Conceptualization and Implementation of 5G Use-cases in a Virtual Network

Undergraduate Thesis

Submitted in partial fulfillment of the requirements of
BITS F421T Thesis

By

Ashwin Sathish Kumar
ID No. 2020B4A80893P

Under the supervision of:

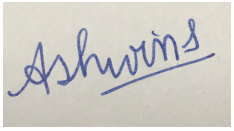
Prof. Dr.-Ing Axel Sikora
&
Prof. Dr. Sandeep Joshi

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS
December 2024

Declaration of Authorship

I, Ashwin Sathish Kumar, declare that this Undergraduate Thesis titled, 'Conceptualization and Implementation of 5G Use-cases in a Virtual Network' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date : 5th December, 2024

Certificate

This is to certify that the thesis entitled, "*Conceptualization and Implementation of 5G Use-cases in a Virtual Network*" and submitted by Ashwin Sathish Kumar ID No. 2020B4A80893P in partial fulfillment of the requirements of BITS F421T Thesis embodies the work done by him under my supervision.

Supervisor
Prof. Dr.-Ing Axel Sikora
Professor,
Hochschule Offenburg
Date:

Co-Supervisor
Prof. Dr. Sandeep Joshi
Asst. Professor,
BITS-Pilani Pilani Campus
Date:

"The real voyage of discovery consists not in seeking new landscapes, but in having new eyes."

Marcel Proust

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS

Abstract

Bachelor of Engineering (Hons.)

Conceptualization and Implementation of 5G Use-cases in a Virtual Network

by Ashwin Sathish Kumar

The rapid evolution of 5G networks has introduced novel challenges in managing diverse use cases with varying requirements. This thesis presents the development of a comprehensive virtualized 5G network testbed that leverages containerization and two open-source implementations: Open5GS for the 5G core network and UERANSIM for the radio access network and user equipment emulation. The testbed architecture emphasizes scalability, flexibility, and isolation of network components through the use of Docker containers and a modular design approach. A significant contribution of this work is the development of an API-driven configuration management framework that abstracts the complexities of the underlying network infrastructure. The proposed API enables programmatic control over major network parameters, simplifying the process of customizing the testbed environment for specific use cases. The effectiveness of the developed API framework is demonstrated through the implementation and evaluation of three representative 5G use cases: enhanced Mobile Broadband (eMBB), Ultra-Reliable Low-Latency Communications (URLLC), and massive Machine-Type Communications (mMTC). The eMBB use case showcases high-bandwidth video streaming, the URLLC scenario simulates a remote surgery application, and the mMTC case study demonstrates the management of a large-scale sensor network with custom traffic generation patterns. Detailed performance analyses are conducted for each use case, validating the testbed's ability to support diverse 5G applications and maintain the required quality of service. The insights gained from this work contribute to a better understanding of the practical considerations for deploying virtualized 5G networks and highlight the potential for further development in this domain.

Acknowledgements

Firstly, I extend my sincere thanks to Prof. Axel Sikora, my supervisor and mentor, whose guidance and expertise have been invaluable throughout this journey. His unwavering support, patience, and insightful feedback have not only significantly shaped my research but also enhanced my understanding of the subject matter. Prof. Sikora's ability to challenge me intellectually while providing a nurturing environment allowed me to explore innovative ideas and approach problems critically. His mentorship has truly inspired me and instilled in me a deeper passion for 5G and beyond.

I am also deeply grateful to Prof. Sandeep Joshi, my on-campus supervisor, for providing me with the opportunity to undertake this project in Germany. His encouragement and belief in my capabilities have been instrumental in my academic growth. Prof. Joshi's guidance in the early stages of my research laid a strong foundation, and his continued support throughout my thesis has been a source of motivation.

My appreciation goes to Fabian and Jubin, research associates at Offenburg Univeristy, with whom I interacted daily. Their direction and insights have been crucial in navigating the complexities of this project. Fabian's expertise in the operational aspects of my work helped clarify numerous challenges, while Jubin's fresh perspectives often sparked innovative solutions. Their collaboration not only enriched my experience but also fostered a sense of teamwork that made the project enjoyable and rewarding. I would also like to thank Seyedali for his valuable assistance in helping me understand the inital setup of Docker and Open5GS. His expertise and readiness to help with any technical challenges greatly facilitated my progress. A special mention goes to Charan, a friend and a fellow associate, who not only provided a supportive presence but also contributed to a positive work environment.

Lastly, I owe a profound debt of gratitude to my family, whose constant encouragement and belief in my potential have been a source of motivation throughout my academic journey. Their trust in my decisions, no matter how unconventional, has allowed me to pursue my passions with confidence ...

Contents

List of Figures

List of Tables

Abbreviations

3GPP	**3**rd **G**eneration **P**artnership **P**roject
5G	**F**ifth-**G**eneration
5QI	**5**G **Q**oS **I**dentifier
AMBR	**A**ggregate **M**aximum **B**it **R**ate
AMF	**A**ccess and **M**obility Management **F**unction
API	**A**pplication **P**rogramming **I**nterface
ARP	**A**llocation and **R**etention **P**riority
AUSF	**A**uthentication **S**erver **F**unction
AVI	**A**udio **V**ideo **I**nterleave
BSF	**B**inding **S**upport **F**unction
CNFs	**C**loud-**N**ative **N**etwork **F**unctions
CPU	**C**entral **P**rocessing **U**nit
CU	**C**entral **U**nit
CUPS	**C**ontrol and **U**ser **P**lane **S**eparation
DCI	**D**igital **C**inema **I**nitiatives
DU	**D**istributed **U**nit
eMBB	**e**nhanced **M**obile **B**road**B**and
EPC	**E**volved **P**acket **C**ore
GBR	**G**uaranteed **B**it **R**ate
gNB	**g**NodeB (5G base station)
GTP-U	**G**PRS **T**unnelling **P**rotocol for **U**ser Plane

HTTP	**H**yper**t**ext **T**ransfer **P**rotocol
ICMP	**I**nternet **C**ontrol **M**essage **P**rotocol
IoT	**I**nternet **o**f **T**hings
IP	**I**nternet **P**rotocol
JSON	**J**ava**S**cript **O**bject **N**otation
KPI	**K**ey **P**erformance **I**ndicator
LTE	**L**ong-**T**erm **E**volution
MAC	**M**edia **A**ccess **C**ontrol
MBR	**M**aximum **B**it **R**ate
MEC	**M**obile **E**dge **C**omputing
MIME	**M**ultipurpose **I**nternet **M**ail **E**xtensions
MKV	**M**atro**s**ka **V**ideo
MME	**M**obility **M**anagement **E**ntity
mMTC	**m**assive **M**achine-**T**ype **C**ommunications
MOV	**MOV**ie File Format
MP4	**M**oving **P**icture **E**xperts **G**roup 4 (MPEG-4) Part 14
NAS	**N**on-**A**ccess **S**tratum
NF	**N**etwork **F**unction
NFV	**N**etwork **F**unction **V**irtualization
NGAP	**N**ext **G**eneration **A**pplication **P**rotocol
NRF	**N**etwork **R**epository **F**unction
NSA	**N**on-**S**tand**a**lone
NSSF	**N**etwork **S**lice **S**election **F**unction
ONOS	**O**pen **N**etwork **O**perating **S**ystem
OSM	**O**pen **S**ource NFV **M**ANO (**MAN**agement and **O**rchestration)
PCC	**P**olicy and **C**harging **C**ontrol
PCI	**P**eripheral **C**omponent **I**nterconnect
PCF	**P**olicy **C**ontrol **F**unction

PDCP **P**acket **D**ata **C**onvergence **P**rotocol

PDN **P**acket **D**ata **N**etwork

PDU **P**rotocol **D**ata **U**nit

PGW-U **P**DN **G**ateway - **U**ser plane

PHY **PHY**sical Layer

PLMN **P**ublic **L**and **M**obile **N**etwork

RAN **R**adio **A**ccess **N**etwork

REST **RE**presentational **S**tate **T**ransfer

RLC **R**adio **L**ink **C**ontrol

RRC **R**adio **R**esource **C**ontrol

SBA **S**ervice-**B**ased **A**rchitecture

SCP **S**ervice **C**ommunication **P**roxy

SCTP **S**tream **C**ontrol **T**ransmission **P**rotocol

SD **S**lice **D**ifferentiator

SDAP **S**ervice **D**ata **A**daptation **P**rotocol

SDN **S**oftware-**D**efined **N**etworking

SDR **S**oftware-**d**efined **r**adio

SEPP **S**ecurity **E**dge **P**rotection **P**roxy

SFC **S**ervice **F**unction **C**haining

SGW-U **S**erving **G**ateway - **U**ser plane

SMF **S**ession **M**anagement **F**unction

S-NSSAI **S**ingle **N**etwork **S**lice **S**election **A**ssistance **I**nformation

SST **S**lice **S**ervice **T**ype

TSN **T**ime-**S**ensitive **N**etworking

TUN **T**unnel

UDP **U**ser **D**atagram **P**rotocol

UE **U**ser **E**quipment

UDM **U**nified **D**ata **M**anagement

UDR	**U**nified **D**ata **R**epository
UHD	**U**ltra **H**igh **D**efinition
UPF	**U**ser **P**lane **F**unction
URI	**U**niform **R**esource **I**dentifier
URLLC	**U**ltra-**R**eliable **L**ow-**L**atency **C**ommunications
VGA	**V**ideo **G**raphics **A**rray
VM	**V**irtual **M**achine
VNF	**V**irtual **N**etwork **F**unction
VVC	**V**ersatile **V**ideo **C**oding
XR	E**X**tended **R**eality

To my grandfather, Mr. Chandran, who stands tall in the realm of creativity, regardless of traditional engineering paths

Chapter 1

Introduction

1.1 Foundations and Tools

The introduction of fifth-generation (5G) cellular networks changed the dynamics of telecommunications technology in a manner that makes it possible to improve industries and change the digital landscapes that we have grown accustomed to. Unlike previous versions of cellular technology, 5G is not an upgrade in mobile communication systems; rather, it is a technology re-imagined to meet the requirements of many different use cases and applications. The key pillars of 5G - enhanced Mobile Broadband (eMBB), Ultra-Reliable Low-Latency Communications (URLLC), and massive Machine-Type Communications (mMTC) - enable a new era of connectivity, from gigabit-speed mobile internet to mission-critical applications and the Internet of Things (IoT) at an unprecedented scale [1]. The 5G network architecture, as defined by the 3rd Generation Partnership Project (3GPP), uses concepts of network slicing and network function virtualization (NFV) [2]. These technologies mark a paradigm shift in the design, deployment and management of mobile networks. Further, the diverse requirements of 5G use cases necessitate flexible and adaptable network configurations that can be tailored to specific applications.

The implementation of such diverse use cases in 5G networks presents significant challenges in terms of testing and validation. Physical deployment of 5G infrastructure for testing purposes is often impractical due to high costs and complex hardware requirements. This necessitates the development of virtualized testing environments that can accurately simulate 5G network behaviour. Virtual network implementations offer significant advantages, allowing researchers and developers to validate network configurations and applications without the constraints of physical infrastructure. In this context, open-source implementations of core network functions and radio access networks have emerged as valuable tools for research and development. These implementations provide modular platforms for testing network configurations and developing applications in controlled environments. The ability to simulate both core network functions

and radio access behaviour enables comprehensive testing of end-to-end scenarios that would be challenging to replicate with physical equipment. The effectiveness of these virtual implementations is further enhanced through containerization technologies, which enable flexible deployment and management of network functions. Containerization allows network functions to operate as isolated software components, facilitating easy configuration changes and rapid testing of different network scenarios [3]. This approach is particularly valuable for developing and validating specific use cases that demonstrate the capabilities of 5G networks.

This chapter examines the fundamental technical concepts of 5G networks - from the service-based architecture that forms the foundation of 5G core networks, to a detailed discussion of protocol stacks that enable communication between different network elements.

1.2 5G System Overview

The 5G network architecture, as specified by the 3rd Generation Partnership Project (3GPP), represents a significant evolution from previous generations of mobile networks. It is designed to support a wide range of services and use cases, from enhanced mobile broadband to mission-critical applications and massive Internet of Things (IoT) deployments. This section outlines the key components and concepts of the 3GPP-specified 5G architecture.

1.2.1 Service-Based Architecture

The 5G core network adopts a Service-Based Architecture (SBA) (as depicted in Figure 1.1 which is a fundamental shift from the traditional point-to-point architecture of earlier generations. In SBA, network functions are implemented as a set of interconnected Network Function (NF) services [4]. This approach enables greater flexibility, scalability, and modularity in network design and deployment. The 3GPP specifications define several key network functions that form the backbone of the 5G core network:

1. **Access and Mobility Management Function (AMF):** Manages UE connections and mobility. It handles registration, provides access authorization, and manages UE reachability. The AMF also facilitates communication between UE and other network functions. [6].

2. **Session Management Function (SMF):** Manages Protocol Data Unit (PDU) Sessions, which are the data connections of UEs. It applies policy and charging rules, and interacts with the User Plane Function to handle user data traffic [6].

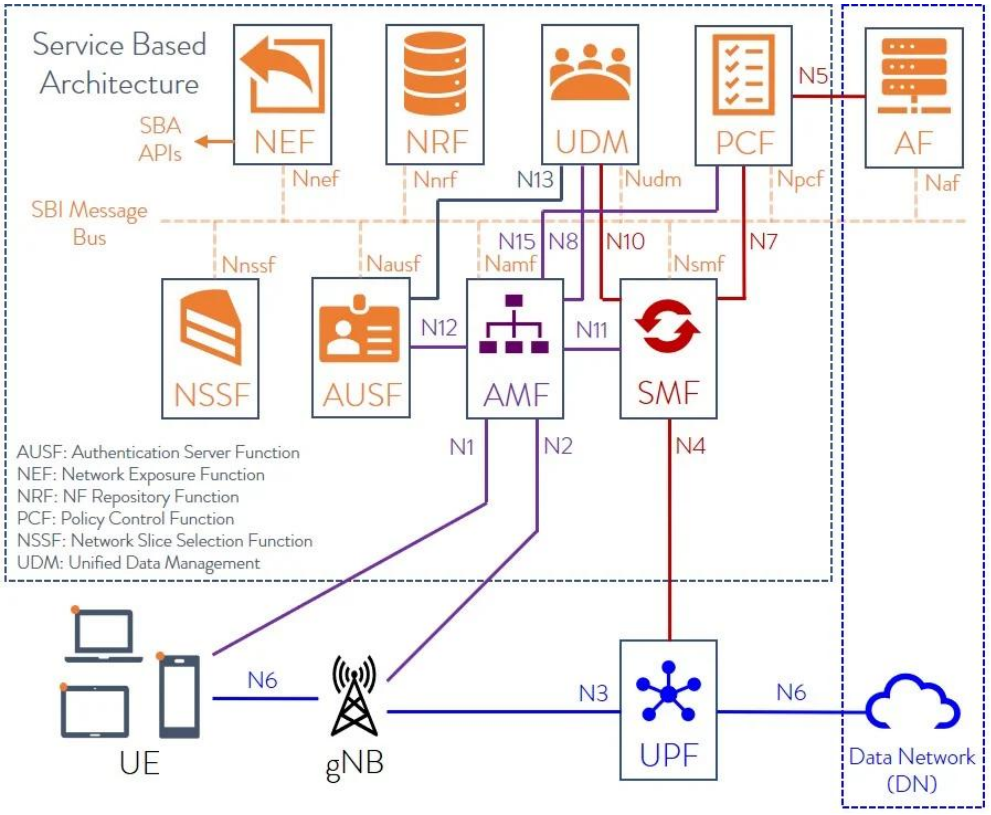


FIGURE 1.1: The 5G Service-based Architecture [5]

3. **User Plane Function (UPF):** Handles user data, applies Quality of Service (QoS) rules and manages traffic routing. It serves as the anchor for mobility across different networks. Open5GS uses GTP-U tunnelling and can inspect traffic through the UPF [7].

4. **Authentication Server Function (AUSF):** Responsible for authenticating UEs. It supports both the 5G AKA and EAP-AKA' methods of authentication. In Open5GS, the AUSF works with the UDM to get authentication data and interacts with the AMF to verify UEs [8].

5. **Unified Data Management (UDM):** Stores subscriber information and manages user profiles. It helps other functions by providing subscription details and supports converting SUCI to SUPI. Open5GS connects the UDM with other functions as outlined in 3GPP standards [9, 10].

6. **Policy and Charging Function (PCF):** Specifies the rules for managing network services and informs the AMF and SMF on how to control access and manage sessions based on policies. The PCF handles various policy aspects, including QoS parameters, Allocation and Retention Priority (ARP), and pre-emption capabilities. It also manages Slice Selection Type (SST) policies for network slicing and defines Aggregate Maximum Bit Rate (AMBR)

for subscribers. These policy controls enable fine-grained management of network resources and service differentiation across diverse use cases [11].

7. **Network Repository Function (NRF):** Helps network functions find each other and share services. It acts as a central directory where functions register their services and discover others [12].

8. **Service Communication Proxy (SCP):** Enables indirect communication between network functions. It acts as an intermediary, facilitating flexible and scalable interactions within the SBA. The SCP enhances the network's ability to manage complex service topologies and improves overall system resilience [6].

9. **Security Edge Protection Proxy (SEPP):** Forms a key component of the roaming security architecture in 5G networks. It provides message filtering and policing on inter-PLMN control plane interfaces and supports topology hiding to protect internal network structure [6].

10. **Unified Data Repository (UDR):** Serves as centralized data storage for various network functions. It holds subscriber data, policy data, and structured data for other network functions. The UDR works closely with the UDM to provide a unified approach to data management in the 5G core network [6].

11. **Network Slice Selection Function (NSSF):** Plays a crucial role in network slicing by determining the appropriate slice instance to serve a UE. It determines the allowed NSSAI and configured NSSAI and helps determine which AMF should serve the UE [6].

12. **Binding Support Function (BSF):** Provides binding support between various network functions. It helps in storing and retrieving binding information, which is crucial for maintaining session continuity and ensuring proper routing of signalling messages [11]. The BSF also supports the registration and discoverability of PCFs [6].

1.2.2 Protocol Stack and Interfaces

The 5G protocol stack builds upon the foundation laid by Long Term Evolution (LTE), incorporating new layers and functionalities to meet the demands of next-generation mobile networks. This section outlines the key components of the 5G protocol stack for both the control plane and user plane. The user plane in 5G retains several layers from LTE, including the Physical (PHY), Media Access Control (MAC), Radio Link Control (RLC), and Packet Data Convergence Protocol (PDCP) layers. A notable addition is the Service Data Adaptation Protocol (SDAP) layer, positioned between the Radio Resource Control (RRC) and PDCP layers [13]. The aforementioned layers in the UE plane serve specific functions:

- PHY: Handles modulation-demodulation of physical channels, error detection, and frequency and time synchronization.
- MAC: Manages error correction through hybrid automated retransmission requests.
- RLC: Organizes and retransmits data as needed.
- PDCP: Builds upon its LTE counterpart; handling security for RRC messages and user data, manages sequence numbering, and performs header compression for efficient data transmission. [14].
- SDAP: Manages transmission by mapping QoS flows to appropriate radio resources [15]
- RRC: Exchanges control information to set session parameters.
- NAS (Network Access Stratum): Enhances accessibility of stored data for networked devices [13].

1.2.2.1 Control Plane Protocol Stack

The control plane protocol stack facilitates communication between the UE, the 5G Access Network (5G-AN), the Access and Mobility Management Function (AMF), and the Session Management Function (SMF) [2]. Key components of this stack include:

- NAS-SM (Session Management): Manages PDU Session establishment, modification, and release between the UE and SMF.
- NAS-MM (Mobility Management): Handles registration, connection management, and security aspects of NAS signaling.
- 5G-AN Protocol Layer: Varies depending on the access network type.
- NG Application Protocol (NG-AP): Facilitates communication between the 5G-AN node and the AMF.
- Stream Control Transmission Protocol (SCTP): Ensures reliable delivery of signaling messages.

1.2.2.2 User Plane Protocol Stack

The user plane protocol stack is responsible for transporting data in PDU Sessions, consisting of:

- PDU Layer: Carries data between the UE and the Data Network (DN).

- GPRS Tunnelling Protocol for User Plane (GTP-U): Encapsulates user data and supports QoS flow marking.
- 5G-AN Protocol Stack: Specific to the access network type.
- UDP/IP: Serve as the backbone network protocols [2].

1.2.2.3 Network Architecture Considerations

In the 5G architecture, the gNB is split into Central Units (CUs) and Distributed Units (DUs). This split allows for dynamic adaptation of QoS functions based on real-time radio conditions, user density, and geographical factors. CUs handle higher-layer protocols (SDAP and PDCP), while DUs manage lower-layer functions (RLC, MAC, and PHY). The gNBs are interconnected via the Xn interface, facilitating seamless communication between network nodes [13].

This comprehensive protocol stack enables 5G networks to deliver enhanced performance, improved QoS, and support for diverse use cases. For instance, the combination of NGAP and NAS protocols ensures that a user's smartphone (UE) can seamlessly connect to the network, authenticate, and maintain connectivity while moving between different base stations [16]. Meanwhile, the GTP protocols and N3/N4 interfaces enable the network to efficiently route high-bandwidth data for applications like 4K video streaming or augmented reality. [17]

1.2.3 Network Slicing

Network slicing is a paradigm that enables the creation of multiple logical networks on a shared physical infrastructure, each customized to meet specific service requirements [18]. It is a key enabler for 5G to support diverse use cases with varying performance needs. The network slicing architecture consists of three layers: the service instance layer, network slice instance layer, and resource layer [19]. Slices are managed throughout their life-cycle, including preparation, instantiation, runtime, and decommissioning phases. While network slicing offers benefits like resource efficiency and service customization, it also introduces challenges such as inter-slice isolation, security concerns, and complex orchestration. The realization of end-to-end network slicing requires advancements in various domains, including RAN virtualization, service composition, and slice management [18, 19].

1.3 Project Objectives

The implementation of 5G networks as described in the previous sections requires significant infrastructure and resources. However, the emergence of open-source implementations has made it

possible to develop and test 5G applications in virtualized environments. Of particular interest are two open-source frameworks - Open5GS for simulating the 5G core network, and UERANSIM for simulating the RAN and UE behavior. These tools, when combined with modern containerization technologies, pave way for the creation of comprehensive test environments for 5G applications.

This thesis aims to demonstrate the feasibility of implementing 5G use cases in a virtualized network environment using open-source tools, with the primary objectives being:

- To establish a fully functional 5G testbed environment by integrating Open5GS and UERANSIM, enabling end-to-end network simulation from core network functions to RAN behaviour
- To develop an Application Programming Interface (API) layer that simplifies network parameter configuration and management, making the testbed more accessible for application development
- To implement and validate three distinct use cases that showcase the key pillars of 5G: eMBB, URLLC, and mMTC
- To evaluate the performance and limitations of virtualized 5G networks in supporting these applications, providing insights for future research and development

1.4 Thesis Structure

The subsequent chapters of this thesis build upon the technical foundation established thus far to develop specific 5G use cases. The structure is as follows: Chapter 2 presents the state of the art in virtualized 5G networks, focusing on other open-source network simulators and associated testbed implementations. Chapter 3 delves into the implementation of containerized environments using Docker, setting the groundwork for application development. It further discusses the setup of sample client-server data transmission over the 5G tunnel and provides a detailed analysis of 5GC configuration parameters for developing our use cases. Chapter 4 discusses the development of an API layer over the Open5GS-UERNASIM setup, that provides developers an option to interact with network parameters during application design. Chapter 5 demonstrates a proof-of-concept 5G testbed framework using the API module. Finally, Chapter 6 outlines a summary of contributions, implications of the work, open issues, and future research directions.

Chapter 2

State-of-the-art Implementations

This chapter studies the current landscape of virtualized 5G network implementations, with a particular focus on open-source solutions and testbed environments. The discussion begins with an overview of major open-source frameworks that enable 5G network simulation, including detailed analyses of core network implementations and RAN simulation tools. We then investigate how these components are enhanced through virtualization technologies, specifically addressing the role of NFV and containerization in modern 5G deployments. Building on this foundation, we also analyze various experimental platforms and testbeds showcasing practical implementations of virtualized 5G networks. A literature study is then conducted on existing works in the domain of virtual 5G Testbed development that have inspired key aspects of this project. Finally, we identify the current capabilities, limitations, and gaps in existing implementations, providing the context for the research objectives and implementation approach.

