group handling a larger number of requests, does not substantially differ from the one of the first group.

Based on the discussion above, we emphasize on three key points. The primary driver of the response time is the delay incurred for processing, a task which is accelerated by its offloading to the MEC servers. Furthermore, the transmission time is negligible with respect to the total response time.

A quantitative analysis of the execution rate of requests is used to evaluate the admission control mechanism. This analysis is performed both locally and on the edge cloud. As can be seen in Figure 3.10, a large percentage of inbound requests, particularly 91.37%, were successfully processed at the edge, but only 8.63% were successfully executed locally.

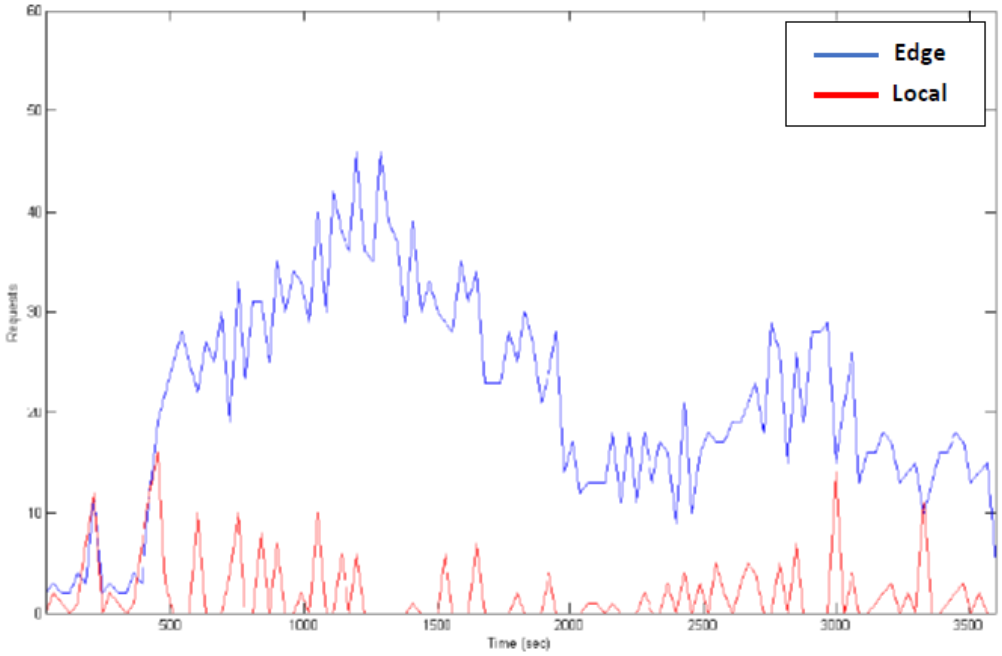


Figure 3.10: Edge vs Local

This result shows that the system is efficient and capable of managing a major amount of user requests within the edge cloud in an efficient manner. The effectiveness of the admission control strategy in preventing the edge cloud infrastructure from becoming overloaded while also maintaining a high level of service quality.

3.6 Related Work

In this section, we present an overview of related work on the domains (i) edge computing and (ii) NFV orchestration.

Edge Computing: An important objective of MEC is computation offloading to satisfy requirements, such as the reduction of power consumption in mobile devices and low response time from mobile and IoT applications. In this respect, Avgeris et al. [39] present an offloading framework for IoT-enabled applications, which is based on vertical and horizontal scaling of the MEC infrastructure. The scaling approach is based on linear systems combined with set-theoretic controllers. In [40] a deep Q-learning method, which significantly improves the performance of the computation offloading is proposed. This can be achieved through the

minimization of the service latency. In [41], a scalable edge computing framework for early fire detection is presented. An image processing service is hosted on an edge cloud and the regulation of the application's response time is based on linear optimization and state feedback controllers. Leivadeas et al. [42] propose a meta-heuristic approach for the placement and the deployment of the VNFs of IoT-enabled applications in order to minimize end-to-end communication delay and the overall deployment cost. MAGA [43] is a mobility-aware, genetic algorithm-based offloading and task allocation system that seeks to reduce the energy consumption of mobile devices and meet the response time requirements of IoT applications. Finally, authors in [44] propose an architecture for collaborative content caching, taking advantage of the available computing and storage resources at the wireless edge (*e.g.,* base stations, Wi-Fi access points).

Beyond existing solutions, this framework provides a holistic resource orchestration framework for mobile and IoT-enabled applications. Load balancing is carried out based on the processing capabilities of the predefined VNF flavours. Furthermore, the offloading decision is based on contextual information (*i.e.,* user's position and wireless signal strength) and the proposed workload estimation methodology, which guarantees the response time requirements and the efficient allocation of MEC resources.

NFV Orchestration. A significant amount of work has been conducted on establishing and advancing the NFV paradigm. Most work on NFV has focused on NFV orchestration, which encompasses aspects, such as service chain embedding [45], VNF scaling [46], and service chaining.

Besides OSM [17], another well-known orchestration platform, which has been developed in the context of the T-NOVA project, is TeNOR [14]. TeNOR acts as an NFV orchestration module and is responsible for the provisioning, configuration, monitoring, and optimized operation of service chains over virtualized infrastructures. However, neither TeNOR nor similar NFV orchestration mechanisms (*e.g.,* [47, 48]) explicitly address the various needs of computational offloading orchestration in edge clouds. Instead, they are mainly targeted at core cloud environments which, inherently, are less dynamic and agile than edge cloud infrastructures.

COSMOS overcomes the limitations of such orchestration mechanisms, offering mobility-aware computation offloading for mobile and IoT-enabled applications. In particular, it focuses on dynamic aspects of NFV orchestration (*e.g.*, dynamic load balancing), going beyond static service provisioning which is the main focus of most existing NFV orchestration mechanisms. Finally, the deployment of COSMOS allows for a preliminary examination of which extent the existing NFV orchestration platforms (*e.g.*, OSM) can cope with service deployment and scaling for mobile and IoT applications in smart city environments.

3.7 Conclusions

In this chapter, we presented the architecture and experimental evaluation of our proposed orchestration framework for computation offloading from IoT/mobile devices to edge clouds. COSMOS effectively integrates edge computing with artificial intelligence and software-based network principles, such as NFV and SDN, providing an end-to-end solution for offloading computationally-intensive workloads such as object identification.

Our evaluation results from a large-scale experimental facility demonstrate that mobile computational offloading is a feasible approach for alleviating the computational burden from client devices. The utilization of proximate MEC resources further facilitates the scaling of such services in response to evolving demands from mobile users. The proposed orchestration framework enables the concurrent handling of requests from multiple users, while considering the workload prediction, load balancing, admission control, and monitoring.

Our experimental evaluation indicates that the proposed orchestration framework sustains high user-perceived service quality, as quantified in terms of response time. By subdividing the response time into individual constituent components, i.e., computation and transmission time, we gain further insights into the performance of the system. Our findings indicate that computation time is the dominant factor in the overall application execution time, while transmission time is negligible.

Overall, the proposed COSMOS orchestration framework provides a comprehensive solution for

the efficient offloading of computationally-intensive workloads from client devices to proximate MEC resources. The framework's ability to sustain high user-perceived service quality and facilitate the scaling of services in response to evolving demands is crucial for providing seamless and efficient services to mobile users. Our experimental evaluation provides valuable insights into the performance of the system and highlights the importance of load balancing and efficient resource utilization for achieving optimal performance.

Chapter 4

Optimized Cross-Slice Communication for Edge Computing

4.1 Motivation

Over the past few years, network slicing has emerged as a powerful concept that allows network providers to lease customized bundles of computing, storage, and network resources to service providers, enabling the creation of innovative applications with stringent performance requirements in various domains of social and economic activity. This technology has been enabled by network function virtualization (NFV) [13, 14, 15] and software-defined networking (SDN) [23], which have revolutionized the way network infrastructure is designed, deployed, and managed.

However, the prevailing way of slice instantiation has inhibited optimized cross-slice communication (CSC), even when the slices are located in the same data center, rack, or server. This has resulted in significant communication latency and increased expenditure due to the consumption of additional bandwidth from the backhaul and transport network. This is especially crucial in the context of edge computing, where the distance between the edge cloud infrastructures and the network core is larger.

For example, in Figure 4.1 an augmented reality (AR) service co-located with a social media

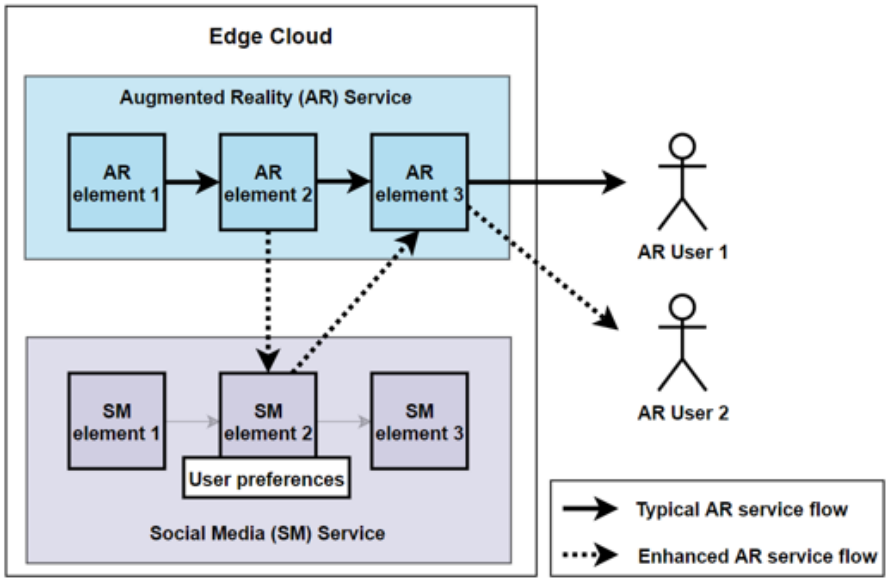


Figure 4.1: Cross-service interaction between an augmented reality (AR) service, and a social media (SM) service. [3]

(SM) service can retrieve user preferences from the SM service to offer an enhanced personalized AR experience to users. In this example, the AR service is the consuming service, while the SM service fulfills the role of the providing service.

4.1.1 Cross Slice Communication (CSC)

In this subsection, we present and discuss the notion of CSC and stress on the need for optimized CSC in the context of edge computing. In this respect, we distinguish between two types of services: (i) consuming services, and (ii) providing services. CSC empowers consuming services to enhance their functionality by accessing service elements or functions provided by other services. Such services may be co-located within the same (edge) computing infrastructure, thereby creating an opportunity for peering between the two services. Various cross-service interactions can be envisaged that would benefit from CSC.

Figure 4.2 highlights this limitation in terms of CSC establishment. The red dotted line illustrates the communication path between two co-located (but isolated) slices. Stemming from the need for resource/traffic isolation, these two slices will be realized as entirely distinct networks. As such, routing policies will direct the traffic from one slice outside of the data center

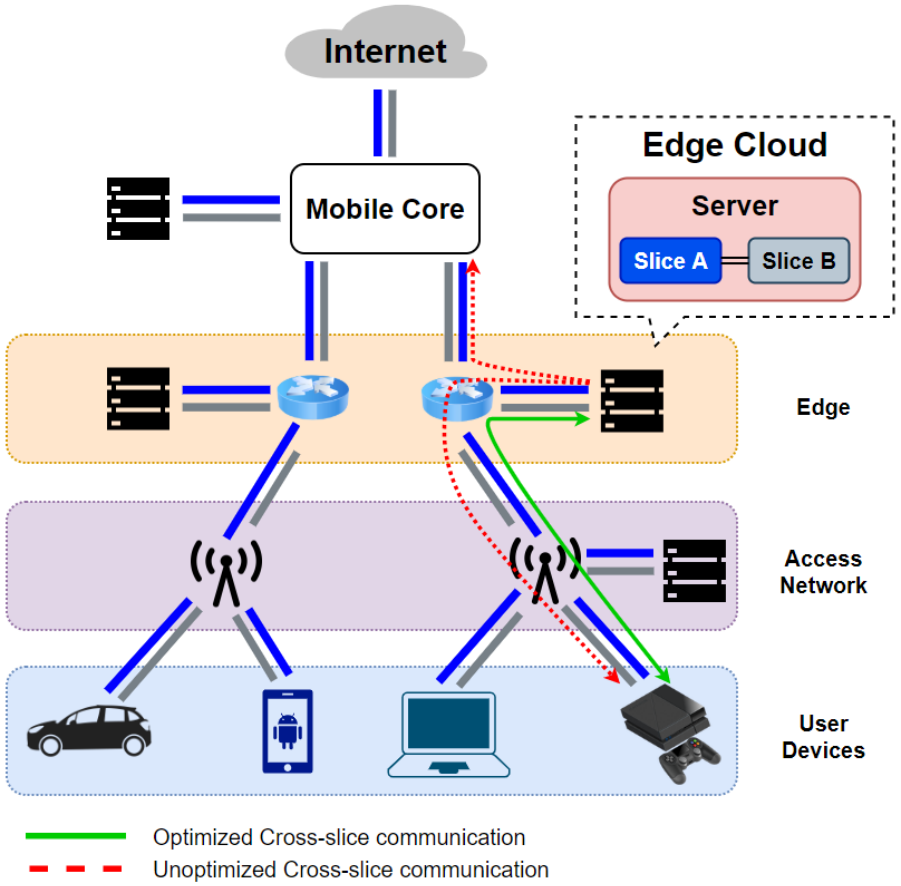


Figure 4.2: Optimized vs. unoptimized cross-slice communication.

(DC), through the (mobile) core network, and back to the same DC to reach the other slice. Nevertheless, the shared nature of the (edge) cloud infrastructure opens up opportunities for shortcuts in this communication. Therefore, we promote a form of direct peering between the two slices, which we call optimized CSC (represented by the green line in Fig. 4.2). In this case, CSC traffic is confined within the DC that hosts the two slices, obviating the need to traverse the entire mobile network infrastructure. The traversal of CSC traffic over short paths is very critical when considering the focus of edge computing on the support of low-latency applications. Besides latency savings, optimized CSC is further expected to reduce the traffic volume in the backhaul/transport network.

To better understand the potential benefits of optimized cross-slice communication (CSC), we conducted an experiment in which we deployed two virtual machines (VMs) corresponding to service components of two different but co-located network slices in an edge cloud infrastructure (Stackpath). We also deployed a third VM in the same region, using Amazon EC2, which

acted as a mobile core gateway that routed traffic between the two other VMs in the case of unoptimized CSC.

We measured the latency with both optimized and unoptimized CSC and found that while latency was low in both cases, the advantage of optimized CSC was significant for delay-sensitive applications. Specifically, we observed that for 80 percent of the delay measurements, optimized CSC yielded a latency saving of approximately up to 70 percent. In contrast, unoptimized CSC tended to skew the tail of the latency distribution, and the latency was not bounded, which could compromise safety in applications such as automotive. For time-critical applications

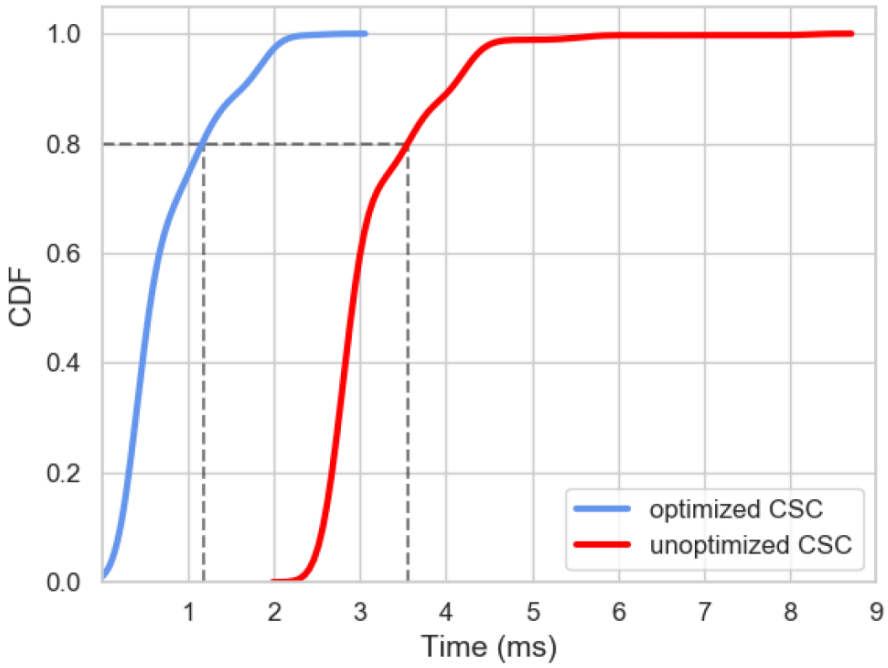


Figure 4.3: CDF of measured delay with optimized and unoptimized CSC.

typical in edge computing, low latency is not only important on an average basis but also on its upper bound. In this respect, unoptimized CSC cannot comply with such stringent delay budgets, which may create serious implications. Consequently, the lower delays incurred by CSC traffic in the case of optimized CSC are critical for the achievement of service key performance indicators (KPIs) in cross-service interactions.

The experiment is illustrated in Figure 4.3 the potential benefits of MESON, which facilitates optimized and secure CSC within edge clouds, fostering cross-slice/tenant interactions for next-generation services. By enabling the creation of highly customized network slices that can

be optimized for specific applications and services, MESON opens up a unique opportunity for services to get deeply integrated into the (edge of the) network infrastructure, realizing innovative applications with stringent performance requirements in various domains of social and economic activity.

4.2 Contributions

The contribution of this work is a new management and orchestration (MANO)-based architecture called optiMized Edge Slice OrchestratioN (MESON) that facilitates optimized and secure CSC within edge clouds, enabling cross-slice/tenant interactions for next-generation services. The MESON architecture provides the necessary means for the establishment of cross-service interactions, exploiting opportunities for service co-location.

MESON encompasses orchestration primitives for CSC-enabled service advertisement, service discovery, and CSC establishment, which are subject to operations/business support system (OSS/BSS)-level procedures expressing the intent of service providers to establish synergy. MESON also encompasses multi-access edge computing (MEC)-based descriptors for the expression of CSC offered services and intents, and we explain the interactions between the MESON architecture components for CSC establishment.

In particular, MESON enables service providers to express their intent to establish cross-slice synergy, which can be used to discover other services that could benefit from co-location. This approach leverages the inherent flexibility of NFV and SDN to enable the creation of highly customized network slices that can be optimized for specific applications and services.

Moreover, MESON aims to enhance the security of cross-slice communication by providing the flexibility to deploy security Virtual Network Functions (VNFs) within the CSC slice. These security VNFs leverage advanced cryptographic techniques and other mechanisms to contribute to the protection of data and services against potential unauthorized access and security threats. This approach acknowledges the specific challenges posed by edge computing,

where the physical proximity of network resources to end-users can introduce unique security considerations that are being actively addressed.

4.3 Architecture for Optimized CSC

In this section, we provide an overview of the MESON reference architecture, on which the MESON orchestration platform is based. More specifically, we outline the main roles involved in CSC, present the MESON architecture components, and also exemplify their main interactions. We will also take the opportunity to introduce the notion of CSC slice, which is instrumental in the establishment of CSC. Since we aim at fostering cross-service interactions, we introduce additional business roles, as beneficiaries of CSC. More specifically, we consider the following roles:

Infrastructure Provider, who owns the (edge) cloud infrastructure and offers resources under lease basis for slice deployment.

CSC-enabled Service Provider (CSC-SP), who is the tenant of a slice leased by the Infrastructure Provider. The CSC-SP deploys services onto certain PoPs, which are available for consumption via means of CSC by other slice tenants.

CSC-enabled Service Consumer (CSC-SC), who is a slice tenant that consumes services from co-located slices (i.e., operated by a CSC-SP) via CSC.

4.3.1 Architecture Components

The MESON architecture fulfills all the main requirements for the discovery of CSC-enabled services and the establishment of CSC between two slice tenants (i.e., CSC-SP and CSC-SC). More precisely, MESON supports the following functionalities:

- The advertisement of CSC offerings by a CSC-SP which can be consumed by another slice tenant.

- The expression of interest from a CSC-SC for the establishment of communication with a CSC-SP.
- The binding of CSC interests with offerings for CSC service consumption.
- The selection of the most appropriate PoP for the deployment of a CSC-enabled service, given that a CSC-SP may have deployed a CSC-enabled service on multiple PoPs.
- Communication path setup for CSC establishment between the slices of the CSC-SP and CSC-SC.

In order to meet the above requirements, we have specified a three-layer architecture that extends the ETSI NFV MANO reference architecture [49] in order to facilitate the discovery and consumption of CSC-enabled services in a controlled manner, exploiting service co-location. MANO provides NFV management and orchestration functionality, spanning the following two layers: (i) the Virtualized Infrastructure Manager (VIM), which is responsible for the management of the virtualized infrastructure across all resource types (*i.e.,*compute, storage, network), and (ii) the NFV orchestrations (NFVO) layer, which provides support of VNF lifecycle management, as well as VNF state management.

MESON introduces a CSC orchestrations layer on top of the two-layer MANO architecture, as illustrated in Figure. 4.4. More specifically, the MESON layer comprises of the following components, (i) **Service Registry**, (ii) **Edge PoP Selection**, (iii) **MESON Agent**, and (iv) **CSC Orchestrator/Optimizer**. Next, we briefly present all the architectural components, and we will elaborate more and present more details of the MESON Agent, and CSC orchestrator/Optimizer components.

The Service Registry handles the binding of CSC interests with CSC offerings, subject to an exposed policy, which we discuss in the next section. The Service Registry maintains a list of service instances offered by CSC-SPs. The interests expressed by CSC-SCs are also directed to the Registry.

PoP Selection identifies the most suitable PoP for the deployment of a slice requested by a CSC-SC. This module generates a PoP ranking, based on hard and soft requirements from

the slice tenant. The hard requirements are associated with attributes, such as the location (availability zone) of the PoP. The soft requirements are associated with computing and network performance indicators (*e.g.,* service response time), cost parameters, and technical support. The above criteria can be expressed by various types of numeric values, such as binary, range values, and unordered sets. Therefore, a multi-criteria decision making (MCDM) approach is deemed suitable for the PoP ranking [50].

MESON Agent performs the following main tasks: (i) exposes an interface to slice tenants for the expression of CSC-enabled service advertisements and interests (by CSC-SPs and CSC-SCs, respectively) and (ii) coordinates the CSC service discovery and instantiation workflows within the MESON layer. The MESON Agent is also responsible for interoperability with the underlying MANO layers via API calls directed to the NFVO. After the edge PoP Selection has generated the edge PoP ranking (in the case of Coordinated CSC), the MESON Agent configures a network slice description with all required instantiation parameters, which is, in turn, conveyed to MANO for the deployment of the CSC-SC's slice. After both slices (i.e., of CSC-SP and CSC-SC) are in place, the MESON Agent triggers CSC setup. To this end, MESON handles CSC establishment through the instantiation of an intermediate slice, dedicated to CSC. The benefits of this approach are two-fold: (i) CSC traffic remains isolated from the rest of the traffic within the edge PoP and (ii) the CSC slice facilitates the deployment of network processing functionality (*e.g.,* in the form of VNF chains), such as packet inspection and monitoring for the purpose of CSC policy control and enforcement, especially from the side of CSC-SP.

CSC Orchestrator/Optimizer is responsible for the lifecycle management of the CSC slice. More precisely, this module collects all required information (*e.g.,* network IDs, IP addresses, etc.) in order to instantiate the CSC slice and, subsequently, generate the required packet forwarding entries for secure communication between the CSC-SP and CSC-SC slice. Upon the preparation of CSC slice specification with all required instantiation parameters, the CSC optimizer conveys this specification to the VIM for the CSC slice deployment. In contrast to the CSC-SC slice, the instantiation of the intermediate CSC slice is handled directly by the VIM, without any intervention from the NFVO layer.

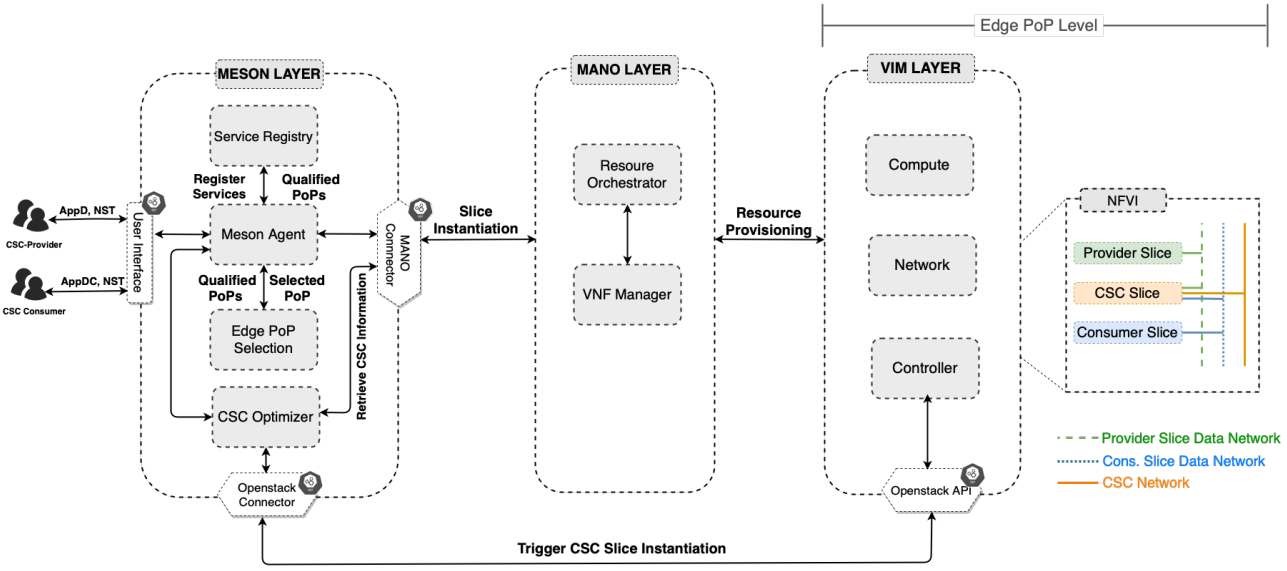


Figure 4.4: MESON architecture.

