Towards the optimal orchestration of service function chains to enable ultra-reliable low latency communication in an NFV-enabled network

Von der Fakultät für Ingenieurwissenschaften,
Abteilung Informatik und Angewandte Kognitionswissenschaft
der Universität Duisburg-Essen

zur Erlangung des akademischen Grades

Doktor der Ingenieurswissenschaften

genehmigte Dissertation

von

Sam Erbati

aus

Teheran, Iran

1. Gutachter: Prof. Dr. Gregor Schiele
2. Gutachter: Prof. Dr. Christian Becker

Tag der mündlichen Prüfung: 29.01.2024

Abstract

The growing utilization of ***U**ltra-**R**eliable **L**ow **L**atency **C**ommunication* (*URLLC*) in 5G/6G networks, the **I**nternet **o**f **T**hings (IoT), and fixed-line networks has considerably increased the significance of **reliability** and **latency** requirements within the telecommunications sector. **C**ommunication **S**ervice **P**roviders (CSPs) encounter emerging challenges in optimizing reliability and latency to support ***U**ltra-**R**eliable **L**ow **L**atency **A**pplications* (*URLLA*). These applications include autonomous driving, remote surgery, tele-operated driving, and virtual reality. Simultaneously enhancing both reliability and latency poses a significant challenge, as enhancing reliability may potentially lead to increased latency. Furthermore, the limited availability of physical network resources increases the complexity of this endeavor.

Network **F**unction **V**irtualization (NFV) is a promising technology that has the potential to overcome some of the limitations associated with conventional network architectures, thereby enabling *URLLC*. The integration of NFV with **S**oftware-**D**efined **N**etworking (SDN) represents a revolutionary technological advancement that has the capacity to fundamentally transform existing network designs. NFV is the deployment of network functions as virtual software running on standard hardware. By decoupling network functions from dedicated hardware in NFV, greater network performance and management flexibility can be achieved. NFV relies heavily on **S**ervice **F**unction **C**hain (SFC) deployment to realize network services. SFC refers to delivering a network service to a customer, which requires that different network functions be concatenated in a specific order. See Chapter 1 for more information. Although NFV is a promising technology for providing elastic network services, it is important to note that there are several concerns related to its reliability and service quality. This creates a new research problem known as the **SFC deployment problem**. This problem is concerned with chaining **V**irtual **N**etwork **F**unctions (VNFs)

while meeting SFC requirements such as latency, reliability, physical resource consumption, power consumption, etc. CSPs must have optimal and efficient SFC embedding techniques for embedding SFC requests to enable *URLLC*. Chapter 1 provides more information.

The goal of this study is to address *URLLC* in an NFV-enabled network. After analyzing state-of-the-art studies in the field of NFV (see Chapter 2), we identified a crucial research obstacle. Consequently, we defined our goal to simultaneously optimize reliability and latency in the SFC deployment phase. We offer a novel and efficient SFC embedding technique that aims to enhance the reliability and latency of *URLLA* simultaneously. Mathematically, we formulate the SFC deployment problem as an integer-linear-programming optimization model to obtain exact numerical solutions. More information can be found in Chapter 4. In our optimization model, we propose an adjustable *priority coefficient factor* and *flow prioritization* to reserve a portion of physical network resources (bandwidth, RAM memory, and CPU) exclusively for embedding *URLLA* to significantly optimize their deployment paths. Since obtaining exact numerical solutions is time-consuming, we provide a set of heuristics and relaxed versions for addressing the scalability issue, reducing execution time, and producing results that are close to optimal for large-scale network topologies. Chapter 5 provides further information about heuristic approaches. In this study, we explore both static and dynamic service function chaining; further information is provided in Chapter 1. The performance evaluations reveal that our proposed algorithms considerably outperform the existing approaches in terms of end-to-end delay, reliability, bandwidth consumption, and SFC acceptance rate. See Chapter 6 for more details.

Kurzfassung

Die zunehmende Nutzung von ***U**ltra-**R**eliable **L**ow **L**atency **C**ommunication* (*URLLC*) in 5G/6G-Netzen, dem Internet der Dinge (IoT) und Festnetzen hat die Bedeutung von **Zuverlässigkeits-** und **Latenzanforderungen** im Telekommunikationssektor erheblich gesteigert. Kommunikationsdienstleister stehen vor neuen Herausforderungen bei der Optimierung von Zuverlässigkeit und Latenz, um ***U**ltra-**R**eliable **L**ow **L**atency Applications* (*URLLA*) zu unterstützen. Zu diesen Anwendungen gehören autonomes und ferngesteuertes Fahren, Fernchirurgie und virtuelle Realität. Die gleichzeitige Verbesserung der Zuverlässigkeit und der Latenzzeit stellt eine große Herausforderung dar, da die Verbesserung der Zuverlässigkeit möglicherweise zu einer Erhöhung der Latenzzeit führen kann. Darüber hinaus erhöht die begrenzte Verfügbarkeit von physischen Netzwerkressourcen die Komplexität dieses Unterfangens.

Network **F**unction **V**irtualization (**NFV**) ist eine vielversprechende Technologie, die das Potenzial hat, einige der mit klassischen Netzwerkarchitekturen verbundenen Einschränkungen zu überwinden und damit *URLLC* zu ermöglichen. Die Integration von NFV mit **S**oftware-**D**efined **N**etworking (SDN) stellt einen revolutionären technologischen Fortschritt dar, der das Potenzial hat, bestehende Netzwerkdesigns grundlegend zu verändern. NFV ist die Bereitstellung von Netzwerkfunktionen als virtuelle Software, die auf Standardhardware läuft. Durch die Entkopplung der Netzwerkfunktionen von dedizierter Hardware in NFV kann eine höhere Netzwerkleistung und Managementflexibilität erreicht werden. NFV stützt sich in hohem Maße auf die Bereitstellung von **S**ervice **F**unction **C**hain (SFC), um Netzwerkdienste zu realisieren. SFC bezieht sich auf die Bereitstellung eines Netzwerkdienstes für einen Kunden, der die Verkettung verschiedener Netzwerkfunktionen in einer bestimmten Reihenfolge erfordert. Siehe Kapitel 1 für weitere Informationen. Obwohl NFV eine vielversprechende Technologie für die Bereitstellung von

elastischen Netzwerkdiensten ist, gibt es einige Bedenken hinsichtlich ihrer Zuverlässigkeit und Servicequalität. Daraus ergibt sich ein neues Forschungsproblem, das als "**SFC Deployment Problem"** bekannt ist. Bei diesem Problem geht es um die Verkettung von Virtual Network Functions (VNFs) unter Einhaltung spezifischer SFC-Anforderungen wie Latenz, Zuverlässigkeit, Verbrauch physischer Ressourcen, Stromverbrauch usw. Kommunikationsdienstleister müssen über optimale und effiziente SFC-Einbettungstechniken für die Einbettung von SFC-Anfragen verfügen, um *URLLC* zu ermöglichen. Kapitel 1 enthält weitere Informationen.

Das Ziel dieser Dissertation ist es, *URLLC* in einem NFV-fähigen Netzwerk zu adressieren. Nach der Analyse der neuesten Studien im Bereich NFV (siehe Kapitel 2) haben wir ein entscheidendes Forschungshindernis identifiziert. Daher haben wir unser Ziel definiert, die Zuverlässigkeit und Latenz in der SFC-Einführungsphase gleichzeitig zu optimieren. Wir bieten einen innovativen und effizienten SFC-Embedding-Algorithmus an, der darauf abzielt, die Zuverlässigkeit und die Latenzzeit von *URLLA* gleichzeitig zu verbessern. Mathematisch formulieren wir das SFC-Deployment-Problem als ein Optimierungsmodell der "Integer-Linear-Programming", um exakte optimale Ergebnisse zu erzielen (weitere Informationen finden Sie in Kapitel 4). In unserem Optimierungsmodell bieten wir eine *Priorisierung der Flows* an und schlagen einen einstellbaren *Prioritätsfaktor* vor, um einen bestimmten Anteil der physischen Netzwerkressourcen (Bandbreite, RAM-Speicher und CPU) ausschließlich für die Integration von *URLLA* zu reservieren. Da die Ermittlung exakter numerischer Lösungen zeitaufwändig ist, bieten wir eine Reihe von Heuristiken und Relaxed-Versionen an, um das Problem der Skalierbarkeit zu lösen, die Rechenzeit zu verringern und Ergebnisse zu erzielen, die für große Netzwerktopologien nahezu optimal sind. Kapitel 5 enthält weitere Informationen über heuristische Methoden. In dieser Studie untersuchen wir

sowohl das statische als auch das dynamische Service-Function-Chaining (weitere Informationen finden Sie in Kapitel 1). Die Leistungsbewertungen zeigen, dass die von uns vorgeschlagenen Algorithmen die bestehenden Methoden in Bezug auf die Ende-zu-Ende-Latenz, die Zuverlässigkeit, den Bandbreitenverbrauch und die SFC-Akzeptanzrate übertreffen; siehe Kapitel 6 für weitere Details.

Table of Contents

Acronyms

5G	The Fifth Generation (5G) of Mobile Communication
6G	The Sixth Generation (6G) of Mobile Communication
API	Application Program Interface
BFS	Breadth-First Search
BGP-LS	Border Gateway Protocol-Link State
BIP	Binary Integer Programming
BRAS	Broadband Remote Access Server
BSS	Business Support System
CAPEX	Capital Expenditure
CDN	Content Delivery Network
CPU	Central Processing Unit
CSP	Communication Service Provider
DAAS	Dynamic Application-Aware SFC
DC	Data Center
DDoS	Distributed Denial of Service
DPI	Deep Packet Inspection
EM	Element Management
ETSI	European Telecommunications Standards Institute
FAS	Fast Application-Aware SFC
FW	Firewall
GGSN	Gateway GPRS Support Node
GPRS	General Packet Radio Service
HW	Hardware
IDS	Intrusion Detection System
ILP	Integer Linear Programming

IoT	Internet of Things
ISP	Internet Service Provider
KPI	Key Performance Indicator
MANO	Management and Orchestration
MILP	Mixed Integer Linear Programming
NAT	Network Address Translation
NF	Network Function
NFV	Network Function Virtualization
NFVI	Network Function Virtualization Infrastructure
NORAAS	Near-Optimal Reliability- and Application-Aware SFC
NSF	Nearest Service-Function First
OAS	Optimal Application-Aware SFC
OPEX	Operational Expenditure
ORAAS	Optimal Reliability- and Application-Aware SFC
OSS	Operation Support System
PCEP	Path Communication Element Communication Protocol
PE Router	Provider Edge Router
PNF	Physical Network Function
QoE	Quality of Experience
QoS	Quality of Service
RAM	Random Access Memory
REST API	Representational State Transfer API
SDN	Software Defined Networking
SFC	Service Function Chain
SGSN	Serving GPRS Support Node
SW	Software
TM	Traffic Monitor
URLLA	Ultra-Reliable Low Latency Applications

URLLC	Ultra-Reliable Low Latency Communication
VM	Virtual Machine
VNF	Virtual Network Function
WAN	Wide Area Network

List of Figures

List of Tables

List of Publications

1. Sam Erbati (Mohammad Mohammadi Erbati) and Gregor Schiele, "*Application- and reliability-aware service function chaining to support low-latency applications in an NFV-enabled network*," in *IEEE NFV-SDN*, November 2021.

2. Sam Erbati (Mohammad Mohammadi Erbati), Mohammad Mahdi Tajiki, Faramarz Keshvari, and Gregor Schiele, "*Service function chaining to support ultra-low latency communication in NFV*," *IEEE-International Conference on Broadband Communications for Next Generation Networks and Multimedia Applications (CoBCom)*, p. 8, July 2022.
   [Wining Gold-Award at CoBCom2022]

3. Sam Erbati (Mohammad Mohammadi Erbati) and Gregor Schiele, "*A novel reliable low-latency service function chaining to enable URLLC in NFV*," *The 9th IEEE International Conference on Communications and Networking (IEEE ComNet'2022),* p. 8, 1-4 November 2022.

4. Sam Erbati (Mohammad Mohammadi Erbati) and Gregor Schiele, "*A novel dynamic service function chaining to enable URLLC in NFV*," *The 17th ConTEL – INTERNATIONAL CONFERENCE ON TELECOMMUNICATIONS,* p. 8, 11-13 July 2023.

5. Sam Erbati (Mohammad Mohammadi Erbati), Mohammad Mahdi Tajiki and Gregor Schiele, "*Service function chaining to support ultra-low latency communication in NFV*," *MDPI-electronics journal,* p. 26, September 2023.

List of Supervised Theses

1. Faramarz Keshvari. "*Investigation of the Reliability of Service Function Chains in an NFV-enabled Network"*. Bachelor's thesis, University of Duisburg-Essen, 2022.
2. Yuning Zhu. "*Investigation of online service function chaining to support low latency network services in an NFV environment"*. Master's thesis, University of Duisburg-Essen, 2023.

A cooperation between Telekom Deutschland GmbH and the University of Duisburg-Essen.

1 Introduction

The telecommunications sector is an essential component of the modern world and has continued to grow each year. Its infrastructure, which includes both wired and wireless networks, is responsible for the delivery of data communications to users. In the coming years, the demand for telecommunications infrastructure will be greater than ever, as each year brings new services with varying **S**ervice **L**evel **A**greements (SLAs). One of the main advancements in this regard is the emergence of ***U**ltra-**R**eliable **L**ow **L**atency **C**ommunication* (*URLLC*) services, which promise to deliver unprecedented levels of reliability and latency in data transfer. Each year, the telecom infrastructure must enable better communication, reduce latency, improve reliability, improve bandwidth, and boost connectivity speed. To stay competitive and ahead of the competition, **C**ommunication **S**ervice **P**roviders (CSPs) must adjust, adapt, and expand their network services, offerings, and business models. CSPs consist of Telecommunications, **I**nternet **S**ervice **P**roviders (ISPs), **D**ata **C**enters (DCs), Enterprises, and the Cloud, which supply and enable communication services. To this end, it is essential for CSPs to continually discover innovative, effective methods for meeting different SLAs [1, 2, 3].

To provide a network service to customers, it is necessary to direct their traffic to pass through different hardware middleboxes in a specified order. Traditionally, these hardware middleboxes providing **N**etwork **F**unctions (NFs) such as **T**raffic **M**onitor (TM), **F**ire**w**all (FW), **D**eep **P**acket **I**nspection (DPI), **N**etwork **A**ddress **T**ranslation (NAT), and **I**ntrusion **D**etection **S**ystem (IDS) are realized on dedicated hardware equipment, see Figure 1. Due to the fact that the hardware equipment is physically present, they are also known as **P**hysical

Network **F**unctions (PNF). In this arrangement, a PNF has access to the full CPU power and memory for the execution of its task, even if complete access is not necessary depending on network demand [4].

Figure 1. Traditional hardware middleboxes [5].

Traditional network deployment is expensive to implement, and the deployment of network devices is also expensive and requires a significant investment in hardware from the network operator. Its operational expenditure is also high, and it is complex to manage. Since network functions are hardware-based and implemented on the underlying infrastructure, adjusting the topology of a network service, such as adding or removing network functions, demands reconfiguring the underlying physical topology, which is a highly complex and error-prone process. Traditional networks are also inflexible; due to the strong coupling between network functions and physical topology, it is extremely difficult to reconstruct the network to accommodate a new topology or service. Changing the service's logical network topology necessitates configuration adjustments. Traditional network deployment suffers from inefficient resource

utilization. When constructing a network service, it is necessary to allocate additional network resources in the event that future resource demand increases. Due to a shift in traffic patterns, computing and bandwidth resources may be underutilized. Due to the dependence of network functions on topology, the reconfiguration of network function chaining in a production environment is both impractical and costly [6, 7, 8].

In recent years, with the rising usage of *URLLC* applications in 5G/6G networks, the Internet of Things (IoT), and fixed-line networks, such as autonomous vehicles, remote surgery, tele-operated driving, virtual reality, augmented reality, and industrial automation, reliability and latency requirements have become even more crucial for CSPs. To fulfill new applications needs and overcome the limitations of traditional network infrastructure, the telecom sector must advance. Network Function Virtualization (NFV) and Software Defined Networking (SDN) are two emerging technologies that have the potential to significantly transform the telecommunications sector. These technologies offer complementary solutions to address the evolving service demands and the limitations of existing network architectures. Providing a multipurpose platform that can handle a wide range of services with varying requirements over a shared infrastructure is one of the primary challenges. The implementation of a platform that enables the establishment of a virtual network for each service on a shared infrastructure can be achieved through the process of slicing. In the next section, we will go deeper into NFV and SDN and explain how they might overcome the limitations of traditional network architectures.

1.1 Network Function Virtualization

NFV is now recognized as a technology with the potential to revolutionize

traditional network design and overcome some of its limitations. In November 2012, NFV was established by the seven most important telecom network providers. The **E**uropean **T**elecommunications **S**tandards **I**nstitute [ETSI] has been producing NFV standards since 2012 and is currently recognized as the most influential standardization group in this sector. The NFV architecture provided by the ETSI contributes to NFV implementation standardization. To provide greater stability and interoperability, each architectural component is based on these standards. NFV is the implementation of specific network functions as virtual software operating on standard hardware. It replaces the dedicated hardware platforms with software implementations by **V**irtual **N**etwork **F**unctions (VNFs) in a virtualized environment. Figure 2 illustrates the network virtualization approach [9].

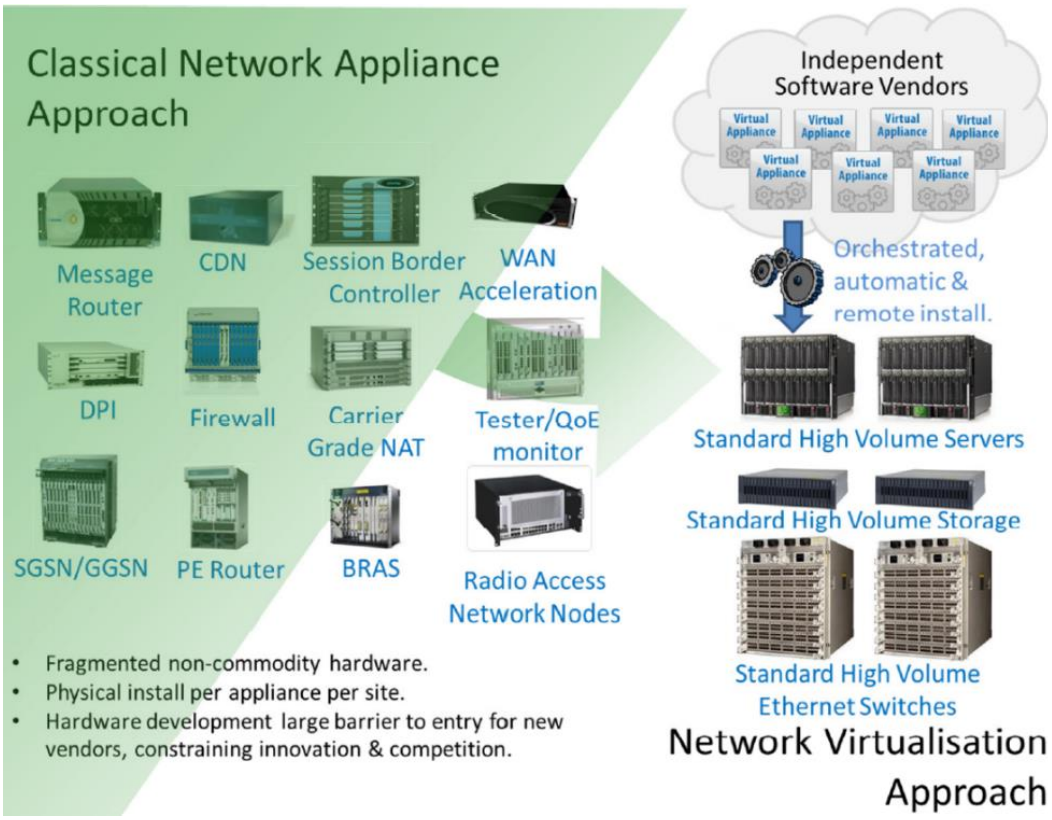

Figure 2. Network function virtualization [10].

VNFs are realized in software and placed on containers, general-purpose **V**irtual **M**achines (VMs), or CPUs on commercial-off-the-shelf equipment such as x86 servers within the cloud infrastructure. This indicates that network functions are executed as applications on virtual machines in **D**ata **C**enters (DCs). This changes how network operators may deliver their services. Depending on the needs, a network operator can operate the same function in a centralized cloud in order to save expenses or decentralized in a cloud edge closer to the user to minimize latency. Figure 3 and 4 show the transformation of network service delivery from a traditional network function approach to an NFV-based strategy. This change can be seen in both figures. Figure 3 depicts a traditional implementation of a network service in which each network function uses its own dedicated hardware. On the other hand, Figure 4 illustrates

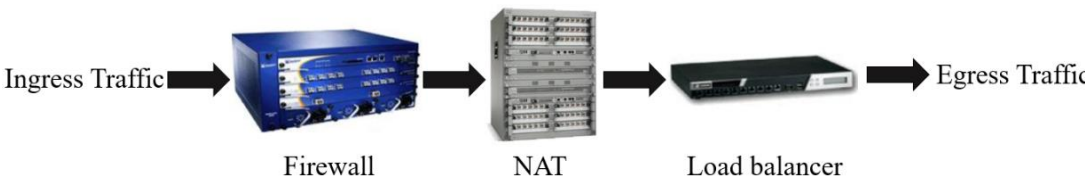

Figure 3. Conventional network service delivery [9].

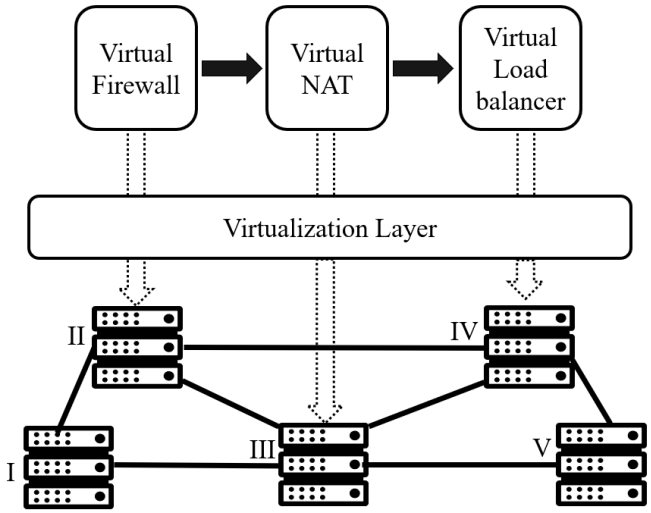

Figure 4. NFV based network service delivery [9].

an NFV-based approach, in which network services are virtualized and make use of generic hardware to provide the desired service [11, 12, 13].

NFV consequently benefits from all the advantages provided by virtualization, regardless of the application in question. The CPU and memory capacities can be adjusted with great flexibility. The ability to dynamically modify the resources allocated to a VNF based on the level of utilization improves the scalability of the VNF. A VNF can be provisioned far faster than a PNF due to the absence of mechanical work (transporting, configuring, and connecting the hardware to the network). This can be enhanced with automated processes that operate without human interaction and are coupled with considerable network simplification (operation). There is no need for mechanical intervention (field service) in the case of a failure. A VNF is entirely software. Therefore, the evolution of the VNF is independent of the hardware. Specifically, the developer of VNF software should be able to respond more quickly to the needs of his clients. The system is programmable via **A**pplication **P**rogram **I**nterfaces (APIs), making it accessible to third-party programs that, for instance, provide a control function. The following is a brief summary of some of the benefits of NFV implementation:

- Enhancing network flexibility
- Enhancing the utilization of the CPU and memory capacity of the hosts
- Enhancing the scalability of a VNF
- Enhancing the dynamic allocation of resources based on the demand for a VNF
- Enhancing centralized management of the NFV infrastructure
- Enhancing operational efficiencies
- Enhancing the pace of service innovation

- Reducing **Op**erational **Ex**penses (OPEX)
- Reducing **Cap**ital **Ex**penses (CAPEX)
- Enhancing the simplification of network operation and fault handling

Figure 5 presents the basic architecture of NFV proposed by ETSI, which consists of three main components [1]:

1. **V**irtual **N**etwork **F**unction (VNF)
2. **N**etwork **F**unction **V**irtualization **I**nfrastructure (NFVI)
3. NFV **Man**agement and **O**rchestration (MANO)

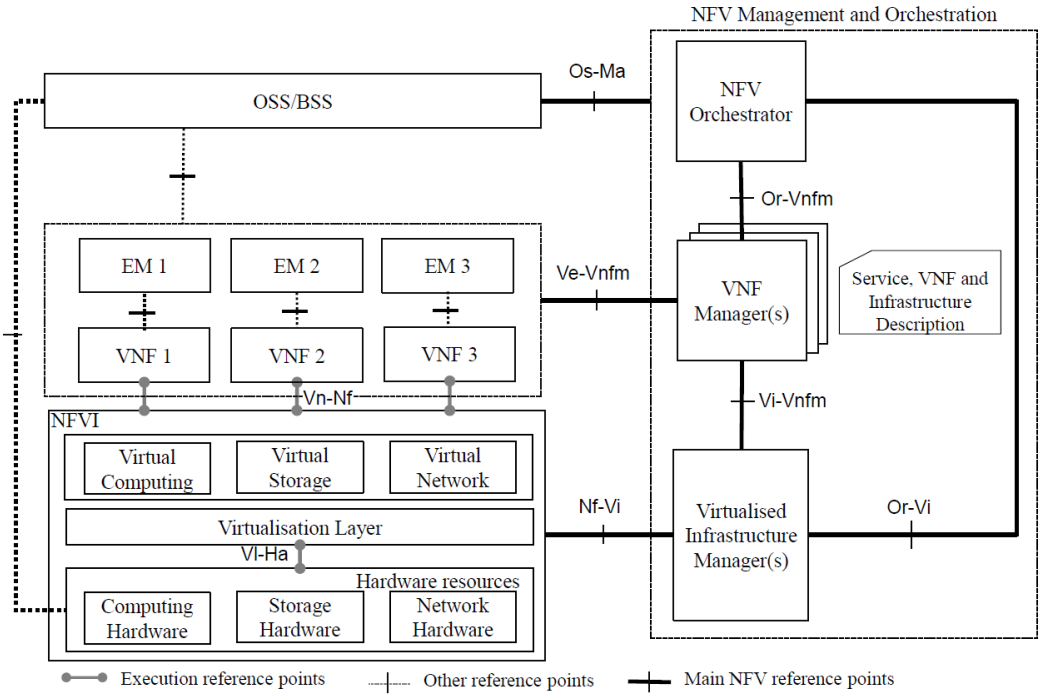

Figure 5. NFV reference architecture framework proposed by ETSI [1].

Virtual Network Function (VNF): A VNF is the basic block in the NFV architecture. It virtualized network functions such as the DHCP server, firewall, router, IDS, and DPI. It runs on one or more virtual machines on top of the hardware networking infrastructure. VNFs are deployed on-demand,

eliminating the deployment delays associated with traditional network hardware [14].

NFV Infrastructure (NFVI): NFVI is as important as any other functional block to realize the business benefits provided by the NFV architecture. It provides the real physical resources and associated software upon which VNFs can be installed. The NFVI generates a virtualization layer that is placed directly on top of the hardware and abstracts the HW resources. This allows the HW resources to be logically partitioned and delivered to the VNF so that it may carry out its functions. Building increasingly complex networks without the geographic limits of traditional network topologies requires NFVI, which is another reason why it is so important [14].

Management and Orchestration (MANO): NFV MANO controls resources, including the NFVI and VNFs, running in a virtualized data center, such as computation, networking, storage, and virtual machines. NFV MANO employs templates for standard VNFs to enable architects to select the necessary NFVI deployment resources. The following are the three functional areas that make up NFV MANO [15]:

1. **NFV Orchestrator:** It is responsible for the onboarding of VNFs, the management of their lifecycles, the management of global resources, and the validation and authorization of resource requests made by NFVI.
2. **VNF Manager:** It is in charge of managing the lifecycle of VNF instances, and it also plays a role in coordinating and adapting the configuration of NFVI and Element/Network Management Systems.
3. **Virtual Infrastructure Manager:** The NFVI computing, storage, and network resources are under the control and management of Virtual Infrastructure Manager.

As a result, because we are investigating the optimal orchestration of service function chains to improve network service quality, it is clear that our research is focused on the MANO component.

OSS (Operation Support System): It is a software component that gives a service provider the ability to manage, monitor, control, and analyze the services that are provided on its network. Together with a business support system, these software programs enable the majority of customer-facing operations, such as ordering, billing, and support [14].

BSS (Business Support System): Business **s**upport **s**ystems (BSS) are the conventional term for business and/or customer-facing functions. BSS platforms are utilized by service providers and telecom operators to supply revenue management, product management, customer management, and order management applications, which aid in the administration of their businesses [14].

EM (Element Management): Element Management provides standard management functions for a single or several VNFs. It is responsible for the lifecycle management of VNFs [14].

To realize network services, NFV primarily relies on the deployment of **S**ervice **F**unction **C**hains (SFCs). An SFC request is a sequence of VNFs that must be concatenated in a predefined order to deliver a network service to a client. Figure 6 illustrates an example of an SFC request comprising two VNF instances, namely VNFa and VNFc. These VNF instances are required to be concatenated in a predetermined sequence, with VNFa being processed first, followed by VNFc. The role of the SFC orchestrator is to effectively manage and direct network traffic in order to ensure the successful delivery of its designated service. NFV facilitates the virtualization of network functions, transforming them into VNFs. SFC allows various VNFs to be concatenated to

provide a network service. The SFC deployment problem is one of NFV's most significant problems, requiring further advancements [16, 17, 18, 19].

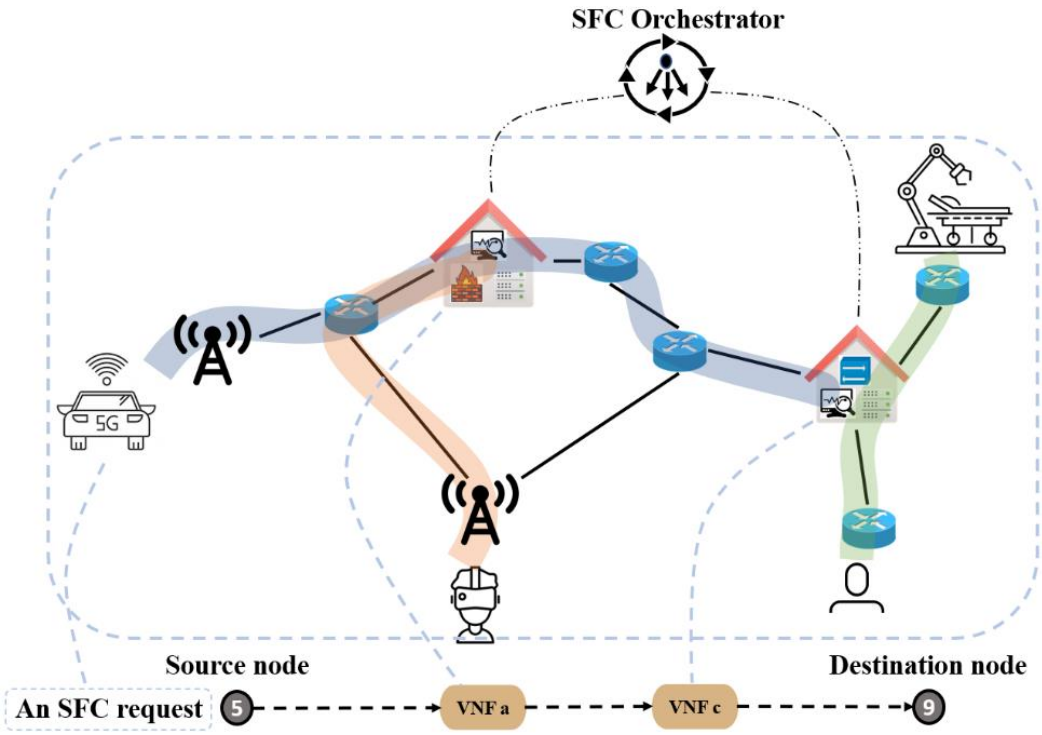

Figure 6. An example of an SFC request.

1.2 Software Defined Networking

Through the separation of data forwarding from network control, **S**oftware-**D**efined **N**etworking (SDN) facilitate centralized network management and dynamic reconfiguration, thereby providing improved agility, scalability, and programmability. SDN-enabled-networks support NFV to control the forwarding of traffic and reduce management complexity. SDN separates the data plane form the control plane. In SDN environment, the control plane is centralized and programmable, making network management and configuration more agile and dynamic. The control plane and data plane of network hardware

like switches and routers are intimately interwoven in traditional networking. The control plane assumes the responsibility of making determinations regarding the appropriate forwarding of network traffic, ascertaining the optimal route for packets, and maintaining the integrity of routing tables. In contrast, the data plane assumes the responsibility of effectively transmitting data packets by executing the directives provided by the control plane.

SDN involves the abstraction of the control plane from the underlying network devices and the centralization of control functions in a software-based controller. The act of separating network management facilitates a more adaptable and customizable method, enabling the ability to dynamically regulate network behavior. Network engineers are able to configure all of the different portions of the virtual network using a hypervisor or SDN controller, and they can even automate the process of establishing the network. Within minutes, IT administrators are able to set up a variety of components for the operation of the network. SDN decouples the control plane from the data plane. It enables a controller to centrally manage a network. A portion of the control plane (or, in severe circumstances, the complete control plane) is passed to the controller. The entire data plane stays on the PNFs. The controller interacts with the PNFs using southbound protocols such as OpenFlow, PCEP, BGP-LS, and a REST API. The controller can be programmed through a Northbound API. Figure 7 presents the basic architecture of SDN. Despite the fact that NFV and SDN are separate technologies, they are frequently used in conjunction to reap greater benefits and enhance overall network capabilities [20, 21].

After learning the fundamental information about NFV, SDN, VNF, and SFC and their relationships among one another, we go on to detail the research problem, the goal of this study, and our proposed solutions to overcome the research problem in the following sections.

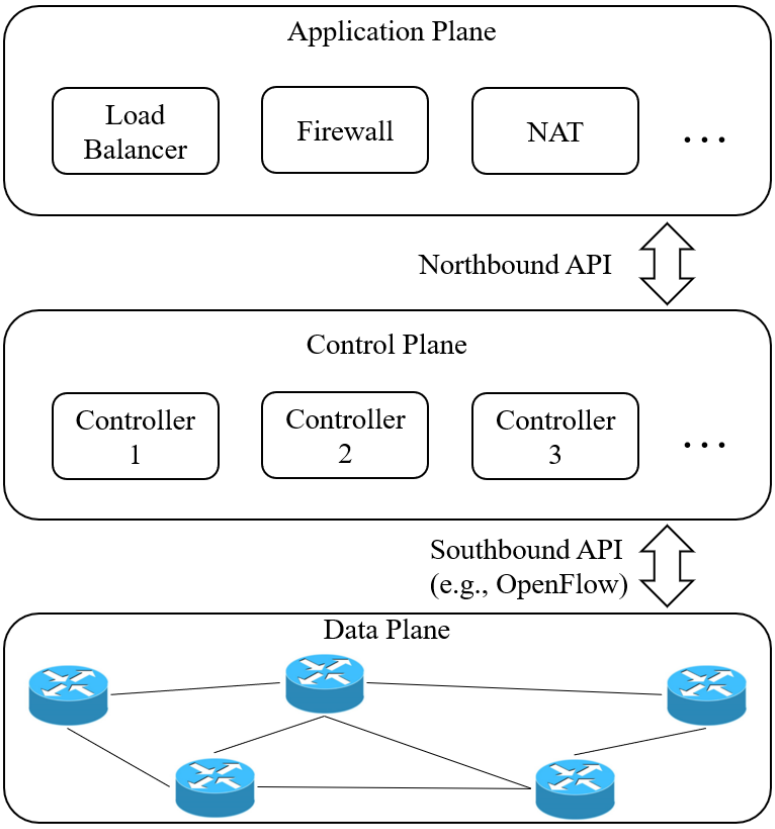

Figure 7. Architecture of SDN [22].

1.3 Research Problem

As stated in the preceding section, to implement network services, NFV strongly relies on SFC deployment. To deliver an end-to-end service to a client, an NFV-enabled network must direct traffic via several VNFs deployed on top of the virtualization layer. This creates a new research problem referred to as the **SFC deployment problem**. This problem is concerned with chaining VNFs while meeting SFC requirements such as latency, physical resource consumption, power consumption, etc. To address SFC deployment problem, a VNF chaining algorithm must be developed that is specifically engineered using the most efficient approach instead of a random process. Various studies have tackled the SFC deployment problem with different objectives (see Chapter 2),

but more developments are still required to enable *URLLC* in an NFV-enabled network [23, 24, 25].

In recent years, *URLLC* has attracted a great deal of interest due to the growing use of ***U**ltra-**R**eliable **L**ow **L**atency **A**pplications* (*URLLA*) such as autonomous vehicles, remote surgery, tele-operated driving, virtual reality, augmented reality, and industrial automation [26]. Consequently, addressing *URLLC* in NFV-enabled networks to optimize reliability and latency requirements has attracted significant research interest. It is extremely difficult to achieve both high reliability and low latency simultaneously. Increasing reliability may result in a rise in latency, and limited physical network resources make it even more challenging. The limited availability of VNF instances and physical network resources, including CPU, RAM memory, and bandwidth, within the CSP's network, adversely affects latency. By analyzing the recent studies, we discovered that backup techniques and using redundant components are often proposed to boost reliability, while latency-aware service function chaining is proposed to minimize latency. More information can be found in Chapter 2. To this end, we provide a novel solution to address *URLLC* in an NFV-enabled network without using backup techniques. In general, we focus on two challenges in our study: first, optimizing the concatenation of VNF instances with respect to **Q**uality **o**f **S**ervice (QoS) and utilization of physical network resources; and second, optimizing resource allocation with respect to the priority of SFC requests.