2.1 Open5GS: An Open-Source 5G Core Implementation

Open5GS is an open-source project that implements the 5G core network functions as per the SBA. It provides a platform for developers to simulate and test 5G setups without the need for expensive hardware. Further, it uses MongoDB to store subscriber data and network state information. This makes it easier to quickly query and update subscriber profiles and session data. The database is structured to handle the complex data needed for 5G subscriber management, including support for features like network slicing and QoS profiles.[20]

2.1.1 System Architecture

Modern 5G core networks leverage Software-Defined Networking (SDN) and NFV principles to implement core functions as software rather than purpose-built hardware. The architecture follows

a service-based approach where the core network is fundamentally divided into control plane and user plane components. In the control plane, network functions are implemented as Cloud-Native Network Functions (CNFs) forming a service mesh that enables inter-service communication through HTTP/REST (Hypertext Transfer Protocol/Representational State Transfer) APIs. The user plane handles data packet processing and forwarding through functions like the UPF, which manages packet routing based on rules provisioned by the SMF. Among various open-source implementations of this architecture, Open5GS provides a comprehensive solution that emphasizes deployment flexibility and resource efficiency. Open5GS, implemented using hardware-close programming in C/C++, is primarily designed for Linux-based environments, with its installation and operation tailored to distributions such as Debian, Ubuntu, and openSUSE through package managers [20]. Its architectural approach allows Open5GS to effectively implement both NR/LTE networks equipped with gNodeB/eNodeB capabilities for private 5G deployment [21].

To give a clear picture of the Open5GS architecture, Figure 2.1 shows how the different network functions are connected in both 4G and 5G. The diagram also highlights the separation between the control plane and user plane (CUPS) in Open5GS.

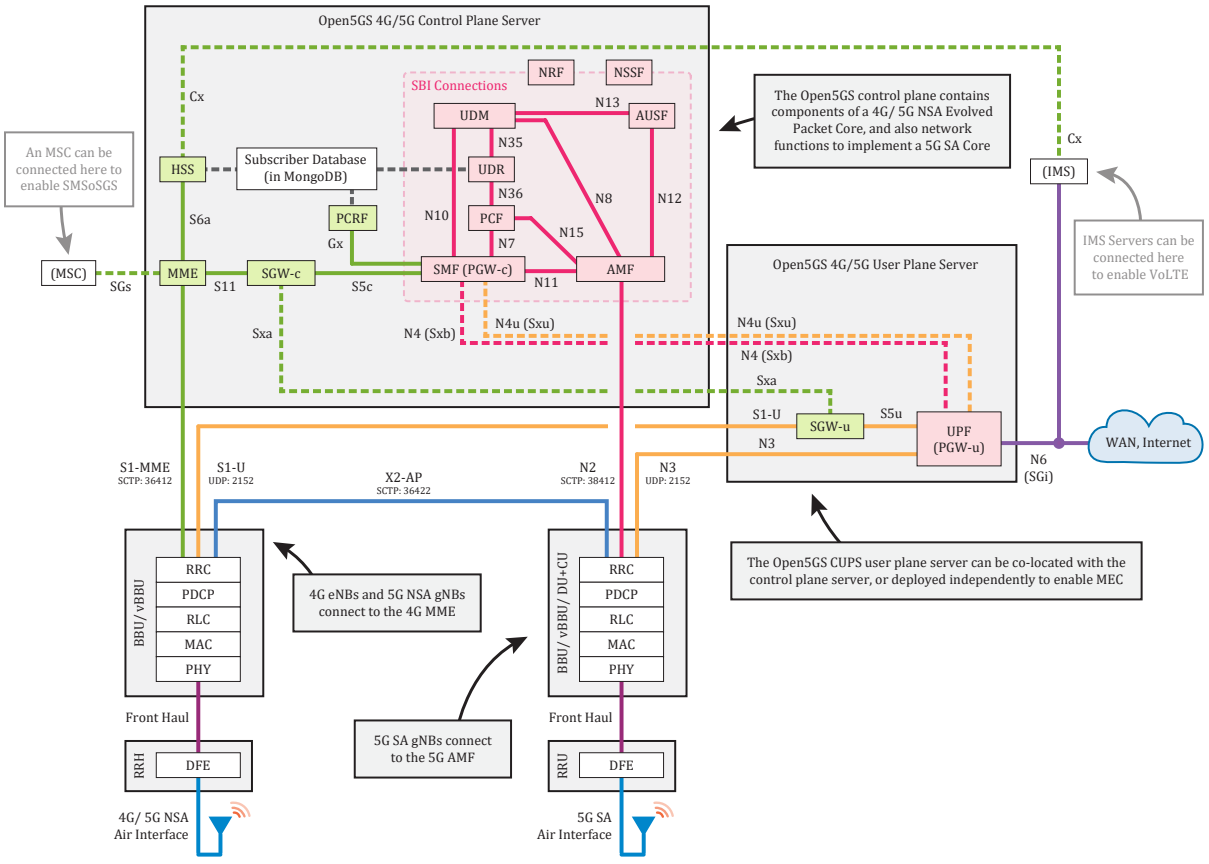


FIGURE 2.1: Open5GS Architecture: Control and User Plane Separation [22]

As shown, Open5GS has a flexible architecture that supports both 4G Evolved Packet Core (EPC) and 5G Core (5GC) functions. On the left side of the diagram, we see the control plane

components, such as the MME for 4G, and the AMF, SMF, PCF, UDM, AUSF, and NRF for 5G. On the right side, we see the user plane components, with SGW-U (Serving Gateway - User plane) and PGW-U (PDN Gateway - User plane) functions for 4G, and the UPF for 5G. Additionally, the diagram shows how Open5GS can support advanced features like Mobile Edge Computing (MEC). By separating control and user plane functions, it allows for flexible network setups to meet different needs.

2.2 UERANSIM: Simulating 5G UE and RAN

UERANSIM is another open-source project that provides a comprehensive simulation environment for 5G standalone UE and RAN components. It complements core network implementations like Open5GS by enabling end-to-end testing of 5G networks. UERANSIM simulates two primary components: the UE, analogous to a 5G-capable mobile device, and the gNodeB, representing a 5G base station. The simulation encompasses three main interfaces: the control interface between RAN and AMF, the user interface between RAN and UPF, and the radio interface between UE and RAN.

The control plane in UERANSIM is divided into two key protocols: Non-Access Stratum (NAS) for UE control and Next Generation Application Protocol (NGAP) for RAN control. The NAS implementation includes crucial features such as authentication, security mode control, registration procedures, and PDU session management. NGAP supports various functions including PDU session resource management, UE context handling, and paging. For the user plane, UERANSIM implements the GTP with current support for IPv4. The radio interface simulation in UERANSIM is partial, focusing on protocols above the RRC layer. Lower layer protocols (PHY, MAC, RLC, PDCP) are abstracted and simulated over UDP, providing a simplified representation of the 5G radio environment.[23]

2.3 NFV and Containerization in 5G Networks

The advent of 5G networks has necessitated a paradigm shift in network architecture to meet the demands of ultra-low latency, high bandwidth, and massive connectivity. NFV and containerization have emerged as key technologies in realizing the full potential of 5G networks, offering improved flexibility, scalability, and cost-effectiveness [24, 3].

2.3.1 Network Function Virtualization

NFV decouples network functions from dedicated hardware, allowing them to run on standard servers [24]. This approach offers significant advantages in the context of 5G networks. By virtualizing network components such as the EPC and RAN functions, NFV enables more efficient resource utilization and dynamic management of network services [25].

The implementation of NFV in 5G networks facilitates the creation of a more flexible and adaptable infrastructure. Network operators can deploy and scale network functions as needed, and respond quickly to changing demands [24]. Furthermore, NFV contributes to significant reductions in both capital expenditure and operational expenditure. By leveraging commodity hardware and centralizing network functions, operators can optimize resource allocation and streamline network management processes [25].

2.3.2 Containerization in 5G

While traditional NFV implementations often rely on virtual machines (VMs), containerization has gained significant traction as an alternative virtualization technology for 5G networks. Containers offer several advantages over VMs, particularly in terms of resource efficiency and deployment speed. This involves packaging software applications along with their dependencies, allowing them to run consistently across different computing environments. In the context of 5G networks, this approach enables the creation of lightweight, portable network functions that can be rapidly deployed and scaled [3].

Recent advancements have demonstrated the feasibility of running a fully containerized 5G core using open-source solutions like Open5GS [3]. This approach leverages container orchestration platforms such as Kubernetes to manage the deployment and scaling of 5G network functions. The benefits of containerization in 5G networks include simplified deployment and management of network functions, improved scalability, and enhanced flexibility in testing and development environments.

Studies comparing the performance of VMs and containers in virtualizing RANs have shown promising results for containerization [25]. Containers exhibit faster instantiation times compared to VMs, lower communication latency between network functions, and improved elasticity in resource utilization. These characteristics make containerization particularly well-suited for the dynamic and demanding environment of 5G networks. As 5G networks continue to evolve, NFV and containerization will play increasingly important roles in enabling the flexibility, scalability, and efficiency required to support next-generation mobile services. By leveraging these technologies, network operators can create more agile and cost-effective infrastructures capable of meeting the diverse requirements of 5G use cases.

2.4 Virtual 5G Testbeds: A Literature Review

Significant research has been conducted in the area of testbed development that utilize open-source implementations and containerization technologies for virtual networks. The authors in [26] describe an enterprise-scale Open RAN testbed designed to overcome the limitations of existing platforms. In their implementation, the authors have developed a highly scalable system capable of supporting a substantial number of simulated users and base stations, enabling comprehensive testing at an enterprise scale. The testbed's architecture incorporates significant flexibility, allowing researchers to reconfigure network parameters and topology with ease, thus facilitating a wide range of experimental setups. Furthermore, the authors implemented this testbed using software-defined radio (SDR) platforms over open-source software components.

The VET5G platform, introduced in [27], represents a significant advancement in security experimentation for 5G networks. The platform's architecture demonstrates sophisticated integration of core network and RAN emulators, creating a comprehensive 5G network environment that enables thorough security testing. A notable feature is its implementation of streamlined Python APIs for network configuration and management. The platform's security-focused design enables researchers to conduct extensive testing of various security scenarios and protocols in a controlled environment, making it an invaluable tool for 5G security research. This methodology forms the foundation of our approach to programmatic control for abstracting the network-level complexities from an application developer's perspective.

A significant contribution to performance testing methodologies is presented in [28], where the authors develop and implement a comprehensive testing framework as an extension of my5G Tester. Their work introduces sophisticated performance evaluation capabilities, particularly focusing on two critical aspects: the user equipment registration and session establishment procedures at scale, and the streaming of user data over parallel data plane connections. The implementation demonstrates valuable insights into open-source 5G core performance, revealing that Open5GS exhibits better stability during multi-device registration processes, maintaining connection times generally under 500 milliseconds, while free5GC shows superior performance in data plane operations, achieving throughput up to approximately 1 Gbps with multiple connected UEs. Another research in [29] provides valuable insights into the performance capabilities of open-source 5G core implementations under stress conditions. Their work specifically focuses on evaluating the User Plane Function (UPF) within Open5GS, employing UERANSIM for testing. Their findings demonstrate impressive resource efficiency, with the Open5GS UPF implementation supporting up to 200 concurrent data connections while utilizing only 17% of CPU resources and handling 200 control traffic connections with just 15% CPU usage.

The BlueArch 5G testbed [30] represents a significant advancement in customizable 5G platforms. The implementation excels in its comprehensive support for multiple use cases, demonstrating

remarkable versatility in accommodating diverse 5G applications. A key feature of the platform is its sophisticated integration of edge computing capabilities, aligning with the industry trend toward edge-centric 5G architectures. The testbed's architecture emphasizes flexibility and scalability, incorporating design elements that facilitate straightforward expansion to support emerging use cases and technologies. Their methodology reinforces our approach to implementing eMBB, URLLC, and mMTC applications within a unified framework.

A noteworthy contribution in [31] demonstrates a practical implementation of network slicing for rural connectivity using open-source components. Their testbed integrates multiple technologies including UERANSIM, Open5GS, ONOS SDN controller, OpenStack, and Open-Source MANO (OSM) to create a comprehensive network slicing framework. The implementation successfully demonstrates the deployment of multiple network slices with distinct PLMNs. Their work has been particularly relevant in our modular approach to API development for configuring PCF parameters.

2.5 Conclusions and Significance of the Study

This review of state-of-the-art implementations in virtual 5G testbeds reveals the following gaps in the existing landscape of virtual 5G testbed development:

- While platforms like VET5G offer API capabilities, there is no unified framework that provides simplified programmatic control over both core network and RAN components, in conjunction with system level operations for 5G use-case development
- Existing implementations focus primarily on infrastructure performance evaluation and possess little control on facilitating rapid application development and testing
- Despite proven capabilities of Open5GS and UERANSIM individually, there is limited work on creating developer-friendly interfaces that abstract their complexity

Our work addresses these gaps by developing an integrated framework that combines the stability of Open5GS, the simulation capabilities of UERANSIM, and the efficiency of containerization to create a comprehensive platform for 5G use-case implementation. The demonstrated success of these open-source components in various implementations, as evidenced by the literature, validates their selection as the foundation for our research while our framework addresses the critical need for simplified application development and testing capabilities.

Chapter 3

Virtualized Network Architecture and Configuration Options

The deployment of telecommunications networks has evolved significantly from traditional hardware-bound implementations to more flexible virtualized solutions. Initially, network elements were tied to physical hardware in a "black box" model, where hardware and software acted as a monolith, limiting operators' visibility and requiring hardware additions for capacity expansion. This evolved into NFV, where network elements could be installed on commercial off-the-shelf hardware or virtual machines, eliminating the need for specialized hardware. However, the industry has recognized the need to run software without virtualizing entire operating systems, leading to the adoption of containerization. This approach enables software to be packaged in a way that facilitates convenient distribution, upgrades, and management while maintaining standardization. Containerization provides several advantages over traditional virtualization: it eliminates the need for complete virtual machines to run different software components, enables efficient resource sharing, and ensures that multiple components can coexist without conflicts [3].

In the context of 5G networks, containerization becomes particularly valuable as it enables rapid deployment and testing of network functions while maintaining isolation between components. This capability is crucial for developing and testing various network configurations and scenarios, especially in research and development environments where flexibility and resource efficiency are paramount [32].

3.1 Containerized Infrastructure Implementation

This implementation establishes a containerized 5G network infrastructure that virtualizes the complete protocol stack from the core network to the RAN. Figure 3.1 illustrates the network

architecture implemented in our setup, highlighting the interconnections between different network functions and their respective reference points.

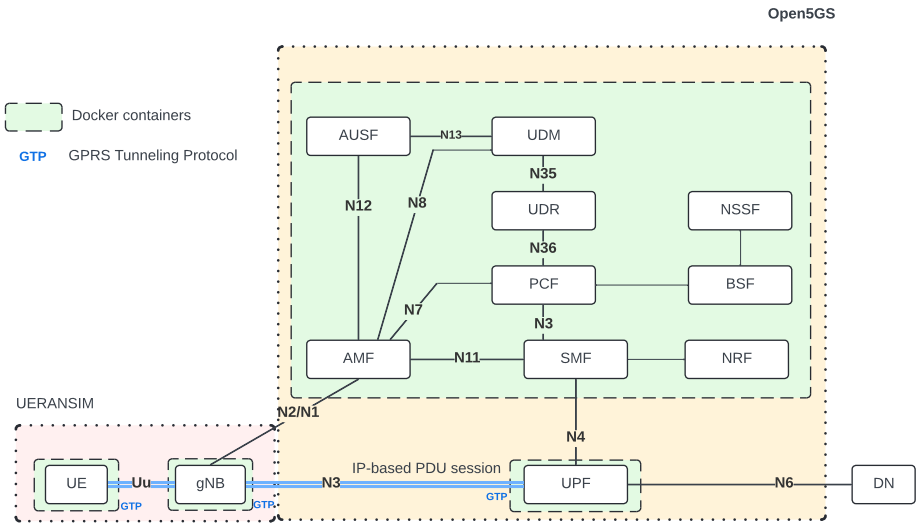


FIGURE 3.1: Containerized Open5GS and UERANSIM Architecture

At the heart of this setup is the Open5GS implementation, containerized to have service-based interfaces between network functions. The container has multiple network functions - AMF, SMF, UDM, AUSF, UDR, PCF, BSF, NSSF, and NRF. Each of these network functions is connected through standardized reference points (N3-N36). This uses MongoDB for storage of subscriber and network config data, with the database container running on a dedicated volume so that the state is preserved across system restarts. The UPF runs in a separate container, implementing N3 for user plane traffic and N4 for control plane communication with SMF. The UERANSIM implementation has two containers - gNB and UE. The gNB container establishes N2/N1 interfaces with AMF and handles RRC and NAS signaling. The UE container simulates end devices and creates virtual network interfaces through TUN devices. These containers implement the Uu interface between UE and gNB, simulating the radio protocol stack through UDP-based communication [22].

The network has two separate bridge networks: an internal 5G signalling network (10.10.0.0/24) and a public network (10.11.0.0/24) for external data services. This segmentation allows for control plane isolation and flexible routing of user plane data to external data networks. This containerized environment enables verification of end-to-end data transmission through the 5G core using Wireshark packet analysis. The testing process involves examining multiple networking layers to confirm proper tunnel establishment. At the container level, traffic flows between Docker-assigned addresses (in the `10.10.0.0/24` subnet) over the internal bridge network.

The key validation comes from analyzing the **GTP (GPRS Tunneling Protocol)** messages that carry the actual user plane data. These GTP packets are exchanged between:

- The UE's tunnel interface (`uesimtun`, with addresses in the `10.45.0.x` range where $x \geq 2$)
- The UPF's tunnel interface (`ogstun`, using `10.45.0.1`)

By filtering Wireshark captures for GTP traffic, we can verify successful tunnel establishment by observing the encapsulated data flow between these endpoints. The presence of regular **GTP-U (user plane)** traffic on UDP port `2152`, along with consistent packet lengths and timing, confirms proper functionality of the end-to-end 5G data path through both radio access and core network segments.

3.2 Gateway Implementation for 5G Tunnel Transmission

In traditional physical network environments, end devices connect to a modem using interfaces such as Peripheral Component Interconnects (PCIs) or USB, alongside modem-specific protocols. These interfaces facilitate communication with the RAN and the core network, which employ IP-based protocols for configuration and data exchange. However, within a virtual network infrastructure, such interfaces either differ significantly or are absent altogether [33]. From the perspective of an application developer, this discrepancy presents unique challenges.

A critical requirement in this setup is abstracting the configuration and control interfaces to provide a stable interaction point for applications. This will ensure seamless integration between the application layer and the underlying network infrastructure, enabling reliable communication and control without dependency on physical interfaces. Furthermore, deployment in virtualized 5G networks requires addressing continuity concerns. When network parameters in an Open5GS environment are modified, the infrastructure often necessitates redeployment, which can disrupt applications running within container volumes. To address these challenges, we implement a gateway system that serves two critical purposes:

1. Ensuring application continuity during network reconfigurations by maintaining application state outside the containerized environment
2. Providing a stable interface for external applications to interact with the 5G network, independent of container-specific addressing

This gateway implementation creates a two-tier architecture where applications operate outside the containerized network environment while interfacing with it through dedicated gateway components. This enables applications to maintain their state and continue functioning even when the underlying container infrastructure is redeployed during network configuration. Additionally,

it simplifies application development by providing consistent endpoints that abstract away the complexity of container-specific networking.

3.2.1 Operational Design and Architecture

The gateway architecture implements distinct components for the UE and UPF sides of the network, each serving specific roles in data transmission. The UE gateway exposes a REST API endpoint (`http://10.10.0.132:<port>`) that accepts HTTP POST requests from source applications. This interface connects to a TUN device that handles network-level communication, which is then forwarded through a UDP client to the 5G tunnel. On the receiving end, the UPF gateway listens for tunnel traffic through a UDP server (`http://10.10.0.112:<port>`). Received data is temporarily stored in a message queue before being made available through a REST API endpoint for destination services to retrieve. This queuing mechanism ensures reliable data delivery while maintaining the asynchronous nature of the communication.

As shown in Figure 3.2, this architecture creates a clear separation between external applications and the internal 5G network components while maintaining a consistent data flow path. The design uses the localhost interface for all external communication, providing a stable interface that remains accessible regardless of container state.

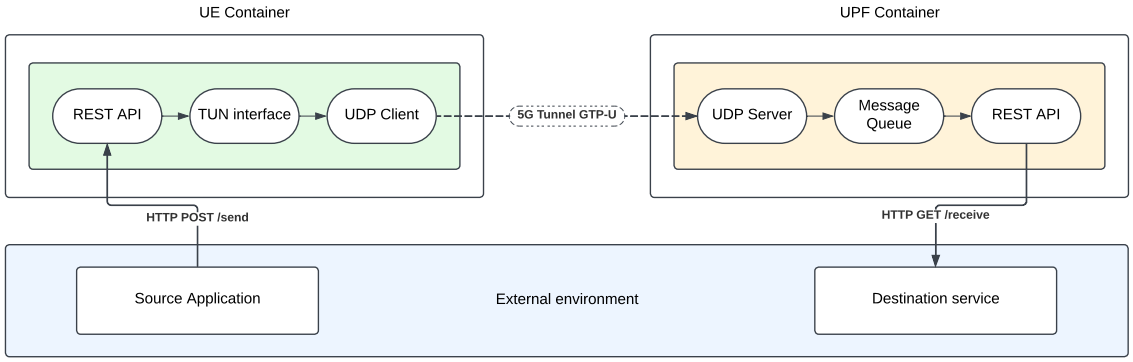


FIGURE 3.2: Gateway Architecture and Data Flow in Virtual 5G Network

In this design, source applications and destination services are outside the containerized network environment, interacting with the 5G infrastructure through dedicated gateways that expose the URLs which can then be used with relevant endpoints to perform transmit or receive operations. This way, applications keep their state and UE/UPF side applications simply run on different ports of the localhost interface, thus supplying webpage content even when the underlying network containers are redeployed during configuration changes.

3.2.2 UE Gateway Architecture

The UE gateway acts as the main entry point for sending data into the 5G network tunnel. It manages interfaces, handles multiple connections at the same time, and processes data effectively. It uses the Flask framework for its REST API, which can handle many requests at once and uses UDP sockets for reliable data transmission.

The core of the UE gateway is its interface management system. In a virtual 5G setup, UERANSIM creates Tunnel (TUN) interfaces (named with the prefix `uesimtun`) to simulate network connectivity for each UE instance. The gateway uses the `netifaces` library to detect and track the IP address of these interfaces and keep an updated list of active UE connections. This dynamic detection is key for handling multiple UE instances because it adjusts automatically to the number of connected devices without needing manual setup. To support multiple connections, the system uses a multi-threaded design. For each detected TUN interface, a new thread is created to run a Flask server instance. These threads are set up as daemon threads, meaning they automatically close when the main program ends. Ports are assigned in order, starting at a default base port (8080) and increasing for each new interface. This setup offers several benefits:

- Independent operation of multiple UE instances without resource conflicts
- Isolation of data streams, preventing cross-interference between different source applications
- Efficient resource utilization through parallel processing of requests

To handle the size limitations of UDP datagrams while maintaining reliable data transmission, the gateway implements a sophisticated chunking mechanism. When incoming data exceeds 65,507 bytes (maximum UDP payload size for IPv4 [34]), the system segments it into 65,000-byte chunks, leaving room for headers and metadata. The chunking process follows a structured protocol ensuring reliable transmission of large datasets while maintaining packet integrity within network constraints:

1. Generation of an initial frame header containing the total chunk count
2. Creation of individual chunk headers with sequence information
3. Sequential transmission of chunks through the UDP socket

3.2.3 UPF Gateway Implementation

The UPF gateway implements a complementary system for receiving and processing data from the 5G tunnel before forwarding to destination services. The implementation centres around

three core components: a UDP server for receiving tunnel data, a message queue for temporary storage, and a REST API for external service interaction.

The UDP server binds to a specific IP address - 10.45.0.1 and port 5005, operating on a dedicated thread to ensure continuous availability for incoming data. This thread isolation is crucial as it prevents blocking other gateway operations during data reception and processing. The server implements a continuous listening loop, processing incoming packets based on their type and content. For handling chunked transmissions, the gateway implements a frame assembler system. This component maintains a state machine for each ongoing transmission, tracking:

- The expected number of chunks for the complete transmission
- Currently received chunks and their sequence numbers
- Partial assembly state for multiple concurrent transmissions

The assembler validates each received chunk against its header information and manages the final reassembly process once all chunks are received. The system uses a thread-safe queue, built with Python's `queue` module, to manage data flow between the UDP listener and API endpoints. First, it provides temporary storage of received data from the UDP listener, preventing data loss during high-load scenarios. Second, it implements non-blocking operations in its data retrieval mechanism. In a traditional blocking operation, when an API endpoint requests data from an empty queue, the request would wait indefinitely until data becomes available. This could lead to system resources being tied up and requests potentially hanging for long periods. Instead, the implementation uses non-blocking operations where requests for data from empty queues receive an immediate response indicating no data is available (HTTP 204 status code). This approach allows the system to maintain responsiveness - API endpoints can immediately inform destination services about the absence of data rather than holding the connection open waiting for data to arrive. The system also maintains the crucial association between received data and its source IP address throughout the transmission path. This metadata preservation ensures that even in scenarios with multiple concurrent transmissions, each data packet can be correctly attributed to its source.

The REST API implementation exposes distinct endpoints, each serving specific data handling requirements:

1. A sensor data endpoint (`/receive/sensor`) specifically for JSON-formatted data, adding timestamps and source information
2. A stream endpoint (`/receive/streamer`) for handling continuous image streams with appropriate MIME type management

This specialized endpoint design allows destination services to efficiently retrieve data in their required format while maintaining proper data typing and metadata management. Each endpoint implements appropriate error handling and response formatting, ensuring reliable data delivery to the destination services.

