

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



Net Neutrality in the 5G Era

Rúben Daniel Nunes Santos

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Prof. Manuel Ricardo

August 1, 2023

Abstract

5G introduced *network slicing*, enabling the existence of independent logical networks supported by a physical infrastructure. These logical networks are specially dedicated to meet the performance requirements of 5G use cases. This concept creates a potential conflict with the premise of network neutrality, which imposes that all traffic should be treated equally, without discrimination.

Thus, the following question arises: How to detect traffic differentiation in a 5G network? To answer that question, this dissertation analyzes the architecture of the 5G network and develops a new approach to the Network Data Analytics Function (*NWDAF*). An essential aspect of this function is its capability to allow a telecom regulator to obtain metrics directly from the 5G Network.

The proposed solution consists of a system composed of the 5G Core Network (*5GC*), the Radio Access Network (RAN), a User Equipment (*UE*), the *NWDAF* Network Function (*NF*), and two video stream Content and Application Providers (*CAPs*). The open-source software, Open Air Interface (*OAI*), emulates the 5G Core (*5GC*) and its components. The *UE* and the RAN are implemented with the *UERANSIM* simulator. As per the standard definition, the *NWDAF* is a web service that utilizes the Hypertext Transfer Protocol Version 2 (HTTP/2) protocol. The information collected by the *NWDAF* function will be presented to the telecom regulator through two web pages, allowing the option to choose which streams to monitor.

Functional tests ensure the accurate operation of the system's modules, whereas performance tests validate the proposed solution. The performance tests specifically target an eMBB 5G use case, where the *UE* will access two video stream flows originating from distinct *CAPs*. The ISP will interfere with one of the *CAPs*, causing it to experience performance degradation. The *NWDAF* will monitor both streams and retrieve network performance metrics. Therefore, the main contribution of the work conducted in this dissertation is the development of an *NWDAF* function capable of obtaining metrics from within the 5G Network. These metrics serve as a valuable tool for assessing Traffic Differentiation (*TD*).

Keywords: *5G, Network Neutrality, NWDAF, Traffic Differentiation*

Resumo

O 5G introduziu o *network slicing*, permitindo a existência de redes lógicas independentes suportadas por uma infraestrutura física. Estas redes lógicas são especialmente dedicadas a satisfazer os requisitos de desempenho de casos de utilização 5G. Este conceito cria um potencial conflito com a premissa da neutralidade da rede, que impõe que todo o tráfego deve ser tratado de forma igual, sem discriminação.

Assim, coloca-se a seguinte questão: Como detetar a diferenciação de tráfego numa rede 5G? Para responder a esta pergunta, esta dissertação analisa a arquitetura da rede 5G e desenvolve uma nova abordagem à "Network Data Analytics Function" (*NWDAF*). Um aspeto essencial desta função é a sua capacidade de permitir que um regulador de telecomunicações obtenha métricas diretamente da rede 5G.

A solução proposta consiste num sistema composto pela 5G Core Network (*5GC*), a Radio Access Network (*RAN*), um User equipment (*UE*), a *NWDAF* e dois Content and Application Providers (*CAPs*) de fluxos de vídeo. O software de código aberto, Open Air Interface (*OAI*), emula o 5G Core (*5GC*) e os seus componentes. O *UE* e a *RAN* são implementados com o simulador *UERANSIM*. De acordo com a definição padrão, o *NWDAF* é um serviço Web que utiliza o protocolo Hypertext Transfer Protocol Version 2 (*HTTP/2*). A informação recolhida pela função *NWDAF* será apresentada ao regulador de telecomunicações através de duas páginas web, permitindo a opção de escolher quais os fluxos a monitorizar.

Os testes funcionais garantem o correto funcionamento dos módulos do sistema, enquanto os testes de desempenho validam a solução proposta. Os testes de desempenho visam especificamente um caso de utilização eMBB 5G, em que o *UE* acede a dois fluxos de vídeo provenientes de *CAPs* distintos. O ISP interferirá com um dos *CAP*, causando-lhe uma degradação do desempenho. O *NWDAF* monitorizará ambos os fluxos e obterá métricas de desempenho da rede. Portanto, a principal contribuição do trabalho realizado nesta dissertação é o desenvolvimento de uma função *NWDAF* capaz de obter métricas de dentro da rede 5G. Essas métricas servem como uma ferramenta valiosa para avaliar a Diferenciação de Tráfego.

Palavras-Chave: 5G, Neutralidade de Rede, *NWDAF*, Diferenciação de tráfego

Contents

1	Introduction	3
1.1	Objectives and contributions	3
1.2	Document structure	4
2	Background and Fundamental Aspects	7
2.1	Network Neutrality	7
2.2	5G	8
2.3	NWDAF	10
2.4	<i>QoS</i>	11
2.4.1	Traffic Shaping	12
2.4.2	Scheduling	12
2.4.3	Implications on Net Neutrality	13
2.5	Docker containers	14
2.5.1	Definition	14
2.5.2	Containers and Virtual Machines	14
2.6	Open-Source 5G software packages	15
2.6.1	Core Software Packages	15
2.6.2	RAN software packages	16
3	State of the Art - Previous Works on Detection of Traffic Differentiation	17
3.1	Wired Networks	17
3.1.1	NetPolice	18
3.1.2	Glasnost	18
3.1.3	POPI	19
3.1.4	DiffProbe	20
3.2	Mobile Networks and Advancements in Content-Based Differentiation	21
3.2.1	Wehe	21
3.3	Conclusion	22
4	System Specification, Design and Implementation	23
4.1	System Specification	23
4.2	System Design	24
4.2.1	5G Core Network	25
4.2.2	User Equipment (UE) and gNB	25
4.2.3	NWDAF	25
4.2.4	Application Servers (<i>tCAP</i> and <i>bCAP</i>)	26
4.2.5	Telecom Regulator	26
4.3	System Implementation	26

4.3.1	5G Core Network	27
4.3.2	UPF	29
4.3.3	EXT-DN	37
4.3.4	UERANSIM	38
4.3.5	NWDAF	39
4.3.6	NWDAF Web Server	40
4.3.7	NWDAF Web Page	46
5	System and <i>NWDAF</i> Validation	53
5.1	Methodology	53
5.2	Component Testing	53
5.2.1	Core Network and <i>NWDAF</i>	54
5.2.2	<i>UERANSIM</i>	55
5.3	Use Case Validation	55
5.3.1	<i>eMBB</i>	55
5.4	Discussion	62
6	Conclusions	65
6.1	Limitations and Future work	66
	References	69

List of Figures

2.1	High-level description of <i>TD</i> on an <i>ISP</i> network	7
2.2	5G <i>eMBB</i> , <i>URLLC</i> , and <i>mMTC</i> network slices use cases	8
2.3	5G network architecture	9
2.4	5G network data plane protocol stack	10
2.5	Data collection procedure from UE	11
2.6	Traffic Shaping with Token bucket	12
2.7	Generalized processor sharing (GPS)	13
2.8	General architecture of docker containers	14
2.9	Differences between Docker containers and Virtual Machines	15
3.1	Port blocking methodology	17
3.2	Detecting various types of differentiation with end-host based probing	18
3.3	The Glasnost system	19
3.4	The Glasnost web interface	20
3.5	Model of access <i>ISP</i> discrimination	20
4.1	Diagram of the 5G network and the proposed solution	24
4.2	Design used to implement and test the proposed solution	25
4.3	Containers of all the components used to implement the system	26
4.4	Network diagram of all the container running on the host PC	27
4.5	Docker networks defined in the <i>docker-compose-basic-vpp-nrf.yaml</i> file	28
4.6	Packages installed as part of the new <i>UPF Docker</i> image	30
4.7	<i>captures.sh</i> script processing of the command line arguments	31
4.8	<i>tshark</i> commands to filter and create a csv file with two fields	32
4.9	processing of the arguments of the <i>calculate_metrics.py</i> Python script	33
4.10	parsing the fields in the <i>csv</i> files, calculating the packets that were received and their associated timestamps	34
4.11	calculation of the packet loss, average delay and standard deviation of the delay in <i>Python</i>	34
4.12	Calculation of the throughput in <i>Python</i>	35
4.13	Creation of the packet delay histogram in <i>Python</i>	35
4.14	Byte range associated with the source IP address in <i>wireshark</i>	36
4.15	Changes made to the <i>UERANSIM</i> docker-compose file	38
4.16	Command used to configure and run the <i>NWDAF</i> container	40
4.17	Definition of the configurations for the <i>NWDAF</i> server	40
4.18	Setting of the IP address of the <i>NWDAF</i> server, as well as the definition of the self-signed certificate related files and creation of the <i>quart</i> object	41
4.19	Root route of the <i>NWDAF</i> web server	42

4.20	First snippet of the <i>'/handler'</i> route of the <i>NWDAF</i> web server	42
4.21	Definition of the Subscription class in <i>Python</i>	43
4.22	Second snippet of the <i>'/handler'</i> route of the <i>NWDAF</i> web server	43
4.23	Third snippet of the <i>'/handler'</i> route of the <i>NWDAF</i> web server	44
4.24	Final snippet of the <i>'/handler'</i> route of the <i>NWDAF</i> web server	44
4.25	<i>'unsubscribe'</i> route of the <i>NWDAF</i> web server	45
4.26	<i>'metrics_A'</i> route of the <i>NWDAF</i> web server	45
4.27	<i>'/delay_histogram_Stream_A/<stream_name>'</i> dynamic route of the <i>NWDAF</i> web server	46
4.28	Homepage of the <i>NWDAF</i> web page	47
4.29	First snippet of the <i>handleSubscribeClick</i> function	48
4.30	Second snippet of the <i>handleSubscribeClick</i> function	48
4.31	First snippet of the <i>buildUrl</i> function	49
4.32	Second snippet of the <i>buildUrl</i> function	49
4.33	Final snippet of the <i>buildUrl</i> function	49
4.34	Overview of the <i>analytics.html</i> page	50
4.35	<i>JavaScript</i> code to handle the update of the metric values displayed in the <i>analytics.html</i> page	51
4.36	<i>JavaScript</i> code to handle the update of the images displayed in the <i>analytics.html</i> page	52
4.37	<i>handleUnsubscribeClick</i> <i>JavaScript</i> function	52
5.1	Successful pings from host PC to each Core component	54
5.2	Successful ping from host PC to <i>NWDAF</i>	54
5.3	Ping from <i>UERANSIM</i> to the Internet (google.com)	55
5.4	<i>Wireshark</i> capture showing ping packets sent from <i>UERANSIM</i> going through the UPF	55
5.5	Delay histograms for HTTP/2 delay test	56
5.6	<i>NWDAF</i> metrics for HTTP/2 delay test	56
5.7	Delay histograms for HTTP/3 delay test	57
5.8	<i>NWDAF</i> metrics for HTTP/3 delay test	57
5.9	Retrieval of the source IP address and the protocol name for a custom stream	58
5.10	Delay histograms for custom stream delay test	59
5.11	<i>NWDAF</i> metrics for custom stream delay test	59
5.12	<i>NWDAF</i> metrics for HTTP/2 loss test	60
5.13	<i>NWDAF</i> metrics for HTTP/3 loss test	60
5.14	<i>NWDAF</i> metrics for custom stream loss test	60
5.15	Delay histograms for HTTP/2 delay and loss test	61
5.16	<i>NWDAF</i> metrics for HTTP/2 delay and loss test	61
5.17	Delay histograms for HTTP/3 delay and loss test	61
5.18	<i>NWDAF</i> metrics for HTTP/3 delay and loss test	62
5.19	Delay histograms for custom stream delay and loss test	62
5.20	<i>NWDAF</i> metrics for custom stream delay and loss test	62
6.1	Overall architecture of NG-RAN	67

List of Tables

2.1	5G Core open-source software packages comparison.	16
2.2	5G Core open-source software packages comparison.	16
3.1	Summary of the tools described and their characteristics	22
4.1	Core Network components and their respective IP addresses	29
4.2	UPF interface names, IP addresses and associated networks	30

Abbreviations and Symbols

3GPP	3rd Generation Partnership Project
5GC	5G Core Network
AJAX	Asynchronous JavaScript And XML
AMF	Access and Mobility Management Function
AP	Application Function
AS	Autonomous System
AUSF	Authentication Server Function
CAP	Content and Application Provider
CSS	Cascading Style Sheets
eMBB	enhanced Mobile Broadband
FCFS	First-Come First-Served
gNB	gNodeB
gNB-CU	gNB Central Unit
gNB-DU	gNB Distributed Unit
GPS	Generalized Processor Sharing
GTP-U	General Packet Radio Service Tunneling Protocol
HTTP	HyperText Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
ICMP	Internet Control Message Protocol
IP	Internet Protocol
ISP	Internet Service Provider
LTE	Long Term Evolution
ML	Machine Learning
mMTC	massive Machine-Type Communication
MTU	Maximum Transfer Unit
NEF	Network Exposure Function
NF	Network Function
NG-RAN	Next Generation Radio Access Network
NIC	Network Interface Card
NN	Network Neutrality
NR	New Radio
NRF	Network Repository Function
NWDAF	Network Data Analytics Function
P2P	Peer-to-Peer
PDU	Protocol Data Unit
PFP	Packet Forwarding Prioritization
OAI	Open Air Interface
OAM	Operations Administration and Maintenance

OS	Operating System
QoS	Quality of Service
RAN	Radio Access Network
RSP	Rogue Super Peer
RTP	Rádio e Televisão de Portugal
SDN	Software-Defined Networking
SDR	Software-Defined Radio
SMF	Session Management Function
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TD	Traffic Differentiation
TSO	TCP segmentation offload
UDM	Unified Data Management
UDP	User Datagram Protocol
UDR	Unified Data Repository
UE	User Equipment
UPF	User Plane Function
URL	Uniform Resource Locator
URLLC	Ultra Reliable Low Latency Communication
VM	Virtual Machine
VOIP	Voice over Internet Protocol
WFQ	Weighted fair queueing

s

Chapter 1

Introduction

5G uses a concept known as network slicing to virtualize the network with dedicated radio and computational resources. These slices are designed to customize certain Quality of Service (*QoS*) requirements such as throughput, latency, reliability, or connection density to account for the diversity of traffic.