4.3.2 MESON Agent

The MESON Agent is a fundamental building block of the MESON architecture. At an abstract level, it can be described as a coordinator or orchestrator of the information exchanged between the elements that make up the broader MESON layer, namely the Service Registry and the PoP Selection. As a result, the MESON Agent is responsible for the proper and smooth operation of the MESON layer, and its placement as a central building block contributes to this direction, as it serves as a central communication node between the other elements of the architecture. The internal architecture of the MESON Agent is summarized in Figure 4.5. Specifically, the MESON Agent is composed of individual internal components, each of which performs a separate function. The MESON Agent was implemented in Python v3.6, while the Flask v2.0[1]package was used to implement the API. Communication with the other building blocks of the architecture is performed through REST API calls. Starting with the description of its basic functions, the MESON Agent is the building block that provides the required interface to the end user. Specifically, it allows CSC providers to upload their service descriptors (appDs) in YAML format we will discuss the appDs later in this chapter. At the same time, it gives CSC consumers the ability to upload their own descriptors, allowing them to express their interest in consuming a service through inter-process communication. These processes are achieved

[1]Source: `https://flask.palletsprojects.com/en/2.0.x/`

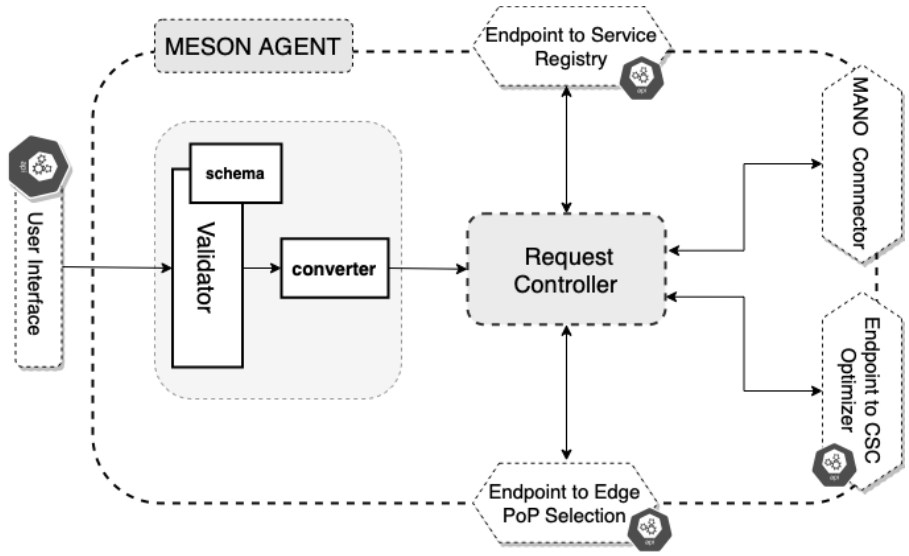


Figure 4.5: Internal architecture of MESON Agent.

through a simple environment implemented in HTML5.

The core component of the MESON Agent is the **Request Controller**, which handles all the communication among MESON-layer components. The Request Controller acts as an internal orchestrator. The interaction among the components is achieved through API calls using Python's request library. The Request Controller receives the offered services along with the candidate edge PoPs from the Service Registry, and subsequently relays this information to the PoP Selection component. The Request Controller also triggers the process of CSC-SC slice instantiation through a REST API call to the NFVO layer. Another key component of MESON Agent is the **Validator**. Exploiting the Cerberus Python library[2], the main task of the Validator module is to determine whether the uploaded descriptors are in the correct YAML format. This validation is carried out in comparison with a well-defined JSON schema. Furthermore, the Converter module is responsible for the conversion of descriptors from YAML to JSON format

Finally, MESON Agent serves as the connection between the MESON layer and the MANO layer. Specifically, the MESON Agent initiates the process of creating the network slices for both CSC providers and consumers. This connection is achieved through the **MANO connector** module, which uses REST API calls to the Northbound interface (SOL 005) of the Open

[2]Source: `https://docs.python-cerberus.org/`

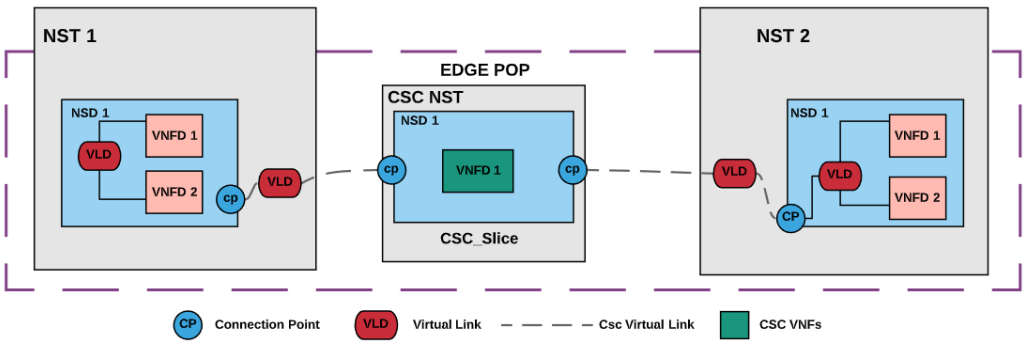


Figure 4.6: Intermediate CSC Slice.

Source MANO[3]to trigger the creation of network slices. Once the creation of the network slices is completed, the MESON Agent informs the end user with the appropriate message.

4.3.3 CSC Slice Instantiation Scenarios

In this section, we elaborate on CSC through the detailed examination of two scenarios that reflect different approaches to CSC. As already mentioned, our CSC setup relies on MANO and the descriptors defined by this NFV orchestration architecture. More precisely, network slices are defined using NSTs, each containing one or multiple NSDs. In turn, the NSDs are associated with VNFs which are defined using VNFDs. Although a NSD can include multiple VNFDs, in our setup, each NSD associated with the intermediate CSC slice contains only a single VNFD. The two main CSC scenarios under our consideration are the following:

Baseline CSC scenario. Figure 4.6 illustrates the baseline scenario for CSC instantiation. We assume that two slices from different tenants, along with the services they deploy, are up and running in the same edge cloud infrastructure. Optimized communication between these two slices is established via an intermediate dedicated slice, as described. More specifically, the intermediate CSC slice is an independent slice owned and operated by the infrastructure provider. Besides communication, a CSC slice can contain management and security functions, in the form of VNFs, used by the communicating slice tenants. We point out that, unlike the communicating slices which may include segments in other infrastructures, the intermediate slice is confined within the edge cloud infrastructure that CSC is being set up. A particular

[3]Source: `https://osm.etsi.org/`

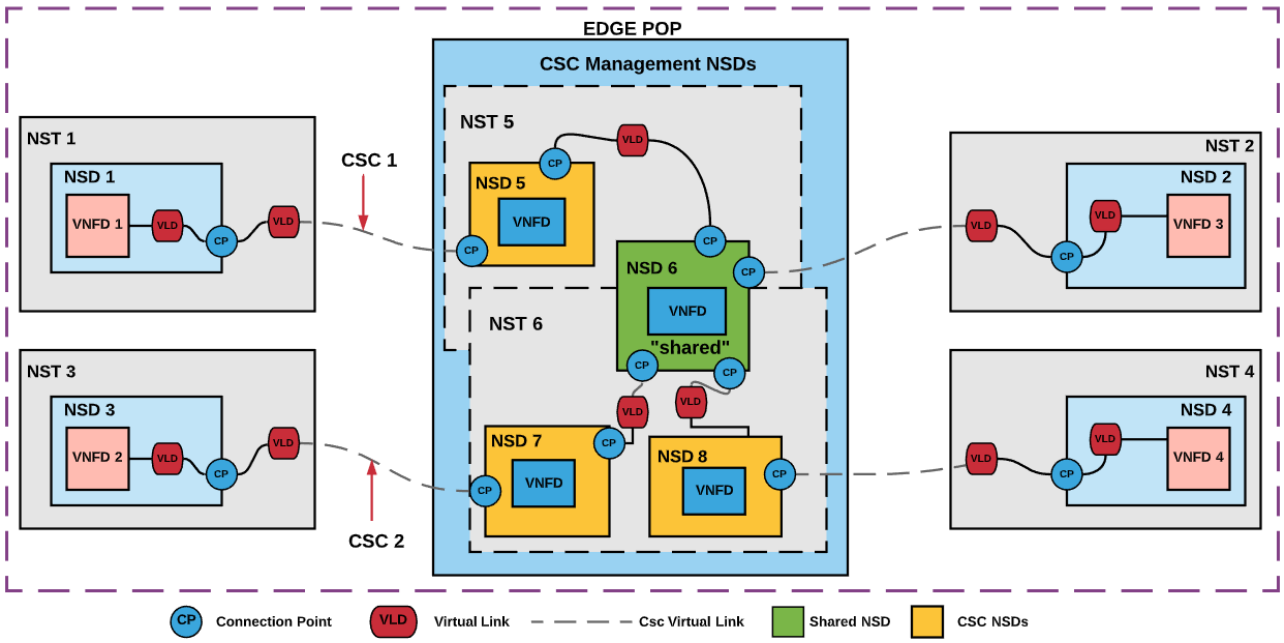


Figure 4.7: Shared CSC scenario.

restriction introduced by this scenario is that each VNF is dedicated to a single CSC slice, and as a consequence, to a specific pair of communicating slices.

Shared CSC scenario. We point out that edge infrastructures are commonly resource-restricted environments and, as such, a note-worthy aspect is the sharing of CSC management and security functions among different CSCs. In such scenario, a pool of CSC management NSDs can be pre-configured and instantiated by the corresponding infrastructure provider, facilitating their binding with CSC slices, during their instantiation. As such, CSC establishment can be accelerated, since the need for spawning new VNF instances is obviated. In essence, CSC setup mainly requires network configurations and service chaining. Nevertheless, careful attention should be paid by the infrastructure provider at the CSC management level, possibly with additional configurations (*e.g.*, VNF state management, policing), to ensure an appropriate level of isolation between pairs of slices.

Fig. 4.7 illustrates such a shared CSC. In particular, two pairs of communicating slices (*i.e.*, NST 1–2 and NST 3–4) share NSD 6, while NSD 5 is dedicated to CSC 1 (connecting NST 1 with NST 2) and NSDs 7–8 are associated with CSC 2 (connecting NST 3 with NST 4). As already explained, the sharing of NSD 6 can lead to resource savings for the infrastructure

provider, as well as faster CSC instantiation.

4.3.4 CSC Orchestrator/Optimizer

4.3.4.1 CSC Establishment Requirements

In this subsection, we elaborate on CSC establishment, using an intermediate CSC slice. We further analyze the main functional requirements to enable cross-service interactions. To enable cross-service interactions between colocated services, we introduce the notion of a dedicated intermediate slice, which effectively binds the two communicating slices. This so-called CSC slice will be instantiated and managed by the infrastructure provider. Figure 4.6 provides a high-level illustration of a CSC slice that facilitates the interaction between two services, based on the previous use-case example. Besides offering connectivity, the CSC slice fulfills the following main requirements:

Traffic isolation. The CSC slice isolates the traffic exchanged between the two communicating slices from the rest of the traffic flowing within the DC. This requires the provisioning of a secure communication path between the two slices. In addition, bandwidth may be reserved for the traffic that will traverse the CSC path.

Policy Control. The CSC slice can facilitate the embedding of processing functionality within the CSC path for CSC policy management and control. A CSC policy can be specified through descriptors and may encompass processing functions (*e.g.,* in the form of VNFs) for packet inspection, monitoring, and billing. As such, any CSC peer can observe whether the traffic exchanged over the CSC path complies with an established SLA. Besides resource reservation for such VNFs (which will be deployed as part of the CSC slice), service chaining is also required to ensure that the CSC traffic will traverse the VNFs in the correct order. This, in turn, will ensure that the CSC policy will be enforced as required by the CSC pair.

The chaining of the CSC-slice VNFs requires information related to (i) the specific application that originates from application descriptors and (ii) the slice instances retrieved from the

NFVO deployment. Other mandatory parameters regarding network-level connectivity, such as the identification of initiated network interfaces and physical hosts of the VNFs have to be considered, alongside a prescribed CSC policy. In essence, the instantiation of a CSC slice is deemed pivotal for the establishment of a secure CSC. Our proposed CSC orchestration architecture which is presented below is centered around the instantiation and management of such CSC slices.

4.3.4.2 CSC Orchestrator Architecture

In this subsection, we present the internal architecture for the management and orchestration of CSC instantiation that fulfills all the technical requirements. We exemplify the CSC instantiation workflow and discuss the main cross-layer interactions. Furthermore, we exemplify the CSC instantiation workflow through a sequence diagram.

The process of achieving cross-service communication at the edge infrastructure level requires careful planning and execution. The first step in this process is data collection, which involves gathering information about the participating slice tenants and the services they require. This information is then processed to determine the specific requirements of each service.

Once the requirements have been determined, the next step is to develop the appropriate interaction and configuration tools. These tools are necessary to manage and operate the various services at different levels. This includes the development of management tools for the service providers and operation tools for the end-users.

To ensure that the cross-service communication requirements are met, the CSC descriptors defined by ETSI MEC [51] specifications are used. These descriptors provide a standardized way of describing the services and their requirements. The AppDs provide a comprehensive overview of the requirements for cross-service communication, which helps to ensure that all the necessary components are in place. This includes ensuring that the VNFs are properly connected and that the peering policy is properly configured. By following the appropriate sequence of processes and using standardized descriptors, the cross-service communication requirements can be met

efficiently and effectively.

The application descriptors (AppDs) contain mandatory information about the CSC establishment. This information includes service-level dependencies, involved VNFs connectivity, and peering policy.

For the establishment and orchestration of secure CSC between the participating tenants at the VIM level. In this respect, Figure 4.8 depicts an overview of the CSC Orchestrator architecture, which includes the following components:

Data Retrieval API. The Data Retrieval API serves as a critical component in the establishment of CSC between different slices. This process begins with the collection of essential data required for the successful creation of CSC, which is done by the Data Retrieval API. This essential data pertains to both the application descriptors of the slices involved as well as their corresponding instantiation records.

To collect this data, the Data Retrieval API integrates two end-points that facilitate communication with the AppD Registry and the NFVO, respectively. Through these endpoints, the API is able to retrieve the necessary information and organize it in a structured format. Specifically, the data retrieved from the AppD Registry is represented in YAML format, while the instantiation logs are represented in JSON format.

By utilizing these two different formats, the Data Retrieval API can effectively manage the different types of data involved in the CSC establishment process. The YAML format is particularly well-suited for the representation of application descriptors, as it allows for the description of complex data structures in a human-readable format. Conversely, the JSON format is more commonly used for the representation of data in web applications and is especially useful for the representation of instantiation logs due to its simplicity and flexibility.

Data Processing. After the Data Retrieval API collects the essential data required for CSC establishment, the next step is to retrieve specific fields from the descriptors and NFVO records. The purpose of this step is to extract the necessary parameters for the configuration of both the CSC network and the VNFs that the CSC slice consists of.

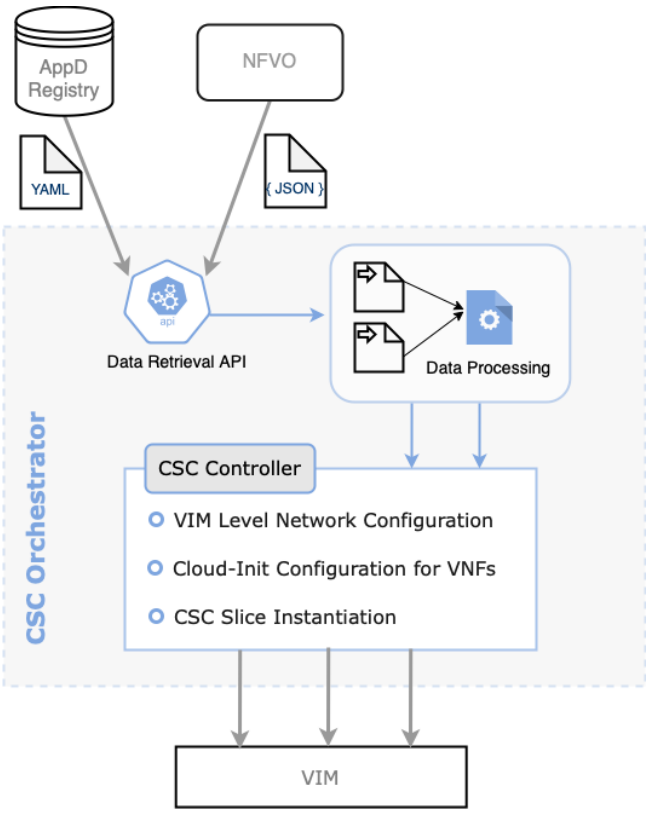


Figure 4.8: CSC Orchestrator architecture.

This is achieved by using the collected data to retrieve involved VNF indexes, PoP, and CSC VNFs descriptor keys from the corresponding AppDs for both tenants. Additionally, VIM level information such as VNF hosts, connection points, and IP addresses are retrieved and stored. These parameters will be used later during the configuration process carried out by the CSC Controller.

By extracting specific fields from the descriptors and NFVO records, the CSC Orchestrator architecture is able to effectively configure the CSC network and VNFs in a manner that meets the requirements of the participating tenants. This process ensures that the CSC slice is optimized for the specific needs of each tenant, thereby improving the overall efficiency and effectiveness of the CSC orchestration process.

CSC Controller.The core component of the CSC Orchestrator is the CSC Controller. This module is responsible for the orchestration of the whole CSC establishment process, from the network and VNF configuration to the CSC slice instantiation. The CSC controller utilizes the cloud-init standard and the scheduler filters of the VIM, aiming at optimized CSC instantiation

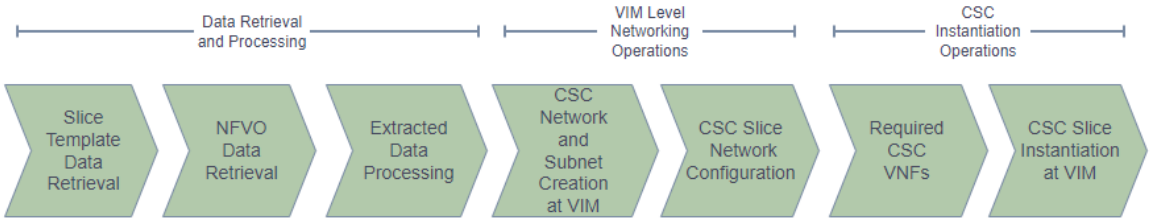


Figure 4.9: CSC establishment workflow.

in terms of physical host placement and network connectivity. This, in turn, can lead to the minimization of the network traffic within the edge infrastructure and to low-latency communication among the slices. The main tasks of the CSC Controller are the following: (i) the CSC network creation and configuration at the VIM layer, (ii) the cloud- init-based configuration of the required CSC VNFs, (iii) the routing of the network traffic between the involved slices through the CSC VNFs, and (iv) the CSC slice instantiation with co-location constraints, using the corresponding APIs of the VIM.

4.3.4.3 CSC establishment workflow

In this subsection, we exemplify the CSC establishment workflow (Fig. 4.9). In particular, CSC establishment consists of three phases, *i.e.,* (i) data retrieval and processing, (ii) network configuration at the VIM level, and (iii) CSC slice instantiation.

To delve into more details, we illustrate all these steps in the sequence diagram of Fig. 4.10. During the initial phase, the CSC Orchestrator requests the tenant details (step 1) from a known application service catalog. These details are parsed from the application descriptors and consist of (i) the network service descriptor name of the running slice, (ii) the VNF index number of the VNF that will be consumed, and (iii) the PoP name. All the data received (step 2) is stored (step 2.1) in a list to use it later for the instantiation phase. Subsequently, the CSC Orchestrator requests the instantiation parameters from the NFVO layer (step 3) through a REST API call. The essential instantiation parameters comprise (i) the *IP addresses* of the VNFs that will participate in the CSC, (ii) the unique *device id* of the server(s) that host VNFs in the underlying VIM, and (iii) the *network id* of the virtual networks that the VNFs are attached to, within the VIM. All received data is stored in a list (step 4.1). The data

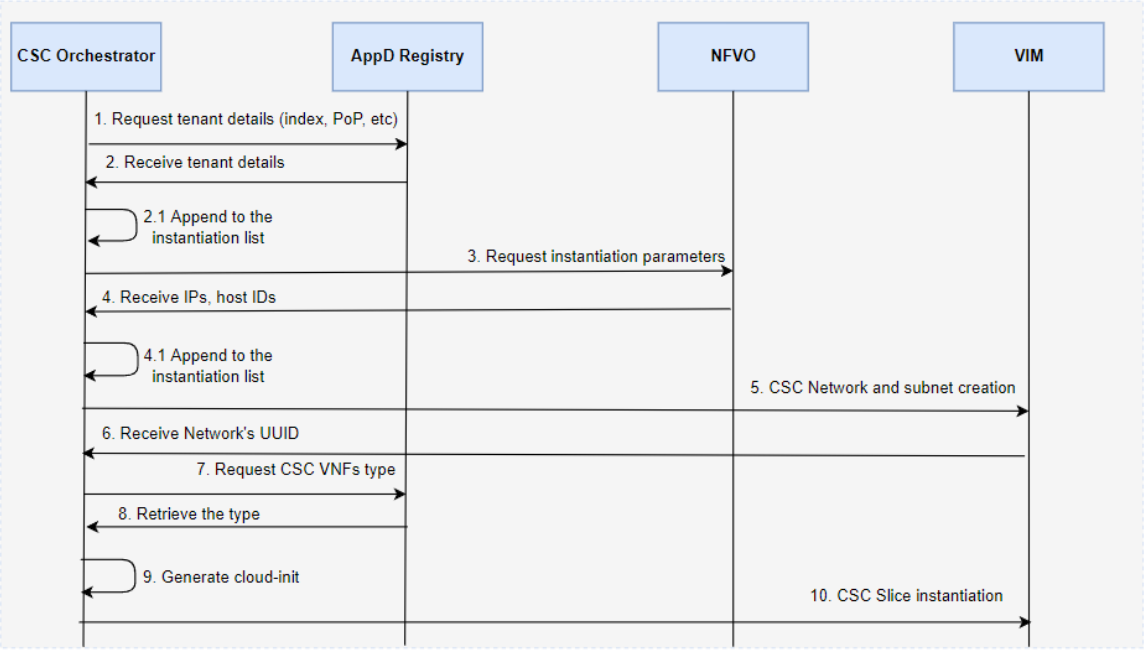


Figure 4.10: CSC Orchestrator sequence diagram.

that has been collected at that point is used to gradually compose the commands with which the CSC Orchestrator will trigger the VIM-to-CSC slice instantiation process in the selected infrastructure and to also perform the basic configuration of the intermediate CSC slice. Next (step 5), the CSC Orchestrator requests the creation of a dedicated network (namely *csc-service-network*), and a dedicated subnet on the underlying VIM. Additionally (step 6), the UUID of the newly created network is discovered, *i.e.,* this is the network to which the CSC slice will be attached for the purpose of isolation, control, and security.

Through this approach and by utilizing a basic configuration, we are able to control the flow of the CSC-enabled traffic. In this context, basic configuration refers to the control of network traffic from the "data" network of the consumer (where the corresponding VNF is connected) to the network to be consumed (provider VNF) and vice-versa. These networks are on purpose only accessible from the CSC slice through corresponding interfaces. Effectively, only the IP address of the CSC slice is known by each corresponding slice network, and only through this, any interaction between the consumer and the provider slices is permitted.

In the next step (step 7), the CSC Orchestrator requests the workload type of the VNFs that will be embedded in the CSC-slice from the application service catalog. Workload types or roles

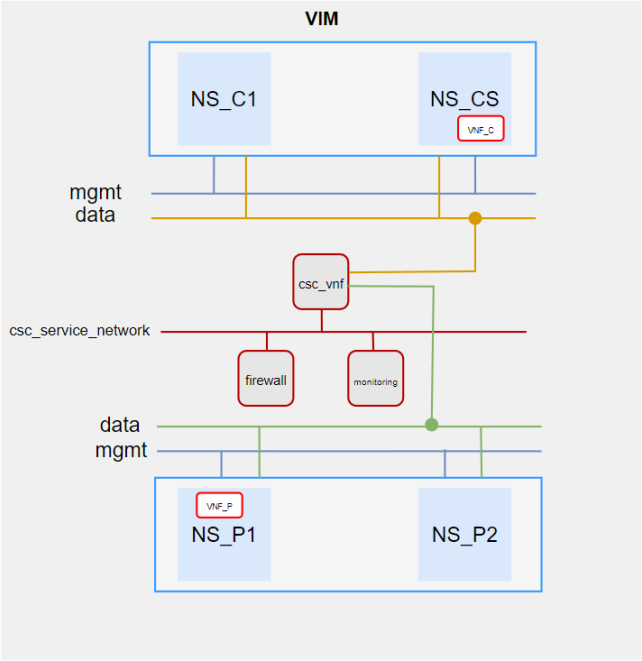


Figure 4.11: VIM-level CSC overview.

can be identified, such as firewall, intrusion detection system (IDS), monitoring, *etc.* After the retrieval (step 8) of the CSC VNF type, the CSC Orchestrator has gained all the necessary information to build the instantiation command. This information encompasses the forwarding rules and policy during the interaction of the deployed VNFs, from corresponding network interfaces. These rules, having as parameters the information collected in the process above, are placed in sequence in a *cloud-init* (step 9) file, which is activated with the instantiation of the virtual machines of the CSC slice. Finally (step 10), the CSC Orchestrator triggers the process of the CSC-slice instantiation directly in the underlying VIM. Fig. 4.11 illustrates an overview of a CSC establishment within a VIM.

4.3.5 Information Model

The MESON information model consists of several key fields, designed to maintain generality while enabling effective communication. **Application descriptions (AppDs)** are used to describe offered CSC services, and consumption profiles convey the requirements for desired CSC services. A central entity of the model is the *Peer Policy*, which encompasses the service provider's CSC policy and requirements. Additionally, the *Dependency Profile* captures

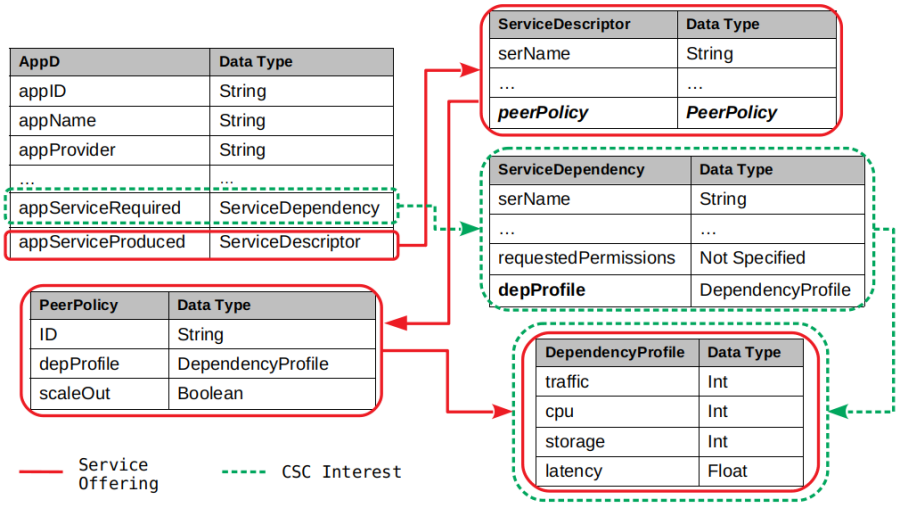


Figure 4.12: Descriptors for CSC interests and service offerings.

properties relevant to CSC policy, including acceptable resource consumption limits such as computing power, storage, and network resources.