3.2.4 Data Handling and Sequence Flow

The data handling architecture implements a dual-pipeline approach optimized for two distinct types of data transmission through the 5G tunnel: structured JSON data for sensor information and frame data for continuous streaming. The system automatically detects the data type and routes it through the appropriate processing pipeline, as depicted in Figure 3.3.

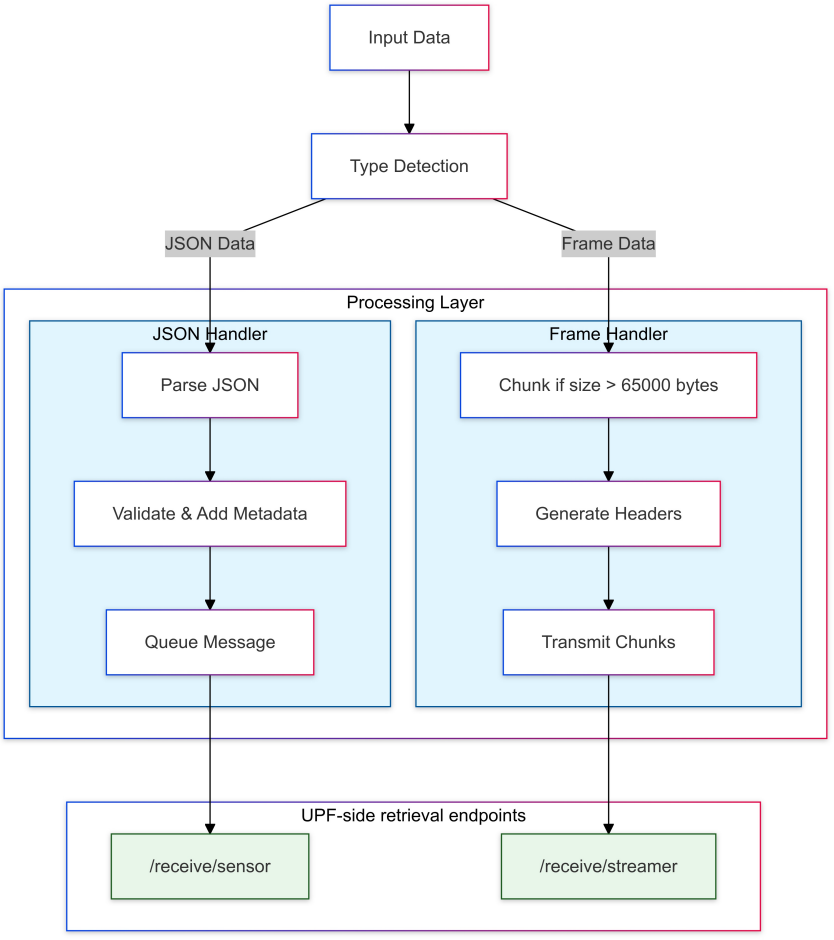


FIGURE 3.3: UE and UPF Gateway Data Processing Flowchart

At the UE gateway, incoming data undergoes type detection through JSON parsing attempts. For JSON data, the system performs direct transmission as the payload sizes are typically small. For video frame-type data, the system implements a chunking mechanism when the data size exceeds the UDP datagram limit of 65,000 bytes. This chunking process involves:

1. Generation of a frame header containing total chunk count
2. Sequential transmission of chunks with individual headers
3. Addition of sequence information for reliable reassembly

At the UPF gateway, two specialized endpoints handle the received data:

- `/receive/sensor`: Processes JSON data with enhanced features:
  - JSON validation
  - Timestamp addition
  - Source IP metadata inclusion
  - Standardized response formatting
- `/receive/streamer`: Manages frame data by:
  - Implementing frame reassembly from chunks
  - Setting appropriate MIME types
  - Providing a direct binary response

3.3 Policy Control Function (PCF) Configuration

The PCF acts as the policy decision point in the 5G core network, using complex rule sets to manage network behaviour and allocate resources. Its configuration framework uses a structured YAML format, allowing for detailed control over network operations and service quality.

3.3.1 Core System Configuration

The PCF ensures persistence by integrating with MongoDB, using a Uniform Resource Identifier (URI) connection string to specify the database location and access details (see Appendix A.1). This connection allows the storage and retrieval of policy decisions, subscriber information, and session data. The logging system follows a hierarchical severity structure, covering everything from critical errors to detailed trace-level debugging. Logs are saved to designated file paths, providing a reliable way to monitor and troubleshoot policy decisions and overall system behaviour.

3.3.2 Service Interface Implementation

The service-based architecture in PCF operates through HTTP/2-based interfaces, binding to specified network addresses and ports, as shown in Appendix A.1. The primary service interface operates on port 7777, handling policy requests and decisions. A separate metrics interface on port 9090 enables performance monitoring and system statistics collection through Prometheus. The architecture supports both direct communication with the NRF and proxy-based communication through the Service Communication Proxy, enabling flexible service discovery and load distribution in the core network.

3.3.3 Network Slice Configuration

Network slicing enables the creation of virtualized network partitions with specific characteristics. The configuration defines slices through S-NSSAI (Single Network Slice Selection Assistance Information) parameters, combining a Slice Service Type (SST) with an optional Slice Differentiator (SD). The implementation supports SST values 1 through 4, representing different service characteristics: eMBB (1), URLLC (2), and massive-IoT (3) [35]. Each slice can be marked as a default slice through the default indicator parameter.

3.3.4 Session Management Framework

Session management in the PCF sets the rules for data connectivity. Each session configuration defines the access type through the session name and specifies the IP addressing scheme. The type parameter determines whether the session uses IPv4 (type 1), IPv6 (type 2), or dual-stack IPv4v6 (type 3) addressing.

The Aggregate Maximum Bit Rate (AMBR) parameters set the total available bandwidth for a UE across all non-guaranteed bit rate flows in a session. The configuration allows for flexible bandwidth units, ranging from bits per second to terabits per second, using unit values from 0 to 4. This provides fine-tuned control over bandwidth, with separate settings for uplink and downlink directions, each defined by a value and unit parameter (0 - bps, 1 - Kbps, 2 - Gbps, 3 - Tbps).

3.3.5 Quality of Service Control

The QoS framework implements a comprehensive 5QI (5G QoS Identifier) system through the QoS index parameter. Values range from 1 to 86, each representing specific service characteristics combining resource type, priority level, packet delay budget, and packet error rate requirements.

The Allocation and Retention Priority (ARP) mechanism manages resource allocation under congestion scenarios through priority levels 1 through 15, where lower values indicate higher priorities [36].

Pre-emption capabilities within ARP are controlled through two parameters: preemption vulnerability and preemption capability. When enabled (value 2), preemption capability allows a bearer to pre-empt other bearers with higher vulnerability values. Conversely, preemption vulnerability determines whether a bearer can be pre-empted by others with preemption capability [37].

3.3.6 Flow Management System

The flow management configuration implements traffic control through detailed flow descriptions. Each flow rule specifies a direction parameter, where value 1 represents uplink traffic (from UE to network) and value 2 represents downlink traffic (from network to UE). The description field contains packet filtering rules using a specialized syntax that defines permitted traffic patterns.

Flow descriptions follow a structured format [22]: 'permit out ip from any to assigned' allows IP traffic in the specified direction between any source and the assigned UE address. Similarly, ICMP-specific rules ('permit out icmp from any to assigned') enable control message protocols. These rules can be combined to create sophisticated traffic management policies, enabling or restricting specific traffic types based on protocol, direction, and addressing patterns, with example configurations provided in Appendix A.1.

3.3.7 Policy and Charging Control (PCC) Rules

PCC rules combine QoS parameters with specific flow definitions to create comprehensive traffic management policies. Each rule specifies Maximum Bit Rate (MBR) and Guaranteed Bit Rate (GBR) parameters for both uplink and downlink directions. These parameters utilize the same unit system as AMBR configurations but apply specifically to the flows defined within the rule. The GBR parameters ensure minimum bandwidth availability for critical services, while MBR parameters cap the maximum resource utilization.

Chapter 4

API Implementation Framework

The complexity of 5G network operations, especially in virtualized testbed environments, requires advanced automation and management capabilities. The creation of a comprehensive Python API module arose from the need to automate essential network management tasks while offering detailed control over network configurations. The primary motivation for this development was the limitations of manual configuration methods in Open5GS, where changes to policies involved directly editing YAML configurations and restarting services, which could lead to inconsistencies and operational inefficiencies.

The API module tackles these challenges by providing programmatic control over three key areas of network management: PCF configuration, UE management, and UPF control. Python was chosen as the implementation language due to its strong support for network programming, its extensive libraries for YAML processing and HTTP communication, and its efficient memory management capabilities, such as deque implementations. In addition, the API implementation includes thorough performance monitoring capabilities, allowing for real-time analysis of network behaviour and QoS compliance. This was essential for validating network configurations and ensuring service level agreements in research and testing environments. Through this automated approach, the system minimizes configuration errors, facilitates quick reconfiguration of testbeds, and lays the groundwork for developing advanced 5G use cases.

4.1 Architectural Overview

4.1.1 Design Philosophy

The API module provides a complete abstraction layer between applications and the Open5GS core network, following key software engineering principles that robustness and scalability. The architecture is based on three core design principles:

- **Singleton Pattern Implementation:** The central Open5GS class uses the singleton pattern to ensure a single point of control for network operations [38]. This was essential for keeping the network state consistent across all components and avoiding resource conflicts. The singleton instance handles shared resources, configuration states, and communication interfaces, offering fine-tuned operations for effective network management.
- **Separation of Concerns:** The architecture separates different functional areas into individual modules, following the principles of high cohesion and low coupling [39]. This ensures that each component focuses on a specific part of the system, making it easier to maintain and extend in the future.
- **Observer Pattern for Metrics:** The observer pattern [38] is used for collecting metrics, enabling real-time network performance monitoring without affecting the core network operations. This asynchronous approach ensures that performance tracking does not add any latency to the main data flow.

4.1.2 Core Components

The Open5GS API package is structured around five primary components that work together to provide comprehensive control and monitoring of the Open5GS network environment, as illustrated in Figure 4.1. Each component serves a specific role in the overall system architecture while maintaining clear interfaces for interaction. Table 4.1 details these components and their key functions, demonstrating how each element contributes to the overall functionality of the API package through specialized responsibilities ranging from configuration management to error handling.

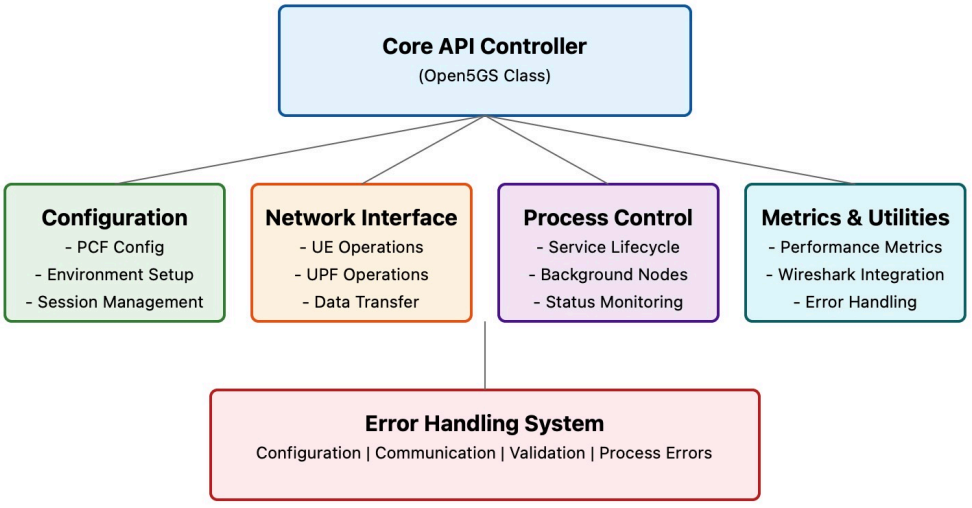


FIGURE 4.1: API Core Components

Table 4.1: API Core Components and Their Functions

Component	**Functions**
Configuration Management	• PCF configuration handling through YAML file management • Environment configuration for UE setup and system parameters • Session management including creation, modification, and deletion • Policy rule configuration and validation
Network Interface	• UE interface operations for user equipment management • UPF interface operations for user plane control • Data transmission and reception handling • Support for multiple data types (sensor data, video frames)
Process Management	• Service initialization and shutdown procedures • Background node management • Container orchestration and deployment • Health monitoring and status reporting
Metrics and Utilities	• Network performance metrics collection and analysis • Specialized metrics for different traffic types • Wireshark integration for packet analysis • System diagnostic utilities
Error Handling System	• Configuration errors (file access, invalid settings) • Communication errors (network connectivity, API endpoints) • Validation errors (parameter validation, constraint checking) • Process management errors (service lifecycle, background processes)

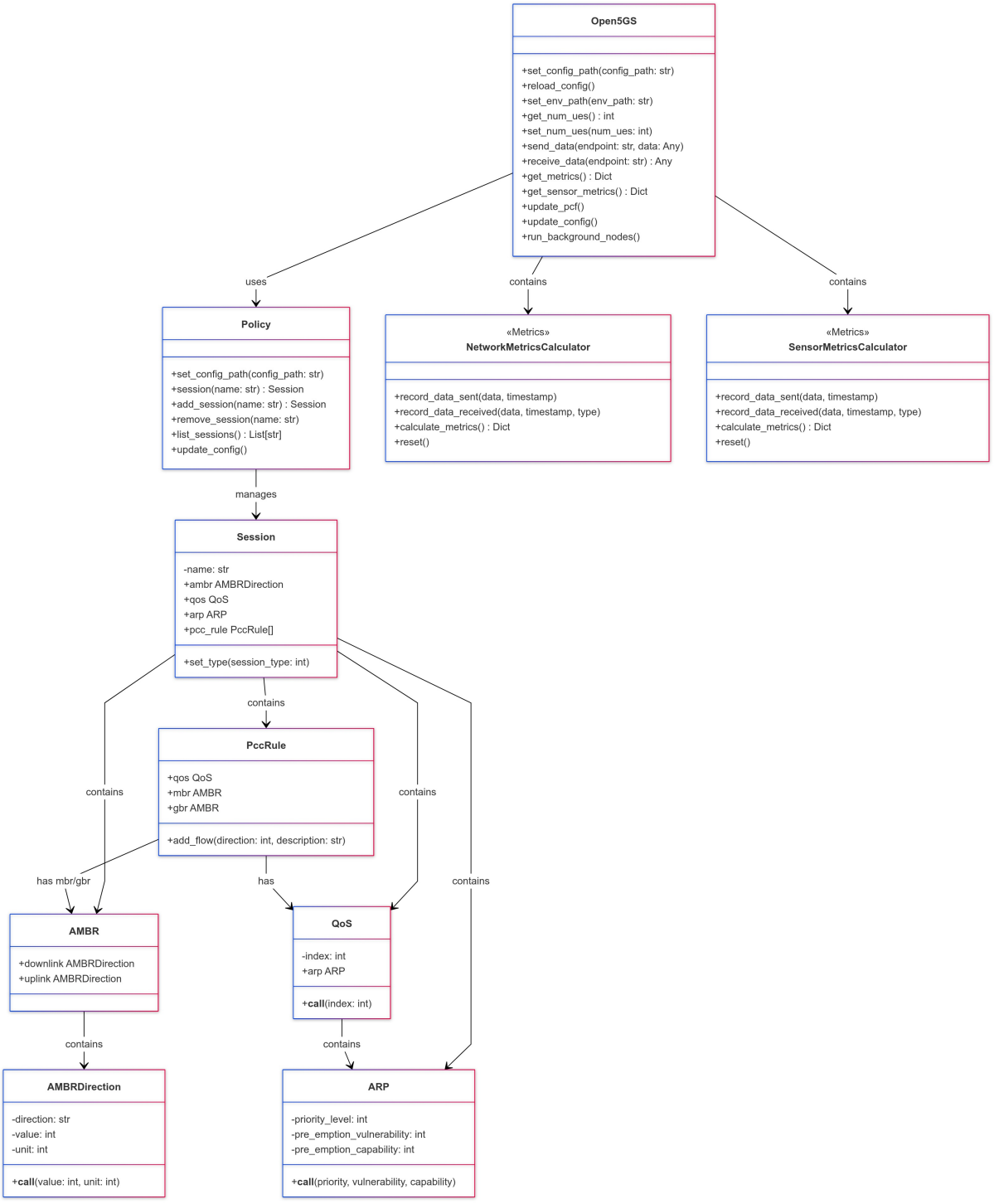


FIGURE 4.2: `open5gsapi` Class diagram depicting network components

The class diagram in Figure 4.2 shows how different parts of the system connect and interact. The Open5GS class works as the main manager, coordinating between components. The design ensures that multiple processes can run smoothly at the same time by using methods to prevent data conflicts. It also includes ways to handle errors, from small fixes within a module to broader system-level recovery. This modular structure makes it easy to test parts of the system separately

and adapt it for future improvements. Further, this API module plays a key role by managing complex 5G setups and providing a reliable interface for app development.

4.2 API Methods and Functionality

The `open5gsapi` package implements a comprehensive set of interfaces that enable programmatic control over the virtualized 5G network environment. The API architecture is organized into six main functional domains, each serving specific aspects of network management and monitoring:

- **Configuration Management:** Provides methods for managing system configurations, including PCF YAML file handling and UE settings management. This domain ensures a consistent configuration state across the network environment.
- **Session Management:** Implements interfaces for creating and managing network sessions, handling QoS configurations, and managing PCC rules. These methods enable dynamic modification of network behaviour based on application requirements.
- **Network Communication:** Offers abstracted interfaces for data transmission through the 5G tunnel, handling both sensor data and streaming media. The communication layer automatically manages data type detection, chunking for large payloads, and metrics collection.
- **Process Management:** Controls the lifecycle of network components, managing container deployment, background processes, and system health monitoring. This domain ensures proper initialization and coordination of network functions.
- **Metrics Collection:** Implements comprehensive performance monitoring capabilities, tracking network metrics, frame statistics for streaming applications, and sensor data measurements. The metrics system supports both real-time monitoring and historical analysis.
- **Session Configuration:** Provides detailed control over network session parameters, including QoS settings, bandwidth management, and flow control rules. This enables fine-grained control over network behaviour for different application requirements.

Each domain implements specific error handling and validation mechanisms, ensuring robust operation while maintaining a clear separation of concerns. The complete API specification, including detailed method signatures and functionality descriptions, is provided in Appendix C.1.

4.3 Sequence Flows

The Open5GS API orchestrates complex interactions between network components systematically. Understanding these flows is crucial for effectively using the API for network management and application development. Figure 4.3 illustrates the key operational sequences, showing how different components interact during typical usage scenarios.

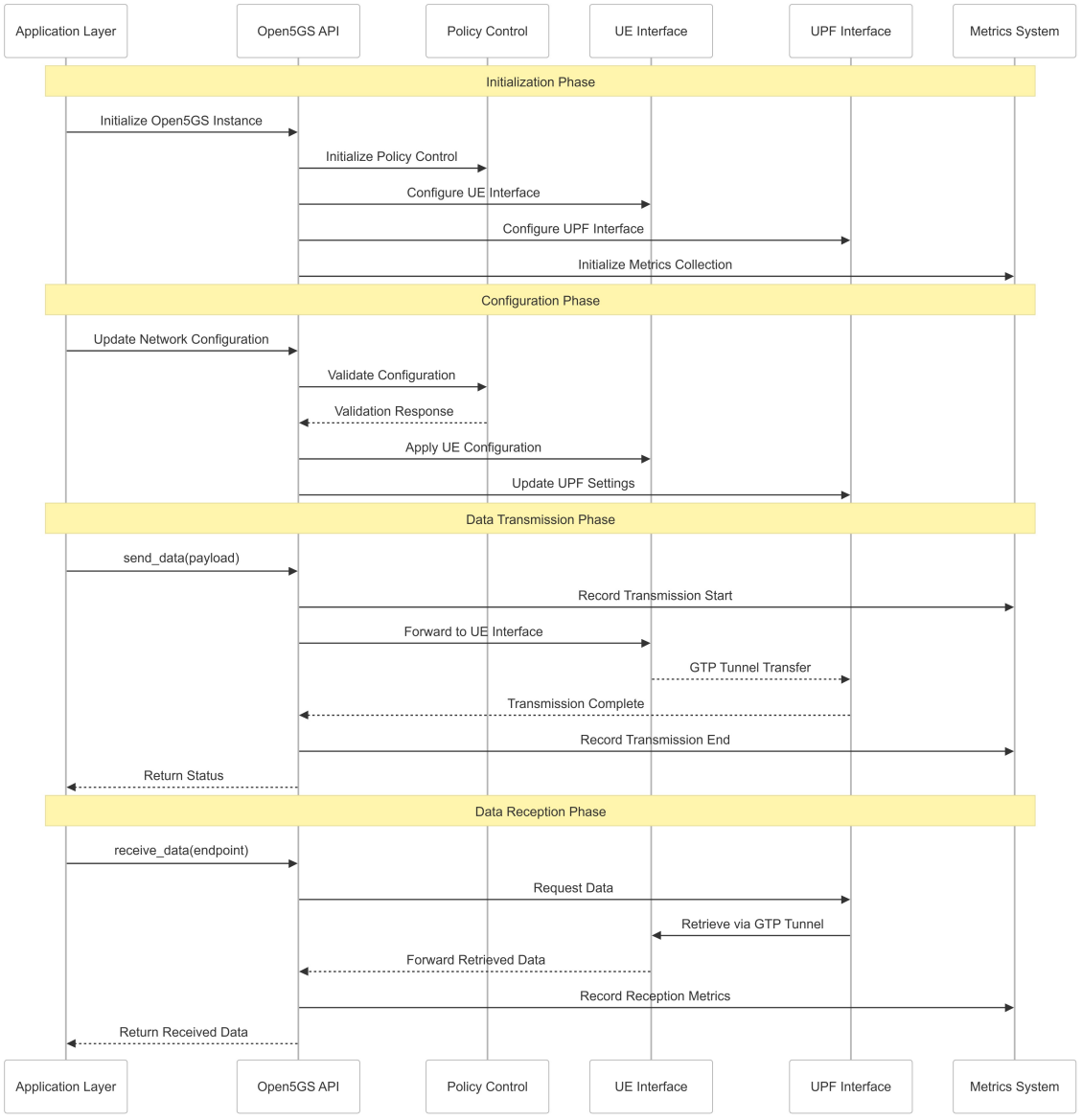


FIGURE 4.3: API sequence diagram

4.3.1 Initialization and Configuration Flow

Before any network operations can begin, the API must properly initialize and configure all components. This process involves three key stages:

1. **Initial Setup:** The application first creates an instance of the Open5GS API using the global `open5gs` object. Paths to the configuration files must then be set using `set_config_path()` for `pcf.yaml` and `set_env_path()` for `.env`. The number of UEs can configured using `set_num_ues()` if multiple UEs are needed.

2. **Configuration Management:** Network sessions are created or modified using the Policy interface (`open5gs.policy`). Each session can be configured with specific QoS parameters, AMBR settings, and PCC rules. Changes are validated automatically to ensure compliance with 5G specifications. The `update_pcf()` method commits changes to the PCF configuration file.

3. **System Deployment:** After setting configuration parameters, `update_config()` tears down and redeploys containers with the new configuration. Following this, `run_background_nodes()` initializes the UE and UPF gateway codes whose status can be monitored using `is_update _config_complete()` and `is_run_background_nodes_complete()` methods.

4.3.2 Data Transmission Flow

Once the system is initialized, data transmission occurs through well-defined pathways that handle different types of traffic. The API supports two main transmission scenarios:

1. **Sensor Data Transmission:**
   - JSON-formatted sensor data is sent using `send_data()` with appropriate endpoint
   - Data automatically includes timestamps for metrics collection
   - Multiple UEs can send data simultaneously using port offsets
   - SensorMetricsCalculator tracks performance metrics specifically for sensor data

2. **Video Stream Transmission:**
   - Binary frame data is transmitted using the same `send_data()` method
   - Frames are automatically sequenced and tracked for loss detection
   - NetworkMetricsCalculator monitors streaming performance metrics
   - Frame timing and delivery statistics are maintained

4.3.3 Data Reception and Metrics Flow

The API incorporates robust mechanisms for data reception and metrics collection, ensuring seamless integration with transmission operations.

1. **Data Reception:** The `receive_data()` method provides an automated framework for processing both JSON and binary data formats. The content type is dynamically handled, ensuring compatibility with different data types. For video streams, the method verifies the completeness of frame sequences to maintain data integrity. Similarly, for sensor data, readings are systematically organized by sensor ID for efficient and accurate data management.

2. **Performance Monitoring:** Metrics collection is integrated into the send and receive operations, enabling real-time monitoring of network and application performance. Key network metrics, such as throughput, latency, and jitter, are automatically calculated to evaluate transmission quality. Video-related metrics, including frame rates, delivery times, and loss statistics, are tracked to ensure reliable streaming performance. For sensor applications, the system monitors reading rates and compiles detailed statistics for each sensor, providing comprehensive insights into data flow efficiency.