The customization of performance parameters in logical networks produces a set of virtual domains that give a different treatment to Internet traffic. This creates a potential conflict with the net neutrality principle, which states that all Internet traffic should be treated equally. The net neutrality concept is still a polemic subject. More conservative approaches indulge the idea that traffic differentiation should not be allowed, while others support the segmentation of Internet traffic into classes, for example. Both points of view seem to agree that net neutrality imposes a non-discriminatory treatment of Internet traffic.

1.1 Objectives and contributions

The research question addressed in this dissertation is the following: How can we identify traffic differentiation in a 5G network in the context of net neutrality? Taking this into consideration, the following objectives were established:

- Define the concept of net neutrality.
- Characterize the 5G network architecture.
- Analyze the Network data analytics function (*NWDAF*).
- Simulate the 5G Core Network and its components.
- Define and implement a use case for testing.
- Test and validate the developed function.

In order to accomplish these objectives, a *NWDAF* will be developed. This specialized *NWDAF* aims at monitoring traffic differentiation that could eventually lead to discriminatory practices and

impact the net neutrality principle. As *NWDAF* is part of the network functions of the 5G network architecture, it relies on data collected inside the network. As such, the data could be accessible by the telecom regulator based on its legal jurisdiction.

To intentionally reduce the performance of specific traffic, a traffic shaper can employ various packet scheduling and buffer management policies. As a result, this will typically lead to increased delays and potentially more instances of packet loss from the user's perspective [1]. These two indicators allow us to determine whether the *ISP* has deliberately tampered with the packets in the UPF packet queue. The *ISP* can purposefully delay packets from a specific content and applications provider (*CAP*) by using Quality of Service (*QoS*) techniques such as traffic shaping. They can go even further and discard the packets entirely. The purpose is to detect these activities using the measurements provided by the Operations, Administration and Maintenance (*OAM*) 5G service and collected by the *NWDAF*.

As a use case, different flows from two separate *CAPs* were monitored. For example, a YouTube and an "RTP" traffic flow were created, with one of them facing throttling. Throttling is an intentional degradation in network performance for a given service, application, or class of applications. Throughout this dissertation, the acronym "RTP" is used to refer specifically to the Portuguese Television Channel, "Rádio e Televisão de Portugal". It should be noted that in this context, "RTP" does not refer to the RTP protocol commonly associated with Real-Time Transport Protocol. The usage of "RTP" within this dissertation is focused solely on the television channel unless explicitly stated otherwise. Everything was done remotely, in an emulated environment, on top of a virtual 5G Core Network. Then, by comparing these flows, the *NWDAF* should be able to detect the *TD* and make the information available for the telecom regulatory entity.

This approach is an innovative way of using the *NWDAF* and, most importantly, it enables the detection of traffic differentiation (*TD*) practices inside the network. This represents an innovative milestone, as it mitigates the information asymmetry between the regulator and the *ISP*, thereby enhancing transparency in traffic management practices.

1.2 Document structure

The document is structured into six chapters, as follows: Chapter 2, Background and Fundamental Aspects, introduces fundamental concepts and technologies required to understand the problem at hand and the solution proposed in this dissertation. Chapter 3, State of the Art, further elaborates on the technical aspect by reviewing some of the prior work focused on solving a similar problem to that of this dissertation. Chapter 4, System Specification, Design and Implementation, specifies the solution used to solve the problem addressed by this dissertation. It outlines the specifications of the system developed while also giving a high-level structure of the system. The overall system design is described, taking into account components, modules, and their relationships. Lastly, it focuses on the implementation of the system. A closer look into the technologies, tools, and programming languages used to develop the system provides a deeper understanding of the solutions implemented, as well as the challenges faced along the development process. It may include code

snippets, algorithms, or data structures relevant to the system's implementation. Chapter 5, System and *NWDAF* Validation, encompasses the activities and methodologies used to ensure that the system and *NWDAF* meet the specified requirements and perform as intended. Chapter 6, conclusion, reinvigorates the main goal of the dissertation and its major contributions. Additionally, it summarizes the shortcomings of the system and mentions possible improvements that can be added to the final product, as future work.

Chapter 2

Background and Fundamental Aspects

2.1 Network Neutrality

According to the authors of [2], net neutrality states that all Internet traffic, regardless of origin, destination, or content, should be treated equally. *ISPs* feel compelled to apply discriminatory traffic management practices to obtain a competitive advantage over the competitors, increase the number of customers, or charge higher fees. Discriminatory traffic management practices are often called Traffic Differentiation (*TD*). In some cases, *TD* can be considered reasonable and allowed, as long as it is transparent, such as i) Addressing illegal content (piracy, spam, or viruses); ii) Prioritizing DNS queries; and iii) Prioritizing special services, e.g., real-time health services. In general, *TD* can be seen as reasonable if it is beneficial to the network and its users as a whole. However, *TD* is detrimental to innovation, fair competition, and consumers' freedom of choice, three important factors deemed essential for the success of the Internet [3]. Three major factors offer a problem to *TD* detection: 1) Detect *TD* as an external observer with no access to the network configurations; 2) Detect *TD* using only traffic (source address, destination address, port number); 3) Identify which features trigger *TD* and where it is happening.

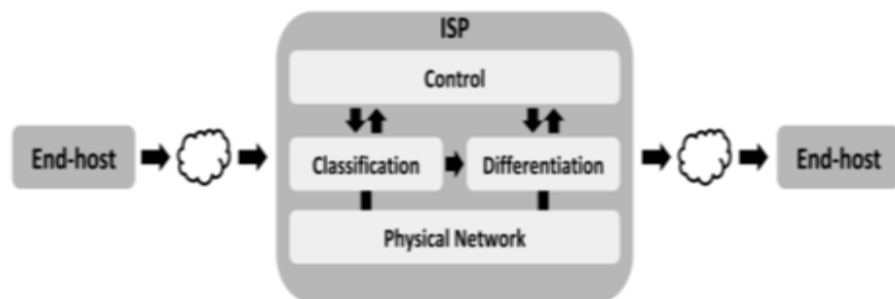


Figure 2.1: High-level description of *TD* on an *ISP* network [2]

Figure 2.1 exemplifies how an *ISP* can affect traffic between two end hosts. Any traffic traversing through the *ISP* is classified and handled accordingly to the assigned class. The classification

of Internet traffic can be based on several criteria, such as: 1) TCP/UDP port, which may contain information about the traffic type; 2) Source address; 3) Destination address; 4) Application protocol.

Just like the Internet, the *TD* mechanisms used by *ISPs* are continuously evolving. Traffic characteristics, such as payload and port number, can be used by commercial traffic shapers to determine many applications. Still, newer technologies, like software-defined networking (SDN), will increase flexibility and enable various types of discriminatory behaviors to be easily implemented in the network. *TD* detection mechanisms will have to be able to adapt to the evolution of networks.

Since 2003, the number of individuals, businesses, and private and public institutions involved in the *NN* discussion has grown. Several *CAPs*, like Google and Netflix, support *NN*, while *ISPs* are generally against it. The scientific community has also chimed in, providing techniques for detecting *NN* infringements. Cases of network neutrality violations have been reported on all five continents, demonstrating that the topic is extremely complex and that monitoring and enforcing compliance with *NN* legislation has become a challenge. Overall, *NN* is a hotly debated topic as policies are adopted and withdrawn worldwide [2].

2.2 5G

5G network slices are designed with the goal of optimizing the priorities of traffic. As such, they aim at providing low latency, high data rates, high reliability and a huge number of connected devices. The three main 5G network slices are: enhanced Mobile Broadband (*eMBB*), massive Machine-Type Communications (*mMTC*), and Ultra-Reliable and Low-Latency Communications (*URLLC*). *eMBB* corresponds to services that enable the transfer of large amounts of data for an improved user experience. *mMTC* corresponds to services designed to server a massive number of connected devices (*IoT*), which have low cost and consume very little energy. *URLLC* was projected for services that require ultra reliability and low latency. Figure 2.2 presents some use cases for each of the three mentioned network slices.

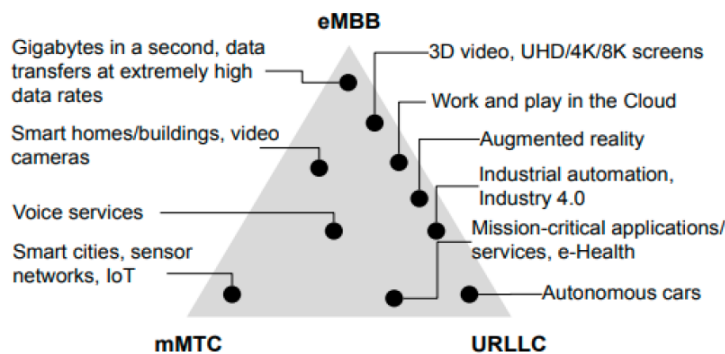


Figure 2.2: 5G *eMBB*, *URLLC*, and *mMTC* network slices use cases [4]

When it comes to its architecture, the 5G network can be decomposed into two planes: control plane and data plane. These two planes are independent, and they are connected by multiple interfaces. To provide scalability, evolution and flexible deployments, their functions are divided. Software-Defined Networking (SDN) makes this possible, by logically separating the two planes. The data plane, also known as user plane, has three main components: User Equipment (UE), Radio Access Network (RAN), and UPF. The UE connects to the UPF through the RAN. The UPF acts as a router to the Internet. The data plane (also known as the user plane) is the part of the 5G network responsible for the transportation of packets. The two planes, their main functions and the dissociation between the access network and Core network is shown in *Figure 2.3*.

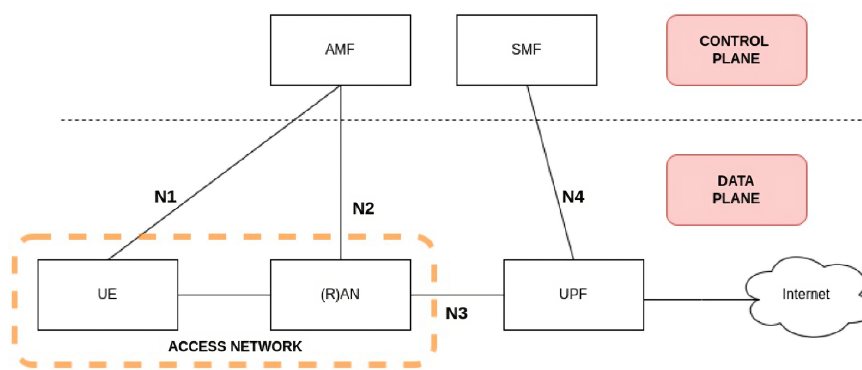


Figure 2.3: 5G network architecture [4]

The UPF acts as a gateway between the RAN and the Internet. As stated above, the UPF acts like a router, so it also has the functions of routing and forwarding packets. Quality of service (QoS) is also handled by the UPF. All the radio networking is ensured by the RAN. The RAN is primarily composed of a node, named gNB. The gNB is responsible for establishing connection between the UE and the core Network. One crucial point to consider regarding the data plane is its protocol stack. Based on the PDU session, the PDU layer determines the packet structure. In other words, if the session is IPv4, IPv4 packets are used, whereas Ethernet packets are used for Ethernet sessions. The General Packet Radio Service Tunneling Protocol (GTP-U) creates a tunnel to transport the user data from the Access network to the UPF (interface N3). It is in charge of encapsulating all end-user PDUs on the network. *Figure 2.4* depicts the data plane stack configuration.

The main functions of the control plane are the Access and Mobility Management Function (AMF) and the Session Management Function (SMF). The AMF is responsible for access control and UE location management, while the SMF is in charge of managing sessions between the UE and the UPF. The control plane functions are virtualized, and they behave like web services. They are called Network Functions (NFs) and they are all independent of each other.

Essentially, the 5G Core Network can be seen as a set of independent Web services that communicate with each other using HTTP, over the standard TCP/IP protocol stack. The HTTP protocol includes a set of methods that perform various tasks. The six main HTTP methods are: 1)

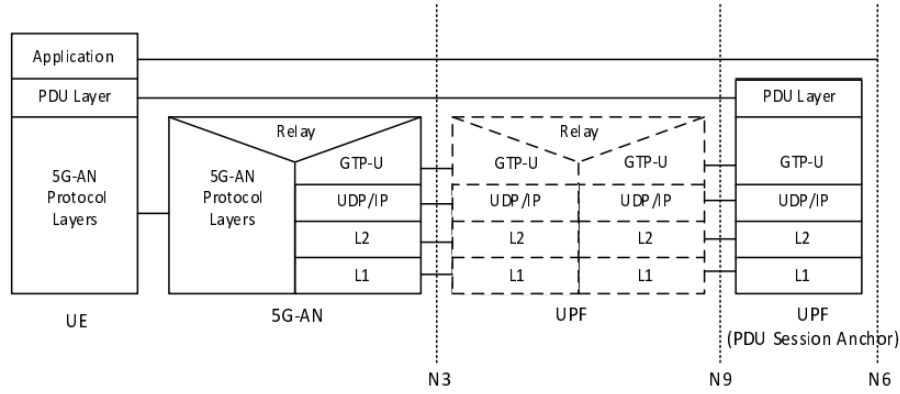


Figure 2.4: 5G network data plane protocol stack [4]

GET, which is used to read a resource; 2) POST, which is used to create a resource; 3) PATCH, used to alter a portion of a resource; 4) DELETE, used to delete a resource; 5) PUT, used to update the full resource; 6) OPTIONS, used to obtain information about a resource. [4]

2.3 NWDAF

The *NWDAF* is one of the Network Functions (NFs) of the 5G network and is able to provide analytics information to an NF consumer. Usually, an NF consumer is another NF of the 5G Network. The *NWDAF* is capable of getting analytics information about the network such as: 1) Slice load level analytics; 2) Observed Service Experience analytics; 3) NF load analytics; 4) Network Performance analytics; 5) UE related analytics; 6) User Data Congestion analytics; 7) *QoS* Sustainability analytics.