Figure 6 shows an example of an SFC request, which each SFC request originating from a source node and ending at a destination node. Each SFC request may contain several VNF types (e.g., a, b, c, d, etc.), which need to be concatenated in a specified order. We implement our proposed efficient physical resource allocation for SFC deployment by utilizing *flow prioritization* and a configurable *priority coefficient factor*. We classify SFC requests into those

with a high-priority (*URLLA*) and those with a low-priority. Then, we reserve an amount of physical network resources (bandwidth, RAM memory, and CPU) using the *priority coefficient factor* for SFC requests with a high-priority in order to improve their QoS explicitly. Figure 8 illustrates the physical link reservation exclusively for high-priority SFC requests. Memory and CPU usage follow the same principles as bandwidth utilization. We also studied the impact of varying the amount of reserved physical network resources for high-priority SFC requests. Chapters 3 and 4 provide further details about our proposed methodology. In this way, SFC requests with a high-priority can obtain a more optimal deployment path and a higher quality of service in terms of latency and reliability. We also take into account the maximum tolerable end-to-end delay and reliability requirements for both high- and low-priority SFC requests to minimize any negative side effects on low-priority SFC requests.

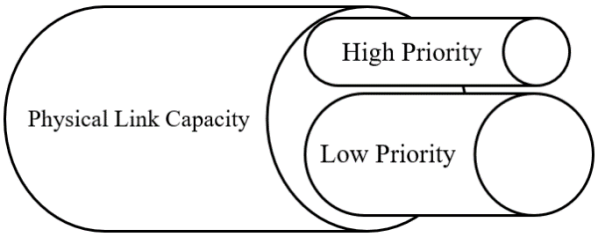

Figure 8. An example of a physical link reservation.

We mathematically formulate the SFC deployment problem as an **I**nteger **L**inear **P**rogramming (ILP) optimization model to obtain exact numerical solutions (see Chapter 4). Since obtaining the exact numerical results for large-scale network topologies is very time-consuming, we also offer a set of heuristic algorithms and relaxed versions to obtain near-optimal solutions in an acceptable time frame for large-scale network topologies. We describe the heuristics in Chapter 5. In terms of end-to-end delay, reliability, bandwidth usage, and SFC acceptance rate, the evaluation findings shown in Chapter 6 indicate that our proposed algorithms outperform the existing approaches. To

address the SFC deployment problem and enable *URLLC* in an NFV-enabled network, our study is divided into three phases:

1. ***Phase One: Ultra-Low Latency Communication* (*ULLC) Study*:** First, we focus on the latency requirement. Our goal is to decrease latency for latency-sensitive applications subject to physical network resource limitations. In this phase, we disregard the requirement for reliability. We provide a novel technique for minimizing latency in the SFC embedding phase, subject to limits on the maximum tolerable end-to-end delay and physical network resources. Chapters 4 and 5 provide more details.

2. ***Phase Two: Ultra-Reliable Low Latency Communication* (*URLLC*) *Study*:** We incorporate the findings from the first phase along with reliability constraints. Our goal is to optimize reliability and minimize latency for *URLLA*. As stated previously, backup techniques and using redundant components are typically proposed to boost reliability, but they may have a negative effect on latency and cause it to increase. It is contradictory to our goal. We propose a method for achieving our goal during the SFC embedding phase without a redundant component; more details can be found in Chapters 4 and 5.

3. ***Phase Three: Dynamic Service Function Chaining Study*:** In the first two phases of our study, we deal with static SFC requests, similar to studies in [27, 28] that imply SFC requests are static inputs and do not include arrival and departure timings (lifetime). In the third phase of our research, we examine dynamic service function chaining, similar to studies in [29, 30]. In this phase, we apply our findings from the second phase of our study to a dynamic SFC embedding scenario. In this case, we address *URLLC* in an

NFV-enabled network in a dynamic scenario where each SFC request has an arrival and departure time, or lifetime, to use physical network resources to deliver its services; once the lifetime expires, the physical network resources are made available for the subsequent SFC request. Chapter 5 provides more information.

1.4 Scientific Contributions

We did an in-depth analysis of *URLLC* in an NFV-enabled network by analyzing several state-of-the-art studies (see Chapter 2) and taking into account open issues that require further development. The main goal of this dissertation is to present a meticulously crafted methodology for the SFC deployment problem to enable *URLLC* in an NFV environment. We made a number of contributions to the relevant scientific communities, which are outlined below:

1. **A novel and efficient service function chaining methodology**: Using *flow prioritization* and a configurable *priority coefficient factor* to reserve physical network resources (bandwidth, RAM memory, and CPU), we propose a novel and efficient SFC embedding approach that simultaneously minimizes latency and optimizes reliability in the SFC embedding phase.

2. **Mathematical formulations of the SFC deployment problem:** We mathematically formulate the SFC deployment problem as an ILP optimization model in order to find exact numerical solutions. To this end, we consider the maximum tolerable end-to-end delay, the consumption of physical network resources (bandwidth, RAM memory, and CPU), reliability, and routing-related constraints.

3. **Set of heuristic approaches**: We provide a set of heuristic algorithms and relaxed versions for minimizing the execution time of the ILP optimization model, making it applicable to large-scale network topologies, and achieving near-optimal solutions with a minimum optimality gap.

4. **Conducting detailed performance evaluations:** We conduct a detailed examination of our proposed SFC embedding algorithms and demonstrate that they significantly improve the end-to-end delay, reliability, bandwidth consumption, and SFC acceptance rate compared to the existing algorithms.

1.5 Outline

The remaining chapters of the dissertation are organized as follows: In Chapter 2, we review related research and present an overview of state-of-the-art works on the subject of SFC deployment problem. In Chapter 3, we discuss the system model and assumptions behind our research. The fourth chapter specifies the problem statement and mathematical formulations, which are provided in the form of an ILP optimization model that takes diverse service requirements into account. The fifth chapter details the proposed set of heuristic algorithms for obtaining near-optimal solutions with minimal execution time and an optimality gap. In Chapter 6, simulation results demonstrating considerable improvements in terms of end-to-end delay, reliability, bandwidth consumption, and SFC acceptance rate are shown. In Chapter 7, we conclude our study and indicate a few areas requiring more investigation as future work. In the appendix, we describe the Python codes.

2 Related Work

In this chapter, we evaluate a number of relevant studies and emphasize their key findings. Although various studies have approached the issue of SFC deployment from various angles and with various goals, they need more advancements to enable *URLLC* in an NFV-enabled network. Prior to conducting a detailed analysis of relevant studies for each phase of our study, we will first present a concise overview of the primary distinctions among related studies. This will aid in enhancing our understanding of the various methodologies employed in related literature.

The present chapter is structured in the following manner: In Section 2.1, we provide a description of the primary distinctions between related studies and emphasize the key attributes of each. Section 2.2 provides a comprehensive review of the state-of-the-art studies associated with *Ultra-Low Latency Communication* (the first phase of our study). In Section 2.3, a concise overview of the scholarly investigations concerning *Ultra-Reliable Low Latency Communications* will be presented (the second phase of our study). In Section 2.4, an analysis will be conducted on the existing research related to dynamic service function chaining (the third phase of our study). In Section 2.5 of this chapter, we present an analysis of the novelty and significance of our research, highlighting its divergence from previous studies in the field.

2.1 Background Information

The SFC deployment problem is one of the primary challenges in NFV that must be improved further. Diverse approaches have been proposed to optimize the SFC deployment problem from different perspectives and with different objectives, but these approaches still require additional developments. Due to

the rapid growth of *URLLA* usage over the past several years, which requires extremely low latency and high reliability, we see a significant need to tackle emerging *URLLC* in NFV environments. In addition, the limited physical network resources of network providers make it more challenging. Various studies addressing service function chaining with low latency and high reliability are reviewed below. To better comprehend these studies, it is necessary to first emphasize and define their key distinctions; we detail some of these differences in the following subsections.

2.1.1 Optimization Models

One area of divergence among the different studies applies to the optimization model employed. The subsequent section presents a concise summary of prevalent optimization models utilized for addressing the SFC deployment problem. It outlines the fundamental characteristics associated with these approaches.

- ***Linear Programming (LP) Model***: Linear programming is a collection of mathematical and computational tools that enable you to discover a specific solution to this system that corresponds to the maximum or minimum of another linear function. Linear programming is a technique that was developed in the 1960s. In some of the research, the SFC deployment problem is formulated using an LP model. The mathematical method known as linear programming seeks to determine the values of decision variables in a way that maximizes or minimizes the value of an objective function while adhering to linear restrictions. It illustrates how the resources that are available may be utilized in the most effective way possible. On the other hand, mathematically determining the objective function and the restrictions is a tough task. There is a chance that the objective function and the restrictions will not be explicitly specified by linear in the equality of

equations, but there is also a chance that they will be. Linear programming is not suitable for problems where the decision variables must have integer or binary values.

- ***Integer Linear Programming (ILP)***: ILP is a kind of optimization problem with integer-valued variables, a linear objective function and equations. ILP is an extension of linear programming. LP solvers are only able to work with real numbers and cannot employ integers as variables. In instances where variables must be whole numbers, integer restrictions allow ILP to offer more realistic and useful answers. The complexity of solving integer programming problems is typically greater than that of standard linear programming problems, primarily due to the inclusion of discrete values, which introduces additional complexities. In many cases, these problems also become *NP-hard*, which makes them computationally difficult for large-scale instances [31].
- ***Mixed Integer Linear Programming (MILP)***: One of the most cutting-edge approaches to resolving the challenge of SFC deployment optimization is the MILP method. MILP is a method for solving optimization issues that is widely used since it is highly desirable and guarantees the discovery of global optimality in linear problems, and has efficient solvers that are commercially available. However, the MILP formulation suffers from a number of severe drawbacks, the most notable of which are its inability to take into account nonlinear effects and its need that all time periods be considered concurrently. Despite the fact that you can approximate non-linear functions using piecewise linear functions, employ semi-continuous variables, represent logical restrictions, and more, there are some non-linear functions that cannot be approximated. The complexity of a problem increases as the combination of continuous and integer variables is introduced, leading to computational demands for MILP, particularly in the

case of large and intricate models. In comparison to conventional linear programming, solving MILP problems may require advanced optimization solvers and take longer.

- ***Binary Integer Programming (BIP):*** With binary integer programming, optimization entails expressing a problem as a mathematical model that can be resolved. Binary integer programming involves decision variables that are restricted to taking on only two values: 0 or 1. This characteristic makes it particularly suitable for addressing problems that involve binary decisions. Consequently, the representation and interpretation of problems in binary integer programming are simplified and more straightforward. It becomes more difficult to tackle large-scale problems as the problem's combinatorial complexity rises with the number of binary variables.
- ***Markov Decision Process (MDP):*** A decision-making paradigm called the Markov Decision Process focuses on maximizing expected cumulative rewards over time in unpredictable circumstances. MDP is suitable for decision-making under uncertainty and reinforcement learning because it offers a framework for determining the best policy that maximizes the predicted cumulative reward. MDP problems can exhibit computational complexity, particularly in cases where the state and action spaces are extensive. This characteristic poses a significant difficulty in the search for optimal solutions. In certain scenarios, the state space may exhibit extensive characteristics, resulting in what is commonly referred to as the "curse of dimensionality." This phenomenon entails a significant escalation in computational resources needed to address the problem, which grows exponentially in relation to the quantity of states involved [34].

2.1.2 Shortest Path Algorithms

Utilizing a proper algorithm to determine the shortest path between two

nodes in a network topology is one of the major factors that contribute to getting the optimum results. In a range of studies, researchers have made use of a number of different algorithms in order to identify the shortest path. Following is a brief summary of the most common algorithms:

- ***Breadth-First Search (BFS) Algorithm***: The BFS is a graph traversal algorithm that begins traversing the graph at the root node and searches all adjacent nodes. When working with graphs that do not have weights assigned to their edges, one of our primary concerns is constantly attempting to cut down on the total number of edges that have been traversed. As a consequence of this, we are certain that all of the immediate neighbors of the source node have a distance that is equal to one. The following item that we are able to determine with absolute certainty is that all of the nodes that are considered to be the source node's second neighbors have a distance that is equal to two, and so on. The BFS algorithm is applied to a straightforward queue that we employ. The overall time complexity is denoted by the notation O(V+E), where V represents the number of vertices and E represents the total number of edges in the graph [32].
- ***Dijkstra Algorithm***: When dealing with weighted graphs, it is not required that adjacent nodes always have the shortest path between them. On the other hand, the neighbor whose edge is the shortest is inaccessible by any route that is shorter. The reason for this is that every other edge has a greater weight, and passing through any of those edges would increase the distance traveled significantly. This concept is used by *Dijkstra*'s algorithm to devise a *Greedy* solution to the problem. At each stage, we select the node that has the path with the least distance. After adjusting this cost, we add the neighbors of this node to the queue. As a result, the queue has to have the capacity to rank the nodes included in it according to the lowest possible cost. To accomplish this goal, we may think about employing a priority

queue. Because we are only going to each node's neighbors one time, this means that we are only going to visit edges once. Additionally, we have the option of utilizing a priority queue that has a time complexity of O(logN) for both the push and pop operations. As a result, the overall time complexity is denoted by the notation O(V+E(logV)) [32].

- ***Greedy Algorithm***: It is an algorithmic paradigm that creates a solution piece by piece, constantly picking the next piece that delivers the greatest evident and immediate benefit. Therefore, *Greedy* is best suited for solving situations in which selecting a locally optimal solution also leads to a solution at the global level. The *Greedy* technique is the strategy that is the least complicated and most direct. It is not an algorithm, but rather an approach that may be used instead. The most important feature of this technique is that it allows a choice to be made on the basis of the information that is now accessible. The choice is taken regardless of the knowledge that is currently available, and there is no concern given to how the current decision may affect events in the future. The total amount of time required to complete the task is O(N × logN). It is quite tempting to adopt this method because of its space and time complexity; nevertheless, there are no assurances that it would deliver the most ideal accumulated reward. Despite the fact that it is difficult in both space and time, it is very tempting to apply it [33].

Consequently, we made the decision to utilize the *Dijkstra* algorithm in our study, as it enables us to acquire the global optimal shortest path while taking into account the propagation delay associated to the links. Following an exploration of diverse strategies employed to tackle the issue of SFC deployment, we proceed to examine the latest scholarly contributions in the subsequent sections.

2.2 Ultra-Low Latency Communication

As the number of applications that require ultra-low latency increases, finding a solution to the SFC deployment problem has become both more important and more difficult, calling for more advancements. Following are a variety of approaches that have been proposed as a means of assisting ultra-low latency applications. To enable latency-aware service function chaining, the great majority of these approaches only consider the latency requirement of SFC requests in conjunction with other objectives and constraints and choose the shortest provisioning path that is still feasible.

Sun *et al.* [28] conducted research on low-latency and resource-efficient orchestration of SFCs in an NFV environment. They offered an **SFC** **D**eployment **O**ptimization (SFCDO) algorithm based on a BFS algorithm for determining the shortest path between the source node and the destination node for all SFC requests and preferentially selecting the path with the shortest hops to minimize the end-to-end delay. They compared the performance of their proposed algorithms to the *Greedy* algorithm and the simulated annealing algorithm and found that their proposed algorithms performed better in terms of the average end-to-end delay and the average bandwidth consumption while dealing with SFC requests of varying lengths and quantities. Alameddine *et al*. [34] studied low-latency service schedule orchestration in NFV-based networks. They address the **L**atency-**A**ware **S**ervice **S**chedule **O**rchestration (LASSO) problem, which tackles the mapping and scheduling of services to VNFs. They describe the problem as a MILP optimization model and provide ENCHAIN, a unique game-theoretic technique that exploits a scalable solution for the LASSO problem while allowing each network service to choose its own mapping and scheduling solution.

Harutyunyan *et al*. [35] investigated latency-aware service function chain placement in 5G mobile networks. Utilizing ILP techniques, they formulate and

solve a joint user association, SFC placement, and resource allocation problem in which SFCs consisting of virtualized service functions represent user-requested services with specific E2E latency and data rate requirements. In particular, they evaluate three implementations of an ILP-based method designed to reduce E2E latency of requested services, service provisioning cost, and VSF migration frequency, in that order. The authors then present a heuristic for addressing the scalability problem of ILP-based solutions. Results from simulations illustrate the efficacy of the suggested heuristic method. Sun *et al*. [36] investigated a cost-efficient SFC orchestration for low-latency applications in NFV networks. They introduced a heuristic approach called **C**losed-**L**oop **F**eedback (CLF) that was designed to determine the shortest route to map an SFC request while also taking into account the amount of resources that may be saved. The performance of their algorithm was superior to that of two of their rivals in terms of communication latency and deployment time. In [30], they examined energy-efficient SFC provisioning with the goal of enabling delay-sensitive applications inside an NFV setting. For dynamic SFC deployment, they present an **E**nergy-**A**ware **R**outing and **A**daptive **D**elayed **S**hutdown (EAR-ADS) algorithm. Latency- and capacity-aware placement of chained virtual network functions was investigated by Hmaity *et al*. [37]. They address two fundamental problems. The first consists of determining where VNFs will be hosted (i.e., VNF placement), and the second consists of determining how to properly direct network traffic to traverse the necessary VNFs in the correct order (i.e., routing), thereby provisioning network services in the form of SFCs. They presented and contrasted a variety of heuristic techniques with regard to the lag time of the links and the computational power of the nodes.

In an SDN-based network, Tajiki *et al.* [27] conducted research on service function chaining that was simultaneously energy-efficient and QoS-aware (latency-aware). They took into account limitations on the maximum amount of

end-to-end delay that could be tolerated, as well as link utilization and server utilization. In order to do this, they model the problems of VNF placement, allocation of VNFs to flows, and flow routing as ILP optimization problems. Because the formulated problems cannot be solved (using ILP solvers) in timescales that are acceptable for realistic problem dimensions, they design a set of heuristics to find near-optimal solutions in timescales that are suitable for practical applications. These heuristics allow us to solve the problems in a manner that is acceptable for real-world applications. They carry out a numerical analysis to determine how well the suggested algorithms function across a real-world topology and in a variety of network traffic patterns. Their findings demonstrate that the suggested heuristic algorithms may produce near-optimal solutions (with an optimality-gap of no more than 14%), and that their execution duration makes them suitable for use in actual networks. Li *et al*. [38] did a study on cost- and QoS-based NFV service function chain mapping mechanisms. They proposed a *Greedy* algorithm for service mapping. They examined cost- and QoS-based NFV service function chain mapping mechanisms and proposed a mathematical model with the goal of cost optimization and QoS assurance. They achieved higher deployment benefits while ensuring QoS requirements.

Fountoulakis *et al*. [39] did an end-to-end performance analysis for service chaining in a virtualized network. They note that the outcomes of the simulation and the analysis are consistent with each other. They provide insights for the decision-making process on traffic flow control and its influence on crucial performance indicators by assessing the system in a variety of different situations and then providing those results. Han *et al*. [40] studied a service function chain deployment method based on network flow theory for load balance in operator networks. They proposed an algorithm to meet the demands of load balance, low delay, and efficient utilization of substrate resources in

operator networks. Wang *et al*. [41] studied service function chain composition and mapping in NFV-enabled networks. Utilizing the resource efficiently is one of the greatest obstacles to adopting SFC. In this study, they examine the composition and mapping of the SFC in consideration of resource optimization. The SFC composition and mapping issue is represented as a weighted graph matching problem. Then, they present a Hungarian-based method to coordinately solve the SFC composition and mapping problem. In [42], Pham *et al*. studied traffic-aware and energy-efficient VNF placement for service chaining. Luizelli *et al*. [43] proposed a heuristic approach for VNF placement and chaining that aims to minimize required resource allocation while meeting network flow requirements. Additionally, we profited from research in [44, 45, 46], which give a good understanding of how to approach *URLLC* in NFV with a wide range of objectives.

2.3 Ultra-Reliable Low Latency Communication

In the second phase of our study, we take into account the reliability constraint so that we may satisfy the criteria of *URLLC* in a network that is enabled for NFV. There are various ways to raise reliability, and while a backup technique is typically suggested as a way to increase the reliability of various components in an NFV environment, we will go through some of the other available options in the following. In the event that either the hardware or the software fails, a redundant component can be swapped in its place in order to extend the amount of time the system is available for use. However, enhancing the system's reliability by adding features that are redundant may have the opposite effect on latency and cause the system to become more complex. This method also necessitates additional physical network resources, which is a disadvantage. In what follows, we will take a look at some of the most recent and cutting-edge studies that deal with the issue of reliability constraints and

make an effort to increase reliability.

Zhou *et al.* [47] looked at the possibility of parallelizing network functions to achieve high reliability and low latency in service delivery. They were able to increase the reliability of the service by adding backup VNF nodes, while simultaneously reducing the flow latency through the use of parallel network function processing. They decided to approach the issue by posing it as an integer programming problem with the objective of reducing the amount of reserved computation and bandwidth resources while maintaining the same level of end-to-end latency and service reliability. They solved the issue by modeling it as a MDP model and then applying a reinforcement-learning method to it. Yin *et al.* [48] did a study on the SFC placement problem while guaranteeing availability. They are concentrating on finding a solution to the issue of SFC placement inside the Mobile-Edge-Computing-NFV system while maintaining reliability. They came up with a backup model to increase the availability of SFC. In addition to this, they provide a placement strategy for SFC that is based on **D**ynamic **P**rogramming (DP). The results of the evaluation reveal that their proposed solutions perform better than the existing techniques with regard to the guarantee of availability and the optimization of resources.

The topic of reliability-focused and resource-efficient SFC construction and backup was discussed in the paper by Wang *et al.* [49]. They investigated how the building phase of the SFC affected the reliability of the system. In order to combine several SFCs into a **S**ervice **F**unction **G**raph (SFG) and perform reliability screening for the SFG set, they presented an algorithm called the **I**nstance-**S**haring and **R**eliable **C**onstruction **A**lgorithm (ISRCA). After that, an algorithm for ranking nodes that takes into account centrality and reliability, called NRCR is suggested for use in selecting backup nodes. Qu *et al.* [50] investigated reliability-aware VNF chain placement and flow routing optimization. This study presents an in-depth analysis of simultaneous VNF

chain placement and flow routing optimization that takes into account the importance of reliability. An incremental method is provided as a means of determining the number of necessary VNF backups. This is done with the intention of ensuring the requisite level of reliability. This study argues for the existence of a VNF assignment technique that is based on the sharing of resources and is capable of trading off all of the reliability, bandwidth, and computing resources that are consumed by a particular service chain. It is suggested to use a heuristic in order to get around the complexity of the already formulated ILP model.

In [51], Kaliyammal-Thiruvasagam *et al.* studied the reliability-aware, delay-guaranteed, and resource efficient placement of SFCs in softwarized 5G networks. Within the scope of this study, the authors tackle the challenge of solving the reliability-aware, delay-guaranteed, and resource-efficient SFC placement problem in 5G networks that are being softwarized. First, in order to improve the reliability of an SFC that does not use backups, they suggest a new way of subchaining an SFC. They add backups to the VNFs in order to fulfill the reliability requirement if, after using the subchaining approach, the criterion for reliability is not met. After that, they turn the problem of reliable SFC placement into an ILP problem in order to solve it in the most effective manner possible. They offer a modified stable matching technique in order to deliver a near-optimal solution in polynomial time. This is in response to the significant computational cost of the ILP issue, which requires the resolution of large input instances. In [52], they initially investigate latency aware and reliable SFC placement in order to suit the expectations of users and improve the reliability of SFCs from VNF failures. Then, they concentrate on the reliable placement of virtual monitoring functions near VNFs in order to discover and mitigate service degradation and security-related concerns in the network. In order to reduce the overall deployment cost, they formulate the problems as ILP problems and

demonstrate that they are NP-hard. They offer new heuristic algorithms based on complex network theory to deliver near-optimal solutions in polynomial time for large input cases in order to overcome the high computational complexity of ILP issues. They demonstrate, via extensive simulations, that their suggested algorithms give a near-optimal (5% optimality gap) solution in a real-world network design.

Lin *et al.* [53] investigated the reliability of service provisioning in a **M**obile Edge **C**omputing (MEC) network by utilizing redundant placement of VNF instances. They assumed that each service request included a SFC requirement and a service reliability requirement. They created a unique reliability-aware service function chain provisioning problem with the objective of maximizing the number of requests accepted while satisfying the reliability requirements of each admitted request. When the problem size was small, they created an ILP solution and offered a heuristic method for addressing scalability.

2.4 Dynamic Service Function Chaining

In the third phase of our study, we review the most recent research concerning dynamic service function chaining. Following is a summary of dynamic service function chaining research. In dynamic SFC embedding studies, every SFC request has an arrival and departure time in order to use physical network resources. SFC requests enter and exit the network at distinct intervals. When the lifetime of SFC requests expires, the resources become accessible for subsequent SFC requests.

Chen *et al.* [54] studied cost-efficient dynamic service function chain embedding in edge clouds. **E**dge **C**omputing (EC) provides delay protection for some delay-sensitive network applications by putting limited-resource cloud infrastructure at the network's edge. In this research, the authors investigate how

to dynamically embed SFC in a geo-distributed edge cloud network to fulfill user requests with varying latency requirements, and present this problem as a MILP model with the goal of minimizing the overall embedding cost. In addition, a unique **SFC C**ost-**E**fficient em**B**edding (SFC-CEB) technique has been presented to embed the required SFC efficiently and optimize embedding cost. Based on the findings of trace-driven simulations, the suggested approach can lower the cost of SFC embedding by up to 37% when compared to existing techniques (e.g., RDIP). Qin et al. [55] investigated dynamic service chaining for ultra-reliable services in softwarized networks. They presented a dynamic service chaining framework for the delivery of ultra-reliable services, where the reliability is described by the probability distribution utilizing extreme value theory. Their design purpose is to limit the number of backup VNF modules subject to resource and reliability restrictions. To deliver ultra-reliable services, main and backup VNFs are re-mapped to more reliable physical machines due to the network's dynamic nature. Utilizing Lyapunov stochastic optimization, the primary VNF mapping and backup VNF selection are conducted on large and small timescales, respectively. Shang *et al*. [56] studied online SFC placement for cost-effectiveness and network congestion control. They offered a novel online technique that reduces operating costs and controls network congestion at the same time. It accomplishes this by co-locating VNFs and routing flows among them. They formulated it as an ILP optimization problem. They also propose a heuristic algorithm named **C**andidate **P**ath **S**election (CPS) algorithm with a theoretical performance guarantee.

Luo *et al.* [57] did a study on an online algorithm for VNF service chain scaling in datacenters. They offer an online scaling technique to adapt the deployment of VNF instances to the fluctuating traffic demand over time, ensuring a competitive advantage. They illustrate the efficiency of the proposed online VNF scaling method via theoretical analysis and trace-driven simulation.

Pei *et al.* [29] addressed efficiently embedding SFCs with dynamic VNF placement in geo-distributed cloud systems. They formulated it in the form of a BIP optimization model, aiming to minimize the embedding cost. Their proposed approach enhances performance in terms of SFC request acceptance rate, network throughput, and mean VNF utilization rate. Liu *et al.* [58] did a study on dynamic service function chain deployment and readjustment. They examine how to optimize SFC deployment and readjustment in a dynamic setting. In particular, they attempt to simultaneously optimize the deployment of new users' SFCs and the readjustment of existing users' SFCs, taking into account the trade-off between resource usage and operational overhead. First, an ILP model is constructed to solve the issue precisely. Then, they propose a **C**olumn **G**eneration (CG) model for the optimization to lower the time complexity. Simulation findings demonstrate that the proposed CG-based algorithm may approximate the performance of the ILP optimization model and outperform an existing benchmark in terms of service provisioning profit.

Li *et al.* [59] studied an efficient algorithm for service function chains reconfiguration in mobile edge cloud networks. As an emerging network architecture, MEC enables ultra-low latency and high-bandwidth applications by putting servers at the network's edge to provide compute and storage capabilities. Their study focuses on the SFCs reconfiguration scheme with resource capacity limits in the MEC network. First, they define the SFCs reconfiguration problem of the edge network as a mathematical model with the goal of minimizing end-to-end delay and operating costs for user services. Then, they turn the problem into an analogous shortest path problem and create a **D**ynamic **P**rogramming based **S**FC **M**igration algorithm (DPSM). Lastly, simulated tests are conducted to evaluate the performance of the method using a real-world dataset. The outcomes of the trial demonstrate the algorithm's usefulness and efficiency.

2.5 Conclusion

To conclude this chapter, after analyzing a number of state-of-the-art studies with diverse objectives, we identified a lack of research on addressing *URLLC* in an NFV-enabled network. It has been observed that, in order to enhance reliability, it is common practice to utilize a backup technique. Similarly, to enhance latency, a commonly employed approach is the implementation of latency-aware service function chaining. Nevertheless, using a backup method may result in an increase in latency. Adding extra physical network resources is not something we want to do; therefore, having a backup element is also undesirable. To this end, we constructed a novel solution to address *URLLC* in an NFV-enabled network. This allowed us to simultaneously improve the reliability and latency of *URLLA* without a backup technique while having a minimal negative impact on other applications.

Different from the aforementioned investigations, we not only provide a reliability- and latency-aware SFC embedding algorithm that simultaneously improves the reliability and latency of *URLLA* without using backup techniques and redundant components, but we also propose a configurable *priority coefficient factor* and utilize *flow prioritization* to provide *URLLA* with dedicated physical network resources (bandwidth, RAM memory, and CPU). We define constraints on maximum tolerable end-to-end delay, consumption of physical network resources, reliability, and routing-related constraints. We employ the *Dijkstra* algorithm, which is optimized for weighted graphs, to locate the shortest path between two nodes in order to arrive at the solution that is optimal on a global scale. To find the exact numerical solutions, we formulated the SFC deployment problem as an ILP optimization problem (see Chapter 4), and we provided a set of heuristic approaches and relaxed versions to minimize the execution time with a minimum optimality gap in order to make it usable for large-scale network topologies and solve the scalability issue.

3 System Model

This chapter illustrates the system model and its underlying assumptions for each phase of our research, which consist of the underlying substrate network, virtual network functions, and service function chains. To solve the SFC deployment problem, an accurate system model must be designed. The system's model plays a crucial role in producing accurate results. Here is how this chapter is structured: In Section 3.1, we present the system model of the *ULLC* study, which is the first phase of our study. In Section 3.2, we explain the system model for analyzing the *URLLC* study, which takes the reliability model into consideration (the second phase of our study). In Section 3.3, we detail the system model used to investigate the dynamic service function chaining to enable *URLLC* (the third phase of our study). In this setting, each SFC request has an arrival time and a departure time to use physical network resources. As stated before, in the first two phases of our research, we dealt with static SFC requests, similar to studies [27, 28] that suggest SFC requests are static inputs and unaffected by arrival and departure timings (lifetime). In the third phase of our research, similar to [29, 30], we investigate dynamic SFC requests. When an SFC request's lifetime expires, the physical network resources become accessible for the next SFC request.

3.1 Ultra-Low Latency Communication

To investigate ultra-low latency communication, we will first present the modeling of a physical network and its underlying assumptions. Gridnet [60] is the network topology that we employed for the first phase of our investigation. Figure 9 presents the Gridnet network topology, which consists of 8 nodes and 18 links. The modeling of physical networks, SFC requests, and the parameters

used to describe them are the subsequent topics discussed in the following subsections.

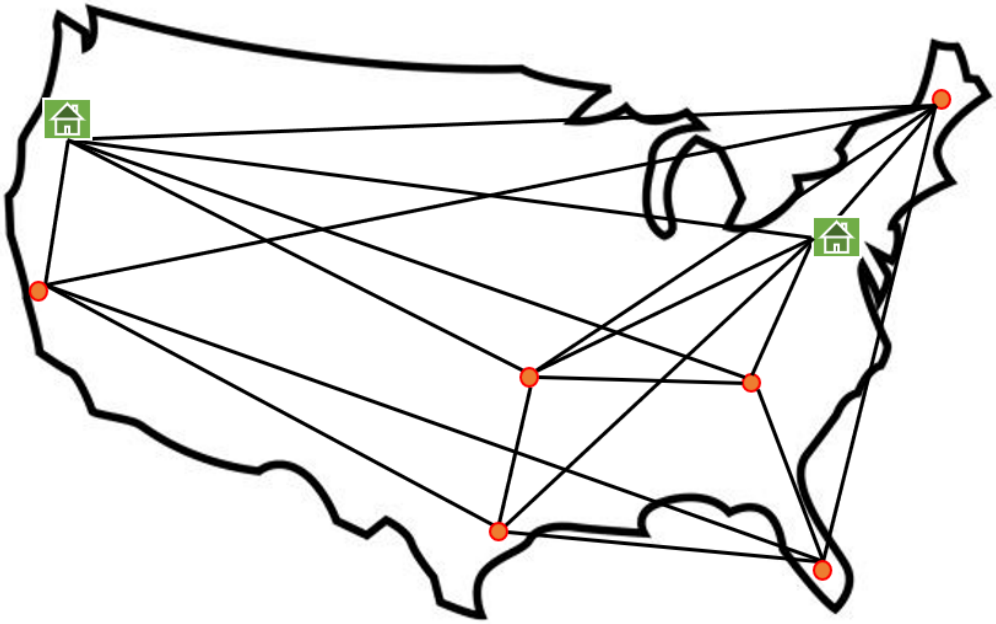

Figure 9. Gridnet network topology [60].

A. *Physical Network*

We represent the physical network as an undirected graph with the notation $G_P=(N_P,L_P)$, where N_P refers to the set of physical nodes and L_P refers to the set of physical links in the physical network. The matrix $PN_{Np \times Np}$ is employed to depict the underlying physical network. It serves as an adjacency matrix, representing the connections between vertices in the undirected graph. The vertices of the graph are represented by the rows and columns of the adjacency matrix, with each cell in the matrix denoting an edge connecting two vertices. We divide nodes into two classes. The first category of nodes is known as ***C**ore-**D**ata-**C**enter (**CDC**) **N**odes* (N_{CDC}), and these are the nodes that host the various VNF types (e.g., *a, b, c, d, etc.*). The second category, ***S**witching **N**odes* (N_S), are the nodes that only send traffic to the subsequent nodes (N_{CDC}, $N_S \subseteq N_P$)

and cannot host any VNF types. On the graph displayed in Figure 9, the *CDC* nodes are represented by the green squares, while the switching nodes are represented by the red dots. In this phase, the physical network and SFC requests are both treated as static inputs. As shown in Figure 9, we have configured two *CDC* nodes depending on node degree to host VNV instances and six switching nodes to pass traffic to the following nodes.

In order to represent a pair of nodes, we make use of the symbols *m* and *n (m, n ∈ Np)*. In our study, we establish constraints on bandwidth use, memory utilization, and CPU usage. To this end, the parameter $C_{(m,n)}^{bw}$ represents the bandwidth capacity of the physical link between node *m* and node *n*, the parameter C_m^{mem} denotes the memory capacity of node *m*, and C_m^{CPU} represents the CPU capacity of node *m.* In this study, since the switching nodes (N_S) only forward the traffic to the next nodes and do not host any VNFs, we assume that they do not require much CPU and memory capacity; therefore, the CPU and memory capacities of the switching nodes are considered infinity.

B. Service Function Chain Requests

An SFC request, denoted as *f*, refers to a sequential arrangement of VNFs that must be concatenated in a specific predetermined order, as depicted in Figure 6 on Page 10. In order to establish a clear understanding of SFC requests, it is necessary to have a collection of input parameters, which are commonly referred to as input matrices. The parameters employed to indicate an SFC request are enumerated as follows. We employ nine parameters to indicate SFC request *f*, $\{Src^f, Des^f, R_x^f, W_x^f, P^f, \tau_{bw}^f, \tau_{cpu}^f, \tau_{mem}^f, \tau_{td}^f\}$. We use Src^f and Des^f in order to represent the source and destination nodes of SFC request *f*. The matrix $R_{F\times X}$ represents the required VNFs for fulfilling SFC requests. In this study, we treat this matrix as a static input. *F* denotes the total number of SFC requests, and *f* is a single SFC request. *X* denotes the total number of VNF

types (e.g., *a, b, c, d,* etc.). In fact, if VNF type *x* is requested for the SFC request *f*, then R_x^f equals 1 (0 otherwise). Matrix $W_{F\times X}$ represents the sequence of required VNFs for all SFC requests, where W_x^f specifics the x^{th} required VNF for flow *f*. It is a matrix with integer values to define the order of each VNF *x*. Indeed, a VNF with lower index needs to deliver service first. P^f is used to denote the priority of SFC request *f*. As stated before, we define high-priority SFC requests for *ultra-low latency applications* (P^f is 1) and low-priority SFC requests for other applications (P^f is 0). The parameters $\tau_{bw}^f, \tau_{cpu}^f$ and τ_{mem}^f denote the required bandwidth, CPU, and memory for each SFC request *f*, respectively. Finally, τ_{td}^f denotes the maximum tolerable end-to-end delay of an SFC request *f*.

3.2 Ultra-Reliable Low Latency Communication

As indicated previously, in the second phase of our study, we study *URLLC* and incorporate SFC reliability requirements. The reliability of an SFC request is demonstrated in Figure 10. In this figure, the symbols Re_{cdc} and Re_{vnf} represent the reliability values for each *CDC* node and each VNF instance, respectively. The figure illustrates that the reliability of SFC request *s* containing VNFa and VNFb, which are deployed on *CDC* node 1, is determined by multiplying their respective reliability values. In our analysis, we take into account both software reliability (Re_{vnf}) and hardware reliability (Re_{cdc}). To examine *Ultra-Reliable Low-Latency communication*, we first outline the reliability viewpoint in our system model and then give the substrate network model and its underlying assumptions. Finally, the modeling of service function chain requests is presented.