3. **Analysis and Reporting:** The API supports real-time retrieval of metrics through dedicated methods such as `get_metrics()` and `get_sensor_metrics()`, offering immediate access to performance data. Additionally, the integration with Wireshark provides advanced capabilities for packet-level analysis. To maintain an updated and clean state, performance data can be reset using the relevant reset methods, ensuring the system remains adaptable to dynamic operational requirements.

As shown in Figure 4.3, these flows create a comprehensive system for managing 5G network operations. The API handles the complexity of network configuration and data transmission while providing detailed performance monitoring capabilities. This enables applications to focus on their specific requirements while ensuring proper network operation and performance tracking.

4.4 System Considerations

The development of a 5G network management API demands crucial attention to system-level considerations to ensure reliability, efficient resource utilization, and maintainable code structure. This section explores key aspects such as error handling, concurrent operations, performance optimization, and scalability.

4.4.1 Error Management Framework

A robust error management system is essential for maintaining network stability and providing clear feedback when issues occur. Our implementation employs a hierarchical exception system,

which effectively addresses different types of failures and provides detailed error information to applications. This framework is built on three major error classes, each tailored to specific aspects of system operations.

- **Configuration Errors**: The `ConfigurationError` class address issues arising from system setup and configuration. Common scenarios include missing or invalid configuration files, failed service initialization, problems with container deployment, and invalid session modifications. To aid troubleshooting, these errors include optional fields for specifying the invalid values encountered and the allowed values for each field. This detailed feedback streamlines the resolution of configuration-related problems, particularly in complex application development environments.

  Communication Errors: The `CommunicationError` class focuses on network-related failures during operation. These errors track the specific endpoint where communication failed, enabling precise identification of the problem source. Typical examples include connectivity issues with UE or UPF interfaces, failed data transmission attempts, inaccessible API endpoints, and background service communication failures.

- **Validation Errors:** The `ValidationError` class ensures compliance with 5G specifications and system constraints, providing detailed context for each violation. For example, errors specify the field name that failed validation, the invalid value provided, and a list of acceptable values. Typical validation scenarios include verifying QoS indices and parameters, AMBR values and units, ARP levels, and flow directions in PCC rules. These detailed insights into failures enhance system robustness by preventing mis-configurations that could compromise network operations.

4.4.2 Concurrent Operation Management

In a 5G network environment, multiple operations need to happen simultaneously—data needs to be transmitted, metrics need to be collected, and network services need to be monitored. This creates two key challenges: how to ensure that these parallel operations don't interfere with each other, and how to prevent data corruption when multiple processes try to access the same information. To address these challenges, our implementation uses several strategies:

- **Safe Metrics Collection:** When measuring network performance, there is a significant risk of metrics data corruption due to simultaneous updates from different operations. To counter this, thread-safe metrics collection is implemented using thread locks (`threading.Lock`) in the `NetworkMetricsCalculator` and `SensorMetricsCalculator` classes. Additionally, fixed-size circular buffers (`collections.deque`) are employed to maintain

a recent history of measurements, ensuring efficient memory usage and quick access to relevant data. Atomic operations are used for updating metrics during send and receive operations, thereby preventing any conflicts or data inconsistencies.

- **Background Process Management:** Monitoring the state of UE and UPF gateway services without blocking primary operations is a critical requirement. This is achieved through asynchronous process monitoring implemented using `ThreadPoolExecutor`, which allows for the execution of non-blocking tasks in the background. A dedicated monitoring thread is used to check the gateway endpoint availability. Non-blocking status updates are managed using the `_background_process_status` dictionary, ensuring that monitoring activities do not interfere with the core data transmission or other API functionalities.

- **Packet ID Management:** Handling multiple data streams at the same time requires a system to assign unique IDs to packets without errors. The API solves this by using a thread-safe lock mechanism and atomic increment operations to generate unique IDs. This ensures shared counters are accessed safely, even when several threads are working together. By doing so, the system avoids issues like packet mis-identifications or losses during transmission.

4.4.3 Performance Optimization

Managing a 5G network involves handling large amounts of data and many simultaneous operations. The challenge is to maintain high performance without consuming excessive system resources. Our implementation focuses on two key areas:

- **Efficient State Management:** The API employs periodic monitoring of network interfaces to track updates at regular intervals. The `_update_interfaces()` function checks the current state of UE and UPF gateway interfaces, capturing significant changes without constant monitoring. This approach ensures a balance between monitoring accuracy and reducing system resource usage.

- **Smart Memory Usage:** For real-time analysis, we need to track recent network performance without storing unlimited historical data. The proposed framework uses circular buffers that maintain a fixed memory footprint It also maintains statistical calculations over the most recent time window for accurate performance measurements. For example, when tracking video streaming performance, we maintain a rolling window of the most recent frame timings (default 100 frames), ensuring constant memory usage remains for meaningful performance analysis.

Chapter 5

Use-case Development

The `open5gsapi` package, as detailed in Chapter 4, establishes a robust foundation for the seamless implementation of advanced 5G network applications by abstracting the complexities of system-level operations. This abstraction layer consolidates critical functionalities, including Docker operations, tunnel configurations, and policy management, into a cohesive framework. By doing so, it enables developers to concentrate on application-specific requirements without being encumbered by the intricacies of underlying system mechanisms. The API's comprehensive design facilitates a streamlined approach to PCF configuration, UE simulation, and performance monitoring. This, in turn, allows for development of sophisticated web-based interfaces and network-driven applications. The architectural framework proves particularly impactful in demonstrating three distinct use cases, each highlighting unique facets of 5G network slicing: an eMBB-based video streaming service, a URLLC-based remote surgery simulator, and a mMTC-based sensor network application.

5.1 eMBB-based Video Streaming Application

The eMBB capabilities of our virtual 5G network setup are demonstrated through an interactive video streaming application designed to handle high-bandwidth data transmission. As shown in Figure 5.1, the implementation features a client-server architecture capable of managing multiple video streams transmitted over the 5G network infrastructure. Here, we consider only a single UE connection to the core network which acts as a video server. This application serves as a practical test case for bandwidth-intensive scenarios, with applications ranging from high-definition video streaming to remote surveillance and multimedia content delivery. It also incorporates real-time adaptation to network conditions and provides comprehensive performance analysis leveraging the metric-calculation functionalities of the `open5gsapi` package. A web-based interface facilitates dynamic upload and management of video content, while video data is streamed through the

configured eMBB slice. The system delivers detailed metrics on key performance indicators (KPIs), including throughput, latency, jitter, and frame delivery statistics.

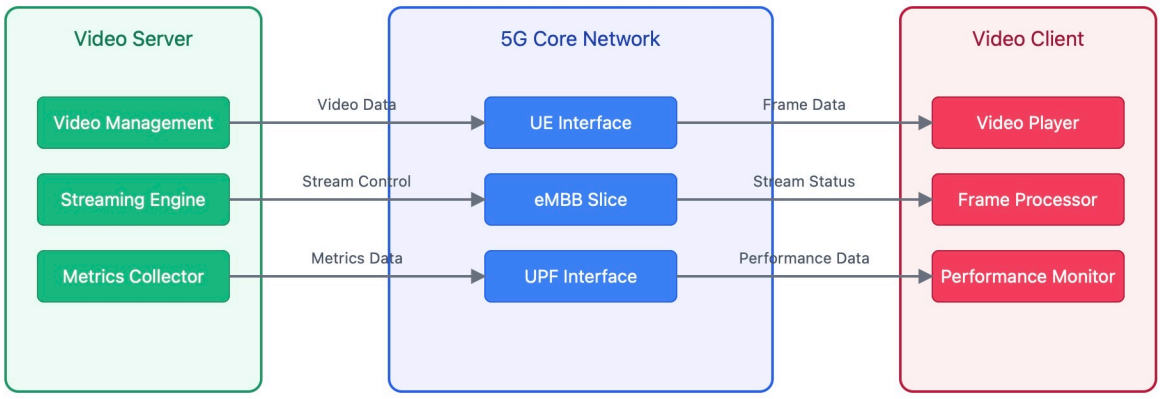


FIGURE 5.1: Architecture of the eMBB Video Streaming Application

5.1.1 Application Workflow

The video streaming application follows a workflow that coordinates interactions between the client, server, and the Open5GS core network, as shown in Figure 5.2. This workflow is divided into four main phases - network configuration, catalogue synchronization, video streaming, and performance monitoring. Each phase is designed to build on the preceding ones, ensuring smooth operation and effective communication across all components.

The sequence begins with the network configuration phase through an interface that enables control over core network configuration. This interface, implemented using a reactive web framework, exposes PCF parameters through a structured configuration form. The system presents essential configuration options such as session type selection, AMBR parameter adjustment for both uplink and downlink channels, QoS index specification, and ARP parameter configuration. Upon parameter modification, the interface triggers a three-step deployment sequence: PCF YAML configuration update, container redeployment, and background process initialization, ensuring synchronized operation between network parameters and application requirements. The progress of these steps is tracked using `open5gsapi` package methods and displayed as an animated progress bar with success/error logs displayed at each step.

Following successful network initialization, the UE-side interface directs the user to a video server interface that synchronizes the video catalog from local files whilst also providing the option to upload video data (within allowed formats of MP4, AVI, MKV, and MOV) from the system. This interface implements content management capabilities while maintaining real-time system status monitoring. The server component maintains a `VideoManager` class that processes uploaded content, extracting video metadata such as frame count, frame rate, resolution, and duration.

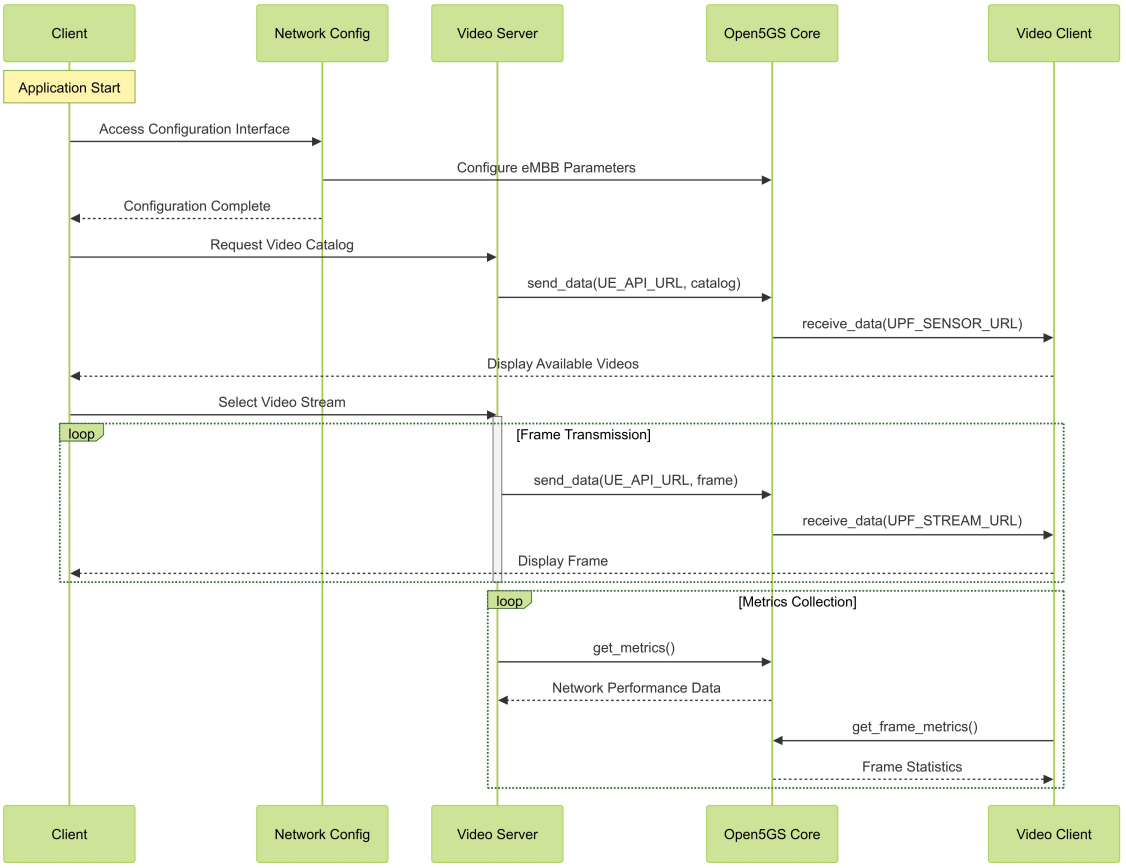


FIGURE 5.2: eMBB Application Sequence Flow

This metadata is transmitted through the 5G tunnel, beginning at the UE gateway endpoint (`open5gs.ue("send")`) and reaching the client through the UPF gateway's sensor endpoint (`open5gs.upf("receive/sensor")`).

The streaming phase activates upon video selection on the client interface, establishing a continuous frame transmission cycle. This interface not only handles video playback controls but also implements real-time performance monitoring capabilities. The server processes video frames, applying necessary optimizations before transmission through the UE endpoint. Frame delivery occurs through a bidirectional communication pathway facilitated by the open5gsapi's `send_data()` and `receive_data()` methods, while the interface continuously updates performance metrics including current throughput, network jitter, and frame delivery statistics.

5.1.2 Implementation Details

The video streaming implementation focuses on the efficient handling of high-definition video content through the 5G network infrastructure. The streaming functionality is implemented through a frame-by-frame processing and transmission pipeline as detailed in Algorithm 1.

Algorithm 1 Video Frame Streaming

Require: Video file path P, Maximum width W_{max}, Frame interval Δt

$cap \leftarrow InitializeCapture(P)$
$n \leftarrow 0$ ▷ Sequence counter
$w_{orig} \leftarrow cap.width$
$h_{orig} \leftarrow cap.height$
if $w_{orig} > W_{max}$ **then**
    $scale \leftarrow W_{max}/w_{orig}$
    $w_{target} \leftarrow W_{max}$
    $h_{target} \leftarrow h_{orig} \times scale$
else
    $w_{target} \leftarrow w_{orig}$
    $h_{target} \leftarrow h_{orig}$
end if
while cap is open **do**
    $(status, F) \leftarrow cap.read()$
    if not $status$ **then**
        $SendEndSignal()$
        break
    end if
    if $w_{orig} > W_{max}$ **then**
        $F \leftarrow Resize(F, w_{target}, h_{target})$
    end if
    $B \leftarrow JPEG.encode(F, quality = 80)$
    $n \leftarrow n + 1$
    $packet \leftarrow "FRAME:" \parallel n \parallel B$ ▷ Concatenate header and data
    $Transmit(packet)$
    $Sleep(\Delta t)$
end while
$cap.close()$

During implementation, the following parameters were established for optimal streaming performance:

- Maximum frame width (W_{max}) = 1280 pixels, chosen to balance quality and network load
- JPEG encoding quality = 80 (reduced to 60 if frame size exceeds 65KB)
- Frame interval (Δt) = $1/fps$ where fps is the source video frame rate
- Initial buffer size = 1MB with dynamic adjustment based on frame size

At the receiving end (reference implementation in Appendix B.1), frames are processed through sequential header detection and validation. The receiving process:

1. Detects frame headers for proper frame sequencing
2. Processes end-of-stream messages for stream termination handling

3. Maintains frame statistics including loss detection through sequence tracking
4. Updates interface metrics in real-time using collected performance data

Error handling incorporates sequence number tracking for frame loss detection and automatic stream termination on critical failures, with comprehensive metrics collection at both transmission and reception endpoints.

5.1.3 Interface Design

The video streaming application provides a streamlined web interface that enables network configuration, content management, and playback control. The network configuration interface (Figure D.1) provides controls for network slice parameters through an organized form layout:

- IP type selection dropdown (IPv4, IPv6, IPv4v6)
- AMBR value and unit configuration for uplink/downlink
- QoS index specification
- ARP settings including priority and pre-emption controls

The server-side interface (Figure D.2) implements video catalogue management with a video upload functionality through local file selection, real-time indicators that show the status of gateway codes (i.e., availability of 5G tunnel send/receive endpoints), video catalogue display showing duration, resolution, frame rate and file size and a manual refresh option for catalogue updates. The client interface (Figure D.3) provides a comprehensive video streaming environment featuring:

- Grid display of available videos with metadata
- Stream initiation controls for each video
- UE and UPF gateway availability indicators
- Video playback interface with pause/resume/stop-stream options
- Real-time metrics display showing:
  - Current FPS, bitrate, jitter
  - Network performance statistics
  - Stream information (resolution, total frames received)
  - Aggregated metrics (average FPS, bitrate and frame loss, peak bitrate and jitter)

5.1.4 Performance Analysis

For eMBB testbed evaluation, we consider a single UE connection and the PCF network parameters as described in Table 5.1:

TABLE 5.1: Video Streaming - PCF Parameters

Parameter	**Value**	**Source**
IP Type	IPv4	-
5QCI Value	4 (Non-conversational Video)	[40]
AMBR (Downlink)	1 Gbps	[41]
AMBR (Uplink)	500 Mbps	[41]
ARP Level	2	[42]
Pre-emption Capability	Disabled	[43]
Pre-emption Vulnerability	Enabled	[43]

Table 5.2 presents the key performance metrics obtained during testing, including bitrate utilization, frame rates, and network stability indicators. While these measurements represent specific test cases of streaming 4 different video resolution types - Standard VGA (640x480p), Full-HD (1920x1080p), 4K UHD (3840X2160P), and DCI 4K (4096x2160p), they demonstrate characteristic trends in bitrate utilization, frame rates, and network behaviour as resolution scales in the developed testbed.

TABLE 5.2: eMBB Application - Network Performance Metrics (representative)

Resolution (Frames)	**Average Bitrate**	**Average FPS**	**Peak Bitrate**	**Peak Jitter**	**Frame Loss**
640×840 (145)	6.96 Mbps	19.2	7.43 Mbps	58.0 ms	0
1920×1080 (383)	6.16 Mbps	13.6	6.66 Mbps	87.2 ms	0
3840×2160 (207)	3.27 Mbps	9.0	4.12 Mbps	120.6 ms	0
4096×2160 (169)	4.19 Mbps	7.7	4.46 Mbps	168.1 ms	0

5.2 URLLC-based Remote Surgery Simulator Application

The URLLC capabilities of this virtual 5G network setup are demonstrated through a remote surgery simulation system that emphasizes precise real-time control and monitoring of robotic

surgical instruments. This application serves as a critical test case for evaluating network performance in scenarios where reliability and low latency are paramount, such as telesurgery, industrial automation, and autonomous vehicle control. As shown in Figure 5.3, the implementation simulates a robotic surgery environment where surgical instruments transmit real-time telemetry data including spatial coordinates, force feedback, and vital sign measurements through the 5G network infrastructure. By utilizing the Locust [44] load testing framework, the system generates realistic surgical operation patterns while maintaining comprehensive monitoring of network performance metrics. This implementation is particularly significant for validating slice configurations under stringent latency and reliability requirements as per URLLC standards while providing insights into the practical feasibility of remote surgical applications over 5G networks.

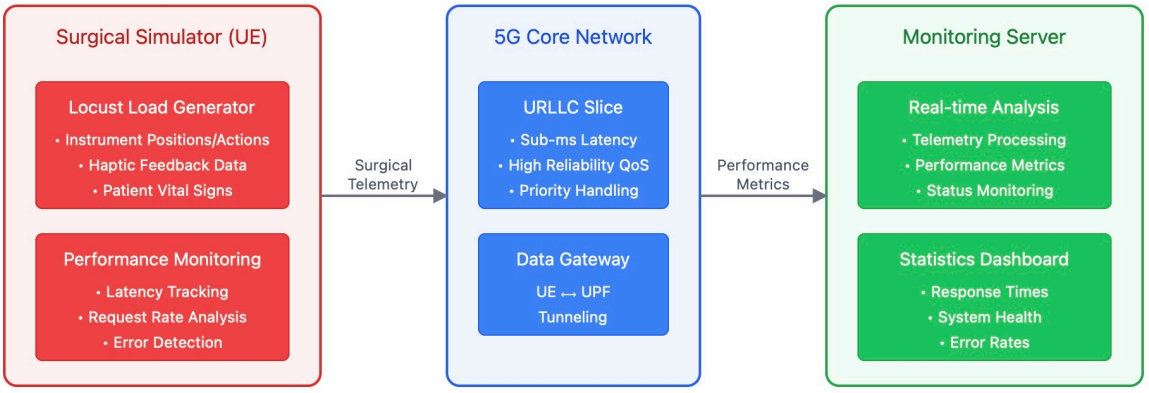


FIGURE 5.3: Architecture of the URLLC Remote Surgery Application

5.2.1 Load Testing Framework Integration

The implementation incorporates Locust, an open-source load testing framework, to simulate realistic surgical instrument data transmission patterns. Through Locust's task-based architecture, the system simulates regular telemetry updates with wait times between 1-3 seconds, reflecting realistic surgical operation timescales. Each telemetry packet is transmitted through the 5G tunnel using the `open5gsapi` framework's data transmission capabilities, enabling systematic testing of the network's ability to handle precise surgical telemetry data while maintaining ultra-low latency requirements.

5.2.2 Application Workflow

The remote surgery simulation workflow comprises three key phases: initial configuration, traffic generation through Locust, and performance monitoring, as illustrated in Figure 5.4.

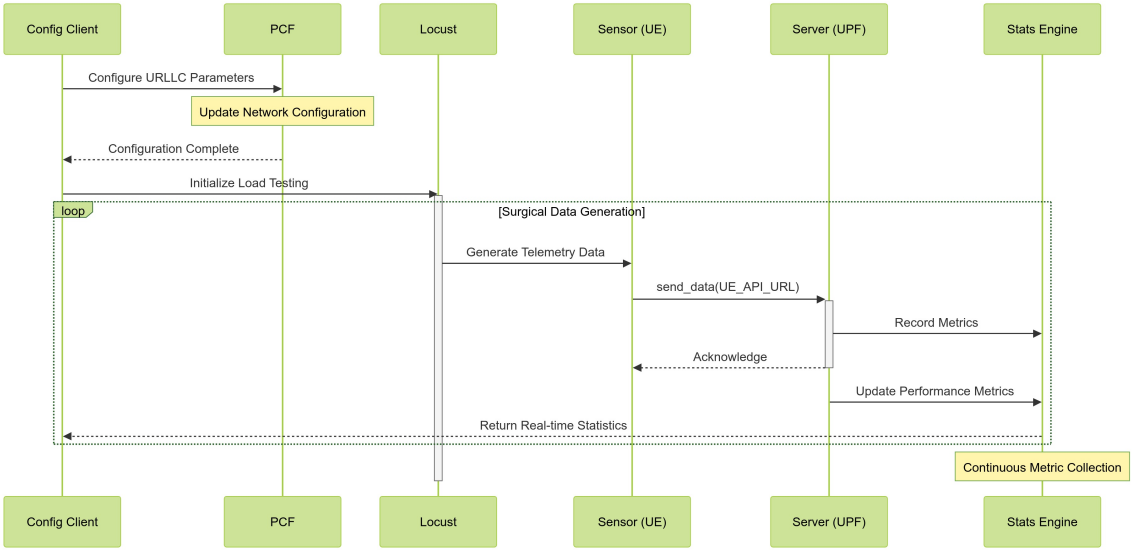


FIGURE 5.4: URLLC Application Sequence Flow

During the configuration phase, the system first establishes URLLC slice parameters through the web interface's PCF configuration. The application then generates a Locustfile that defines the test parameters for surgical simulation. This configuration specifies:

- The UPF endpoint (10.10.0.112:8081) for telemetry reception
- Task parameters for surgical data generation with 1-3 second intervals
- The local send_data endpoint (localhost:8003/send_data) interfacing with `open5gsapi`

Test execution is managed through Locust's web interface, accessible at localhost:8089. Once initiated, Locust generates surgical telemetry data and transmits it directly to the UPF endpoint via the 5G network tunnel. This transmission is handled by the send_data application route which internally uses `open5gsapi` methods to establish the tunnel connection. By specifying the UPF IP as the target host in Locust, we ensure that the telemetry data traverses through the 5G core network.

The server implementation processes incoming telemetry through the receive_data route which handles two primary functions. First, it retrieves the latest surgical telemetry from the UPF endpoint for visualization. Second, it collects performance statistics directly from Locust's API endpoint (localhost:8089/stats/requests). The server also implements continuous metric collection through periodic polling of Locust's statistics. These metrics, including response times, request rates, and failure counts, are accessed directly via Locust's REST API.