Matching service consumption requests with available services is a core capability of the MESON information model. By comparing the characteristics of services through *Service Dependencies*, as well as considering both providers' and consumers' Peer Policies, the Service Registry identifies services that fulfill specific consumption requests. In cases of multiple matches, an Infrastructure Selection component sorts the results based on predefined criteria, guiding the deployment of the Consumer CSC component on the most suitable infrastructure.

4.4 Evaluation Results

In this section, we discuss all the evaluation results of the MESON platform. All tests are conducted on a Dell PowerEdge R440 server with a single Intel Xeon Silver 4110 CPU with 8 CPU cores at 2.1 GHz and 128 GB of RAM. The server hosts OSM version 7.0 and OpenStack version 17 (Queens), which fulfills the role of the VIM. OpenStack manages three virtualized compute nodes deployed on top of VMWare. For each one of the compute nodes, we have allocated 8 VCPUs, 32 GB of RAM, and 100 GB of storage. For the implementation of the VNFs deployed within the CSC slice, we rely on Click Modular Router, which is a modular framework for building flexible and configurable packet processing systems, such as routers,

Table 4.1: Network slice request specifications

Request Type	**Type 1**	**Type 2**	**Type 3**
Number of NSes	1	2	3
VNFs (per NS)	1	1	1
vCPU Cores (Total)	2	4	6
RAM (Total)	2GB	4GB	6GB
Disk (Total)	10GB	80GB	120GB

switches, and firewalls [52].

4.4.1 MESON Evaluation

4.4.1.1 Performance tests

Our focus in evaluating the performance of MESON lies on the impact of the MESON layer on the establishment of the Cross-Slice Communication (CSC). We take into account various factors, such as the interactions between the MESON layer, the Virtual Infrastructure Manager (VIM), and the Network Function Virtualization Orchestrator (NFVO). This includes information retrieval, network configuration, and execution of the instantiation process.

To accurately measure the performance overhead of CSC establishment, we gauge the delay incurred during the instantiation of a slice, with and without CSC, across requests of three different types related to different network services (NSes). In addition, we also take into account the types of Virtual Network Function (VNF) instances required, as shown in Table 4.1. This table provides a detailed breakdown of the total resource demands for each slice request. Our findings indicate that the CSC establishment only incurs a delay of less than 9 seconds across all request types, as illustrated in Fig. 4.13. This delay constitutes a negligible fraction of the overall time required for the instantiation of the CSC-SC slice.

To provide a more in-depth analysis of the overhead introduced by the MESON layer, we conducted additional tests to measure the execution time of each step involved in the instantiation of the CSC. The results of these tests are presented in Table 4.2. Our findings reveal that the

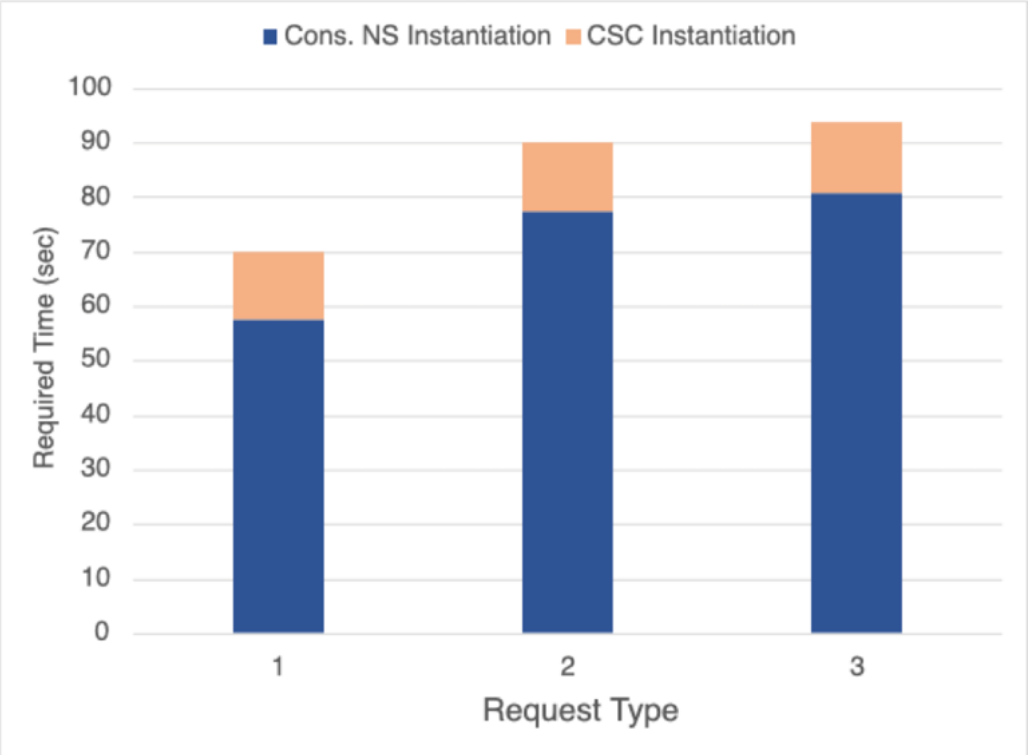


Figure 4.13: CSC instantiation overhead

time required for CSC network creation and CSC VNF instantiation are the major contributors to the overall delay, accounting for approximately 95% of the total CSC instantiation time, with an average of 8 seconds per step. It is worth noting that both these processes are executed by OpenStack, rather than MESON per se.

In contrast, MESON, and more specifically, the CSC Optimizer, is responsible for information retrieval related to the slices of CSC-SP and CSC-SC, as well as executing calls to the lower layers (VIM/NVFO) to initiate CSC instantiation. Our tests demonstrate that the delays incurred by these MESON-related processes are negligible for all types of CSC requests, resulting in a minimal overhead on slice instantiation.

In conclusion, our performance tests reveal that the MESON layer has a minor impact on the establishment of the CSC, with CSC instantiation occurring in a timely manner. We believe that this information will be useful in optimizing system performance and improving the overall efficiency of the network.

Table 4.2: CSC instantiation breakdown

CSC instantiation steps	Execution Time (s)			
	Average	**Min**	**Max**	**SD**
Tenant data retrieval	0.0044	0.0042	0.0047	0.0001
VNF OSM data retrieval	0.1762	0.1725	0.1798	0.0020
Cloud-init configuration	0.1762	0.1725	0.1798	0.0030
Cross-layer API calls	0.186	0.1841	0.1917	0.0028
CSC network creation	8.303	8.07	8.96	0.3137
CSC VNF instantiation	7,68	6,91	8,9	0.709
Total	**16.34**	**15.37**	**18.06**	**0.82**

4.4.1.2 Scalability tests

In addition to our performance tests, we also conducted scalability tests on two crucial processes involved in the coordinated CSC instantiation: CSC service discovery and edge PoP selection. The Service Registry component handles CSC service discovery, while the edge PoP selection is performed by the respective component at the MESON layer. Both these processes are triggered when a request for the consumption of a service via CSC is made.

Our tests were conducted on the coordinated CSC case, as it involves a higher degree of coordination and complexity. Similar to our performance tests, we sought to determine the scalability of these processes in the face of an increasing number of requests.

To evaluate the scalability of the MESON-layer operations, we conducted tests on a diverse range of edge Points of Presence (PoPs) varying from 10 to 100. For these tests, we populated the Service Registry's database with a sufficient number of service instances stored as AppDs. Furthermore, 100 AppDs were assigned to each registered edge PoP to simulate CSC-enabled services.

As shown in Fig. 4.14, the time required for service discovery and edge PoP selection falls within the range of 10 to 40 ms. It is worth noting that the delay incurred for both operations combined is considered negligible, given that slice instantiation typically takes tens of seconds to complete. Our findings indicate that the scalability of service discovery is linear, whereas the time required for edge PoP selection increases polynomially with an increasing number of

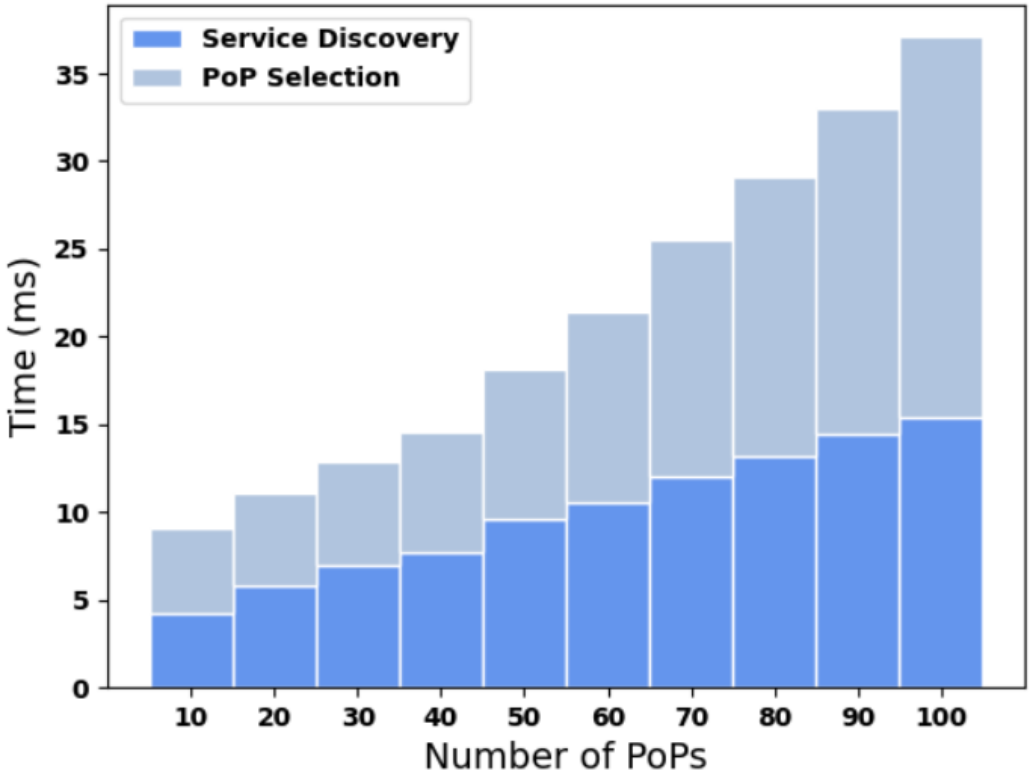


Figure 4.14: CSC instantiation overhead

PoPs. This is due to the use of Multi-Criteria Decision Making (MCDM) algorithms employed for PoP ranking.

In conclusion, our scalability tests demonstrate that delays incurred during service discovery and edge PoP selection are insignificant compared to the overall time required for CSC slice instantiation. These results provide useful insights into the system's performance, which can be used to optimize its efficiency and improve its overall scalability.

4.4.2 CSC Orchestrator Evaluation

4.4.2.1 Instantiation Overhead with Different CSC Scenarios

In the following, we discuss our experimental setup for the baseline and the shared CSC scenario. Fig. 4.15 illustrates the VNF chain, deployed in the intermediate slice for the baseline scenario. The service chain consists of three VNFs for security and management of the ingress/egress traffic. More specifically, we employ Vuurmuur [53] for the firewall, we utilize the open-source

Suricata [54], capable of real-time intrusion detection and network security monitoring, and also use Monitorix [55], which is an open-source tool for monitoring services and system resources. All VNFs utilize the same amount of resources *i.e.*, 2 vCPUs, 2GB RAM, and 20GB of storage.

For the shared CSC scenario, we consider two CSCs (*i.e.,* between two pairs of CSC-enabled Consumers and Providers). Each CSC is established through an individual CSC slice. However, unlike the baseline scenario, we permit the sharing of selected NSDs between the two (intermediate) CSC slices. The shared NSD, which, in our case, is associated with a single VNF, is reused by the second CSC slice, obviating the need for the deployment of another VNF instance. As such, there is a great potential for both instantiation delay and resource savings, since services of high demand can be provisioned with a smaller number of instances.

Although in our tests we utilized the same service chain (*i.e.,* firewall-IDS-monitoring) for both CSCs, VNF sharing may be as well enabled among different service chains that have common VNFs. For our experimental purposes, we run tests with two variants of the shared CSC scenario: (i) one VNF (*i.e.,* IDS) shared among the two CSC slices, and (ii) two VNFs (*i.e.,* firewall, IDS) shared among the two CSC slices.

In order to compare the performance between these CSC scenarios, we measure the delay incurred till the CSC slice has been instantiated and the peering between the two communicating slices is established and fully functional. This essentially includes the time spent for VNF instantiation, chaining, and the bridging of the CSC slice with the two communicating slices. The reported CSC instantiation times are obtained through parsing the instantiation logs from the VIM. Instantiation times are measured from the moment a CSC request is initiated, till all required VNF instances have been built, chained, and bridged to the slice of the Provider and the Consumer. All tests are executed across 30 runs, for each CSC scenario.

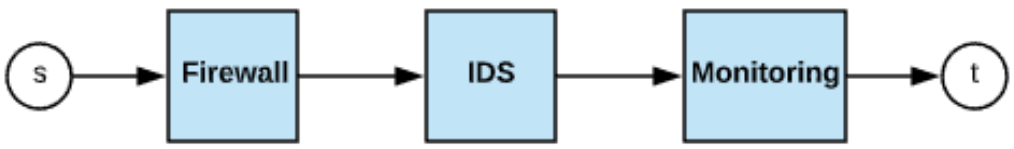


Figure 4.15: Service chain deployed in the CSC slice.

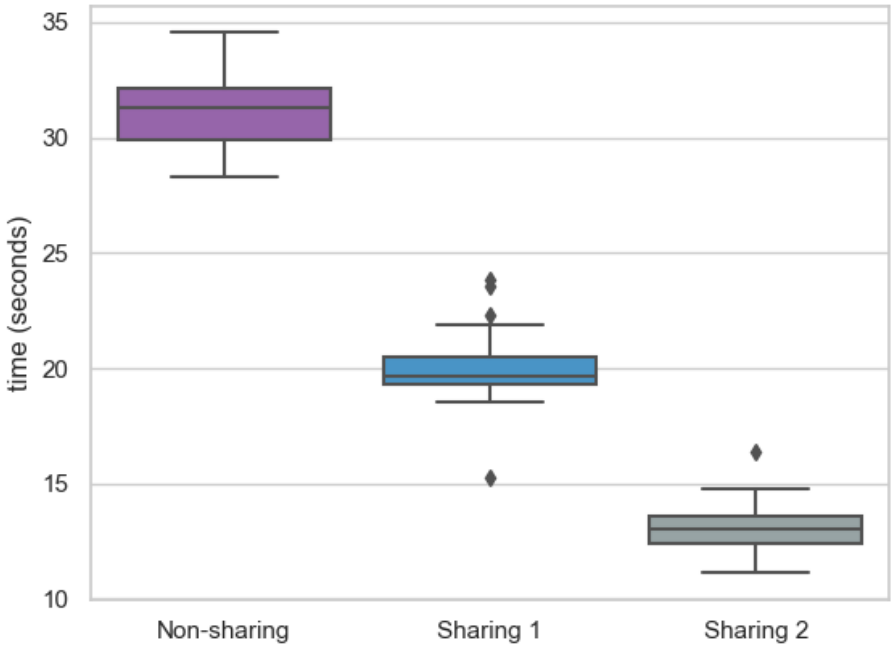


Figure 4.16: CSC instantiation time for the baseline (non-sharing) scenario, the scenario with 1 shared VNF (sharing 1), and the scenario with 2 shared VNFs (sharing 2).

We hereby discuss the measured CSC instantiation times for the various CSC scenarios. Besides CSC instantiation, we also present various statistics to gain more insights from our experimentation. Fig. 4.16 illustrates box plots for the (i) baseline (non-sharing) scenario, (ii) 1 shared VNF (Sharing 1), and (iii) 2 shared VNFs (Sharing 2). As shown in this plot, VNF sharing yields significantly lower instantiation time, since the respective shared VNFs are already deployed; hence, only a subset of the service chain needs to be instantiated. We further observe that the span in the measured times is shorter in the two variants of shared CSC. This is attributed to the following two factors: (i) the reduction of the instantiation time, since fewer VNFs need to be instantiated in the shared scenarios, and (ii) the reduction of instantiation times observed in cases where an already existing VNF is re-spawned in the VIM.

With respect to the latter, we note that individual instantiation times for each VNF appear to be reduced by up to 24% in the case of the monitoring VNF, compared to non-sharing. This stems from image caching and/or reusing by the VIM[4] and is not a factor pertaining directly

[4]https://docs.openstack.org/nova/latest/admin/image-caching.html

Table 4.3: Statistics of CSC instantiation

Non-Shared Scenario:

	Firewall	**IDS**	**Monitoring**	**Duration**
Average (sec)	17.34	17.28	16.21	31.26
Standard Deviation (sec)	1.95	1.38	1.44	1.70
MAX (sec)	21.84	19.54	20.32	34.54
MIN (sec)	14.55	14.59	14.22	28.32

Shared Scenario 1:

	Firewall	**IDS**	**Monitoring**	**Duration**
Average (sec)	15.64	0	14.63	20.06
Standard Deviation (sec)	1.72	0	1.14	1.59
MAX (sec)	20.46	0	18.18	23.81
MIN (sec)	13.90	0	12.84	15.23

Shared Scenario 2:

	Firewall	**IDS**	**Monitoring**	**Duration**
Average (sec)	0	0	13.04	13.04
Standard Deviation (sec)	0	0	1.15	1.15
MAX (sec)	0	0	16.39	16.39
MIN (sec)	0	0	11.12	11.12

to the CSC. This issue has a significant impact when comparing any of the two shared CSC variants with the baseline scenario. On the other hand, image caching is not expected to make a difference between the two shared CSC variants, since all images are already instantiated in the first CSC slice, and thus, no gains can be directly attributed to image caching. As such, the gap in the instantiation time between one and two shared VNFs is lower (compared to baseline vs. sharing), as shown in Table 4.3.

The time required for bridging is sampled and found to have a minimal contribution to the total instantiation time, with a minimal standard deviation. Given that VNF instance spawning comprises the dominant factor for CSC setup, we expect lower instantiation times with VNFs assigned to compute nodes hosted on different servers. Overall, our experimental results indicate that CSC can be established in the order of seconds, with the potential to further reduce this delay through VNF sharing among CSC slices.

4.4.2.2 CSC Latency gains

Next, we investigate the latency inflation with different forms of CSC. To this end, we execute three different scenarios: (i) services are co-located in the same DC and can communicate through an intermediate CSC slice (*i.e.,* MESON approach), (ii) services are deployed on the same DC and can communicate directly without any provision for security or resource isolation, and (iii) services are located in different DCs on the same availability zone.

Fig. 4.17 depicts the average latency with a total of 100 ICMP messages across 10 runs, for all three aforementioned CSC scenarios. Direct communication between services that reside in the same DC incurs the lowest latency among the three CSC scenarios under consideration. However, this approach is inherently insecure, since this form of communication does not provide any isolation between slices. Therefore, the underlying VIM can not guarantee secure communication between tenants nor can reap the benefits of network slicing for the services deployed therein. In comparison, the CSC scenario promoted by this work (*i.e.,* co-located slices with a secure CSC) yields latency almost on par with the direct communication (*i.e.,* best case in terms of latency). Therefore, our approach of instantiating an intermediate CSC slice with plugged-in processing functionality (*e.g.,* traffic inspection, monitoring) does not lead to any perceptible latency inflation, while providing the means for secure and policy-compliant cross-service communication. Lastly, significant latency is incurred when communication is established among slices located in different DCs on the same availability zone. Such latency inflation can have an adverse impact on latency-sensitive services (*e.g.,* location-based services) that require the consumption of another service instance (deployed within a slice). As such, the need for the discovery and consumption of co-located service instances is crucial.

4.4.2.3 Co-location Gains

The proposed CSC Orchestrator enforces the co-location of the CSC slice and the consuming service. This co-location can lead to gains, such as the minimization of the communication overhead inside the DC, the reduction of latency, as well as the increase of throughput. The

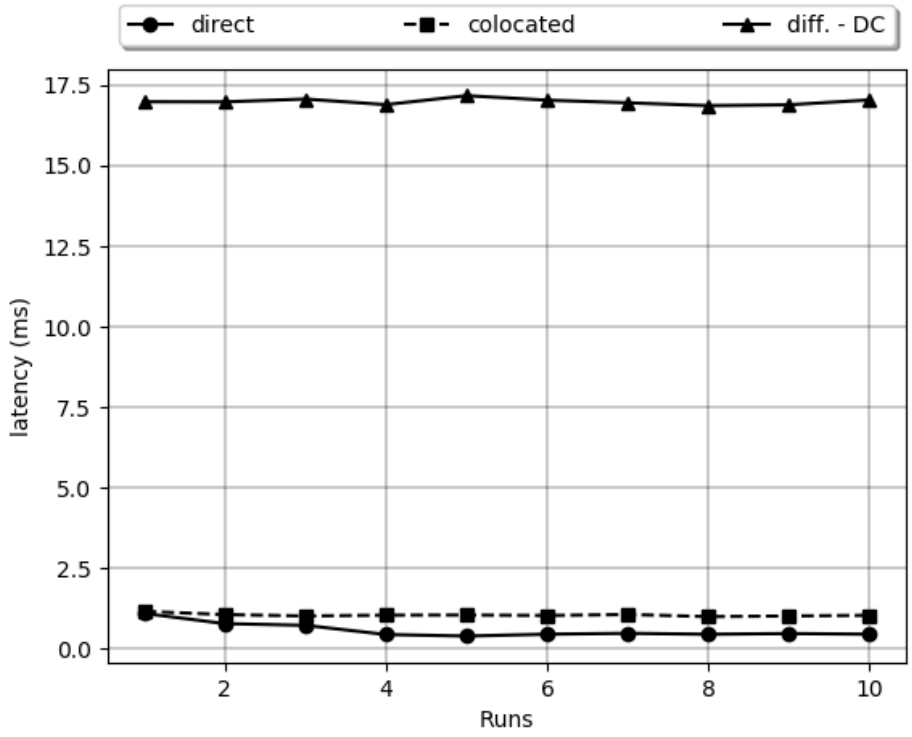


Figure 4.17: Latency with different CSC scenarios.

CSC Orchestrator achieves this co-location leveraging on the *near-host* filter of the underlying VIM. To examine the potential gains from co-location, we run two different experiments. In both experiments, the basic setup utilizes a VNF acting as a firewall, implemented with Click, which resides within the CSC slice. Using a UDP traffic generator, we send packets of 1024 bytes from the consuming service to the providing service. In the first experiment, the CSC slice is co-located with the consuming services in the same compute node, whereas the providing service is deployed in a different compute node. In the second experiment, the providing service, the consuming service, and the CSC slice reside in separate compute nodes.

Fig. 4.18 confirms our intuition that co-locating the CSC slice with the consuming service leads to approximately 2% increased throughput performance. While someone could argue that this gain is negligible, we stress on the fact that edge computing environments are commonly resource-constrained and, as such, even small gains can be significant with respect to services with ultra-low latency and high throughput requirements [56].

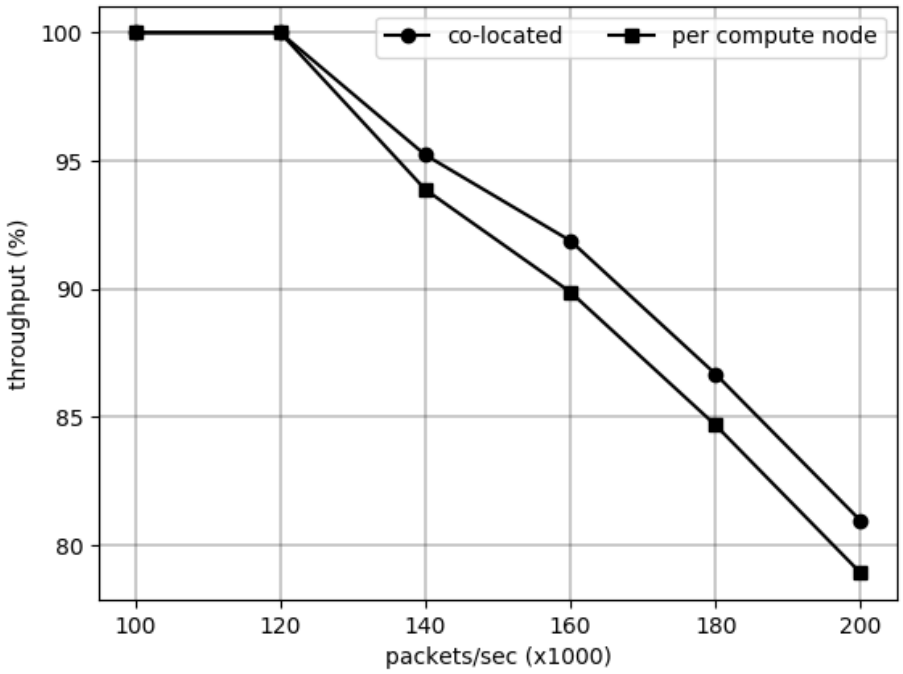


Figure 4.18: Throughput gains due to service co-location.

4.5 Related Work

In this section, we present a comprehensive overview of other noteworthy projects in the domain of network slicing. These projects have made significant contributions to advancing the concept of network slicing and its practical implementations.

4.5.1 Network Slicing Architectures

4.5.1.1 NECOS

NECOS [57] focuses on cloud slicing and aims to create and manage slices of infrastructure over multiple providers, which may include network infrastructure in addition to cloud infrastructure. For this purpose, NECOS has implemented a slicing platform called Lightweight Software-Defined Cloud (LSDC). The architecture of this platform is depicted in Figure 4.19.

NECOS network slicing is performed by the Virtual Infrastructure Manager (VIM), and each

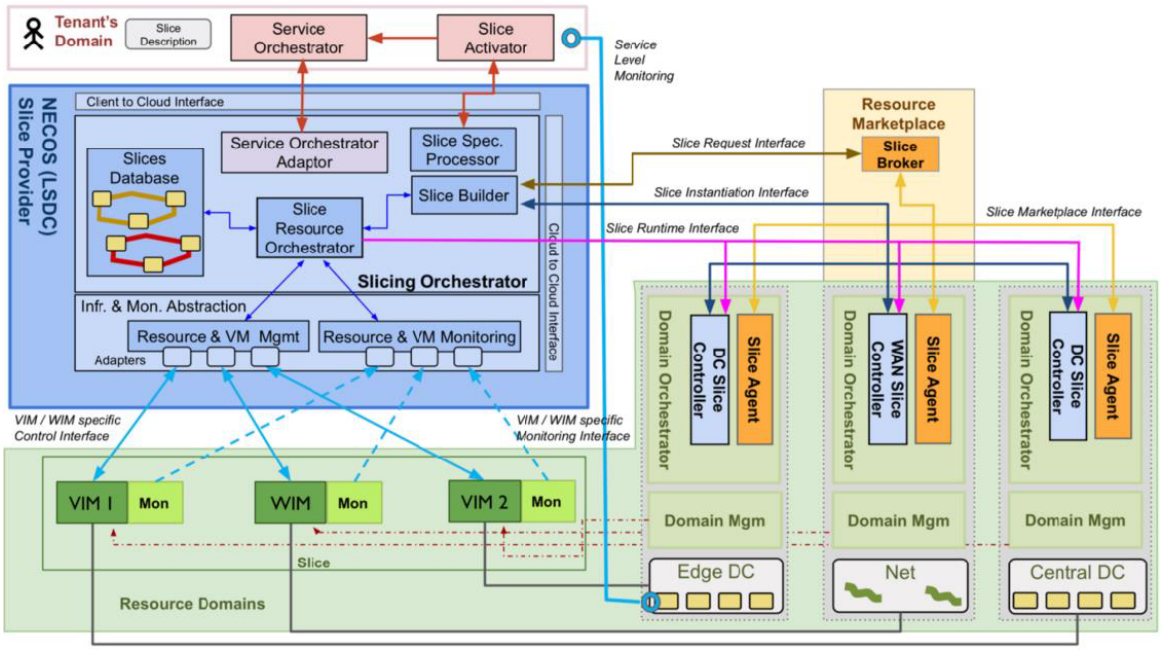


Figure 4.19: NECOS architecture

installed slice contains its own autonomous VIM to facilitate its management by the slice tenant. NECOS decouples service orchestration from the installation and management of each slice, assuming that the tenant will later install a preferred service orchestrator. However, it provides support for integrating the service orchestrator into the monitoring subsystem of the LSDC platform.