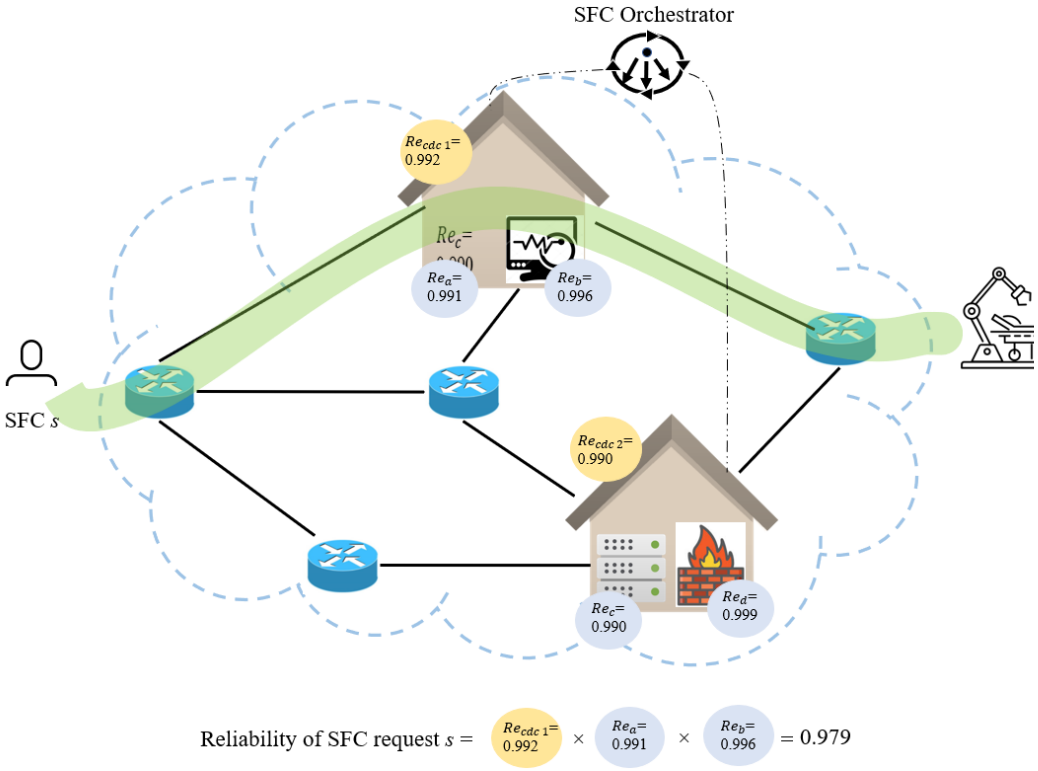

Figure 10. Reliability of an SFC request.

A. *Reliability Model*

In this study, both software and hardware failures are taken into consideration, which means both VNF and *CDC* node failures are taken into account. When a component in this model fails, it ceases to function and stops producing any output or results. Although it may crash or halt, it never delivers inaccurate or distorted data. In an NFV context, the availability of a component, such as *CDC* node or VNF, can be determined as the ratio of the mean time the component is up for service delivery to the sum of the mean time the component is up for service delivery and the mean time the component is down for repair. To this end, the availability of a component is defined as follows [51]:

$$\text{Availability}=\frac{MTBF}{MTBF+MTTR} \quad (1)$$

Where *MTBF* denotes the mean time between failures of the component and *MTTR* denotes the mean time to repair the failed component. The chance that a component (a VNF or *CDC* node) will be available for delivering services without failure for a certain amount of time is defined as reliability. Failures in either the software or the hardware might result in disruptions to the services that are provided.

In our formulations, the parameter Re_x denotes the software reliability of VNF *x*, while the parameter Re_{CDC} reflects the hardware reliability of *CDC* node *i* that hosts the VNF. As can be seen in Figure 10, the reliability of an SFC request *s* including VNF^a and VNF^b traversing the CDC^1 node is determined as Equation (2) [49]:

Reliability of $SFC^s \rightarrow Re^s = Re_{CDC\ 1} \times Re_a \times Re_b$ (2)

As a result, we reference to the reliability of SFC request *s* as Re^s, which is determined as shown in Equation (3), where $E^s_{i,x}$ indicates whether VNF *x* of SFC request *s* is served via *CDC* node *i* ($E^s_{i,x}$=1) or not ($E^s_{i,x}$=0).

$$Re^s = \prod_{x \in X} Re_x \,.\, Re^i_{CDC} \,.\, E^s_{i,x} \qquad \forall i \in N_{CDC}, \forall s \in S \tag{3}$$

Equation (3) is nonlinear; hence it cannot be used in our optimal optimization algorithm. Therefore, we present an approximation-based formulation for linear reliability. It is described in Chapter 4.

B. Substrate Network

We represent the substrate network as an undirected graph G= (*N*, *L*), where *N* denotes the set of physical nodes and *L* the set of physical links in our substrate network. Similar to phase one of our study, the nodes are divided into two types. The first category of nodes Core-Data-Center Nodes (N_{CDC}), hosts the different VNF types (e.g., a, b, c, d, etc.). Switching Nodes (N_S) are the

nodes that only forward traffic to succeeding nodes (N_{CDC}, $N_S \subseteq N$) and are incapable of hosting any VNF kinds. On the network depicted in Figure 9, *CDC* nodes are depicted as green squares, whereas switching nodes are depicted as red dots. The physical network and SFC requests are both considered as static inputs at this phase. As illustrated, we have setup two *CDC* nodes based on node degree to host VNF instances and six switching nodes to forward traffic to the subsequent nodes. We use *i* and *j* to represent two nodes in the substrate network, and (*i, j*) to represent the link between node *i* and *j*. In our investigation, we impose limitations on bandwidth use, memory utilization, and CPU utilization. The parameter $K_{(i,j)}^{bw}$ represents the total bandwidth capacity of link (*i*, *j*). The parameters K_i^{cpu} and K_i^{mem} denote the CPU and the memory capacities of node *i*, respectively.

C. Service Function Chain Requests

An SFC request *s* is a sequence of VNFs that must be concatenated in a certain order, see Figure 6 on Page 10. To define SFC requests, a set of input parameters, which we refer to as input matrices, is required. The parameters we utilize to signify an SFC request *s* are listed below. We employ ten parameters to indicate SFC request *s*, $\{Src^s,\ Des^s,\ A_x^s, O_x^s,\ P^s, \mathrm{r}^s,\ \psi_{bw}^s,\ \psi_{cpu}^s,\ \psi_{mem}^s,\ \psi_{td}^s\}$. The parameters Src^s and Des^s represent the source and destination of SFC request *s*, respectively. The parameter *x* represents the VNF type and *S* denotes the total number of SFC requests. The matrix $A_{S\times X}$ represents the needed VNFs for each SFC request *s*, where A_x^s equals 1, if the SFC request s requests the VNF *x* (0 otherwise). We take *A* to be a static input in our study. The integer matrix $O_{S\times X}$ specifies the ordering of the needed VNFs for SFC request *s*. The O_x^s denotes the x^{th} required VNF for SFC request *s*, VNFs with a lower index must provide service first. P^s denotes the priority of each SFC request *s*, high and low-priority SFC requests. The parameter r^s represents the

reliability requirement of SFC request *s*. The bandwidth, CPU, and memory requirements of SFC request *s* is represented as ψ_{bw}^{s} , ψ_{cpu}^{s} , ψ_{mem}^{s}, respectively. Finally, ψ_{td}^{s} represents the maximum tolerable end-to-end delay of SFC request *s*.

3.3 Dynamic Service Function Chaining

Given that in the third phase of our research we study dynamic service function chaining and present a heuristic way to handle *URLLC* in an NFV-enabled network, we chose a bigger network topology. We chose EliBackbone network topology consisting of 19 nodes and 28 links [60]. In a dynamic context, each SFC request has an arrival time and a departure time in order to use physical network resources. Similar to a real-world scenario in which SFC requests enter and exit the network at varied time intervals and use the network for a certain amount of time. When the allotted amount of time for an SFC request has passed, the underlying physical network resources become available for the subsequent SFC request to use.

A. Substrate Network

We present the *Substrate Network* as an undirected graph $G_s=(N_s,L_s)$, where N_s represents the set of physical nodes and L_s represents the set of physical links. We represent two substrate network nodes as *m* and *n,* and the link between nodes *m* and *n* as (*m*,*n*). The parameter $C_{(m,n)}^{BW}$ denotes the total bandwidth capacity of the link (*m*,*n*). The parameters C_m^{CPU} and C_m^{Mem}, respectively, represent the CPU and the memory capacity of node *m*. The set of physical nodes (N_s) consist of two types of nodes, Core-Data-Center Nodes (N_{CDC}) and Switching Nodes (N_{SW}), (N_{CDC}, $N_{SW} \subseteq N_s$). In our setting, only N_{CDC} can host the VNF types, whereas N_{SW} only transmits network traffic to the

following nodes. As shown in Figure 11, we consider three nodes to be *CDC* nodes (green squares) and sixteen nodes to be switching nodes (red dots).

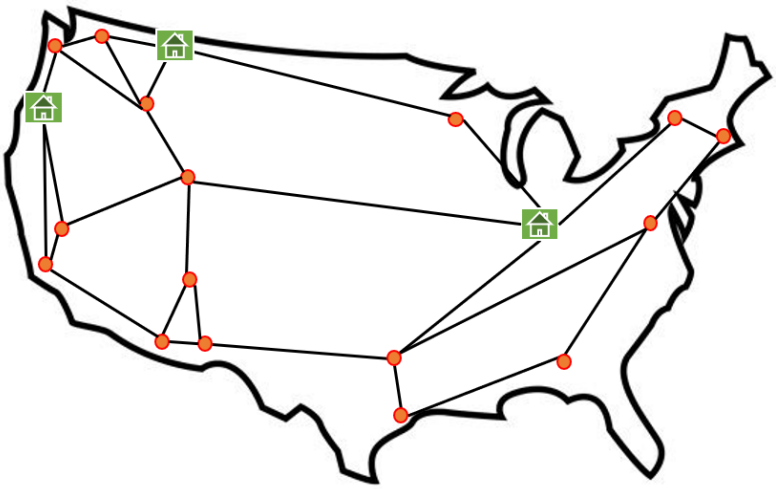

Figure 11. EliBackbone network topology [60].

B. Service Function Chain Request

An SFC request *f* consists of a series of VNFs that must be concatenated in a certain order. $\{Src^f, Des^f, T_a^f, T_d^f A_x^f, O_x^f, P^f, \mathrm{r}^f, \Omega_{BW}^f, \Omega_{CPU}^f, \Omega_{Mem}^f\}$ are eleven parameters used to identify SFC request *f*. The parameters Src^f and Des^f denotes, respectively, the source and destination of SFC request *f*. T_a^f and T_d^f are the arrival and departure times for SFC request *f*, respectively. The parameter *x* denotes the VNF type and *F* represent the total number of SFC requests. $A_{F\times X}$ denotes the required VNFs for each SFC request *f*, where A_x^f = 1, if the SFC request *f* requests the VNF *x* (0 otherwise). The integer matrix $O_{F\times X}$ specifies the ordering of the needed VNFs for SFC request *f*. The O_x^f denotes the x^{th} required VNF for SFC request *f*, VNFs with a lower index must provide service first. P^f denotes the priority of each SFC request *f*, high and low-priority SFC requests. The parameter r^f denotes the reliability requirement

of SFC request *f*. The bandwidth, CPU, and memory requirements of SFC request *f* are denoted as Ω_{BW}^{f} , Ω_{CPU}^{f} , Ω_{Mem}^{f}, respectively.

C. Reliability

Similar to the second phase of our study, failures of software and hardware components are considered in the third phase, which means VNFs and *CDC* nodes failures. This model stops working and stops delivering any output or results when a component fails. It may stop or crash, but it never provides distorted or false data. In an NFV setting, the availability of a component, such as *CDC* node or VNF, can be measured as the ratio of the mean time the component is up for service delivery to the sum of the mean time the component is up for service delivery and the mean time the component is down for repair. It is defined as follows [51]:

$$\text{Availability} = \frac{MTBF}{MTBF + MTTR} \tag{4}$$

Where *MTBF* is the mean time between failures of the component and *MTTR* denotes the mean time to repair the failed component. Reliability is the probability that a component (VNF or *CDC* node) will be available to supply services without interruption for a certain period of time. Interruptions in services are possible as a consequence of software and hardware failures. In this study, the parameter Re_x represents the software reliability of VNF *x*, while the parameter Re_{CDC} denotes the hardware reliability of *CDC* node *m* that hosts the VNF. As it can be seen in Figure 10, the reliability of an SFC request involving VNF^a and VNF^b traversing the CDC^1 node is determined by Equation (5) [61]:

$$\text{Reliability of } SFC^f \rightarrow Re^f = Re_{CDC\,1} \times Re_a \times Re_b \tag{5}$$

To this end, we refer to the reliability of SFC request f as Re^f, which is determined as shown in Equation (6), where $E^f_{m,x}$ indicates whether VNF x of SFC request f is served via *CDC* node m ($E^f_{m,x}$=1) or not ($E^f_{m,x}$=0).

$$Re^f = \prod_{x \in X} Re_x \,.\, Re^m_{CDC} \,.\, E^f_{m,x} \qquad \forall m \in N_{CDC}, \forall f \in F \tag{6}$$

After illustrating the system models and their underlying assumptions, the subsequent chapter formulates the SFC deployment problem as an ILP optimization model.

4 Integer-Linear-Programming Optimization Model

In this chapter, we formulate our proposed methodology in the form of an ILP optimization model. Using the ILP optimization model, we are able to obtain exact numerical solutions to the SFC deployment problem. The task of constructing all of the constraints and the objective function in the form of an ILP optimization model is a very difficult one, as we discussed in the second chapter. In order to address *URLLC* in an NFV-enabled network, we first created a comprehensive list of required constraints in order to obtain exact numerical results. This was done by doing an analysis of the studies that were relevant to the topic in Chapter 2. Following that, we took this set of identified constraints and converted them into linear mathematical formulations based on our system model and the problem statement, both of which are detailed in the upcoming subsections.

This chapter is constructed as follows: In the first section of this chapter, we will go through the mathematical formulations of the necessary constraints to address ultra-low latency communication. In accordance with this, we construct an ILP optimization model consisting of the required constraints along with an objective function. In the second section of this chapter, we investigate ultra-reliable low-latency communication and develop an ILP optimization model to address *URLLC* in a network that supports NFV. This model includes the essential constraints and the objective function.

4.1 Ultra-Low Latency Communication

It is of the utmost priority for suppliers of network services to fulfill the various requirements of network services. Applications that require ultra-low latency are highly sensitive and important and therefore require special

handling. As a result, it is essential to have an efficient SFC embedding algorithm. In this section of the chapter, we will discuss how to structure the SFC embedding problem as an ILP optimization model. This model will handle applications that require extremely low latency in a different manner than other applications.

As stated before, we consider two priorities for SFC requests: high-priority for *ultra-low latency applications* and low-priority for other network services. We use a *priority coefficient factor* (μ) to apply physical resource reservations (bandwidth, RAM memory, and CPU) for high-priority network services. We assume the initial *priority coefficient factor (μ)* is 0.9, which means a maximum of 90 percent of physical resources can be used by low-priority SFC requests and 10 percent of the physical resources are reserved for high-priority SFC requests. In our study, we investigate the change in *priority coefficient factor (μ)* in Chapter 6.

First, we will establish the appropriate bandwidth, CPU, and memory utilization restrictions for SFC requests based on the priority of each SFC request and the priority coefficient factor. These requirements are guaranteed by (7-12). Constraint (7) ensures that the total bandwidth utilization of SFC requests cannot exceed the total physical bandwidth capacity of the link between node *m* and node *n*. In this context, *F* is the total number of SFC requests in the network, which we take as a static input for our investigation. Constraint (8) forces that the total bandwidth utilization of SFC requests with low-priority cannot exceed 90 percent (given that the initial value of μ is 0.9) of the physical bandwidth capacity of the link between node *m* and node *n*. F' is the total number of SFC requests with low-priority. We use the binary variable $H_{(m,n)}^{f} \in \{0, 1\}$ to indicate whether the SFC request *f* traverses the link (*m*,*n*) or not. $H_{(m,n)}^{f}$ equals 1, if the SFC request *f* traverses the link (*m*,*n*), and 0 otherwise.

In Table 1, you will find a list of all symbols along with an explanation.

$$\sum_{f=1}^{F} \tau_{bw}^{f} \,.\, H_{(m,n)}^{f} \;\leq\; C_{(m,n)}^{bw}, \qquad \forall\, m,n \in\; N_P \tag{7}$$

$$\sum_{f\prime=1}^{F'} \tau_{bw}^{f\prime}.H_{(m,n)}^{f\prime} \leq C_{(m,n)}^{bw}.\mu \;, \quad \forall m,n \in N_P \tag{8}$$

Constraint (9) makes certain that the CPU consumption of SFC requests does not go over the total CPU capacity of *CDC* node *m*. Constraint (10) guarantees that the CPU utilization of SFC requests with low-priority cannot exceed 90 percent (given that the initial value of μ is 0.9) of the total CPU capacity of *CDC* node *m*.

$$\sum_{f=1}^{F} \tau_{cpu}^{f}.\; H_{(m,n)}^{f} \leq C_{m}^{cpu}, \qquad \forall m \in N_{CDC}, \forall n \in N_P \tag{9}$$

$$\sum_{f\prime=1}^{F'} \tau_{cpu}^{f\prime}.H_{(m,n)}^{f\prime} \leq C_{m}^{cpu}.\mu \;, \quad \forall m \in N_{CDC}, \forall n \in N_P \tag{10}$$

The same logic for CPU utilization is applied for memory utilization on *CDC* node *m.* Constraint (11) ensures that the memory utilization of SFC requests does not exceed the total memory capacity of *CDC* node *m*. Constraint (12) guarantees that the memory utilization of SFC requests with low-priority cannot exceed 90 percent (given that the initial value of μ is 0.9) of the memory capacity of *CDC* node *m*.

$$\sum_{f=1}^{F} \tau_{mem}^{f}.H_{(m,n)}^{f} \;\leq\; C_{m}^{mem}, \qquad \forall m \in N_{CDC}, \forall n \in N_P \tag{11}$$

Table 1. Symbols and variables used in the first phase of our study.

Symbols	Description
G_P	The physical network
N_P	The set of the physical nodes
L_P	The set of the physical links
N_S	The set of the switching nodes ($N_S \subseteq N_P$)
N_{CDC}	The set of the *CDC* nodes ($N_{CDC} \subseteq N_P$)
F	The total number of SFC requests (flows)
F´	The total number of low-priority SFC requests (flows)
X	The total number of VNF types (e.g., *a*, *b*, *c*, *d*, etc.)
P^f	The priority of SFC requests *f*
μ	The *priority coefficient factor* for physical resource reservation ($\mu = 0.9$ as the initial value)
$H^f_{(m,n)}$	A binary variable, whether flow *f* traverses the link (*m*,*n)* or not
$K^f_{m,x}$	A binary variable, whether flow *f* uses VNF type *x* which is placed at *CDC* node *m* or not
$C^{bw}_{(m,n)}$	The total bandwidth capacity of link (*m*,*n*)
C^{cpu}_m	The total CPU capacity of node *m*
C^{mem}_m	The total memory capacity of node *m*
Src^f	The source node of SFC request *f*
Des^f	The destination node of SFC request *f*
$R_{F \times X}$	The matrix of required VNFs by SFC request *f*
$W_{F \times X}$	The matrix of ordering of VNFs requested by SFC request *f*
τ^f_{bw}	The required bandwidth by SFC request *f*
τ^f_{cpu}	The required CPU by SFC request *f*
τ^f_{mem}	The required memory by SFC request *f*

τ_{td}^{f}	The maximum tolerated delay by SFC request *f*
$D_{(m,n)}$	The propagation delay on link (*m,n)*
$L_{\mathrm{Ncdc}\times \mathrm{X}}$	The matrix represents the VNF types placed on each *CDC* node
$T_{(Np\times Np\times F)}$	The matrix *T* represents the ordering-aware rerouting matrix
Z_{R^f}	Contains all the required VNFs with a higher order (i.e., lower index) than R^f

$$\sum_{f\prime=1}^{F'} \tau_{mem}^{f\prime}.H_{(m,n)}^{f\prime} \leq C_m^{mem}.\mu\ , \quad \forall m \in N_{CDC}, \forall n \in N_P \tag{12}$$

Constraint (13) ensures that the propagation delay of SFC request *f* cannot exceed the maximum tolerated end-to-end delay of SFC request *f*. We apply it to both low- and high-priority SFC requests to minimize any side effects on low-priority applications.

$$\sum_{m=1}^{Np}\sum_{n=1}^{Np} D_{(m,n)}.H_{(m,n)}^{f} \leq \tau_{td}^{f}, \quad \forall f \in F \tag{13}$$

In this formulation, $D_{(m,n)}$ is the propagation delay on the link between node *m* and node *n*. We ensure flow control by using Constraint (14). We make sure that the links on the deployment path of SFC request *f* are connected head-to-tail.

$$\sum_{n=1}^{Np} H_{(m,n)}^{f} - \sum_{n=1}^{Np} H_{(n,m)}^{f} = \begin{cases} 1 & m = Src^f \\ -1 & m = Des^f \\ 0 & m \neq Src^f, Des^f \end{cases}$$

$$\forall f \in F, \forall m \in N_P \tag{14}$$

To guarantee that each VNF type *x* required by SFC request *f* is supported by the server hosting it, we apply Constraint (15).

$$K^{f}_{m,x} \leq L_{m,x}, \qquad \forall f \in F, \forall m \in N_{CDC}, \forall x \in X \tag{15}$$

$K^{f}_{m,x}$ is a binary variable. $K^{f}_{m,x}$ equals 1, if the SFC request *f* uses VNF type *x*, which is placed on *CDC* node *m*, and 0 otherwise. The matrix $L_{(Ncdc \times X)}$ identifies the VNF types placed on each *CDC* node, which is given as an input. If VNF type *x* is placed on *CDC* node *m*, then $L_{m,x}$=1 (and 0 otherwise). We consider Constraint (16), in order to prevent a loop for SFC request *f*.

$$\sum_{n=1}^{Np} H^{f}_{(m,n)} \leq 1, \qquad \forall m \in N_p, \forall f \in F \tag{16}$$

Constraint (17) is considered to ensure that SFC request *f* crosses a valid VNF chain while traversing through the nodes.

$$\sum_{m=1}^{Ncdc} K^{f}_{m,x} = R^{f}_{x}, \qquad \forall x \in X, \forall f \in F \tag{17}$$

We define Constraint (18) to make sure that each VNF type *x* is used by at most one SFC request.

$$\sum_{m=1}^{Ncdc} K^{f}_{m,x} \leq 1, \qquad \forall x \in X, \forall f \in F \tag{18}$$

In the following, we consider ordering constraints to enforce an ordered sequence of VNFs to concatenate different VNFs of SFC requests. We introduce matrix T ($N_p \times N_p \times F$) as ordering-aware rerouting matrix. *T* includes the notion of ordering for the nodes and links appearing in the deployment path. The elements of T^f are integer values starting from 1, which means that node need to be pathed first and so on.

$$T^f = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \\ 0 & 2 & 0 & 0 \end{bmatrix}$$

The matrix T^f represents that the SFC request *f* leaves node *one* in the first step and goes to node *four* (since the fourth column is one), then in the second step, from node *four* goes to node *two* (since the second column in row *four* is two), and finally from node *two* reaches node *three*, which is the destination of the SFC request *f*. The value four in T^f means that SFC request *f* crossed four nodes to reach the destination. Indeed, $T^f_{(m,n)}$ specifies the number of previously crossed nodes. The values stored in the matrix T^f are integer and need to be equal or higher to the corresponding one stored in the rerouting matrix $H^f_{(m,n)}$. Therefore, we define Constraint (19).

$$T^f_{(m,n)} \geq H^f_{(m,n)}, \qquad \forall f \in F, \forall m, n \in N_p \tag{19}$$

We ensure that $T^f_{(m,n)}$ becomes zero, if $H^f_{(m,n)}$ is zero. Therefore, we introduce Constraint (20).

$$T^f_{(m,n)} = T^f_{(m,n)} . H^f_{(m,n)}, \qquad \forall f \in F, \forall m \in N_p - \{Des^f\}, \forall n \in N_p \tag{20}$$

As we can see, Constraint (20) is a nonlinear constraint. We need to rewrite it as a linear constraint for our optimization model. To this end, since each SFC request f traverses at most all the nodes, we can change Constraint (20) to Constraint (21).

$$T^f_{(m,n)} \leq N_p . H^f_{(m,n)}, \qquad \forall f \in F, \forall m \in N_p - \{Des^f\}, \forall n \in N_p \tag{21}$$

The elements of the ordering-aware rerouting matrix for the output links must be zero for the destination node. Therefore, we apply Constraint (22).

$$T^f_{(m,n)} = 0, \quad \forall f \in F, \forall m \in Des^f, \forall n \in N_p - \{Des^f\} \tag{22}$$

Except for the source and destination nodes, when SFC request f enters a node in its n^{th} step, it leaves that node in the $(n+1)^{th}$ step. Therefore, we introduce the Constraint (23).

$$\sum_{n=1}^{Np} T_{(m,n)}^{f} = \sum_{n=1}^{Np} T_{(n,m)}^{f} + \sum_{n=1}^{Np} H_{(n,m)}^{f} \quad \forall f \in F, \forall m \in N_p - \{Src^f, Des^f\} \quad (23)$$

We need to make sure that the SFC requests leave the source nodes. Therefore, we introduce the Constraint (24).

$$\sum_{n=1}^{Np} T_{(m,n)}^{f} = 1, \qquad \forall f \in F, \forall m \in Src^f \quad (24)$$

Finally, to force the sequence of VNF chaining, Constraint (25) is introduced. The purpose of this process is to determine if the VNFs with higher ordering, i.e., lower index in W^f, are delivered to the SFC request *f* on one of the crossed servers or not.

$$\sum_{m=1}^{Np} \sum_{n=1}^{Np} T_{(m,n)}^{f} \cdot K_{(m, W_{R^f}^{f})}^{f} \geq \sum_{m=1}^{Np} \sum_{n=1}^{Np} T_{(m,n)}^{f} \cdot K_{(m, W_{Z_{R^f}}^{f})}^{f}$$

$$\forall f \in F, \forall R^f \in \{1, \dots, len(W^f)\}$$

$$\forall Z_{R^f} \in \{1, \dots, R^f - 1\},\ T_{(m,n)}^{f} \in \mathbb{Z} \geq 0, \forall m, n \in N_p \quad (25)$$

In Constraint (25), Z_{Rf} stores all the required VNFs with a higher order, i.e., lower index, than R^f. For instance, if $W^f = [4\ \ 3\ \ 2\ \ 6]$ then $R^f \in \{1, 2, 3, 4\}$. If we assume $R^f = 4$, then Z_{Rf} is a member of {2, 3}. Using the same approach as [27], the ordering constraints of the VNFs belonging to a flow are expressed in a different form. Therefore, the Constraint (25) is replaced with the Constraint (26).

$$\left(1 - K^{f}_{(m\,,W^{f}_{Rf})}\right).(2N-1) + \sum_{n=1}^{Np} T^{f}_{(m,n)} \geq (K^{f}_{(I\,,W^{f}_{Z_{Rf}})}).(2N-1) + \sum_{n=1}^{Np} T^{f}_{(I,n)}$$

$$\forall f \in F, \forall R^{f} \in \{1, \dots, len(W^{f})\}, \forall Z_{R^{f}} \in \{1, \dots, R^{f}-1\}, \forall I, m \in N_{p} \quad (26)$$

In Constraint (26), if the *CDC* node *m* hosts the VNF W_{R^f}, then $K^{f}_{(m\,,W_{Rf})} = 1$. Therefore, the left-side of Constraint (26) considers the step of the *CDC* node *m* and it must be greater than the step of all *CDC* nodes (*I*) hosting a VNF with an index lower than the index of VNF W_{R^f} in W^f. By considering the Z_{R^f} as the index of any VNF in W^f with an index lower than VNF W_{R^f}, which means the flow *f* must pass VNF $W^{f}_{Z_{Rf}}$ before W^{f}_{Rf}. If the *CDC* node *m* (*I*) hosts the VNF $W^{f}_{Z_{Rf}}$, then $K^{f}_{(\mathrm{m}\,,W^{f}_{Z_{Rf}})} = 1$. Therefore, the right-side of Constraint (26) considers the step of the *CDC* node *m* (*I*) and it must be greater than the step of all *CDC* nodes *m* hosting a VNF with an index greater than the index of VNF $W^{f}_{Z_{Rf}}$ in W^f. If either $K^{f}_{m\,,W_{Rf}}$ or $K^{f}_{(I\,,W_{Zf})}$ are equal to zero, the value of $\sum_{\mathrm{n=1}}^{\mathrm{Np}} T^{f}_{(m,n)}$ is always lower than (2*N*-1), then the constraint is fulfilled. $\sum_{\mathrm{n=1}}^{\mathrm{Np}} T^{f}_{(m,n)}$ and $\sum_{\mathrm{m=1}}^{\mathrm{Np}} T^{f}_{(m,n)}$ are always lower than (2N-1), since in the worst case, the flow crosses all nodes, which means the value of $\sum_{\mathrm{n=1}}^{\mathrm{Np}} T^{f}_{(m,n)}$ is at most (N-1)+N. The destination has a flow to itself with a step of at most N+1. When both $K^{f}_{(m\,,W_{Rf})}$ and $K^{f}_{(I\,,W_{Zf})}$ are equal to one, the constraint is satisfied on the condition that the value of $\sum_{\mathrm{n=1}}^{\mathrm{Np}} T^{f}_{(m,n)}$ is greater than $\sum_{\mathrm{n=1}}^{\mathrm{Np}} T^{f}_{(I,n)}$. It means that a *CDC* node that delivers the lower index VNF is crossed before the *CDC* nodes that deliver higher index VNFs [27].

We discussed the mathematical formulation of the SFC deployment problem in the earlier formulations. These formulations handle the prioritized SFC requests in a different way than in related studies. As was mentioned earlier, by utilizing the *priority coefficient factor* (μ) to reserve physical network resources for embedding high-priority SFC requests and optimize their provisioning paths, and also by taking into consideration the maximum tolerable end-to-end delay of low-priority SFC requests to minimize side effects on low-priority SFC requests, we define the objective function as (27), which is to optimize the provisioning paths of SFC requests with respect to the end-to-end delay, subject to: (7)-(19), (21)-(24) and (26).

$$Minimize \sum_{m=1}^{Np} \sum_{n=1}^{Np} D_{(m,n)}.H_{(m,n)}^{f}, \qquad \forall f \in F \tag{27}$$

The objective function (27) aims to find the most optimal deployment paths for each SFC request with respect to the end-to-end delay. This methodology we have developed is referred to as the ***O**ptimal **A**pplication-Aware **S**FC* (*OAS*) embedding methodology. The objective is to reduce the number of intermediate nodes that a flow traverses from the source node to the destination node, while considering the overall delay experienced by the flow. *OAS* is an *NP-hard* problem because, as can be seen, it maps to the *weight constrained shortest path problem* (*WCSPP*), which is also an *NP-hard* problem [62, 63]. An *NP-Hard* problem is a type of complexity class in which obtaining the exact numerical solutions for large-scale network topologies is very time-consuming. In Chapter 5, we propose a heuristic algorithm (the *FAS* algorithm) to obtain near-optimal solutions close to the results produced by Equation (27) within an acceptable time frame for large network topologies.

4.2 Ultra-Reliable Low Latency Communication

As mentioned earlier, we address *URLLC* in an NFV-enabled network in the second phase of our study by incorporating a reliability viewpoint into our findings from the first phase of our study. Providing a guaranteed quality of service for *URLLA* is one of the main challenges for NSPs, and it has become more complicated by the limited physical network resources. Thus, in an NFV environment, having an efficient SFC embedding algorithm that supports *URLLA* is essential. We construct our ***O**ptimal **R**eliability- and **A**pplication-**A**ware **S**FC (ORAAS)* embedding algorithm as an ILP optimization model, as described in the following. In Table 2, we have provided a list of all symbols along with their respective definitions.

The first thing that has to be done is define the reliability constraints. In accordance with what was discussed in Chapter 3, the reliability of SFC requests is assessed using Equation (3). As a direct consequence of this, Constraint (28) ensures that the reliability of the deployment path of SFC request *s* is higher than the minimum needed reliability of SFC request *s*. As it is shown, Constraint (28) is a nonlinear constraint.

$$\prod_{x \in X} Re_x \,.\, Re^i_{CDC} \,.\, E^s_{i,x} \;\geq\; \mathrm{r}^s \,, \quad \forall i \in N_{CDC}, \forall s \in S \tag{28}$$

In order to use the Equation (28) in our optimization model, we need to linearize it. In order to do this, we apply the method of approximation outlined in [50] and replace Equation (28) with Equation (29).

$$1 - \sum_{x=1}^{X} \left(1 - Re_x \,.\, E^s_{i,x}\right) - \sum_{x=1}^{X} \left(1 - Re^i_{CDC} \,.\, E^s_{i,x}\right) \geq \mathrm{r}^s \,,$$

$$\forall i \in N_{CDC}, \forall s \in S \tag{29}$$

Table 2. Symbols and variables used in the second phase of our study

Symbols	Description
G	The physical network
N	The set of the physical nodes
L	The set of the physical links
N_S	The set of the switching nodes ($N_S \subseteq N$)
N_{CDC}	The set of the *CDC* nodes ($N_{CDC} \subseteq N$)
S	The total number of SFC requests (flows)
S´	The total number of low-priority SFC requests (flows)
X	The total number of VNF types (e.g., *a*, *b*, *c*, *d*, etc.)
P^s	The priority of SFC requests *s*
r^s	The required reliability by SFC request *s*
Re^s	The reliability of SFC request *s*
Re_a	The reliability of VNF *a*
Re_{CDC}	The reliability of *CDC* node
δ	The *priority coefficient factor* for physical resource reservation ($\mu = 0.9$ as the initial value)
$R^s_{(i,j)}$	A binary variable, whether flow *s* traverses the link (*i,j)* or not
$K^s_{i,x}$	A binary variable, whether flow *s* uses VNF type *x* which is placed at *CDC* node *i* or not
$K^{bw}_{(i,j)}$	The total bandwidth capacity of link (*i,j*)
K^{cpu}_i	The total CPU capacity of node *i*
K^{mem}_i	The total memory capacity of node *i*
Src^s	The source node of SFC request *s*
Des^s	The destination node of SFC request *s*
$A_{S \times X}$	The matrix of required VNFs by SFC request *s*
$O_{S \times X}$	The matrix of ordering of VNFs requested by SFC request *s*

ψ_{bw}^{s}	The required bandwidth by SFC request *s*
ψ_{cpu}^{s}	The required CPU by SFC request *s*
ψ_{mem}^{s}	The required memory by SFC request *s*
ψ_{td}^{s}	The maximum tolerable delay by SFC request *s*
$D_{(i,j)}$	The propagation delay on link (*i,j)*
$L_{\mathrm{Ncdc} \times \mathrm{X}}$	The matrix represents the VNF types placed on each *CDC* node
$Y_{(N \times N \times S)}$	The matrix *Y* represents the ordering-aware rerouting matrix
J_{A^s}	Contains all the required VNFs with a higher order (i.e. lower index) than A^s

In the next step, we will specify the limitations on the consumption of physical network resources (bandwidth, memory, and CPU) in proportion to the priority of each SFC request *s*. Constraint (30) ensures that the bandwidth utilization of all SFC requests on link (*i, j*) does not exceed the total bandwidth capacity of substrate link (*i, j*). The binary variable $R_{(i,j)}^{s}$ indicates the routing path of SFC request *s*, where $R_{(i,j)}^{s}$=1, if SFC request *s* traverses the link (*i, j*), and 0 otherwise. The parameter *S* indicates the total number of SFC requests. Constraint (31) ensures that the low-priority SFC requests cannot exceed (δ × bandwidth of substrate link *(i, j)*). The parameter δ is the adjustable *priority coefficient factor* for reserving physical network resources for high-priority SFC requests. We assume the initial *priority coefficient factor* (δ) as 0.9, which means that 10% of physical resources are reserved for embedding only high-priority SFC requests (*URLLA*). The parameter *S'* denotes the total number of low-priority SFC requests.

$$\sum_{s=1}^{S} \psi_{bw}^{s} \cdot R_{(i,j)}^{s} \leq K_{(i,j)}^{bw}, \qquad \forall\, i,j \in N \tag{30}$$

$$\sum_{s\prime=1}^{S\prime} \psi_{bw}^{s\prime}.\, R_{(i,j)}^{s\prime} \leq K_{(i,j)}^{bw}.\,\delta\,, \qquad \forall i,j \in N \tag{31}$$

Constraint (32) guarantees that the CPU usage of SFC requests on node *i* does not exceed the overall CPU capacity of *CDC* node *i*. Constraint (33) prevents low-priority SFC requests from exceeding (δ × CPU capacity of the *CDC* node *i*).

$$\sum_{s=1}^{S} \psi_{cpu}^{s}.\; R_{(i,j)}^{s} \leq K_i^{cpu}, \qquad \forall i \in N_{CDC}, \forall j \in N \tag{32}$$

$$\sum_{s\prime=1}^{S\prime} \psi_{cpu}^{s\prime}.\; R_{(i,j)}^{s\prime} \leq K_i^{cpu}.\,\delta\,, \qquad \forall i \in N_{CDC}, \forall j \in N \tag{33}$$

The same logic as for CPU utilization of *CDC* node *i* is applied for memory utilization of *CDC* node *i* via Constraint (34) and (35).

$$\sum_{s=1}^{S} \psi_{mem}^{s}.\; R_{(i,j)}^{s} \leq K_i^{mem}, \qquad \forall i \in N_{CDC}, \forall j \in N \tag{34}$$

$$\sum_{s\prime=1}^{S\prime} \psi_{mem}^{s\prime}.\; R_{(i,j)}^{s\prime} \leq K_i^{mem}.\,\delta\,, \quad \forall i \in N_{CDC}, \forall j \in N \tag{35}$$