TABLE 5.3: Remote Surgery Telemetry Parameters and Ranges

Component	**Parameter**	**Range/Values**
Instrument Position	X-coordinate	[-100, 100] mm
	Y-coordinate	[-100, 100] mm
	Z-coordinate	[0, 150] mm
Instrument Status	Tool type	scalpel, forceps, needle_holder, cautery
	Action state	cutting, grasping, idle, retracting
	Applied force	[0, 5] N
	Tool angle	[0, 360] degrees
Haptic Feedback	Tissue resistance	[0, 100] units
	Vibration intensity	[0, 50] Hz
	Local temperature	[35, 40] °C
Vital Signs	Heart rate	[60, 100] bpm
	Blood pressure	[110-140]/[60-90] mmHg
	Oxygen saturation	[95, 100]%
	Blood loss estimation	[0, 100] mL

5.2.3 Interface Design

The application provides several web interfaces for configuration, monitoring, and control of the surgical simulation. The simulation is conducted with a single UE and network parameters as described in Table 5.4 to establish baseline performance metrics for surgical telemetry transmission. These interfaces are implemented using Flask's template engine with responsive design elements.

TABLE 5.4: Configuration Parameters for Surgical Simulation [45]

Parameter	**Value**
IP Type	IPv4
5QCI Value	3
AMBR (Uplink)	20 Kbps
AMBR (Downlink)	20 Kbps
ARP Level	1
Pre-emption Capability	Enabled
Pre-emption Vulnerability	Enabled

The configuration interface (Figure D.5) presents network slice parameters through an intuitive form layout, same as the eMBB application:

- IP type selection dropdown (IPv4, IPv6, IPv4v6)
- AMBR settings with value and unit selection for both uplink and downlink
- QoS index specification
- ARP parameter controls including priority level and pre-emption settings

Load testing is managed through Locust's web interface (Figure D.6), where users can configure the number of concurrent users, user spawn rate and target host specification (set by default to the UPF gateway endpoint http://10.10.0.112:8081).

The performance metrics collected through Locust (Figures 5.5 and 5.6) demonstrate the system's capability to maintain consistent low-latency transmission of surgical telemetry data. As shown in Figure 5.5, the system maintains a steady request rate of approximately 0.5-0.6 requests per second, with response times consistently below 12 milliseconds. The detailed statistics in Figure 5.6 reveal zero failures across all requests, with an average response time of 9.32 milliseconds and a payload size of 773.72 bytes, indicating reliable and efficient data transmission suitable for surgical telemetry applications. The narrow spread between minimum (8ms) and maximum (12ms) response times suggests highly stable network performance, which is crucial for maintaining precise control in surgical operations.

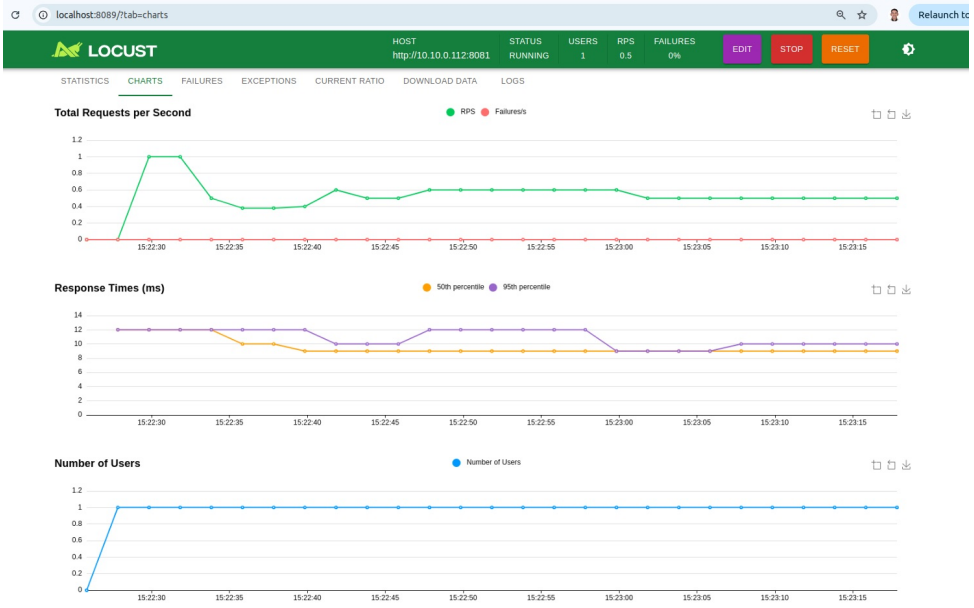


FIGURE 5.5: Locust Performance Charts

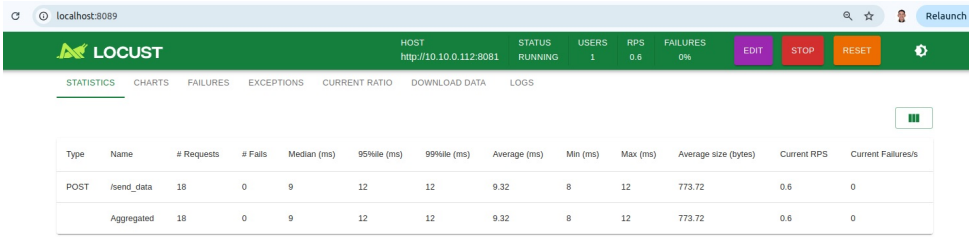


FIGURE 5.6: Statistics from Locust Load Testing

TABLE 5.5: Locust Performance Metrics for Surgical Telemetry Transmission

Metric	**Value**
Request Rate	0.5-0.6 RPS
Median Response Time	9 ms
95th Percentile Response Time	12 ms
Average Response Time	9.32 ms
Minimum Response Time	8 ms
Maximum Response Time	12 ms
Average Payload Size	773.72 bytes
Failure Rate	0%

The monitoring dashboard (Figure D.7 and Figure D.8) provides comprehensive real-time visualization of surgical telemetry and network performance:

- Surgical instrument status and positioning
- Patient vital signs monitoring
- Network performance metrics including throughput and response times
- Response time distribution graphs
- Historical data table for recent operations

Additional features of the monitoring interface include Wireshark launch capability for packet-level analysis (provided by `open5gsapi` methods), traffic status indicators and error rate monitoring. The interface is designed to provide immediate feedback on both surgical operations and network performance, enabling real-time assessment of the URLLC slice capabilities under test conditions.

5.3 mMTC-based Sensor Network Implementation

The mMTC capabilities of this virtual 5G network are demonstrated through the implementation of an industrial sensor network that simulates diverse IoT monitoring scenarios. This application simulates an industrial environment where multiple sensor types - environmental, industrial process, critical safety, and system status monitors - transmit data through the 5G infrastructure at different intervals and priorities, as depicted in Figure 5.7. Using the `open5gsapi` framework and a custom traffic generator class called `UEDataGenerator`, the system simulates multiple concurrent sensor units, each maintaining independent data generation patterns based on their category.

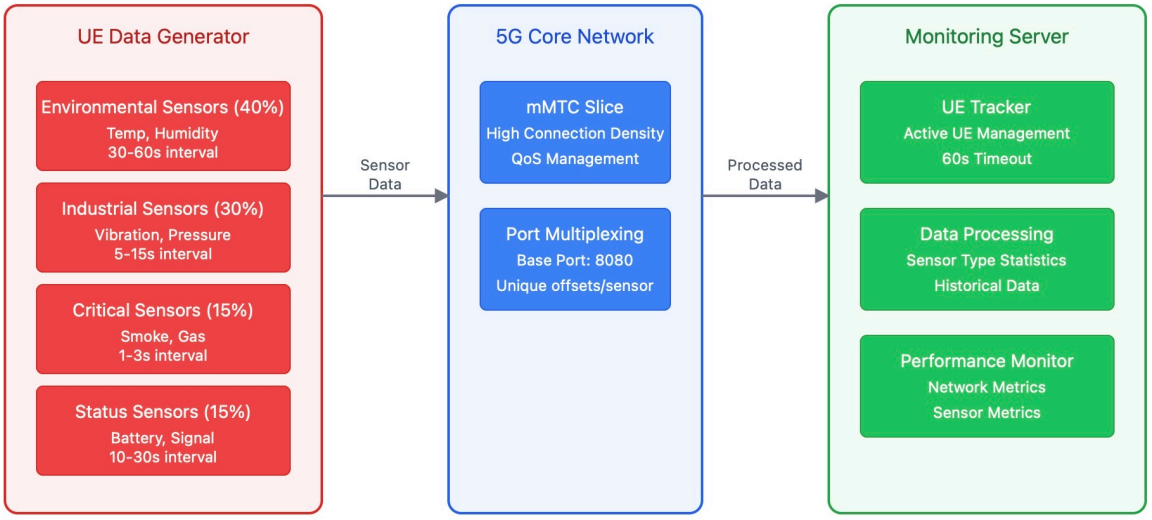


FIGURE 5.7: Architecture of the mMTC Remote Surgery Application

5.3.1 Application Workflow

The implemented mMTC application serves as a comprehensive testbed for evaluating the capability of 5G networks to handle massive concurrent IoT device connections with diverse communication patterns. This implementation specifically addresses the growing need to validate network performance under scenarios that simulate real-world industrial IoT deployments, where hundreds or thousands of devices must communicate reliably and efficiently through a single network infrastructure.

The application architecture comprises three primary components: a custom traffic generator that simulates multiple concurrent IoT devices, a server component that processes and analyzes the received data, and the underlying Open5GS network infrastructure that facilitates communication between these components. The traffic generator implements a sophisticated device simulation

framework that can spawn and manage multiple virtual UEs, each operating independently with its own communication patterns and priorities.

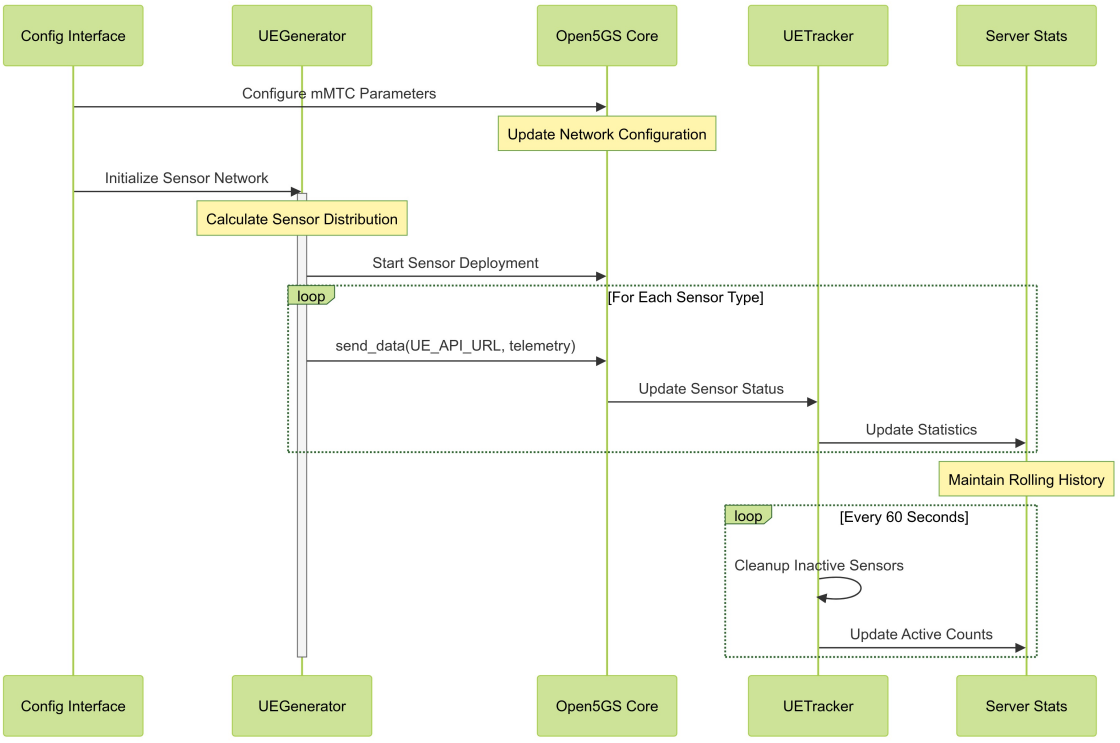


FIGURE 5.8: mMTC Application Sequence Flow

Central to the implementation is the `UEDataGenerator` class, which provides control over the virtual device ecosystem. This component enables systematic testing of network performance under varying loads and communication patterns. The application's workflow is divided into four key phases: network configuration, device initialization, continuous data transmission, and performance monitoring. Each phase focuses on testing specific mMTC capabilities, offering insights into the network's performance in massive IoT scenarios. The workflow coordinates interactions between multiple sensor units, server components, and the Open5GS core network, as shown in Figure 5.8.

5.3.2 Implementation Details

The implementation demonstrates a practical approach to simulating massive IoT connectivity in a 5G network environment. At its core, it utilizes Open5GS's virtualization capabilities to simulate multiple UE devices (`uesimtun` interfaces), each representing different types of industrial sensors. When the system initializes, it creates the specified number of UE interfaces, each assigned sequential IP addresses starting from 10.45.0.2. For each UE interface, a corresponding gateway endpoint is established at 10.10.0.132 with incrementing port numbers starting from 8080 (e.g., 8080, 8081, 8082, etc.).

5.3.2.1 Sensor Specifications and Behavior

The application simulates an industrial IoT environment by dividing sensors into four categories, each with specific monitoring parameters, update intervals, and behavioural modes, thus implementing a custom traffic generator tool to assess the abilities of the constricted virtual network in complex settings. (also refer Appendix B.3 for complete implementation). The comprehensive specification of these sensors is detailed in Table 5.6.

TABLE 5.6: Comprehensive Sensor Specifications and Behavior

Sensor Type	**Distribution**	**Data Ranges**	**Base Update Interval**	**Critical Condition**	**Burst Rate**
Environmental	40%	Temperature: 18-28°C Humidity: 30-70%	30-60s	Temp > 28°C or Humidity > 70%	15s
Industrial	30%	Vibration: 0-100 Pressure: 980-1020 mb	5-15s	Vibration > 80 or Pressure deviation > 20mb	1s
Critical Safety	15%	Smoke Level: 0-50 Gas Concentration: 0-100	1-3s	Smoke > 30 or Gas > 70	100ms
Status	15%	Battery: 0-100% Signal: -100 to -50 dBm	10-30s	Battery < 20% or Signal < -90 dBm	2s

This application uses a staggered transmission mechanism to effectively manage network traffic in real-world IoT deployments. As different sensor types operate at varying frequencies, the system assigns each sensor a unique transmission schedule consisting of a random initialization offset and an independently randomized transmission interval within its specified range. Furthermore, a 500ms delay between consecutive sensor startups ensures smooth initialization of the network. This distribution of transmissions naturally helps manage traffic flow, particularly in scenarios where multiple sensors of the same type might need to transmit data simultaneously during critical events.

5.3.2.2 Data Transmission and Retrieval Mechanism

The data transmission process follows a structured path through the virtual network:

1. Each simulated sensor generates data according to its type-specific parameters and current operating mode
2. Data is transmitted over individual URLs (10.10.0.132:808X) exposed by UE gateway code, through the `/send` endpoint
3. The UPF receives data through a dedicated sensor endpoint (`/receive/sensor`) over the 10.10.0.112:8081 URL.

The transmission payload includes essential tracking information such as unique sensor and UE identifiers, source IP address and port, timestamp, sequence number and sensor-specific measurements. The server maintains basic tracking of active UEs through a dictionary structure that records last-seen timestamps for each UE, current count of UEs by sensor type and historical data with a maximum of 100 entries per UE. A 60-second timeout mechanism is also implemented to automatically remove inactive UEs from the tracking system.

5.3.3 Interface Design

5.3.3.1 Network Configuration Interface

The configuration interface, shown in Figure D.9, enables users to:

- Set the number of UEs with a real-time preview of their distribution across sensor categories
- View estimated data generation rates for each sensor type
- Configure network session parameters including:
  - IP type selection dropdown (IPv4, IPv6, IPv4v6)
  - AMBR settings with value and unit selection for both uplink and downlink
  - QoS index specification
  - ARP parameter controls including priority level and pre-emption settings

5.3.3.2 Sensor Control Panel

The sensor control interface, shown in Figure D.10, allows users to:

- Modify operating ranges for each sensor type: Environmental sensors (temperature and humidity thresholds), Industrial sensors (vibration and pressure ranges), Critical sensors (smoke level and gas concentration limits), status sensors (battery drain rate and signal strength boundaries)
- Set individual update intervals for each sensor category
- Monitor real-time traffic generation metrics including active UEs, messages sent, current transmission rates, and error counts
- View the distribution statistics of different sensor categories and the number of messages sent by each of them.

5.3.3.3 Server-side Monitoring Dashboard

The server-side monitoring dashboard, illustrated in Figure D.11 and Figure D.12, provides comprehensive real-time visualization of the mMTC network through four main components:

1. **Network Status Overview:** Displays critical system metrics through three panels: active UEs count with distribution across sensor types, a network health percentage with a dynamic progress indicator, and system uptime tracking operational duration. Additionally features a Wireshark launch option for detailed packet analysis. The custom-defined network health is calculated as a percentage based on current network latency:

$$\text{Health}(\%) = \min(100, \max(0, \frac{1000 - \text{latency}_{\text{ms}}}{10})) \tag{5.1}$$

   This produces a health score from 0-100% with colour indicators: green (>90%), yellow (50-90%), and red (<50%).

2. **Sensor Category Visualization:** Four interactive panels (Environmental, Industrial, Critical, Status) present real-time sensor data through dual-axis charts. Each panel shows:
   - Current readings (e.g., temperature, humidity, vibration, pressure, battery levels etc.,)
   - Active sensor count and data transmission rates
   - Time-series graphs of sensor measurements
   - Detailed listings of active sensors with their network identifiers

3. **QoS Performance Metrics:** Comprehensive network performance monitoring through four key metrics: latency (current/average/min/max in ms), throughput (in Kbps), jitter (in ms), and packet size (in bytes).

4. **Real-time Event Log:** A chronological data stream showing:

   - Timestamped sensor readings and status updates
   - Sensor identification (UE ID and IP:PORT)
   - Event types and measured values
   - Transmission status indicators
   - Filterable view options by sensor category

5.3.4 Performance Analysis

The application makes use of the `SensorMetricsCalculator` class of `open5gsapi` package to track transmission performance through 4 key metrics: latency (ms), throughput (Kbps), jitter(ms) and packet size (bytes). For the mMTC demonstration, the configuration parameters are set as shown in Table 5.7 with 6 UEs configured and distributed according to the predefined ratios (2 environmental, 2 industrial, 1 critical, and 1 status sensor). The performance of this testbed was evaluated through continuous monitoring of key network metrics over time as shown in Figures 5.9, 5.10. The measured average values of 21 ms for latency, 15.99 ms for jitter, and 165.37 Kbps for throughput indicate stable network performance suitable for mMTC applications.

TABLE 5.7: Simulation Parameters and Sources

Parameter	**Value**	**Source**
IP Type	IPv4	-
5QCI Value	82 (delay critical GBR)	[46]
AMBR (Uplink)	100 Kbps	[47]
AMBR (Downlink)	100 Kbps	[47]
ARP Level	9	[2]
Pre-emption Capability	Disabled	[48]
Pre-emption Vulnerability	Disabled	[48]

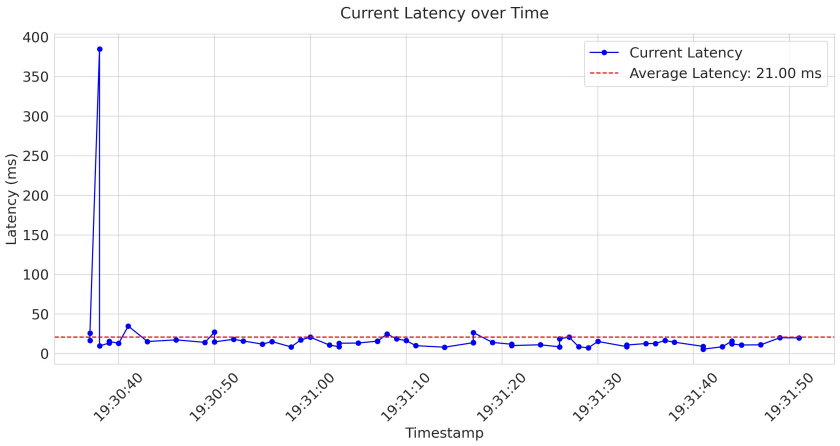


FIGURE 5.9: mMTC - Network latency over time

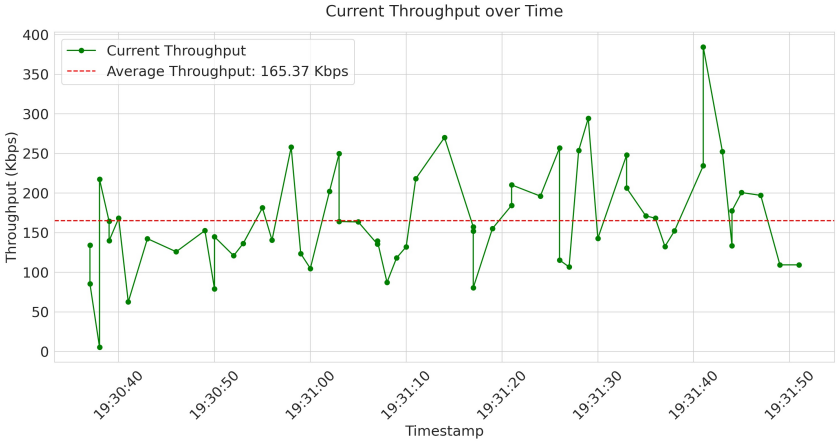


FIGURE 5.10: mMTC - Network throughput over time

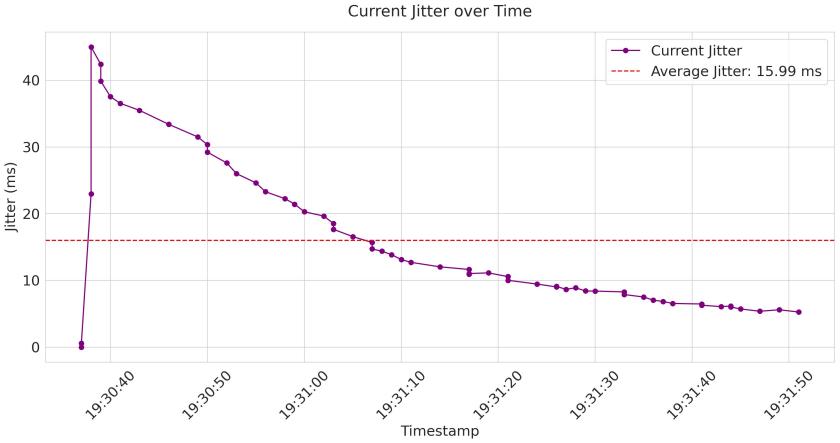


FIGURE 5.11: mMTC - Network jitter over time

Chapter 6

Conclusions

6.1 Summary of Contributions

The primary contributions of this implementation include both advancing methodologies and delivering practical solutions for virtualized 5G networks. The development of a containerized testbed environment integrating Open5GS and UERANSIM frameworks demonstrates the practical feasibility of implementing network configuration capabilities in a controlled setting. This implementation is augmented by a novel API framework that enables precise control over network parameters while maintaining comprehensive performance monitoring capabilities.

The effectiveness of this API package in isolating network-related adjustments from application development is demonstrated through three distinct use cases, encompassing the key pillars of 5G capabilities. The eMBB implementation validates the system's ability to maintain consistent high-bandwidth transmission for video streaming applications, while the URLLC implementation demonstrates the maintenance of stringent latency requirements critical for remote surgery scenarios. The mMTC implementation further validates the system's capability to manage multiple concurrent connections while maintaining consistent network quality.

A significant contribution lies in the development of accurate performance monitoring systems that provide real-time analysis of network behaviour across different slice configurations. These systems enable detailed examination of network performance characteristics while maintaining minimal overhead, providing valuable insights into the operational aspects of network slicing implementations.

6.2 Production Value and Real-World Applications

While this work utilizes a virtualized testbed environment, its findings hold significant relevance for specific real-world applications, particularly in research and development contexts. The developed environment demonstrates particular utility in educational settings, where it can serve as a practical platform for understanding the network concepts, experimenting different network configurations and developing 5G applications. This application is especially relevant given the increasing importance of practical experience in telecommunications education and the limited availability of physical 5G infrastructure for educational purposes.

In testbed development contexts, the framework provides a valuable tool for prototyping and validating new 5G applications and services. The ability to quickly reconfigure core and RAN parameters, simulate traffic conditions, and test different slice configurations enables rapid testing of new concepts before deployment on physical infrastructure. This is particularly valuable for organizations developing 5G applications and services, allowing them to validate concepts and identify potential issues in a controlled environment.

The methodologies presented in this thesis provide a foundation for understanding the practical considerations in network slice deployment, while the documented limitations and challenges offer valuable insights for future research directions. Through careful consideration of implementation challenges and comprehensive validation across multiple use cases, this work contributes to bridging the gap between theoretical network slicing concepts and practical deployment considerations, while maintaining a realistic perspective on its immediate applications in virtualized environments.

6.3 Open Issues

The current implementation, while demonstrating the feasibility of network slice management through a programmatic interface, reveals certain technical limitations that warrant further investigation. These stem from both architectural constraints and implementation complexities inherent to the virtualized setup.