NECOS provides a Marketplace [58] for searching and reserving computing and networking resources to create slices over multiple infrastructures. The Broker in the Marketplace segments each slice into parts and searches for the most suitable resources for each part from individual providers, ultimately forming the final slice by connecting the selected parts. Part of each slice also consists of network resources reserved from network providers to interconnect the slice's parts.

For the description and specification of components and individual parts, an information model is used. The same model is employed for describing sliced infrastructures, including entities for servers, network devices, and their respective elements (*e.g.,* interfaces). NECOS has not examined any form of granular communication, focusing instead on the creation and management of autonomous slices over different cloud infrastructures. Additionally, it places more emphasis

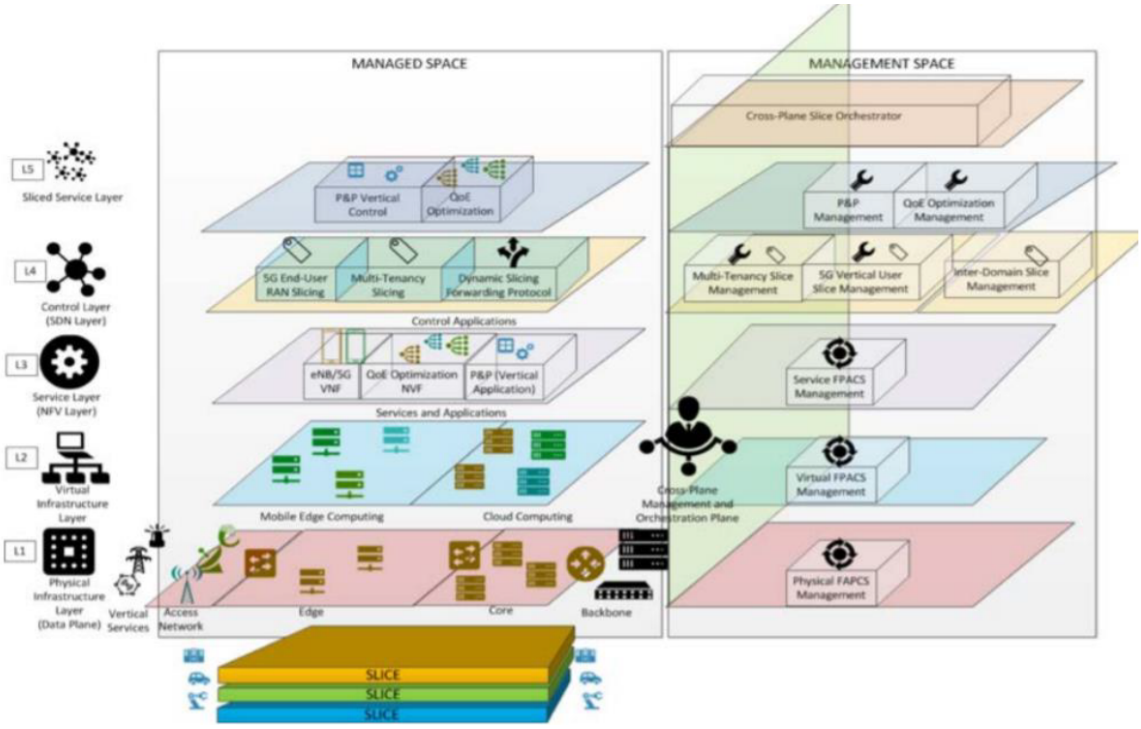


Figure 4.20: SLICENET Architecture

on sliced infrastructures at the core of the network and less on the edges.

4.5.1.2 SLICENET

SLICENET is primarily centered around end-to-end network slicing of the existing infrastructure. It offers a comprehensive framework for managing slices within a multi-domain, virtualized 5G network environment with numerous tenants. By leveraging virtualization technologies, SLICENET enables efficient resource allocation, service orchestration, and dynamic management of network slices across diverse domains.

The slicing architecture proposed by SLICENET is depicted in Figure 4.20. It illustrates the hierarchical structure of slice management, showcasing the relationships between the management space and the managed spaces. This architecture facilitates the effective coordination and control of resources, services, and policies across the entire slice lifecycle.

SLICENET aims to provide flexible and scalable solutions for network slicing, catering to the diverse requirements of different verticals and applications. By offering granular control and customization options, SLICENET empowers network operators to deliver tailored services

while maximizing resource utilization and ensuring the quality of service for each slice.

Following a hierarchical approach based on the layer names as indicated in Figure 4.20 (L1-L5), the first layer corresponds to the physical infrastructure level, reflecting the physical deployments. The second layer (L2) represents the Virtual Infrastructure level, primarily focused on 5G implementations. It leverages technologies from the multi-access edge computing paradigm to utilize edge resource virtualization. The third layer (L3) encompasses all services, particularly virtual network function (VNF) services of SLICENET, and operates on the underlying infrastructure. The fourth layer (L4) is the control layer, decoupled from the software-defined networking layer to enable centralized control of physically distributed services. The fifth and final layer (L5) is the sliced network services layer, introduced in SLICENET to support multiple tenants. This layer provides the functionality of a control plane and is customized for each tenant of vertical services, differing from the common control plane for all infrastructure users. This layer hosts services directly offered to vertical industries to meet their specific needs.

4.5.1.3 SONATA

5G SONATA focused on enabling flexible software network programming and optimizing their applications. From the perspective of 5G SONATA, a slice represents a consolidated set of resources composed of virtual network functions that can be utilized within an end-to-end network service. Slices are comprised of multiple isolated resources, allowing for the creation of logically separated network partitions. A slice that utilizes network, computing, and storage resources is considered the basic programming unit. Figure 4.21 illustrates the proposed architecture of SONATA for deployment and orchestration purposes.

The architecture of the SONATA platform 4.22 consists of a set of components that run as microservices and interact with each other. In terms of network slice management, the Network Slice Manager handles the installation of network slices that contain multiple interconnected network services with specific requirements defined by service level agreements (SLAs). By introducing an additional layer above the network services and creating network slices, SONATA can provide better and more isolated services to end-users. The Network Slice Manager has the

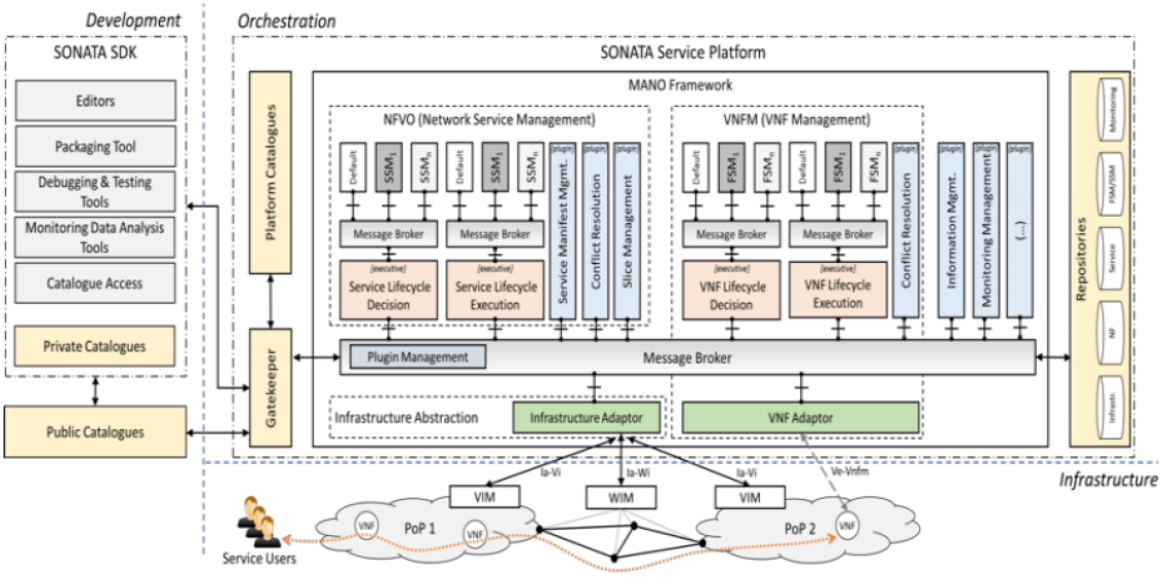


Figure 4.21: SONATA Architecture Overview

capability to deploy services that form a network slice in a single or multiple scenarios. The latter allows for the distribution of services across different virtualized infrastructures based on resource availability and the requirements of each service. Additionally, a service can be shared among multiple network slices, enabling SONATA to perform more efficient resource allocation.

4.5.1.4 Differentiation of MESON

Regarding the aforementioned standards and architectures, which primarily aim at the management and orchestration of sliced infrastructures, MESON focuses on a specific aspect of slicing, namely cross-slice communication. With the goal of orchestrating cross-slice communication, MESON is closely aligned with several standards, such as ETSI NFV MANO (regarding orchestration of virtualized network functions) and ETSI MEC (regarding the information model for describing CSC services), as well as the approaches followed by standardization and certification organizations for transitioning to next-generation network services.

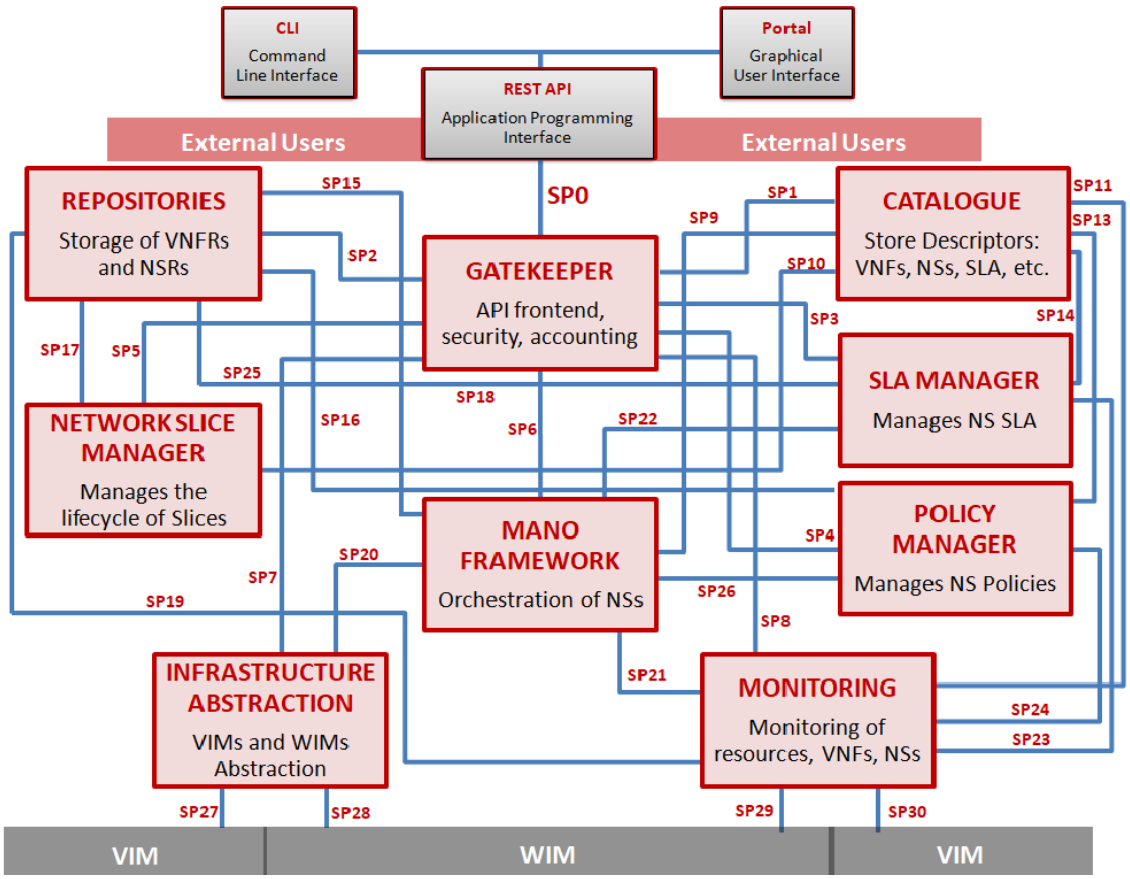


Figure 4.22: SONATA Platform Overview

4.6 Conclusions

In this chapter, we delved into the comprehensive architecture, implementation details, and rigorous experimental evaluation of the MESON platform. This cutting-edge platform serves as a facilitator for Cross-Slice Communication (CSC) and seamless service interactions among various slice tenants. With CSC being established in a meticulously optimized manner, the platform goes beyond mere co-location of services on the same edge Point of Presence (PoP) by taking into account resource availability and even considering service co-location on the same server.

A notable highlight of the MESON platform lies in its ability to instantiate dedicated slices specifically designed for CSC. This intricate process is skillfully orchestrated in conjunction with the deployment of the CSC-SC (Cross-Slice Communication Slice Controller) slice. By ensuring dedicated slices for CSC, the platform achieves heightened efficiency and efficacy in

enabling seamless communication and collaboration among services.

The experimental evaluation conducted on the MESON platform effectively demonstrates that the establishment of CSC introduces minimal performance overhead when compared to the time required for slice instantiation within the Virtualized Infrastructure Manager (VIM). Through meticulous testing conducted on a larger scale, it is evident that activities such as service discovery and edge PoP selection, even with a substantial number of edge PoPs reaching up to a hundred, incur delays within the range of tens of milliseconds. These findings solidify the feasibility and practicality of the MESON platform, showcasing its potential for real-world implementation.

Overall, we firmly believe that Cross-Slice Communication (CSC) stands as a key enabler for the advent of next-generation Service Marketplaces. In this landscape, network services are no longer confined to their own standalone functionalities. Instead, they are empowered to seamlessly consume and interact with other potentially co-located service elements, resulting in an unparalleled quality experience for end-users. Moreover, to further amplify the ability to discover and utilize CSC-enabled services across an extensive array of (edge) PoPs, leveraging state-of-the-art Artificial Intelligence (AI) and Machine Learning (ML) methods can unlock new frontiers of efficiency and effectiveness.

In summary, the MESON platform, with its robust architecture, seamless CSC establishment, and emphasis on optimized service interactions, paves the way for a future where dynamic Service Marketplaces thrive. By harnessing the potential of CSC and incorporating intelligent algorithms, the platform has the potential to reshape the service landscape and deliver unprecedented quality experiences to users across diverse domains.

Chapter 5

Fine-Grained Resource Orchestration

5.1 Motivation

Middleboxes comprise an indispensable part of the network infrastructure, performing a wide range of network processing operations, such as intrusion detection, load balancing, network address translation (NAT), and redundancy elimination [59, 60]. Network Function Virtualization (NFV) decouples network functions from middleboxes, by facilitating their deployment on commodity servers in the form of virtual network functions (VNFs) [49, 14]. This yields, among others, cost-efficiency, scalability, and flexibility, which comprise features aligned with the endeavor of cloud operators towards agile and autonomous network management. However, to fulfill the potential of NFV, certain challenges need to be addressed, with a prominent one being *resource optimization* [61]. More precisely, optimized resource allocation can lead to high and predictable packet processing performance, empowering commodity servers to host VNFs associated with demanding service Key Performance Indicators (KPIs). At the same time, resource optimization can minimize resource wastage (*e.g.,* reducing the amount of *empty* CPU cycles), enabling datacenter operators to better monetize their infrastructure.

VNF resource allocation has been studied at various levels. In particular, several studies (*e.g.*, [62, 63, 64]) tackle the VNF placement problem over Points-of-Presence (PoP)-level topologies. In such a setting, the resource allocation unit is the PoP itself, which exposes

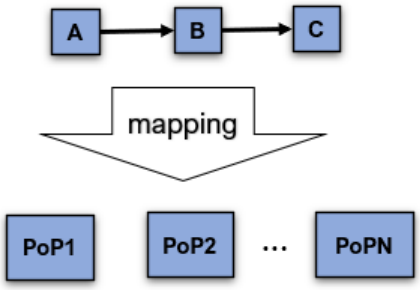


Figure 5.1: SFC embedding at PoP level

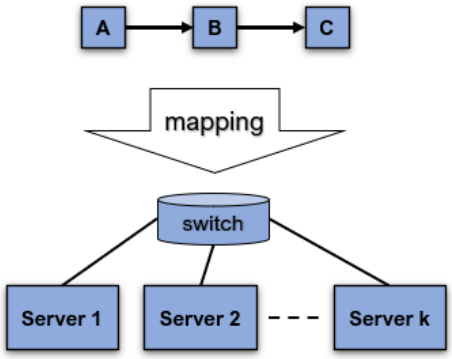


Figure 5.2: SFC embedding at server level

its available CPU (which is presumably an aggregate of the available CPUs of the underlying servers) to a decision maker. A high-level illustration of this approach is depicted in Fig. 5.1. In addition, other studies (*e.g.*, [65, 45, 66]) propose solutions for the VNF placement problem within a single PoP, commonly expressed by a datacenter (DC) network topology, which comprises the NFV infrastructure (NFVI). In this case, the resource allocation unit is a server, which, following a similar abstraction as before, exposes a single value to express its available CPU (which typically corresponds to the aggregated computing capacity across all server's CPU cores). This SFC embedding approach is illustrated in Fig. 5.2

With respect to VNF (or VNF chain) embedding, the two preceding approaches complement each other, enabling the selection of the most suitable NFVI PoP, followed by the placement of the VNF(s) onto the assigned PoP's servers (subject to server and link capacity constraints). Nevertheless, resource optimization entails another challenging aspect, commonly overlooked by these approaches, *i.e.,* the intra-server resource assignment problem, which consists of the optimized assignment of VNFs onto the CPU cores of the selected server. We deem that this last-level resource assignment problem is of crucial importance for the following main reasons. First, packet processing workloads exhibit diverse resource profiles (*i.e.,* CPU-intensive, memory-intensive), which require particular care in their assignment to CPU cores in order to avoid CPU cycle wastage. Second, despite the various advancements in network interface cards (NICs) that enable various forms of processing offload (*e.g.,* TCP segmentation offload, packet classification using mechanisms, such as SR-IOV) as well as packet I/O optimizations

(*e.g.,* Intel's DPDK), the CPU still has the burden of performing computationally-intensive operations, stemming from various forms of deep packet inspection that may need to be carried out at line rate.

5.2 Contributions

The contributions of this research study are split into two categories: (i) directions to schedulers for efficient intra-server VNF embedding, and (ii) impact of different allocation strategies on performance.

Providing Directions to Schedulers for Efficient Intra-Server VNF Embedding. In this research, we make a significant contribution by offering valuable directions and guidelines to resource schedulers for achieving efficient intra-server VNF embedding within NFV infrastructures. By thoroughly investigating the complexities and challenges associated with resource allocation, we aim to equip schedulers with actionable insights to enhance the overall performance and resource utilization of VNF deployments. Through our experimental study, we identify and evaluate various allocation strategies, taking into account factors such as CPU and memory demands, workload characteristics, and packet forwarding performance. By analyzing the strengths and weaknesses of different embedding techniques, we shed light on the optimal approaches that schedulers can employ to achieve efficient resource allocation within a server.

Our research findings can empower schedulers to make informed decisions when it comes to placing and allocating VNFs within a server. By considering the impact of workload heterogeneity, inter-core packet transfers, and the trade-off between VNF consolidation and resource utilization efficiency, schedulers can develop intelligent and adaptive resource management frameworks that enhance service performance and maximize resource utilization.

Assessing the Impact of Different Allocation Strategies on Service Chain Performance. Another significant contribution of our research is the evaluation and assessment of the impact of different allocation strategies on the performance of service chains within NFV

infrastructures. Service chains, consisting of a series of interconnected VNFs, play a crucial role in delivering specific network services. By studying various allocation strategies and their implications on packet forwarding performance, we provide valuable insights into how different approaches affect the overall performance and quality of service of the deployed service chains. This includes quantifying the impact of inter-core packet transfers, examining CPU core utilization efficiency under different workloads, and assessing the trade-off between VNF consolidation and resource utilization efficiency.

Our findings help shed light on the performance trade-offs and considerations that schedulers need to take into account when allocating resources for service chains. This information enables schedulers to make informed decisions regarding allocation strategies, leading to improved service chain performance, reduced latency, and enhanced user experience.

Overall, our research provides valuable directions to schedulers on achieving efficient intra-server VNF embedding, while also assessing the impact of different allocation strategies on service chain performance. By incorporating our insights, schedulers can optimize resource allocation, improve service delivery, and ensure efficient utilization of resources within NFV infrastructures.

5.3 Problem Description

In this chapter we stress on the need for fine-grained resource allocation in NFVI, promoting the CPU core as the resource allocation unit (instead of the server). As such, this chapter is focused on this last-level VNF-to-CPU assignment problem, which we consider as complementary to the existing literature on NFV resource allocation. In particular, we highlight the combinatorial nature of this problem and also uncover that VNF-to-core assignment combinations have a major impact on the throughput achieved by VNFs in a service chain, using an experimental setup with diverse packet processing workloads running in Click Modular Router [52]. Commencing with simple experimental scenarios and gradually progressing to more complex ones, we shed light on CPU core allocation implications and gain various insights that can

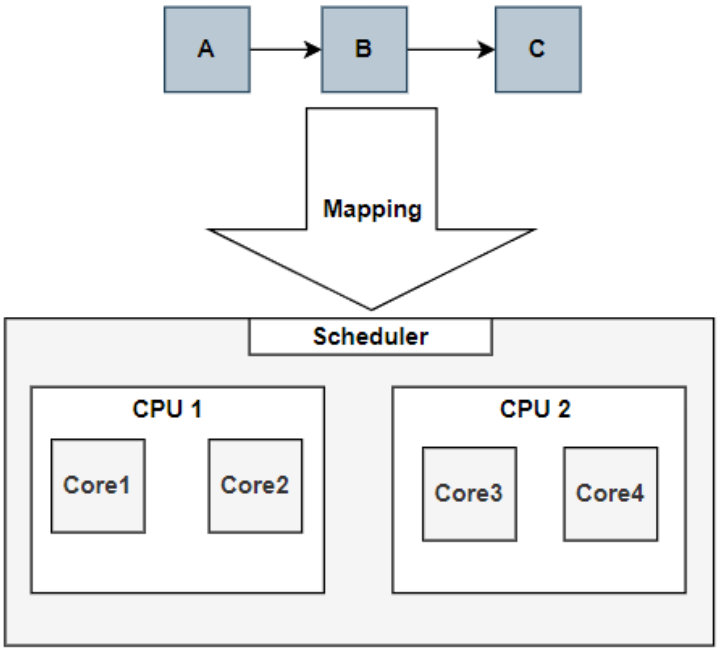


Figure 5.3: Intra-Server SFC embedding.

be of great value to a resource scheduler within a server (*e.g.*, hypervisor scheduler).

5.4 Methodology

Our main goal in our experimental study is to explore the following question: *Given a VNF chain with its resource demands and a number of CPU cores within a server, what is the VNF-to-core assignment that yields the highest packet forwarding performance?*

Intuitively, co-locating adjacent VNFs within the same server or rack in an SFC has several benefits. One significant advantage is the minimization of generated traffic, resulting in the conservation of bisection bandwidth, particularly in fat-tree or leaf-spine datacenter network topologies. By co-locating VNFs that are connected sequentially in the VNF-graph, the need for inter-server communication is reduced, leading to improved network efficiency and reduced latency.

However, our main consideration lies in the splitting of VNF chains across multiple CPU cores. For instance, a scenario may arise where a chain of three VNFs needs to be placed on only two

CPU cores. This raises important questions that need to be addressed in order to optimize the SFC embedding process.

Firstly, we need to examine the impact of inter-core transfers on VNF performance. When VNFs are assigned to different CPU cores, communication between them may require inter-core transfers, which can introduce additional latency and overhead. Understanding and quantifying this impact is crucial for making informed decisions regarding VNF to CPU core assignment.

Secondly, CPU core allocation with heterogeneous workloads needs to be considered. VNFs within an SFC often exhibit diverse resource profiles, with varying computational requirements and resource demands. Efficiently allocating CPU cores to accommodate these heterogeneous workloads becomes crucial for achieving optimal performance and resource utilization.

In the following section of this chapter, we provide answers to both of these critical questions. By investigating the impact of inter-core transfers on VNF performance and exploring CPU core allocation strategies for heterogeneous workloads, we aim to identify favorable CPU allocations, even in more complex SFC embedding scenarios. Through our research findings, we can offer insights and guidelines for making informed decisions on VNF-to-CPU core assignment, ultimately enabling improved SFC performance and resource utilization.

5.4.1 Experimental Setup

For our experimentation, we utilize an Emulab testbed [67]. More specifically, we run our experiments on three commodity servers, each one with three Intel E5-2630 Hasswell-based CPUs and 64GB of main memory. Each CPU has eight cores running at 2.4GHz and is equipped with a dedicated L1 (32KB) and L2 (256KB) caches. The cores of each CPU share a 20MB L3 cache. The servers under test are further equipped with a quad-port Intel X710 10G NIC. For our performance measurements, we rely on the following processing workloads, which are implemented using Click [68].

Click Router is a software architecture for building flexible and configurable packet processing systems, such as routers, switches, and firewalls. Click provides a wide range of packet pro-

cessing modules (*e.g.,* packet I/O, schedulers, queues, classifiers, table lookup, etc.), termed as *elements*, which can be combined in the form of directed acyclic graphs. A Click graph can contain chains of pipelined elements, which can be scheduled to run on different threads on specific cores, enabling parallelism and core affinity. When running in the kernel, Click provides a separate scheduler for each Click kernel thread, which schedules the execution of the schedulable chains of elements assigned to the thread. Alternatively, Click also provides a user-level version, which can be coupled with packet I/O engines to accelerate packet forwarding performance (*e.g.,* FastClick [69] employs DPDK for this purpose).