The *NWDAF*, just like other NFs of the Core Network, uses a subscribe/unsubscribe methodology. It can subscribe to specific events and receive notifications about them. As stated earlier, the NFs are basically web servers that interact with each other via HTTP (HTTP/2 more specifically). As such, the *NWDAF* uses an HTTP/2 based API to communicate with other NFs. It is important to note that the *NWDAF* can also subscribe to other NFs, and act as a consumer. That is how *NWDAF* is able to get analytics information from the network.

Figure 2.5 shows how a NF can obtain analytic data about a User equipment (UE) from the *NWDAF*. It can be observed that the *NWDAF* can obtain information from other NFs (acting like a NF consumer), but it can also provide information for other NFs. The data collection function presented in Figure 2.5, consists of the following steps:

1. An NF subscribes to analytics from the *NWDAF*;
2. *NWDAF* discovers the Application Function (AF) that provides data collection;
3. *NWDAF* subscribes to the AF for UE data collection;
4. The AF collects the UE data;

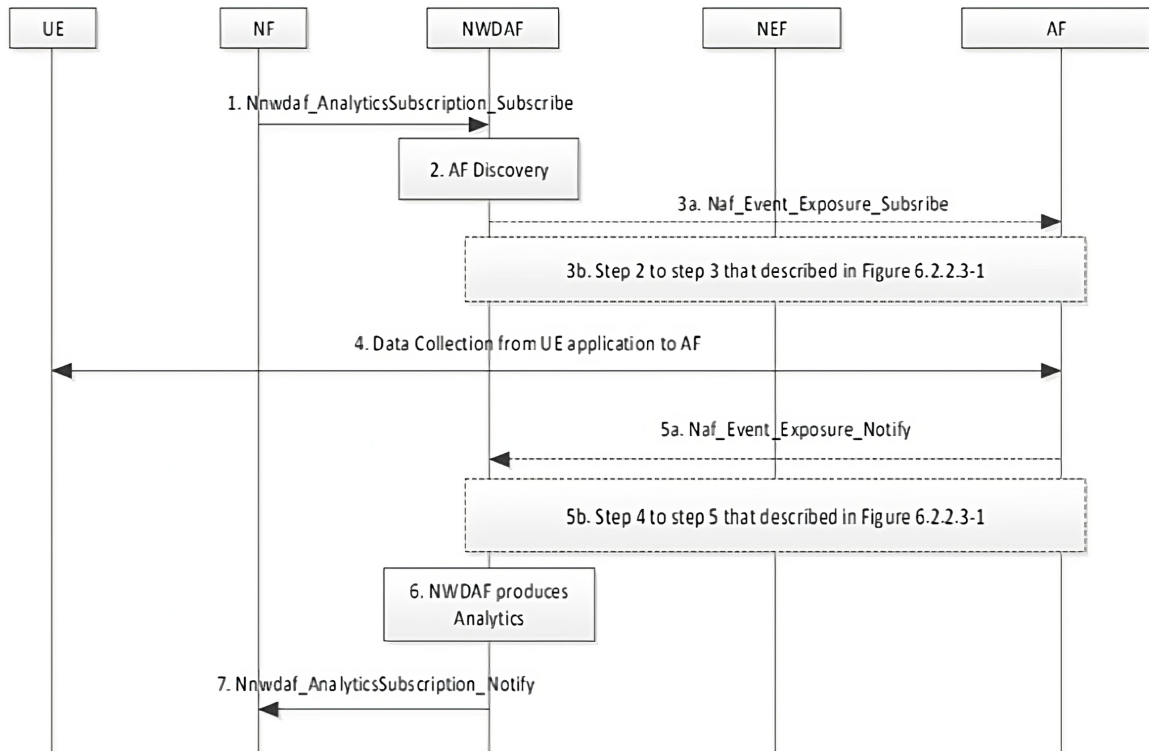


Figure 2.5: Data collection procedure from UE [5]

5. The AF receives the input data from the UE, and processes the data. The AF then notifies the *NWDAF* on the processed data;
6. The *NWDAF* produces analytics using the UE data received from the AF;
7. The *NWDAF* provides analytics to the consumer NF.

The *NWDAF* also has Machine Learning (ML) services. It follows the same subscribe/unsubscribe paradigm. A consumer can be notified when an ML model matching the subscription parameters becomes available, and it can also provide information about the ML model [5].

2.4 QoS

This section provides an overview of Quality of Service (*QoS*) as interpreted by [6]. In theory, all packets on the Internet are treated equally, since the Internet is at its core a best-effort service. However, in practice, things are a bit different. Sometimes VOIP packets can be queued up behind other types of packets, and video stream packets can be throttled to favor other types of packets. It is well known that the performance of the Internet can be improved by treating packets differently.

Moreover, the network requirements for different types of applications are vastly different, e.g., some applications are very sensitive to delays, but can tolerate lower throughput, while others require high throughput and are resilient to delay. As a result, the packets of some applications

can be given different treatment in order to improve their performance, without having noticeable effects on others. These ideas are implemented by some networks using *QoS* techniques.

2.4.1 Traffic Shaping

A router can get congested in two situations: 1) When the packet arrival rate is higher than the output rate of the router; 2) When a large burst of packets arrives at the router. Traffic shaping is a *QoS* technique that controls packet bursts and helps prevent the congestion of routers. There are two main traffic shaping methods: leaky bucket and token bucket. The leaky bucket approach uses a leaky bucket to control the rate at which traffic leaves a node. The leaking bucket maintains a steady outflow rate regardless of the rate of inflow. Any packets that exceed the bucket's capacity are discarded. The size of the bucket and the transmission rate are two characteristics that are unique to this technology and are usually user customizable [7].

In contrast, the token bucket method is less strict in terms of controlling the rate at which traffic leaves a node. Figure 2.6 represents a token bucket that shapes a packet flow. Aside from the Packet Buffer that stores the packets, there is also a Token Counter. The tokens arrive at the Token counter at a constant arrival rate of a tokens per second, and the token counter is limited to B tokens. If a packet with P bytes arrives at the buffer, it is only transmitted if there are at least P tokens in the counter. Then, the P tokens are removed from the counter, and the packet is sent.

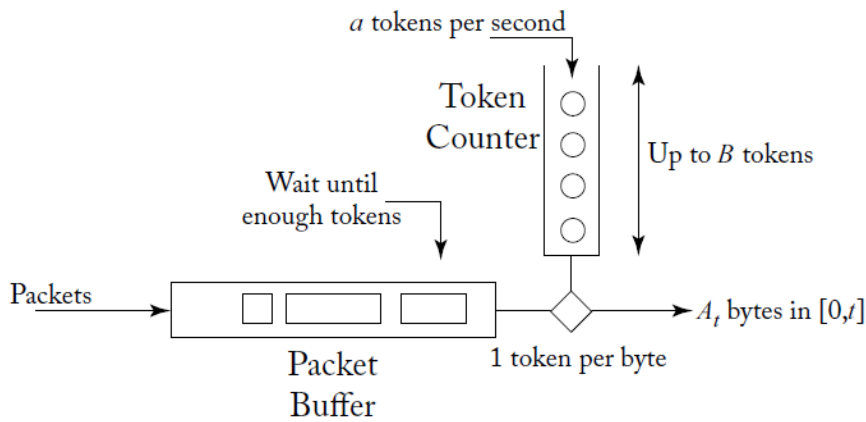


Figure 2.6: Token bucket shapes the traffic and limits its burst size [6]

2.4.2 Scheduling

Scheduling is a *QoS* technique that guarantees a minimum service rate. To control the sharing of a link, amongst packets of different classes, a scheduling strategy known as Weighted Fair Queuing (WFQ) may be used. Before analyzing and defining WFQ, it is necessary to consider an idealized version of WFQ, called Generalized Processor Sharing (GPS).

The GPS system is represented in Figure 2.7. First, K classes are defined to classify each packet that arrives. Each packet is classified accordingly and waits in the corresponding queue to

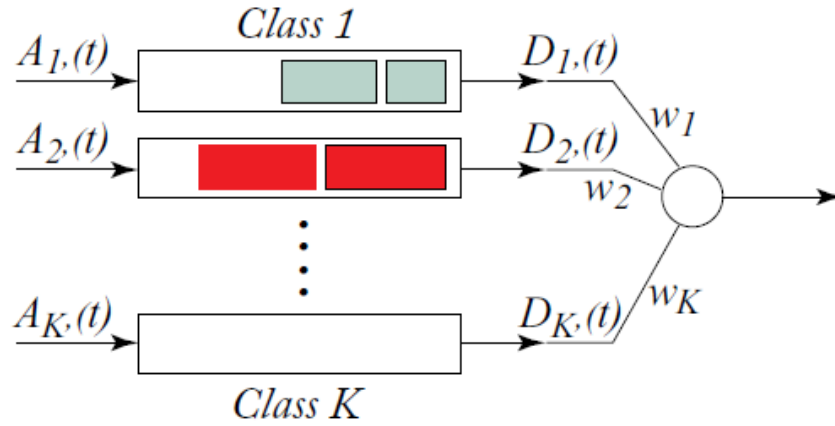


Figure 2.7: Generalized processor sharing (GPS) [6]

be transmitted. To each class is assigned a weight w_k , where a higher weight represents a higher priority. This weight is used by the scheduler to determine the serving rate of the class. The rate at which a packet of each class is served is proportional to their weight:

$$R_k = \frac{w_k C}{W} \quad (2.1)$$

where k represents the class of the packet, C is the line rate out of the router, and W is the sum of all the weights of the queues that are backlogged at time t . Since the scheduler cannot completely separate the packet, i.e., some bits of different packets are mixed, and the queues cannot be served simultaneously, this model is only theoretical.

WFQ is the model that best approximates GPS. The classification and queuing of packets follows the same procedure as GPS. However, just one packet at a time is transmitted by the scheduler, at the line rate. The scheduler calculates the finishing time of the packets and assigns the next packet to be transmitted as the packet that GPS would finish transmitting first among the remaining packets. [6]

2.4.3 Implications on Net Neutrality

The network neutrality topic sparks intense debate, leading to two conflicting viewpoints on the matter. On the one hand, the defenders of an “Open Internet” reason that maintaining neutrality is crucial to prevent large corporations from exerting control over the services provided by the Internet. ISPs should not be allowed to provide better services for specific content providers as they please. The users who cannot afford better services could also be affected by the lack of neutrality. On the other hand, opponents of neutrality argue that differentiation of services is clearly a positive asset for users. Also, ISPs might provide worse services overall for their customers because of the lack of incentive to upgrade their network. Additionally, if ISPs are

unable to deliver high bandwidth content reliably, content providers may be hesitant to offer such content.

To try to satisfy both parties, regulations can be designed to allow service differentiation, while still maintaining part of the network under the neutrality paradigm [6].

2.5 Docker containers

2.5.1 Definition

A container provides a lightweight runtime environment that isolates packages, applications and its dependencies. A container is defined by a docker image, which uses a Dockerfile as a configuration setup. In the Dockerfile you can specify the Operating System (OS) you want to base your container on and modify it by installing the software packages you desire. A Docker container image is thus, an executable package of software that includes all the requirements needed to run an application. A Container image becomes a container at runtime, i.e., when using a proper command, in the case of the docker engine, *docker run*. Containers are independent of the environment, they will run the same, regardless of the Host OS. This makes them reliable applications that can run on a variety of different systems. Each App represents a container and the Docker engine operates on top of the Host OS, as shown in Figure 2.8.

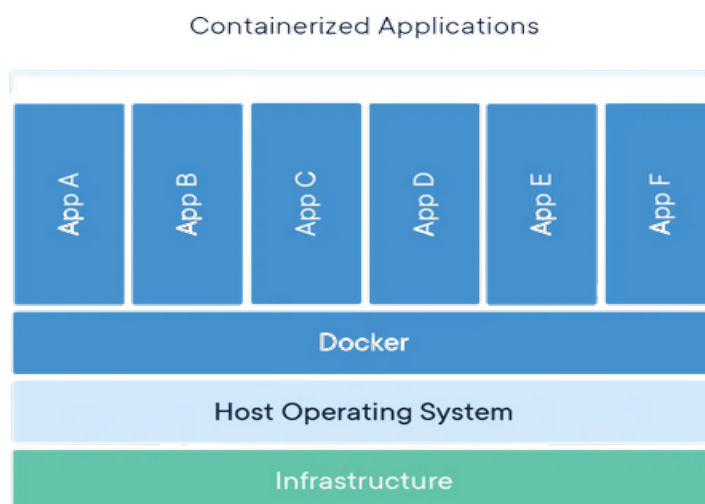


Figure 2.8: General architecture of docker containers [8]

2.5.2 Containers and Virtual Machines

Containers and Virtual Machines (VMs) are similar in the sense that they isolate resources. However, the main difference is that containers virtualize an OS, while VMs abstracts the hardware and software of a computer. Containers are generally considered more lightweight and efficient.