Constraint (36) is used to guarantee the maximum tolerable end-to-end delay for SFC request *s*.

$$\sum_{i=1}^{N} \sum_{j=1}^{N} D_{(i,j)}.\, R_{(i,j)}^{s} \leq \psi_{td}^{s}\,, \qquad \forall s \in S \tag{36}$$

The parameter $D_{(i,j)}$ denotes the propagation delay on link *(i, j)*. The flow control is guaranteed via Constraint (37), which ensures that the links on the deployment path of SFC request *s* are connected head-to-tail.

$$\sum_{j=1}^{N} R^{S}_{(i,j)} - \sum_{j=1}^{N} R^{S}_{(j,i)} = \begin{cases} 1 & i = Src^{s} \\ -1 & i = Des^{s} \\ 0 & i \neq Src^{s}, Des^{s} \end{cases}$$

$$\forall i \in N, \forall s \in S \tag{37}$$

We use Constraint (38) to prevent a loop for SFC request *s*.

$$\sum_{j=1}^{N} R^{s}_{(i,j)} \leq 1, \qquad \forall i \in N, \forall s \in S \tag{38}$$

To ensure that SFC request *s* crosses a valid VNF chain while traversing through the nodes, we apply Constraint (39).

$$\sum_{i=1}^{Ncdc} E^{s}_{i,x} = A^{s}_{x}, \qquad \forall x \in X, \forall s \in S \tag{39}$$

Each VNF *x* of SFC request *s* needs to be supported by the server hosting it. Therefore, Constraint (40) is needed.

$$E^{s}_{i,x} \leq L_{i,x}, \qquad \forall i \in N_{CDC}, \forall x \in X, \forall s \in S \tag{40}$$

To identify the VNF types placed on *CDC* node *i*, we use the matrix $L_{i,x}$ (N × X). To make sure that each VNF is used by at most one SFC request, the Constraint (41) is proposed.

$$\sum_{i=1}^{Ncdc} E^{s}_{i,x} \leq 1, \qquad \forall x \in X, \forall s \in S \tag{41}$$

With the following constraints, we apply the VNF ordering in the SFC deployment path. As stated before, the VNF instances of SFC request *s* need to be concatenated in a predefined order. Therefore, we apply the following ordering constraints. First, we define a matrix Y^{s}(N × N) to store the deployment path of SFC request *s*. The stored values in Y^{s} are integer; therefore, its elements

need to be equal or higher to the corresponding rerouting matrix $R^s_{(i,j)}$. Thus, we introduce Constraint (42).

$$Y^s_{(i,j)} \geq R^s_{(i,j)}, \qquad \forall s \in S, \forall i,j \in N \tag{42}$$

If $R^s_{(i,j)}$ becomes zero, then we need to be sure that $Y^s_{(i,j)}$ becomes zero as well. To apply this, we utilize Constraint (43).

$$Y^s_{(i,j)} = Y^s_{(i,j)}.\ R^s_{(i,j)}, \quad \forall j \in N, \forall s \in S, \forall i \in N - \{Des^s\} \tag{43}$$

Since Constraint (43) is nonlinear, we need to make it linear. We know that each SFC request *s* can traverse at most all the nodes in our substrate network. Therefore, we can apply Constraint (44) instead of Constraint (43), which is linear.

$$Y^s_{(i,j)} \leq N\ .R^s_{(i,j)}, \qquad \forall i \in N - \{Des^s\}, \forall j \in N, \forall s \in S \tag{44}$$

The output links of the rerouting matrix must be zero for the destination node. To apply this, we use Constraint (45).

$$Y^s_{(i,j)} = 0, \qquad \forall i \in Des^s, \forall j \in N - \{Des^s\}, \forall s \in S \tag{45}$$

Next, we need to make sure that when an SFC request enters a node in its n^{th} step, it leaves that node in its $(n+1)^{th}$ step. Therefore, we use Constraint (46).

$$\sum_{j=1}^{N} Y^f_{(i,j)} = \sum_{j=1}^{N} Y^s_{(j,i)} + \sum_{j=1}^{N} R^s_{(j,i)} \qquad \forall i \in N - \{Src^s, Des^s\}, \forall s \in S \tag{46}$$

We apply Constraint (47) to make sure that the SFC requests leave the source nodes.

$$\sum_{j=1}^{N} Y^s_{(i,j)} = 1, \qquad \forall i \in Src^s, \forall s \in S \tag{47}$$

As the last ordering constraint in our optimization model to check if VNFs with higher ordering are delivered to the SFC request *s* in one of the crossed serves or not, we apply Constraint (48).

$$\sum_{i=1}^{N}\sum_{j=1}^{N} Y^{s}_{(i,j)} \cdot E^{s}_{(i,\, o^{s}_{A^s})} \geq \sum_{i=1}^{N}\sum_{j=1}^{N} Y^{s}_{(i,j)} \cdot E^{s}_{(i,\, o^{s}_{J_{A^s}})}$$

$$\forall s \in S, \forall A^s \in \{1, \dots, len(O^s)\}, \forall J_{A^s} \in \{1, \dots, A^s - 1\},$$

$$Y^{s}_{(i,j)} \in \mathbb{Z} \geq 0, \forall i, j \in N \qquad (48)$$

The matrix J_{A^s} stores all the required VNFs with a higher order than A^s, i.e., if $O^s = [4 \;\; 3 \;\; 2 \;\; 6\,]$ then $A^s \in \{1, 2, 3, 4\}$. If we assume $A^s = 4$, then J_{A^s} is a member of $\{2, 3\}$. Using the same approach as in [27], the ordering constraints of the VNFs belonging to a flow are expressed in a different form. Therefore, the Constraint (48) is replaced with the Constraint (49).

$$\left(1 - E^{s}_{(i\,,o^{s}_{A^s})}\right).(2N-1) + \sum_{i=1}^{N} Y^{s}_{(i,j)} \geq (E^{s}_{(I\,,o^{s}_{J_{A^s}})}).(2N-1) + \sum_{j=1}^{N} Y^{s}_{(I,j)}$$

$$\forall s \in S, \forall A^s \in \{1, \dots, len(O^s)\}, \forall J_{A^s} \in \{1, \dots, A^s - 1\}, \forall I, i \in N \qquad (49)$$

In Constraint (49), if the *CDC* node *i* hosts the VNF O_{A^s}, then $E^{s}_{(i\,,O^{s}_{A^s})} = 1$. Therefore, the left-side of Constraint (49) considers the step of the *CDC* node *i* and it must be greater than the step of all *CDC* nodes (I) hosting a VNF with an index lower than the index of VNF O_{A^s} in O^s. By considering the J_{A^s} as the index of any VNF in O^s with an index lower than VNF O_{A^s}, which means the flow *s* must pass VNF $O^{s}_{J_{A^s}}$ before $O^{s}_{A^s}$. If the *CDC* node *i* (I) hosts the VNF $O^{s}_{J_{A^s}}$, then $E^{s}_{(I\,,O^{s}_{J_{A^s}})} = 1$. Therefore, the right-side of Constraint (49) considers the step of the *CDC* node *i* (I) and it must be greater than the step of all *CDC* nodes *i* hosting a VNF with an index greater than the index of VNF $O^{s}_{J_{A^s}}$ in O^s.

If either $E^{s}_{i,O_{A^s}}$ or $E^{s}_{(I,O_{J^s})}$ are equal to zero, the value of $\sum_{j=1}^{N} Y^{s}_{(i,j)}$ is always lower than (2*N*-1), then the constraint is fulfilled. $\sum_{j=1}^{N} Y^{s}_{(i,j)}$ and $\sum_{i=1}^{N} Y^{s}_{(i,j)}$ are always lower than (2N-1), since in the worst case, the flow crosses all nodes, which means the value of $\sum_{j=1}^{N} Y^{s}_{(i,j)}$ is at most (N-1)+N. The destination has a flow to itself with a step of at most N+1. When both $E^{s}_{(i,O_{A^s})}$ and $E^{s}_{(I,O_{J^s})}$ are equal to one, the constraint is satisfied on the condition that the value of $\sum_{j=1}^{N} Y^{s}_{(i,j)}$ is greater than $\sum_{j=1}^{N} Y^{s}_{(I,j)}$. It means, a *CDC* node that delivers the lower index VNF is crossed before the *CDC* nodes that deliver higher index VNFs [27].

After specifying all the necessary constraints for our optimization model, we must define the objective function. Using the adjustable *priority coefficient factor* (δ), we reserve a certain amount of physical network resources exclusively for embedding *URLLA*. To minimize negative effects on low-priority SFC requests, we consider constraints on the maximum tolerable end-to-end delay and reliability for low-priority SFC requests. To this end, we define the objective function as (50), optimizing SFC request deployment paths in respect of end-to-end delay and subject to (29)-(42), (44-47), and (49).

$$Minimize \sum_{i=1}^{N} \sum_{j=1}^{N} R^{s}_{(i,j)} \cdot D_{(i,j)} \qquad \forall s \in S \tag{50}$$

Using Equation (50), we obtain the optimal results, which we refer to as the ***O**ptimal **R**eliability- and **A**pplication-**A**ware **S**FC* (*ORAAS*) embedding. As stated before, given that Equation (50) is an *NP-Hard* problem and getting exact numerical solutions is very time consuming, we provide a heuristic algorithm (*NORAAS* algorithm) in Chapter 5 in order to obtain near-optimal results with a minimal execution time and optimality gap.

5 Heuristic SFC Embedding Algorithms

Obtaining exact numerical solutions of the SFC deployment problem by utilizing an ILP optimization model is an *NP-hard* problem, which was proofed in the previous chapter. The goal of the proposed heuristic approaches is to achieve near-optimal results with minimum execution time to solve the scalability problem. When applied to real-world network topologies, the SFC embedding technique must be able to get near-optimal results in an acceptable amount of time. To this end, achieving near-optimal results with a small optimality gap and the shortest possible execution time plays an important role.

In this chapter, we provide a set of heuristic algorithms and relaxed versions for achieving near-optimal outcomes with minimal execution time and an optimality gap, making them usable in real-world use cases. These algorithms are designed to generate results as efficiently as possible. In Section 5.1, we will discuss our ***F**ast **A**pplication-Aware **S**ervice Function Chaining* (*FAS*) algorithm to address *Ultra-Low Latency Communication* and obtain near-optimal outcomes to the *OAS* approach that was discussed in Chapter 4. In Section 5.2, we will describe our heuristic approach to address *Ultra-Reliable Low-Latency Communication*, named the ***N**ear-**O**ptimal **R**eliable **A**pplication **A**ware **S**ervice Function Chaining* (*NORAAS*) algorithm. Our goal is to obtain results that are close to the optimal level of the *ORAAS* algorithm that was presented in Chapter 4. As mentioned before, in the first two phases of our investigation, our focus is on static SFC requests similar to [27, 28]. The SFC requests are considered static inputs, and the lifetime of SFC requests is not taken into account. In the final stage of our investigation, we look at a dynamic service function chaining scenario with flow arrival and departure times. In this scenario, each SFC request has a lifetime that specifies the amount of time that

it needs to use physical network resources in order to receive the service that it requests. Following the expiration of this lifetime, the physical network resources can be made available for subsequent SFC requests. Since the objective of our study is to address *Ultra-Reliable Low Latency Communication*, in Section 5.3, we offer a set of heuristic algorithms to address *URLLC* in a dynamic scenario similar to [29, 30]. In Section 5.4, we describe the ***N**earest-**S**ervice-Function-**F**irst (**NSF**)* algorithm proposed by Tajiki *et al*. [27]. It is used as a comparison algorithm. In Section 5.5, we present the well-known *Greedy* algorithm, which is also employed as a comparison algorithm as in [48, 45, 64, 65, 46, 36].

5.1 Fast Application-Aware SFC (FAS) Algorithm

In the following, we illustrate our proposed heuristic algorithm, named the *FAS* algorithm, in order to get results that are close to optimal using the optimal optimization model (*OAS* algorithm). Similar to the *OAS* approach, we classify SFC requests into those with a high-priority and those with a low-priority. Then, utilizing the *priority coefficient factor*, we set aside a certain portion of the available physical network resources (bandwidth, RAM memory, and CPU) for high-priority SFC requests. That means low-priority SFC requests can use a maximum of 90 percent of physical resources (since (μ × physical resources) and the initial value of μ is 0.9), and we reserve ((1- μ)× physical resources) of the total physical resources for high-priority SFC requests to optimize their deployment paths and hence provide a guaranteed QoS. In the sixth chapter, we investigate the impact of changing the *priority coefficient factor* (μ). The pseudocode of the *FAS* algorithm is presented in *Algorithm 1*.

Algorithm 1. ***FAS Algorithm***

Input: $G_p = (N_P, L_P)$ ← Physical Network;

$Src^f, Des^f, W_{(F \times X)}$

Output: SP^f ← Selected Path for SFC request *f*;

1: **for** each SFC request *f* **do**
2: SP^f = empty;
3: *CN = Src (f)* ← Set source of *f* as ***C**urrent **N**ode* (*CN*);
4: Free Resources = *Calculate_Free_Resources* (*Flow f*)
(Bandwidth, memory, and CPU);
5: Prune (N_{CDC}, L_P) ←Pruning the *CDC* nodes and the links,
which cannot be used to serve SFC request *f*;
6: **for** each VNF *x* in W_x^f **do**
7: Find Nearest *CDC* Providing *x* (CN, *x*) ← *Dijkstra*;
8: CN = next *CDC;*
9: Update *Used Resources*;
10: Update *SP;*
11: **end for**
12: Find shortest path from *CDC* to the *Des (flow)* ←*Dijkstra*;
13: Update *Used Resources*;
14: Update *SP;*
15: **end for**
16: **return** *SP;*

To execute *Algorithm 1*, the following list of input parameters is required. We require the physical network graph, including its physical network resources. We require the information concerning SFC requests, which includes the source node and the destination node of each SFC request as well as the priority of the request for use in *Algorithms 2* and *3*. The optimal *selected path* for each SFC request is the output that is intended to be produced. The first SFC request that is received by the system serves as our starting point. It is necessary to start with the source node of each SFC request and work our way to the destination of each SFC request while adhering to the constraints that are demanded.

In *Algorithm 1*, line 1 does an iteration loop for each SFC request (*F*) in order to identify the optimal deployment path for SFC requests. In line 2, we create a ***S**elected **P**ath* (*SP*) list in which we will record the optimal deployment paths that have been obtained for each SFC request *f*. The source of the SFC request *f* is specified to be the current node in line 3, which allows the algorithm to begin from this node. In line 4, we compute the free physical resources (bandwidth, memory, and CPU) with respect to the priority of each SFC request *f*, which is detailed in further detail in *Algorithms 2* and *3*. The purpose of line 5 is to remove the *CDC* nodes and physical links that are unable to be utilized in order to fulfill the SFC request *f*. Lines 6-8 use the *Dijkstra* method to locate the *CDC* nodes that are closest to the SFC request *f* that can supply the necessary VNFs. These nodes are located in accordance with the values that are recorded in the VNF ordering matrix *W*. After determining the routes with the shortest distances to the *CDC* nodes that supply the necessary VNFs for the SFC request *f*, we will update both the *Used Resources* (line 9) and the *Selected Path* (line 10). Following that, we identify the shortest path between the last *CDC* node that is giving the last required VNF of SFC request *f* and the destination node of SFC request *f* in line 12, and then we update the *Used Resources* in line 13 and the

Selected Path in line 14, respectively. Finally, in line 16, the algorithm provides a report of the most efficient deployment routes for the SFC request.

Algorithm 2 calculates the free physical resources with respect to the priority of each SFC request. In line 2, we reduce the *Used Resources* (calculated in *Algorithm 3*) from the 90 percent (since the initial value of μ is 0.9) of the total network capacity (bandwidth, memory, and CPU). In lines 3-4, if the SFC request *f* has high-priority, then it can use the reserved 10 percent of the network capacity. Line 6 makes the *Free Resources* equal zero if low-priority SFC requests require more than 90 percent of physical resources. Line 8 returns the *Free Resources*.

Algorithm 2. ***Calculate_Free_Resources***

Input: *Flow f*, P^f

Output: *Free Resources (bandwidth, memory, and CPU);*

1: **Calculate_Free_Resources (*Flow f*)**
2: Free Resources= (μ × Network Capacity) – *Used Resources*;
3: **if** *Flow f* has high-priority **then**
4: Free Resources= Free Resources + ((1- μ) × Network Capacity);
5: **end if**
6: i**f** *Free Resource* < 0 **then** *Free Resources*=0; ← Since (1- μ)% is reserved for high-priority, the *Free Resources* value for low-priority can become negative.
7: **end if**
8: **return** *Free Resources*;

Algorithm 3 updates *Used Resources* of SFC request *f* with respect to its priority. First, it checks the *Selected Path* of SFC request *f* in lines 1-2. In line 3, if the SFC request *f* has high-priority, then the algorithm uses the reserved 10 percent of network capacity for path deployment. If the reserved 10 percent of the network capacity is not enough for path deployment, then it uses the rest of the 90 percent of the network capacity (line 6). In line 9, if the SFC request *f* has low-priority, the algorithm reduces the required physical resource only from the 90 percent of the total network capacity. Line 12 returns the *Used Resources*

Algorithm 3. ***Update_Used_Resources***

Input:*Route, Flow f,* P^f

Output: *Used Resources (bandwidth, memory, and CPU);*

1: **Used Reources (*Route, Flow f*)**
2: **for** each (*node m* → *node n)* in *Route* **do**
3: **if** *Flow f* has high-priority **then**
4: reduce *Required Resources* from ((1- μ) × Network Capacity);
5: **if** ((1- μ) × Network Capacity) < *Required Resources*
6: **then** reduce the remaining from (μ × Network Capacity);
7: **end if**
8: **else**
9: reduce the *Required Resources* from (μ × Network Capacity);
10: **end if**
11: **end for**
12: **return** *Used Resources;*

for deployment of SFC request *f*. Using *Algorithms 1-3*, we obtain near-optimal results of our proposed ***O**ptimal **A**pplication-Aware **S**FC* (*OAS*) embedding algorithm with a minimum execution time.

5.2 Near-Optimal Reliability- and Application-Aware SFC (NORAAS) Algorithm

In this section, we illustrate our proposed heuristic algorithm to reduce the execution time of the optimal optimization model using Equation (50) and obtain near-optimal results for large-scale network topologies. Our heuristic algorithm handles high-priority SFC requests identically to the *ORAAS* algorithm, which means that we reserve a certain amount of physical network resources exclusively for embedding high-priority SFC requests to optimize their latency and reliability. We refer to our heuristic algorithm as the ***N**ear-**O**ptimal **R**eliability- and **A**pplication-**A**ware **S**FC* (*NORAAS*) embedding algorithm, and its pseudocode is presented in *Algorithm 4*.

Algorithm 4 returns the shortest route for deploying SFC request *s* in terms of end-to-end delay with respect to reliability. To do this, we need the following parameters: source node, destination node, VNF ordering matrix, and reliability requirement of SFC request *s*. It begins with the source node of each SFC request *s* (line 3), calculates the available physical network resources via *Algorithm 5* (line 4), then prunes the physical links and nodes in accordance with the available physical network resources (line 5*).* For each VNF in the ordering matrix of SFC request *s* (line 6), the *Dijkstra Algorithm* will be used to find the nearest *CDC* nodes that support the required VNFs (line 7). The reliability requirement of the *selected route* will be checked in line 8. Then it keeps a record of the physical resources that have been used (line 10). After locating the required VNFs for SFC request *s*, it determines the shortest route

Algorithm 4. ***NORAAS Algorithm***

Input: $G = (N, L)$, Src^s , Des^s, O_x^s, r^s;

Output: $Route^s$ ← Selected route for SFC request *s*;

1: **for** each SFC request *s* **do**
2: $Route^s$= empty;
3: *CN = Src (s)* ← Set source of *s* as ***C**urrent-**N**ode* (*CN*);
4: Calculate_Free_Resources (Bandwidth, memory, CPU)
← Via *Algorithm 5*;
5: Prune (N_{CDC}, *L*) ←Pruning the *CDC* nodes and the links,
which cannot be used to serve SFC request *s*;
6: **for** each VNF *x* in O_x^s **do**
7: Find nearest *CDC* providing *x* (CN, *x*) ← *Dijkstra*;
8: **if** Reliability [**Selected-Route]** ≥ r^s **then**
9: CN = next *CDC;*
10: Update *Used Resources*;
11: Update *Route;*
12 **end if**
13: **end for**
14: Find shortest path from *CDC* to the *Des (flow)* ←*Dijkstra*;
15: Update *Used Resources*;
16: Update *Route;*
17: **end for**
18: **return** *Route;*

between the latest *CDC* node and the SFC request's destination node (line 14). The *Used Resources* and *Selected Route* in lines 15 and 16, respectively, are then updated. The algorithm finally outputs the shortest route in line 18.

Algorithm 5 calculates the available physical network resources for mapping SFC requests based on their priority. It subtracts the *Used Resources* (calculated via *Algorithm 6*) from 90% of the total network capacity (because the initial value of δ is 0.9) in line 2. If the SFC request has a high-priority, it can utilize the reserved 10% of network capacity in lines 3-4. If low-priority SFC requests require more than 90% of physical resources, line 6 sets the *Free Resources* to zero. Line 8 returns the *Free Resources.*

Algorithm 5. ***Calculate_Free_Resources***

Input: *Flow s, P^s;*

Output: *Free Resources (Bandwidth, memory, CPU);*

1: **Calculate_Free_Resources (*Flow s*)**
2: Free Resources= (δ × Network Capacity) – *Used Resources*;
3: **if** *Flow s* has high-priority **then**
4: Free Resources= Free Resources + ((1- δ) × Network Capacity);
5: **end if**
6: i**f** *Free Resources* < 0 **then** *Free Resources*=0; ← Since (1- δ)% is reserved for high-priority, the *Free Resources* value for low-priority can become negative.
7: **end if**
8: **return** *Free Resources*;

Algorithm 6 updates the *Used Resources* for SFC request *s* in accordance with its priority. To begin, it verifies the *Selected Path* of SFC request *s* specified in lines 1-2. Then, if the SFC request *s* has a high-priority, the algorithm uses the reserved 10% of the total network capacity first (line 4) and, if necessary, the rest 90% of network capacity (lines 5-6). If the SFC request *s* is not a high-priority SFC request, it can use only 90% of network capacity (line 9). Line 12 returns the *Used Resources* for SFC deployment.

Algorithm 6. ***Update_Used_Resources***

Input*: Route, Flow s,* P^s*;*

Output: *Used Resources (Bandwidth, memory, CPU);*

1: ***Used Reources (Route, Flow s)***
2: **for** each *(node i → node j)* in *Route* ***do***
3: **if** *Flow s* has *high-priority* **then**
4: reduce *Required Resources* from *((1- δ) × Network Capacity);*
5: **if** *((1- δ) × Network Capacity) < Required Resources*
6: **then** reduce the remaining from *(δ × Network Capacity);*
7: **end if**
8: **else**
9: reduce the *Required Resources* from *(δ × Network Capacity);*
10: **end if**
11: **end for**
12*:* **return** *Used Resources;*

5.3 Dynamic Application Aware SFC (DAAS) Algorithm

As stated earlier, in the first two developed SFC embedding algorithms (*FAS* and *NORAAS*), we assumed that the SFC requests were static input as in [27, 28] and did not take into account the lifetime of each SFC request. In this subsection, we will add dynamicity to our SFC embedding algorithm. We'll look at dynamic SFC requests that have flow arrival and departure times similar to [29, 30]. Each SFC request has a lifetime to use physical network resources, and after the expiration of this lifetime, the physical network resources can be released for following SFC requests. In this part, we discuss our dynamic SFC embedding technique to address *ultra-reliable low latency communication*. This follows our work with a master student [66], in which we examined ultra-low latency communication in an NFV- enabled network in a dynamic environment. We construct our proposed ***D**ynamic **A**pplication-**A**ware **S**FC* (*DAAS*) embedding algorithm as following. All the symbols and variables of the *DAAS* algorithm are summarized in Table 3.

First, we establish the limitations on the utilization of physical network resources (bandwidth, memory, and CPU) in accordance with the priority of each SFC request *f*. Using *flow prioritization* and a configurable *priority coefficient factor*, we are able to reserve a portion of physical network resources for high-priority SFC requests or *URLLA* as follows. Constraint (51) guarantees that the bandwidth utilization of all SFC requests on link (*m*,*n*) does not exceed the total bandwidth capacity of substrate link (*m*,*n*). The binary variable $R^{f}_{(m,n)}$ indicates the routing path of SFC request *f*, where $R^{f}_{(m,n)}$=1, if SFC request *f* traverses the link (*m*,*n*), and 0 otherwise. The parameter *F* indicates the total number of SFC requests. Constraint (52) ensures that the low-priority SFC requests cannot exceed (μ × bandwidth of substrate link (*m*,*n*)). The parameter μ is the adjustable *priority coefficient factor* for reserving physical network resources for high-priority SFC requests. We assume the initial *priority*

coefficient factor (μ) to be 0.9, which means that 10 percent of physical resources are reserved for embedding only high-priority SFC requests (*URLLA*). The parameter *F'* denotes the total number of low-priority SFC requests.

$$\sum_{f=1}^{F} \Omega_{BW}^{f} . R_{(m,n)}^{f} \leq C_{(m,n)}^{BW}, \qquad \forall m,n \in N_s \tag{51}$$

$$\sum_{f\prime=1}^{F\prime} \Omega_{BW}^{f\prime} . R_{(m,n)}^{f\prime} \leq C_{(m,n)}^{BW} . \mu, \qquad \forall m,n \in N_s \tag{52}$$

Constraint (53) ensures that the CPU use of SFC requests on node *m* does not exceed the node's total CPU capability. Constraint (54) prohibits low-priority SFC requests from exceeding (μ × CPU capacity of the *CDC* node *m*). Therefore, we reserve ((1-μ) × CPU capacity) only for high-priority SFC requests.

$$\sum_{f=1}^{F} \Omega_{CPU}^{f} . R_{(m,n)}^{f} \leq C_{m}^{CPU}, \qquad \forall m \in N_{CDC}, \forall n \in N_s \tag{53}$$

$$\sum_{f\prime=1}^{F\prime} \Omega_{CPU}^{f\prime} . R_{(m,n)}^{f\prime} \leq C_{m}^{CPU} . \mu, \qquad \forall m \in N_{CDC}, \forall n \in N_s \tag{54}$$

Using Constraints (55) and (56), the same reasoning as for CPU utilization of *CDC* node *m* is applied to memory consumption of *CDC* node *m*. Constraint (55) ensures that the memory use of SFC requests on node *m* does not exceed the node's total memory capability. Constraint (56) prohibits low-priority SFC requests from exceeding (μ × memory capacity of the *CDC* node *m*). Therefore, we reserve ((1-μ) × memory capacity) only for high-priority SFC requests.

Table 3. Symbols and variables used in the third phase of our study

Symbols	Description
G_s	The substrate network
N_s	The set of the physical nodes
L_s	The set of the physical links
N_{SW}	The set of the switching nodes ($N_{Sw} \subseteq N_s$)
N_{CDC}	The set of the *CDC* nodes ($N_{CDC} \subseteq N_s$)
F	The total number of SFC requests (flows)
F´	The total number of low-priority SFC requests (flows)
X	The total number of VNF types (e.g., a, b, c, d, etc.)
P^f	The priority of SFC requests *f*
r^f	The required reliability by SFC request *f*
Re^f	The reliability of SFC request *f*
Re_a	The reliability of VNF *a*
Re_{CDC}	The reliability of *CDC* node
μ	The priority coefficient factor for physical resource reservation ($\mu = 0.9$ as the initial value)
$R^f_{(m,n)}$	A binary variable, whether flow *f* traverses the link (*m*,*n*) or not
$E^f_{m,x}$	A binary variable, whether flow *f* uses VNF type *x* which is placed at *CDC* node *m* or not
$C^{BW}_{(m,n)}$	The total bandwidth capacity of link (*m*,*n*)
C^{CPU}_m	The total CPU capacity of node *m*
C^{Mem}_m	The total memory capacity of node *m*
Src^f	The source node of SFC request *f*
Des^f	The destination node of SFC request *f*
T^f_a	The arrival time of SFC request *f*

T_d^f	The departure time of SFC request *f*
$A_{F\times X}$	The matrix of required VNFs by SFC request *f*
$O_{F\times X}$	The matrix of ordering of VNFs requested by SFC request *f*
Ω_{BW}^f	The required bandwidth by SFC request *f*
Ω_{CPU}^f	The required CPU by SFC request *f*
Ω_{Mem}^f	The required memory by SFC request *f*
Ω_{td}^f	The maximum tolerable delay by SFC request *f*
$Dl_{(m,n)}$	The propagation delay on link *(m,n)*

$$\sum_{f=1}^{F} \Omega_{Mem}^{f}.R_{(m,n)}^{f} \le C_m^{Mem}, \qquad \forall m \in N_{CDC}, \forall n \in N_s \tag{55}$$

$$\sum_{f\prime=1}^{F\prime} \Omega_{Mem}^{f\prime}.R_{(m,n)}^{f\prime} \le C_m^{Mem}.\mu\,, \quad \forall m \in N_{CDC}, \forall n \in N_s \tag{56}$$

Using Constraint (57), we ensure the reliability requirement of SFC request *f*.

$$Re^f = \prod_{x \in X} Re_x \,.\, Re_{CDC}^m \,.\, E_{m,x}^f \;\ge\; \mathrm{r}^f, \quad \forall m \in N_{CDC}, \forall f \in F \tag{57}$$

After specifying the necessary constraints, we present our proposed heuristic method for dynamic service function chaining with the aim of facilitating *URLLC* in NFV in a dynamic scenario. As stated before, in order to optimize the latency and reliability of high-priority SFC requests, our heuristic algorithm reserves a specific amount of physical network resources only for their embedding. Our heuristic technique is referred to as the *DAAS* embedding algorithm, and its pseudocode is shown in *Algorithm 7*.

Algorithm 7. ***DAAS Algorithm***

Input: G_s=(N_s,L_s), Src^f, Des^f, O_x^f, r^f, T_a^f, T_d^f;

Output: $Selected\_route^f$ ← Selected route for SFC request *f*;

1: **for** *t* in range 1,10 **do**

2: **for** each SFC request *f* at time *t* **do**

3: $Selected\_route^f$ = empty;

4: *CN = Src (f)* ← Set source of *f* as Current-Node (*CN*);

5: *Calculate_Free_Resources* (Bandwidth, memory, CPU) ← Via *Algorithm 8*;

6: Prune (N_{CDC},L_s) ← Pruning *CDC* nodes and links that are unable to serve SFC request *f* (*Constraints 51-56*);

7: **for** each VNF *x* in O_x^f **do**

8: Find nearest *CDC* providing *x* (CN, *x*) ← *Dijkstra*;

9: **if** Reliability [**Selected-Route]** ≥ r^s (*Constraints 57*)

10: CN = next *CDC;*

11: Update *Used Resources (Algorithm 9)*;

12: Get Used Resources← Record the used resources *(Algorithm 10)*

13: Update *Selected_Route;*

14: **end if**

15: **end for**

16: Find shortest path from *CDC* to the *Des (flow)*←*Dijkstra*;

17: Update *Used Resources (Algorithm 9)*;

18: Get Used Resources← Record the used resources *(Algorithm 10)*

19: Update *Selected_Route;*

20: **end if**

21: **end for**

22: **end for**

23: **return** $Selected\_route^f$ *;*

The optimum path for delivering SFC request *f* in conjunction with end-to-end delay and reliability requirements is returned by *Algorithms 7-10*. To do this, we require the following parameters for each SFC request: source node, destination node, VNF ordering matrix, reliability requirement, and arrival and departure time of SFC request *f*. In *Algorithm7*, we establish 10 time-cycles for mapping SFC requests (line 1), which some SFC requests entering and leaving the network at their arrival time (T_a^f) and departure time (T_d^f). At each time-cycle, we collect all incoming SFC requests (line 2), then define an empty *Selected_Route* list to record each SFC request's optimum path (line 3). Line 4 begins at the source node of each SFC request. Line 5 computes the available physical network resources (bandwidth, memory, and CPU) using *Algorithm 8*, and then prunes the physical links and nodes based on the available physical network resources (line 6). For each VNF in the ordering matrix of SFC request *f* (line 7), the *Dijkstra* algorithm will be utilized to identify the closest *CDC* nodes that can provide the necessary VNFs (line 8). In line 9, the *Selected_Route* reliability requirement will be examined. The program then keeps track of the used physical resources (line 11) and calculates the available physical resources that are released at the departure time of SFCs (line 12) using *Algorithm 10*. After discovering the necessary VNFs for SFC request *f* (line 13), the algorithm

identifies the shortest path between the most recent *CDC* node and the SFC request's destination node (line 16). Finally, it updates the used resources, released resources at departure time, and *selected route* (line 17-19) before returning the shortest route (line 23).

Algorithm 8 calculates the free network resources for mapping SFC requests based on their priority. In order to have dynamic SFC embedding, we utilize SFC arrival time (T_a^f) and departure time (T_d^f). In the first time-cycle (line 2), it subtracts the *Used_Resources* (derived via *Algorithm 9*) from 90 percent of the total network capacity (because the starting value of μ is 0.9) (line 3). In line 4, if the flow has high-priority, then it can utilize the reserved 10 percent ((1- μ) × Network Capacity) of physical network resources (line 5). Line 7 sets the free resources to 0, if low-priority SFC requests require more than 90 percent of physical resources. The *Free Resources* after time t=2 are calculated on line 10. The released resources (derived in *Algorithm 10*) must be taken into account here; thus, we not only subtract the used resources but also add the released resources. In line 11, if the flow has high-priority, it can utilize the reserved 10 percent ((1- μ) × Network Capacity) of physical network resources (line 12). Line 15 returns the *Free_Resources.*

Algorithm 9 updates *Used_Resources* for SFC request *f* according to its priority. It begins by examining the *Selected_Path* of SFC request *f* in lines 1-2. In line 3, if the SFC request *f* has a high-priority, then the algorithm uses the reserved 10 percent of network capacity for path deployment. If the allocated 10 percent of the network capacity is insufficient for path deployment, the remaining 90 percent of the network capacity is utilized (line 6). In line 9, if the SFC request *f* has low-priority, the algorithm reduces the required physical resource only from the 90 percent of the total network capacity. Line 12 returns the *Used_Resources* for SFC request deployment.

Algorithm 8. ***Calculate_Free_Resources***

Input: *Flow f,* P^f , T_a^f , T_d^f ;

Output: *Free_Resources (Bandwidth, memory, CPU);*

1: **Calculate_Free_Resources (*Flow f,* T_a^f, T_d^f)**

2: **if** t < 2 **then**

3: *Free_Resources*= (μ × Network Capacity) – *Used_Resources;*

4: **if** *Flow f* has high-priority **then**

5: *Free_Resources*= *Free_Resources* + ((1- μ) × Network Capacity);

6: **end if**

7: **if** *Free_Resources* < 0 **then** *Free_Resources*=0; ← Since (1- μ)% is reserved for high-priority, the *Free Resources* value for low priority can become negative.

8: **end if**

9: **else**

10: *Free_Resources*= (μ × Network Capacity) – *Used_Resources* + *Released Resources* at time (T_d^f)

11: **if** Flow *f* has high-priority **then**

12: *Free_Resources*= *Free_Resources* + ((1- μ) × Network Capacity) + *Released Resources* at time (T_d^f);

13: **end if**

14: **end if**

15: **return** *Free_Resources*;

Algorithm 9. ***Update_Used_Resources***

Input: *Selected_Route, Flow f,* P^f *;*

Output: *Used_Resources (Bandwidth, memory, CPU);*

1: **Used Reources (*Selected_Route, Flow f*)**
2: **for** each (*node m → node n)* in *Selected_Route* **do**
3: **if** *Flow f* has high-priority **then**
4: reduce *Required Resources* from ((1- μ) × Network Capacity);
5: **if** ((1- μ) × Network Capacity) < *Required Resources* **then**
6: reduce the remaining from (μ × Network Capacity);
7: **end if**
8: **else**
9: reduce the *Required Resources* from (μ × Network Capacity);
10: **end if**
11: **end for**
12: **return** *Used_Resources;*

Algorithm 10 computes the released resource when an SFC request's lifetime expires at time *t*. To do this, the route, priority, and departure time of each SFC request are required. The used resources that are waiting to be released are recorded by *Algorithm 10*. In order to make the method dynamic, we now introduce two new types of three-dimensional matrices, *High_used_resources* and *Low_used_resources,* which record the *Used_Resources* by high-priority and low-priority SFC requests, respectively. Each matrix indicates how much resource should be released at the departure time *t* (T_d^f). As for the algorithm, first it checks the *Selected Route* of SFC request *f* in line 1-2. In line 3-7, if the

request has high-priority, it will first record the *Used Resources i*n *High_used_resources* with its departure time (T_d^f). When the reserved 10 percent of network capacity is not enough for this request, the rest of *Used Resources* will be recorded in *Low_used_resources* with its departure time (T_d^f). Line 9 records the *Used Resources* of low-priority in the *Low_used_resources* with its departure time (T_d^f). Line 12 returns the *Used resources* at time *t*, which is considered as resources that should be released at time *t*.