6.3.1 PCF-Specific Control Interface

The present implementation confines programmatic control primarily to PCF parameters, specifically those about slice configuration and session management. While this enables basic network slice customization, it represents only a subset of potential control interfaces. The system currently supports modifications to session parameters including IP addressing schemes, QoS profiles,

and ARP settings. However, comprehensive network function control would necessitate extending the API framework to encompass other network functions such as AMF, SMF, UPF, NRF, and SMMF etc., Further, in the existing PCF configuration implementation, some domains remain outside programmatic control, such as - network repository settings, service communication proxy configurations, slice service type settings, and security edge protection parameters.

6.3.2 Hardware Constraints on UE Scalability

A significant limitation encountered during the implementation was the inability to consistently configure more than 50 UEs in the Open5GS framework due to hardware resource constraints. When attempting to configure and manage a large number of concurrent UE connections, there have been intermittent PLMN selection failures. This limitation poses significant challenges for emulating large-scale network deployments and testing the scalability of network slicing implementations. Further investigation is needed to determine the exact hardware specifications and resource allocation strategies required to support a higher number of concurrent UE connections in the Open5GS framework.

6.3.3 Subscriber Database Initialization Constraints

The subscriber database initialization process manifests a significant architectural limitation in its current implementation, particularly concerning the programmatic control of UE parameters. The initialization mechanism, primarily governed by the `open5gs-dbctl` configuration utility, implements a deterministic parameter assignment protocol that establishes inflexible default values during the subscriber registration phase. The `open5gs-dbctl` utility, functioning as the primary database configuration interface, utilizes MongoDB operations to establish subscriber profiles with predetermined parameter sets as:

Table 6.1: Configuration Parameters

Parameter	SST	PDU Session Type	QoS	Priority Level	Pre-emption Capability	Pre-emption Vulnerability
Value	1	IPv4v6 (type 3)	9	8	1 (disabled)	2 (enabled)

These rigid parameter settings presents a constraint in scenarios requiring dynamic subscriber profile management. Particularly noteworthy is the disconnect between PCF configuration modifications and subscriber database states - while the API framework enables PCF parameter adjustments, these modifications do not reflect the corresponding subscriber database entries.

6.4 Future Research Directions

The findings from this work highlight several areas for future investigation in virtualized 5G networks, primarily focusing on data transmission performance improvements and enhanced network function control. Based on our implementation results, the following improvements are proposed:

- **eMBB application:** Exploring emerging video coding standards like VVC (Versatile Video Coding) [49] and deep learning-based compression techniques is an interesting research direction that could enabling more efficient bandwidth utilization. Further, implementation of content-aware networking principles using artificial intelligence-driven QoS optimization could enhance streaming performance for emerging applications like extended reality (XR).
- **URLLC application:** Future development could explore integration with hardware-accelerated processing or implementation of time-sensitive networking (TSN) [50] principles to enhance ultra-reliable communication, particularly suited for mission-critical use cases demanding strict latency guarantees.
- **mMTC application:** Implementation of adaptive resource allocation with per-device QoS prioritization is a primary objective in optimizing concurrent connection handling, which demands comprehensive network control requirements as detailed in further points.

Network configuration control improvements could focus on developing a comprehensive framework that better aligns with production environment requirements:

- Implementation of UE-UPF handshake mechanisms and bidirectional data transfers with dedicated endpoints for process log transmission
- Development of extended API interfaces for programmatic control of subscriber parameters and additional network functions beyond PCF
- Integration of recurring functionalities into developer-focused macros

These architectural enhancements will strengthen autonomous network management capabilities and enable advanced service differentiation in virtualized 5G environments. Additional research priorities include developing security frameworks for virtualized networks, focusing on network slice isolation and containerized function security. The integration of edge computing capabilities and dynamic slice allocation also presents opportunities for performance optimization in developing advanced 5G use cases.

Appendix A

Gateway Codes and Network Function Configurations

A.1 Policy Configuration Options in PCF

```
policy:
- plmn_id:
   mcc: 999
   mnc: 70
 slice:
   - sst: 1  # 1,2,3,4
     default_indicator: true
     session:
       - name: internet
         type: 3  # 1:IPv4, 2:IPv6, 3:IPv4v6
         ambr:
           downlink:
             value: 1
             unit: 3  # 0:bps, 1:Kbps, 2:Mbps, 3:Gbps, 4:Tbps
           uplink:
             value: 1
             unit: 3
         qos:
           index: 9  # 1, 2, 3, 4, 65, 66, 67, 75, 71, 72, 73, 74, 76, 5, 6, 7, 8, 9, 69, 70, 79, 80, 82, 83, 84, 85, 86
           arp:
             priority_level: 8  # 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
             pre_emption_vulnerability: 1  # 1: Disabled, 2:Enabled
             pre_emption_capability: 1  # 1: Disabled, 2:Enabled
       - name: ims
         type: 3  # 1:IPv4, 2:IPv6, 3:IPv4v6
         ambr:
```

```
27          downlink:
28            value: 1
29            unit: 3  # 0:bps, 1:Kbps, 2:Mbps, 3:Gbps, 4:Tbps
30          uplink:
31            value: 1
32            unit: 3  # 0:bps, 1:Kbps, 2:Mbps, 3:Gbps, 4:Tbps
33        qos:
34          index: 5  # 1, 2, 3, 4, 65, 66, 67, 75, 71, 72, 73, 74, 76, 5, 6, 7, 8,
    9, 69, 70, 79, 80, 82, 83, 84, 85, 86
35          arp:
36            priority_level: 1  # 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
37            pre_emption_vulnerability: 1  # 1: Disabled, 2:Enabled
38            pre_emption_capability: 1  # 1: Disabled, 2:Enabled
39        pcc_rule:
40          - qos:
41              index: 1  # 1, 2, 3, 4, 65, 66, 67, 75, 71, 72, 73, 74, 76, 5, 6, 7,
    8, 9, 69, 70, 79, 80, 82, 83, 84, 85, 86
42              arp:
43                priority_level: 1  # 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
     15
44                pre_emption_vulnerability: 1   # 1: Disabled, 2:Enabled
45                pre_emption_capability: 1   # 1: Disabled, 2:Enabled
46              mbr:
47                downlink:
48                  value: 82
49                  unit: 1  # 0:bps, 1:Kbps, 2:Mbps, 3:Gbps, 4:Tbps
50                uplink:
51                  value: 82
52                  unit: 1  # 0:bps, 1:Kbps, 2:Mbps, 3:Gbps, 4:Tbps
53              gbr:
54                downlink:
55                  value: 82
56                  unit: 1  # 0:bps, 1:Kbps, 2:Mbps, 3:Gbps, 4:Tbps
57                uplink:
58                  value: 82
59                  unit: 1  # 0:bps, 1:Kbps, 2:Mbps, 3:Gbps, 4:Tbps
60            flow:
61              - direction: 2
62                description: "permit out icmp from any to assigned"
63              - direction: 1
64                description: "permit out icmp from any to assigned"
65              - direction: 2
66                description: "permit out udp from 10.200.136.98/32 23455 to
    assigned 1-65535"
67              - direction: 1
68                description: "permit out udp from 10.200.136.98/32 1-65535 to
    assigned 50021"
69          - qos:
```

```
              index: 2  # 1, 2, 3, 4, 65, 66, 67, 75, 71, 72, 73, 74, 76, 5, 6, 7, 8, 9, 69, 70, 79, 80, 82, 83, 84, 85, 86
              arp:
                priority_level: 4  # 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
                pre_emption_vulnerability: 2   # 1: Disabled, 2:Enabled
                pre_emption_capability: 2   # 1: Disabled, 2:Enabled
              mbr:
                downlink:
                  value: 802
                  unit: 1  # 0:bps, 1:Kbps, 2:Mbps, 3:Gbps, 4:Tbps
                uplink:
                  value: 802
                  unit: 1  # 0:bps, 1:Kbps, 2:Mbps, 3:Gbps, 4:Tbps
              gbr:
                downlink:
                  value: 802
                  unit: 1  # 0:bps, 1:Kbps, 2:Mbps, 3:Gbps, 4:Tbps
                uplink:
                  value: 802
                  unit: 1  # 0:bps, 1:Kbps, 2:Mbps, 3:Gbps, 4:Tbps
- plmn_id:
  mcc: 001
  mnc: 01
 slice:
  - sst: 1  # 1,2,3,4
    sd: 000001
    default_indicator: true
    session:
      - name: internet
        type: 3  # 1:IPv4, 2:IPv6, 3:IPv4v6
        ambr:
          downlink:
            value: 1
            unit: 3  # 0:bps, 1:Kbps, 2:Mbps, 3:Gbps, 4:Tbps
          uplink:
            value: 1
            unit: 3
        qos:
          index: 9  # 1, 2, 3, 4, 65, 66, 67, 75, 71, 72, 73, 74, 76, 5, 6, 7, 8, 9, 69, 70, 79, 80, 82, 83, 84, 85, 86
          arp:
            priority_level: 8  # 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
            pre_emption_vulnerability: 1  # 1: Disabled, 2:Enabled
            pre_emption_capability: 1  # 1: Disabled, 2:Enabled
```

Listing A.1: Default policy configuration options provided by pcf.yaml of Open5GS

Appendix B

Use-case Development

B.1 eMBB Video Streaming Application

Server-side application video streaming function:

```
def stream_video(video_path, stop_event):
    cap = cv2.VideoCapture(video_path)
    fps = cap.get(cv2.CAP_PROP_FPS)
    frame_delay = 1/fps if fps > 0 else 1/30

    orig_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    orig_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
    total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    current_frame = 0
    sequence_number = 0

    MAX_WIDTH = 1280
    if orig_width > MAX_WIDTH:
        scale_factor = MAX_WIDTH / orig_width
        target_width = MAX_WIDTH
        target_height = int(orig_height * scale_factor)
    else:
        target_width = orig_width
        target_height = orig_height

    logger.info(f"Starting video stream: {video_path}")
    logger.info(f"Original resolution: {orig_width}x{orig_height}")
    logger.info(f"Streaming resolution: {target_width}x{target_height}")
    logger.info(f"Total frames: {total_frames}, FPS: {fps}")

    try:
        while not stop_event.is_set():
```

```
            success, frame = cap.read()
            if not success:
                end_message = {
                    "type": "video_end",
                    "message": "Video playback completed",
                    "timestamp": time.time(),
                    "total_frames": total_frames,
                    "frames_sent": current_frame,
                    "final_sequence": sequence_number,
                    "duration": total_frames/fps if fps > 0 else 0,
                    "completion_percentage": (current_frame/total_frames * 100) if
    total_frames > 0 else 0
                }
                try:
                    end_message_bytes = b'END:' + json.dumps(end_message).encode('
    utf-8')
                    open5gs.send_data(UE_API_URL, end_message_bytes)
                    logger.info(f"Video playback completed. Sent {current_frame} out
     of {total_frames} frames")
                except Exception as e:
                    logger.error(f"Error sending video end message: {e}")
                break

            current_frame += 1
            sequence_number += 1

            try:
                if orig_width > MAX_WIDTH:
                    frame = cv2.resize(frame, (target_width, target_height),
                                       interpolation=cv2.INTER_AREA)

                encode_params = [
                    int(cv2.IMWRITE_JPEG_QUALITY), 80,
                    int(cv2.IMWRITE_JPEG_OPTIMIZE), 1
                ]

                ret, buffer = cv2.imencode('.jpg', frame, encode_params)
                if not ret:
                    logger.error("Failed to encode frame")
                    continue
                frame_data = buffer.tobytes()

                if len(frame_data) > 65000:
                    logger.warning(f"Frame size too large: {len(frame_data)} bytes.
    Reducing quality.")
                    encode_params[1] = 60
                    ret, buffer = cv2.imencode('.jpg', frame, encode_params)
                    if not ret:
```

```
                        continue
                    frame_data = buffer.tobytes()
                frame_header = f"FRAME:{sequence_number}:".encode('utf-8')
                marked_frame_data = frame_header + frame_data
                open5gs.send_data(UE_API_URL, marked_frame_data)

                time.sleep(frame_delay)
            except Exception as e:
                logger.error(f"Error processing/sending frame {current_frame}: {e}")
                time.sleep(0.1)
                continue
    except Exception as e:
        logger.error(f"Streaming error: {e}")
    finally:
        cap.release()
        logger.info(f"Stream ended after {current_frame} frames")

        if current_frame < total_frames and not stop_event.is_set():
            try:
                end_message = {
                    "type": "video_end",
                    "message": "Stream terminated",
                    "timestamp": time.time(),
                    "total_frames": total_frames,
                    "frames_sent": current_frame,
                    "final_sequence": sequence_number,
                    "duration": total_frames/fps if fps > 0 else 0,
                    "completion_percentage": (current_frame/total_frames * 100) if
    total_frames > 0 else 0
                }
                end_message_bytes = b'END:' + json.dumps(end_message).encode('utf-8'
    )
                open5gs.send_data(UE_API_URL, end_message_bytes)
                logger.info(f"Stream terminated. Sent {current_frame} out of {
    total_frames} frames " +
                            f"({(current_frame/total_frames * 100):.1f}% complete)")
            except Exception as e:
                logger.error(f"Error sending final stream statistics: {e}")
```

Listing B.1: Server-side video transmission

Client-side application video retrieval function:

```
def get_frame():
    try:
        response = open5gs.receive_data(UPF_STREAM_URL)
        if response is None:
            return jsonify({"status": "no_data"}), 404
```

```
        if isinstance(response, bytes):
            if response.startswith(b'END:'):
                # Parse end message
                end_message = json.loads(response[4:].decode('utf-8'))
                return jsonify({
                    "status": "video_end",
                    "message": end_message.get('message'),
                    "frames_sent": end_message.get('frames_sent'),
                    "total_frames": end_message.get('total_frames'),
                    "duration": end_message.get('duration')
                })
            elif response.startswith(b'FRAME:'):
                try:
                    second_colon = response.index(b':', 6)
                    frame_data = response[second_colon + 1:]
                    return Response(frame_data, mimetype='image/jpeg')
                except Exception as e:
                    logger.error(f"Error processing frame data: {e}")
                    return jsonify({"status": "error", "message": "Invalid frame
    format"}), 400

        return jsonify({"status": "error", "message": "Invalid data format"}), 400

    except Exception as e:
        logger.error(f"Error receiving video frame: {e}")
        return jsonify({
            "status": "error",
            "message": str(e)
        }), 500
```

Listing B.2: Client-side video data retieval

B.2 URLLC Remote Surgery Application

Locustfile for traffic generation:

```
from locust import HttpUser, task, between
import random
import time
import json

class RemoteSurgeryUser(HttpUser):
    wait_time = between(1, 3)
    host = "http://10.10.0.112:8081"

    @task
```

```
    def send_surgery_data(self):
        payload = {
            "sensor_id": "surgery_robot_1",
            "instrument_position": {
                "x": round(random.uniform(-100, 100), 3),
                "y": round(random.uniform(-100, 100), 3),
                "z": round(random.uniform(0, 150), 3)
            },
            "instrument_status": {
                "type": random.choice(["scalpel", "forceps", "needle_holder", "
    cautery"]),
                "action": random.choice(["cutting", "grasping", "idle", "retracting"
    ]),
                "force": round(random.uniform(0, 5), 2),
                "angle": round(random.uniform(0, 360), 2)
            },
            "haptic_feedback": {
                "resistance": round(random.uniform(0, 100), 2),
                "vibration": round(random.uniform(0, 50), 2),
                "temperature": round(random.uniform(35, 40), 2)
            },
            "vital_signs": {
                "heart_rate": random.randint(60, 100),
                "blood_pressure": f"{random.randint(110,140)}/{random.randint(60,90)
    }",
                "oxygen_saturation": round(random.uniform(95, 100), 1),
                "blood_loss": round(random.uniform(0, 100), 1)
            },
            "system_status": {
                "latency": round(random.uniform(1, 5), 2),
                "connection_strength": round(random.uniform(90, 100), 1),
                "battery_level": round(random.uniform(80, 100), 1),
                "error_flags": []
            },
            "timestamp": time.time()
        }

        response = self.client.post("http://localhost:8003/send_data", json=payload)
```

Listing B.3: Locustfile for traffic generation

B.3 mMTC Sensor Network Application

A custom data generator class to simulate different traffic patterns for IoT sensors in the mMTC demonstration:

```
import threading
import time
import random
import requests
import logging
import queue
from typing import Dict, List
from open5gsapi import open5gs, ConfigurationError, CommunicationError
import json
from dataclasses import dataclass, asdict
from datetime import datetime

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class UEDataGenerator:
    def __init__(self, config=None):
        self.base_port = 8080
        self.ue_url = open5gs.ue("send")
        self.active_ues: Dict[int, Dict] = {}
        self.stop_event = threading.Event()
        self.stats_queue = queue.Queue()
        self.statistics = {
            'environmental': {'data_rate': 0, 'error_rate': 0, 'packets_sent': 0, '
    errors': 0},
            'industrial': {'data_rate': 0, 'error_rate': 0, 'packets_sent': 0, '
    errors': 0},
            'critical': {'data_rate': 0, 'error_rate': 0, 'packets_sent': 0, 'errors
    ': 0},
            'status': {'data_rate': 0, 'error_rate': 0, 'packets_sent': 0, 'errors':
     0}
        }
        self.stats_lock = threading.Lock()
        self.global_packet_count = 0
        self.config = config if config else self._get_default_config()
        logger.info(f"Initialized UE Generator with config: {self.config}")
        self._start_stats_calculator()
        self.sensor_metrics_lock = threading.Lock()

    def _get_default_config(self):
        return {
            'environmental': {
                'temperature': {'min': 18, 'max': 28},
                'humidity': {'min': 30, 'max': 70},
                'interval': {'min': 30, 'max': 60}
            },
            'industrial': {
```

```
                'vibration': {'min': 0, 'max': 100},
                'pressure': {'min': 980, 'max': 1020},
                'interval': {'min': 5, 'max': 15}
            },
            'critical': {
                'smoke': {'min': 0, 'max': 50},
                'gas': {'min': 0, 'max': 100},
                'interval': {'min': 1, 'max': 3}
            },
            'status': {
                'battery_drain': {'min': 0.1, 'max': 0.5},
                'signal': {'min': -100, 'max': -50},
                'interval': {'min': 10, 'max': 30}
            }
        }

    def _start_stats_calculator(self):
        def calculate_rates():
            last_packets = {t: 0 for t in self.statistics}
            last_errors = {t: 0 for t in self.statistics}

            while not self.stop_event.is_set():
                time.sleep(1)
                with self.stats_lock:
                    for sensor_type in self.statistics:

                        new_packets = self.statistics[sensor_type]['packets_sent']
                        data_rate = new_packets - last_packets[sensor_type]
                        self.statistics[sensor_type]['data_rate'] = data_rate
                        last_packets[sensor_type] = new_packets

                        new_errors = self.statistics[sensor_type]['errors']
                        error_rate = new_errors - last_errors[sensor_type]
                        self.statistics[sensor_type]['error_rate'] = error_rate
                        last_errors[sensor_type] = new_errors

        self.stats_thread = threading.Thread(target=calculate_rates, daemon=True)
        self.stats_thread.start()
        logger.info("Started statistics calculator thread")

    def start_ue(self, port: int, sensor_type: str):
        try:
            response = requests.get(f"http://10.10.0.132:{port}/get_ue_ip", timeout
=2)
            if response.status_code == 200:
                ue_ip = response.json()['ue_ip']
                logger.info(f"Starting UE on port {port} with IP {ue_ip}")
                thread = threading.Thread(
```

```
                    target=self._ue_data_loop,
                    args=(port, ue_ip, sensor_type),
                    daemon=True
                )
                thread.start()

                self.active_ues[port] = {
                    'thread': thread,
                    'ip': ue_ip,
                    'type': sensor_type,
                    'started_at': time.time(),
                    'last_seen': time.time()
                }
                logger.info(f"Successfully started UE on port {port} with IP {ue_ip}
")
                return True
            else:
                logger.error(f"Failed to get UE IP for port {port}: Status {response
.status_code}")
                return False
        except Exception as e:
            logger.error(f"Failed to start UE on port {port}: {str(e)}")
            return False

    def _generate_sensor_data(self, sensor_type: str) -> Dict:
        try:
            if sensor_type == 'environmental':
                return {
                    'temperature': round(random.uniform(
                        self.config['environmental']['temperature']['min'],
                        self.config['environmental']['temperature']['max']
                    ), 2),
                    'humidity': round(random.uniform(
                        self.config['environmental']['humidity']['min'],
                        self.config['environmental']['humidity']['max']
                    ), 2)
                }
            elif sensor_type == 'industrial':
                return {
                    'vibration': round(random.uniform(
                        self.config['industrial']['vibration']['min'],
                        self.config['industrial']['vibration']['max']
                    ), 2),
                    'pressure': round(random.uniform(
                        self.config['industrial']['pressure']['min'],
                        self.config['industrial']['pressure']['max']
                    ), 2)
                }
```

```
            elif sensor_type == 'critical':
                return {
                    'smoke_level': round(random.uniform(
                        self.config['critical']['smoke']['min'],
                        self.config['critical']['smoke']['max']
                    ), 2),
                    'gas_concentration': round(random.uniform(
                        self.config['critical']['gas']['min'],
                        self.config['critical']['gas']['max']
                    ), 2)
                }
            elif sensor_type == 'status':
                return {
                    'battery_level': round(random.uniform(0, 100), 2),
                    'signal_strength': round(random.uniform(
                        self.config['status']['signal']['min'],
                        self.config['status']['signal']['max']
                    ), 2)
                }
            else:
                logger.error(f"Unknown sensor type: {sensor_type}")
                return {}
        except Exception as e:
            logger.error(f"Error generating sensor data: {str(e)}")
            return {}

    def _get_interval(self, sensor_type: str) -> float:
        try:
            interval_config = self.config[sensor_type]['interval']
            return random.uniform(interval_config['min'], interval_config['max'])
        except KeyError:
            logger.error(f"Missing interval configuration for sensor type {
    sensor_type}")
            return 10.0

    def _ue_data_loop(self, port: int, ue_ip: str, sensor_type: str):
        port_offset = port - self.base_port
        sequence = 0

        while not self.stop_event.is_set():
            try:
                sensor_data = self._generate_sensor_data(sensor_type)
                ue_id = f"{sensor_type}_{port_offset}"

                payload = {
                    'sensor_type': sensor_type,
                    'sensor_id': ue_id,
                    'ue_id': ue_id,
```

```
                    'ue_ip': ue_ip,
                    'port': port,
                    'timestamp': time.time(),
                    'sequence': sequence,
                    **sensor_data
                }

                response = open5gs.send_data(self.ue_url, payload, port_offset)

                with self.stats_lock:
                    if response and response.get('status') == 'success':
                        self.statistics[sensor_type]['packets_sent'] += 1
                        self.global_packet_count += 1
                        sequence += 1

                        sensor_metrics = open5gs.get_sensor_metrics()
                        if sensor_metrics and 'sensor_metrics' in sensor_metrics:
                            if sensor_type in sensor_metrics['sensor_metrics'].get('by_sensor', {}):
                                metrics = sensor_metrics['sensor_metrics']['by_sensor'][sensor_type]
                                self.statistics[sensor_type]['data_rate'] = metrics.get('current_rate', 0)
                    else:
                        self.statistics[sensor_type]['errors'] += 1

                interval = self._get_interval(sensor_type)
                time.sleep(interval)

            except Exception as e:
                logger.error(f"Error in UE {ue_ip}:{port} data loop: {str(e)}")
                with self.stats_lock:
                    self.statistics[sensor_type]['errors'] += 1
                time.sleep(5)

    def get_sensor_metrics(self):
        try:
            metrics = open5gs.get_sensor_metrics()
            if metrics and 'sensor_metrics' in metrics:
                return metrics['sensor_metrics']
            return None
        except Exception as e:
            logger.error(f"Error getting sensor metrics: {e}")
            return None

    def start_all_ues(self, num_ues: int):
        logger.info(f"Starting {num_ues} UEs")

```

```
229         env_ues = round(num_ues * 0.4)  # 40%
230         ind_ues = round(num_ues * 0.3)  # 30%
231         crit_ues = round(num_ues * 0.15)  # 15%
232         stat_ues = num_ues - env_ues - ind_ues - crit_ues
233 
234         ue_types = (
235             ['environmental'] * env_ues +
236             ['industrial'] * ind_ues +
237             ['critical'] * crit_ues +
238             ['status'] * stat_ues
239         )
240 
241         logger.info(f"UE distribution: env={env_ues}, ind={ind_ues}, crit={crit_ues
    }, stat={stat_ues}")
242 
243         for i, sensor_type in enumerate(ue_types):
244             port = self.base_port + i
245             if not self.start_ue(port, sensor_type):
246                 logger.warning(f"Failed to start UE {i} of type {sensor_type}")
247             else:
248                 logger.info(f"Started UE {i} of type {sensor_type} on port {port}")
249 
250     def stop_all(self):
251         logger.info("Stopping all UEs...")
252         self.stop_event.set()
253 
254         for ue_info in self.active_ues.values():
255             if 'thread' in ue_info and ue_info['thread'].is_alive():
256                 ue_info['thread'].join(timeout=2)
257 
258         with self.stats_lock:
259             self.active_ues.clear()
260             self.global_packet_count = 0
261             for stats in self.statistics.values():
262                 stats['packets_sent'] = 0
263                 stats['errors'] = 0
264                 stats['data_rate'] = 0
265                 stats['error_rate'] = 0
266 
267         logger.info("All UEs stopped and statistics reset")
268 
269     def get_stats(self) -> Dict:
270         with self.stats_lock:
271             stats = {
272                 'total_ues': len(self.active_ues),
273                 'total_packets': self.global_packet_count,
274                 'total_errors': sum(stats['errors'] for stats in self.statistics.
    values()),
```

```
                'ues_by_type': {
                    sensor_type: len([
                        1 for ue in self.active_ues.values()
                        if ue['type'] == sensor_type
                    ])
                    for sensor_type in self.statistics
                },
                'data_rates': {
                    sensor_type: stats['data_rate']
                    for sensor_type, stats in self.statistics.items()
                },
                'error_rates': {
                    sensor_type: stats['error_rate']
                    for sensor_type, stats in self.statistics.items()
                },
                'packets_by_type': {
                    sensor_type: stats['packets_sent']
                    for sensor_type, stats in self.statistics.items()
                }
            }
            stats['total_rate'] = sum(stats['data_rates'].values())
            return stats

if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(description='UE Data Generator')
    parser.add_argument('--num-ues', type=int, default=10, help='Number of UEs to
    simulate')
    args = parser.parse_args()

    generator = UEDataGenerator()
    try:
        generator.start_all_ues(args.num_ues)
        while True:
            stats = generator.get_stats()
            logger.info(f"Active UEs: {stats['total_ues']}, "
                        f"Packets: {stats['total_packets']}, "
                        f"Errors: {stats['total_errors']}")
            time.sleep(5)
    except KeyboardInterrupt:
        generator.stop_all()
```

Listing B.4: UEDataGenerator class for mMTC Application

Appendix C

API Methods

C.1 List of `open5gsapi` Methods and their Functionalities

Table C.1: Configuration Management API Methods

Method Syntax	**Functionality**
`set_config_path(path:str)`	Sets the path to the PCF YAML configuration file. Must be called before any policy operations.
`set_env_path(path:str)`	Sets the path to the environment configuration file containing UE settings. Required for UE operations.
`reload_config()`	Forces a reload of the PCF configuration from disk. Useful when external changes are made to the config file.
`reload_env()`	Forces a reload of the environment configuration. Updates UE-related settings.
`get_num_ues()->int`	Returns the current number of configured UEs from the environment file.
`set_num_ues(num_ues:int)`	Sets the number of UEs to be configured. Must be called before `update_config()`.