Multiple container instances can run on the same Host OS, each running as a separate process. Additionally, containers typically take up less space than VMs. Even large container images reach about 1 GB of disk space, which is about an order of magnitude lower than a VM usually takes up.

VMs on the other hand, take up more space as they also virtualize the hardware components of a computer. They usually take up tens of GBs, and like a real computer, they have to boot the system. This process makes them slower than containers. The hypervisor, e.g., Virtual Box, allows multiple VMs to run on a single host machine. The diagrams in Figure 2.9 show the fundamental differences between containers and VMs. [8]

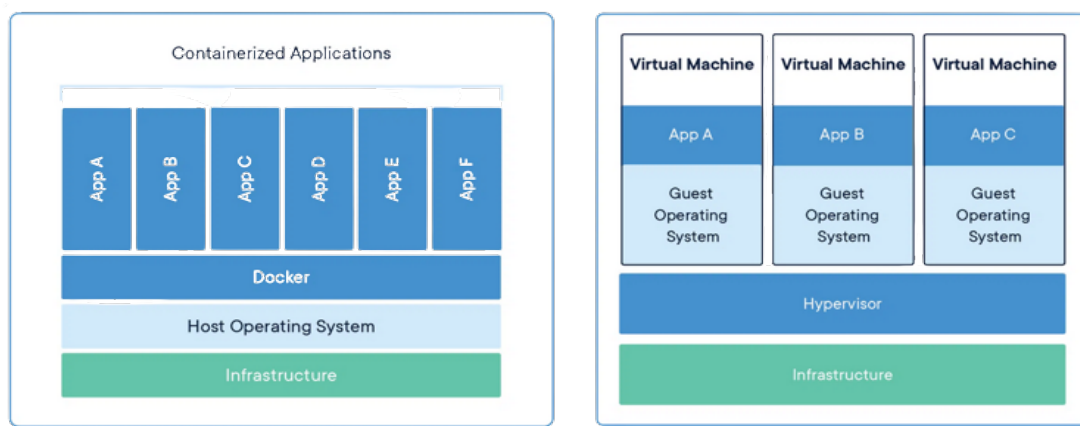


Figure 2.9: Differences between Docker containers and Virtual Machines [8]

2.6 Open-Source 5G software packages

2.6.1 Core Software Packages

To implement the 5G Core Network as a virtual network, there are a couple of software alternatives available. Only open-source software is included in this study.

Open Air Interface (OAI) [9] is one of the projects developed by the OpenAirInterface Software Alliance (OSA) and provides a 5G Core Network (5GC) that currently supports all components to emulate the basic functionality of the Core Network. It has three deployment modes: 1) Minimalist 5GC; 2) Basic 5GC; 3) Slicing 5GC. *NWDAF* and Network Exposure Function (NEF) are said to be released in the future.

Free5GC [10] possesses a Web UI, but unlike OAI, there are no release dates announced for the *NWDAF* and NEF. Moreover, it is implemented using the GO language, which is not as widespread as the C language used by the other options available.

Open5GS [11] also has a Web UI, the documentation is detailed but not as good as OAI, plus no mention of *NWDAF* or NEF for future releases.

Magma [12] is a community project supported by many other entities, such as OAI and Advanced Micro Devices (AMD). It is developed using various programming languages, such as Python, C++ and Go.

Table 2.1 presents the four 5G software packages for the Core network and compares them.

Table 2.1: 5G Core open-source software packages comparison.

	OAI	Free5GC	Open5GS	Magma
Programming Language	C, C++	Go	C	Various
Interface	CLI	CLI, WebUI	CLI, WebUI	CLI
Platform Support	Linux	Linux	Linux, MacOS	Linux, MacOS
NWDAF	Available in the future	No	No	No
NEF	Under-development	No	No	No
Documentation	Very Good	Good	Good	Good

2.6.2 RAN software packages

To implement the 5G RAN as a virtual network, there are a couple of software alternatives available. Only open-source software is included.

OAI [9] offers two possibilities for RAN emulation: 1) OAI-RAN; 2) OAI-Emulator. Both are very complete, since they also implement the physical layer, unlike other similar software.

UERANISM [13] is a 5G UE and RAN (gNodeB) simulator. It can implement all the basic functions of UE and gNodeB, but some features are incomplete. Its simplicity makes it a suitable choice for testing. It emulates the RAN independently of the Core Network. It is no longer under active development.

Free5GRAN [14] is a 5G RAN stack. It is currently under active development, however it proposes a strategy centered on seamless integration with Software-Defined Radio (SDR).

srsRAN [15] was initially a 4G LTE project known as srsLTE. With the release of 5G new radio (NR), the developers started implementing new features to the code and renamed the project to srsRAN. It is under active development and new features are still being added every in release.

Table 2.2 presents the five 5G software packages for the RAN and compares them.

Table 2.2: 5G Core open-source software packages comparison.

	OAI-RAN	UERANSIM	Free5GRAN	srsRAN	OAI-Emulator
Programming Language	C, C++	C++	C, C++	C, C++	C, C++
SDR Compatibility	Yes	No	Yes	Yes	No
Platform Support	Linux	Linux	Linux	Linux	Linux
Documentation	Very Good	Good	Good	Good	Very Good

Chapter 3

State of the Art - Previous Works on Detection of Traffic Differentiation

3.1 Wired Networks

In the literature, there are various studies regarding the detection of traffic differentiation by *ISPs*. These have drawn considerable attention for over fifteen years.

One of the first studies was carried out by [16], which measured the blocking of traffic belonging to a specific application or class of applications in their designated TCP or UDP port. It is claimed that their findings are among the first measurements of network neutrality.

The methodology used relies on P2P network protocols, more specifically on the Gnutella protocol. By managing a Rogue Super Peer (RSP) and a measurement host, it is possible to redirect a client trying to join the Gnutella network to the measurement host's IP address and to a desired port. Figure 3.1 depicts the base methodology used in this study.

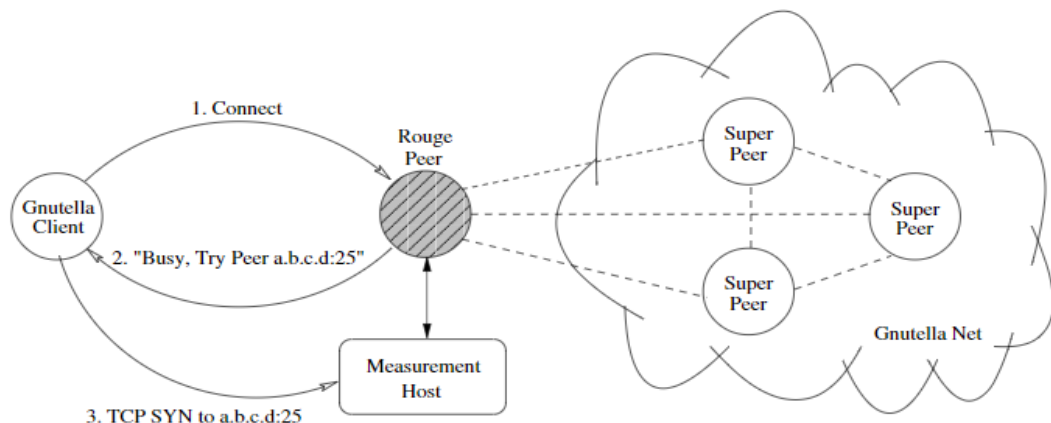


Figure 3.1: Methodology: The Rogue SuperPeer joins the Gnutella network. (1) Client attempts to connect; (2) RSP rejects, referring the client to a measurement host under our control; (3) By correlating connections, Internet port blocking is mapped. [16]

3.1.1 NetPolice

In 2009, [17] developed NetPolice as a system to verify traffic differentiation in backbone *ISPs* by taking loss measurements from end hosts. To accomplish this, numerous probes are launched in a distributed manner from a wide range of end systems. The study was conducted on 18 large *ISPs* across 3 continents over the span of 10 weeks.

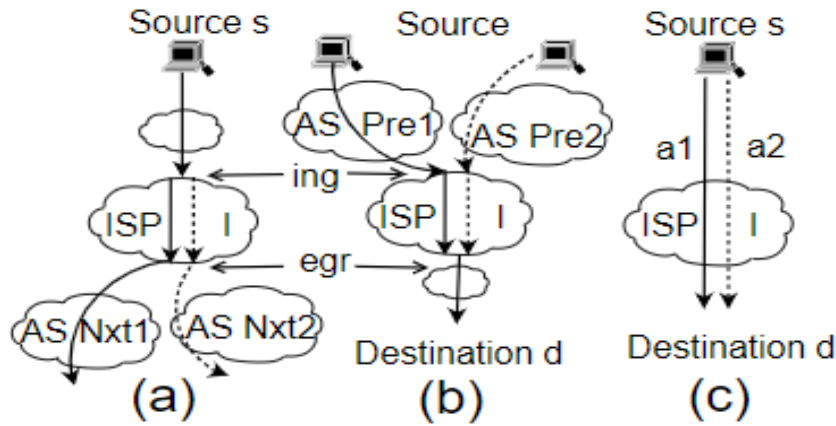


Figure 3.2: Detecting various types of differentiation with end-host based probing [17]

Figure 3.2 showcases the methodology used for detecting *TD* with NetPolice. In Figure 3.2(a) the end host (Source *s*) probes two different destination paths through the same Ingress and Egress of *ISP I*. This allows for the detection of differentiation based on the next-hop ASes. Figure 3.2(b) treats the opposite case, in which the previous-hop ASes are different. In Figure 3.2(c) the end host sends two probes through the same Ingress and Egress of *ISP I* with the same final destination but for different applications (*a1* and *a2*). By comparing the performance of both paths, it can be detected eventual different treatment of the traffic due to its application type.

The authors of [17] state that this study is a crucial step towards the increase of transparency on the Internet. They also claim that, in contrast to previous researches, which focus on Broadband *ISPs*, NetPolice studies the detection of traffic differentiation in backbone *ISPs*. This requires the development of an intelligent path selection algorithm capable of measuring numerous *ISPs* internal paths.

3.1.2 Glasnost

In 2010, Glasnost was introduced by [18], with the purpose of identifying if an *ISP* was performing application-specific traffic shaping and reporting back that information to ordinary Internet users. The design of a system with this goal poses three main challenges:

- The system must be simple and generate quick results to gain popularity among users;
- To not have negative effects on the reputation and business of *ISPs*, the system should be resilient to measurement noise to avoid false accusations of differentiation;

- To keep up with the ever-changing differentiation policies of *ISPs* around the world, the system must be able to be upgraded;

Glasnost employs a client-server architecture. Clients connect to a Glasnost server and download and run a variety of tests. Each test creates a flow of application-level data that is utilized to measure the path between the client and the server. The data in these flows is specifically designed to identify traffic differentiation on the link between the client and the server.

Figure 3.3 provides an overview of how clients measure their Internet paths. A client first connects to a central website, which then redirects to a Glasnost measurement server. This dynamic redirection provides load balancing between the measurement servers and also simplifies the process of adding more serves to the redirection list.

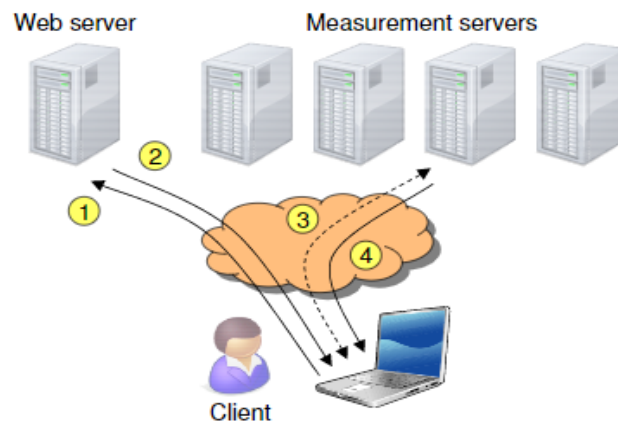


Figure 3.3: The Glasnost system.(1) The client contacts the Glasnost webpage. (2) The webpage returns the address of a measurement server. (3) The client connects to the measurement server and loads a Java applet. The applet then starts to emulate a sequence of flows. (4) After the test is done, the collected data is analyzed and a results page is displayed to the client [18].

Figure 3.4 shows the user interface presented by the measurement server after the redirection of the client. It is a simple interface that lets the user select what type of traffic he/she wants to test. To start the test, the user just has to click on the “Start testing” button. Then, the Browser on the client’s side downloads a java Applet that exchanges packets with the measurement server.

Glasnost has been discontinued, as can be seen in [19]. The reason presented to abandon the project was that Java Applets no longer work on modern Web Browsers.