We implemented two different types of VNFs:

Encryption, which is based on the Advanced Encryption Standard and SHA1 for hash-based message authentication. Furthermore, we utilize the Encapsulation Security Payload (ESP) IPsec tunneling mode, which increases packet size by an extra ESP header. In our experimentation, encryption is employed as a CPU-intensive workload [70].

NAT, which is implemented using the *IPRewritter* Click element. Since NAT is purposed as a memory-intensive workload, we populate a NAT table with approximately 65K entries, stored in main memory.

Our experimental setup consists of (i) a traffic generator, (ii) chains of encryption and NAT instances, and (iii) a sink, each one implemented on a separate server. All three servers are connected via 10G links. For traffic generation, we utilize Click's *FastUDPSource*, configured to generate packets of 1200 bytes. The sink, also implemented using Click, measures the achieved packet forwarding rates.

5.4.1.1 Terminology

For simplicity, we use C and M to refer to the encryption and NAT, respectively. In this context, we use a right arrow pointing from one VNF to another to express VNF chains. For example, $C \rightarrow M$ denotes a VNF chain, at which incoming packets have to initially be processed by C and then by M. Utilizing these two VNFs, we define several CPU core allocation scenarios

with a multitude of VNF chains of length up to four, in the rationale that longer chains are less likely to be assigned into a single server. Furthermore, we use a hyphen (-) between VNFs to denote the allocation of different CPU cores within a single socket, whereas a hyphen between two parentheses indicates the allocation of different CPUs. For instance, given the chain $C_1 \rightarrow C_2 \rightarrow M_1 \rightarrow M_2$, the allocation $(C_1C_2 - M_1) - (M_2)$ implies that VNFs C_1 and C_2 are executed in the same core, M_1 in a different core of the same CPU, while M_2 has been placed on a core from a different CPU.

5.5 Evaluation Results

In this section, we delve into an intriguing and comprehensive experimental study focused on the critical aspect of intra-server resource allocation within NFV (Network Function Virtualization) infrastructures. By undertaking this study, we aim to shed light on the intricacies and challenges associated with efficiently managing and distributing resources among virtualized network functions residing within a single server.

5.5.1 Intra-Server Resource Allocation

5.5.1.1 Impact of Inter-Core Packet Transfers

Initially, we focus on quantifying to which extent inter-core transfers affect the performance of VNFs within a chain. To this end, we deploy the VNF chain $C_1 \rightarrow C_2$, and measure throughput with the following three allocations: (i) same CPU core, (ii) different cores within the same CPU, and (iii) cores from different CPUs. Since the allocations (ii) and (iii) utilize two cores, in these cases we deploy a second instance of the chain $C_1 \rightarrow C_2$, *i.e.*, $C_3 \rightarrow C_4$. As such, the achievable throughput from a single chain instance is comparable with the throughput of the allocation (i).

Fig. 5.4 confirms our intuition that avoiding inter-core packet transfers yields the highest throughput. However, there are some interesting observations that could prove useful for a

decision-making algorithm. First, at low sending rates ($\leq$ 20k packets/sec, which translates into $\leq$ 200 Mbps) there is no perceptible performance penalty due to inter-core packet transfers. In fact, this observation holds for even higher sending rates, in the case where the two cores belong to the same CPU. For higher sending rates ($\geq$ 40K packets/sec), though, throughput degradation, while using different cores, is apparent. For instance, at 40K packets/sec, the average throughput of the *same core* allocation is 80%, whereas the *diff. cores* and the *diff. CPUs* are slightly above and well below 70%, respectively.

Following a similar approach, we measure the performance of a $M_1 \rightarrow M_2$ chain. Since memory-intensive VNFs are harder to saturate, we run tests with higher sending rates. The results shown in Fig. 5.5 exhibit a similar trend with the corresponding results of the previous experiment. In particular, the different allocations yield similar levels of throughput for up to 100k packets/sec (or $\approx$1Gbps). From that point on, we observe a relatively steady difference, which is bounded at 12%, favoring the *same core* allocation.

Based on these measurements, we observe the following. For low data rates, inter-core packet transfers do not lead to perceptible throughput degradation. Instead, for high processing demands, a resource scheduler should seek to allocate cores from the same CPU, since intra- and inter-CPU placements significantly affect the overall throughput of the chain. Finally, *same core* allocations consistently result in higher throughput. We note that these observations are prevalent for both CPU- and memory-intensive VNFs.

5.5.1.2 CPU Allocation Among Heterogeneous VNFs

We hereby investigate alternative CPU core allocations among heterogeneous workloads and their impact on performance. To this end, we deploy two identical chains, *i.e.,* $C_1 \rightarrow M_1$ and $C_2 \rightarrow M_2$, on two cores of the same CPU. In particular, we consider the two alternative CPU core allocations illustrated in Fig. 5.6. The main difference between the two allocations is that in the first case, each core executes two heterogeneous workloads, as opposed to the second case where the workloads are the same.

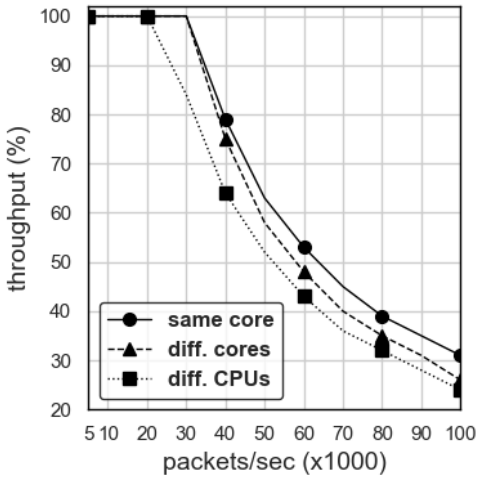


Figure 5.4: Throughput of $C_1 \rightarrow C_2$, with respect to different CPU core allocations.

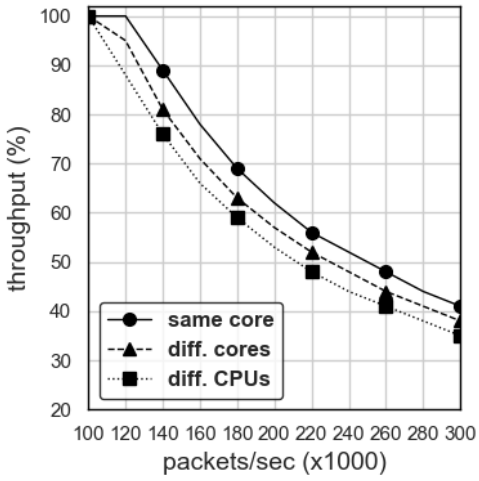


Figure 5.5: Throughput of $M_1 \rightarrow M_2$, with respect to different CPU core allocations.

According to Fig. 5.7, allocating cores to heterogeneous workloads leads to higher performance compared to workloads with similar resource profiles (*e.g.,* CPU-intensive). This result highlights the impact of resource contention, which needs to be taken into account when VNFs should share the same CPU core. More specifically, in the case of *same workloads* per core, C_1 and C_2, as well as M_1 and M_2, intensely content for computing resources (*e.g.*, clock cycles) and memory accesses, respectively. As such, bundling VNFs with mixed resource profiles (*e.g., C* with *M*) and executing them on the same core can alleviate resource contention, and, thereby, improve the VNF throughput performance. We deem this observation as a valuable input to a resource optimization method, executed by the hypervisor, for informed decisions on VNF-to-core assignments, based on VNFs' resource demands.

5.5.1.3 VNF Chain Splitting

In this experimental scenario, we deploy the chain $C_1 \rightarrow C_2 \rightarrow M_1 \rightarrow M_2$, which consists of both CPU- and memory-intensive VNFs. We assess multiple allocations given (i) two available cores, and (ii) three available cores, ultimately focusing on validating some of the previous insights that emerged, as well as gaining new ones.

Two cores. Initially, we split the VNF-chain among two CPU cores, while preserving the

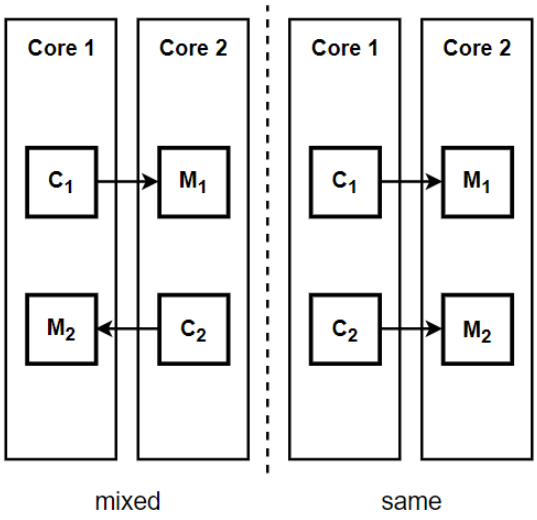


Figure 5.6: CPU core allocation among heterogeneous (left) and homogeneous workloads (right).

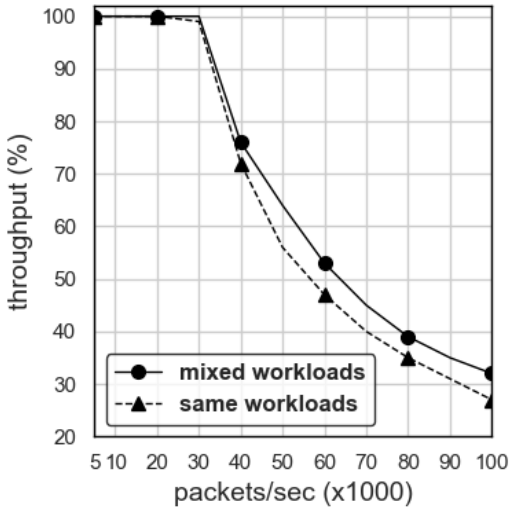


Figure 5.7: Average throughput of the two $C \rightarrow M$ chains.

same order of VNFs. The splitting alternatives, given two cores, are: (i) $C_1 - C_2M_1M_2$, (ii) $C_1C_2 - M_1M_2$, and (iii) $C_1C_2M_1 - M_2$. According to Fig. 5.8, there is performance degradation when sharing three VNFs within a single core, in contrast to the $C_1C_2 - M_1M_2$ split, which yields the highest average throughput (among the three splitting alternatives). In particular, regarding $C_1 - C_2M_1M_2$, the performance drop is attributed to the incapacity of C_2 in processing the traffic received from C_1, due to the context switching between C_2, M_1, M_2, as they are sharing the same CPU core. The alternative chain splitting that consolidates three VNFs within a single core (*i.e.,* $C_1C_2M_1 - M_2$) results in even worse performance, due to the co-location of the two CPU-intensive VNFs (the CPU-intensive VNFs are the main responsible for CPU consumption, considering the relatively low sending rates which refrain the memory-intensive VNFs from stress).

One could argue that, this analysis contradicts the previous findings regarding the perks stemming from heterogeneous workload placement. More precisely, both $C_1 - C_2M_1M_2$ and $C_1C_2M_1 - M_2$ exercise the notion of mixing workloads, yet they perform poorly compared to the $C_1C_2 - M_1M_2$ which consolidates homogeneous workloads. In fact, as we will shortly demonstrate, this is not the case. Instead, these findings highlight another aspect that need to be considered by a resource scheduler, *i.e.,* the number of VNFs running on a core.

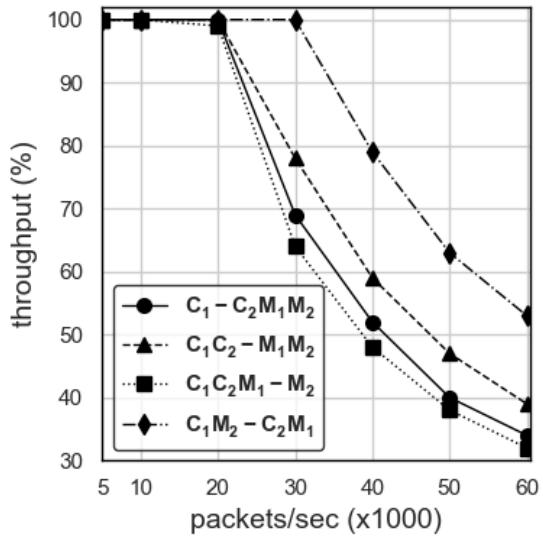


Figure 5.8: Throughput of the $C_1 \rightarrow C_2 \rightarrow M_1 \rightarrow M_2$ chain under different allocations on two cores (same CPU).

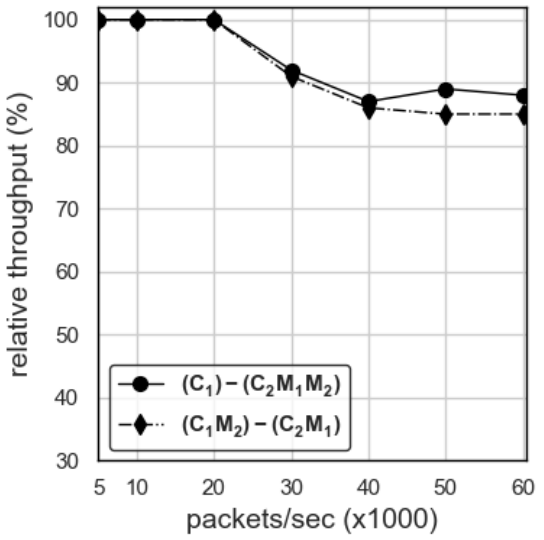


Figure 5.9: Throughput of inter-CPU allocations, relative to the respective intra-CPU allocations.

To this end, we evaluate an additional chain splitting alternative, *i.e.,* C_1M_2-C_2M_1. At first look, this splitting is not promising, since it requires multiple inter-core packet transfers. Nevertheless, this split combines workloads on the two cores in the most efficient manner (*i.e.,* C_1M_1-C_2M_2 would result in even more inter-core transfers). Surprisingly, as illustrated in Fig. 5.8, C_1M_2-C_2M_1 is by far the most efficient splitting of this chain, given two CPU cores. The main insight here is that mixing workloads, while keeping CPU core utilization even (*i.e.,* two VNFs per core), is performance-wise more critical than minimizing inter-core transfers.

However, the above inter-core transfers pertain to cores within the same socket. Therefore, we additionally examine to which extent such inter-core transfers affect throughput when cores reside in different CPUs. To this end, we consider two different allocations, *i.e.*, C_1M_2-C_2M_1, and $C_1 - C_2M_1M_2$, which are of particular interest. The former (which performed the best in the previous experiment) incurs two inter-core transfers, whereas the latter is split on the most intensive communication edge. According to Fig. 5.9, both splittings yield ≈15% throughput drop (at steady state), relatively to their respective intra-CPU placements. We note that the rest of the splittings exhibit a similar behavior; as such, the respective plots are omitted for brevity. Overall, C_1M_2-C_2M_1 is promoted as the preferable splitting alternative, even when the two cores belong to different CPUs (since the performance drop is low and almost the same for

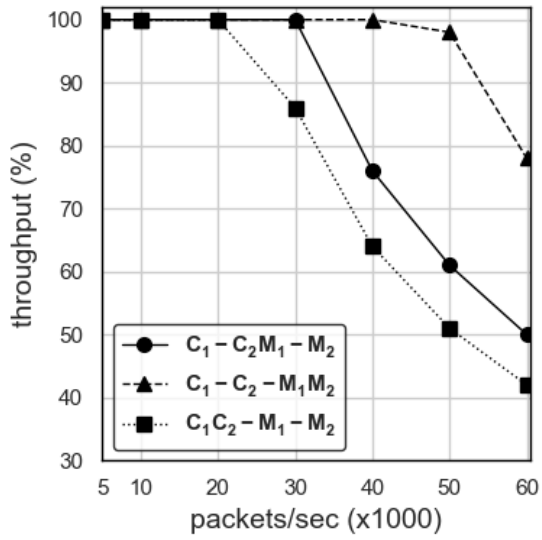


Figure 5.10: Throughput of the $C_1 \rightarrow C_2 \rightarrow M_1 \rightarrow M_2$ chain under different allocations on three cores (same CPU).

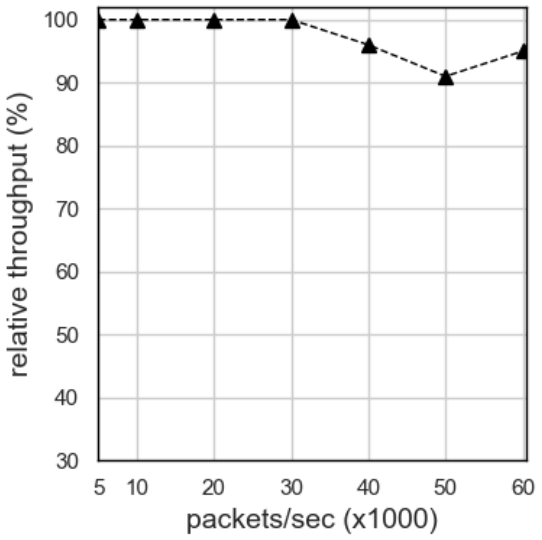


Figure 5.11: Throughput of inter-CPU allocation, relative to the respective intra-CPU allocation.

each splitting alternative).

Three cores. In this scenario, we split the same VNF-chain among three CPU cores. The splitting alternatives under consideration are as follows: (i) $C_1C_2-M_1-M_2$, (ii) $C_1-C_2M_1-M_2$, and (iii) $C_1 - C_2 - M_1M_2$. Fig. 5.10 illustrates a significant performance degradation when entire cores are not dedicated to CPU-intensive VNFs. In more detail, it can be observed that $C_1C_2 - M_1 - M_2$ yields the worst performance compared to the other two alternatives. As already explained, this performance drop stems from the similar resource consumption patterns of C_1 and C_2, along with the fact that they are stressed by the traffic injected in this experiment. This implies that a resource scheduler should place them in separate cores in order to alleviate resource contention. Even combining workloads with different resource profiles (*i.e.,* $C_1 - C_2M_1 - M_2$) does not lead to performance gains (compared to the most efficient splitting alternative), due to context switching between C_2 and M_1.

Finally, we assess the performance of $C_1 - C_2 - M_1M_2$ when splitting the chain among two different CPUs. In particular, we assign C_2 to a core belonging to a different CPU than the rest of VNFs, enforcing two inter-CPU transfers. According to Fig. 5.11, a perceptible performance drop occurs at rates beyond 30 Kbps. Nevertheless, this performance degradation remains

below 10%.

This experimentation leads to certain key observations. First, bundling heterogeneous (resource-wise) workloads increases performance, regardless of the inter-core transfers. This uncovers a potential trade-off between the consolidation of adjacent VNFs and the efficiency in CPU core utilization. Second, we empirically corroborate the arguably good practice to allocate separate cores to stressed VNFs, especially when these are subject to similar resource consumption patterns. Last, in our experimental setup, when the servers do not execute other workloads besides the VNF chains under test, chain splitting among CPUs leads to a perceptible decrease in throughput. Nevertheless, we expect a more severe impact of such CPU allocations at the presence of additional load on the server.

5.6 Related Work

In this section, we provide an overview of related work on (i) VNF performance evaluation and (ii) VNF CPU allocation.

VNF Performance Evaluation. In [71], the authors address the issue of contention for shared resources on multicore systems, specifically focusing on NUMA (non-uniform memory access) systems with multiple memory controllers. They propose a contention-aware scheduling algorithm tailored for NUMA systems, which outperforms previous NUMA-unaware algorithms and the default Linux scheduler. Dobrescu *et al.* [72] investigate the impact of resource contention on VNF performance. In particular, they analyze several factors that lead to resource contention, uncovering that access to LLC is one of the most critical contention aspects.

Towards more predictable VNF performance, the authors propose a packet processing system, which predicts performance drops in flow processing by taking into account the number of references to LLC that are associated with competing flows. Gaining insights and directions from [72], authors in [73] also analyze the impact of resource contention regarding VNF performance. In particular, the authors argue that LLC contention among VNFs is a critical performance bottleneck. They manage to overcome this issue by employing Intel's Cache Allo-

cation Technology, which enables the partitioning of the LLC, ultimately making it feasible to dedicate isolated LLC segments to individual cores. Eventually, they reach the conclusion that LLC isolation, along with a careful sizing of I/O, can result in a high degree of performance isolation between VNFs that are executed in the same CPU.

Last, in [74] authors present an analysis of the benefits associated with the utilization of SR-IOV (Single Root I/O Virtualization) enabled devices in combination with DPDK (Data Plane Development Kit) for the purpose of creating efficient and high-performance deployments of Virtual Network Functions (VNFs). The research findings indicate that the combination of SR-IOV and DPDK yields notable enhancements in packet throughput performance, as compared to use the native Linux kernel network stack for packet processing. This conclusion is drawn based on performance assessments conducted on different versions of VNFs using LibPCAP, SR-IOV, and DPDK. For their experiments, authors use a DPI VNF.

VNF CPU Allocation. In their study, Wang et al. [75] introduced a locality-first-mapping approach that involves consolidating an entire SFC within a single node to mitigate cross-node cost. However, the potential effects of intranode contention were not taken into account in their analysis. The study conducted by Hu et al. [76] examined the impact of core placement on the performance of pipelined software components, using Network Function Virtualization (NFV) as a case study. Nevertheless, their proposed method relies on dynamic scheduling, a factor that may prove unworkable in real-world NFV systems, as previously mentioned.

Octans [77] aims at maximizing the packet processing performance on many-core systems. More relevant to our work is the placement module of *Octans*, which strives to efficiently allocate cores to VNFs. The authors model the problem as a non-linear integer program. However, to alleviate the high solver runtime typically introduced by such methods, they devise a heuristic that is incorporated into the respective module. Authors in [78] experiment with a NFV framework that tackles the optimization of intra-server VNF-chain placements. Specifically, they subdivide VNFs into μVNFs (which can be seen as Click elements, as explained in Chapter 2), and they focus on optimally assigning those into cores. The problem is framed under a game-theoretic approach, and solved by utilizing neural combinatorial optimization and reinforcement learning

principles.

Beyond existing studies, our work positions heterogeneous workload mapping in the spotlight of the intra-server VNF placement problem which, to the best of our knowledge, has not been examined so far.

5.7 Conclusions

In this chapter, we set forth the imperative for precise and granular resource allocation within commodity servers to accommodate the execution of processing workloads characterized by potentially diverse demands in terms of CPU and memory. Consequently, our primary focus was directed towards the last-level resource allocation problem, specifically the allocation of CPU cores within a server. Through meticulous investigation, we explored various alternatives for VNF (Virtual Network Function) chain splitting and their direct implications on packet forwarding performance.

To begin with, we started by quantifying the impact of inter-core packet transfers, shedding light on the intricate dynamics involved in distributing network traffic across CPU cores within a server. By comprehensively examining the efficiency of CPU core utilization under both homogeneous and heterogeneous workloads, we sought to understand how different resource profiles affect the overall performance of the system.

Furthermore, our study revealed a crucial trade-off that arises when combining a mix of workloads and aiming for VNF consolidation. On one hand, consolidating multiple VNFs onto a single server enhances resource utilization efficiency by maximizing the utilization of CPU cores. On the other hand, this consolidation introduces challenges in terms of maintaining performance levels and ensuring optimal packet forwarding performance.

Through our findings, we equip resource schedulers with invaluable insights that enable them to make informed decisions when addressing the intra-server VNF placement problem. By understanding the impact of inter-core packet transfers, comprehending the implications of

homogeneous and heterogeneous workloads on CPU core utilization, and recognizing the trade-off between VNF consolidation and resource utilization efficiency, resource schedulers can design efficient and optimized resource allocation strategies.

In summary, our research serves to emphasize the criticality of fine-grained resource allocation within commodity servers for processing workloads with diverse demands. By focusing on the last-level resource allocation problem and thoroughly examining VNF chain splitting alternatives, we unravel the intricacies of packet forwarding performance and CPU core utilization. These insights provide a foundation for resource schedulers to make well-informed decisions, ultimately facilitating efficient intra-server VNF placement and optimizing resource utilization within NFV infrastructures.

Chapter 6

Time-Sensitive Networking for the Network Edge

6.1 Motivation

Over the last years, we have seen the emergence of (hyper-)distributed applications from different business models (Fig. 6.1), which tend to be provisioned over resources that span IoT devices, IoT gateways, and compute nodes from edge computing infrastructures. The deployment and orchestration of such services benefit from types of IoT middleware, known as Virtual Objects (VOs). However, in order to reap the benefits of VOs, low-latency communication is required between the IoT devices and their associated VOs.

In this respect, Deterministic Networking (DetNet) constitutes a promising solution for low-latency communication. DetNet has evolved as a crucial need for a wide range of application domains, such as industrial networks, automotive communications, mobile wireless networks, and service provider networks [79, 80, 81, 82, 83, 84, 85]. DetNet, in particular, aims at sustaining bounded latency and jitter, low packet loss, and high reliability for a subset of flows with stringent latency and/or throughput requirements. As such, the data of delay-sensitive flows can be delivered within certain deadlines, which becomes critical in *e.g.,* converged Internet-of-Things (IoT) and cloud infrastructures, where IoT devices need to establish low-latency

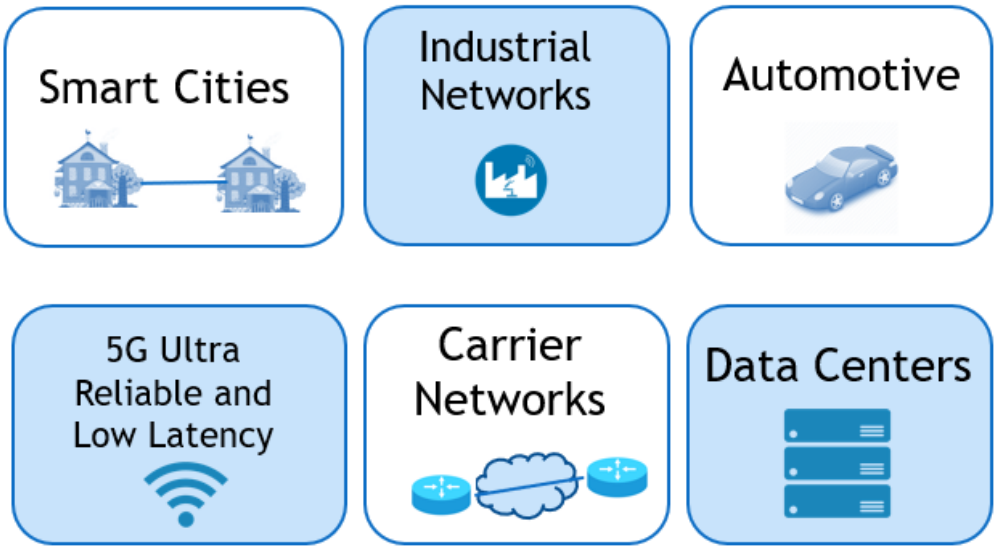


Figure 6.1: Different business models

communication either with application components or their digital twins.