Algorithm 10. ***Get_Used_Resources***

Input: *Selected_Route, Flow f,* P^f, T_d^f;

Output: *Released Resources at time t (Bandwidth, memory, CPU);*

1: **Released Reources (*Selected_Route, Flow f,* T_d^f)**
2: **for** each (*node m* → *node n)* in *Route* of *f* at T_d^f **do**
3: **if** *Flow f* has high-priority **then**
4: add *Required Resources* to *high_used_recources*;
5: **if** ((1- μ) × Network Capacity) < *Required Resources* **then**
6: add *Required Resources* - (1- μ) × Network Capacity) to *low_used_recources;*
7: **end if**
8: **else**
9: add *Required Resources* to *low_used_recources*;
10: **end if**
11: **end for**
12: **return** *Released Resources at time t;*

5.4 Nearest Service Function First (NSF) Algorithm

The ***N**earest-**S**ervice-Function-**F**irst (**NSF**)* algorithm proposed by Tajiki *et al.* [27] is used as a comparison algorithm. *Algorithm 11* contains a presentation of the pseudocode for the *NSF* algorithm. Their method makes use of the commonsense approach of locating the closest server that supports the VNFs in the chain of the flow *f* to deliver service. It is important to note that the *NSF* method does not utilize any sort of prioritizing or physical resource reserve while determining the optimal deployment routes. It treats SFC requests in an equal manner. The pseudocode of the *NSF* algorithm is presented in *Algorithm 11*.

As described in [27], for proper operation of this algorithm, the following parameters must be provided as input. We need the ordering list of needed VNFs that is specified by the matrix *K* for each SFC request *f*. It is necessary for us to know the source and destination of each SFC request, denoted by *s* and *d* correspondingly. The *N* denotes the number of servers in the physical network. Line 1 does an algorithmic iteration for each SFC request (F) to discover the most efficient deployment paths for SFC requests. Line 2 constructs a *Selected Path* (*SP*) list containing the ideal deployment pathways for each SFC request *f*. In Line 3, the source of the SFC request *f* is declared to be the current node, allowing the algorithm to begin here. Lines 4-8 employ the *Dijkstra* algorithm to identify the nearest server that can provide the required VNFs of the SFC request *f* in a row according to the VNF ordering matrix. If the selected route meets the reliability requirement of the SFC request (line 6), then the utilized bandwidth resources will be updated in line 9 in the appropriate manner. Then, locate the shortest path from the most recently formed VNF to the destination node (line 11), and add that path to the *selected path*. This process will take place in line 12. It is necessary for us to get the resource for total bandwidth usage up to date (line 13). Line 15 will return the best possible deployment path.

Algorithm 11. ***NSF Algorithm***

Input: K, s, d, N

Output: ***SP*** ← *SP* is the Selected Path;

1: **for** each flow *f* in *F* **do**
2: SP^f = empty;
3: *CN* = *s* ← *CN* is the current server;
4: **for** each VNF *k* in K **do**
5: [*v*, *p*] = *Find_Nearest_Providers* (*CN*, *k*, *N*);
6: **if** Reliability [**Selected-Route]** ≥ required-reliability **then**
7: add *p* to SP^f
8: *CN* = *v*;
9: B^{max}= *Reduce_Capacity* (B^{max}, *MFS*, *p*);
10: **end for**
11: *p = Shortest_Path (CN, d);*
12: add *p* to SP^f
13: B^{max}= *Reduce_Capacity* (B^{max}, *MFS*, *p*);
14: **end for**
15: **return** *SP;*

5.5 Greedy Algorithm

A *Greedy* algorithm is a type of algorithm that solves a problem by selecting the optimal option currently available. Numerous studies [48, 45, 64, 65, 46, 36] employ the well-known *Greedy* method as a comparison technique for the SFC deployment problem. Although a *Greedy* algorithm can be straightforward and effective, it may not necessarily result in the optimal solution. It is unconcerned

about whether the current best result will lead to the overall best result. The algorithm never undoes a previous choice, even if it was erroneous. It operates in a top-down fashion. This algorithm might not produce the best solution for all problems. Because it always chooses the best option locally to produce the best outcome globally [67, 68]. The pseudocode of *Greedy Algorithm* is presented in *Algorithm 12*. When it comes to determining the optimal deployment pathways, the *Greedy* algorithm does not make use of any sort of prioritization or physical resource reservation, and it handles SFC requests in an equal manner. This is something that has to be emphasized.

The following set of input parameters must be supplied for the algorithm to be executed. We need access to the physical network and all of its physical network resources $G_p = (N_P, L_P)$. Following that, we need information on service function chains, including the origin and destination of each SFC request, Src^f, Des^f. We need to have the VNF ordering matrix $W_{(F \times X)}$ and the maximum tolerable delay for each SFC request, τ_{td}^f. The expected output consists of the optimally determined path for each SFC request. The initial SFC request received by the system serves as our starting point. While adhering to the required limitations, we must begin with the source node of each SFC request and make our way to the destination of each SFC request.

Line 1 executes an algorithmic iteration for each SFC request (*F*) to discover the most efficient deployment paths for SFC requests. In line 2, we construct a *Selected Path* (SP) list to keep track of the optimal deployment pathways for each SFC request *f*. Line 3 specifies the present node as the source of the SFC path between the last *CDC* node that is providing the final required VNF for SFC request *f* and the destination node for SFC request *f*. We then update the request *f*, allowing the algorithm to begin at this node. Line 4's objective is to

Algorithm 12. ***Greedy Algorithm***

Input: $G_p = (N_P, L_P)$ ← Physical Network;

$Src^f, Des^f, W_{(F \times X)}$

Output: SP^f ← Selected Path for SFC request *f*;

1: **for** each SFC request *f* **do**
2: SP^f = empty;
3: *CN = Src (f)* ← Set source of *f* as Current Node (*CN*);
4: Prune (N_{CDC}, L_P) ←Pruning the *CDC* nodes and the links,
which cannot be used to serve SFC request *f*;
5: **for** each VNF *x* in W_x^f **do**
6: Find Nearest *CDC* Providing *x* (CN, *x*) ← *Greedy* algorithm;
7: **if** Reliability [**Selected-Route]** ≥ r^f **then**
8: CN = next *CDC;*
9: Update *Used Resources*;
10: Update *SP;*
11: **end for**
12: Find shortest path from *CDC* to the *Des (flow)* ←*Greedy* algorithm;
13: Update *Used Resources*;
14: Update *SP;*
15: **end for**
16: **return** *SP;*

eliminate the *CDC* nodes and physical links that cannot be employed to satisfy SFC request *f*. Lines 5-7 employ the *Greedy* algorithm to identify the *CDC* nodes nearest to the SFC request *f* that can provide the required VNFs. These nodes are positioned based on the values included in the VNF ordering matrix *W*. In the meantime, it examines the reliability constraint of the selected path in line 7. After identifying the routes with the shortest distances to the *CDC* nodes that provide the required VNFs for the SFC request *f*, we will update both the *Used Resources* (line 9) and the *Selected Path* fields (line 10). Line 12 finds the shortest path from the *CDC* node that provides the last required VNF to the destination of the flow in line 12. Then, we update *Used Resources* in line 12 and the *Selected Path* in line 14. Line 16 returns the SFC request's most efficient deployment paths.

6 Performance Evaluations

In this chapter, we present the results of the performance evaluation conducted on the algorithms proposed in our study, covering the different phases of our research. It provides a comprehensive description of the parameters employed in the performance evaluation and conducts an assessment of the resulting outcomes. Python was employed as the programming language for executing the simulations, in conjunction with the PuLP library and CBC Solver. The computational task is executed on a computer system equipped with an Intel Core i7-8550U CPU, operating at a frequency range of 1.80GHz to 1.99GHz, and a memory capacity of 24 GB.

To assess the efficacy of our proposed algorithms, we have identified a set of Key Performance Indicators (KPIs) for various stages of our investigation through a comprehensive examination of cutting-edge studies. The objective of this study is to investigate *Ultra-Reliable Low Latency Communication* within a network enabled by NFV. The KPIs that we will focus on are latency, reliability, bandwidth utilization, and SFC acceptance rate. In order to ensure comparability and fairness, we utilize identical parameter settings when conducting comparisons between different algorithms. The current chapter is organized in the subsequent manner: In the subsequent section, denoted as Section 6.1, an examination is conducted to explore the outcomes of the investigation regarding *Ultra-Low Latency Communication*. In Section 6.2, this study investigates the outcomes associated with *Ultra-Reliable Low Latency Communication*. In Section 6.3, an analysis is conducted on the outcomes of the Dynamic Service Function Chaining, which was executed to tackle *URLLC* in a dynamic setting. The subsequent sections provide an expanded description of our research findings.

6.1 Ultra-Low Latency Communication

This section will outline the simulation setup and performance evaluation results of the algorithms proposed to address *Ultra-Low Latency Communication*. In this study, we conduct a comparative analysis of four embedding algorithms, namely: (a) the ***OAS*** embedding algorithm, which utilizes Equation (27) to obtain exact numerical solutions; (b) the ***FAS*** embedding algorithm, which yields near exact numerical solutions; (c) the ***NSF*** algorithm as proposed in a related study [27]; and (d) the ***Greedy*** algorithm, a widely recognized baseline algorithm [48, 45, 64, 65, 46, 36]. Both the *NSF* method and the *Greedy* algorithm are based on the closest server that provides the required VNFs, as indicated in Chapter 5. However, the *Greedy* approach uses the *Greedy* algorithm whereas the *NSF* algorithm uses the *Dijkstra* algorithm to find the shortest path between two nodes. It is important to note that neither the *NSF* nor the *Greedy* algorithms take *flow prioritization* or physical network resource reservation into account.

In our simulation, we use the Gridnet network topology [69] as our substrate network, which consists of 8 nodes and 18 links. Two nodes are considered *CDC* nodes according to node degree to host the VNF instances, and six nodes are regarded as switching nodes to pass traffic to the following nodes. We define six different VNF types, and each *CDC* node can host a maximum of three different VNF types. The bandwidth capacity between nodes *m* and *n (m, n* $\in$ *Np)* is considered to be in the range of [500-1000] Mbps, location-dependent. We set different *CDC* node capacities in terms of storage and computing capabilities based on their different network locations. We set the memory and *CPU* capacity of the first *CDC* node at 1500 MB and 1500 MIPS, respectively, and for the second *CDC* node at 2000 MB and 2000 MIPS. As previously stated, we assume that the memory and CPU capabilities of switching nodes are limitless, as they just transmit traffic to the next nodes and do not require a great

deal of memory and CPU capacity.

The propagation delay of link *(m,n)* follows a uniform distribution in the range of [2, 21] milliseconds, computed based on the nodes' distances and the medium type of Gridnet network topology [60]. We assume the initial *priority coefficient factor* (μ) to be 0.9, and the change in this value is also studied. Indicating low-priority SFC requests are permitted to use up to 90 percent of physical network resources, and 10 percent of physical network resources are reserved for SFC requests with high-priority. In this simulation setup, we generate 150 SFC requests with a random source and destination node. We assume ten percent of SFC requests are high-priority SFC requests, which are among the second half of SFC requests. The required bandwidth (τ_{bw}^{f}), *CPU* (τ_{cpu}^{f}) and memory (τ_{mem}^{f}) of each SFC request *f* are set as numbers distributed randomly between (0, 10] [29]. We set the number of required VNFs per SFC request at three [29]. Last but not least, the maximum tolerated delay (τ_{td}^{f}) of each SFC request is in the range of [50,100] ms [29]. When comparing *OAS*, *FAS*, and *NSF* algorithms, it is vital that we utilize the same settings for our simulation parameters. This is done for the sake of fairness [70].

The end-to-end delay is the most crucial KPI in this investigation. Figure 12 displays the measurements of end-to-end delay for the *OAS*, *FAS*, *NSF*, and *Greedy* algorithms. In this graph, (×) represents the average end-to-end delay of these four algorithms. The results of the study provide clear evidence that both *OAS* and *FAS* exhibit significant efficacy in improving the average end-to-end delay, particularly for SFC requests with high-priority, when compared to *NSF* and *Greedy*. When examining high-priority SFC requests, it is observed that the *OAS* and *FAS* algorithms demonstrate a significant decrease of 22 percent and 17 percent, respectively, in the average end-to-end delay when compared to the *Greedy* algorithm. The *OAS* and *FAS* algorithms facilitate the optimization of

provisioning paths by enabling ultra-low latency applications to utilize the reserved 10 percent of physical network resources allocated for high-priority SFC requests. Additionally, the findings show that the average end-to-end delay for SFC requests with low-priority remains similar when comparing the *FAS* and *NSF* algorithms. Nevertheless, the *FAS* algorithm exhibits a modest decrease of 2 percent in the average end-to-end delay, in contrast to the *Greedy* algorithm. On the other hand, the *OAS* algorithm demonstrates a significant enhancement of 15 percent in the average end-to-end delay when compared to the *Greedy* algorithm. The results of this study emphasize the considerable advantages provided by *OAS* and *FAS* algorithms in decreasing end-to-end delay for high-priority SFC requests, thereby improving the overall efficiency and performance of ultra-low latency applications.

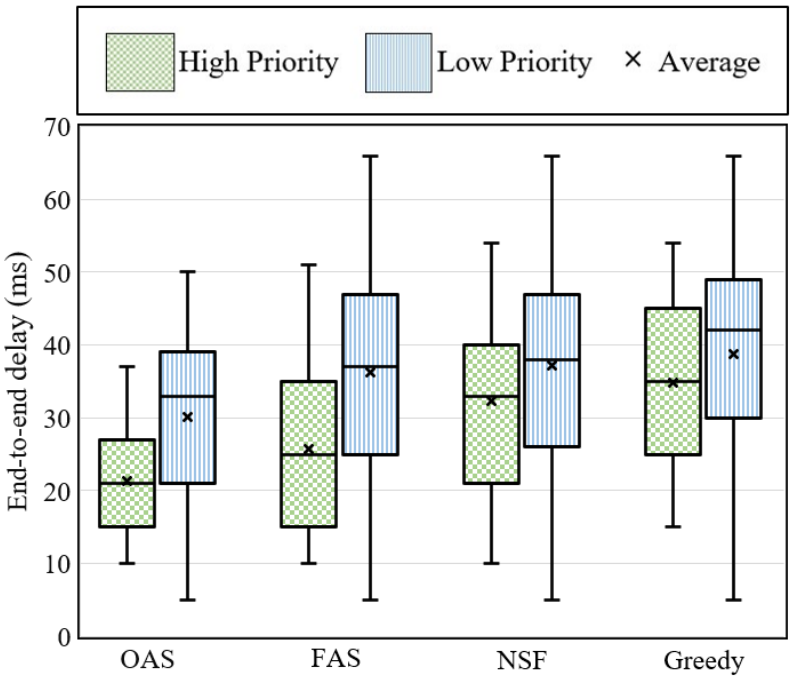

Figure 12. End-to-end delay (the first phase of the study)

The second KPI in this study applies to the efficient utilization of bandwidth resources. In recent years, the use of bandwidth has increased dramatically due

to an increase in the use of new services. Therefore, it is regarded as an essential KPI and is of great importance to network providers. Figure 13 illustrates the utilization of bandwidth resources in the provisioning paths of SFCs by the four algorithms. (×) reflects the average bandwidth use of these four approaches in this graph.

The results indicate that, on average, the *OAS* and *FAS* algorithms demonstrate reduced bandwidth utilization (28 percent and 24 percent, respectively) in comparison to the *Greedy* algorithm when handling high-priority SFC requests. The reason for this can be attributed to the utilization of reserved physical resources by *OAS* and *FAS* algorithms, which leads to the creation of shorter provisioning paths for high-priority requests. On the other hand, the *Greedy* algorithm may produce outcomes that are locally optimal but do not achieve global optimality. Furthermore, the findings indicate that, on average, low-priority SFC requests in the *FAS* consume approximately two percent more bandwidth resources in comparison to the *NSF*. The observed discrepancy can be attributed to the assignment of dedicated physical resources for SFC requests with high-priority, resulting in less optimal deployment paths for requests with low-priority. In addition, it can be observed that the *OAS* algorithm demonstrates a decrease in bandwidth utilization of 10 percent and 12 percent, respectively, in comparison to the *NSF* and *Greedy* algorithms. Overall, when considering both priority levels, the *OAS* algorithm demonstrates a reduction in bandwidth resource consumption of approximately 14 percent in comparison to the *Greedy* algorithm. The aforementioned discoveries provide insights into the advantages provided by *OAS* and *FAS* algorithms in enhancing the optimization of bandwidth utilization within SFC provisioning paths. Consequently, this leads to enhanced resource efficiency and allocation.

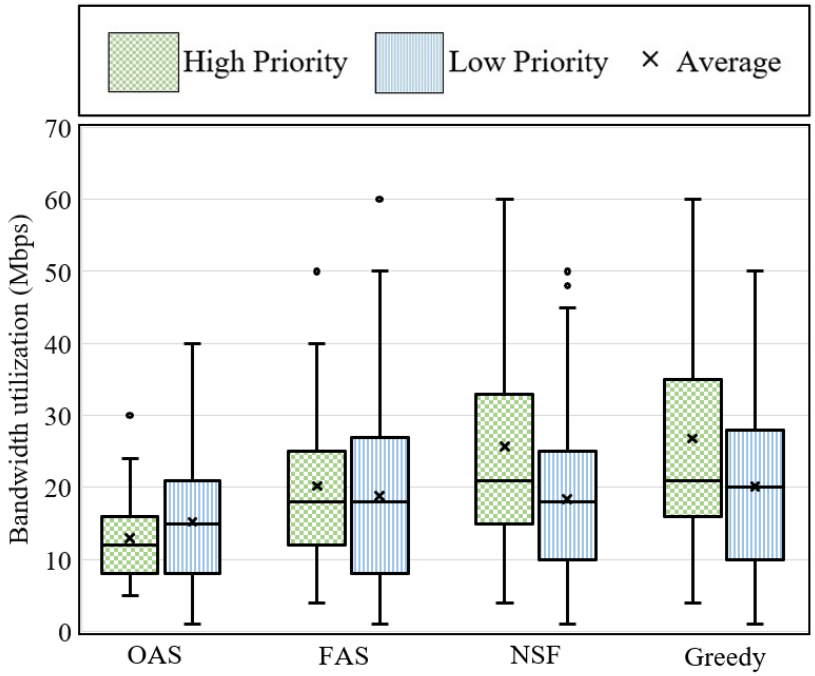

Figure 13. Bandwidth utilization (the first phase of the study).

Changing the proportion of high-priority SFC requests to low-priority SFC requests is an additional element worth investigating. What happens if the fraction of SFC requests with a high-priority rises? The *priority coefficient factor* remains unchanged in this case. Figure 14 presents an analysis of the average end-to-end delay experienced by provisioning paths, with a focus on the varying proportion of high-priority SFC requests in relation to the total SFC requests. The findings indicate that when the percentage of SFC requests classified as high-priority rises, reaching a maximum of 10 percent, the *OAS* and *FAS* algorithms exhibit the lowest average end-to-end delay in comparison to the *NSF* and *Greedy* algorithms for high-priority SFC requests. The observed enhancement can be attributed to the effective utilization of allocated physical network resources. Nevertheless, upon surpassing this threshold, specifically when the proportion of high-priority SFC requests reaches 30 percent, a notable escalation in the average end-to-end delay for high-priority SFC requests can be

observed in both the *OAS* and *FAS* algorithms, in contrast to the *NSF* and *Greedy* algorithms. This phenomenon occurs as a result of the full utilization of the 10 percent reserved physical resources by a subset of high-priority SFC requests. As a result, the remaining SFC requests with high-priority are obligated to utilize the non-reserved physical resources. This ultimately results in suboptimal mapping and non-optimal provisioning paths.

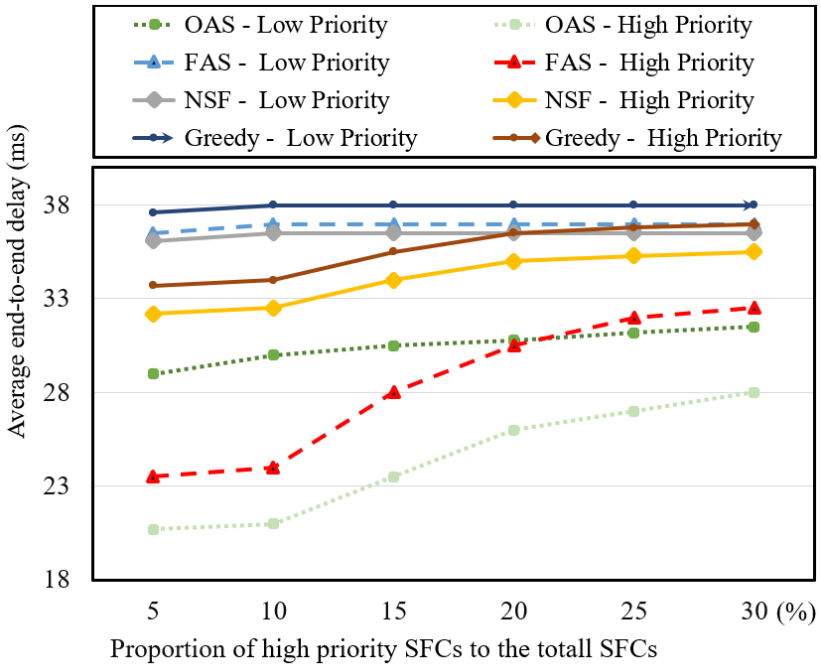

Figure 14. Average end-to-end delay over proportion of high-priority SFCs to the total SFCs (the first phase of the study).

Figure 15 illustrates the SFC acceptance rate attained by the four aforementioned algorithms, which serves as a measure of their effectiveness in mapping SFC requests. The SFC acceptance rate is the percentage of effectively mapped SFC requests relative to the total number of SFC requests. Results indicate a decline in SFC acceptance rates as the number of VNFs required for each SFC request increases. Nevertheless, it is noteworthy that *OAS* and *FAS* algorithms, which utilize allocated physical resources, exhibit smaller decreases in SFC acceptance rates for high-priority SFC requests compared to *NSF* and

Greedy algorithms when the number of VNFs increases. When six VNFs are requested per SFC, the acceptance rates for high-priority SFC requests are as follows: 95% for *OAS*, 90% for *FAS*, 85% for *NSF*, and 82% for *Greedy*. This demonstrates that *OAS* and *FAS* algorithms are able to mitigate the decline in SFC acceptance rates to a greater degree, emphasizing their efficiency in processing high-priority SFC requests even with a larger number of VNFs.

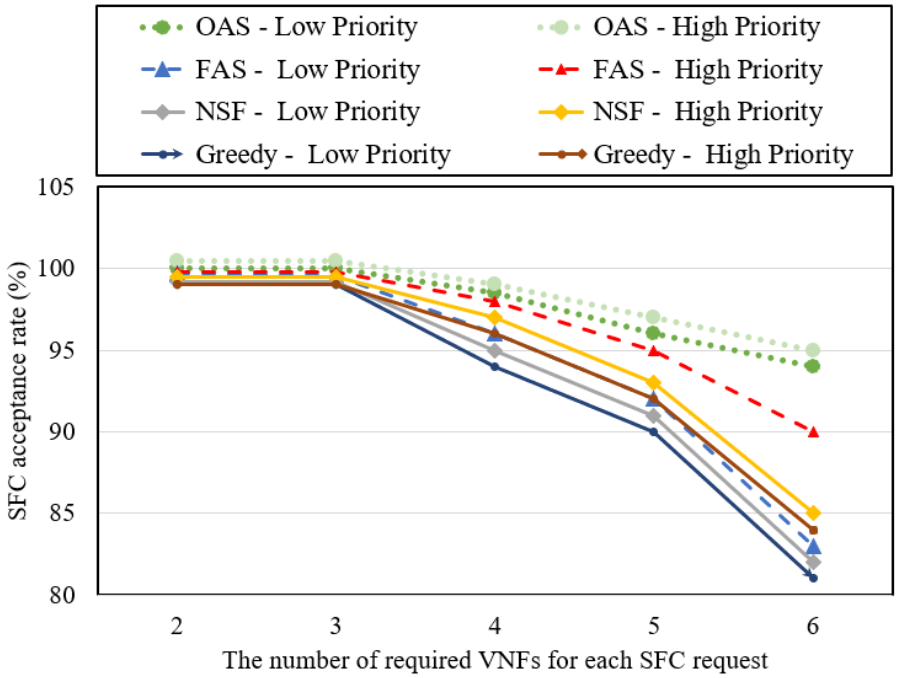

Figure 15. SFC acceptance rate (the first phase of the study).

Finally, as previously stated, we examine the variation of the *priority coefficient factor* (μ) in our research. The selected initial *priority coefficient factor* (μ) was 0.9, leading to a reservation of 10 percent of physical network resources that are exclusively designated for accommodating high-priority SFC requests. Figure 16 presents an analysis of the impact of varying the proportion of physical network resources allocated to high-priority SFC requests, denoted by the *priority coefficient factor* (μ). The findings indicate a consistent decrease in the average end-to-end delay of high-priority SFC requests for both *OAS* and

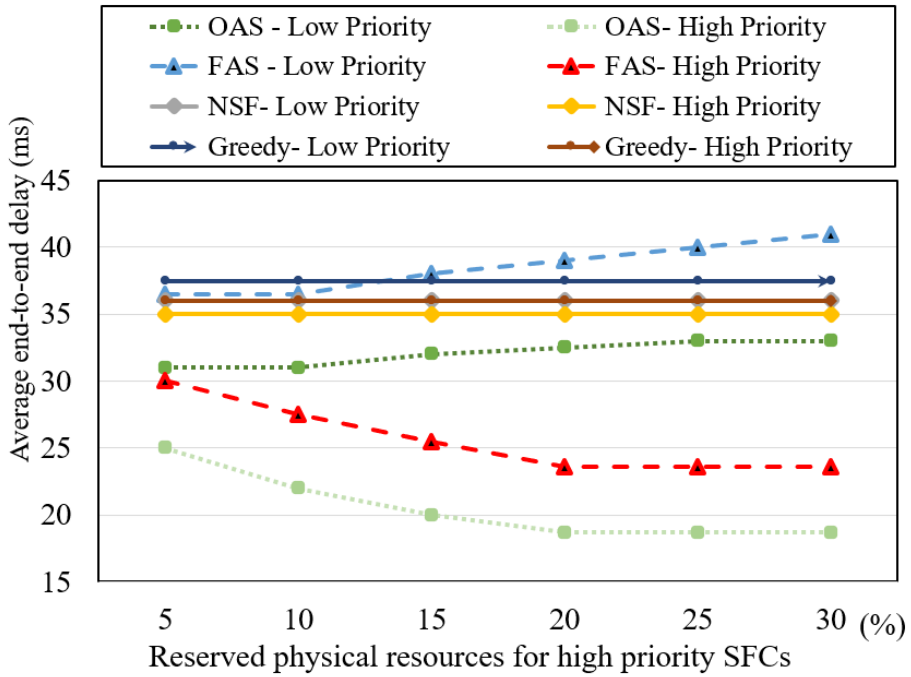

Figure 16. Average end-to-end delay over reserved physical resources (the first phase of the study).

FAS algorithms as the proportion of allocated physical network resources increases, until reaching the 20% data point. At this point, the average end-to-end delay reaches its lowest value and remains unchanged as the reserved physical resources are further increased, up to 30 percent. The results of this study indicate that in the given test scenario, high-priority SFC requests achieve optimal outcomes when a 20% reservation of physical resources is made. Furthermore, it was observed that further reservation of physical resources does not have any effect on the provisioning paths of high-priority SFC requests. In contrast, as expected, the average end-to-end delay of low-priority SFC requests for *OAS* and *FAS* algorithms experiences a significant increase after the 15% data point. This is attributable to the greater reservation of physical resources for high-priority SFC requests, resulting in insufficient resources for low-priority requests. Consequently, low-priority SFC requests are compelled to select longer paths due to the scarcity of physical resources. The results for the

NSF and *Greedy* algorithms remain unaffected by changes in physical resource reservations for high-priority SFC requests, as they operate independently of such reservations. These findings suggest that the incorporation of a dynamic *priority coefficient factor* (μ) holds the potential to optimize overall results further. By dynamically adjusting the *priority coefficient factor* (μ), the system can adapt to variations in network load or changes in the proportion of high-priority to low-priority requests. This dynamic adjustment mechanism allows for a flexible and responsive approach, enabling the system to achieve optimal outcomes based on the prevailing conditions and priorities within the network environment.

6.2 Ultra-Reliable Low Latency Communication

By expanding the use of *URLLC* applications such as autonomous vehicles, remote surgery, tele-operated driving, virtual reality, augmented reality, and industrial automation, addressing the SFC deployment problem has received considerable attention from the scientific community. In the second section of this chapter, we will discuss the outcomes of the second phase of our study regarding *Ultra-Reliable Low Latency Communication*. We compare the results of our proposed optimal SFC embedding algorithm, *ORAAS*, and our proposed heuristic algorithm, *NORAAS*, which were described in detail in Chapters 4 and 5 of our study, with the *NSF* algorithm proposed in [27] and the well-known *Greedy* algorithm as a baseline as used in [48, 45, 64, 65, 46, 36]. As stated in Chapter 5, both the *NSF* algorithm and the *Greedy* algorithm are based on the nearest server that offers the needed VNFs. However, while the *NSF* algorithm employs the *Dijkstra* algorithm to determine the shortest path between two nodes, the *Greedy* approach uses the *Greedy* algorithm. It should be highlighted that both the *NSF* and *Greedy* algorithms do not account for *flow prioritization* or physical network resource reservation.

Following is a description of the parameters utilized in the performance evaluation. Similar to the previous phase of our research, Gridnet network topology [60] is utilized in our simulation for the second phase of our study. It consists of 8 nodes and 18 links, of which we define two nodes as *CDC* nodes according to node degree to host the VNF instances and six nodes as switching nodes to forward traffic to the next nodes. Six different VNF types are defined, and each *CDC* node may support a maximum of three different VNF types. Each physical link is assumed to have a bandwidth capacity in the range of [500-1000] Mbps, depending on the network location. We set up *CDC* nodes with varied capacities for storage and processing based on their network locations. The first *CDC* node's memory and CPU capabilities were set to 1500 MB and 1500 MIPS, respectively, while the second *CDC* node's memory and CPU capacities were set to 2000 MB and 2000 MIPS, respectively. We assume that *Switching Nodes* (N_S) do not need a large amount of CPU and memory resources, as they just transmit network traffic to the following nodes and do not require a great deal of memory and CPU. We thus regard their CPU and memory as infinite.

In our simulation setting, we set the propagation delay of each link as a uniform distribution in a range of [3, 41] milliseconds. As stated, we assume the initial *priority coefficient factor* (δ) as 0.9, which means that 10 percent of physical network resources are reserved for embedding only high-priority SFC requests (*URLLA*) and low-priority SFC requests are allowed to use up to 90 percent of physical network resources. In this simulation scenario, the bandwidth, CPU, and memory requirements of each SFC request are specified as random values between (0,10] [29]. We set the maximum tolerable delay at each SFC request in the range of [50, 100] milliseconds [29]. We produce 150 SFC requests with a random source and destination node. We assume that ten percent of SFC requests have a high-priority and that the first fifty percent of

SFC requests have a low-priority. We set the number of VNF requested per SFC request as three. The software reliability of VNF instances is distributed in the range of [0.99, 0.999] similar to [49]. The first and second *CDC* nodes have hardware reliability of 0.99 and 0.999, respectively. The required reliability for each SFC request was chosen from the following ranges: [0.95, 0.98, 0.99, 0.992, 0.999] [49]. We use the same simulation parameter settings when comparing the algorithms to guarantee comparability. Since we deal with *ultra-reliable low latency communication* in an NFV-enabled network, our primary KPIs are end-to-end delay, reliability, bandwidth usage, and SFC acceptance rate. These are the primary KPIs determined after examining various cutting-edge studies, as discussed in Chapter 2.

One of the most important KPIs that we study is the end-to-end delay of SFC requests using these four algorithms (*ORAAS, NORAAS, NSF*, and *Greedy*). The term "end-to-end delay" pertains to the duration required for a packet to be transmitted from its originating node to its intended destination node across a network. As can be seen in Figure 17, it presents the average end-to-end delay obtained by these four algorithms for low- and high-priority SFC requests. The green color represents the average end-to-end delay of high-priority SFC requests, whereas the blue color represents the average end-to-end delay of low-priority SFC requests.

As it is presented, *ORAAS* and *NORAAS* significantly reduce the average end-to-end delay of high-priority SFC requests compared to *NSF* and *Greedy*. The reason is that they use of 10 percent reserved physical network resources (bandwidth, memory, and CPU) for mapping high-priority SFC requests and can obtain more optimal deployment paths. More in detail, *ORAAS* (33%) and *NORAAS* (19%) have a smaller end-to-end delay for high-priority SFC requests than *Greedy*. Moreover, *ORAAS* (30%) and *NORAAS* (17%) reduce end-to-end delay for low-priority SFC requests compared to *Greedy*. We conclude that

ORAAS and *NORAAS* utilize their physical network resources more efficiently than *NSF* and *Greedy*. This approach is highly advantageous for applications that are sensitive to latency, such as autonomous cars, remote surgery, augmented reality, etc.

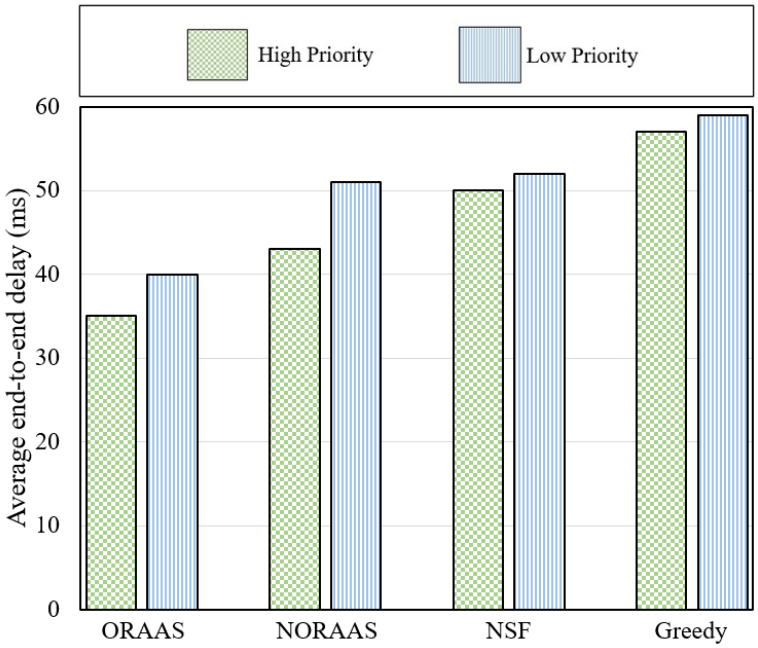

Figure 17. Average end-to-end delay (the second phase of the study).

The next parameter to analyze is bandwidth resource consumption. Figure 18 depicts the bandwidth usage of SFC deployment paths. (×) reflects the average bandwidth usage of these four algorithms in this graph. As can be seen, the *Greedy* algorithm consumes more bandwidth than *ORAAS*, *NORAAS*, and *NSF*. In this setup, as represented, *ORAAS* and *NORAAS* use bandwidth resources more effectively than *NSF* and *Greedy*. More in detail, *ORAAS* and *NORAAS* consume 30 percent and 22 percent fewer bandwidth resources for high-priority SFC requests, respectively, than *Greedy*. Additionally, *ORAAS* consumes 18 percent, 20 percent, and 24 percent less bandwidth than *NORAAS*, *NSF*, and *Greedy* for low-priority SFC requests, respectively. When the deployment path is more optimal, the amount of bandwidth required for SFC deployment is also

reduced. As a result of the increased demand for bandwidth during the past few years, *ORAAS* and *NORAAS* will be essential for optimizing resource use.

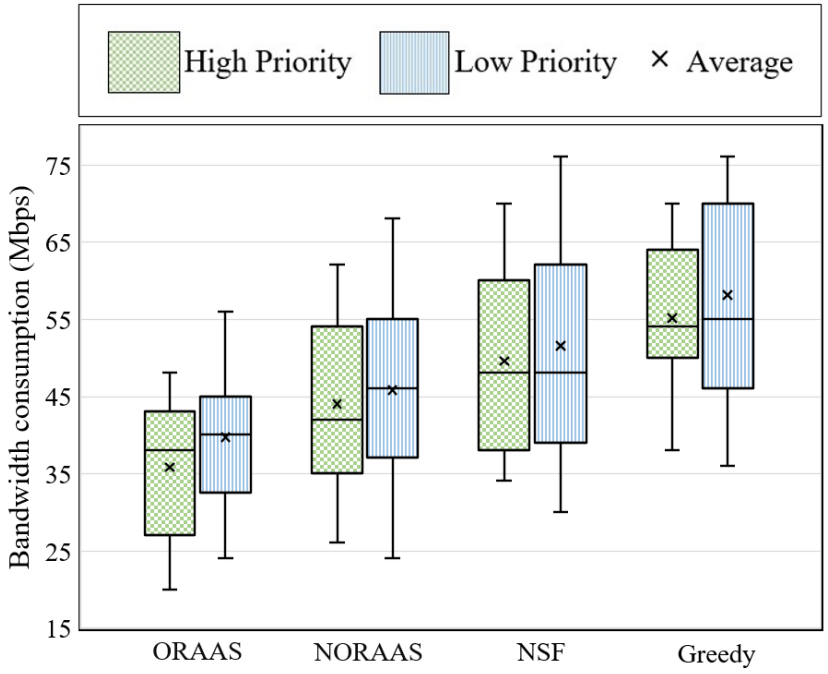

Figure 18. Bandwidth consumption (the second phase of the study).

As indicated previously, we examine the effect of altering the *priority coefficient factor* (δ), specifically what occurs when we reserve additional physical resources for SFC requests with high-priority. The impact of varying the *priority coefficient factor* (δ) on *ORAAS* and *NORAAS* is represented in Figure 19. As shown, increasing the amount of reserved physical network resources for high-priority SFC requests reduces their average end-to-end delay while increases the average end-to-end delay of low-priority SFC requests. As we raise the reserved physical resources up to 30 percent, the average end-to-end delay of high-priority SFC requests falls gradually until the 20 percent data point, where it stays unchanged after reaching this point. The reason is that high-priority SFC requests get the most optimal deployment paths by reserving 20 percent of physical network resources, and higher physical network resource reserve has no effect on provisioning paths. In contrast, the exclusive physical

network resources allocation for high-priority SFC requests increases the average end-to-end delay of low-priority SFC requests, indicating that a sweet spot for the *priority coefficient factor* (δ) must be identified.