3.1.3 POPI

Also in 2010, [20] presented a user-level tool named POPI, which can detect packet forwarding prioritization (PFP) in routers through an end-to-end measurement. Three different metrics could be used to infer PFP: loss, delay and Out-Of-Order. The packet Loss based method has larger overhead, because differences only become observable when the associated link router queue becomes full and packets start to be dropped. Despite this, the loss metric was chosen as the best

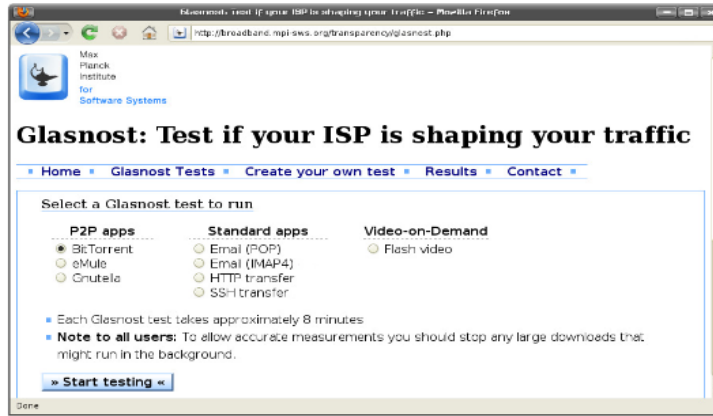


Figure 3.4: The Glasnost web interface [18]

metric to infer PFP. It can detect various kinds of *QoS* techniques, and it is also less affected by parallelism in the forwarding paths.

The POPI tool was evaluated using statistical analysis, a simulation and a wide-area experiment in PlanetLab. The POPI tool was used to analyze 156 paths among 162 PlanetLab sites. Also, 15 paths were flagged as having multiple priorities, of which, 13 were validated by loss rate measurements. The authors of [20] concluded that POPI can infer the router's packet forwarding priority with high precision.

3.1.4 DiffProbe

Still in 2010, [21] proposed Differential Probing or DiffProbe, with the objective of detecting if *ISPs* are employing *QoS* techniques to discriminate some traffic flows of their customers. It mainly focuses on delay discrimination and loss discrimination, by comparing a Probing flow *P* with an Application flow *A*.

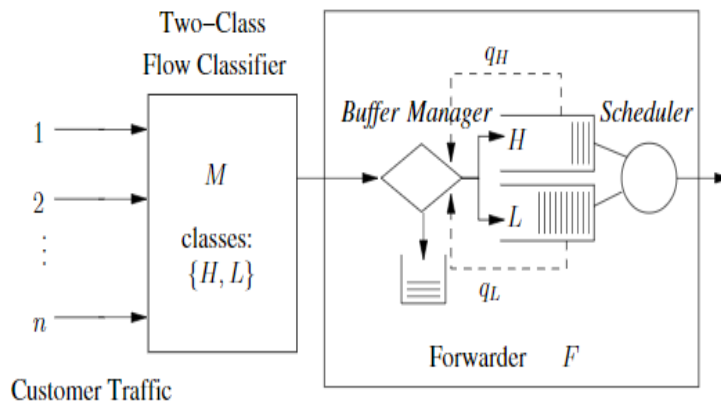
Figure 3.5: Model of access *ISP* discrimination [21]

Figure 3.5 describes the standard model used to implement DiffProbe. n users are connected to an *ISP* I and the user flows go through a classifier M . The flows are then marked as being High priority (H) or Low priority (L). If the Low priority flows are discriminated by the *ISP*, it is expected to see the traffic go to a different queue where a discriminatory packet schedule is applied. On the contrary, if the *ISP* does not discriminate traffic, the First-Come First-Served (FCFS) schedule should be applied. All the traffic should go to a single queue and packets should be forward as soon as they arrive, assuming the queue is empty.

The DiffProbe tool was tested on multiple *ISPs*, and the results show that the tool is accurate, as long as the discrimination of the traffic is perceivable by the user.

3.2 Mobile Networks and Advancements in Content-Based Differentiation

All the studies referenced up until this point, attempt to detect discrimination of traffic, however, they only focus on wired networks. This trend continued throughout the years, as evidenced by the statement of the 2018 study on neutrality in European Mobile Networks [22]: “Research about net neutrality mostly focused on the wired Internet, and little effort has been devoted to wireless scenarios”.

In 2019, [23] performed a large-scale study to assess the impact of content-based traffic differentiation methods applied in operational mobile networks, and fixed-rate bandwidth constraints were discovered to be the most frequently seen behavior.

This study points out two important differences between Internet traffic compared to previous years. Nowadays, the primary source of Internet traffic is video streaming platforms and not BitTorrent. Likewise, cellular network usage has skyrocketed in recent years, as more people than ever before access the Internet via mobile devices.

3.2.1 Wehe

With this in mind, the authors of [23] designed a mobile app named *Wehe* [24] that tests apps like YouTube, Netflix, Amazon Prime Video, NBC Sports, Vimeo, Spotify, Skype, and more.

A one-year study was conducted to evaluate the content-based traffic differentiation policies deployed in operational networks. The methodology employed involved gathering data from individual devices and using this information to make reliable and statistically significant conclusions about differentiation practices, such as fixed-rate bandwidth limits (throttling) and delayed throttling techniques.

To test for differentiation practices, the study used the “record and replay” technique. Network traffic generated by an application (e.g., streaming a video using the YouTube app) is recorded and included in the corresponding app’s trace. During testing, this recorded traffic is replayed between the user’s device and a *Wehe* server. The replay process includes two back-to-back pairs: one with the original payload and another with the same traffic patterns but obscured payload, named

control replay, to evade detection by deep packet inspection devices. Both plaintext and encrypted flows are supported by this approach. During the execution of *Wehe*, users have the option to choose the apps they want to test, and each test can involve up to two replay pairs. After replaying the traces for a specific app, *Wehe* examines the presence of differentiation and presents the result to the user.

This study identified Traffic Differentiation in both cellular and Wi-Fi networks, comprising 30 ISPs in 7 countries. The findings indicate that *ISPs* utilize various throttling techniques and that the majority of throttling affects video streaming.

3.3 Conclusion

Table 3.1 provides a summary of all the tools discussed in this chapter, highlighting the main differences between each of them. To underline that the only reviewed tool that applies to mobile networks is *wehe*. All the tools and techniques used in these works attempt to detect traffic differentiation (*TD*) and network neutrality violations without information about the *ISP*'s internal network. This lack of information makes it more complicated to detect such practices, as explained in the next Chapter.

Table 3.1: Summary of the tools described and their characteristics

Tool	Year	Purpose	Methodology	Metrics
[16]	2007	Measure network neutrality and aversion	P2P networks (Gnutella protocol)	Blocking of traffic based on TCP/UDP ports
NetPolice [17]	2009	Verify traffic differentiation	Distributed probes from end hosts	Loss measurements
Glasnost [18]	2010	Identify application-specific shaping	Client-server architecture with flow emulation	Path measurements, traffic differentiation
POPI [20]	2010	Detect packet forwarding prioritization	End-to-end measurement using loss metric	Loss, delay, and out-of-order metrics
DiffProbe [21]	2010	Detect ISP discrimination	Model-based approach and traffic comparison	Delay and loss discrimination
Wehe [23]	2019	Assess impact of content-based shaping	Mobile app testing popular streaming platforms	Throttling techniques, video streaming

Chapter 4

System Specification, Design and Implementation

This dissertation primarily focuses on addressing the following objectives: 1) Increasing the transparency of traffic management practices; 2) Demonstrate that the *NWDAF* can be used as a tool to assess *TD*; 3) Demonstrate how a Regulator can obtain performance information about the 5G network to evaluate potential discriminatory practices from the *ISPs*. By achieving these objectives, the study aims to develop a new use case for the Network Data Analysis Function (*NWDAF*) capable of identifying traffic differentiation within the *5G Core Network*.

One of the main shortcomings of previous studies on the subject was that the mechanisms used were located outside the *ISP* network. As a result, traffic discrimination based on IP addresses, peering arrangements, interconnection congestion, traffic volume, or other characteristics other than IP payloads could not be detected [17]. Typically, an *ISP* could simply argue that the abnormal traffic behavior presented to them by network regulators was caused by external circumstances beyond the *ISP*'s control.

To overcome this limitation, the proposed solution involves the utilization of the Network Data Analysis Function (*NWDAF*). By employing the *NWDAF*, network regulators will be able to access data directly from the 5G network, including metrics such as packet loss, delay, and throughput, in a timely manner.

4.1 System Specification

Figure 4.1 showcases the proposed architecture to implement and test the *NWDAF*. Two traffic streams of the same class (*eMBB*) from different Content and Application Providers (*CAPs*) will be compared. *bCAP* is the base *CAP* and will be used as a reference. *tCAP* is the target *CAP* and will be the one monitored to detect traffic differentiation from the part of the *ISP*. t_1 represents the time in which a packet from *tCAP* arrives at the queue of the *UPF*. t_2 represents the time that the same packet leaves the *UPF* to be routed to the corresponding gNB. The *NWDAF* can obtain this information from the Operations Administration and Maintenance (OAM) service and process the

data. With this information, it is possible to determine the delay of the packet, infer packet loss and calculate the throughput of the whole stream of packets. An outside entity, in this case the Telecom Regulator, could have access to this data via the Network Exposure Function (NEF).

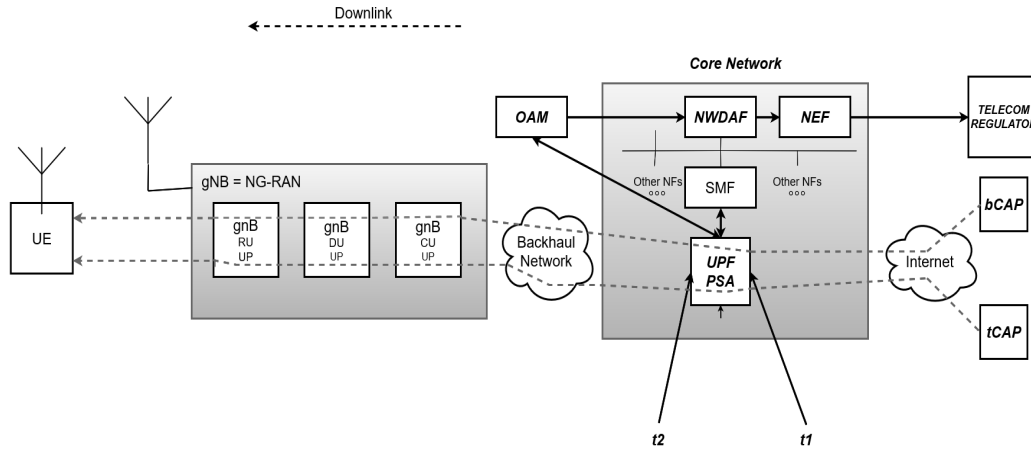


Figure 4.1: Diagram of the 5G network and the proposed solution

There are various types of discriminatory practices that an *ISP* can apply through traffic management mechanisms, such as blocking, filtering, throttling, and paid prioritization. This dissertation places a specific emphasis on the detection of throttling, mostly because it is the most common practice. Throttling has a detrimental influence on network performance and could be detected using performance measurements. Three metrics are involved in the detection of throttling: 1) Packet delay, 2) Packet loss ratio, and 3) Throughput. The impact of throttling will be noticeable on the throughput of a network flow, but throughput alone cannot provide evidence of throttling. To infer throttling accurately, packet delay and packet loss ratio are key metrics to consider. If the *tCAP* has a lower throughput than the *bCAP*, it is not possible to claim that throttling is happening. However, if the *tCAP* not only has a lower throughput but also exhibits higher delay/latency or packet loss ratio compared to the *bCAP*, it becomes more likely that throttling is taking place. This premise only makes sense if the *QoS* requirements are similar for both traffic flows.

4.2 System Design

Figure 4.2 illustrates the approach used to develop and test the system specified in the previous section. This diagram goes into a more detailed representation of the components, taking into consideration the environment and the software used to implement them. The complete system, including the *5G Core Network*, was all created inside a host PC. The *5G Core Network* was implemented using Open Air Interface (OAI) [9]. Further description and specifics regarding the diagram's components can be found below.

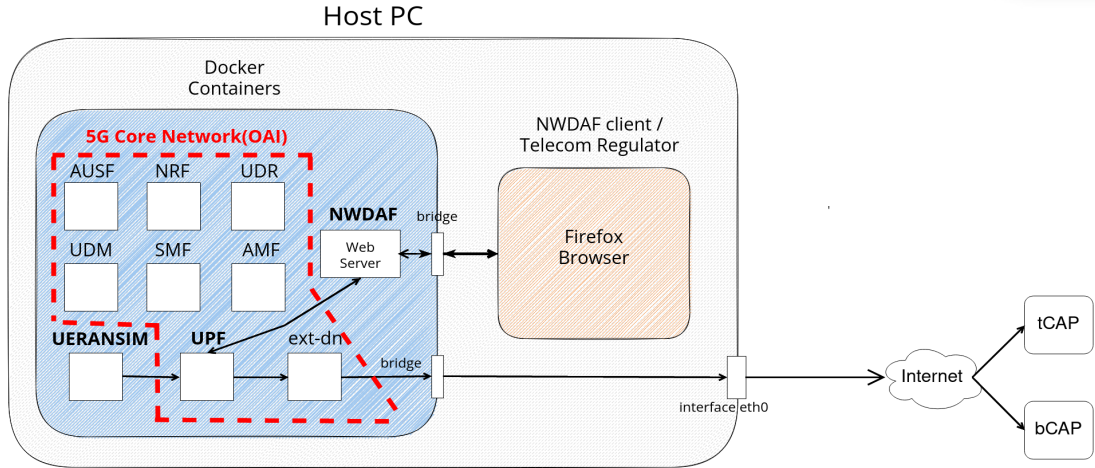


Figure 4.2: Design used to implement and test the proposed solution

4.2.1 5G Core Network

The decision to utilize the 5G open-source software Open Air Interface (OAI) for simulating the *5G Core Network* was based on several factors, including the extensive availability of information and tutorials related to OAI compared to other options. Furthermore, it is compatible with *UERANSIM* [13], which will be used to implement the User Equipment and the gNB. OAI uses the *Docker container* technology to deploy various network nodes corresponding to components of the *5G Core Network*. *Docker containers* [8] offer benefits such as easy scalability, portability, and isolation of Network Functions (NFs). The user may choose which NFs to deploy, with the minimalist deployment mode consisting of just AMF, SMF, NRF, and UPF. The basic deployment mode was used in this work. It is composed of the AMF, SMF, NRF, UDM, AUSF, and UDR NFs, as well as the UPF. The OAI 5G Core network and its components are highlighted in Figure 4.2 by the red dotted shape.