Table C.2: Session Management API Methods

Method Syntax	**Functionality**
`list_sessions() -> List[str]`	Returns a list of all configured session names in the PCF configuration.
`get_session_details(name: str) -> Dict`	Returns detailed configuration parameters for the specified session, including QoS settings and PCC rules.
`rename_session(old_name: str, new_name: str)`	Renames an existing session while preserving all its configurations.
`policy.session(name: str) -> Session`	Gets or creates a session object for configuration. Enables fluent configuration of session parameters.
`policy.add_session(name: str) -> Session`	Creates a new session with default parameters. Returns session object for further configuration.
`policy.remove_session(name: str)`	Removes the specified session from the configuration.

Table C.3: Network Communication API Methods

Method Syntax	**Functionality**
`ue(endpoint: str) -> str`	Constructs a UE API URL by appending the endpoint to the base UE URL (http://10.10.0.132:8080/).
`upf(endpoint: str) -> str`	Constructs a UPF API URL by appending the endpoint to the base UPF URL (http://10.10.0.112:8081/).
`send_data(endpoint: str, data: Any, port_offset: int = 0) -> Dict`	Sends data to specified endpoint. Automatically handles JSON and binary data, adds timestamps for metrics, and supports port offsets for multiple UEs.
`receive_data(endpoint: str) -> Any`	Receives data from specified endpoint. Automatically detects data type (JSON/binary) and processes metrics information.

Table C.4: Process Management API Methods

Method Syntax	**Functionality**
`update_pcf()`	Updates the PCF YAML file with current configuration changes. Must be called after modifying session parameters.
`update_config()`	Tears down and redeploys containers with new configuration. Required after updating PCF or environment settings.
`run_background_nodes()`	Initializes and starts background processes in UE and UPF containers based on status of creation of uesimtun interfaces.
`is_update_pcf_complete()->bool`	Checks if pcf.yaml has been modified successfully as per user inputs.
`is_update_config_complete()-> bool`	Checks if container redeployment has completed successfully.
`is_run_background_nodes_complete() -> bool`	Checks if background nodes are fully operational.
`get_background_process_status() -> Dict`	Returns detailed status of background processes including UE/UPF API readiness and error messages.

TABLE C.5: Metrics Collection API Methods

Method Syntax	**Functionality**
`get_metrics() -> Dict`	Returns comprehensive network performance metrics for video streaming data: • Throughput (Mbps) = $\frac{\text{total\_bytes} \times 8}{\text{elapsed\_time} \times 10^6}$ • Latency statistics (min, max, avg in ms) • RFC 3550 [51] jitter calculation: $J_i = J_{i-1} + \frac{\lvert D(i-1,i) \rvert - J_{i-1}}{16}$ • Frame-specific metrics including: – Current FPS = $\frac{1}{\text{mean(frame\_intervals)}}$ – Frame time = Average interval between consecutive frames (ms) – Frame size statistics – Frame counts (total, received, lost)
`get_frame_metrics() -> Optional[Dict]`	Returns subset of frame-specific metrics from `get_metrics()`. Used for convenience when only frame statistics are needed. Includes frame rate statistics (current FPS, frame time), frame size statistics (avg, min, max bytes) and frame delivery tracking (total, received, lost).
`get_sensor_metrics() -> Dict`	Returns metrics specific to sensor data transmissions: • Reading rate (readings/sec) • Latency statistics (min, max, avg in ms) • Jitter calculation using RFC 3550 • Per-sensor statistics (in case of multiple UEs) • Packet size
`reset_metrics()`	Resets streamer endpoint related metrics.
`reset_sensor_metrics()`	Resets all sensor endpoint related metrics.

Table C.6: Session Configuration API Methods

Method Syntax	**Functionality**
`session.set_type(session_type: int)`	Sets IP session type (1:IPv4, 2:IPv6, 3:IPv4v6).
`session.ambr.downlink(value:int, unit:int)`	Sets Aggregate Maximum Bit Rate for downlink. Unit values: 0:bps, 1:Kbps, 2:Mbps, 3:Gbps, 4:Tbps.
`session.ambr.uplink(value: int, unit: int)`	Sets Aggregate Maximum Bit Rate for uplink. Uses same unit values as downlink.
`session.qos(index:int)`	Sets QoS index for the session. Valid indices include standardized 5G QoS identifiers.
`session.arp(priority_level: int,pre_emption_vulnerability: int, pre_emption_capability: int)`	Sets Allocation and Retention Priority parameters. Priority: 1-15, Pre-emption: 1-2.
`session.pcc_rule[index].qos(index: int)`	Sets QoS index for specific PCC rule.
`session.pcc_rule[index]. mbr.downlink/uplink(value:int, unit: int)`	Sets Maximum Bit Rate for PCC rule.
`session.pcc_rule[index].gbr. downlink/uplink(value: int, unit: int)`	Sets Guaranteed Bit Rate for PCC rule.
`session.pcc_rule[index]. add_flow (direction:int, description:str)`	Adds flow rule to PCC rule. Direction: 1:uplink, 2:downlink.

Appendix D

Application Interfaces

D.1 Video Streaming Application WebUI

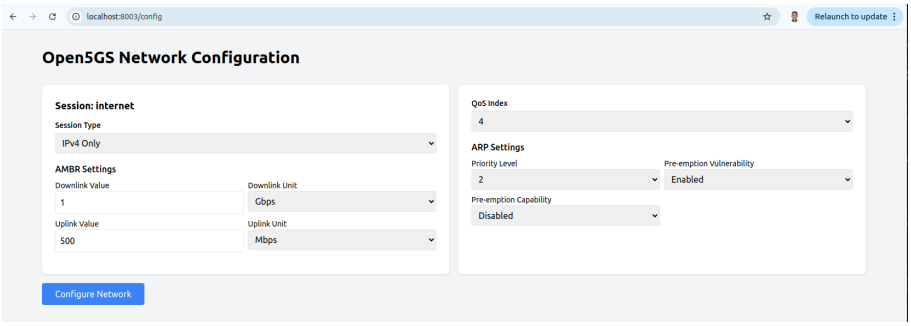


FIGURE D.1: eMBB - Network Configuration Interface

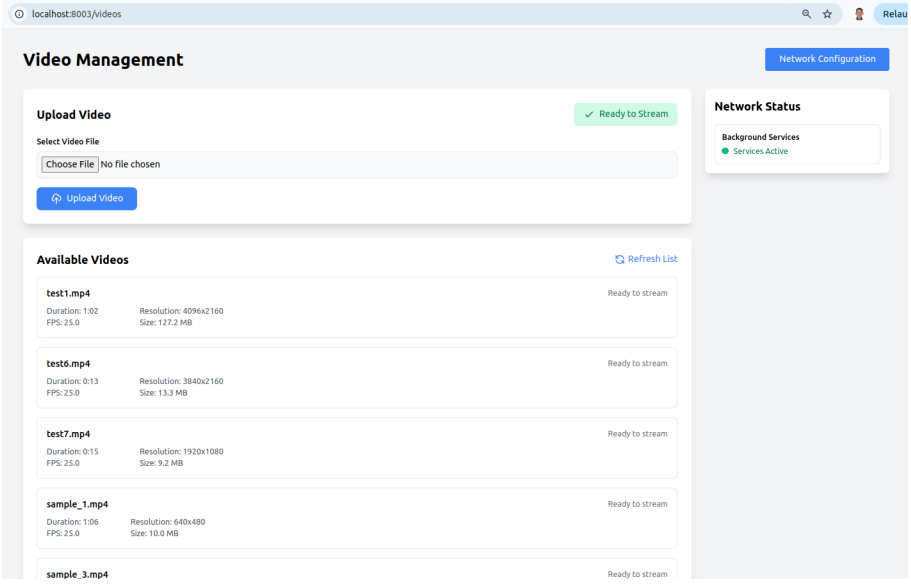


FIGURE D.2: eMBB - Video Management Interface

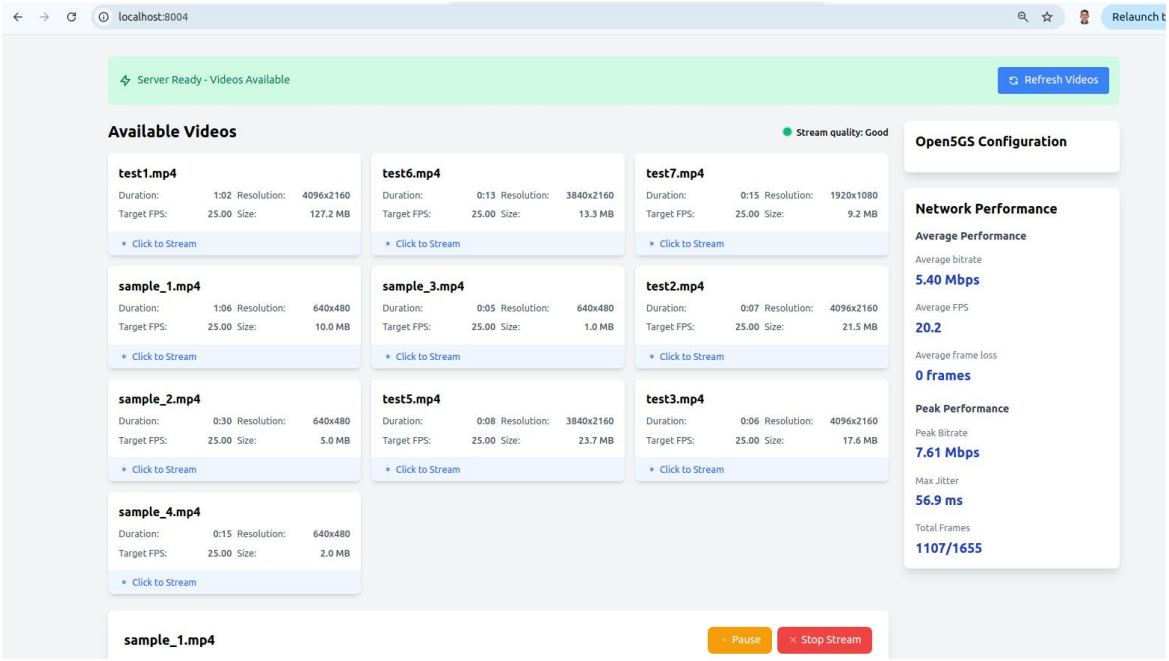


FIGURE D.3: eMBB - Video Streaming Client Interface - Catalog and Aggregate Metrics Display (640x480 video selected)

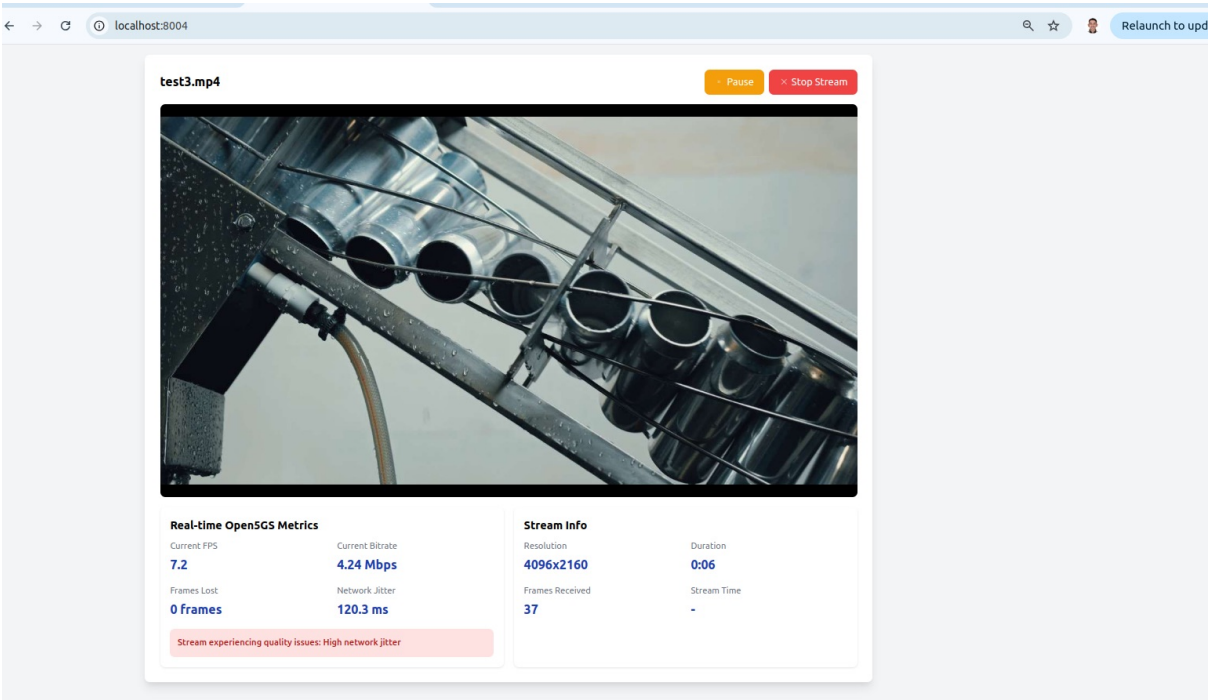


FIGURE D.4: eMBB - Video Streaming Client Interface - DCI 4K Video Playback and Metrics

D.2 Remote Surgery Simulator Application WebUI

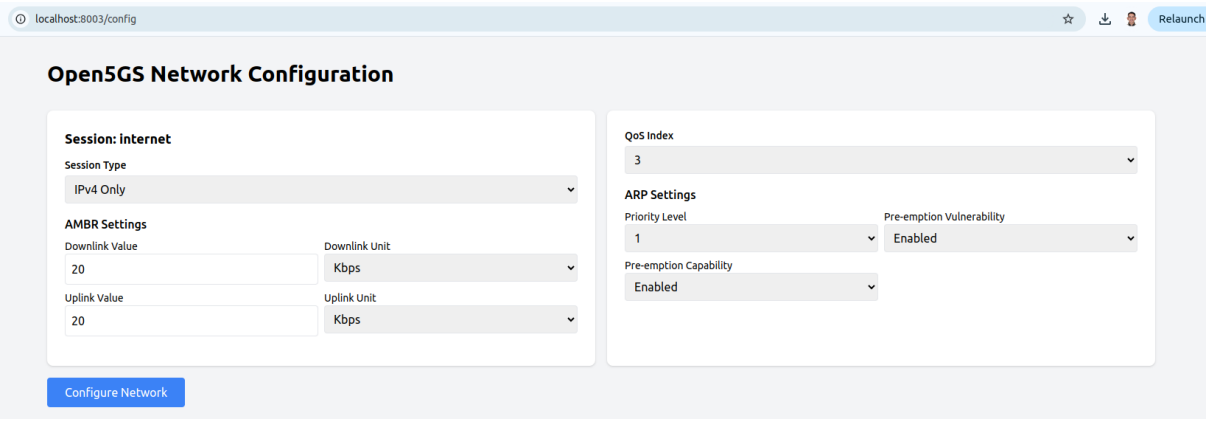


FIGURE D.5: URLLC - PCF Configuration Interface

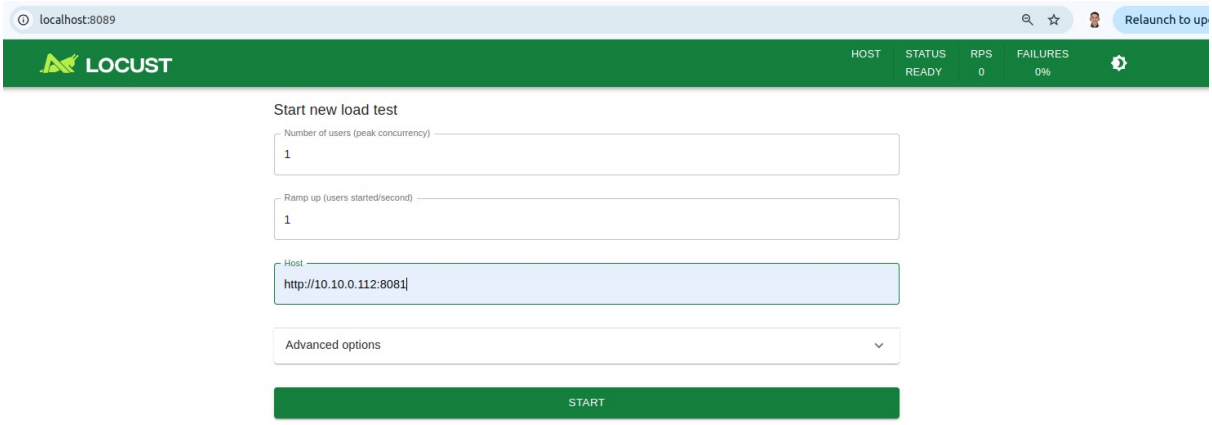


FIGURE D.6: URLLC - Locust Startup Page (Load Testing Settings)