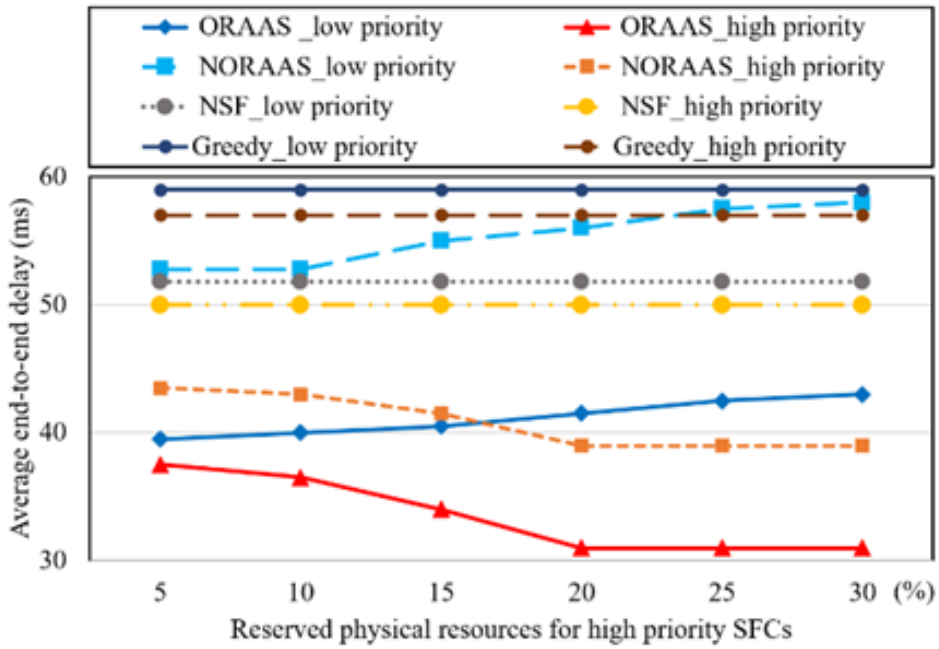

Figure 19. Average end-to-end delay over reserved physical resources for high-priority SFC requests (the second phase of the study).

The next essential KPI is the SFC acceptance rate, which represents the algorithm's effectiveness in terms of mapping SFC requests. The SFC acceptance rate is the percentage of mapped SFC requests compared to the total number of SFC requests. In order to evaluate the SFC acceptance rate, the number of VNF instances requested by each SFC request is increased from two to six, and the SFC acceptance rate of these four algorithms is evaluated by increasing the network load. Figure 20 presents the SFC acceptance rate in relation to the change in the number of required VNFs for each SFC request. As expected, the acceptance rate for SFC requests declines as the number of needed VNFs rises. As network demand grows, there will not be sufficient resources for mapping SFC requests. Therefore, algorithms that make more effective use

of physical network resources will have a higher SFC acceptance rate. In more detail, this reduction is more gradual for *ORAAS* and *NORAAS* than for *NSF* and *Greedy*. As shown in the graph, at the six needed VNF datapoints, *ORAAS*, *NORAAS*, *NSF*, and *Greedy* achieve acceptance rates of 95 percent, 92 percent, 91 percent, and 88 percent, respectively. In this sense, *ORAAS* has the highest SFC acceptance rate, whereas *Greedy* has the lowest.

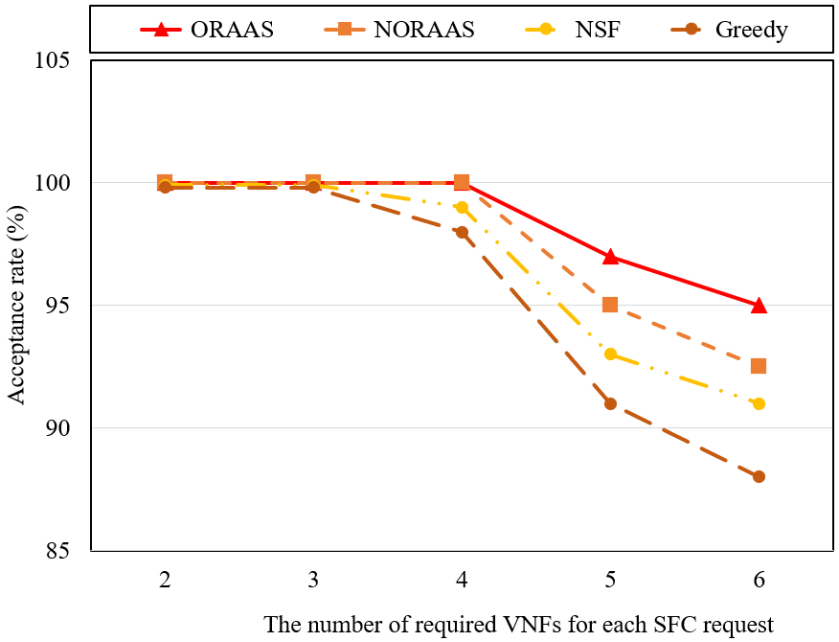

Figure 20. SFC acceptance rate (the second phase of the study).

In the final stage, we analyze the average reliability of these four algorithms. Since our goal is to address *URLLC* in NFV-enabled networks. Elements have different reliability values, as specified in Chapter 3, and we consider both hardware and software reliability. In order to do this, we vary the network load and assess the average reliability attained by these four algorithms. Figure 21 depicts the average reliability in response to the varying number of VNF instances required by each SFC request. As it can be seen, the average reliability decreases as the length of SFC requests increases, and this decline is more moderate for high-priority SFC requests by *ORAAS* and *NORAAS*. This is due

to the fact that SFC requests with high-priority have the benefit of accessing reserved physical network resources and hence the most reliable components. To this end, *ORAAS* and *NORAAS* are superior to *NSF* and *Greedy* in terms of average reliability and are optimum for addressing *URLLC*. In this regard, the *ORAAS* algorithm has the highest degree of reliability, whereas the *Greedy* algorithm has the lowest.

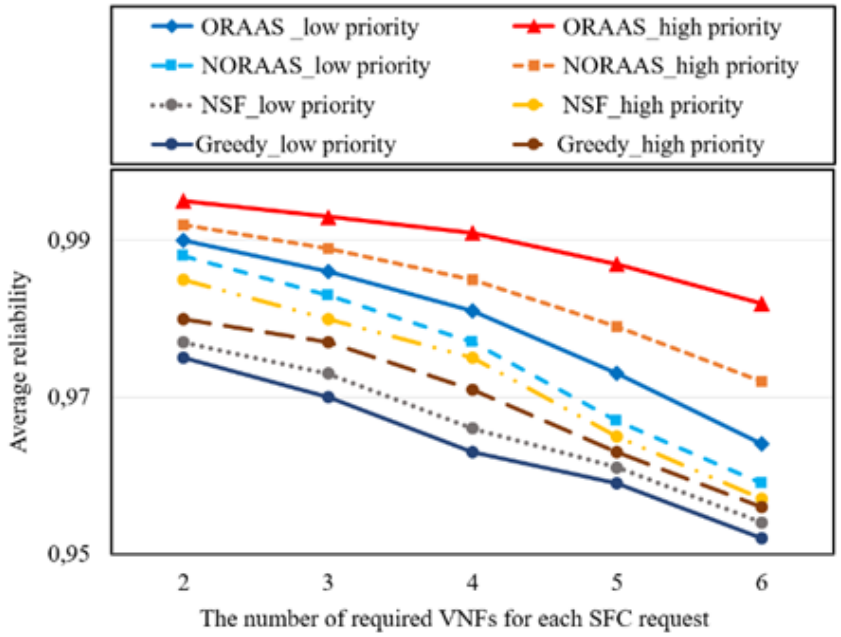

Figure 21. Average reliability over varying the number of required VNFs (the second phase of the study).

6.3 Dynamic Service Function Chaining

In the final section of this chapter, we explore the outcomes of the dynamic SFC deployment problem, in which each SFC request has an arrival and departure time to utilize physical network resources. Since the goal of this research is to investigate *URLLC* in an NFV-enabled network, we examined *URLLC* in a dynamic scenario. As we discussed in Chapter 5, we offered the *DAAS* algorithm, a heuristic technique, to handle *URLLC* in a dynamic scenario.

In the next section, we analyze the effectiveness of our suggested heuristic dynamic SFC embedding technique, the *DAAS* algorithm. We compare the *DAAS* algorithm with the *NSF* algorithm developed in [27] and the well-known *Greedy* algorithm as used in [48, 45, 64, 65, 46, 36]. As noted earlier, both the *NSF* and *Greedy* algorithms are built on the closest server that provides the required VNF instances. Nevertheless, whereas the *NSF* technique utilizes the *Dijkstra* algorithm to establish the shortest path between two nodes, which obtains a global optimal result, the *Greedy* approach employs the *Greedy* algorithm, which may obtain a local optimal result. It should be noted that neither the *NSF* nor the *Greedy* Algorithms account for *flow prioritization* or physical network resource reservation. The simulation is implemented in *Python* using the *PuLP* library. It is performed on a laptop with a 1.80GHz 1.99GHz Intel Core i7-8550U processor and 24 GB of RAM.

Different from the previous two phases of our research, we chose the EliBackbone network topology [60] as our substrate network, which is a larger network topology than the Gridnet network topology. Figure 11 represents the EliBackbone network topology. It consists of 19 nodes and 28 links, of which we define three nodes as *CDC* nodes according to node degree to host the VNF instances and sixteen nodes as switching nodes to forward traffic to the next nodes. Six different VNF types are defined, and each *CDC* node may support up to a maximum of three different VNF types.

In our simulation setting, we employ the following input values. It is considered that each link has a bandwidth capacity between 800 and 2000 Mbps, depending on the network location. We configure *CDC* nodes with varying storage and CPU processing capacities based on their network locations. The memory and CPU capacities of two *CDC* nodes are 2500 MB and 2500 MIPS, respectively, while the memory and CPU capacities of the third *CDC* node are 3000 MB and 3000 MIPS, respectively. We suppose that *Switching Nodes* (N_S)

do not demand a considerable amount of CPU and memory resources, as they only transmit network traffic to subsequent nodes and do not need a lot of memory and CPU. Consequently, we consider their CPU and memory to be unlimited. We set the propagation delay of each link as a uniform distribution in a range of [2, 23] milliseconds, which is determined by the nodes' distances and the EliBackbone network topology's medium type. As stated, we assume the initial *priority coefficient factor* (μ) to be 0.9, which means that low-priority SFC requests are permitted to utilize only up to 90 percent of physical network resources (bandwidth, memory, and CPU), and ten percent of physical network resources are reserved for embedding only high-priority SFC requests (*URLLA*). In this simulation scenario, the bandwidth, CPU, and memory requirements of each SFC request are specified as random values between (0,10] [29].

To evaluate these three algorithms (*DAAS, NSF*, and *Greedy*), 600 SFC requests with random source and destination nodes and varying lifetimes are generated. In our dynamic scenario, we set 10 time-cycles in which a certain number of SFC requests may enter or depart the network throughout each time cycle. We assume that 10 percent of SFC requests have high-priority. We assume that the first 50 percent of SFC requests entering the network have low-priority. We set the number of VNF requested per SFC request as three. The software reliability of VNF instances is distributed in the range of [0.99, 0.999] similar to [61]. The hardware reliability of *CDC* nodes is distributed in the range of [0.99, 0.999]. The required reliability for each SFC request was chosen from the following ranges [0.95, 0.98, 0.99, 0.992, 0.999] [61]. We use the same simulation parameter settings when comparing the algorithms to guarantee comparability and fairness.

End-to-end delay is one of the most important KPIs to access. End-to-end delay is the amount of time it takes for a network packet to be sent from its source node to its destination node. The end-to-end delay produced by these

three algorithms (*DAAS*, *NSF*, and *Greedy* algorithms) for low- and high-priority SFC requests is depicted in Figure 22. In this graph, the end-to-end delay for high-priority SFC requests is shown in green, while the end-to-end delay for low-priority SFC requests is shown in blue. In this figure, (×) represents the average end-to-end delay of these three algorithms. As depicted, by reserving 10 percent of physical network resources for high-priority SFC requests, *DAAS* reduces the average end-to-end delay of high-priority SFC requests significantly in comparison to *NSF* and *Greedy*. *DAAS* enables ultra-low latency applications to utilize the 10 percent of reserved physical resources for high-priority SFC requests; hence, the provisioning paths of *DAAS* are more optimal. *DAAS* achieves 9 percent less average end-to-end delay than the *NSF* method and 13 percent less average end-to-end delay than the *Greedy* algorithm. For low-priority SFC requests, *DAAS* achieves 4 percent less average end-to-end delay than *Greedy* but 2 percent higher than *NSF*. In comparison to *Greedy*, *NSF* achieves a 7 percent and 5 percent reduced average end-to-end delay for

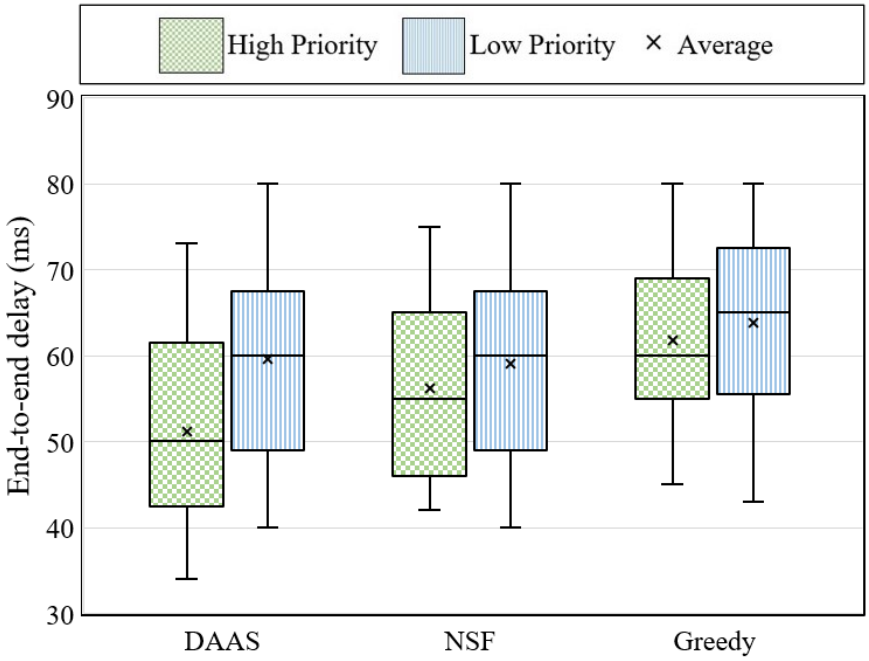

Figure 22. Average end-to-end delay (the third phase of the study).

high-priority and low-priority SFC requests, respectively.

Bandwidth consumption is the second key performance indicator we consider. Figure 23 depicts the bandwidth usage for SFC deployment paths using these three algorithms. In this graph, (×) represents the average bandwidth utilization of these three methods. The results indicate that, on average, the *DAAS* algorithm uses less bandwidth resources than the *NSF* approach and the *Greedy* algorithm while handling high-priority SFC requests. As demonstrated, *DAAS* utilizes 11 percent and 17 percent less bandwidth resources for high-priority SFC requests than *NSF* and *Greedy*, respectively. Due to the allocated physical resources, the *DAAS* algorithm provides shorter provisioning paths for high-priority SFC requests than the *NSF* method and the *Greedy* algorithm. As a result, the *DAAS* algorithm utilizes less bandwidth than the *NSF* algorithm and the *Greedy* algorithm. In this configuration, *DAAS* consumes 4 percent less bandwidth than the *Greedy* algorithm for low-priority SFC requests but 2

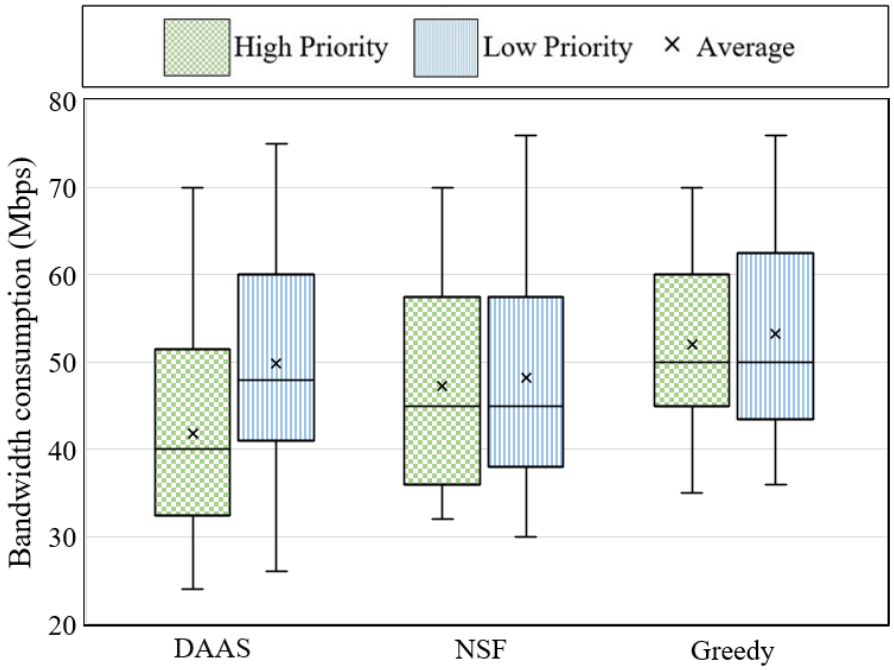

Figure 23. Bandwidth consumption (the third phase of the study).

percent more than the *NSF* algorithm. As can be seen, *Greedy* uses the most bandwidth resources for both high- and low-priority SFC queries. Since it may obtain local optimum route deployment, additional bandwidth resources are required.

The average path length is the next KPI that can demonstrate the success of these three algorithms (*DAAS*, *NSF*, and *Greedy* algorithms). It demonstrates how effectively these three algorithms can map SFC requests. The average path length of these three methods is depicted in Figure 24. As presented, *DAAS* achieves the shortest average path length among other algorithms for high-priority SFC requests. Therefore, *DAAS* may have a considerable positive effect on mapping SFC requests with the highest priority. This is because SFC requests with high-priority in *DAAS* have access to the reserved physical network resources and may thus acquire a more optimal deployment path. Although

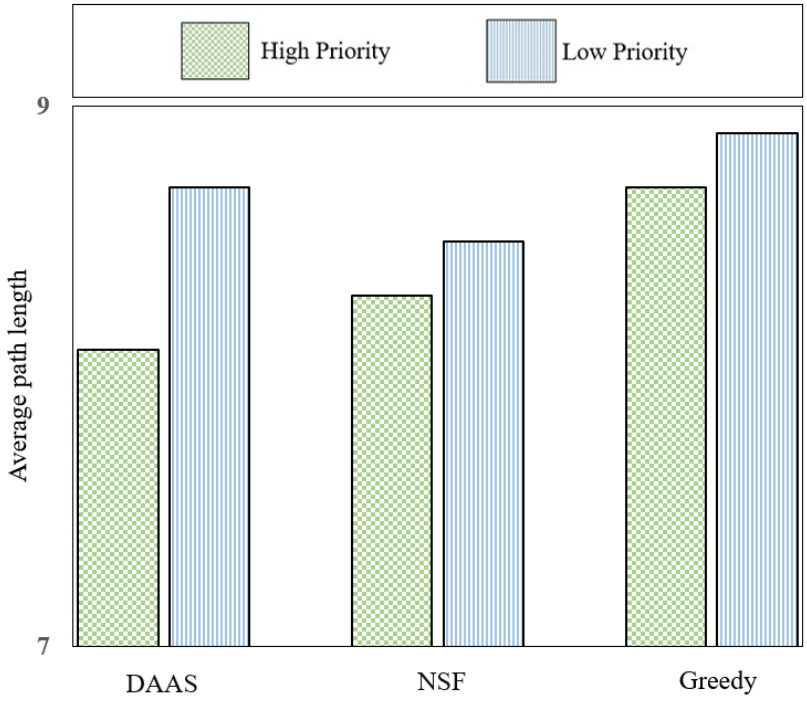

Figure 24. Average path length (the third phase of the study).

DAAS has achieved the shortest path length for high-priority SFC requests among the proposed algorithms, it achieves a longer path length for low-priority SFC requests than the *NSF* algorithm. As can be observed in the graph, the *NSF* approach achieves shorter average path lengths than the *Greedy* algorithm. This is because *NSF* gets global optimal results using the *Dijkstra* algorithm, whereas the *Greedy* technique may return local optimal results.

Figure 25 presents the SFC acceptance rate for each of the three algorithms (*DAAS*, *NSF*, and *Greedy*). The SFC acceptance rate is an important KPI that demonstrates the algorithm's efficacy in mapping SFC requests. The proportion of mapped SFC requests relative to the total number of SFC requests is the SFC acceptance rate. As indicated previously, 600 SFC requests with random source and destination nodes and varying lifetimes have been produced. As can be seen

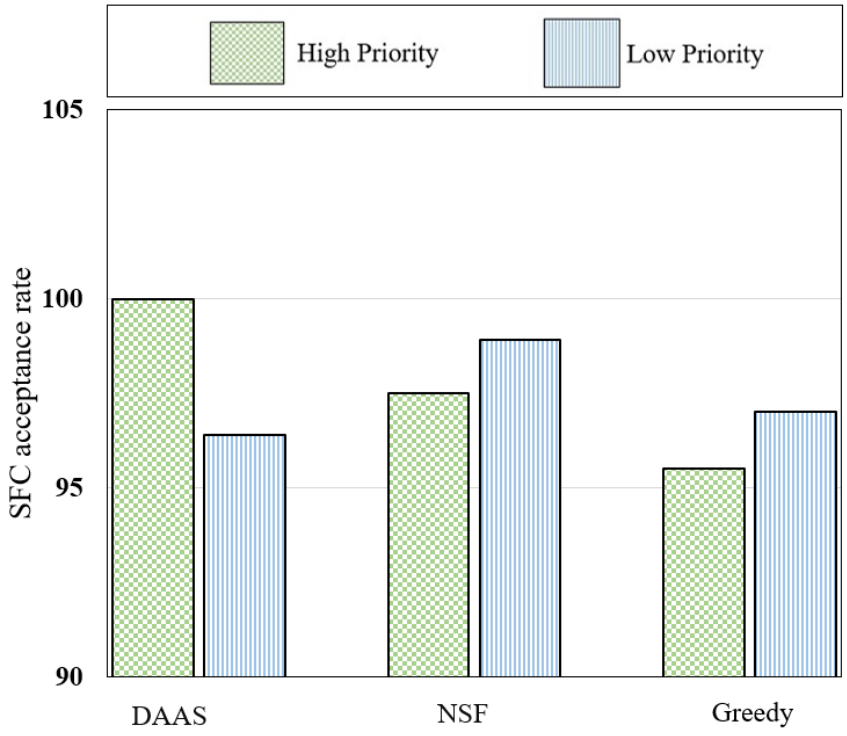

Figure 25. SFC acceptance rate (the third phase of the study).

in the graph, the *DAAS, NSF,* and *Greedy* algorithms achieve 100 percent, 97 percent, and 96 percent of the SFC acceptance rate for SFC requests with high-priority, respectively. *DAAS* has a 100 percent acceptance rate for SFC requests with high-priority. It indicates that all high-priority SFC requests function properly and are implemented into the network. This is because, while utilizing *DAAS*, SFC requests with high-priority have access to reserved physical resources. Clearly, it helps a great deal in facilitating *URLLC*, but we must also consider its disadvantages with respect to low-priority SFC requests. It is crucial to maintain a balance between SFC requests of high- and low-priority.

Figure 26 depicts the average end-to-end delay of SFC requests utilizing these three algorithms while chaining the proportion of high-priority SFC requests to total SFC requests. As can be seen in the graph, *DAAS* achieves a lower average end-to-end delay compared to the other two algorithms up to the 20 percent data point. At the 25 percent and 30 percent data points, the average end-to-end delay of high-priority SFC requests increases significantly. This is due to the fact that at these two datapoints, the reserved physical resources have been used completely by some high-priority SFC requests, and the remaining high-priority SFC requests were unable to find the optimal deployment paths. To this end, it is essential to find an appropriate value for the *priority coefficient factor* in order to obtain the optimum results. This issue may be resolved by employing a dynamic *priority coefficient factor* that adjusts based on network service demand. This suggests that, utilizing network load prediction methods, the dynamic *priority coefficient factor* may self-adjust to achieve optimal outcomes according to the network load. As depicted, the *Greedy* algorithm obtains the highest end-to-end delay among the other proposed algorithms (*DAAS* and *NSF*).

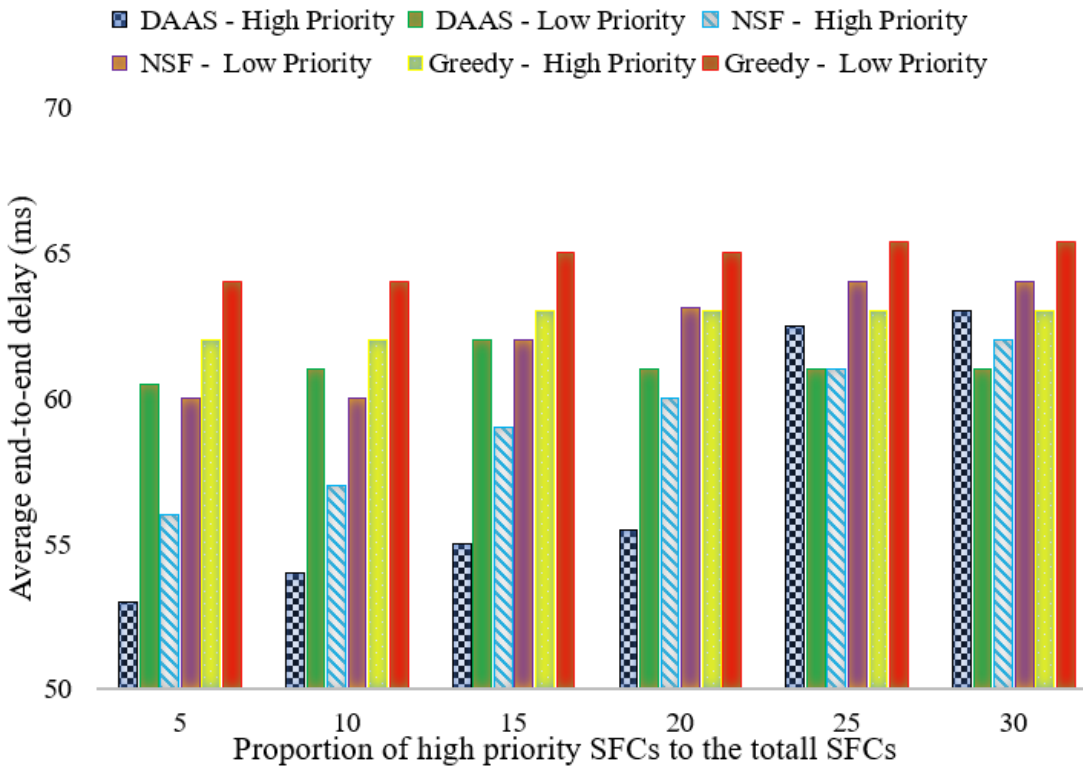

Figure 26. Average end to end delay over varying proportion of high-priority SFCs to the total SFCs (the third phase of the study).

As the final step, we assess the average reliability of these three algorithms (*DAAS*, *NSF*, and *Greedy*). Considering that our objective is to handle *URLLC* in NFV-enabled networks. In Chapter 3, reliability is defined, and it is emphasized that each element has a specific reliability value. We take into account both hardware and software reliability. Figure 27 depicts the average reliability of these three algorithms. As can be observed, the *DAAS* algorithm obtains the maximum level of reliability for high-priority SFC requests. Due to the utilization of physical resource reservations, SFC requests with the highest priority have access to the most reliable components. Therefore, this strategy appears to be quite helpful in tackling *URLLC*. For SFC requests with low-priority, *DAAS* achieves the same degree of reliability as the *NSF* algorithm. As shown in this graph, the *Greedy* algorithm achieves the lowest average

reliability in this configuration for both high- and low-priority SFC requests. This is because the *Greedy* algorithm obtains the least optimal deployment path by obtaining locally optimal outcomes.

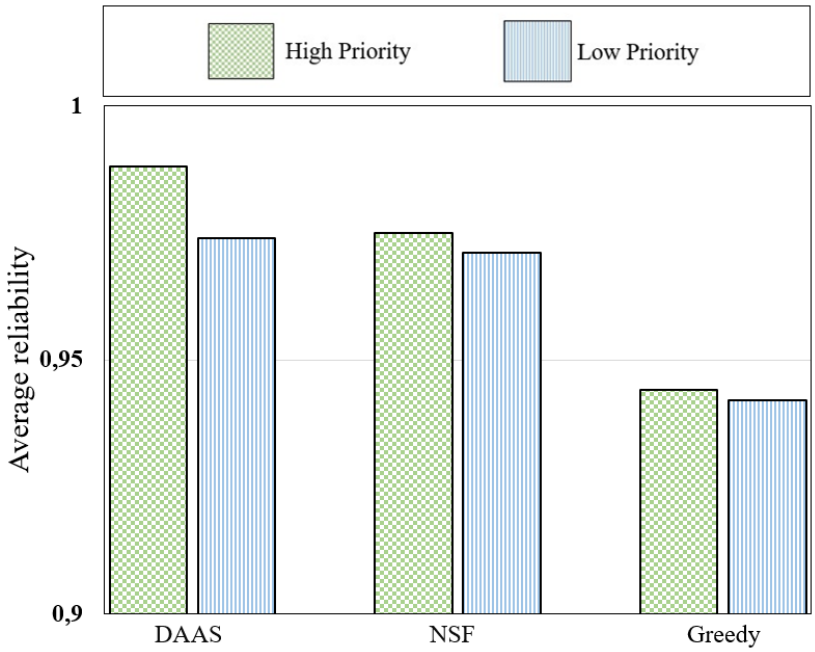

Figure 27. Average reliability (the third phase of the study).

7 Conclusion and Future Work

In this thesis, we conducted research on NFV. We observed that NFV has the potential to revolutionize traditional network architectures and eliminate some of their limitations. Enabling *URLLC* in NFV is one of the most essential topics in this field, as *URLLA* use has increased dramatically in recent years. In the first chapter of the thesis, the NFV architecture was depicted and the advantages of NFV were discussed. We observed that the successful implementation of network services in NFV heavily depends on the deployment of SFC. Therefore, the SFC deployment problem remains an important challenge in NFV, necessitating further research. In the second chapter, we conducted a study on the existing studies in this field and reviewed the proposed methodologies for enhancing reliability and latency in NFV. We observed that in order to enhance reliability, backup mechanisms and redundant elements are frequently employed, whereas latency-aware service function chaining is implemented to reduce latency. We discussed that improving both reliability and latency at the same time is exceedingly challenging since they may have negative interactions. Improving reliability may result in an increase in latency, and vice versa. Additionally, the network's physical resource limitations make it more challenging. To this end, in order to simultaneously improve reliability and latency in the SFC embedding phase, we proposed a novel solution with considerable benefits.

In Chapter 3, the system model and its underlying assumptions that are used to evaluate the proposed methodology are presented. The presence of a well-defined system model is crucial in order to generate outcomes that are both precise and closely aligned with the real-world context. In Chapter 4, the mathematical formulations and optimization model were discussed. The SFC

deployment problem was structured as an ILP optimization problem with restrictions on maximum tolerable end-to-end delay, reliability, bandwidth, memory, and CPU usage. Establishing all the required constraints in a linear format is a very difficult operation. In Chapter 5, a set of heuristic algorithms and relaxed versions were provided to attain near-optimal results while minimizing both the execution time and optimality gap in order to make them applicable to large-scale network topologies and solve the scalability problem. In Chapter 6, the assessment findings were presented and showed the significant enhancements realized by the proposed methodologies in terms of end-to-end delay, bandwidth consumption, SFC acceptance rate, and reliability.

7.1 Conclusion

The deployment of SFC is a highly difficult process since the traffic flow must be directed through a series of functions, and it is difficult to establish a good trade-off between a number of essential requirements. The objective of this study was to enable *URLLC* within a network that is enabled by NFV. In order to facilitate *URLLC* within the context of NFV, we focused on the **SFC Deployment Problem**. This particular problem represents a significant obstacle to the implementation of an NFV-enabled network. We introduced a novel and efficient algorithm for SFC deployment to address both latency and reliability. Our algorithm aims to minimize latency and optimize reliability for *URLLA* during the SFC embedding phase. Notably, our approach does not rely on backup methods or redundant elements.

Using a configurable *priority coefficient factor* and *flow prioritization*, we were able to simultaneously enhance the latency and reliability of *URLLA* without the need for backup techniques. To do this, we reserved a certain amount of physical network resources (bandwidth, RAM memory, and CPU)

exclusively for SFC requests with high-priority (*URLLA*). In order to minimize any side effects on low-priority SFC requests, we imposed constraints on the reliability and the maximum tolerable end-to-end delay not just for high-priority SFC requests but also for low-priority SFC requests. We mathematically formulated the SFC deployment problem as an ILP optimization model to obtain exact numerical solutions. Following that, we also offered a set of heuristic approaches and relaxed versions to minimize execution time with a minimal optimality gap in order to solve the scalability problem and make our proposed approach usable for large-scale network topologies.

The performance evaluations revealed that our suggested algorithms can significantly enhance the end-to-end delay, reliability, bandwidth utilization, and SFC acceptance rate of *URLLA*. We discovered that our suggested *ORAAS* and *NORAAS* algorithms reduced the end-to-end latency of *URLLA* by 33 percent and 19 percent, respectively, compared to the *Greedy* algorithm. In addition, for low-priority applications, *ORAAS* and *NORAAS* achieved 30 percent and 17 percent less end-to-end latency than the *Greedy* method, respectively. In terms of bandwidth consumption, *ORAAS* and *NORAAS* spent 30 percent and 22 percent less bandwidth resources than the *Greedy* algorithm for *URLLA*, respectively. Moreover, *ORAAS* utilized 18 percent, 20 percent, and 24 percent less bandwidth for low-priority SFC requests than *NORAAS*, *NSF*, and *Greedy*, respectively. In terms of reliability, *ORAAS* and *NORAAS* achieved the highest reliability for *URLLA* compared to the *NSF* and *Greedy* algorithms. Last but not least, compared to *NSF* and *Greedy*, *ORAAS* and *NORAAS* achieved the highest SFC acceptance rate for *URLLA*. *ORAAS*, *NORAAS*, *NSF*, and *Greedy* attain acceptance rates of 95 percent, 92 percent, 91 percent, and 88 percent, respectively, at the six required VNFs datapoints. Chapter 6 contains further details. To this end, we observed that our proposed methodology is a promising solution to enable *URLLC* in an NFV-enabled network.

7.2 Future Work

The investigation focused on the SFC deployment problem in order to facilitate *URLLC* within an NFV-enabled network. A novel mathematical model incorporating all essential constraints was developed in this regard, and different techniques were presented to address the SFC deployment problem from various viewpoints. The findings of our study are published in [71, 72, 73, 74, 75]. These publications provide a valuable foundation for future advancements in *URLLC* within an NFV-enabled network. The **SFC deployment problem** is one of the main challenges in an NFV-enabled network, and it requires more development. We only covered a small portion of it. Incorporation of additional constraints, such as power consumption limitations and load balancing prerequisites, within our optimization framework is feasible. Moreover, as we discussed in Chapter 6, our research can be improved by using a dynamically configurable *priority coefficient factor* to dynamically modify the reservation of physical network resources based on the load of different network services. To fully leverage the promises of NFV, there are still a vast number of considerations to make. In the following, we present a list of recommendations for further research:

A. A trendy field to research is the use of an effective backup mechanism with the least negative influence on latency to increase system resilience further. The valuable analysis of a student on backup approaches to address *URLLC* in an NFV environment [76] and several backup techniques presented in [77, 78, 79, 80, 81, 82, 83] can be applied to our proposed optimization model to further improve *URLLC* in an NFV-enabled network.

B. The integration of Machine Learning techniques emerges as a promising approach for addressing the SFC deployment problem. Machine learning has recently been widely recognized as a promising approach for addressing the SFC deployment problem. Different machine learning solutions, as in [84, 85, 86, 87, 88, 89], have been proposed by researchers, which require further developments. Integrating our proposed methodology into the SFC deployment problem using machine learning techniques is a fascinating research topic.

C. The next suggested field of research is Multi-access Edge Computing (MEC). MEC enables the deployment of applications at the network's edge, which is close to end users. Using this capability, MEC offers an environment with ultra-low latency, and several new applications and companies are emerging on the MEC platform [90, 65, 91, 92, 93, 94, 95]. This is another promising area of study that can be incorporated into our suggested approach.

D. The final proposed research area involves obtaining exact numerical solutions for dynamic SFC embedding scenarios with regard to network load prediction techniques, which has significant potential for future advancements as in [96, 97, 98, 99, 100, 101, 102]. This is another interesting topic that needs to be studied.

Appendix

A.1. Implementation of the ORAAS algorithm in Python

This section describes the formulation of the SFC deployment problem as an ILP optimization model in Python. Given that our primary objective is to address *URLLC* in an NFV-enabled network, we present the codes for the second phase of our study, which focuses on *Ultra-Reliable Low Latency Communication.* The second phase of our study is the completed version, building upon the first phase of our study. As a result, the *ORAAS* algorithm, which is covered in Chapter 4, is presented in the following Code-Listings. We used CBC Solver Version 2.10.3 to solve the optimization of the SFC deployment problem. We explain all the parameters and variables that have been utilized in coding in the following. We begin by creating the SFC deployment problem as Code Listing 1, which is the first step.

```
''' Step1. Create an ILP problem '''

optimization_lp_problem= pulp.LpProblem("SFC_Deployment_Problem", pulp.LpProblem)
```

Code Listing 1. Create the ILP optimization problem

In the second step, we define the variables that are involved in our optimization model as in Code Listing 2.

```
''' Step2. Define ILP variables --> cat={'Binary','Continuous','Integer'} '''

H_s_mn= pulp.LpVariable.dicts("H_s_mn", ((s,m,n) for s in range (S)
                                for m in range(N)for n in range(N)),cat='Binary')

K_s_m_x= pulp.LpVariable.dicts("K_s_m_x", ((s,m,x) for s in range (S)
                                for m in range(N)for x in range(X)),cat='Binary')

T_s_mn= pulp.LpVariable.dicts("T_s_mn", ((s,m,n) for s in range (S) for m in range(N)
                              for n in range(N)), lowBound=0, upBound= N, cat='Integer')
```

Code Listing 2. Define variables.