4.2.2 User Equipment (UE) and gNB

The *UERANSIM* emulator was employed to simulate both the User Equipment (UE) and the gNodeB (gNB) in this dissertation. Since the focus of the work did not involve configuring the radio components of the 5G Network, *UERANSIM*, a simple simulator, proved to be suitable for simultaneously emulating the UE and the gNB. Moreover, *UERANSIM* is recognized for its compatibility and seamless integration with OAI. *UERANSIM* is a *Docker container* as well. *UERANSIM* will simulate the UE that will connect to two video streams provided by the CAPs.

4.2.3 NWDAF

The Network Data Analytics Function (NWDAF) was added to the NFs deployed by OAI, as another container in the network. The arrows between the NWDAF and the UPF in Figure 4.2

indicate that information is shared between the two NFs. In practice, a shared directory was set up as a *Docker* volume, allowing the host PC to share a directory to the *NWDAF* and the *UPF*. The *NWDAF* hosts a web server that initiates the procedure of analytics collection from the *UPF* and exposes the results to a web page. The arrows between the *NWDAF* and the client (the Firefox browser in the host PC) in Figure 4.2 identify this connection.

4.2.4 Application Servers (*tCAP* and *bCAP*)

These *CAPs* will consist of two distinct web pages that host a video stream, e.g., YouTube and “RTP Play”. They will be accessed by the *UE* (*UERANSIM* container) and both will be played simultaneously to compare the two streams in terms of metrics, such as packet loss, packet delay, and throughput. The *ISP* will then interfere with one of the streams, affecting its performance. This stream will be the target *CAP* (*tCAP*) and the one unaffected will be the base *CAP* (*bCAP*). Their designations can be interchanged. The terminology only depends on which one is being manipulated by the *ISP*.

4.2.5 Telecom Regulator

In a real application of the system, the Telecom Regulator would typically exist as a node on the Internet, outside the *5G Core Network*. Then, it would access the information through the NEF. In this work, since we are dealing with a self-contained system, the Telecom Regulator will be represented by the host PC. As the primary objective of this dissertation is to demonstrate the proper functioning of the developed *NWDAF*, an exact representation of the Telecom Regulator is not a top priority. Instead, the host PC will fulfill the role of the Telecom Regulator within the system. To test the developed *NWDAF*, a browser running on the host PC will access the web page provided by the *NWDAF* using its IP address. The intention of this explanation is to clarify that while the actual setup involving the Telecom Regulator may differ in real-world scenarios, for the purposes of this research, the host PC serves as a suitable substitute. The focus remains on evaluating the performance and functionality of the *NWDAF*.

4.3 System Implementation

CONTAINER ID	NAMES	IMAGE	STATUS
2f1acc27c4a0	ueransim	ueransim:latest	Up About a minute (healthy)
27d62e43595b	oai-NWDAF	nwdaf	Up About a minute
842e54b86fe8	oai-smf	oaisoftwarealliance/oai-smf:v1.5.0	Up About a minute (healthy)
5f07febe3f55	oai-amf	oaisoftwarealliance/oai-amf:v1.5.0	Up About a minute (healthy)
c745178ed288	oai-ausf	oaisoftwarealliance/oai-ausf:v1.5.0	Up About a minute (healthy)
56b879e9889a	oai-udm	oaisoftwarealliance/oai-udm:v1.5.0	Up About a minute (healthy)
14a4aea5b38a	vpp-upf	oaisoftwarealliance/oai-upf-vpp:v1.5.0	Up About a minute (healthy)
300b42fc405b	oai-udr	oaisoftwarealliance/oai-udr:v1.5.0	Up About a minute (healthy)
af97e53ced08	oai-ext-dn	oaisoftwarealliance/trf-gen-cn5g:latest	Up About a minute (healthy)
70f0850fbac5	oai-nrf	oaisoftwarealliance/oai-nrf:v1.5.0	Up About a minute (healthy)
58764eefb2b6	mysql	mysql:8.0	Up About a minute (healthy)

Figure 4.3: Containers of all the components used to implement the system

Figure 4.3 showcases all the containers deployed to implement the system. Open Air Interface (OAI) provides all the components of the *5G Core Network* presented in Figure 4.3, except *UER-ANSIM*, which is launched separately, and the *NWDAF*, which was developed as part of the work of this dissertation.

Figure 4.4 showcases a network diagram of all the containers belonging to the system. The diagram can be interpreted as a zoom-in on the box surrounding the *Docker Containers* in Figure 4.2. The IP addresses of all the Network Functions (NFs) are represented, as well as all the important network interfaces on the network. At the top, the three networks regarding the 5G network are presented. In the next sections, a more detailed explanation of the implementation of each component will be given along with their configurations.

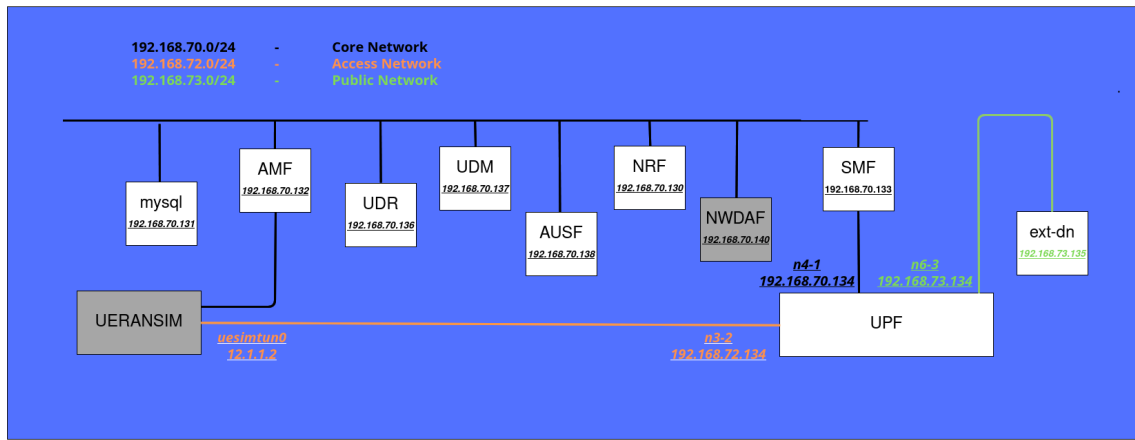


Figure 4.4: Network diagram of all the container running on the host PC

4.3.1 5G Core Network

To define and launch the containers associated with the *5G Core Network*, a *Docker-compose.yaml* [25] file was used. In it, it is possible to create various services associated with a Docker image [26] and a Docker container. Apart from selecting the *Docker* image and the *Docker* container name, the options provided include volumes [27], environment variables [28], healthcheck [29], and networks. The volumes and the environment variables options will be useful later on when setting up the *UPF* and the *UERANSIM*.

The networks attributed to the services are defined at the end of the file. Three networks are created in this file: *public_net*, *public_net_access* and *public_net_core*. Each network is assigned a bridge to allow connection to the host PC: "*demo-oai*", "*cn5g-access*", and "*cn5g-core*", respectively. Figure 4.5 depicts the network configurations in the file. When the file is executed (*docker-compose up*) all the containers defined will be executed, in this particular case: *mysql*, *oai-udr*, *oai-udm*, *oai-ausf*, *oai-nrf*, *oai-amf*, *oai-smf*, *vpp-upf*, and *oai-ext-dn*. The "*demo-oai*" and the "*cn5g-core*" bridges are presented in Figure 4.2. The "*demo-oai*" bridge links the *NWDAF* with the host PC, while the "*cn5g-core*" bridge connects the *5G Core Network* with the standard interface of the host PC.

```

networks:
  public_net:
    driver: bridge
    name: demo-oai-public-net
    ipam:
      config:
        - subnet: 192.168.70.0/24
      driver_opts:
        com.docker.network.bridge.name: "demo-oai"
  public_net_access:
    name: oai-public-access
    ipam:
      config:
        - subnet: 192.168.72.0/24
      driver_opts:
        com.docker.network.bridge.name: "cn5g-access"
  public_net_core:
    name: oai-public-core
    ipam:
      config:
        - subnet: 192.168.73.0/24
      driver_opts:
        com.docker.network.bridge.name: "cn5g-core"

```

Figure 4.5: Docker networks defined in the *docker-compose-basic-vpp-nrf.yaml* file

To launch the entire system, including the *UERANSIM* and the *NWDAF*, a shell script [30] named *start_5G_VPP_NRF.sh* is used. All the commands in this script will be explained independently along the document. To run just the *docker-compose-basic-vpp-nrf.yaml* file, the command used is depicted in the next line:

```
sudo python3 ./core-network.py -type start-basic-vpp -scenario 1
```

The *Python* script *core-network.py* is executed taking in as arguments *-type start-basic-vpp*, which refers to the *docker-compose-basic-vpp-nrf.yaml* file and *-scenario 1*, indicating that the NRF NF will be used. The *core-network.py* file is a large *Python* script that deploys the 5G Core Network based on the *docker-compose* file specified (there are numerous *docker-compose* files available as part of the OAI software package) and starts a healthcheck to verify that all the OAI components are working properly. The healthcheck is configured in the respective *docker-compose* file. If there are no problems with the components, *healthy* should appear in the status of the Docker containers as seen in Figure 4.3. This is valid for all the containers, except the *NWDAF* container, *nwdaf*, which was not set up to run an healthcheck.

Table 4.1 provides a summary of the OAI 5G Core Network components along with their corresponding IP addresses.

Table 4.1: Core Network components and their respective IP addresses

Core Network component	Container name	IP address
NRF	oai-nrf	192.168.70.130
mysql	mysql	192.168.70.131
AMF	oai-amf	192.168.70.132
SMF	oai-smf	192.168.70.133
UPF	vpp-upf	192.168.70.134
UDR	oai-udr	192.168.70.136
UDM	oai-udm	192.168.70.137
AUSF	oai-ausf	192.168.70.138

4.3.2 UPF

4.3.2.1 Docker image

The *UPF* container was customized to meet the specific requirements for this dissertation. To achieve this, modifications were made to the associated docker image. In addition to the initially installed packages, some more were added, and a new docker image was created. *xauth* [31] is required to run graphical applications, *wireshark* [32] and *tshark* [33] are necessary to capture packets within the container. Initially, the ability to run graphical applications was used to capture and analyze packets with *wireshark*. After getting a feel of the profile of the traffic and its characteristics, a different approach was taken. A script with *tshark* was developed with concrete commands to capture and filter the traffic. Two *Python* packages, namely *matplotlib* [34] and *pandas* [35], were also installed. The purpose of these packages will be explained later.

4.3.2.2 Docker-compose configuration

Using the *docker-compose-basic-nrf.yaml* the *UPF* was configured with some additional options. The options added to the *docker-compose-basic-nrf.yaml* can be observed in Figure 4.6. The environment variable [28] *DISPLAY* was created and assigned the value of the *DISPLAY* variable of the host PC. This, along with the first volume (*X11*) enables the use of graphical interface applications within the container. The second volume is a shared directory (*UPF_NWDAF*) between the host PC and the *UPF* container. This allows for code development on the host PC instead of relying solely on the container's terminal. It also automatically loads the directory in the container upon launch, preventing the container's file system from being restored to its default state. The last volume added corresponds to a *wireshark* profile configured to compute the *hash* of the captured packet data within the *UPF*. This profile will be utilized to calculate metrics from the capture files provided by *tshark*.

```

- DISPLAY=$DISPLAY
volumes:
- /tmp/.X11-unix:/tmp/.X11-unix
- /home/ruben/Desktop/Tese/share/UPF_NWDAF:/UPF_NWDAF
- /home/ruben/Desktop/Tese/Wireshark_hash-profile_share:/root/.config/wireshark/profiles/hash

```

Figure 4.6: Packages installed as part of the new *UPF Docker* image

4.3.2.3 Capture script

To capture packets going through the *UPF*, the script *captures.sh* was developed. The *UPF* has three network interfaces, one for each network defined by OAI. Table 4.2 presents a list of the network interfaces of the *UPF*, along with their respective IP addresses and the network to which each interface connects to.

Table 4.2: *UPF* interface names, IP addresses and associated networks

Interface name	IP address	Network
<i>n3-2</i>	192.168.72.134	public_net_access
<i>n6-3</i>	192.168.73.134	public_net_core
<i>n4-1</i>	192.168.70.134	public_net

The *n6-3* interface communicates with the ext-dn container, which in turn directs the traffic to the Internet through the *cn5g-core* bridge (192.168.73.1). The *n3-2* interface connects the *UPF* with the *UE* (*UERANSIM*). The *n4-1* interface allows the *UPF* to exchange information with the NFs of the *5G Core Network*. As the goal is to capture traffic coming from the Internet into the *UE*, the *n6-3* and the *n3-2* interfaces are the ones whose packets should be captured and analyzed.