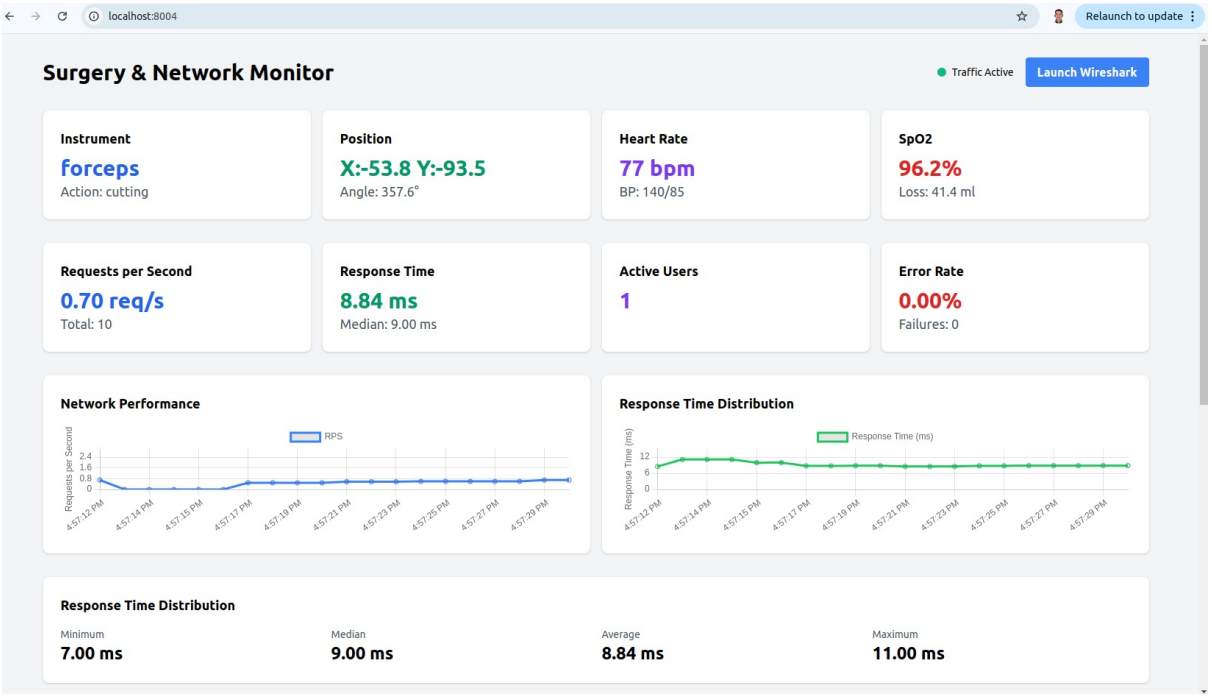


FIGURE D.7: URLLC - Surgery and Network Monitoring Dashboard - Data Charts

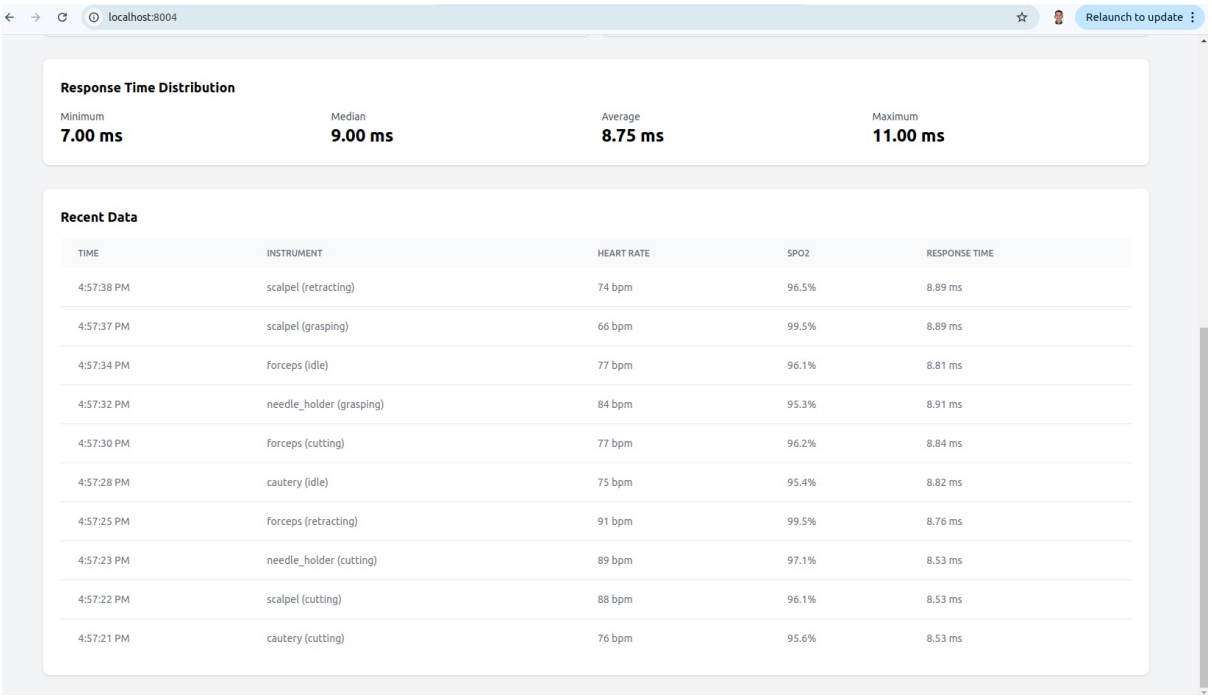


FIGURE D.8: URLLC - Surgery and Network Monitoring Dashboard - Data List

D.3 Sensor Network Application WebUI

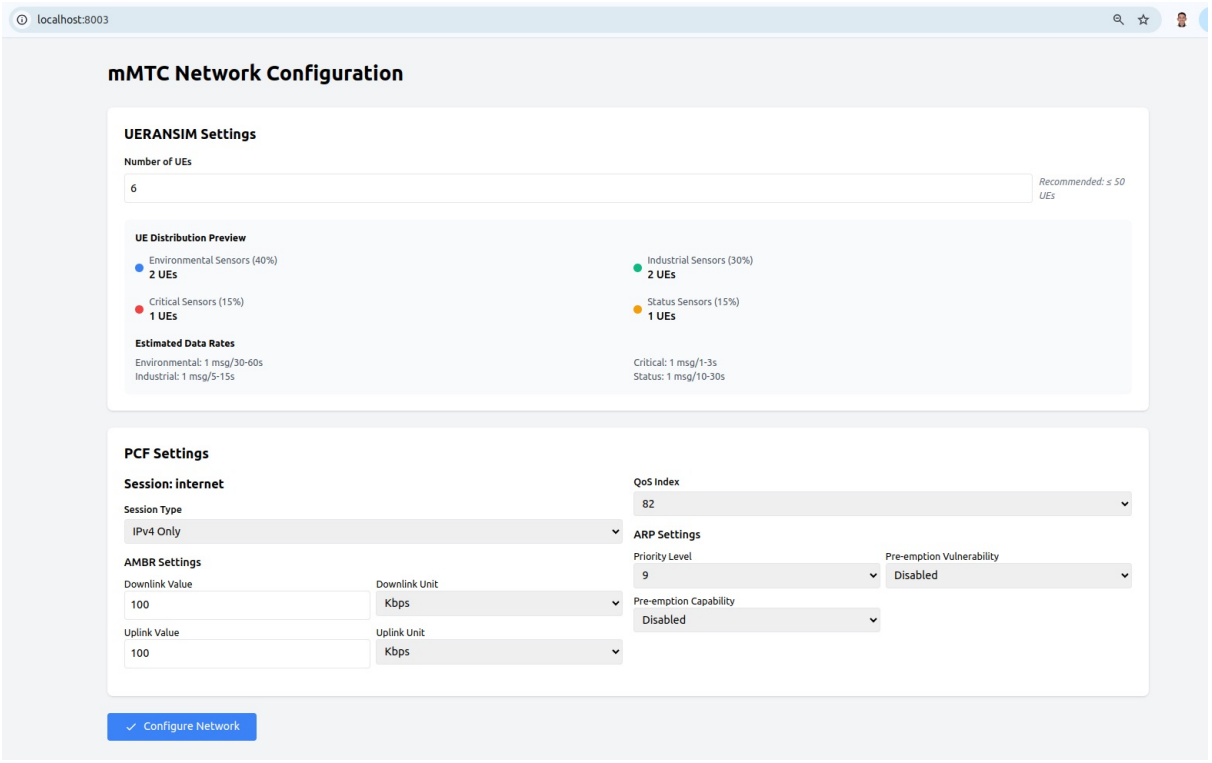


FIGURE D.9: mMTC - Network Configuration Interface

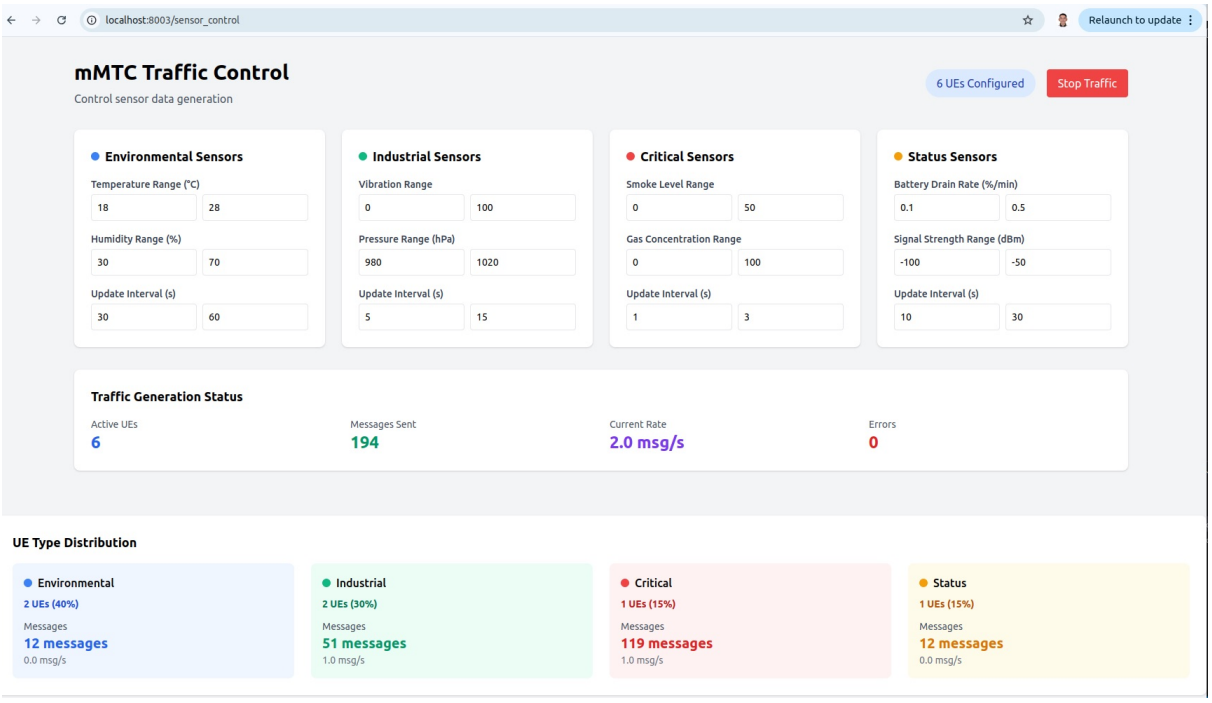


FIGURE D.10: mMTC - Sensor Control Panel

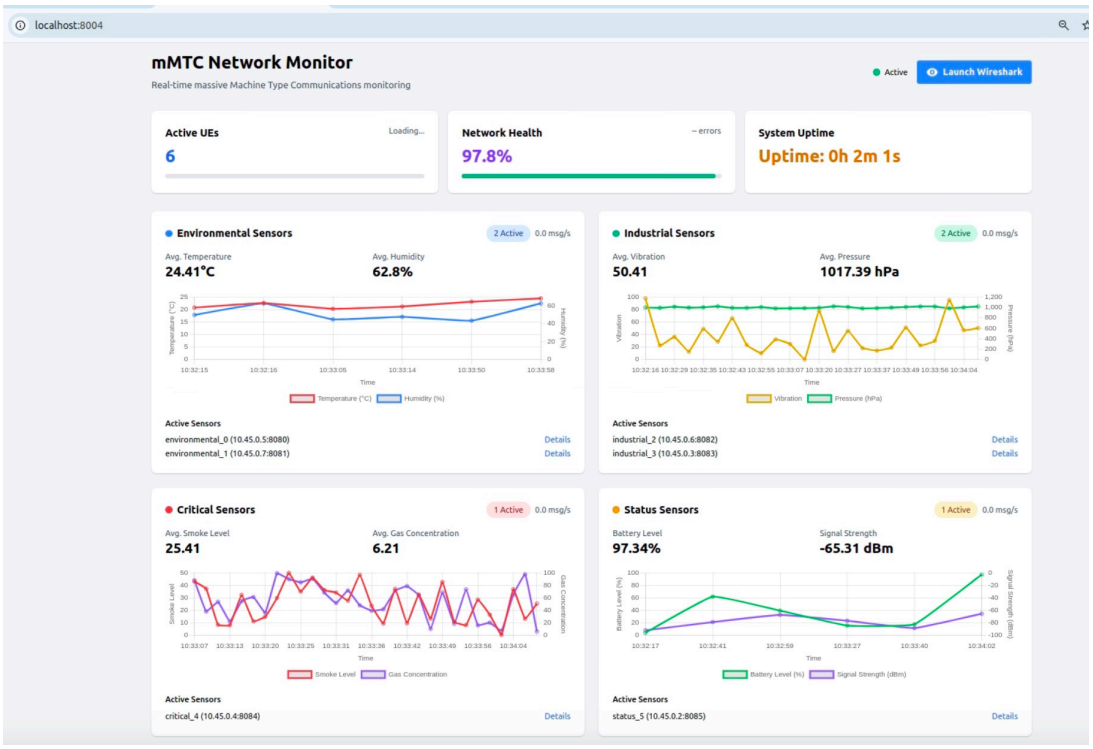


FIGURE D.11: mMTC - Server Dashboard (Network Status & Data Charts)

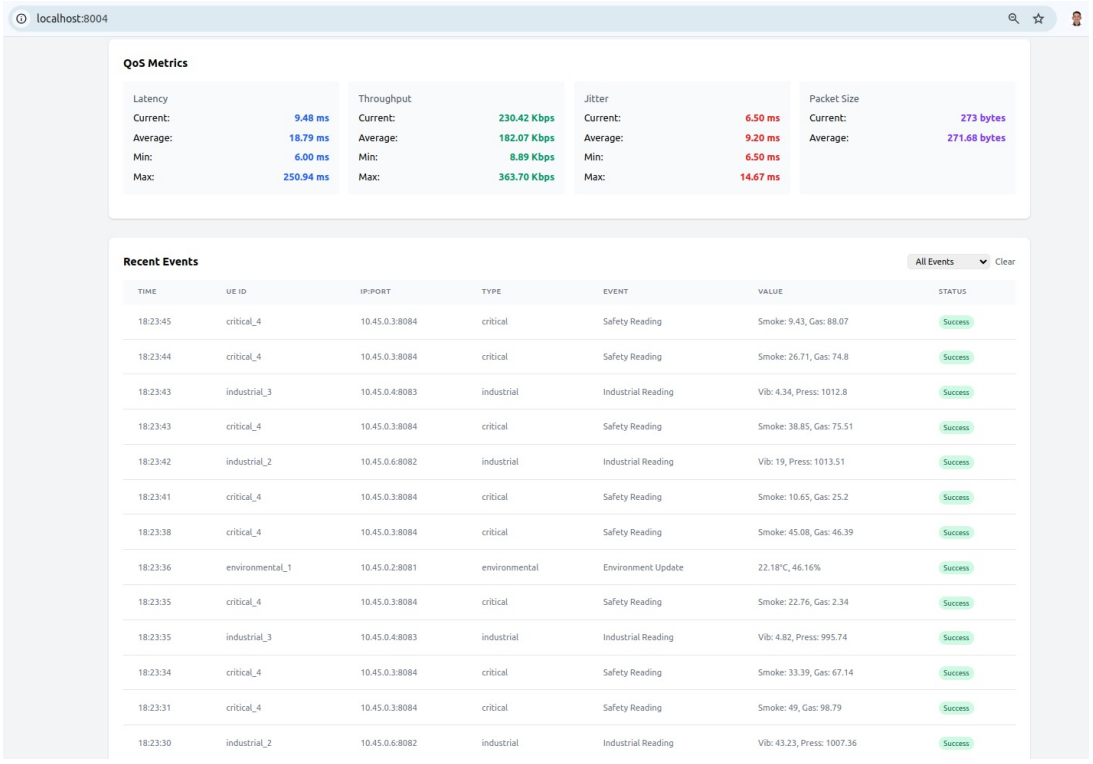


FIGURE D.12: mMTC - Server Dashboard (Metrics & Event Logs)

Bibliography

[1] I. T. Union. *Recommendation ITU-R M.2150-2: Detailed specifications of the terrestrial radio interfaces of International Mobile Telecommunications-2020 (IMT-2020)*. 2020. URL: https://www.itu.int/dms_pubrec/itu-r/rec/m/R-REC-M.2150-2-202312-I!!PDF-E.pdf.

[2] *5G System Overview*. URL: https://www.3gpp.org/technologies/5g-system-overview (visited on 08/08/2022).

[3] A. Bełciak. "5G Lab – Running a Fully Containerized 5G Core with Open5GS". In: *Software Mind* (Feb. 2024). URL: https://softwaremind.com/blog/5g-lab-running-a-fully-containerized-5g-core-with-open5gs/.

[4] G. Baldoni, J. Quevedo, C. Guimarães, A. de la Oliva, and A. Corsaro. "Data-Centric Service-Based Architecture for Edge-Native 6G Network". In: *IEEE Communications Magazine* 62.4 (2024), pp. 32–38. DOI: 10.1109/MCOM.001.2300178.

[5] ComSoc. *5G Security explained: 3GPP 5G core network SBA and Security Mechanisms*. Accessed: 2024-12-01. 2022. URL: https://techblog.comsoc.org/2022/01/01/5g-network-security-threats-and-3gpp-security-mechanisms/.

[6] ETSI. *5G; System Architecture for the 5G System (3GPP TS 23.501 version 17.5.0 Release 17)*. Technical Specification. Version 17.5.0. July 2022. URL: https://www.etsi.org/standards.

[7] A. Koranga. *Open5gs UPF code explanation with flow C*. June 2024. URL: https://medium.com/@aditya.koranga/open5gs-upf-code-explanation-with-flow-c-79c50f253dd1.

[8] ETSI. *5G; 5G System; Authentication Server Services; Stage 3 (3GPP TS 29.509 version 15.4.0 Release 15)*. Technical Specification. Version 15.4.0. July 2019. URL: https://www.etsi.org/deliver/etsi_ts/129500_129599/129509/15.04.00_60/ts_129509v150400p.pdf.

[9] ETSI. *5G; 5G System; Unified Data Management Services; Stage 3 (3GPP TS 29.503 version 15.2.1 Release 15)*. Technical Specification. Version 15.2.1. Apr. 2019. URL: https://www.etsi.org/deliver/etsi_ts/129500_129599/129503/15.02.01_60/ts_129503v150201p.pdf.

[10] A. Koranga. *ECIES in 5G Core: SUPI to SUCI Conversion*. https://medium.com/@aditya.koranga/ecies-in-5g-core-supi-to-suci-conversion-15c0fb8373e2. Medium. June 2024.

[11] ETSI. *5G; 5G System; Policy and Charging Control signalling flows and QoS parameter mapping; Stage 3 (3GPP TS 29.513 version 15.3.0 Release 15)*. Technical Specification. Version 15.3.0. Apr. 2019. URL: https://www.etsi.org/deliver/etsi_ts/129500_129599/129513/15.03.00_60/ts_129513v150300p.pdf.

[12] ETSI. *5G; 5G System; Network function repository services; Stage 3 (3GPP TS 29.510 version 15.3.0 Release 15)*. Technical Specification. Version 15.3.0. Apr. 2019. URL: https://www.etsi.org/deliver/etsi_ts/129500_129599/129510/15.03.00_60/ts_129510v150300p.pdf.

[13] I. Javid and S. Khara. "5G Network: Architecture, Protocols, Challenges and Opportunities". In: *2022 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*. 2022, pp. 1–5. DOI: 10.1109/ICCCIS56430.2022.10037605.

[14] F. Launay. "5G-NR Radio Interface – Data Link Layer". In: *NG-RAN and 5G-NR: 5G Radio Access Network and Radio Interface*. 2021, pp. 203–235. DOI: 10.1002/9781119851288.ch8.

[15] A. Papa, R. Durner, L. Goratti, T. Rasheed, and W. Kellerer. "Controlling Next-Generation Software-Defined RANs". In: *IEEE Communications Magazine* 58.7 (2020), pp. 58–64. DOI: 10.1109/MCOM.001.1900732.

[16] A. Peltonen, R. Sasse, and D. Basin. "A comprehensive formal analysis of 5G handover". In: *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec '21. Abu Dhabi, United Arab Emirates: Association for Computing Machinery, 2021, pp. 1–12. ISBN: 9781450383493. DOI: 10.1145/3448300.3467823. URL: https://doi.org/10.1145/3448300.3467823.

[17] Arminderkaur. *What is n3 interface in 5g*. Sept. 2024. URL: https://telcomatraining.com/what-is-n3-interface-in-5g/ (visited on 08/08/2022).

[18] R. F. Olimid and G. Nencioni. "5G Network Slicing: A Security Overview". In: *IEEE Access* 8 (2020), pp. 99999–100009. DOI: 10.1109/ACCESS.2020.2997702.

[19] X. Foukas, G. Patounas, A. Elmokashfi, and M. K. Marina. "Network Slicing in 5G: Survey and Challenges". In: *IEEE Communications Magazine* 55.5 (2017), pp. 94–100. DOI: 10.1109/MCOM.2017.1600951.

[20] *Open5GS Documentation*. URL: https://open5gs.org/open5gs/docs/guide/01-quickstart/ (visited on 08/08/2022).

[21] R. Reddy, M. Gundall, C. Lipps, and H. D. Schotten. "Open Source 5G Core Network Implementations: A Qualitative and Quantitative Analysis". In: *2023 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*. 2023, pp. 253–258. DOI: 10.1109/BlackSeaCom58138.2023.10299755.

[22] *Open5GS*. URL: https://open5gs.org (visited on 08/08/2022).

[23] *UERANSIM*. URL: https://github.com/aligungr/UERANSIM (visited on 08/08/2022).

[24] C. Bouras, A. Kollia, and A. Papazois. "SDN NFV in 5G: Advancements and challenges". In: *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*. 2017, pp. 107–111. DOI: 10.1109/ICIN.2017.7899398.

[25] A. Gopalasingham, D. G. Herculea, C. S. Chen, and L. Roullet. "Virtualization of radio access network by Virtual Machine and Docker: Practice and performance analysis". In: *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. 2017, pp. 680–685. DOI: 10.23919/INM.2017.7987358.

[26] P. Bahl, M. Balkwill, X. Foukas, A. Kalia, D. Kim, M. Kotaru, Z. Lai, S. Mehrotra, B. Radunovic, S. Saroiu, C. Settle, A. Verma, A. Wolman, F. Y. Yan, and Y. Zhang. "Accelerating Open RAN Research Through an Enterprise-scale 5G Testbed". In: *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*. ACM MobiCom '23. Madrid, Spain: Association for Computing Machinery, 2023. ISBN: 9781450399906. DOI: 10.1145/3570361.3615745. URL: https://doi.org/10.1145/3570361.3615745.

[27] Z. Wen, H. S. Pacherkar, and G. Yan. "VET5G: A Virtual End-to-End Testbed for 5G Network Security Experimentation". In: *Proceedings of the 15th Workshop on Cyber Security Experimentation and Test*. CSET '22. Virtual, CA, USA: Association for Computing Machinery, 2022, pp. 19–29. ISBN: 9781450396844. DOI: 10.1145/3546096.3546111. URL: https://doi.org/10.1145/3546096.3546111.

[28] G. Lando, L. A. F. Schierholt, M. P. Milesi, and J. A. Wickboldt. "Evaluating the performance of open source software implementations of the 5G network core". In: *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*. 2023, pp. 1–7. DOI: 10.1109/NOMS56928.2023.10154399.

[29] L. Mamushiane, A. A. Lysko, T. Makhosa, J. Mwangama, H. Kobo, A. Mbanga, and R. Tshimange. "Towards Stress Testing Open5GS Core (UPF Node) On A 5G Standalone Testbed". In: *2023 IEEE AFRICON*. 2023, pp. 1–6. DOI: 10.1109/AFRICON55910.2023.10293284.

[30] "BlueArch–An Implementation of 5G Testbed". In: *Journal of Communications* 14 (Dec. 2019), pp. 1110–1118. DOI: 10.12720/jcm.14.12.1110-1118.

[31] B. Koné, A. D. Kora, and B. Niang. "Network Resource Management and Core Network Slice Implementation: A Testbed for Rural Connectivity". In: *2022 45th International Conference on Telecommunications and Signal Processing (TSP)*. 2022, pp. 200–205. DOI: 10.1109/TSP55681.2022.9851288.

[32] B. Dzogovic, V. T. Do, B. Feng, and T. van Do. "Building virtualized 5G networks using open source software". In: *2018 IEEE Symposium on Computer Applications Industrial Electronics (ISCAIE)*. 2018, pp. 360–366. DOI: 10.1109/ISCAIE.2018.8405499.

[33] K. Suo, Y. Zhao, W. Chen, and J. Rao. "An Analysis and Empirical Study of Container Networks". In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 2018, pp. 189–197. DOI: 10.1109/INFOCOM.2018.8485865.

[34] E. Gamess and R. Surós. "An upper bound model for TCP and UDP throughput in IPv4 and IPv6". In: *Journal of Network and Computer Applications* 31.4 (2008), pp. 585–602. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2007.11.009. URL: https://www.sciencedirect.com/science/article/pii/S1084804507000732.

[35] S. Zahoor, I. Ahmad, M. Othman, A. Mamoon, A. Rehman, M. Shafiq, and H. Hamam. "Comprehensive Analysis of Network Slicing for the Develop-ing Commercial Needs and Networking Challenges". In: *Sensors* 22 (Sept. 2022), p. 6623. DOI: 10.3390/s22176623.

[36] Y. Cao, Y. Sun, and B. Liu. "5G QoS Resource Allocation Optimization Mechanism for gNB". In: *2022 International Wireless Communications and Mobile Computing (IWCMC)*. 2022, pp. 1194–1199. DOI: 10.1109/IWCMC55113.2022.9824982.

[37] ETSI. *Digital cellular telecommunications system (Phase 2+); Universal Mobile Telecommunications System (UMTS); LTE; Quality of Service (QoS) concept and architecture (3GPP TS 23.107 version 9.1.0 Release 9)*. Technical Specification. Version 9.1.0. June 2010. URL: https://www.etsi.org/deliver/etsi_ts/123100_123199/123107/09.01.00_60/ts_123107v090100p.pdf.

[38] B. Ellis, J. Stylos, and B. Myers. "The Factory Pattern in API Design: A Usability Evaluation". In: *29th International Conference on Software Engineering (ICSE'07)*. 2007, pp. 302–312. DOI: 10.1109/ICSE.2007.85.

[39] M. Kassab, C. Constantinides, and O. Ormandjieva. "Specifying and Separating Concerns from Requirements to Design: A Case Study." In: Jan. 2005, pp. 18–27.

[40] *What is 5QI in 5G?* Accessed: 2024-11-30. URL: https://www.5gworldpro.com/uncategorized/what-is-5qi-in-5g.html.

[41] GSMA. *5G Implementation Guidelines*. Tech. rep. Accessed: 2024-11-30. 2019. URL: https://www.gsma.com/solutions-and-impact/technologies/networks/wp-content/uploads/2019/05/5G-Implementation-Guidelines_v1_nonconfidential-R1.pdf.

[42] I. Vilà, O. Sallent, A. Umbert, and J. Pérez-Romero. "An Analytical Model for Multi-Tenant Radio Access Networks Supporting Guaranteed Bit Rate Services". In: *IEEE Access* 7 (2019), pp. 57651–57662. DOI: 10.1109/ACCESS.2019.2913323.

[43] S. (Louvros, M. Paraskevas, and T. Chrysikos. "QoS-Aware Resource Management in 5G and 6G Cloud-Based Architectures with Priorities". In: *Information* 14.3 (2023). ISSN: 2078-2489. DOI: 10.3390/info14030175. URL: https://www.mdpi.com/2078-2489/14/3/175.

[44] *Locust Documentation*. URL: https://docs.locust.io/en/stable/ (visited on 11/30/2024).

[45] L. Skorin-Kapov and M. Matijasevic. "Analysis of QoS Requirements for e-Health Services and Mapping to Evolved Packet System QoS Classes". In: *International Journal of Telemedicine and Applications* 2010.1 (2010), p. 628086. DOI: https://doi.org/10.1155/2010/628086. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1155/2010/628086. URL: https://onlinelibrary.wiley.com/doi/abs/10.1155/2010/628086.

[46] Y.-C. Tung, Z.-Q. Zhan, M.-H. Chuang, S.-H. Peng, C.-H. Ho, and C.-C. Chen. "5G Smart IoT Poles". In: *2022 International Conference on Wireless Communications Signal Processing and Networking (WiSPNET)*. 2022, pp. 153–157. DOI: 10.1109/WiSPNET54241.2022.9767154.

[47] Cisco. *3 Services to Help 5G Deliver Massive IoT*. Accessed: 2024-12-01. 2024. URL: https://blogs.cisco.com/sp/3services5gmassmachine.

[48] N. A. Mohammed, A. M. Mansoor, and R. B. Ahmad. "Mission-Critical Machine-Type Communication: An Overview and Perspectives Towards 5G". In: *IEEE Access* 7 (2019), pp. 127198–127216. DOI: 10.1109/ACCESS.2019.2894263.

[49] International Telecommunication Union. *ITU-T Recommendation H.266: Versatile Video Coding*. Accessed: 2024-12-04. Sept. 2023. URL: https://www.itu.int/rec/T-REC-H.266-202309-I/en.

[50] IEEE 802. *Time-Sensitive Networking (TSN) Task Group*. Accessed: 2024-12-04. 2024. URL: https://1.ieee802.org/tsn/.

[51] H. Schulzrinne, S. L. Casner, R. Frederick, and V. Jacobson. *RTP: A Transport Protocol for Real-Time Applications*. RFC 3550. July 2003. DOI: 10.17487/RFC3550. URL: https://www.rfc-editor.org/info/rfc3550.