We define the following variables in Code Listing 2. The binary variable "*H_s_mn*" denotes the routing path of SFC request *s* between nodes *m* and *n*. "*H_s_mn=1*" if the SFC request *s* traverses the node *m* and *n*, and 0 otherwise. The binary variable "*K_s_m_x*" indicates whether VNF *x* of SFC request *s* is served by *CDC* node *m*. "*K_s_m_x*" equals 1, if the SFC request *s* uses VNF type *x*, which is placed on *CDC* node *m*, and 0 otherwise. The integer variable "*T_s_mn*" is our final required variable to implement our optimization model. It specifies the number of previously crossed nodes. Our SFC deployment optimization methodology requires these three variables.

In the next step, we define the objective function. We construct the optimization problem's objective function, which must be minimized, as in Code Listing 3. It is the Python implementation of Equation (50) from Chapter 4.

```
''' Define Objective Function'''

optimization_lp_problem+= pulp.lpSum(H_s_mn[s,m,n] * D_mn[m][n] for s in range(S)
                                     for m in range(N)for n in range(N))

optimization_lp_problem.sense =1
```

Code Listing 3. Define objective function.

In step four, we can define all of the necessary constraints, as illustrated in Chapter 4. To this end, we begin by defining the reliability constraint as Code Listing 4. It is the Python implementation of Equation (27) from Chapter 4.

```
for s in range(S):
    optimization_lp_problem+= pulp.lpSum([(1-Reliability_VNFs[x]) * K_s_m_x [s,m,x]
                for m in Nodes_CDC for x in range(X)]) - 1 <= Reliability_SFCs [s]
```

Code Listing 4. Reliability constraint.

Then, we continue with defining consumption constraints for physical network resources (bandwidth, memory, and CPU). The bandwidth consumption restriction

is shown in Code Listing 5. As shown in this formulation, low-priority SFC requests (specified in the *Flows_priority_Two* list) are authorized to use up to 90 percent of the available bandwidth resources and 10 percent of the bandwidth resources are reserved for high-priority SFC requests.

```
''' Step3. Define Constraints '''
#Bandwidth Utilization
for m in range(N):
    for n in range(N):
        optimization_lp_problem += pulp.lpSum([Bandwidth_both_priorities[s] * H_s_mn[s,m,n]
                                    for s in Flows_priority_Two]) <= BW_Capacity_Phy[m][n] * 0.90

for m in range(N):
    for n in range(N):
        optimization_lp_problem += pulp.lpSum([Bandwidth_both_priorities[s] * H_s_mn[s,m,n]
                                        for s in range(S)]) <= BW_Capacity_Phy[m][n]
```

Code Listing 5. Bandwidth utilization constraint.

Code Listing 6 displays the memory utilization restriction. Low-priority SFC requests (specified in the *Flows_priority_Two* list) are allowed to use up to 90 percent of *CDC* nodes' available memory resources. In contrast, there are no restrictions on the usage of physical network resources (bandwidth, memory, and CPU) for high-priority SFC requests (*URLLA*).

```
for m in Nodes_CDC:
    for n in range(N):
        optimization_lp_problem += pulp.lpSum([Memory_sfc[s] * H_s_mn[s,m,n] for s
                                        in range (S)]) <= Memory_Phy[m]

for m in Nodes_CDC:
    for n in range(N):
        optimization_lp_problem += pulp.lpSum([Memory_sfc[s] * H_s_mn[s,m,n] for s
                                    in Flows_priority_Two]) <= Memory_Phy[m] * 0.9
```

Code Listing 6. Memory utilization constraint.

As the last limitation on memory resource usage, Code Listing 7 applies the same logic to CPU consumption as it does to memory utilization. Low-priority SFC requests (specified in the *Flows_priority_Two* list) may occupy up to 90

percent of the available CPU resources of *CDC* nodes. Therefore, 10 percent of CPU resources are reserved for SFC requests with the highest priority.

```
for m in Nodes_CDC:
    for n in range(N):
        optimization_lp_problem += pulp.lpSum([CPU_sfc[s] * H_s_mn[s,m,n] for s
                                               in range (S)]) <= CPU_Phy[m]

for m in Nodes_CDC:
    for n in range(N):
        optimization_lp_problem += pulp.lpSum([CPU_sfc[s] * H_s_mn[s,m,n] for s
                                              in Flows_priority_Two]) <= CPU_Phy[m] * 0.90
```

Code Listing 7. CPU utilization constraint.

Code Listing 8 is the formula that we use to determine the maximum acceptable propagation delay. The end-to-end delay of the selected path cannot exceed the maximum tolerable end-to-end delay of SFC requests. See Constraint (36) in Chapter 4.

```
for s in range(S):
    optimization_lp_problem += pulp.lpSum([H_s_mn[s,m,n] * D_mn[m][n] for m
                                           in range(N) for n in range(N)]) <= D_s_td[s]
```

Code Listing 8. Propagation delay constraint.

The flow control is formulated as Code Listing 9. Using this constraint, we make sure that the links on the deployment path of SFC request *s* are connected head-to-tail. It is the Python implementation of Constraint (37) in Chapter 4.

```
for s in range(S):
    for m in range(N):
        if m == Src[s]:
            optimization_lp_problem += pulp.lpSum([H_s_mn[s,m,n] for n in
            range(N)]) - pulp.lpSum([H_s_mn[s,n,m] for n in range(N)])==1
        elif m== Des[s]:
            optimization_lp_problem += pulp.lpSum([H_s_mn[s,m,n] for n in
            range(N)]) - pulp.lpSum([H_s_mn[s,n,m] for n in range(N)])==-1
        else:
           optimization_lp_problem += pulp.lpSum([H_s_mn[s,m,n] for n in
           range(N)]) - pulp.lpSum([H_s_mn[s,n,m] for n in range(N)])==0
```

Code Listing 9. Flow control constraint.

In order to ensure that there is a connection between VNFs and the *CDC* nodes that correspond to them, we use the formulation in Code Listing 10.

```
for s in range(S):
    for m in range(N):
        for x in range(X):
            optimization_lp_problem += K_s_m_x[s,m,x] <= P_Np_X[m][x]
```

Code Listing 10. VNF and its corresponding host constraint.

In order to validate the selected route across the actual physical network, we have formulated it as Code Listing 11. *Graph[m][n]* denotes the adjacent matrix that represents the substrate network.

```
for s in range(S):
    for m in range(N):
        for n in range(N):
            optimization_lp_problem += H_s_mn[s,m,n] <= Graph[m][n]
```

Code Listing 11. Physical link insurance constraint.

Code Listing 12 specifies the loop restriction to avoid a loop in the SFC embedding. It is the Python formulation of Constraint (38).

```
for s in range(S):
    for m in range(N):
        optimization_lp_problem += pulp.lpSum([H_s_mn[s,m,n] for n in range(N)]) <=1
```

Code Listing 12. Loop constraint.

The purpose of Code Listing 13 is to verify that SFC request *s* traverses a proper VNF chain while traversing the nodes.

```
for s in range(S):
    for x in range(X):
        optimization_lp_problem += pulp.lpSum([K_s_m_x[s,m,x] for m in range(N)]) ==R_S_X[s][x]
```

Code Listing 13. VNF chain constraint.

As the next constraint, we define Code Listing 14 to make sure that each VNF type *x* is used by at most one SFC request.

```
for s in range(S):
    for x in range(X):
        optimization_lp_problem += pulp.lpSum([K_s_m_x[s,m,x] for m in Nodes_CDC]) <=1
```

Code Listing 14. VNF usage constraint.

Using the following constraints, we guarantee the VNF ordering in our problem formulation. The values stored in the matrix '*T_s_mn*' are integers and need to be equal to or higher than the corresponding one stored in the rerouting matrix '*H_s_mn'*. Therefore, we define it as Code Listing 15. It is the Python formulation of Constraint (42) in Chapter 4.

```
for s in range(S):
    for m in range(N):
        for n in range(N):
            optimization_lp_problem += T_s_mn[s,m,n] >= H_s_mn[s,m,n]
```

Code Listing 15. Ordering matrix constraint one.

As the next constraint, we ensure that '*T_s_mn'* becomes zero, if '*H_s_mn'* is zero. Therefore, we use Code Listing 16.

```
for s in range(S):
    for n in range(N):
        for m in range(N):
            if Des != Nodes:
                optimization_lp_problem += T_s_mn[s,m,n]<= N * H_s_mn[s,m,n]
```

Code Listing 16. Ordering matrix constraint two.

As the next constraint, we define that the elements of the ordering-aware rerouting matrix for the output links must be zero for the destination node. Therefore, we define it as Code Listing 17.

```
for s in range(S):
    for n in range(N):
        for m in range(N):
            if m == Des[s] and n != m:
                optimization_lp_problem += T_s_mn[s,m,n]==0
```

Code Listing 17. Ordering matrix constraint three.

Except for the source and destination nodes, when SFC request *s* enters a node in its n^{th} step, it leaves that node in the $(n+1)^{th}$ step. Therefore, we define it as Code Listing 18.

```
for s in range(S):
    for m in range(N):
        if m != Src[s] and m != Des[s]:
            optimization_lp_problem+= pulp.lpSum([T_s_mn[s,m,n] for n in range(N)]) == pulp.lpSum([T_s_mn[s,n,m]
                                      for n in range(N)]) + pulp.lpSum([H_s_mn[s,n,m] for n in range(N)])
```

Code Listing 18. Flow cross constraint.

We must ensure that SFC requests exit their source nodes. We define it as in Code Listing 19.

```
for s in range(S):
    for m in range(N):
        if m==Src[s]:
            optimization_lp_problem+= pulp.lpSum([T_s_mn[s,m,n] for n in range(N)])==1
```

Code Listing 19. Flow initiation constraint.

As the last constraint, Code Listing 20 is used to impose the sequence of VNF chaining.

```
for s in range(S):
    for v_s in range(sum(R_S_X[s][:])):
        for z_v_s in range(v_s-1):
            for M in range(N):
                for m in range(N):
                    optimization_lp_problem += (1 - K_s_m_x[s, m, v_s]) * ((2 * N) - 1)
                    + (pulp.lpSum([T_s_mn[s, m, n] for n in range(N)])) >= (K_s_m_x[s, M, z_v_s]
                    - 1) * ((2 * N) - 1) + (pulp.lpSum([T_s_mn[s, M, n] for n in range(N)]))
```

Code Listing 20. Check VNF order constraint.

Finally, we will use Code Listing 21 to solve our described SFC deployment problem and report the results.

```
'''Solving the SFC Deployment Problem'''

optimization_lp_problem.solve()

print('Problem Solving Status: {}'.format(pulp.LpStatus[optimization_lp_problem.status]))

print("Objective function value: {}".format(pulp.value(optimization_lp_problem.objective)))

print('Variables: ')
```

Code Listing 21. Solve SFC deployment problem.

Using Code Listing 1 through Code Listing 21, exact numerical solutions are provided for the SFC deployment problem presented in Chapter 4. Given that obtaining exact numerical solutions is an *NP-hard* problem and its execution is time-consuming, we present the Python implementation for our proposed heuristic method for generating a near-optimal solution in a reasonable time frame so that it may be applied in a real-world scenario.

A.2. Implementation of the NORAAS algorithm in Python

In this section, we provide the Python implementation of our suggested heuristic method, *NORAAS*, to achieve near-optimal solutions of *ORAAS* in an acceptable time frame with a minimal optimality gap. As stated before, *NORAAS* addresses ultra-reliable low latency communication in NFV. We employ the same strategy as *ORAAS*, utilizing *traffic prioritization* and physical network resource reservation to provide *URLLA* with guaranteed QoS.

To this end, first, we divide the physical network resources for high-priority SFC requests and low-priority SFC requests, as can be seen in Code Listing 22. As it is presented, using the *priority coefficient factor*, we reserve ten percent of physical network resources (bandwidth, memory, and CPU) exclusively for high-priority SFC requests.

```
ninetyPercent_Bandwidth_Phy = [[0.9 * element for element in row] for row in Bandwidth_Phy]
tenPercent_Bandwidth_Phy = [[0.1 * element for element in row] for row in Bandwidth_Phy]

ninetyPercent_CPU_Phy = [0.9 * element for element in CPU_Phy]
tenPercent_CPU_Phy = [0.1 * element for element in CPU_Phy]

ninetyPercent_Disk_Phy = [0.9 * element for element in Disk_Phy]
tenPercent_Disk_Phy = [0.1 * element for element in Disk_Phy]
```

Code Listing 22. Physical resource reservation.

```
def Dijkstra(Graph, start, goal):
    shortest_distance = {}
    predecessor = {}
    unseenNodes = Graph
    infinity = 999999
    path = []

    for node in unseenNodes:
        shortest_distance[node] = infinity
    shortest_distance[start] = 0

    while unseenNodes:
        minNode = None
        for node in unseenNodes:
            if minNode is None:
                minNode = node
            elif shortest_distance[node] < shortest_distance[minNode]:
                minNode = node

        for childNode, weight in Graph[minNode].items():
            if weight + shortest_distance[minNode] < shortest_distance[childNode]:
                shortest_distance[childNode] = weight + shortest_distance[minNode]
                predecessor[childNode] = minNode
        unseenNodes.pop(minNode)

    currentNode = goal
    while currentNode != start:
        try:
            path.insert(0, currentNode)
            currentNode = predecessor[currentNode]
        except KeyError:
            return nan, []
    path.insert(0, start)

    return shortest_distance[goal], path
    if shortest_distance[goal] != infinity:
        print('Shortest distance is ' + str(shortest_distance[goal]))
        print('And the path is ' + str(path))
```

Code Listing 23. Dijkstra algorithm.

As stated in Chapter 2, we employ the *Dijkstra* algorithm as shown in Code Listing 23 to determine the shortest path between two nodes (the start node and the goal node). We described the *Dijkstra* algorithm in Chapter 2.

Following that, we need to determine available resources (bandwidth, memory, and CPU). Code Listing 24 is where we define the function that will be used to compute the available bandwidth.

```
def Calculate_free_bandwidth(flow_i):
    FreeBandwidth = ninetyPercent_Bandwidth_Phy[:][:]
    # if flow_i has high priority:
    if flow_i in high_priority_matrix:
        # FreeBandwidth = FreeBandwidth + tenPercent_Bandwidth_Phy
        FreeBandwidth = [[FreeBandwidth[i][j] + tenPercent_Bandwidth_Phy[i][j]
                                                for j in range(N)] for i in range(N)]
    return FreeBandwidth
```

Code Listing 24. Free bandwidth calculation.

Then, similarly to the computation of available bandwidth, we develop a function that computes the amount of available CPU resource, as it is presented in Code Listing 25.

```
def Calculate_free_cpu(flow_i):
    FreeCPU = ninetyPercent_CPU_Phy[:]
    # if flow_i has high priority:
    if flow_i in high_priority_matrix:
        # FreeCPU = FreeCPU + tenPercent_CPU_Phy
        FreeCPU = [FreeCPU[i] + tenPercent_CPU_Phy[i] for i in range(N)]
    return FreeCPU
```

Code Listing 25. Free CPU calculation.

Last but not least, we compute the available memory resources by defining a function that is represented by Code Listing 26. Using Code Listing 24, Code Listing 25, and Code Listing 26, we calculate the available resources (bandwidth, memory, and CPU).

```
def Calculate_free_disk(flow_i):
    FreeDisk = ninetyPercent_Disk_Phy[:]
    # if flow_i has high priority:
    if flow_i in high_priority_matrix:
        # FreeDisk = FreeDisk + tenPercent_Disk_Phy
        FreeDisk = [FreeDisk[i] + tenPercent_Disk_Phy[i] for i in range(N)]
    return FreeDisk
```

Code Listing 26. Free memory calculation.

After providing the necessary functions to compute the availability of physical resources, we need to build the functions that allow us to keep track of how those resources are being utilized following the embedding of each SFC request. To this end, utilizing Code Listing 27, we update the bandwidth resources that are now being used for embedding an SFC request. *Algorithm 6* in Chapter 5 provides the pseudocode for this function.

```
def Update_used_bandwidth(route, flow_i):
    global tenPercent_Bandwidth_Phy, ninetyPercent_Bandwidth_Phy
    for node_I in Nodes:
        for node_J in Nodes:
            nodeI_nodeJ_is_in_route = sum([1 if route[i]==node_I and
            route[i+1]==node_J else 0 for i in range(len(route)-1)])
            if nodeI_nodeJ_is_in_route:
                if flow_i in high_priority_matrix:
                    if tenPercent_Bandwidth_Phy[node_I][node_J] < Bandwidth_sfc[flow_i]:
                        ninetyPercent_Bandwidth_Phy[node_I][node_J] -= Bandwidth_sfc[flow_i]
                        - tenPercent_Bandwidth_Phy[node_I][node_J]
                        tenPercent_Bandwidth_Phy[node_I][node_J] = 0
                    else:
                        tenPercent_Bandwidth_Phy[node_I][node_J] -= Bandwidth_sfc[flow_i]
                else:
                    ninetyPercent_Bandwidth_Phy[node_I][node_J] -= Bandwidth_sfc[flow_i]
```

Code Listing 27. Update used bandwidth.

Similar to the previous function that updated the bandwidth resource, in Code Listing 28, we construct a function that will update the CPU resource that is currently being used for mapping an SFC request.

```
def Update_used_cpu(route, flow_i):
    global tenPercent_CPU_Phy, ninetyPercent_CPU_Phy
    for node_I in Nodes:
        for node_J in Nodes_MDC:
            nodeI_nodeJ_is_in_route = sum([1 if route[i]==node_I and route[i+1]==node_J
            else 0 for i in range(len(route)-1)])
            if nodeI_nodeJ_is_in_route:
                if flow_i in high_priority_matrix:
                    if tenPercent_CPU_Phy[node_J] < CPU_sfc[flow_i]:
                        ninetyPercent_CPU_Phy[node_J] -= CPU_sfc[flow_i]
                        - tenPercent_CPU_Phy[node_J]
                        tenPercent_CPU_Phy[node_J] = 0
                    else:
                        tenPercent_CPU_Phy[node_J] -= CPU_sfc[flow_i]
                else:
                    ninetyPercent_CPU_Phy[node_J] -= CPU_sfc[flow_i]
```

Code Listing 28. Update used CPU.

In the end, similar to the previous two functions, we build a function to update the memory resources that have been used for mapping an SFC request, which is shown as Code Listing 29.

```
def Update_used_disk(route, flow_i):
    global tenPercent_Disk_Phy, ninetyPercent_Disk_Phy
    for node_I in Nodes:
        for node_J in Nodes_MDC:
            nodeI_nodeJ_is_in_route = sum([1 if route[i]==node_I and route[i+1]==node_J
            else 0 for i in range(len(route)-1)])
            if nodeI_nodeJ_is_in_route:
                if flow_i in high_priority_matrix:
                    if tenPercent_Disk_Phy[node_J] < Disk_sfc[flow_i]:
                        ninetyPercent_Disk_Phy[node_J] -= Disk_sfc[flow_i]
                        - tenPercent_Disk_Phy[node_J]
                        tenPercent_Disk_Phy[node_J] = 0
                    else:
                        tenPercent_Disk_Phy[node_J] -= Disk_sfc[flow_i]
                else:
                    ninetyPercent_Disk_Phy[node_J] -= Disk_sfc[flow_i]
```

Code Listing 29. Update used memory.

To this end, using Code Listing 27, Code Listing 28, and Code Listing 29, we are able to update the physical resources used for embedding an SFC request.

In our code, the conversion from list to dictionary is handled by Code Listing 30.

```
def convert_list_to_dic(bandwidth_list):
    new_graph = dict()
    for i in range(len(bandwidth_list)):
        dic_tmp = dict()
        for j in range(len(bandwidth_list)):
            if bandwidth_list[i][j] > 0:
                dic_tmp[j]=bandwidth_list[i][j]
        if len(dic_tmp) > 0:
            new_graph[i] = dic_tmp
    return new_graph
```

Code Listing 30. Convert list to dictionary.

Following the implementation of *Algorithms 5* and *Algorithm 6* from Chapter 5, we build several functions to calculate the KPIs used in the performance evaluation chapter, Chapter 6.

As stated in Chapter 6, the first important KPI is the end-to-end delay. Code Listing 31 contains the definition of a function that we use to calculate the propagation delay of an SFC request.

```
def Calculate_Delay (route_selected_for_each_flow):
    E2EDelay = dict()
    for flow_i in range(len(route_selected_for_each_flow)):
        E2EDelay[flow_i] = 0
        for node_I in Nodes:
            for node_J in Nodes:
                nodeI_nodeJ_is_in_route = sum([1 if route_selected_for_each_flow[flow_i][i] ==
                node_I and route_selected_for_each_flow[flow_i][i + 1] == node_J else 0 for i
                in range(len(route_selected_for_each_flow[flow_i]) - 1)])
                if nodeI_nodeJ_is_in_route:
                    E2EDelay[flow_i] += Distance[node_I][node_J]
    return E2EDelay
```

Code Listing 31. Calculate delay.

The path length of an SFC request is the following KPI that we calculate.

Code Listing 32 defines a function that will be used to calculate the length of the SFC deployment route.

```
def Calculate_Plength (route_selected_for_each_flow):
    Plength = dict()
    for flow_i in range(len(route_selected_for_each_flow)):
        Plength[flow_i] = 0
        for node_I in Nodes:
            for node_J in Nodes:
                nodeI_nodeJ_is_in_route = sum([1 if route_selected_for_each_flow[flow_i][i] ==
                node_I and route_selected_for_each_flow[flow_i][i + 1] == node_J else 0 for i
                in range(len(route_selected_for_each_flow[flow_i]) - 1)])
                if nodeI_nodeJ_is_in_route:
                    Plength[flow_i] += Graph[node_I][node_J]
    return Plength
```

Code Listing 32. Calculate path length.

The next important KPI to calculate is bandwidth utilization. Calculating the amount of bandwidth used by each SFC request is the responsibility of the function that is written in Code Listing 33.

```
def Calculate_BWconsumption(route_selected_for_each_flow):
    BWconsumption=dict()
    for flow_i in range(len(route_selected_for_each_flow)):
        BWconsumption[flow_i]=Plength[flow_i] * Bandwidth_sfc[flow_i]
    return BWconsumption
```

Code Listing 33. Calculate bandwidth consumption.

Last but not least, one of the most important KPIs for evaluating the performance of a developed algorithm is its reliability. The definition of reliability is defined in Chapter 3. To this end, we determine the reliability of each SFC request by utilizing the function that is outlined in Code Listing 34. As indicated in Chapter 3, we take into account both hardware reliability, which is the reliability of *CDC* nodes, and software reliability, which is the reliability of VNF instances. In this regard, we have defined all the necessary functions to implement the *NORAAS* algorithm. In the following stage, we will define the core of the implementation in order to determine the optimal deployment path.

```
def Calculate_Reliability(route_selected_for_each_flow):
    flow_reliability=dict()
    for flow_i in range(len(route_selected_for_each_flow)):
        flow_reliability[flow_i]=0
        for node_I in Nodes:
            for node_J in Nodes:
                nodeI_nodeJ_is_in_route = sum([1 if route_selected_for_each_flow[flow_i][i] ==
                node_I and route_selected_for_each_flow[flow_i][i + 1] == node_J else 0 for i
                in range(len(route_selected_for_each_flow[flow_i]) - 1)])
                if nodeI_nodeJ_is_in_route:
                    flow_reliability[flow_i] *= Reliability_CDCs[node_I] * Reliability_CDCs[node_J]
    return flow_reliability
```

```
reliability_flow=1
for s in range (S):
    for v in range (len(Req_VNF_for_Flows[s])):
        reliability_flow*=Reliability_VNFs[(Req_VNF_for_Flows[s][v])]
    reliability_flow= flow_reliability [s] * reliability_flow
    print(reliability_flow)
    reliability_flow=1
```

Code Listing 34. Calculate reliability.

Code Listing 35 and Code Listing 36 serve as the main components of the *NORAAS algorithm* that determine the optimal deployment path for each SFC request. The *Algorithm 4* in Chapter 5 represents its pseudocode.

Using Code Listing 22 through Code Listing 36, we can achieve near-optimal *ORAAS* algorithm outcomes with minimal execution time and an optimality gap.

```
for VNF in Req_VNF_for_Flows[flow]:
    routes, lengths = dict(), dict()
    FreeBandwidth = Calculate_free_bandwidth(flow)
    FreeCPU= Calculate_free_cpu(flow)
    FreeDisk= Calculate_free_disk(flow)
    new_distance = [[Distance[i][j] if FreeBandwidth[i][j]>=Bandwidth_sfc[flow] else 0 for j
    in range(N)] for i in range(N)]
    new_graph = convert_list_to_dic(new_distance)
    new_P_Np_X_new = []
    for ele in P_Np_X_new[VNF]:
        if FreeCPU[ele] >= CPU_sfc[flow] and FreeDisk[ele]>=Disk_sfc[flow]:
            new_P_Np_X_new.append(ele)
    #for mdc in P_Np_X_new[VNF]:
    for mdc in new_P_Np_X_new:
        tmp_lenght, tmp_route = Dijkstra(new_graph, currentNode, mdc)
        if len(tmp_route) != 0 and Reliability_CDCs [mdc]* Reliability_VNFs [VNF]  > Reliability_SFCs[flow]:
            lengths[mdc], routes[mdc] = tmp_lenght, tmp_route
            new_graph = convert_list_to_dic(new_distance)
    if len(routes) == 0:
        print("Cannot route flow {0} -- no enough bandwidth to allocate.".format(flow+1))
        flow_is_routeable = False
        break
    # for i in len(new_P_Np_X_new):
    lengthss = copy.deepcopy(lengths)
    for cdc in new_P_Np_X_new:
        curNode = cdc
        Freewidth = Calculate_free_bandwidth(flow)
        n_distance = [[Distance[i][j] if Freewidth[i][j] >= Bandwidth_sfc[flow] else 0 for j
        in range(N)] for i in range(N)]
        n_graph = convert_list_to_dic(n_distance)
        leng, sel_route = Dijkstra(n_graph, curNode, Des[flow])
        lengthss[cdc] = lengthss[cdc] + leng


    next_CDC = min(lengthss,key=lengthss.get)
    tmp_next_CDC= min(lengthss,key=lengthss.get)

    selected_route = routes[next_CDC]
    #Move_to_(next_CDC, routes)
    currentNode = next_CDC
    route_selected_for_each_flow[flow].extend(selected_route)
    Update_used_bandwidth(selected_route, flow)
    Update_used_cpu(selected_route, flow)
    Update_used_disk(selected_route,flow)
```

Code Listing 35. Find optimal deployment path (first part).

```
if flow_is_routeable:
    if currentNode != Des[flow]:
        # move to destination
        FreeBandwidth = Calculate_free_bandwidth(flow)
        FreeCPU = Calculate_free_cpu(flow)
        new_distance = [[Distance[i][j] if FreeBandwidth[i][j]>=Bandwidth_sfc[flow] else 0 for j
        in range(N)] for i in range(N)]

        new_graph = convert_list_to_dic(new_distance)
        length, selected_route = Dijkstra(new_graph, currentNode, Des[flow])
        if len(selected_route) == 0:
            print("Cannot route flow {0} -- no enough bandwidth to allocate.".format(flow+1))
            continue
        route_selected_for_each_flow[flow].extend(selected_route)
        Update_used_bandwidth(selected_route, flow)
        Update_used_cpu(selected_route, flow)
        Update_used_disk(selected_route, flow)
    print(route_selected_for_each_flow[flow])
```

Code Listing 36. Find optimal deployment path (second part).

A.3. Implementation of the DAAS algorithm in Python

The *DAAS* algorithm is essentially a dynamic variant of the *NORAAS* algorithm. In a dynamic scenario, each SFC request has a lifetime during which it can access physical network resources (bandwidth, memory, and CPU); once this lifetime expires, the physical network resources can be made available for the next SFC embedding. In this section, we present the programming of our proposed dynamic SFC embedding method, *DAAS*. To this purpose, we employ the same strategy as the *NORAAS* algorithm, i.e., *flow prioritization* and a configurable *priority coefficient factor*, to reserve a quantity of physical network resources only for SFC requests with the highest priority in order to guarantee their QoS. In a dynamic scenario, time-cycles are specified, which a certain number of SFC requests entering and exiting the network in each time-cycle. Following is the Python implementation of the necessary functions required to implement the *DAAS* algorithm. In a dynamic context, Code Listings 37 through 55 are responsible for

supplying the optimal deployment path for SFC requests. The next section will provide further clarification.

First, the physical network resources are divided into high-priority SFC requests and low-priority SFC requests so that a part may be reserved for high-priority SFC requests. Physical network resource reservation (bandwidth, memory, and CPU) for high-priority SFC requests is applied using Code Listing 37.

```
ninetyPercent_Bandwidth_Phy = [[0.9 * element for element in row] for row in Bandwidth_Phy]
tenPercent_Bandwidth_Phy = [[0.1 * element for element in row] for row in Bandwidth_Phy]
ninetyPercent_CPU_Phy = [0.9 * element for element in CPU_Phy]
tenPercent_CPU_Phy = [0.1 * element for element in CPU_Phy]
ninetyPercent_Disk_Phy = [0.9 * element for element in Disk_Phy]
tenPercent_Disk_Phy = [0.1 * element for element in Disk_Phy]
```

Code Listing 37. Physical resource reservation.

After applying physical network resource reservation, we use Code Listing 38 to set arrays to get used resources at different times.

```
l_Used_bandwidth_Phy=np.zeros([10,19,19], dtype = float)
l_Used_bandwidth_Phy.tolist()
h_Used_bandwidth_Phy=np.zeros([10,19,19], dtype = float)
h_Used_bandwidth_Phy.tolist()
l_Used_CPU_Phy=np.zeros([10,19], dtype = float)
l_Used_CPU_Phy.tolist()
h_Used_CPU_Phy=np.zeros([10,19], dtype = float)
h_Used_CPU_Phy.tolist()
l_Used_Disk_Phy=np.zeros([10,19], dtype = float)
l_Used_Disk_Phy.tolist()
h_Used_Disk_Phy=np.zeros([10,19], dtype = float)
h_Used_Disk_Phy.tolist()
```

Code Listing 38. Set array to get used resources in different times.

As described in Chapter 2 regarding the various algorithms for determining the shortest path between two nodes, we utilize the Dijkstra algorithm to get the optimal global shortest path in a weighted graph. Similarly to the *NORAAS* algorithm, the Dijkstra algorithm is used to determine the shortest route between

two nodes. Code Listing 39 illustrates the Dijkstra method and its necessary parameters for Python implementation.

```
def Dijkstra(Graph, start, goal):
    shortest_distance = {}
    predecessor = {}
    unseenNodes = Graph
    infinity = 999999
    path = []

    for node in unseenNodes:
        shortest_distance[node] = infinity
    shortest_distance[start] = 0

    while unseenNodes:
        minNode = None
        for node in unseenNodes:
            if minNode is None:
                minNode = node
            elif shortest_distance[node] < shortest_distance[minNode]:
                minNode = node

        for childNode, weight in Graph[minNode].items():
            if weight + shortest_distance[minNode] < shortest_distance[childNode]:
                shortest_distance[childNode] = weight + shortest_distance[minNode]
                predecessor[childNode] = minNode
        unseenNodes.pop(minNode)

    currentNode = goal
    while currentNode != start:
        try:
            path.insert(0, currentNode)
            currentNode = predecessor[currentNode]
        except KeyError:
            return nan, []
    path.insert(0, start)

    return shortest_distance[goal], path
    if shortest_distance[goal] != infinity:
        print('Shortest distance is ' + str(shortest_distance[goal]))
        print('And the path is ' + str(path))
```

Code Listing 39. Dijkstra algorithm.

Next, we must determine the available resources (bandwidth, memory, and CPU). Using a function demonstrated in Code Listing 40, free bandwidth resources are computed and released. The information about this function is explained in greater detail in Chapter 5.

```
def Calculate_free_bandwidth(flow_i,t):
    if t<2:
        FreeBandwidth = ninetyPercent_Bandwidth_Phy[:][:]
        # if flow_i has high priority:
        if flow_i in high_priority_matrix:
            # FreeBandwidth = FreeBandwidth + tenPercent_Bandwidth_Phy
            FreeBandwidth = [[FreeBandwidth[i][j] + tenPercent_Bandwidth_Phy[i][j] for j in range(N)] for i in range(N)]
    else:
        FreeBandwidth = ninetyPercent_Bandwidth_Phy[:][:]+l_Used_bandwidth_Phy[t-1][:][:]
        # if flow_i has high priority:
        if flow_i in high_priority_matrix:
            # FreeBandwidth = FreeBandwidth + tenPercent_Bandwidth_Phy
            FreeBandwidth = [[FreeBandwidth[i][j] +h_Used_bandwidth_Phy[t-1][i][j]+ tenPercent_Bandwidth_Phy[i][j]
            for j in range(N)] for i in range(N)]
    return FreeBandwidth
```

Code Listing 40. Calculate and release free bandwidth resource.

The reasoning behind CPU resources follows the same logic as that behind bandwidth resources. Utilizing a function illustrated in Code Listing 41, available CPU is calculated and released.

```
def Calculate_free_cpu(flow_i,t):
    if t<2:
        FreeCPU = ninetyPercent_CPU_Phy[:]
        # if flow_i has high priority:
        if flow_i in high_priority_matrix:
            # FreeCPU = FreeCPU + tenPercent_CPU_Phy
            FreeCPU = [FreeCPU[i] + tenPercent_CPU_Phy[i] for i in range(N)]
    else:
        FreeCPU = ninetyPercent_CPU_Phy[:]+l_Used_CPU_Phy[t-1][:]
        if flow_i in high_priority_matrix:

            FreeCPU = [FreeCPU[i] + h_Used_CPU_Phy[t-1][i]+ tenPercent_CPU_Phy[i] for i in range(N)]
    return FreeCPU
```

Code Listing 41. Calculate and release free CPU.

Last but not least, the calculation and release of memory resources. Similar to CPU resources and bandwidth resources, Code Listing 42 demonstrates a function to calculate and release free memory resources in Python. The information about this function is explained in greater detail in Chapter 5.

After establishing the necessary functions in order to calculate and release physical network resources, the next necessary step is to present the functions that are needed in order to update the used physical network resources for mapping SFC requests based on the utilized resources. The following paragraph will offer much more elaboration.

```
def Calculate_free_disk(flow_i,t):
    if t<2:
        FreeDisk = ninetyPercent_Disk_Phy[:]
        # if flow_i has high priority:
        if flow_i in high_priority_matrix:
            # FreeDisk = FreeDisk + tenPercent_Disk_Phy
            FreeDisk = [FreeDisk[i] + tenPercent_Disk_Phy[i] for i in range(N)]
    else:
        FreeDisk = ninetyPercent_Disk_Phy[:]+l_Used_Disk_Phy[t-1][:]
        # if flow_i has high priority:
        if flow_i in high_priority_matrix:
            # FreeDisk = FreeDisk + tenPercent_Disk_Phy
            FreeDisk = [FreeDisk[i] +h_Used_Disk_Phy[t-1][i] + tenPercent_Disk_Phy[i] for i in range(N)]
    return FreeDisk
```

Code Listing 42. Calculate and release free memory.

When a mapping request for SFC is processed, we are required to bring the available physical network resources up to date. In order to accomplish this, Code Listing 43 is referred to in order to update the consumed bandwidth resources in order to map a SFC request.

```
def Update_used_bandwidth(route, flow_i):
    global tenPercent_Bandwidth_Phy, ninetyPercent_Bandwidth_Phy
    for node_I in Nodes:
        for node_J in Nodes:
            nodeI_nodeJ_is_in_route = sum([1 if route[i]==node_I and route[i+1]==node_J else 0 for i in range(len(route)-1)])
            if nodeI_nodeJ_is_in_route:
                if flow_i in high_priority_matrix:
                    if tenPercent_Bandwidth_Phy[node_I][node_J] < Bandwidth_sfc[flow_i]:
                        ninetyPercent_Bandwidth_Phy[node_I][node_J] -= Bandwidth_sfc[flow_i] - tenPercent_Bandwidth_Phy[node_I][node_J]
                        tenPercent_Bandwidth_Phy[node_I][node_J] = 0
                    else:
                        tenPercent_Bandwidth_Phy[node_I][node_J] -= Bandwidth_sfc[flow_i]
                else:
                    ninetyPercent_Bandwidth_Phy[node_I][node_J] -= Bandwidth_sfc[flow_i]
```

Code Listing 43. Update used bandwidth.

The same logic as for bandwidth resources is applied to CPU resources. When processing a mapping request for SFC, we must bring the available physical network resources up to date. Code Listing 44 is consulted to update the utilized CPU resources required to map an SFC request. More information regarding this function is further discussed in Chapter 5.

```
def Update_used_cpu(route, flow_i):
    global tenPercent_CPU_Phy, ninetyPercent_CPU_Phy
    for node_I in Nodes:
        for node_J in Nodes_MDC:
            nodeI_nodeJ_is_in_route = sum([1 if route[i]==node_I and route[i+1]==node_J else 0 for i in range(len(route)-1)])
            if nodeI_nodeJ_is_in_route:
                if flow_i in high_priority_matrix:
                    if tenPercent_CPU_Phy[node_J] < CPU_sfc[flow_i]:
                        ninetyPercent_CPU_Phy[node_J] -= CPU_sfc[flow_i] - tenPercent_CPU_Phy[node_J]
                        tenPercent_CPU_Phy[node_J] = 0
                    else:
                        tenPercent_CPU_Phy[node_J] -= CPU_sfc[flow_i]
                else:
                    ninetyPercent_CPU_Phy[node_J] -= CPU_sfc[flow_i]
```

Code Listing 44. Update used CPU.