The *captures.sh* script uses *tshark* to start a capture of five seconds for interface *n6-3* and *n3-2*. The captures are executed as concurrent processes so that they can start at the same time. Then, the results are saved into two different *pcap* files. The command to accomplish this objective is as follows:

```

tshark -C hash -i n6-3 -a duration:5 -w /UPF_NWDAF/n6-3.pcap &
tshark -C hash -i n3-2 -a duration:5 -w /UPF_NWDAF/n3-2.pcap

```

The saved files are then filtered using another *tshark* command into two other capture files. Two *csv* files are then generated from these capture files, containing two fields obtained from the information present in the packets.

```

case $1 in
    "rtp" | "RTP")
        filter_1="udp && ip.dst==12.1.1.2 && ip.src==146.75.90.192"
        proto_1="HTTP/3"
        stream_A=$1
    ;;
    "tvi" | "TVI")
        filter_1="ip.dst==12.1.1.2 && ip.src==195.23.102.0/24"
        proto_1="HTTP/2"
        stream_A=$1
    ;;
    "sic" | "SIC")
        filter_1="ip.dst==12.1.1.2 && tcp && ip.src==13.225.244.0/24"
        proto_1="HTTP/2"
        stream_A=$1
    ;;
    "youtube" | "YOUTUBE")
        filter_1="udp && ip.dst==12.1.1.2 && ip.src==195.8.0.0/16"
        proto_1="HTTP/3"
        stream_A=$1
    ;;
    *)
        ip=$(echo "$1" | cut -d' ' -f1)
        proto_1=$(echo "$1" | cut -d' ' -f3)
        transport_proto=$(echo "$1" | cut -d' ' -f4-)
        filter_1="$transport_proto && ip.dst==12.1.1.2 && ip.src==$ip"
        stream_A="custom_A"
esac

```

Figure 4.7: *captures.sh* script processing of the command line arguments

These fields depend on the protocol of the packets to analyze. If the expected protocol is HTTP/2 or TCP, then the fields of interest are the timestamp of the packets (*frame.time_epoch*) and the TCP sequence number (*tcp.seq*). For HTTP/3 or UDP, the fields would be the timestamp of the packets (*frame.time_epoch*) and the hash value of the data of the packets (*data.md5_hash*). The custom *wireshark* profile defined in the configuration of the *UPF* container (Figure 4.6), named *hash*, has to be used in this case.

The filters used by the *tshark* commands, the name of the traffic streams, and the protocol names are defined by processing two command line arguments passed to the script at runtime. The script receives the name of two video streams to be analyzed as the first two arguments. The logic used to process these values is displayed in Figure 4.7. For the second line argument, the code used is similar. There are four pre-defined video streams: “RTP”, *TVI*, *SIC*, and *YOTUBE*. The experiments conducted in *wireshark* as part of this work allow for the selection of the appropriate filter and protocol for these streams. There is also a default option, which parses the first argument and defines the value of the variables of interest. More will be explained about this in the *NWDAF* section.

The *filter_1*, the *stream_A*, and the *proto_1* store these values. The same procedure as Figure 4.8 is then repeated for the second stream, with the variables *filter_2*, *stream_B*, and *proto_2* instead. Following this, the *Python* script *calculate_metrics.py* is executed taking in five command line arguments:

```

# stream 1
if [ "$proto_1" = "HTTP/3" ]; then
    tshark -C hash -r /UPF_NWDAF/n6-3.pcap -Y "$filter_1" -w /UPF_NWDAF/n6-3_${stream_A}.pcap
    tshark -C hash -r /UPF_NWDAF/n3-2.pcap -Y "$filter_1" -w /UPF_NWDAF/n3-2_${stream_A}.pcap

    tshark -C hash -r /UPF_NWDAF/n6-3_${stream_A}.pcap -T fields -e frame.time_epoch -e data.md5_hash \
    -E header=y -E separator=, -E quote=d -E occurrence=f > /UPF_NWDAF/n6-3_${stream_A}.csv 2>/dev/null

    tshark -C hash -r /UPF_NWDAF/n3-2_${stream_A}.pcap -T fields -e frame.time_epoch -e data.md5_hash \
    -E header=y -E separator=, -E quote=d -E occurrence=f > /UPF_NWDAF/n3-2_${stream_A}.csv 2>/dev/null

elif [ "$proto_1" = "HTTP/2" ]; then
    tshark -r /UPF_NWDAF/n6-3.pcap -Y "$filter_1" -w /UPF_NWDAF/n6-3_${stream_A}.pcap >/dev/null 2>&1
    tshark -r /UPF_NWDAF/n3-2.pcap -Y "$filter_1" -w /UPF_NWDAF/n3-2_${stream_A}.pcap >/dev/null 2>&1

    tshark -r /UPF_NWDAF/n6-3_${stream_A}.pcap -T fields -e frame.time_epoch -e tcp.seq \
    -E header=y -E separator=, -E quote=d -E occurrence=f > /UPF_NWDAF/n6-3_${stream_A}.csv 2>/dev/null

    tshark -r /UPF_NWDAF/n3-2_${stream_A}.pcap -T fields -e frame.time_epoch -e tcp.seq \
    -E header=y -E separator=, -E quote=d -E occurrence=f > /UPF_NWDAF/n3-2_${stream_A}.csv 2>/dev/null
fi

```

Figure 4.8: *tshark* commands to filter and create a csv file with two fields

```

python3 /UPF_NWDAF/calculate_metrics.py "$stream_A" "$stream_B"
"$proto_1" "$proto_2" "$cur_time"

```

4.3.2.4 Calculate metrics script

The *calculate_metrics.py* script takes in as arguments five variables. The first four, namely, *stream_A*, *stream_B*, *proto_1*, and *proto_2* were described in the previous section. The last argument *cur_time* is related to the *NWDAF* server and will be explained later on. The *Python* script uses the first two arguments to read the *csv* files. The *Pandas Python* package was incorporated as part of the *UPF* container image to allow *Python* to read the *csv* files. Figure 4.9 shows how the arguments are processed in the program.

The columns of the corresponding *csv* files are then parsed. The first column corresponds to the packet timestamps, while the second column is either the hash of the packet data or the TCP sequence number of the packets. The values on the second column will be used as a packet identifier. The objective is to identify correspondences between the fields in the packets belonging to the packets of the *n6-3* interface and the *n3-2* interface. In other words, identify how many packets arrived at the *n3-2* interface from the *n6-3* interface by counting how many correspondences are found. For the HTTP/3 protocol, which corresponds to UDP packets, the hash value of the packet data is used. For the HTTP/2 protocol, which corresponds to TCP/TLS packets, the TCP sequence number is used instead. Not all TCP/TLS packets contain data, so it was decided to go for this alternative. This alternative, however, provided a challenge due to a TCP configuration on the ext-dn

```

# Access command-line arguments
arguments = sys.argv

# Subsequent elements are the command-line arguments
stream_A = arguments[1]
print(stream_A)

csv_file_path = "/UPF_NWDAF/n6-3_" + stream_A + ".csv"
n6_3_Stream_A = pd.read_csv(csv_file_path)

csv_file_path = "/UPF_NWDAF/n3-2_" + stream_A + ".csv"
n3_2_Stream_A = pd.read_csv(csv_file_path)

stream_B = arguments[2]
print(stream_B)
csv_file_path = "/UPF_NWDAF/n6-3_" + stream_B + ".csv"
n6_3_Stream_B = pd.read_csv(csv_file_path)

csv_file_path = "/UPF_NWDAF/n3-2_" + stream_B + ".csv"
n3_2_Stream_B = pd.read_csv(csv_file_path)

proto_A = arguments[3]
proto_B = arguments[4]

cur_time = int(arguments[5])

```

Figure 4.9: processing of the arguments of the *calculate_metrics.py* Python script

container. More details will be entailed about this problem later on. An extra step was performed for the HTTP/2 protocol. Before finding correspondences, all the repeated TCP sequence numbers were removed, leaving only the last occurrence. This is because repeated TCP sequence numbers are associated with TCP control packets like TCP retransmission or TCP packet-out-of-order. Those packets should not be accounted for when calculating the number of packets that arrived at the *n3-2* interface. Doing so would result in an inaccurate calculation of the Packet loss ratio metric. Also, instead of keeping the last occurrence of a TCP sequence number, the first occurrence could have been kept disregarding all the others. That option led to higher delay values, even in cases where it should not happen. For that reason, it was decided to opt for keeping the last occurrence only. Figure 4.10 shows the code needed to carry out these tasks. The *timepackets_n3* and the *timepackets_n6* variables contain the timestamps of the packets that were recognized in both interfaces, i.e., the packets that were not considered lost.

The packet delay is calculated as the difference between *timepackets_n6* and *timepackets_n3*.

```

timestamps_n6 = n6[n6.columns[0]]
timestamps_n3 = n3[n3.columns[0]]

if (proto == "HTTP/3") :
    mask = n6[n6.columns[1]].isin(n3[n3.columns[1]])
    matched_packets = mask.sum()
    timepackets_n6 = timestamps_n6[mask].tolist()
    timepackets_n3 = timestamps_n3[n3[n3.columns[1]].isin(n6[n6.columns[1]])].tolist()
elif (proto == "HTTP/2"):
    # Identify and remove duplicates in n3 and n6
    n3_seq = n3.drop_duplicates(subset=n3.columns[1],keep='last')
    n6_seq = n6.drop_duplicates(subset=n6.columns[1],keep='last')

    sorted_n3_seq = n3_seq.sort_values(by=n3_seq.columns[1])
    sorted_n6_seq = n6_seq.sort_values(by=n6_seq.columns[1])
    mask = sorted_n6_seq[sorted_n6_seq.iloc[:,1].isin(sorted_n3_seq.iloc[:,1])]
    matched_packets = len(mask)
    timepackets_n3 = sorted_n3_seq[sorted_n3_seq.iloc[:,1].isin(mask.iloc[:,1])].iloc[:,0].tolist()
    timepackets_n6 = sorted_n6_seq[sorted_n6_seq.iloc[:,1].isin(mask.iloc[:,1])].iloc[:,0].tolist()

```

Figure 4.10: parsing the fields in the *csv* files, calculating the packets that were received and their associated timestamps

The resulting value is multiplied by a factor of 1000 to convert seconds (*s*) into milliseconds (*ms*). In *Python* this is accomplished by the following line of code:

```

delay = [(t_n3 - t_n6) * 1000 for t_n3, t_n6 in zip(timepackets_n3,
                                                    timepackets_n6)]

```

The *matched_hashes* value obtained earlier is used to determine packet loss. Packet loss is calculated by determining the proportion of unmatched packets in the *n6-3* interface and then converted into a percentage value by multiplying by a factor of 100. The *average delay* is obtained by summing up the individual delay values and dividing it by the total number of packets. The *standard deviation* of the delay is calculated with the *stdev* function of the *statistics Python* package [36]. These three operations are represented in Figure 4.11.

```

loss = (1 - (matched_packets / len(timestamps_n6))) * 100
avg_delay = sum(delay) / len(delay)
std_dev = statistics.stdev(delay)

```

Figure 4.11: calculation of the packet loss, average delay and standard deviation of the delay in *Python*

To calculate the throughput, the bitrate of the capture file will be used as an approximation. The *capinfos* [37] tool will be used to obtain information about the *pcap* files. First, the duration of the capture is stored. Then, the packet rate is calculated taking into account the number of packets that were not lost. This is done to compensate for any packet losses that may occur. If the total number of packets of the *pcap* file were to be used, the impact of packet loss would not be reflected in the throughput. The next step is to extract the average size of the total number of packets of the capture file. This is performed with the *capinfos* tool, once again. The average packet size is converted into bits, then multiplied by the packet ratio. Finally, to get the estimate of the throughput, in *Mb/s*, the result is divided by a factor of 1000000. Figure 4.12 depicts the code used to perform the calculation of the throughput estimation.

```

command = "capinfos -u /UPF_NWDAF/n3-2_{}.format(stream) + ".pcap" + " | awk '{print $3}' | tail -n 1"
result = subprocess.run(command, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, universal_newlines=True)

if (result.returncode == 0) :
    output = result.stdout.strip().split()
    if len(output) == 1:
        capture_duration = float(output[0])

packet_rate = matched_packets / capture_duration

command = "capinfos -z /UPF_NWDAF/n3-2_{}.format(stream) + ".pcap" + " | awk '{print $4}' "
result = subprocess.run(command, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, universal_newlines=True)

if (result.returncode == 0) :
    output = result.stdout.strip().split()
    if len(output) == 1:
        packet_size = float(output[0]) * 8

bitrate = (packet_rate * packet_size) / 1000000

```

Figure 4.12: Calculation of the throughput in *Python*

To give a visual representation of the calculated metrics, a histogram of the packet delay and a graph of throughput over time were created. The *Python* package *matplotlib*, part of the collection of packages installed as part of the *UPF* Docker image, was used for this effect. In Figure 4.13 illustrates the use of the *matplotlib* package to construct a histogram. The histogram is then saved as a *png* image. The next to last line of code calls a function to generate a throughput over time graph. The procedure is similar to the one shown in the figure, except that an additional file is needed to save the values between each call of the *calculate_metrics.py* file. Note that the histogram figure being created here, shows the packet delay for a specific time period, e.g., one second to five seconds. The complete histogram that takes into account the entire time period, e.g., 40 seconds, is defined in the *NWDAF* server and will be explained later.

```

num_bins = 22
bins = np.linspace(0, avg_delay + 5*std_dev, num_bins+1)
delay_hist = plt.figure(figsize=(20, 12))
plt.hist(delay, bins=bins, alpha=0.5, color='blue')
plt.xticks(bins, ['{:0.2f}'.format(x) for x in bins], fontsize=16)
plt.yticks(fontsize=16)
plt.title("Delay distribution for {} ({} -> {}) s".format(stream, cur_time-4, cur_time), fontsize=20)
plt.xlabel('Delay (ms)', fontsize=16)
plt.ylabel('Number of packets', fontsize=16)

#### Save histogram ####
file_path_delay = "/UPF_NWDAF/delay_histogram_{}.png".format(stream)
delay_hist.savefig(file_path_delay)

#### Bitrate Graphs ####

bitrate_graphs(bitrate, stream)

plt.clf() # Clear the current figures

```

Figure 4.13: Creation of the packet delay histogram in *Python*

Also, the metric values calculated are written into two different files. One containing the metrics for the first stream and the other for the second. If no packets belonging to that stream were captured, then the corresponding file will be saved with the message: "No stream activity". 100% loss! will be written if packets were capture in the *n6-3* interface, but no packets were capture in the *n3-2* interface. Like with the histogram, these files contain the metrics for a specific

period.