In this respect, DetNet has attracted significant attention and is evolving rapidly under the auspices of various working groups (*e.g.,* IETF DETNET and IEEE 802.1 Time-Sensitive Networking – TSN) [84]. The former encompasses a set of standards and mechanisms for deterministic QoS, such as explicit routes, end-to-end synchronization, packet replication and elimination. On the other hand, IEEE 802.1 TSN has introduced a wide range of techniques for time synchronization (802.1 AS [33]), traffic scheduling (802.1 Qbv [31]), frame preemption (802.1 Qbu [32]), asynchronous traffic shaping (802.1 Qcr), and stream reservation (802.1 Qcc [86]), among others.

Although DetNet/TSN are evolving as an indispensable ingredient of network infrastructures for the needs of various application domains, in this chapter, we particularly focus on emerging IoT-cloud environments, which are characterized by the convergence of IoT and edge/cloud computing technologies. Such environments tend to utilize IoT platforms in the form of software stacks to augment the interaction between IoT devices and application components, often deployed on cloud infrastructures. This notion of IoT middleware can be incarnated through the so-called Virtual Objects (VOs) that essentially comprise virtual counterparts of either a single or a cluster of IoT devices [87].

A VO can execute generic or device-specific functions for data processing and fusion, alleviating

the computational burden of IoT devices and also reducing their energy consumption, which is of crucial importance for battery-operated IoT devices. Furthermore, VOs can facilitate the representation and management of (clusters of) IoT devices through unified abstractions, whereas, at the same time, they can pave the way for tackling various convergence and interoperability aspects.

Such VOs will be instantiated in edge cloud facilities, preferably in proximity to the IoT devices that they are associated with. For instance, IoT devices can be connected to an IoT gateway in order to establish connectivity with the network infrastructure, and, more specifically, with a proximate edge cloud that hosts the VOs in the form of containers. Nevertheless, an optimized VO placement cannot guarantee on its own certain bounds on latency and jitter for the communication between any IoT device and its respective VO. We stress on the need for IoT/VO low-latency communication, since any latency inflation (*i.e.,* due to interference with other traffic) can introduce implications (*e.g.,* high response time) on applications that interact with IoT devices through VOs or can hinder the synchronization in data acquisition processes from multiple devices.

In this context, there exists a significant requirement for deterministic communication between Internet of Things (IoT) devices and virtual organizations (VOs) in order to fulfill the key performance indicators (KPIs) of applications. These applications are deployed across resources that encompass IoT devices, IoT gateways, and compute nodes located at edge cloud facilities. To this end, we utilize Time-Sensitive Networking (TSN), and more specifically, the Time-Aware Shaper (TAS) mechanism based on IEEE 802.1 Qbv [9]. This approach can ensure bounded latency and jitter in the communication between IoT devices and their associated VOs.

A particular challenge entailed by TSN is the complex calculation of TSN schedules, which, within the framework of TAS, are equivalent to the configurations of Gate Control Lists (GCLs). The primary function of a TSN scheduling engine is to determine the optimal number of queues needed and also compute the transmission intervals for every traffic class across all TSN bridges along the path connecting each IoT-VO pair. Another notable challenge related to the TSN

control plane functionality is the management of the complexity and scalability of network configurations. As TSN networks expand in scale and intricacy, effectively configuring and overseeing the control plane becomes progressively more difficult. This raises the need for agility in terms of TSN bridge configuration in order to accommodate fluctuations in traffic patterns.

6.2 Contributions

The contribution of this research study is centered around two categories: (i) the design and implementation of a prototype TSN platform and (ii) the formulation of a TAS scheduling model:

1. **TSN Platform Architecture.** We present the architecture of a TSN platform, which stands as a key contribution to this chapter and integrates the following functionalities across its data plane and control plane:

    - *A TAPRIO-enabled Switch Datapath* that allows for the configuration of traffic flows to specific queues on egress ports. These queues are associated with a configurable Gate Control List (GCL).
    - *Centralized Network Controller (CNC)*, which automates the TSN schedule configuration and enhances the scalability and management efficiency. We utilize NETCONF and a TSN-YANG model for the binding between CNC and TAPRIO, enabling the installation of GCL configurations from the control plane into TSN-switches.

2. **TAS Scheduling Model**. The second major contribution in this chapter is the development of a TAS scheduling model, the cornerstone of the CNC modules. The CNC module integrates TAS scheduling that utilizes the principles of constraint programming. This methodology facilitates the establishment and improvement of intricate scheduling strategies, while simultaneously accommodating a wide array of limits and requirements.

By employing Constraint Programming as the foundational basis for our scheduling algorithm, we are capable of dynamically allocating resources and time slots, while also giving precedence to certain traffic flows. This strategy not only facilitates the efficient exploitation of the network resource but also ensures strict compliance with QoS regulations.

In addition, we analyze in detail the effects of different TSN schedules on critical network attributes, such as latency and jitter. This encompasses a comprehensive examination of both static TAPRIO schedules and the dynamic schedules computed by the proposed TAS scheduling model. We also quantify the TAPRIO setup overhead, with specific emphasis on the population of TSN scheduling configurations from CNC to TAPRIO, using NETCONF. This is achieved by adapting TAPRIO into Mininet, building upon our prior research efforts [79].

6.3 TSN Platform

In this section, we present our TSN platform design and implementation. The platform is depicted in Figure. 6.2 and couples a TAPRIO-based TSN datapath with a TSN controller (*i.e.,* CNC), empowering experimentation with a fully-fledged TSN bridge that encompasses both data and control plane elements (Fig. 6.2). In the following, we discuss in detail the functionality of the TSN platform.

TSN Data Plane. We hereby describe the TSN data plane functionality, activated on network devices (*e.g.,* IoT Gateways) that reside between IoT devices and applications. For TSN data plane activation, we utilize TAPRIO - a powerful queuing discipline available in the Linux kernel's traffic control (tc) tool. TAPRIO plays a crucial role in simulating the behavior of IEEE 802.1Qbv, which is a standard for enhancing time-aware scheduling in Ethernet networks. By integrating TAPRIO, we allow the configuration of a series of gate states, each one responsible for enabling outgoing traffic for specific subsets of traffic classes based on the concept of time slices. To ensure proper packet classification into the appropriate traffic class, TAPRIO utilizes

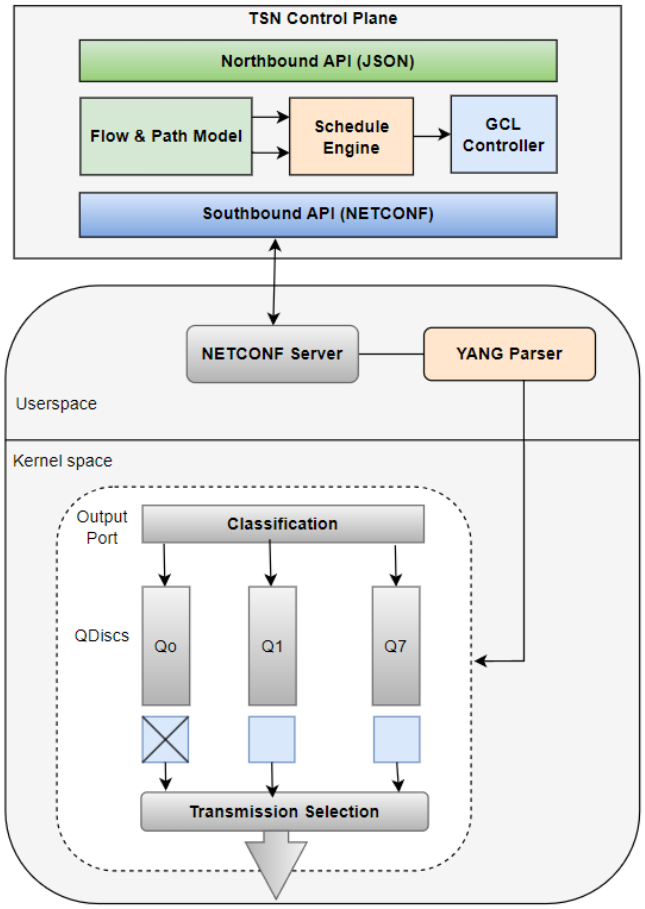


Figure 6.2: TSN platform overview.

the priority field of the socket buffer (skb) employed by the network stack of the Linux Kernel. This enables TAPRIO to effectively assign time-sensitive flows to their respective priority queues. In our implementation, we map traffic classes to queues by modifying the DSCP (Differentiated Services Code Point) field of the packet header. As such, we prioritize traffic based on specific service requirements and deliver the desired quality of service (QoS) to different types of data streams. To achieve the modification of the skb priority field before packets are directed to the queuing discipline, we employ the use of iptables, a versatile packet filter tool operating at the IP layer. By incorporating the relevant classifier rules into iptables, we effectively manipulate the skb priority field with precision. As such, we establish the appropriate priority for the skb (socket buffer) as packets traverse the network. Through this comprehensive setup, we effectively integrate TAPRIO into the data plane of our IoT Gateway, enabling the timely delivery of data between IoT devices and applications.

More precisely, the workflow for packet handling and classification by TAPRIO is the following:

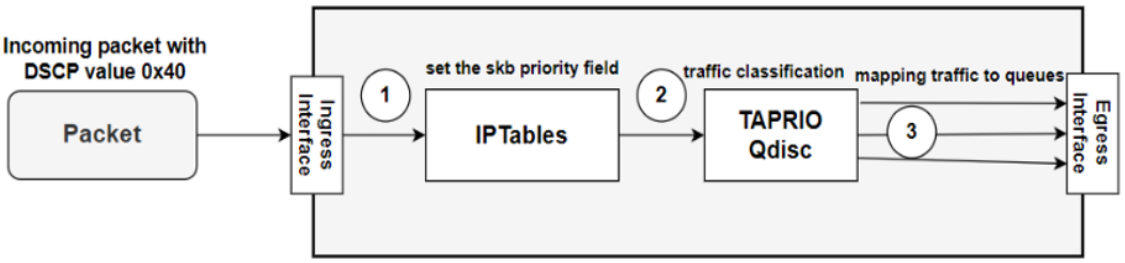


Figure 6.3: Packet handling workflow in TAPRIO.

(i) the incoming packet, which is tagged with a DSCP value of 0x40 indicating high priority, arrives at the ingress interface; (ii) the first step of classification involves using IPTables to set the skb priority field to 0x40; (iii) subsequently, the TAPRIO queuing discipline assigns the incoming packet to high-priority queues. The aforementioned steps are depicted in Fig. 6.3.

TSN Control Plane. As far as the TSN Control plane is concerned, we utilize a prototype implementation of a Central Network Controller (CNC) that consists of three internal modules: (i) Flow & Path Model, (ii) Schedule Engine, and (iii) GCL Controller.

- **Flow & Path Model.** Path and Flow Model module has two main functionalities. The first one is the categorization of incoming flows into distinct traffic classes, such as high-priority and best-effort. This is achieved based on some predefined rules that can match applications' network requirements (*e.g.,* latency less than 1 ms) to traffic classes and determine whether the request should be categorized as critical or non-critical. The second functionality of this module is the path configuration and by path, we define the fixed network between the IoT Gateway and the applications. This path will be used as an input to the Schedule Engine module.

- **Schedule Engine.** The Schedule Engine module is implemented as an internal component and is responsible for implementing a scheduling model to determine a scheduling pattern for the incoming flows in order to satisfy their latency requirements. In this implementation, the scheduling model is based on constrained programming, since the latter offers versatility in problem modeling, and efficient heuristic search in combination with powerful constraint propagation techniques. It should be mentioned that the TSN

scheduling problem is classified as an NP-complete problem and various strategies have been proposed to address this problem in the literature, including Satisfiability Modulo Theories (SMT) [83, 88], Constraint Programming [89], Heuristics [90, 91], and Genetic Algorithms [92]. The implementation of our proposed TAS scheduling mechanism is presented later, in the Section 6.4.

- **GCL Controller.** The GCL Controller module receives the output of the Schedule Engine as its input and is responsible for configuring time intervals on the Gate Control List, and determining the duration over which each queue is open for transmission. The GCL Controller utilizes YANG [36] in order to send the GCL configuration to the IoT Gateway.

Furthermore, the TSN Control Plane incorporates two application programming interfaces (APIs). The proposed system includes a Northbound API, implemented using a well-defined JSON schema, which is capable of processing requests related to application configuration and requirements. Additionally, a technology-specific Southbound API, utilizing NETCONF [93], is responsible for transmitting the GCL configuration to the IoT Gateway through the use of Remote Procedure Calls (RPC).

6.3.1 Interactions between TSN Control and Data Plane

In the TSN architectural framework illustrated in Figure. 6.2, a YANG Parser, deployed at the userspace of the TSN bridge, parses the YANG-TSN model to a set of actions that can be applied directly to the queuing disc layer of the Linux kernel (Fig. 6.2). The CNC establishes communication through the NETCONF [93] plugin by utilizing the YANG data model. The NETCONF plugin functions as a management client and establishes communication with the NETCONF server that is operational on each TSN bridge, such as an IoT Gateway. Following the completion of their computational process, the TSN schedules are transmitted to the IoT gateway.

```
module: tsn-taprio
  +--rw tsn-taprio-structure
     +--rw interface* [dev]
        +--rw dev                string
        +--rw parent?            string
        +--rw handle?            uint32
        +--rw num-tc?            uint8
        +--rw map?               string
        +--rw queues
        |  +--rw queue* [id]
        |     +--rw id       uint32
        |     +--rw elem?    string
        +--rw base-time?         uint64
        +--rw clockid?           string
        +--rw sched-entries
           +--rw sched-entry* [id]
              +--rw id            uint32
              +--rw command?      string
              +--rw gatemask?     string
              +--rw interval?     uint64
  rpcs:
    +---x taprio-set
       +---w input
       |  +---w command?   string
       +--ro output
          +--ro result?   boolean
```

Figure 6.4: Example of TSN-TAPRIO module.

An illustrative example of this model appears in Fig. 6.4. In particular, this figure illustrates the *tsn-taprio* module for transmitting the TAPRIO configuration to the relevant network interface via a RPC. This module allows for the definition of the interface through which the TAPRIO configuration will be activated, as well as the specification of the number of traffic classes (referred to as *num-tc*) and the number of hardware queues to which the traffic classes will be mapped. Furthermore, the Gate Control List schedule can be defined using the *sched-entries* parameter, which allows for the configuration of time intervals. These intervals determine the length for which each scheduled entry will be active before transitioning to the next entry.

6.4 TSN Scheduler

In this section, we elaborate on the functionality of the *Scheduling Engine Module* of our TSN platform, which is responsible for computing feasible TSN schedules. In this respect, we delve into the TSN scheduling problem, which is considered as NP-complete problem. Our main

objective is to determine a feasible scheduling pattern for incoming flows with respect to their specified requirements (*e.g.,* latency, jitter), based on a defined model and constraints. The scheduling model in this work is built on the principles of constraint programming, due to its ability to provide flexibility in terms of problem modeling and effective heuristic search combined with robust constraint propagation techniques.

We follow a similar approach to modeling the TSN scheduling problem in constraint programming, as those in [94], [83], [95]. Thus, we assume the usual graph representation found in [94], to represent the network, *i.e.,* we consider a graph $G = (V, E)$, where vertices (nodes n_i) are either switches or end-points, whereas edges are links ($l_{ij} \in E$) connecting the former, *i.e.,* link l_{ij} connects nodes i and j respectively. Each link l_{ij} is characterized by its propagation delay $d_{l_{ij}}$, its speed sp_{ij}, and a number of queues Q_{ij} that are at most 8 according to IEEE 802.1 Qbv [31]. Switches also have a processing (or fabric) delay sd_i which is constant.

In the current setting, we consider a set of periodic flows F of single packets (frames), *i.e.,* the payload of each flow can fit into a single Ethernet frame, where each flow $f_k \in F$, has a deadline df_k, a packet size p_k, a period T_k that determines the frequency of the transmission, and is associated with a valid path P^k, that consists of an ordered list of links $[l_{ti}, l_{ij}, \dots l_{nl}]$ from the transmitting end-point n_t, *i.e.,* the *talker*, to the receiving end-point n_l, (*i.e.,* the *listener* of the flow.

The problem can be considered as a classic job-shop scheduling problem, where each flow transmission is considered as a *task*, consisting of as many transmission *operations* as the links it traverses along the path to the listener, with a number of additional constraints that will be discussed below. Thus, a flow f_k on a path P_k of length N can be modeled as a set of operations $1..N$, each having a start time (offset) relative to the start of the schedule (time point 0), *i.e.,* the set $\{S^k_{1,i,j}, S^k_{2,j,k}, ...S^k_{N,k,l}\}$ where each variable $S^k_{x,i,j} | x \in 1..N$ represents the start time of the transmission of the packet of flow k on the link l_{ij}. We define this set as the *primary flow*, for reasons explained later in this section. Additionally, for each flow f_k we define a queue Q_k, which is considered to be the same for all links of the flow.

Scheduling of flows that have different periods occurs in a time domain defined by the *hyperperiod*[83],

where the later is defined as least common multiple of all flow periods, *i.e.,* $HP = lcm(T_k | f_k \in F)$, thus flows may occur multiple times in a hyperperiod. In the later case, each flow f_k that occurs $M = HP/T_i$ times in the hyperperiod, consists of $N * M$ start times, as indicated in Eq. (6.1):

$$\begin{aligned} f_k = \{&S^k_{1,i,j}, \dots S^k_{N,k,l}, \\ &S^k_{N+1,i,j}, \dots S^k_{2*N,k,l}, \\ &\dots \\ &S^k_{N*(M-1)+1,i,j}, \dots S^k_{M*N,k,l}\} \end{aligned} \tag{6.1}$$

We define subflows $S^k_{x,i,j} | x > N$ as *secondary flows* (mentioned as flow occurrences in [94]); we make this distinction since primary flow start time variables and, those of the secondary flow, participate in different constraint in the model. In the CP model, the decision variables are the start times in both primary and secondary flows and the flow's queue, *i.e.,* $< \{S^k_{o,i,j} \in [0..HP] | o \in 1..N * M\}, Q_k \in 1..7 >$. The domain of Q_k is restricted to the range [1..7], since queue 0 is reserved for unscheduled best-effort traffic.

Decision variables in the primary and secondary flows are linked by the requirement for zero jitter, which dictates that the time difference in corresponding variables in each secondary subflow should obey the constraint in Eq. (6.2):

$$\begin{aligned} &\forall n \in 1..N, m \in 2..M, \forall l_{ij} \in P_k \\ &S^k_{n,i,j} + (m-1) * T_k = S^k_{N*(m-1)+n,i,j} \end{aligned} \tag{6.2}$$

The first scheduling constraint imposes an *ordering* among the start times of each operation in a task, ensuring that a packet on link l_{ij} has arrived on node n_j, before being transmitted over the link l_{jl}. Enforcement of the constraint occurs only on decision variables of the primary flow and is depicted in Eq. (6.3). There is no need to define the constraint for secondary flows, due to the equality constraints of Eq. (6.2), which ensure that the adequate time distance is maintained among operations of each subflow.

$$\begin{aligned} &\forall f_k \in F, \forall o \in 1..(N-1), \\ &S^k_{o,i,j} + sd_j + d_{l_{ij}} + \lceil \frac{p_k}{sp_{ij}} \rceil =< S^k_{o+1,j,l} \end{aligned} \tag{6.3}$$

Eq. (6.3) factors in the propagation delay $d_{l_{ij}}$ of the link, the transmission delay computed as the ceiling of the packet size over the speed of the link $\lceil \frac{p_k}{sp_{ij}} \rceil$, and the switch delay sd_j of the receiving node.

Given that flows have a deadline, the flow's packet must arrive to the listener node n_l over a link l_{ml} before the deadline df_k:

$$\forall f_k \in F, S^k_{N,m,l} + d_{l_{ml}} + \lceil \frac{p_k}{sp_{ml}} \rceil <= df_k \tag{6.4}$$

Once more, the deadline constraint is imposed only on the primary flow variables, and equality constraints of Eq. (6.2) ensure that is enforced for all flows in the hyperperiod.

Each egress link transmits a single packet at each time point, thus, no two transmissions on the same link may overlap, yielding the constraint in Eq. (6.5):

$$\begin{aligned} &\forall l_{ij} \in E, f_k, f_r \in F | k < r, \forall S^k_{x,i,j} \in f_k, \forall S^r_{y,i,j} \in f_r, \\ &S^k_{x,i,j} + d_{l_{ij}} + \lceil \frac{p_k}{sp_{ij}} \rceil <= S^r_{y,i,j} \\ &\vee \\ &S^r_{y,i,j} + d_{l_{ij}} + \lceil \frac{p_r}{sp_{ij}} \rceil <= S^k_{x,i,j} \end{aligned} \tag{6.5}$$

Eq. (6.5), commonly found in scheduling problems, is handled by the *global* constraint *disjunctive* [96], with a plethora of dedicated algorithms to efficiently tackle variable domain reductions. Note that Constraint (6.5) is enforced in all decision variables of both primary and secondary flows.

Finally, since a frame is fully received before being copied to its destination queue and frames are transmitted according to their reception order, it must be ensured that packets addressed to the same egress link, arrive in the correct order when placed in the same egress queue or

are forced to be placed in different queues. The latter is referred to as the *frame isolation* constraint, and is modelled by the disjunction depicted in Eq. (6.6):

$$\begin{aligned}
&\forall l_{ij} \in E, f_k, f_r \in F | k < r, \forall S^k_{x,i,j} \in f_k, \forall S^r_{y,i,j} \in f_r, \\
&\quad \Big((Q_k = Q_r) \Rightarrow \\
&\qquad (S^k_{x,i,j} < S^r_{y,i,j} \Leftrightarrow Arr^k_{x,i} < Arr^r_{y,i}) \wedge \\
&\qquad (S^k_{x,i,j} > S^r_{y,i,j} \Leftrightarrow Arr^k_{x,i} > Arr^r_{y,i}) \Big) \\
&\quad \vee \\
&\quad Q_k \neq Q_r
\end{aligned} \tag{6.6}$$

where $Arr^k_{x,i}$ is the arrival time of the packet of flow f_k at node i, and is given by Eq. (6.7):

$$Arr^k_{x,i} = S^k_{x-1,a,i} + d_{l_{ai}} + \lceil \frac{p_k}{sp_{ai}} \rceil \tag{6.7}$$

The definition of $Arr^r_{y,i}$ is similar. If the frames of flows f_k and f_r arrive on the same ingress link, then the link constraints of Eq. (6.5) ensure that are transmitted in the correct order by imposing a much more restrictive constraint. If frames arrive on different ingress links, then we ensure that the reception of one of the frames is completed before the other, and thus, maintaining the order of placing the frames in the queue. This constraint can be easily implemented using *reified constraints*, offered in most CP solvers.

6.5 Evaluation Results

In this section, we discuss all the evaluation results in relation to the efficiency of the proposed TSN scheduler and the performance of the TSN platform overall.

6.5.1 TSN Scheduler Evaluation

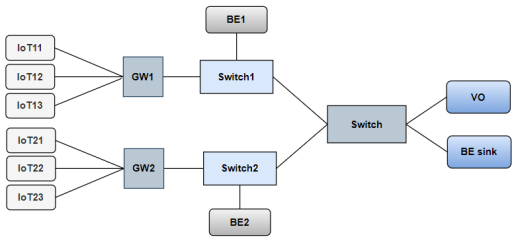
The validation of our proposed TAS scheduler is conducted on the OMNeT++ v6.0.1[1] simulation platform, using the INET 4.5.0[2] framework. All tests are executed on a workstation with 3.2 GHz CPU and 16 GB RAM.

In order to assess the proposed TSN scheduling model, we utilize three different topologies, as depicted in Fig. 6.5. All topologies include paths between IoT nodes to VOs, as well as additional communicating nodes responsible for cross-traffic. The number of IoT nodes varies between 6 and 12, depending on the topology. In each topology, traffic is forwarded via TSN switches that utilize GCLs at their egress ports in order to handle traffic prioritization. Link capacities are set to 100 Mbps, whereas an additional propagation delay ($d_{l_{ij}}$) of $1\mu s$ is introduced as a simulation parameter. Although OMNeT++ does not inherently introduce processing delays on switches, our tests have uncovered an observed processing delay (sd_i) of approximately $1\mu s$. Consequently, we account for this delay in our simulations.

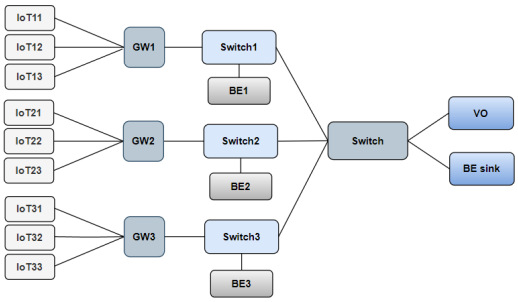
Across all three simulated topologies, the IoT nodes are responsible for generating time-sensitive traffic, which is treated as high-priority (also termed as *scheduled* traffic). The period of all these flows is set to $800\mu s$, which is also considered as the deadline for each flow, thus, ensuring that all packets are delivered to the VO within a cycle. The length of each packet in the scheduled traffic is set to *400 bytes*. Scheduled traffic is tagged with Priority Code Point (PCP) value 4 to augment TSN switches with its handling (*i.e.,* as high-priority). Furthermore, nodes tagged as BE (Fig. 6.5) serve the purpose of traffic interference, by injecting Best-Effort (BE) traffic towards sink nodes. Each packet of BE traffic has a size of 800 bytes. BE nodes generate traffic at intervals of 10 μs, tagged with PCP value of 0.

We utilize our TSN scheduling model to derive the GCL schedules and conduct a set of simulations in order to assess the efficiency of the computed schedules for high-priority traffic. For each simulation topology, the search is based on a value ordering heuristic, selecting the minimum value for the start time from each domain, thus, reporting a valid solution, which is

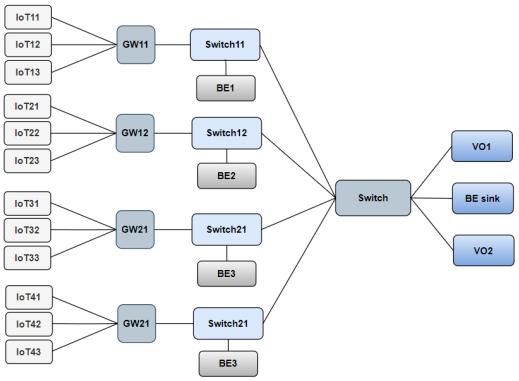
[1]https://omnetpp.org/
[2]https://inet.omnetpp.org/



(a) Topology 1.



(b) Topology 2.



(c) Topology 3.

Figure 6.5: Simulation topologies.

not optimized according to some metric. This helps maintain a negligible runtime for schedule computation, *i.e.,* approximately 4 ms with the largest simulated topology.

The main aim of our study is two-fold: (i) to evaluate the levels of jitter and latency on a TSN topology, and (ii) to showcase the practicality of our model's scheduling capabilities. As a baseline for evaluations, we rely on static schedules which have been employed by various studies for traffic prioritization (*e.g.,* [85, 79]). In such a static scheduling, the high-priority queue is associated with an $800\mu s$ scheduling interval, whereas the best-effort queue is scheduled at $200\mu s$. In the following, we present evaluation results[3] of the proposed TSN scheduling mechanism for

[3]Results with best-effort traffic are omitted due to space limitations.