Last but not least, Code Listing 45 demonstrates the function used to update the consumed memory for mapping an SFC request.

```
def Update_used_disk(route, flow_i):
    global tenPercent_Disk_Phy, ninetyPercent_Disk_Phy
    for node_I in Nodes:
        for node_J in Nodes_MDC:
            nodeI_nodeJ_is_in_route = sum([1 if route[i]==node_I and route[i+1]==node_J else 0 for i in range(len(route)-1)])
            if nodeI_nodeJ_is_in_route:
                if flow_i in high_priority_matrix:
                    if tenPercent_Disk_Phy[node_J] < Disk_sfc[flow_i]:
                        ninetyPercent_Disk_Phy[node_J] -= Disk_sfc[flow_i] - tenPercent_Disk_Phy[node_J]
                        tenPercent_Disk_Phy[node_J] = 0
                    else:
                        tenPercent_Disk_Phy[node_J] -= Disk_sfc[flow_i]
                else:
                    ninetyPercent_Disk_Phy[node_J] -= Disk_sfc[flow_i]
```

Code Listing 45. Update used memory.

Next, we require some functions in order to get used physical network resources at each time-cycle. Code Listing 46 illustrates the function used to get the utilized bandwidth for embedding an SFC request.

```
def GET_used_bandwidth(route,flow_i):
    global tenPercent_Bandwidth_Phy,h_Used_bandwidth_Phy,l_Used_bandwidth_Phy
    for node_I in Nodes:
        for node_J in Nodes:
            nodeI_nodeJ_is_in_route = sum([1 if route[i]==node_I and route[i+1]==node_J else 0 for i in range(len(route)-1)])
            if nodeI_nodeJ_is_in_route:
                if flow_i in high_priority_matrix:
                    if tenPercent_Bandwidth_Phy[node_I][node_J] < Bandwidth_sfc[flow_i]:
                        l_Used_bandwidth_Phy[T_l[flow_i]-1][node_I][node_J] += Bandwidth_sfc[flow_i] - tenPercent_Bandwidth_Phy[node_I][node_J]
                        h_Used_bandwidth_Phy[T_l[flow_i]-1][node_I][node_J] += tenPercent_Bandwidth_Phy[node_I][node_J]
                    else:
                        h_Used_bandwidth_Phy[T_l[flow_i]-1][node_I][node_J] += Bandwidth_sfc[flow_i]
                else:
                    l_Used_bandwidth_Phy[T_l[flow_i]-1][node_I][node_J] += Bandwidth_sfc[flow_i]
```

Code Listing 46. Get used bandwidth.

Similar to bandwidth resources, Code Listing 47 is used to get utilized CPU.

```
def GET_used_CPU(route,flow_i):
    global l_Used_CPU_Phy, h_Used_CPU_Phy,tenPercent_CPU_Phy
    for node_I in Nodes:
        for node_J in Nodes_MDC:
            nodeI_nodeJ_is_in_route = sum([1 if route[i]==node_I and route[i+1]==node_J else 0 for i in range(len(route)-1)])
            if nodeI_nodeJ_is_in_route:
                if flow_i in high_priority_matrix:
                    if tenPercent_CPU_Phy[node_J] < CPU_sfc[flow_i]:
                        l_Used_CPU_Phy[T_l[flow_i]-1][node_J] += CPU_sfc[flow_i] - tenPercent_CPU_Phy[node_J]
                        h_Used_CPU_Phy[T_l[flow_i]-1][node_J] += tenPercent_CPU_Phy[node_J]
                    else:
                        l_Used_CPU_Phy[T_l[flow_i]-1][node_J] += CPU_sfc[flow_i]
                else:
                    l_Used_CPU_Phy[T_l[flow_i]-1][node_J] += CPU_sfc[flow_i]
```

Code Listing 47. Get used CPU.

Finally, Code Listing 48 presents the function used to get the used memory for mapping a SFC request.

```
def GET_used_disk(route, flow_i):
    global tenPercent_Disk_Phy, ninetyPercent_Disk_Phy,l_Used_Disk_Phy, h_Used_Disk_Phy
    for node_I in Nodes:
        for node_J in Nodes_MDC:
            nodeI_nodeJ_is_in_route = sum([1 if route[i]==node_I and route[i+1]==node_J else 0 for i in range(len(route)-1)])
            if nodeI_nodeJ_is_in_route:
                if flow_i in high_priority_matrix:
                    if tenPercent_Disk_Phy[node_J] < Disk_sfc[flow_i]:
                        l_Used_Disk_Phy[T_l[flow_i]-1][node_J] += Disk_sfc[flow_i] - tenPercent_Disk_Phy[node_J]
                        h_Used_Disk_Phy[T_l[flow_i]-1][node_J] += tenPercent_Disk_Phy[node_J]
                    else:
                        h_Used_Disk_Phy[T_l[flow_i]-1][node_J] += Disk_sfc[flow_i]
                else:
                    l_Used_Disk_Phy[T_l[flow_i]-1][node_J] += Disk_sfc[flow_i]
```

Code Listing 48. Get used memory.

We use Code Listing 49 to convert a list to a dictionary.

```
def convert_list_to_dic(bandwidth_list):
    new_graph = dict()
    for i in range(len(bandwidth_list)):
        dic_tmp = dict()
        for j in range(len(bandwidth_list)):
            if bandwidth_list[i][j] > 0:
                dic_tmp[j]=bandwidth_list[i][j]
        if len(dic_tmp) > 0:
            new_graph[i] = dic_tmp
    return new_graph
```

Code Listing 49. Convert list to dictionary.

After describing the functions necessary to determine the optimal deployment path for SFC requests, we present the functions necessary to compute the assessed KPIs.

The first important KPI to assess is the path deployment end-to-end delay. The end-to-end delay is calculated as presented in Code Listing 50.

```
def Calculate_Delay (route_selected_for_each_flow):
    E2EDelay = dict()
    for flow_i in range(len(route_selected_for_each_flow)):
        E2EDelay[flow_i] = 0
        for node_I in Nodes:
            for node_J in Nodes:
                nodeI_nodeJ_is_in_route = sum([1 if route_selected_for_each_flow[flow_i][i] == node_I and
                route_selected_for_each_flow[flow_i][i + 1] == node_J else 0 for i in
                range(len(route_selected_for_each_flow[flow_i]) - 1)])
                if nodeI_nodeJ_is_in_route:
                    E2EDelay[flow_i] += Distance[node_I][node_J]
    return E2EDelay
```

Code Listing 50. Calculate delay.

The next KPI to analyze is path length. Code Listing 51 calculates the path length of each SFC request.

```
def Calculate_Plength (route_selected_for_each_flow):
    Plength = dict()
    for flow_i in range(len(route_selected_for_each_flow)):
        Plength[flow_i] = 0
        for node_I in Nodes:
            for node_J in Nodes:
                nodeI_nodeJ_is_in_route = sum([1 if route_selected_for_each_flow[flow_i][i] == node_I and
                route_selected_for_each_flow[flow_i][i + 1] == node_J else 0 for i in
                range(len(route_selected_for_each_flow[flow_i]) - 1)])
                if nodeI_nodeJ_is_in_route:
                    Plength[flow_i] += Graph[node_I][node_J]
    return Plength
```

Code Listing 51. Calculate path length.

Following that, bandwidth utilization is the next crucial KPI to examine. Code Listing 52 calculates the bandwidth consumption of each SFC request.

```
def Calculate_BWconsumption(route_selected_for_each_flow):
    BWconsumption=dict()
    for flow_i in range(len(route_selected_for_each_flow)):
        BWconsumption[flow_i]=Plength[flow_i] * Bandwidth_sfc[flow_i]
    return BWconsumption
```

Code Listing 52. Calculate bandwidth consumption.

Lastly, one of the most essential KPIs for measuring the success of an algorithm is its reliability. The definition of reliability may be found in Chapter 3. In order to verify the reliability of each SFC request, we use the code shown in Code Listing 53. As described in Chapter 3, we consider both hardware reliability, which is the reliability of *CDC* nodes, and software reliability, which is the reliability of VNF instances. In this sense, we have defined every function required to implement the *NORAAS* algorithm. In the subsequent stage, we will define the main body to obtain the optimal deployment path for each SFC request.

```
def Calculate_Reliability(route_selected_for_each_flow):
    flow_reliability=dict()
    for flow_i in range(len(route_selected_for_each_flow)):
        flow_reliability[flow_i]=1
        for node_I in Nodes:
            for node_J in Nodes:
                nodeI_nodeJ_is_in_route = sum([1 if route_selected_for_each_flow[flow_i][i] == node_I and
                route_selected_for_each_flow[flow_i][i + 1] == node_J else 0 for i in
                range(len(route_selected_for_each_flow[flow_i]) - 1)])
                if nodeI_nodeJ_is_in_route:
                    flow_reliability[flow_i] *= Reliability_CDCs[node_I] * Reliability_CDCs[node_J]
    return flow_reliability
```

```
reliability_flow=1
for s in range (S):
    for v in range (len(Req_VNF_for_Flows[s])):
        reliability_flow*=Reliability_VNFs[(Req_VNF_for_Flows[s][v])]
    reliability_flow= flow_reliability [s] * reliability_flow
    print(reliability_flow)
    reliability_flow=1
```

Code Listing 53. Calculate reliability.

In the last step, we will provide the Python code that is responsible for determining the most optimal deployment path for an SFC request. Code Listing 54 and Code Listing 55 define the main body of code that is responsible for determining the optimum deployment path. As stated before, it handles high-priority SFC requests differently to guarantee their QoS. Therefore, in a dynamic setting, we built the *DAAS* algorithm, which is a dynamic variation of the *NORAAS* algorithm, employing Code Listing 37 through Code Listing 55.

```
route_selected_for_each_flow = dict()
for t in range(1,10):
    F= list(range(len(Src[t-1]))) # list of flows
    for flow in F:
        if t==1:
            flow1=flow
        else:
            flow1=1+flow1

        flow_is_routeable = True
        print('flow {0} from {1} to {2}.  Reqruired VNFs: {3}. Required Bandwidth:{4}. Required CPU: {5}.Time come:{6}.Time leave:{7}'
        .format([flow1],Src[t-1][flow],Des[t-1][flow], Req_VNF_for_Flows[flow1], Bandwidth_sfc[flow1],CPU_sfc[flow1],T_c[flow1],T_l[flow1]))
        route_selected_for_each_flow[flow1] = []
        currentNode = Src[t-1][flow]

        #for VNF in requested_SFC
        for VNF in Req_VNF_for_Flows[flow1]:
            routes, lengths = dict(), dict()
            FreeBandwidth = Calculate_free_bandwidth(flow1,t)
            FreeCPU= Calculate_free_cpu(flow1,t)
            FreeDisk= Calculate_free_disk(flow1,t)

            new_distance = [[Distance[i][j] if FreeBandwidth[i][j]>=Bandwidth_sfc[flow1] else 0 for j in range(N)] for i in range(N)]
            new_graph = convert_list_to_dic(new_distance)
            new_P_Np_X_new = []
            for ele in P_Np_X_new[VNF]:
                if FreeCPU[ele] >= CPU_sfc[flow1] and FreeDisk[ele]>=Disk_sfc[flow1]:
                    new_P_Np_X_new.append(ele)
            #for mdc in P_Np_X_new[VNF]:
            for mdc in new_P_Np_X_new:
                tmp_lenght, tmp_route = Dijkstra(new_graph, currentNode, mdc)
                if len(tmp_route) != 0 and Reliability_CDCs [mdc]* Reliability_VNFs [VNF]  >= Reliability_SFCs[flow]:
                    lengths[mdc], routes[mdc] = tmp_lenght, tmp_route
                    new_graph = convert_list_to_dic(new_distance)
            if len(routes) == 0:
                print("Cannot route flow {0} -- no enough bandwidth to allocate.".format(flow1+1))
                flow_is_routeable = False
                break
            # for i in len(new_P_Np_X_new):
            lengthss = copy.deepcopy(lengths)
            for cdc in new_P_Np_X_new:
                curNode = cdc
                Freewidth = Calculate_free_bandwidth(flow1,t)
                n_distance = [[Distance[i][j] if Freewidth[i][j] >= Bandwidth_sfc[flow1] else 0 for j in range(N)] for i in range(N)]
                n_graph = convert_list_to_dic(n_distance)
                leng, sel_route = Dijkstra(n_graph, curNode, Des[t-1][flow])
                lengthss[cdc] = lengthss[cdc] + leng
```

Code Listing 54. Main code (part one).

```
            next_CDC = min(lengthss,key=lengthss.get)
            tmp_next_CDC= min(lengthss,key=lengthss.get)

            selected_route = routes[next_CDC]
            #Move_to_(next_CDC, routes)
            currentNode = next_CDC
            route_selected_for_each_flow[flow1].extend(selected_route)
            Update_used_bandwidth(selected_route, flow1)
            Update_used_cpu(selected_route, flow1)
            Update_used_disk(selected_route,flow1)
            GET_used_bandwidth(selected_route,flow1)
            GET_used_CPU(selected_route,flow1)
            GET_used_disk(selected_route,flow1)

        if flow_is_routeable:
            if currentNode != Des[t-1][flow]:
                # move to destination
                FreeBandwidth = Calculate_free_bandwidth(flow1,t)
                FreeCPU = Calculate_free_cpu(flow1,t)
                FreeDisk= Calculate_free_disk(flow1,t)
                new_distance = [[Distance[i][j] if FreeBandwidth[i][j]>=Bandwidth_sfc[flow1] else 0 for j in range(N)] for i in range(N)]

                new_graph = convert_list_to_dic(new_distance)
                length, selected_route = Dijkstra(new_graph, currentNode, Des[t-1][flow])
                if len(selected_route) == 0:
                    print("Cannot route flow {0} -- no enough bandwidth to allocate.".format(flow+1))
                    continue
                route_selected_for_each_flow[flow1].extend(selected_route)
                Update_used_bandwidth(selected_route, flow1)
                Update_used_cpu(selected_route, flow1)
                Update_used_disk(selected_route, flow1)
                GET_used_bandwidth(selected_route, flow1)
                GET_used_CPU(selected_route,flow1)
                GET_used_disk(selected_route,flow1)
            print('Selected path:', route_selected_for_each_flow[flow1])
```

Code Listing 55. Main code part two.

Bibliography

[1] E. T. S. I. (ETSI), "ETSI," [Online]. Available: https://www.etsi.org/technologies/nfv. [Accessed 07 07 2022].

[2] A. Perrin, "adapt IT," 04 January 2021. [Online]. Available: https://telecoms.adaptit.tech/blog/the-future-of-the-telecommunication-industry/. [Accessed 10 2022].

[3] "New Services & Applications with 5G Ultra-Reliable Low Latency Communicaitons," *5G Americas Whitepaper*, p. 60, November 2018.

[4] European Telecommunicatin Standards Institute, "ETSI," 2013. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV/001_099/001/01.01.01_60/gs_NFV001v010101p.pdf. [Accessed January 2020].

[5] B. Casey, "Slideplayer," [Online]. Available: https://slideplayer.com/slide/11886058/ . [Accessed 05 10 2022].

[6] J. Halpern and C. Pignataro, "IETF," [Online]. Available: https://datatracker.ietf.org/doc/rfc7665/.

[7] H. Umar Adoga and D. P. Pezaros, "Network Function Virtualization and Service Function Chaining Frameworks: A Comprehensive Review of Requirements, Objectives, Implementations, and Open Research Challenges," *MDPI,* 15 February 2022.

[8] H. U. Adoga and D. P. Pezaros, "Network Function Virtualization and Service Function Chaining Frameworks: A Comprehensive Review of Requirements, Objectives, Implementations, and Open Research Challenges.," *Future Internet,* 15 February 2022.

[9] AT&T, BT, CeturyLink, China Mobile, Colt, Deutsche Telekom, KDDI, NTT, Orange, Telecom Italia, Telecom Italia, Telefonica, Telstra, Verizon, "ETSI," [Online]. Available:

http://portal.etsi.org/NFV/NFV_White_Paper.pdf. [Accessed January 2020].

[10] N. Operators, "Portal ETSI," 2012. [Online]. Available: https://portal.etsi.org/NFV/NFV_White_Paper.pdf. [Accessed 05 10 2022].

[11] A. Fischer, J. F. Botero, M. T. Beck, H. de Meer and X. Hesselbach, "Virtual Network Embedding: A Survey.," *IEEE Communications Surveys and Tutorials.*, vol. 15, 2013.

[12] 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 and Tutorials*, vol. 18, 2016.

[13] "RedHat," RedHat, 16 August 2019. [Online]. Available: https://www.redhat.com/en/topics/virtualization/what-is-nfv. [Accessed 22 11 2022].

[14] "SDx Central Studios," [Online]. Available: https://www.sdxcentral.com/networking/nfv/definitions/whats-network-functions-virtualization-nfv/nfv-elements-overview/nfv-infrastructure-nfvi-definition/. [Accessed 10 10 2022].

[15] A. Leonhardt, "EQUINIX," [Online]. Available: https://blog.equinix.com/blog/2019/10/17/networking-for-nerds-defining-the-elements-of-nfv-architectures/. [Accessed 10 10 2022].

[16] Internet Engineering Task Force , "RFC7665," [Online]. Available: https://www.rfc-editor.org/info/rfc7665. [Accessed 09 2020].

[17] P. Z. Y. Z. Y. W. X. L. Y. J. Bin Zhang, "Co-Scaler: Cooperative scaling of software-defined NFV service functin chain.," *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks.*, 2016.

[18] K. Karamjeet, V. Mangat and K. Kumar, "A comprehensive survey of service function chain provisioning approaches in SDN and NFV architecture.," *Computer Science Review,* November 2020.

[19] D. Bhamare, R. Jain, M. Samaka and A. Erbad, "A Survey on Sevice Function Chaining," *Journal of Network and Computer Applications,* 2016.

[20] G. Mirjalily and Z. Luo, "Optimal Network Function Virtualization and Service Function Chaining: A Survey.," *Chinese Journal of Electronics,* vol. 27, 2018.

[21] "VMWARE," [Online]. Available: https://www.vmware.com/topics/glossary/content/network-functions-virtualization-nfv.html#:~:text=Network%20functions%20virtualization%20(NFV)%20is,as%20routing%20and%20load%20balancing.. [Accessed 20 11 2022].

[22] "IEEE Software Defined Networks," [Online]. Available: https://sdn.ieee.org/standardization. [Accessed 28 11 2022].

[23] S. Wang, H. Cao and L. Yang, "A Survey of Service Function Chains Orchestration in Data Center Networks," *EEE Globecom Workshops (GC Workshops),* 07-11 December 2020.

[24] H. Huang, W. Miao, G. Min, J. Tian and A. Alamri, "NFV and Blockchain Enabled 5G for Ultra-Reliable and Low-Latency Communications in Industry: Architecture and Performance Evaluation," *IEEE Transactions on Industrial Informatics,* pp. 5595-5604, 10 November 2021.

[25] D. Rico and P. Merino, "A Survey of End-to-End Solutions for Reliable Low-Latency Communications in 5G Networks," *IEEE Access,* January 2020.

[26] M. Osama, A.Ateya, Abdelhamied, S. Ahmed Elsaid and A. Muthanna, "Ultra-Reliable Low-Latency Communications: Unmanned Aerial Vehicles Assisted Systems," *Advances in Wireless Communications Systems (MDPI)*, 12 September 2022.

[27] M. M. Tajiki, S. Salsano, L. Chiaraviglio, M. Shojafar and B. Akbari, "Joint Energy Efficient and QoS-Aware Path Allocation and VNF Placement for Service Function Chaining," *IEEE Transactions on Network and Service Management*, vol. 16, 2019.

[28] G. Sun, Z. Xu, H. Yu, X. Chen, V. Chang and A. V. Vasilakos, "Low-Latency and Resource-Efficient Service Function Chaining Orchestration in Netwrok Function Virtualization," *IEEE Internet of Things Journal*, vol. 7, 2020.

[29] J. Pei, P. Hong, K. Xue and D. Li, "Efficiently Embedding Service Function Chains with Dynamic Virtual Network Function Placement in Geo-Distributed Cloud System.," *IEEE Transactions on parallel and distributed systems.*, vol. 30, 2019.

[30] G. Sun, R. Zhou, J. Sun, H. Yu and A. V. Vasilakos, "Energy-Efficient Provisioning for Service Function Chains to Support Delay-Sensitive Applications in Netwrok Function Virtualization.," *IEEE Internet of Things Journal*, vol. 7, 2020.

[31] M. Labonne, "Towarddatascience," 7 April 2022. [Online]. Available: https://towardsdatascience.com/integer-programming-vs-linear-programming-in-python-f1be5bb4e60e. [Accessed 22 November 2022].

[32] S. Sryheni, "Baeldung," 25 August 2021. [Online]. Available: https://www.baeldung.com/cs/graph-algorithms-bfs-dijkstra. [Accessed 15 January 2023].

[33] "Geeksforgeeks," 25 October 2022. [Online]. Available: https://www.geeksforgeeks.org/greedy-algorithms/. [Accessed 15 January 2023].

[34] H. A. Alameddine, C. Assi, M. H. K. Tushar and J. Y. Yu, "Low-Latency Service Schedule Orchestration in NFV-based Networks," in *IEEE Conference on Network Softwarization (NetSoft)*, Paris, France, 2019.

[35] D. Harutyunyan, N. Shahriar, R. Boutaba and R. Riggio, "Latency-Aware Service Function Chain Placement in 5G Mobile Networks," in *IEEE Conference on Network Softwarization (NetSoft)*, Paris, France, 2019.

[36] G. Sun, G. Zhu, D. Liao, H. Yu, X. Du and M. Guizani, "Cost-Efficient Service Function Chain Orchestraion for Low-Latency Application in NFV Networks.," *IEEE Systems Journal*, vol. 13, 2019.

[37] A. Hmaity, M. Savi, L. Askari, F. Musumeci, M. Tornatore and A. Pattavina , "Latency- and capacity-aware placement of chainded Virtual Network Functions in FMC metro networks," *Optical Switching and Networking*, vol. 35, 2020.

[38] Y. Li, L. Gao, S. Xu, Q. Qu, X. Yuan, F. Qi, S. Guo and X. Qiu, "Cost-and-QoS-Based NFV Sevice Function Chain Mapping Mechanism," in *IEEE Symposium on Network Operations and Management*, Budapest, 2020.

[39] E. Fountoulakis, Q. Liao and N. Pappas, "An End-to-End Performance Analysis for Service Chaining in a Virtualized Network.," *IEEE Open Journal of the Communication Society*, February 2020.

[40] X. Han, X. Meng, Z. Yu, Q. Kang and Y. Zhao, "A Service Function Chain Deployment Method Based on Network Flow Theory for Load Balance in Operator Networks.," *IEEE Access*, June 2020.

[41] M. Wang, B. Cheng, B. Li and J. Chen , "Service Function Chain Composition and Mapping in NFV-enabled Network," in *IEEE World Congress on Services*, 2019.

[42] C. Pham, N. H. Tran, S. Ren, W. Saad and C. S. Hong, "Traffic-Aware and Energy-Efficient vNF Placement for Service Chaining: Joint Sampling and Matching Approach," *IEEE Transactions on Services Computing,* vol. 13, January 2020.

[43] M. Caggiani Luizelli, W. Luis da Costa Cordeiro, L. S. Buriol and L. Paschoal Gaspary, "A fix- and -optimize approach for efficient and large scale virtual network function placement and chaining," *Computer Communications,* vol. 102, pp. 67-77, 2017.

[44] Y. Zhao, J. Shen, Q. Wang, S. Zheng, K. Xie, J. Gao and L. Feng, "Service Function Chain Deployment for 5G Delay-Sensitive Network Slicing," *International Wireless Communications and Mobile Computing (IWCMC),* p. 6, 2021 .

[45] M. A. Khoshkholghi, M. Gokan Khan, K. Alizadeh Noghani, J. Taheri, D. Bhamare, A. Kassler, Z. Xiang, S. Deng and X. Yang, "Service Function Chain Placement for Joint Cost and Latency Optimization," *Mobile Networks and Applications,* p. 2191–2205, 21 November 2020.

[46] L. Wang, M. Dolati and M. Ghaderi, "CHANGE: Delay-Aware Service Function Chain Orchestration at the Edge," *IEEE 5th International Conference on Fog and Edge Computing (ICFEC),* p. 10, 2021.

[47] J. Zhou, G. Feng and Y. Gao, "Network Function Parallelization for High Reliability and Low Latency Services," *IEEE Access,* 15 April 2020.

[48] X. Yin, B. Cheng, M. Wang and J. Chen, "Availability-aware Service Function Chain Placement in Mobile Edge Computing," in *IEEE World Congress on Services (SERVICES)*, 2020.

[49] Y. Wang, L. Zhang , P. Yu, K. Chen, X. Qiu, L. Meng, M. Kadoch and M. Cheriet, "Reliability-oriented and Resource-efficient Service Function Chain Constrution and Backup," *IEEE Transactions on Network and Service Management*, 2020.

[50] L. Qu, M. Khabbaz and C. Assi, "Reliability-Aware Service Chaining In Carrier-Grade Softwarized Networks," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, 2018.

[51] P. K. Thiruvasagam, V. J. Kotagi and C. S. R. Murthy, "A Reliability-Aware, Delay Guaranteed, and Resource Efficient Placement of Service Function Chains in Softwarized 5G Networks.," *IEEE Transactions on Cloud Computing*, 2020.

[52] P. Kaliyammal Thiruvasagam, A. Chakraborty, A. Mathew and C. S. Ram Murthy, "Reilable placement of service function chains and virtual monitoring functions with minimal cost in softwarized 5G networks," *IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT*, vol. 18, no. 2, 2021.

[53] S. Lin, W. Liang and J. Li, "Reliability-Aware Service Function Chain Provisioning in Mobile Edge-Cloud Networks," *29th International Conference on Computer Communications and Networks (ICCCN)*, p. 9, 03-06 August 2020.

[54] W. Chen, Z. Wang, H. Zhang, X. Yin and X. Shi, "Cost-Efficient Dynamic Service Function Chain Embedding in Edge Clouds," *17th International Conference on Network and Service Management (CNSM)*, 25-29 October 2021.

[55] S. Qin, M. Liu and G. Feng, "Dynamic Service Chaining for Ultra-reliable Services in Softwarized Networks," *IEEE Transactions on Network and Service Management*, 17 January 2023.

[56] X. Shang, Z. Liu and Y. Yang, "Online Service Funciton Chain Placement for Cost-effectiveness and Network Congestion Control," *IEEE Transactions on Computers*, p. 13, 2020.

[57] Z. Luo and C. Wu, "An Online Algorithm for VNF Service Chain Scaling in Datacenters," *IEEE/ACM Transaction on Networking*, p. 13, 2020.

[58] J. Liu, W. Lu, F. Zhou, P. Lu and Z. Zhu, "On Dynamic Service Function Chain Deployment and Readjustment," *IEEE Transactions on Network and Service Management*, pp. 543-553, 05 June 2017.

[59] B. Li, B. Cheng and J. Chen, "An Efficient Algorithm for Service Function Chains Reconfiguration in Mobile Edge Clouds Networks," *IEEE International Conference on Web Services (ICWS)*, 11 November 2021.

[60] T. u. o. Adelaide, "The Internet Topology Zoo," [Online]. Available: http://www.topology-zoo.org/explore.html . [Accessed 02 2021].

[61] Y. Wang, L. Zhang , P. Yu, K. Chen, X. Qiu, L. Meng, M. Kadoch and M. Cheriet, "Reliability-oriented and Resource-efficient Service Function Chain Constrution and Backup," *IEEE Transactions on Network and Service Management*, 2020.

[62] S. Choudhury, "The weight-constrained shortest path problem," *https://web.stanford.edu/~shushman/math15_report.pdf*, p. 5, 16 December 2015.

[63] M. M. Tajiki, S. Salsano, L. Chiaraviglio, M. Shojafar and B. Akbari, "Joint Energy Efficient and QoS-aware Path Allocation and VNF Placement for Service Funciton Chaining," IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT, July 2018. [Online]. Available: https://arxiv.org/pdf/1710.02611.pdf. [Accessed 10 2019].

[64] D. Zhai, X. Meng, Z. Yu and X. Han, "Reliability-Aware Service Function Chain Backup Protection Method," *IEEE Access*, p. 17, 7 January 2021.

[65] S. Zheng, Z. Ren, W. Cheng and H. Zhang, "Minimizing the Latency of Embedding Dependence-Aware SFCs into MEC Network via Graph Theory.," *IEEE Global Communications Conference,* p. 6, 2021.

[66] Y. Zhu, "Investigation of online service function chaining to support low latency network services in an NFV environment.," Master-Thesis, University of Duisburg-Essen, 2023.

[67] K. Moore, K. Khim and E. Ross, "Briliant," Briliant, [Online]. Available: https://brilliant.org/wiki/greedy-algorithm/. [Accessed 22 11 2022].

[68] "Geeksforgeeks," 18 November 2021. [Online]. Available: https://www.geeksforgeeks.org/markov-decision-process/. [Accessed 22 November 2022].

[69] The University of Adelaide, "The Internet Topology Zoo," [Online]. Available: http://www.topology-zoo.org/explore.html. [Accessed February 2019].

[70] J. Pei, P. Hong, K. Xue and D. Li, "Resource Aware Routing for Service Function Chains in SDN and NFV-Enabled," *IEEE Transactions on Service Computing*, 2018.

[71] M. Mohammadi Erbati and G. Schiele, "Application- and reliability-aware service function chaining to support low-latency applications in an NFV-enabled network," in *IEEE NFV-SDN*, Virtual Event, 2021.

[72] M. Mohammadi Erbati, M. M. Tajiki, F. Keshvari and G. Schiele, "Service function chaining to support ultra-low latency communication in NFV," *IEEE-International Conference on Broadband Communications for Next Generation Networks and Multimedia Applications (CoBCom),* p. 8, July 2022.

[73] M. Mohammadi Erbati and G. Schiele, "A novel reliable low-latency service function chaining to enable URLLC in NFV," *The 9th IEEE*

International Conference on Communications and Networking (IEEE ComNet'2022), p. 8, 1-4 November 2022.

[74] M. Mohammadi Erbati and G. Schiele, "A novel dynamic service function chaining to enable URLLC in NFV," *The 17th ConTEL – INTERNATIONAL CONFERENCE ON TELECOMMUNICATIONS,* p. 8, 11-13 July 2023.

[75] M. Mohammadi Erbati, M. M. Tajiki and G. Schiele, "Service function chaining to support ultra-low latency communication in NFV," *MDPI-electronics,* p. 26, 2023.

[76] F. Keshvari, "Investigation of the Reliability of Service Function Chains in an NFV-enabled Network.," Bachelor Thesis, University of Duisburg Essen, 2022.

[77] Z. Dong, M. Xiangru, Y. Zhenhua and H. Xiaoyang , "Reliability-Aware Service Function Chain Backup Protection Method.," *IEEE Access,* p. 17, 26 01 2021.

[78] Y. Qiu, J. Liang, V. C. M. Leung, X. Wu and X. Deng, "Online Reliability-Enhanced Virtual Network Services Provisioning in Fault-Prone Mobile Edge Cloud," *IEEE Transactions on Wireless Communications,* pp. 7299 - 7313, 15 March 2022.

[79] G. Liu, S. Huang and K. Li, "Reliability deployment of service function chain based on multi-agent reinforcement learning.," *IEEE 6th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC),* 03-05 October 2022.

[80] X. Chen, J. Zhou and S. Wei, "SFC-HO: Reliable Layered Service Function Chaining," *IEEE Access,* p. 17, 12 October 2022.

[81] M. Niu, Q. Han, B. Cheng, M. Wang, Z. Xu, W. Gu, S. Zhang and J. Chen, "HARS: A High-Available and Resource-Saving Service Function Chain

Placement Approach in Data Center Networks," *IEEE Transactions on Network and Service Management,* pp. 829 - 847, 21 January 2022.

[82] Y. Tibebu Woldeyohannes, B. Tola, Y. Jiang and K. K. Ramakrishnan, "CoShare: An Efficient Approach for Redundancy Allocation in NFV," *IEEE/ACM Transactions on Networking,* pp. 1014 - 1028, June 2022.

[83] G. Liu, S. Huang and K. Li, "Reliability deployment of service function chain based on multi-agent reinforcement learning," *2022 IEEE 6th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC),* 09 November 2022.

[84] T. J. Wassing, D. De Vleeschauwer and C. Papagianni, "A Machine Learning Approach for Service Funciton Chain Embedding in Cloud Datacenter Netwroks," *IEEE 10th International Conference on Cloud Networking (CloudNet),* p. 7, 2021.

[85] L. Wang, W. Mao, J. Zhao and Y. Xu, "DDQP: A Double Deep Q-Learning Approach to Online Fault-Tolerant SFC Placement," *IEEE Transactions on Network and Service Management,* p. 15, 03 2021.

[86] T. Wang, Q. Fan, X. Li, X. Zhang, Q. Xiong, S. Fu and M. Gao, "DRL-SFCP: Adaptive Service Function Chains Placement with Deep Reinforcement Learning.," *IEEE International Conference on Communication,* 2021.

[87] J. Jia, L. Yang and J. Cao, "Reliability-aware Dynamic Sercie Chains Scheduling in 5G Networks based on Reinforcement Learning.," *IEEE INFOCOM- IEEE Conference on Computer Communications.,* p. 10, 2021.

[88] J. Fernando Cevallos Moreno, R. Sattler, R. P. Caulier Cisterna, L. Ricciardi Celsi, A. Sánchez Rodríguez and M. Mecella, "Online Service Function Chain Deployment for Live-Streaming in Virtualized Content

Delivery Networks: A Deep Reinforcement Learning Approach," *Future Internet (MDPI)*, p. 28, 29 October 2021.

[89] H. Qu, K. Wang and J. Zhao, "Reliable Service Function Chain Deployment Method Based on Deep Reinforcement Learning," *Sensor (MDPI)*, 21 April 2021.

[90] J. Bai, X. Chang, F. Machida, L. Jiang, Z. Han and K. Trivedi, "Impact of Service Function Aging on the Dependability fopr MEC Service Function Chain," *IEEE Transactins on Dependable and Secure Computing*, p. 14, 2022.

[91] T. V. Doan, G. T. Nguyen, M. Reisslein and F. H. P. Fitzek, "SAP: Subchain-Aware NFV Service Placement in Mobile Edge Cloud," *IEEE Transactions on Network and Service Management*, p. 22, 2022.

[92] T. Subramanya, D. Harutyunyan and R. Riggio, "Machine Learning-Driven Service Function Chain Placement and Scaling in MEC-enabled 5G Networks," https://www.robertoriggio.net/papers/compnets2020.pdf , 2022.

[93] M. A. Khoshkholghi and T. Mahmoodi, "Edge intelligence for service function chain deployment in NFV-enabled networks.," *Computer Netwroks*, 24 December 2022.

[94] H. Chen, S. Wang, G. Li, L. Nie, X. Wang and Z. Ning, "Distributed Orchestration of Service Function Chains for Edge Intelligence in the Industrial Internet of Things," *IEEE Transactions on Industrial Informatics* , pp. 6244-6254, September 2022.

[95] A. Abouaomar, S. Cherkaoui, Z. Mlika and A. Kobbane, "Service Function Chaining in MEC: A Mean-Field Game and Reinforcement Learning Approach," *IEEE Systems Journal*, pp. 5357-5368, 30 May 2022.

[96] X. Shang, Z. Liu and Y. Yang, "Online Service Function Chain Placement for Cost-Effectiveness and Network Congestion Control.," *IEEE Transactions on Computers.*, p. 13, 01 2022.

[97] Y. Qiu, J. Liang, V. C. M. Leung, X. Wu and X. Deng, "Online Reliability-Enhanced Virtual Network Service Provisioning in Fault-Prone Mobile Edge Cloud.," *IEEE Transactions on Wireless Communicaitons* , p. 15, 09 2022.

[98] X. Wei, Y. Sheng, L. Li and C. Zhou , "DRL-Deploy: Adaptive Service Function Chains Deployment with Deep Reinforcement Learning.," *IEEE Intl Conf on Parallel and Distributed Processing with Applicaitons, Big Data and Cloud Computing, Sustainable Computing and Communications, Social Computing and Networking (ISPA/BDCloud/SocialCom/SustainCom)*, p. 8, 2021.

[99] H. Yu, Z. Chen, G. Sun, X. Du and M. Guizani, "Profit Maximization of Online Service Function Chain Orchestration in an Inter-Datacenter Elastic Optical Network.," *IEEE Transactions on Network and Service Management*, p. 13, 01 03 2021.

[100] J. Fernando Cevallos Moreno, R. Sattler, R. P. Caulier Cisterna, L. Ricciardi Celsi, A. Sánchez Rodríguez and . M. Mecella, "Online Service Function Chain Deployment for Live-Streaming in Virtualized Content Delivery Networks: A Deep Reinforcement Learning Approach," *Future Internet (MDPI)*, p. 28, 29 October 2021.

[101] S. Qin, M. Liu and G. Feng, "Dynamic Service Chaining for Ultra-reliable Services in Softwarized Networks," *IEEE Transactions on Network and Service Management*, 17 January 2023.

[102] X. Shang, Z. Liu and Y. Yang, "Online Service Function Chain Placement for Cost-Effectiveness and Network Congestion Control," *IEEE Transactions on Computers* , pp. 27-39, Juanuary 2022.

DOI: 10.17185/duepublico/81494
URN: urn:nbn:de:hbz:465-20240206-083710-7