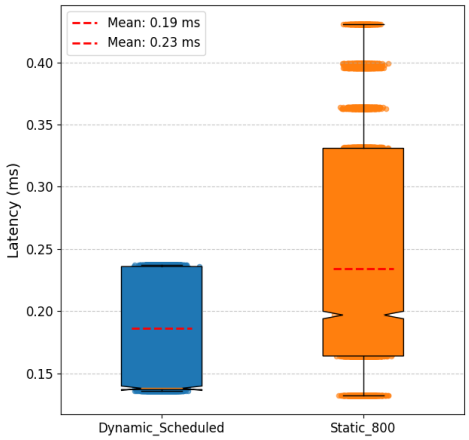


Figure 6.6: Latency of scheduled traffic on topology 1.

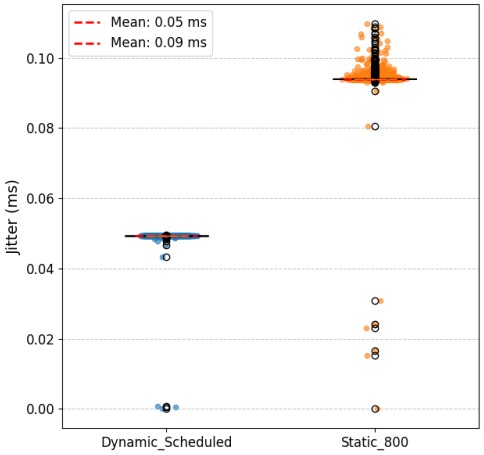


Figure 6.7: Jitter of scheduled traffic on topology 1.

scheduled from simulations conducted on all three topologies, as shown in Fig. 6.5.

Figs. 6.6 and 6.7 illustrate the measured latency and jitter from the simulations on topology 1. The proposed scheduler (indicated as *Dynamic_Scheduled* in the following plots) outperforms the static scheduler both in terms of latency and jitter. More precisely, our schedule model yields a median delay of 0.19 ms. The small size of the interquartile range indicates that a significant number of latency values are tightly positioned around the median, which indicates a much lower degree of deviation from the median. This combined with the lack of outliers corroborates the superiority of the proposed scheduler, especially in terms of latency bounds which are more significant (than mean values) in the context of TSN. For instance, the latency with the proposed scheduler remains bounded below 0.25 ms (Fig. 6.6), as opposed to the static schedules that may lead to delays in excess of 0.4 ms.

Similar observations can be drawn with respect to the jitter that the scheduled traffic experiences (Fig. 6.7). Scheduling traffic with static intervals yields an increase by 80% in the median value of jitter (compared to the proposed scheduler); however, the margin between the two scheduling techniques is even larger in terms of jitter bounds. While the deviation around the mean is barely perceptible for the proposed scheduler, jitter can exceed 0.1 ms with the static schedules.

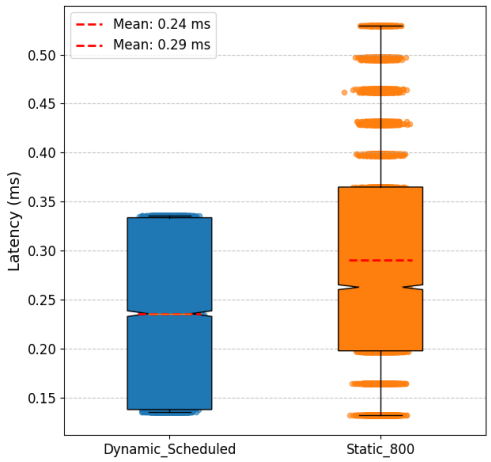


Figure 6.8: Latency of scheduled traffic on topology 2.

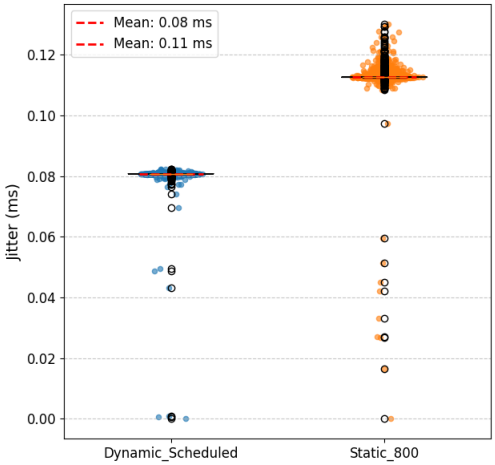


Figure 6.9: Jitter of scheduled traffic on topology 2.

We now proceed with the discussion of latency (Fig. 6.8) and jitter (Fig. 6.9) measurements with the topology 2, which includes additional talkers that generate larger volumes of traffic (both in terms of IoT-VO and BE). The two scheduling methods under test exhibit similar margins in their performance, compared to the tests with topology 1. More specifically, the proposed TSN scheduler achieves a notable reduction in the mean values of latency and jitter, compared to the static counterpart. In addition, the deviation around the mean is relatively confined for our scheduler, which can effectively provide guarantees both in terms of latency and jitter (*e.g.,* we observe a latency bound at approximately 0.35 ms).

Finally, we conduct an additional round of simulations over a more complex topology (*i.e.,* topology 3 in Fig. 6.5c), which is characterized by a higher interference among the two classes of traffic (*i.e.,* high-priority and BE). Figs. 6.10 and 6.11 illustrate the corresponding results for the latency and jitter experienced by the scheduled traffic with both TSN schedule variants (*i.e.,* dynamic and static). The results indicate that the proposed scheduler is less susceptible to interference among the two traffic classes, by sustaining lower latency and jitter both in terms of mean value and upper bound. Instead, the static scheduler exhibits a substantial deviation in the values of latency and jitter, where, for instance, the latency can inflate to more than 0.4 ms on a topology with minimal propagation delay. The scheduled traffic suffers even more in terms of jitter (Fig. 6.11), when the static scheduler is in use. It can be observed that even the

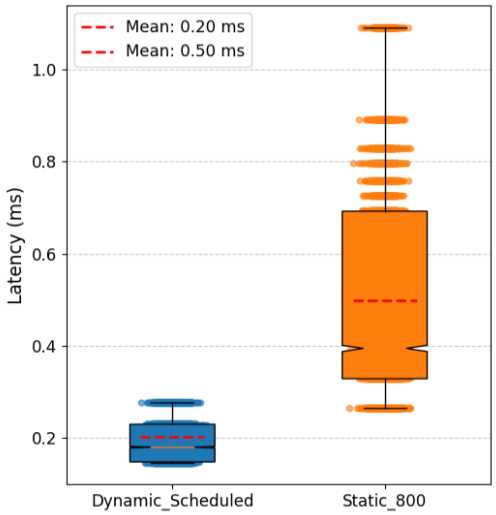


Figure 6.10: Latency of scheduled traffic on topology 3.

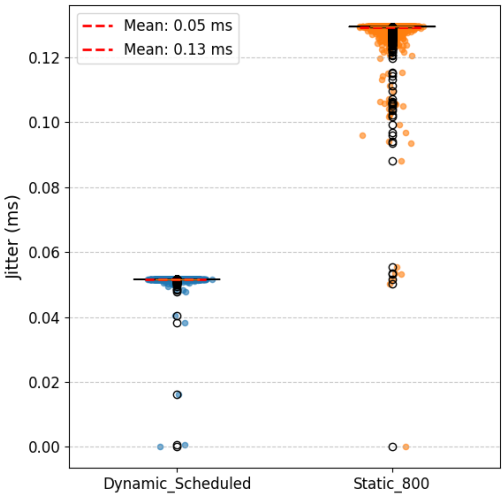


Figure 6.11: Jitter of scheduled traffic on topology 3.

mean jitter value is substantially higher compared to our scheduler. Apparently, conditions of high interference require dynamically computed schedules in order to alleviate the implications on scheduled traffic.

6.5.2 TSN Platform Evaluation

In this section, we utilize our TSN platform and conduct a set of experiments to assess various aspects of TSN. More specifically, we assess the impact of TAPRIO scheduling on high-priority and best-effort traffic, and we also quantify the control communication overhead in relation to the CNC. In order to assess the benefits of TSN, we activate TAPRIO on the egress port of an IoT Gateway using the topology depicted in Fig. 6.12, which is created through a modified version of Mininet 2.3.1 [97]. In order to use TAPRIO in this topology, based on [79] we modify Mininet in order to support multi-queued NIC interfaces, since by default Mininet only supports single-queue interfaces. More specifically, the modified Mininet version can support up to 8 TX/RX queues. We also use IPMininet [98] in order to support IPV6 addressing. All experiments are carried out on an Ubuntu 20.04.1 LTS Virtual Machine with 8 virtual CPUs and 8 GB of RAM, running kernel version 5.4.0-139.

The goal of this experiment is to demonstrate the impact of TAPRIO scheduling on an IoT

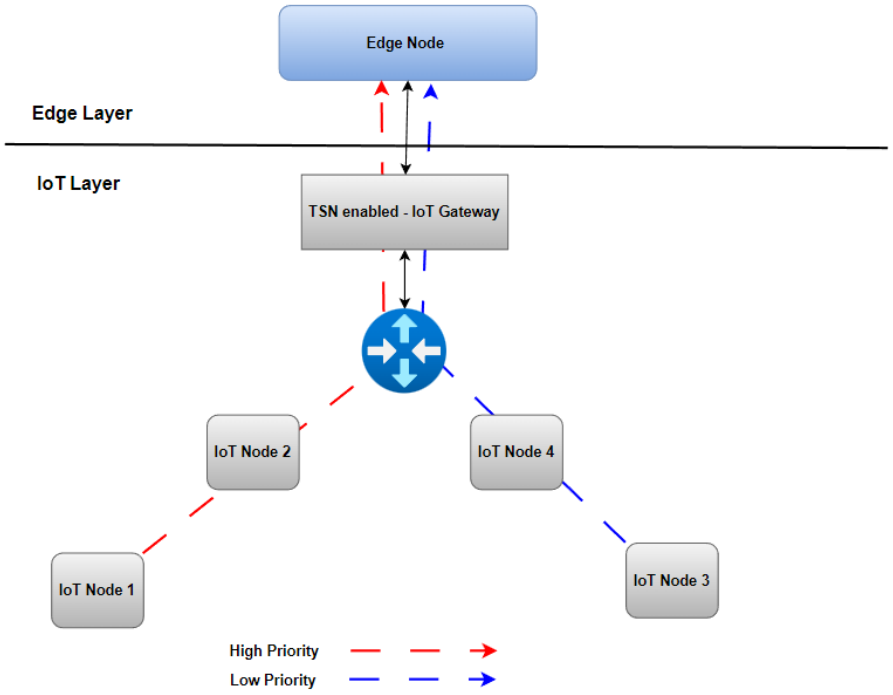


Figure 6.12: Experimental topology in Mininet.

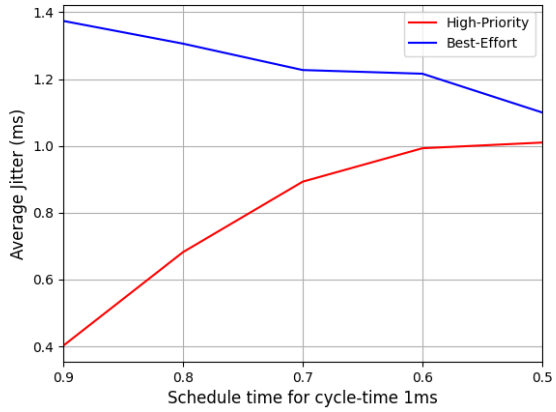


Figure 6.13: Impact of 802.1Qbv on jitter for High-Priority and Best-Effort traffic.

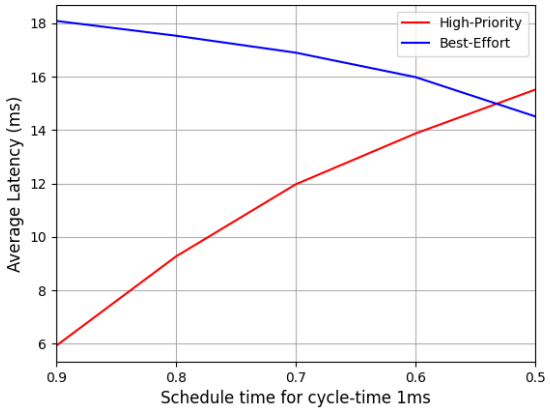


Figure 6.14: Impact of 802.1Qbv on latency for High-Priority and Best-Effort traffic.

Gateway. Based on the Mininet topology depicted in Fig. 6.12, we set up the TAPRIO Qdisc with two traffic classes: (i) High Priority and (ii) Best Effort, where the former is configured to match traffic with DSCP field value of 0x40, wheras the latter matches best-effort traffic with DSCP field value of 0x00. To generate traffic, we rely on Iperf [99]. Specifically, we inject packets of size 1,440 bytes at a rate of 2,000 packets/s for High-Priority traffic and CBR traffic of 1,440 bytes packet size for Best-Effort traffic.

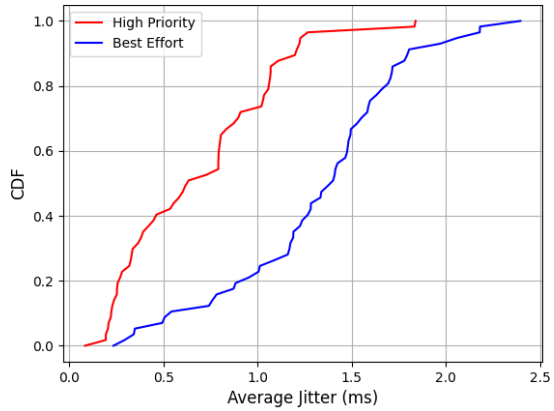


Figure 6.15: CDF of jitter with TAPRIO 800:200 (80% of the time allocated to High-Priority traffic and 20% to Best-Effort traffic on a cycle time of 1 ms).

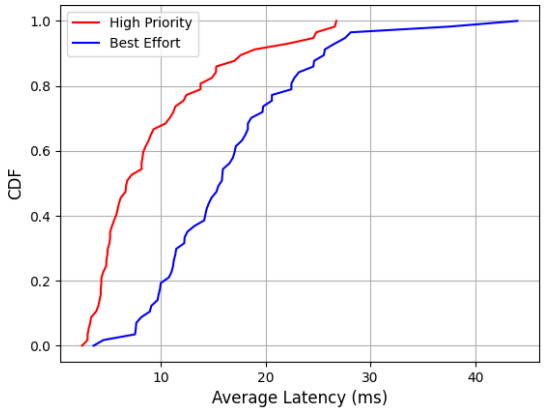


Figure 6.16: CDF of latency with TAPRIO 800:200 (80% of the time allocated to High-Priority traffic and 20% to Best-Effort traffic on a cycle time of 1 ms).

The impact of 802.1Qbv scheduling on latency and jitter for a 1 ms cycle time is shown in Figs. 6.13 and 6.14, by varying the allocated time ratio between High-Priority and Best-Effort traffic. For the sake of simplicity, only average values are presented as the network cycle is not synchronized with the application cycle (such synchronization is extremely difficult to attain within Mininet).

The average latency and jitter for High-Priority traffic yield a linear increase as the percentage of allocated cycle time decreases. Conversely, the average latency and jitter for Best-Effort traffic decreases as the percentage of allocated time for this traffic class increases. It is important to note that these results are consistent with the expected behavior of TAPRIO's scheduling mechanism, where High-Priority traffic is prioritized over Best-Effort traffic. In addition, we observe that the maximum latency and jitter are highly dependent on the current policy in effect, which is determined by the allocated time ratio between the two traffic classes. Overall, these results highlight the effectiveness of TAPRIO in managing traffic and prioritizing delay-sensitive traffic in a network environment.

We conduct another experiment, at which we utilize again the Mininet topology of the previous experiment and send 1440-byte packets at a rate of 2000 packets per second both for High-Priority and Best-Effort traffic. The traffic is injected using Iperf, and the primary goal of this

experiment is to measure the effect of different allocation times in the GCL. More specifically, we allocate 80% of the time to high-priority traffic and 20% to low-priority traffic on a cycle time of 1 ms. We measure the impact of this allocation on jitter and latency, by plotting the CDF for each metric (Figs. 6.15 and 6.16, respectively).

Fig. 6.15 illustrates the distribution of jitter for High-Priority and Best-Effort traffic. This plot confirms that High-Priority is associated with lower jitter compared to Best-Effort traffic. Specifically, the measurements of High-Priority traffic indicate a jitter of less than 1 ms for 80 percent of the observations, whereas Best-Effort traffic experiences jitter of less than 1.6 ms for the same percentage of observations. These results imply that allocating a higher percentage of the GCL time to High-Priority traffic can lead to reduced (and potentially bounded) jitter in the network, which is beneficial for real-time applications that require both low latency and jitter.

The plot in Fig. 6.16 depicts the distribution of latency for High-Priority and the Best-Effort traffic. We reach a similar observation, as High-Priority traffic exhibits lower latency. In particular, on the 80 percent of the observations, the latency for High-Priority traffic does not exceed 13 ms, whereas Best-Effort traffic can experience latency up to 23 ms. Furthermore, we observe a long tail in the latency distribution of Best-Effort traffic (as opposed to High-Priority). This implies that TAPRIO can practically lead to bounded latency for traffic scheduled with high priority.

It is important to note that the high latency measurements in our experiments are an artifact of Mininet. Achieving stable and accurate measurements in a software-based emulation environment is difficult, and as such, high-precision measurements mandate hardware solutions, such as programmable NICs or NetFPGAs, in conjunction with specialized capture cards. Also, recall the difficulty in enabling synchronization among the TSN switches and the talkers within an emulation environment. In the future, we will seek to gain access to such specialized equipment and utilize our platform for measurements over TSN with higher precision.

In addition to the previous experiments on the effect of TAPRIO on scheduled traffic, we extend our experimentation to the interaction between the CNC and the TSN datapath. In particular,

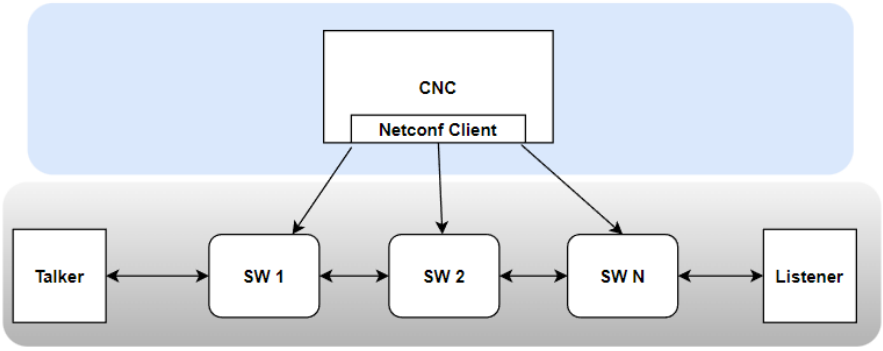


Figure 6.17: Experimental setup for the measurement of control communication overhead (N = 1...10).

our aim is to measure and understand the communication overhead of the CNC when TSN schedules are populated into the switch. To this end, we perform tests on a topology consisting of a talker-listener pair, 1–10 switches, and the CNC (Fig. 6.17).

Initially, we measure the control communication overhead when inserting the TSN schedule configuration into only one switch. Subsequently, we increase the number of switches and measure the delay, as we populate configurations into switches ranging from two to ten. To better reason about the time spent for TAPRIO configuration, we measure (i) the time required to generate the TAPRIO configuration at the CNC, (ii) the communication delay between the NETCONF server and the NETCONF client, and (iii) the delay incurred by the server to parse the configuration from the NETCONF client and convert it to the appropriate format. These essentially comprise three subsequent steps for the configuration of TAPRIO by the CNC.

Table 6.1: Control communication overhead

Switches	**TAPRIO Configuration Generation (sec)**	**NETCONF Communication (sec)**	**YANG Parsing (sec)**
1	0,128	0,026	0,019
2	0,257	0,024	0,019
3	0,482	0,025	0,020
4	0,625	0,025	0,020
5	0,803	0,022	0,021
6	1,017	0,024	0,021
7	1,198	0,025	0,020
8	1,379	0,024	0,020
9	1,599	0,025	0,022
10	1,781	0,027	0,021

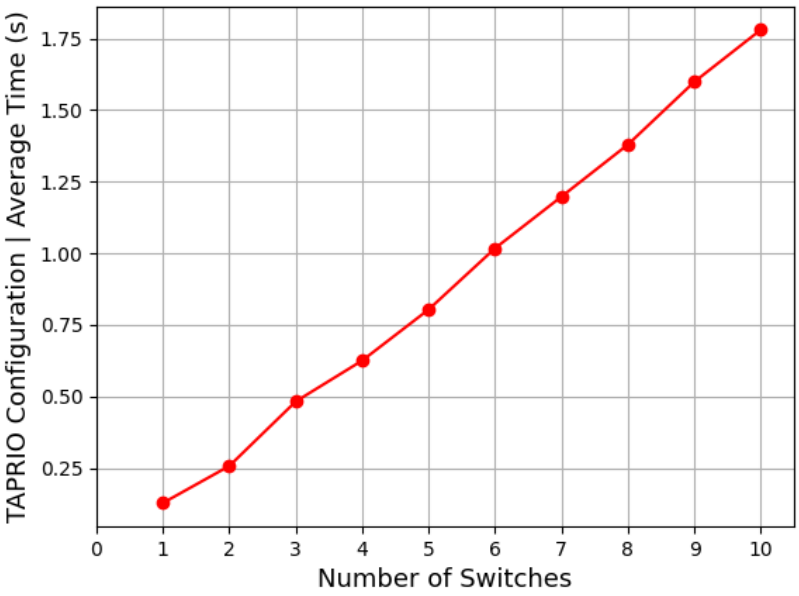


Figure 6.18: Control communication overhead vs. number of switches.

The corresponding results are shown in Table 6.1. The time required by the CNC to generate the TAPRIO configuration represents a significant portion of the control communication overhead, in comparison to the communication delay and the XML parsing overhead. More specifically, we observe that the time required to complete the first step (*i.e.,* TAPRIO schedule generation at the CNC) increases linearly with the number of switches, as illustrated more clearly in Fig 6.18. The delays incurred for the subsequent steps (2 and 3) are minimal and mostly unaffected by the increase in the number switches. This stems from the parallelization of each process across the switches. Overall, the low control communication overhead measured in our experiments corroborates the efficacy of our TSN platform, especially in terms of CNC and TAPRIO interaction.

6.6 Related Work

In this section, we provide an overview of related work on (i) TSN Scheduling with respect to 802.1Qbv (ii) TSN data plane TSN data plane and (iii) TSN control plane respect to IEEE 802.1Qbv.

TAS Scheduling. Authors in [100, 83] tackle the TSN scheduling problem by utilizing Satisfiability Modulo Theory (SMT) and Optimization Modulo Theory (OMT) techniques. The study in [101] investigates the impact of routing time-triggered flows on their capacity to adhere to a schedule. In addition, an Integer Linear Programming (ILP) formulation is proposed for the purpose of constructing predictable routes. Authors in [102] also resort to ILP for addressing the combined routing and scheduling time-triggered traffic problem within the context of Software-Defined Networking (SDN). Vlk, *et al.* [94] address the problem of joint routing and scheduling in a CP setting, proposing two CP models and evaluating mainly the schedulability (*i.e.,* percentage of successful computation of a TSN schedule).

The study in [103] stresses on the need for fast scheduling algorithms in the TSN domain. More specifically, the authors introduce a heuristic scheduler, namely HERMES, designed for time-triggered traffic in TSNs. Authors in [104] introduce a schedulability analysis for the Best-Effort (BE) traffic class in the context of TSNs. The emphasis of this study lies in on the BE traffic class, which is commonly used for traffic that does not exhibit stringent delay requirements. Zhao in [105] raises the need for worst-case end-to-end latency analysis in distributed safety-critical applications across various domains, including industrial automation, aerospace, and automotive industries. A delay analysis for AVB traffic is also presented in [106].

A different approach to 802.1 Qbv scheduling is presented in [107]. Specifically, the authors introduce a heuristic scheduler designed for TSNs. This scheduler utilizes a window-based approach, eliminating the need for isolating open gate windows. Machine Learning (ML) based approaches for schedulability analysis on TSNs are introduced in [108, 109]. In particular, the authors employ supervised learning in order to verify the TSN configurations. Finally, the study in [110] addresses the requirement for real-time communication in the context of Industry 4.0 and intelligent manufacturing facilities. The objective is to establish systems that facilitate real-time communication within networked industrial environments.

Data Plane. Authors in [84] provide an overview of various existing strategies, including TSN, in order to achieve ultra-low latency. Furthermore, [111] performs a comparison between IEEE 802.1Qbv and 802.1Qbu based on simulations, highlighting the drawbacks of TAS in terms of

configuration complexity and guard bands inefficiencies. The TSN 802.1Qbv scheduling problem has been addressed using various techniques. For example, the problem has been tackled using Satisfiability Modulo Theory (SMT) and Optimization Modulo Theory (OMT) [100, 83]. Authors in [112] implement a simulation TSN environment by leveraging on OMNet++ and examine the impact of TSN scheduling on different classes of traffic. Finally, the work in [113] describes the development of a TSN system that is reliable and scalable. The system has successfully implemented the IEEE 802.1ASrev standard for clock synchronization and the IEEE 802.1Qbv standard for ordered packet sending on Linux end-devices.

Control Plane. As far as the control plane is concerned, authors in [80] present an approach for ultra-fast recovery of TSN using Software-Defined Networking (SDN). More specifically, they employ Source Routing for a nearly-stateless data plane in order to achieve fast failure recovery. Another approach introduces a mechanism for managing reliability in SDN-controlled TSN networks, by utilizing NETCONF and Finite State Machines (FSMs) [114]. By proactively instructing nodes to perform specific actions in response to certain events such as performance degradation or failures, the proposed approach reduces recovery time and can effectively bypass the SDN controller. Alvarez et al. [115] have implemented a dockerized CNC, which supports various TSN implementations. Authors in [116] argue that P4 is not sufficiently abstract to program a TSN-enabled switch on its own, but can be used instead in combination with other tools or platforms (*e.g.,* tc, DPDK, OpenvSwitch) in order to achieve the necessary modularity.

In relation to the previous TSN scheduling techniques, we employ constraint programming to devise a TSN schedule computation engine tailored to the needs of low-latency communication between IoT devices and VOs in converged IoT-cloud environments. Our proposed mechanism is compliant with TAS and addresses the dynamicity of (edge) cloud environments through the ability of rapid TSN schedule computation.

6.7 Conclusions

In this chapter, we presented a TAS-compliant scheduler specifically designed to meet the stringent requirements of IoT applications. Our evaluations corroborate the efficacy of TSN schedules in reducing latency and jitter for scheduled traffic. The TAS model, in particular, has demonstrated its superiority in ensuring bounded latency and minimal schedule computation times. The constraint programming model significantly outperforms static scheduling methods, especially in maintaining tight control over latency and jitter deviations. This is critical for scheduled traffic where consistency is as crucial as performance.

Moreover, we presented a TSN platform that enables experimentation with TSN mechanisms in both testbed and network emulation environments. Our TSN platform encompasses (i) a TAPRIO-enabled switch datapath that can be configured to handle traffic under pre-defined TSN schedules and (ii) a prototype implementation of a CNC that leverages on the the proposed TSN scheduling mechanism in order to generate GCL schedules for critical traffic.

Chapter 7

Conclusion

7.1 Summary of Thesis Achievements

In the dynamic landscape of modern networking, the emergence of the IoT and the advent of 5G networks have ignited a surge of research activities in the realms of edge computing, network slicing, TSN, and IoT convergence. This thesis represents a comprehensive journey through these evolving domains, where the orchestration of complex services, the optimization of resource allocation, the facilitation of low-latency communication, and the imperative need for cross-service interaction at the edge of the network take center stage. Resource-constrained environments, such as edge clouds, pose unique challenges where computational capabilities, storage, and bandwidth are limited. To address these challenges and harness the full potential of edge resources, innovative solutions have been proposed and explored. In the following, we summarize the distinct achievements and contributions of this thesis across these research areas.

1. **Resource Orchestration at the Edge and Computation Offloading.** Edge cloud environments offer a promising prospect for hosting applications and enhancing end-user experience. Nevertheless, it is worth noting that edge nodes, such as servers, often have restricted resources. This necessitates the careful allocation of resources, particularly when

we have to deal with strict KPIs. Our primary achievement in this area is the design, implementation, and evaluation of an orchestration framework that enables the computation offloading from IoT and mobile devices to the edge cloud servers. By incorporating control-plane components for workload prediction, load balancing, and admission control, the proposed framework can efficiently orchestrate the underlying resources of servers that reside at the edge of the network.

2. **Network Slicing and Optimized Cross-Slice Communications.** Recognizing the critical role of network slicing in the 5G landscape, we have contributed to the domain of management and orchestration (MANO) frameworks. Our research introduces the concept of CSC, which empowers B2B interactions, cross-service communication, as well as service composition in the domain of edge computing. Delving deeper into the importance of CSC for enhanced service interactions, we have implemented an orchestration framework that enables the establishment of cross-slice communication in a secure and controlled manner. Our experimental results corroborate the minimal performance overhead and latency inflation incurred by the CSC orchestration framework, highlighting its potential for secure and resource-efficient cross-slice communication.

3. **Fine-Grained Resource allocation.** Given the importance of optimized resource allocation in NFV and edge infrastructures, we have investigated the challenge of intra-server resource allocation, which entails the assignment of (virtualized) network functions onto CPU cores. Our experimental study of various VNF chain allocation strategies, under different utilization levels and processing workloads fills a crucial gap in the existing literature. Our results lay the ground for fine-grained resource allocation strategies within multi-core servers, complementing the field of traditional VNF placement where a server is deemed as the minimum resource allocation unit.

4. **Low-Latency Communication in Converged IoT/Cloud Infrastructures.** Emerging hyper-distributed applications tend to be provisioned over resources that span from IoT devices to edge computing infrastructures. These services benefit from types of IoT middleware known as Virtual Objects (VOs). To reap the benefits of VOs, low-latency

communication is required between the devices and their associated VOs. To this end, we have designed a TSN switching architecture that couples a TAS-based TSN bridge with centralized TSN control, aligned with the notion of CNC. A salient feature of the TSN control plane is a TAS scheduler that efficiently computes schedules for traffic classes with different latency/jitter requirements, to protect high-priority traffic from cross-traffic interference. The TAS scheduler is tailored to converged IoT/edge cloud environments, which exhibit high dynamicity due to the short life-time of services and the potential re-associattion of IoT nodes with VOs (*e.g.,* due to mobility). Our experimentation validates the efficacy of the proposed TAS scheduler, which can sustain bounded latency and jitter across of a range of experimental environments.

7.2 Applications

The innovations and research contributions outlined in this thesis demonstrate considerable potential for practical applications in diverse fields. These accomplishments provide flexible solutions that can be adapted to tackle practical issues in the subsequent domains:

- **Edge Computing and IoT Deployment.** The prevalence of IoT necessitates the implementation of effective edge computing solutions. The orchestration framework, which has been developed in the context of this thesis, has a particular emphasis on resource allocation and computational offloading. This framework has direct applicability in the context of IoT deployments. The efficiency and responsiveness of IoT applications can be improved by effectively transferring computational duties from IoT and mobile devices to edge cloud servers. This technology has the potential to provide advantages to various industries, encompassing smart cities, industrial automation, and environmental monitoring.

- **Synergies among services.** With the advent of 5G networks, the demand for network slicing and synergies between services hosted in the same location is increasing. The thesis

presents significant contributions in the areas of network slicing and cross-slice communication orchestration, which hold the promise of transforming the telecommunications sector. These advances can be utilized by service providers to enhance the efficiency of network resources, improve quality of service, and accommodate a wide range of usage scenarios. The enhanced network performance and service-tailored offerings, facilitated by CSC, can be advantageous to industrial sectors, such as video streaming, augmented reality, and self-driving vehicles.

- **Edge cloud resource management.** Beyond edge computing and 5G networks, the fine-grained resource allocation strategies examined herein have broad applications in edge datacenters and cloud environments. As data centers continue to expand and virtualization technologies evolve, the need for efficient resource allocation becomes increasingly critical. The research outcomes in this domain offer solutions to optimize the allocation of resources within multi-core servers, contributing to improved scalability, resource utilization, and cost efficiency in cloud computing infrastructures.

- **TSN for IoT Deployments.** The design of a holistic TSN architectural framework featuring a Traffic-Aware Scheduler (TAS) effectively caters for the requirement for low-latency communication in converged IoT/cloud environments. The utilization of this technology is applicable to situations where the processing and control of data in real-time are of utmost importance, such as in the domains of industrial automation, healthcare, and autonomous systems. Our TSN platform can provide better support for emerging applications that depend on real-time and accurate data flow between devices and their associated middleware, by sustaining bounded latency and jitter.

7.3 Future Work

Although this thesis has made notable progress in tackling various challenges entailed by edge computing and IoT, there exists space for further investigation and advancements.

A potential direction for future investigation involves the continued development and enhance-

ment of the orchestration framework presented in this study, specifically focusing on resource allocation at the edge. As the field of edge computing progresses and incorporates various applications, there is potential to improve the framework in order to accommodate changing workloads and evolving KPIs, hence, optimizing resource management.

Furthermore, the use of cross-slice communication in this research presents significant opportunities for future investigation. Future study can explore the advancement of comprehensive service chaining mechanisms, with a specific focus on optimizing their performance in large-scale networks. Additionally, there is a need to design enhanced security protocols, which are tailored to the security requirements stemming from cross-slice communication in edge environments, which are becoming progressively intricate.

In addition, the domain of fine-grained resource allocation in edge cloud nodes poses continuous barriers and prospects. Investigating innovative approaches and algorithms to effectively manage resource constraints and enhance resource efficiency presents a promising direction for research. We can also explore the integration of machine learning and artificial intelligence methods for the purpose of developing adaptive resource allocation systems that exhibit better response to evolving workload patterns.

Lastly, in the endeavor to sustain low latency at the network edge, future research can extend the TSN architecture that was proposed in this thesis. Additional experimentation and optimization of the TAS scheduler could yield valuable insights regarding its performance across diverse network conditions and traffic patterns. Another challenging aspect that is expected to spur significant research interest is the translation of high-level network requirements or intents into low-level TSN schedule configurations that can be readily populated into TSN bridges. Such a system could bridge the gap between the abstract view of a service provider and the detailed view of a network operator that is entitled with low-level configurations in a network infrastructure.

Bibliography

[1] S. Bhattacharjee, R. Schmidt, K. Katsalis, C.-Y. Chang, T. Bauschert, and N. Nikaein, "Time-sensitive networking for 5g fronthaul networks," in *ICC 2020-2020 IEEE International Conference on Communications (ICC)*. IEEE, 2020, pp. 1–7.

[2] J. Ordonez-Lucena, P. Ameigeiras, D. Lopez, J. J. Ramos-Munoz, J. Lorca, and J. Folgueira, "Network slicing for 5g with sdn/nfv: Concepts, architectures, and challenges," *IEEE Communications Magazine*, vol. 55, no. 5, pp. 80–87, 2017.

[3] A. Pentelas and P. Papadimitriou, "Network service embedding for cross-service communication," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2021, pp. 424–430.

[4] Ericsson. (2023) Ericsson mobility report - june 2023. Accessed on November 28, 2023. [Online]. Available: https://www.ericsson.com/49dd9d/assets/local/reports-papers/mobility-report/documents/2023/ericsson-mobility-report-june-2023.pdf

[5] C. Bravo and H. Bäckström, "Edge computing and deployment strategies for communication service providers," *White Paper, Ericsson*, 2020.

[6] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed, "Edge computing: A survey," *Future Generation Computer Systems*, vol. 97, pp. 219–235, 2019.

[7] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1657–1681, 2017.

[8] M. Chen and Y. Hao, "Task offloading for mobile edge computing in software defined ultra-dense network," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 587–597, 2018.

[9] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019.

[10] D. Spatharakis, I. Dimolitsas, D. Dechouniotis, G. Papathanail, I. Fotoglou, P. Papadimitriou, and S. Papavassiliou, "A scalable edge computing architecture enabling smart offloading for location based services," *Pervasive and Mobile Computing*, vol. 67, p. 101217, 2020.

[11] G. Papathanail, I. Fotoglou, C. Demertzis, A. Pentelas, K. Sgouromitis, P. Papadimitriou, D. Spatharakis, I. Dimolitsas, D. Dechouniotis, and S. Papavassiliou, "Cosmos: An orchestration framework for smart computation offloading in edge clouds," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–6.

[12] M. Aleksandrova, "The impact of edge computing on iot: The main benefits and real-life use cases," *Eastern Peak*, 2019.

[13] NFV White Paper, "Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action. Issue 1," Oct. 2012.

[14] M. Kourtis *et al.*, "T-nova: An open-source mano stack for nfv infrastructures," *IEEE Transactions on Network and Service Management*, vol. 14, no. 3, pp. 586–602, Sept 2017.

[15] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications surveys & tutorials*, vol. 18, no. 1, pp. 236–262, 2015.

[16] M. Villamizar *et al.*, "Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures," in *CCGrid*, 2016.

[17] "Open Source MANO," https://osm.etsi.org/, accessed: 2023-01-30.

[18] "Open Baton," https://openbaton.github.io/, accessed: 2023-01-30.

[19] "OpenMano," https://github.com/nfvlabs/openmano, accessed: 2023-01-30.

[20] "Openstack," http://mininet.org/, accessed: 2023-02-28.

[21] "openvim," https://github.com/nfvlabs/openvim, accessed: 2023-01-30.

[22] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2014.

[23] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: An intellectual history of programmable networks," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, Apr. 2014.

[24] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[25] N. Alliance, "5g white paper," *Next generation mobile networks, white paper*, vol. 1, no. 2015, 2015.

[26] 3GPP, "Study on architecture for next generation system," 2016.

[27] I.-T. Y. 3011, "Framework of network virtualization for future networks," *Next Generation Networks-Future Networks*, 2012.

[28] I. Afolabi, T. Taleb, K. Samdanis, A. Ksentini, and H. Flinck, "Network slicing and softwarization: A survey on principles, enabling technologies, and solutions," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 3, pp. 2429–2453, 2018.

[29] N. Finn, P. Thubert, B. Varga, and J. Farkas, "Deterministic networking architecture," *RFC 8655*, 2019.

[30] P. Danielis, J. Skodzik, V. Altmann, E. B. Schweissguth, F. Golatowski, D. Timmermann, and J. Schacht, "Survey on real-time communication via ethernet in industrial automation environments," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*. IEEE, 2014, pp. 1–8.

[31] IEEE, *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks—Amendment 25: Enhancements for Scheduled Traffic*, ser. IEEE Std 802.1Qbv-2015. IEEE, Mar. 2016.

[32] ——, *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks—Amendment 26: Frame Preemption*, ser. IEEE Std 802.1Qbu-2016. IEEE, Aug. 2016.

[33] IEEE, "Standard for local and metropolitan area networks—timing and synchronization for time-sensitive applications in bridged local area networks, ieee standard 802.1as-2011, pp. 1–292, mar. 2011."

[34] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "Network configuration protocol (netconf), rfc 6241," *IETF, June*, 2011.

[35] A. Bierman, M. Bjorklund, and K. Watsen, "Rfc 8040: Restconf protocol," 2017.

[36] M. Bjorklund, "Yang-a data modeling language for the network configuration protocol (netconf)," Tech. Rep., 2010.

[37] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Journal of basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960.

[38] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*. Springer, 2014, pp. 740–755.

[39] M. Avgeris, D. Dechouniotis, N. Athanasopoulos, and S. Papavassiliou, "Adaptive resource allocation for computation offloading: A control-theoretic approach," *ACM Transactions on Internet Technology (TOIT)*, vol. 19, no. 2, p. 23, 2019.

[40] M. G. R. Alam, M. M. Hassan, M. Z. Uddin, A. Almogren, and G. Fortino, "Autonomic computation offloading in mobile edge for iot applications," *Future Generation Computer Systems*, vol. 90, pp. 149–157, 2019.

[41] M. Avgeris, D. Spatharakis, D. Dechouniotis, N. Kalatzis, I. Roussaki, and S. Papavassiliou, "Where there is fire there is smoke: a scalable edge computing framework for early fire detection," *Sensors*, vol. 19, no. 3, p. 639, 2019.

[42] A. Leivadeas, G. Kesidis, M. Ibnkahla, and I. Lambadaris, "Vnf placement optimization at the edge and cloud," *Future Internet*, vol. 11, no. 3, p. 69, 2019.

[43] Y. Shi, S. Chen, and X. Xu, "Maga: A mobility-aware computation offloading decision for distributed mobile cloud computing," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 164–174, 2017.

[44] Z. Cao and P. Papadimitriou, "Collaborative content caching in wireless edge with SDN," in *1st ACM CONEXT Workshop on Content Caching and Delivery in Wireless Networks, Heidelberg, Germany, December 1-4, 2015*, pp. 6:1–6:7.

[45] D. Dietrich, A. Abujoda, A. Rizk, and P. Papadimitriou, "Multi-provider service chain embedding with nestor," *IEEE Transactions on Network and Service Management*, vol. 14, no. 1, pp. 91–105, 2017.

[46] Z. Cao, A. Abujoda, and P. Papadimitriou, "Distributed data deluge (d3): Efficient state management for virtualized network functions," in *2016 IEEE Conference on Computer Communications Workshops*, April 2016, pp. 782–787.

[47] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: a framework for nfv applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 121–136.

[48] M. Mechtri, C. Ghribi, O. Soualah, and D. Zeghlache, "Nfv orchestration framework addressing sfc challenges," *IEEE Communications Magazine*, vol. 55, no. 6, pp. 16–23, 2017.

[49] "ETSI Network Function Virtualization," http://www.etsi.org/technologies-clusters/technologies/nfv, accessed: 20-04-2023.

[50] D. Dechouniotis, I. Dimolitsas, K. Papadakis-Vlachopapadopoulos, and S. Papavassiliou, "Fuzzy multi-criteria based trust management in heterogeneous federated future internet testbeds," *Future Internet*, vol. 10, no. 7, p. 58, 2018.

[51] "ETSI Multi-access edge computing," http://www.etsi.org/technologies/multi-access-edge-computing.

[52] E. Kohler *et al.*, "The click modular router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, 2000.

[53] "Vuurmuur," https://www.vuurmuur.org/trac/.

[54] "Suricata," https://suricata-ids.org/.

[55] "Monitorix," https://www.monitorix.org/.

[56] I. Dimolitsas, M. Avgeris, D. Spatharakis, D. Dechouniotis, and S. Papavassiliou, "Enabling industrial network slicing orchestration: A collaborative edge robotics use case," in *2021 IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*. IEEE, 2021, pp. 215–220.

[57] F. S. D. Silva, M. O. Lemos, A. Medeiros, A. V. Neto, R. Pasquini, D. Moura, C. Rothenberg, L. Mamatas, S. L. Correa, K. V. Cardoso *et al.*, "Necos project: Towards lightweight slicing of cloud federated infrastructures," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE, 2018, pp. 406–414.

[58] P. D. Maciel, F. L. Verdi, P. Valsamas, I. Sakellariou, L. Mamatas, S. Petridou, P. Papadimitriou, D. Moura, A. I. Swapna, B. Pinheiro *et al.*, "A marketplace-based approach to cloud network slice composition across multiple domains," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 480–488.

[59] J. Sherry *et al.*, "Making middleboxes someone else's problem: Network processing as a cloud service," in *ACM SIGCOMM*, 2012, p. 13–24.

[60] A. Abujoda and P. Papadimitriou, "Midas: Middlebox discovery and selection for on-path flow processing," in *IEEE COMSNETS 2015*, 2015.

[61] J. G. Herrera and J. F. Botero, "Resource allocation in nfv: A comprehensive survey," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 518–532, 2016.

[62] H. Moens and F. De Turck, "Vnf-p: A model for efficient placement of virtualized network functions," in *IFIP/IEEE CNSM*, 2014.

[63] R. Cohen *et al.*, "Near optimal placement of virtual network functions," in *IEEE INFOCOM*, 2015.

[64] M. C. Luizelli *et al.*, "A fix-and-optimize approach for efficient and large scale virtual network function placement and chaining," *Computer Communications*, vol. 102, pp. 67–77, 2017.

[65] A. Abujoda and P. Papadimitriou, "Distnse: Distributed network service embedding across multiple providers," in *COMSNETS*, 2016.

[66] A. Pentelas *et al.*, "Network service embedding across multiple resource dimensions," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 209–223, 2021.

[67] B. White *et al.*, "An integrated experimental environment for distributed systems and networks," *ACM SIGOPS*, 2002.

[68] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[69] T. Barbette *et al.*, "Fast userspace packet processing," in *ACM/IEEE ANCS*, 2015.

[70] A. Abujoda and P. Papadimitriou, "Profiling packet processing workloads on commodity servers," in *IFIP WWIC*, 2013.

[71] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A case for numa-aware contention management on multicore systems," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 2010, pp. 557–558.

[72] M. Dobrescu *et al.*, "Toward predictable performance in software packet-processing platforms," in *USENIX (NSDI)*, 2012.

[73] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker, "{ResQ}: Enabling {SLOs} in network function virtualization," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 283–297.

[74] M.-A. Kourtis, G. Xilouris, V. Riccobene, M. J. McGrath, G. Petralia, H. Koumaras, G. Gardikis, and F. Liberal, "Enhancing vnf performance by exploiting sr-iov and dpdk packet processing acceleration," in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE, 2015, pp. 74–78.

[75] Y. Wang, "Numa-aware design and mapping for pipeline network functions," in *2017 4th International Conference on Systems and Informatics (ICSAI)*. IEEE, 2017, pp. 1049–1054.

[76] Y. Hu and T. Li, "Towards efficient server architecture for virtualized network function deployment: Implications and implementations," in *2016 49th annual IEEE/ACM international symposium on microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[77] Z. Zheng *et al.*, "Octans: Optimal placement of service function chains in many-core systems," in *IEEE INFOCOM*, 2019.

[78] P. Krämer *et al.*, "sfc2cpu: Operating a service function chain platform with neural combinatorial optimization," in *IM*, 2021.

[79] G. N. Kumar, K. Katsalis, and P. Papadimitriou, "Coupling source routing with time-sensitive networking," in *IFIP Networking Conference (Networking)*, 2020, pp. 797–802.

[80] G. N. Kumar, K. Katsalis, P. Papadimitriou, P. Pop, and G. Carle, "Failure handling for time-sensitive networks using sdn and source routing," in *7th IEEE International Conference on Network Softwarization (NetSoft)*, 2021, pp. 226–234.

[81] ——, "Srv6-based time-sensitive networks (tsn) with low overhead rerouting," *International Journal of Network Management, Wiley*, 2022.

[82] S. Fu, H. Zhang, and J. Chen, "Time sensitive networking technology overview and performance analysis," *ZTE Communications*, vol. 16, no. 4, pp. 57–64, 2018.

[83] S. S. Craciunas, R. S. Oliver, M. Chmelík, and W. Steiner, "Scheduling real-time communication in ieee 802.1 qbv time sensitive networks," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, 2016, pp. 183–192.

[84] 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*, vol. 21, no. 1, pp. 88–145, 2018.

[85] G. Papathanail, L. Mamatas, and P. Papadimitriou, "Towards the integration of taprio-based scheduling with centralized tsn control," in *2023 IFIP Networking Conference (IFIP Networking)*. IEEE, 2023, pp. 1–6.

[86] *IEEE, Std 802.1Qcc-2019: Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks–Amendment 9: Stream Reservation Protocol (SRP) Enhancements and Performance Improvements*. IEEE, 2019.

[87] "Nephele project," https://nephele-project.eu/.

[88] W. Steiner, "An evaluation of smt-based schedule synthesis for time-triggered multi-hop networks," in *2010 31st IEEE Real-Time Systems Symposium*. IEEE, 2010, pp. 375–384.

[89] J. Dai, Z. Wang, and L. Zhong, "Research on gating scheduling of time sensitive network based on constraint strategy," in *Journal of Physics: Conference Series*, vol. 1920, no. 1. IOP Publishing, 2021, p. 012089.

[90] M. Pahlevan, N. Tabassam, and R. Obermaisser, "Heuristic list scheduler for time triggered traffic in time sensitive networks," *ACM Sigbed Review*, vol. 16, no. 1, pp. 15–20, 2019.

[91] F. Ansah, M. A. Abid, and H. de Meer, "Schedulability analysis and gcl computation for time-sensitive networks," in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, vol. 1. IEEE, 2019, pp. 926–932.

[92] M. Pahlevan and R. Obermaisser, "Genetic algorithm for scheduling time-triggered traffic in time-sensitive networks," in *2018 IEEE 23rd international conference on emerging technologies and factory automation (ETFA)*, vol. 1. IEEE, 2018, pp. 337–344.

[93] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "Network Configuration Protocol (NETCONF)," in *RFC 6241*, 2011.

[94] M. Vlk, Z. Hanzálek, and S. Tang, "Constraint programming approaches to joint routing and scheduling in time-sensitive networks," *Computers & Industrial Engineering*, vol. 157, p. 107317, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360835221002217

[95] M. Vlk, Z. Hanzálek, K. Brejchová, S. Tang, S. Bhattacharjee, and S. Fu, "Enhancing schedulability and throughput of time-triggered traffic in ieee 802.1qbv time-sensitive networks," *IEEE Transactions on Communications*, vol. 68, no. 11, pp. 7023–7038, 2020.

[96] P. Baptiste, P. Laborie, C. L. Pape, and W. Nuijten, "Chapter 22 - constraint-based scheduling and planning," in *Handbook of Constraint Programming*, ser. Foundations of Artificial Intelligence, F. Rossi, P. van Beek, and T. Walsh, Eds. Elsevier, 2006, vol. 2, pp. 761–799. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S157465260680026X

[97] "Mininet," http://mininet.org/, accessed: 2023-02-28.

[98] "IPMininet," https://ipmininet.readthedocs.io/en/latest/, accessed: 2023-02-28.

[99] V. GUEANT, "iperf-iperf3 and iperf2 user documentation," *Iperf. fr. Np*, 2017.

[100] W. Steiner, S. S. Craciunas, and R. S. Oliver, "Traffic planning for time-sensitive communication," *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 42–47, 2018.

[101] N. G. Nayak, F. Dürr, and K. Rothermel, "Routing algorithms for ieee802. 1qbv networks," *ACM Sigbed Review*, vol. 15, no. 3, pp. 13–18, 2018.

[102] ——, "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.

[103] D. Bujosa, M. Ashjaei, A. V. Papadopoulos, T. Nolte, and J. Proenza, "Hermes: Heuristic multi-queue scheduler for tsn time-triggered traffic with zero reception jitter capabilities," in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, 2022, pp. 70–80.

[104] B. Houtan, M. Ashjaei, M. Daneshtalab, M. Sjödin, S. Afshar, and S. Mubeen, "Schedulability analysis of best-effort traffic in tsn networks," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2021, pp. 1–8.

[105] L. Zhao, P. Pop, and S. S. Craciunas, "Worst-case latency analysis for ieee 802.1 qbv time sensitive networks using network calculus," *Ieee Access*, vol. 6, pp. 41 803–41 815, 2018.

[106] D. Maxim and Y.-Q. Song, "Delay analysis of avb traffic in time-sensitive networks (tsn)," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, 2017, pp. 18–27.

[107] N. Reusch, L. Zhao, S. S. Craciunas, and P. Pop, "Window-based schedule synthesis for industrial ieee 802.1 qbv tsn networks," in *2020 16th IEEE International Conference on Factory Communication Systems (WFCS)*. IEEE, 2020, pp. 1–4.

[108] T. L. Mai, N. Navet, and J. Migge, "A hybrid machine learning and schedulability analysis method for the verification of tsn networks," in *2019 15th IEEE International Workshop on Factory Communication Systems (WFCS)*. IEEE, 2019, pp. 1–8.

[109] ——, "On the use of supervised machine learning for assessing schedulability: application to ethernet tsn," in *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, 2019, pp. 143–153.

[110] D. Hellmanns, A. Glavackij, J. Falk, R. Hummen, S. Kehrer, and F. Dürr, "Scaling tsn scheduling for factory automation networks," in *2020 16th IEEE International Conference on Factory Communication Systems (WFCS)*. IEEE, 2020, pp. 1–8.

[111] A. Arestova, K.-S. J. Hielscher, and R. German, "Simulative evaluation of the tsn mechanisms time-aware shaper and frame preemption and their suitability for industrial use cases," in *IFIP Networking Conference (IFIP Networking)*, 2021, pp. 1–6.

[112] C. Park, J. Lee, T. Tan, and S. Park, "Simulation of scheduled traffic for the ieee 802.1 time sensitive networking," in *Information Science and Applications (ICISA) 2016*. Springer, 2016, pp. 75–83.

[113] J. Lázaro, J. Cabrejas, A. Zuloaga, L. Muguira, and J. Jiménez, "Time sensitive networking protocol implementation for linux end equipment," *Technologies*, vol. 10, no. 3, p. 55, 2022.

[114] N. Sambo, S. Fichera, A. Sgambelluri, G. Fioccola, P. Castoldi, and K. Katsalis, "Enabling delegation of control plane functionalities for time sensitive networks," *IEEE Access*, vol. 9, pp. 136 151–136 163, 2021.

[115] I. Álvarez, A. Servera, J. Proenza, M. Ashjaei, and S. Mubeen, "Implementing a first cnc for scheduling and configuring tsn networks," in *27th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2022, pp. 1–4.

[116] S. T. Borda and J. Ermont, "An evaluation of software-based tsn traffic shapers using linux tc," in *2022 IEEE 18th International Conference on Factory Communication Systems (WFCS)*, 2022, pp. 1–4.