4.3.2.5 Traffic shaping script

To simulate the influence that an *ISP* would have over a flow, the *ubuntu tc* tool [38] was used to manipulate traffic. The shell script named *traffic_shaping.sh* creates a queue disc for an interface:

```
tc qdisc add dev $interface root handle 1: prio
```

It essentially creates another queue in that interface, which will handle traffic differently. The *interface* variable is defined in the script as *n3-2*, as the objective is to act on incoming traffic to the *UE*. To define the profile of the traffic that goes into the newly created queue, a filter needs to be configured:

```
tc filter add dev $interface parent 1:0 protocol all prio 1 u32
match u32 $source_ip_address_hex $mask at $source_ip_byte flowid 2:1
```

This filter looks for the matching of certain fields in a packet. In this case, it looks for a source IP address of choice in specific byte/bytes of the packet.

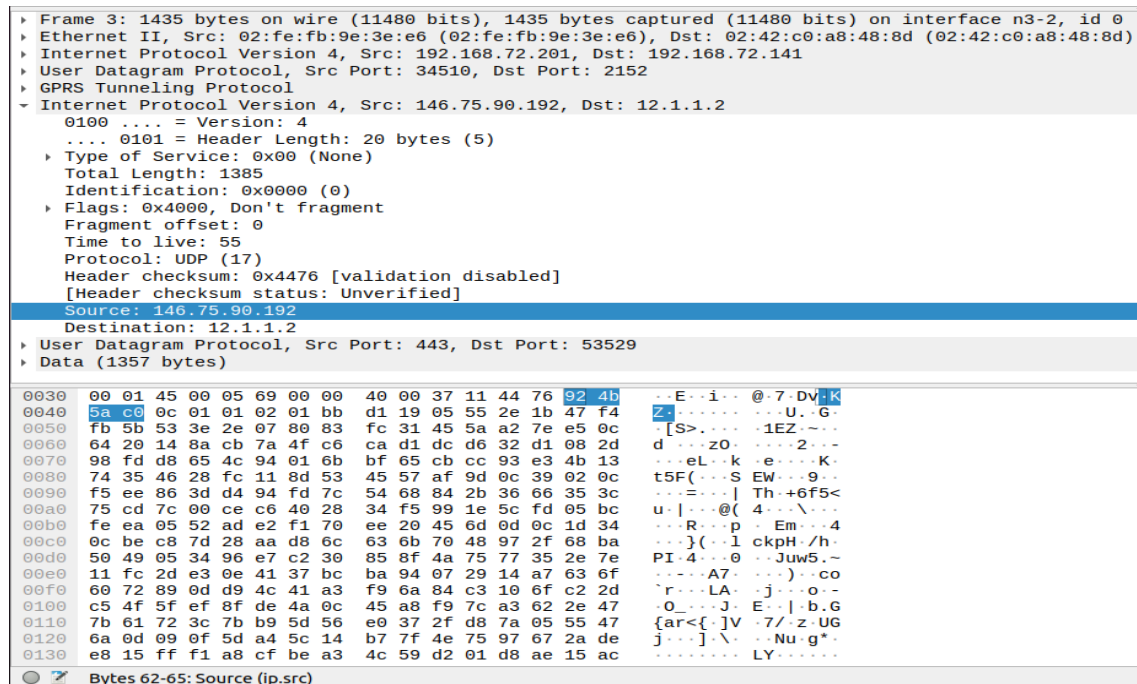


Figure 4.14: Byte range associated with the source IP address in wireshark

The number of the desired byte and the hexadecimal value of the source IP address was obtained by analyzing *wireshark* captures on the *n3-2* interface. As observed in Figure 4.14, the source IP address field is located in the 62 – 65 byte range. Note that this is the inner source IP address of the *GTP-U* tunnel. It appears that the byte of interest would be 62. However, after

executing the script, it follows that the correct byte number is not 62. After further research [39], it was found out that the bytes displayed in *wireshark* include the *Ethernet* frame headers, which usually accounts for 14 bytes. As a result, the byte at which the source IP address starts is 48. After performing some experiments this value proved to be correct.

After creating a different queue in the interface and filtering the traffic flow, traffic shaping can be applied using the *tc* tool. To apply packet delay and/or packet loss to the filtered traffic flow the following command was used:

```
tc qdisc add dev $interface parent 1:1 handle 2:  netem delay $4ms
    loss $5%
```

The delay values and the loss values are provided by the user when executing the script. In the command shown, both delay and loss are being applied, however they can be imposed separately. Furthermore, it is also possible to adjust the values of delay and/or loss settings without deleting the created queue. The command to remove a queue is very similar to the one used to create it:

```
tc qdisc d dev $interface root handle 1:  prio
```

4.3.3 EXT-DN

As stated earlier, an issue arose when analyzing TCP packets. Under normal circumstances, i.e., without the interference of the *ISP*, the same number of packets from a flow should be present in the *n6-3* interface and in the *n3-2* interface. This was not the case and a lot of times the *n3-2* interface actually had more packets than the *n6-3* interface. This behavior was unexpected, as the number of packets in the *n3-2* interface should be the same or fewer than in *n6-3*. Upon further inspection, it was found that the packets in *n6-3* interface had a size of more than the standard MTU value (1500 bytes) [40]. These packets had a size of around 5000 bytes. When entering the GTP-U tunnel, fragmentation was performed, and a single packet was converted into smaller packets with sizes within the MTU value. This resulted in the presence of more packets on the *n3-2* interface. After delving deeper into the problem, it was found that the culprit for this issue was *TCP segmentation offload (TSO)* [41]. Essentially this option relegates the segmentation of network packets from the TCP protocol to the Network Interface Card (NIC) [42]. As it appears, the NIC of the ext-dn container was not performing the segmentation operation, as turning this option off effectively resolved the issue. The command to disable TSO is the following:

```
ethtool -K eth0 tso off
```

This command was added to the options of the oai-ext-dn service in the *docker-compose-basic-vpp-nrf.yaml* configuration file.

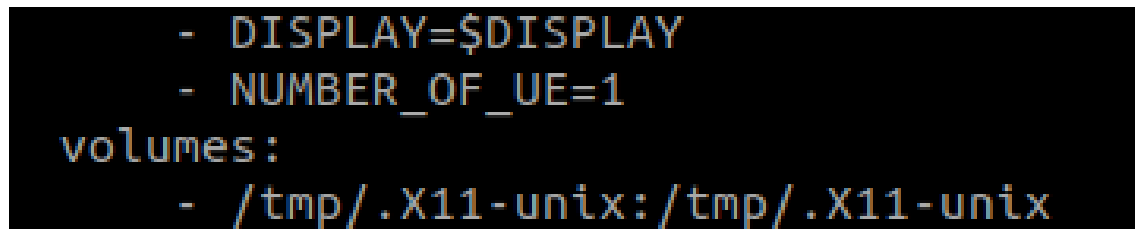
4.3.4 UERANSIM

4.3.4.1 Docker image

For the *UERANSIM*, some modifications to the *Docker* image were made, with the intent of providing access to the *CAP* servers. Two video streaming websites will be accessed through a browser. So first, the Firefox browser was added to the list of applications of the *docker* image. Along with it, extra video codec libraries were installed to provide additional support for the browser. Finally, to allow the launch of the Firefox browser, *xauth* was necessary. This package is responsible for the execution of graphical applications.

4.3.4.2 UERANSIM Configurations

UERANSIM has its own *docker-compose* file. Like the *docker-compose* file for the *5G Core Network*, it allows further configuration of containers. To complete the support for graphical interface applications, the *DISPLAY* environment variable of the container was defined with the same value as the *DISPLAY* environment variable of the host PC. Additionally, a *Docker* volume was created to map the *X11* directory of the host PC with a directory with the same name inside the *Docker* container. These changes can be observed in Figure 4.15 along with other options. To note that among them is the *NUMBER_OF_UE* environment variable, which allows for a simple selection of the number of User Equipments (UEs) desired. For this work, only one *UE* will suffice, since two video streams can be played simultaneously in the *UE* and then filtered.



```
- DISPLAY=$DISPLAY
- NUMBER_OF_UE=1
volumes:
- /tmp/.X11-unix:/tmp/.X11-unix
```

Figure 4.15: Changes made to the *UERANSIM* *docker-compose* file

To launch the *UERANSIM* container, the *docker-compose-ueransim-vpp.yaml* file is executed using the following command:

```
sudo docker-compose -f docker-ueransim-vpp.yaml up -d
```

Furthermore, additional commands need to be executed to properly configure the network connection of *UERANSIM*. In particular, the default route needs to be reconfigured and the host needs to give permission to execute graphical applications inside the container.

```
sudo docker exec ueransim ip r d default
sudo docker exec ueransim ip r a default via 12.1.1.2
sudo xhost +
```

The first two commands displayed above configure the new route and the last command gives permission for the container to launch the graphical interface applications. The default route of the *UERANSIM* container is the *cn5g-access* bridge. This means that by default, the packets going for the Internet would be forwarded by the host PC, without ever going through the *5G Core Network*. To change this, the default route was defined as the IP address of the GTP-U tunnel that connects the *UE* with the Core Network through the *UPF*. Note that since *UERANSIM* does not separate the gNB from the *UE*, the same IP address is used for both, in this case, 12.1.1.2.

4.3.5 NWDAF

4.3.5.1 Docker image

Apart from the previous containers used up to this point, the *NWDAF* was not downloaded from any software repository. Instead, it was developed from scratch. The *NWDAF* container uses a simple Docker image, created with the solo purpose of supporting the basic functionalities of this work's *NWDAF*. In addition to standard software packages, such as *iproute2* and *curl*, it is important to highlight the presence of other crucial packages for the *NWDAF* functionality. The *NWDAF* will host a web server, and for that purpose, two *Python* packages were installed, named *flask* [43] and *quart* [44]. *Flask* is a *Python* web framework used to develop simple web applications. However, the *3GPP* specification states that the *5G Core Network* services use HTTP/2, a feature not available in *flask*. Therefore, the *Python* web framework named *quart* was used in its place. *Quart* was built on top of the *flask* framework, but it supports HTTP/2 development. The web server is responsible for plotting the histogram showing for the complete averaging window. This window will be defined by the user in the web page. The *matplotlib* library is thus required to be a part of the *NWDAF* container software. The *NWDAF* server needs to be able to execute the *captures.sh* from within the *UPF* container. To accomplish it, *Docker* was installed inside the *NWDAF* container. This way, the *NWDAF* can use the *Docker* environment to execute a command inside another container, in this case the *UPF* container. Also, for convenience, it was defined that the default directory of the *NWDAF* container upon launch is the shared directory */UPF_NWDAF*.

4.3.5.2 NWDAF Configurations

For the *NWDAF*, instead of using a *docker-compose.yaml* [45] file, the *docker run* command was used. Figure 4.16 depicts the *docker run* command used to configure and launch the *NWDAF* container. Breaking down the command, it first defines the *-d* option, which stands for *detached mode*. This option tells *Docker* to run the container as a background process. The *-volumes-from* option will mount all the volumes from the *UPF* container into the *NWDAF* container. This entails that the *NWDAF* container will have access to the volumes defined for the *UPF* container. For this work, this means that the shared directory */UPF_NWDAF* will be available in the *NWDAF* container. Then, a volume specific to *NWDAF* is created. This volume allows the container to access the *docker socket* file of the host PC. This allows communication between the *Docker* engine installed in the *NWDAF* and the *Docker* daemon [46] of the host PC (a background process

responsible for managing *Docker* containers and resources). In other words, this will allow the *NWDAF* to interact with all the containers launched by the host PC. Following this, the network and the IP address of the *NWDAF* container are defined by the `--net` and `--ip` options. The next option `--cap-add` adds *Linux* capabilities to the container. *NET_ADMIN* allows the container to modify the network interfaces. Finally, the `--name` option assigns a name to the container (*oai-NWDAF*), which is created based on the *Docker* image *nwdaf*.

```
sudo docker run -d --volumes-from vpp-upf --volume \
/var/run/docker.sock:/var/run/docker.sock --net \
demo-oai-public-net --ip 192.168.70.140 --cap-add=NET_ADMIN \
--name oai-NWDAF nwdaf
```

Figure 4.16: Command used to configure and run the *NWDAF* container

4.3.6 NWDAF Web Server

As stated previously, the *Python* web framework *quart* was used to implement the web server. Figure 4.17 showcases the main loop configuration that defines the server. Note that this code will only be executed if the current module is being run as the main program.

```
if __name__ == '__main__':

    config = Config()
    config.bind = [IP_address]
    config.certfile = certfile_path # Path to your SSL certificate
    config.keyfile = keyfile_path # Path to your SSL private key
    config.use_http2 = True # Enable HTTP/2 support
    loop = asyncio.get_event_loop()
    loop.run_until_complete(serve(app, config))
```

Figure 4.17: Definition of the configurations for the *NWDAF* server

The server is bounded to the IP address of the *NWDAF* container (192.168.70.140) and to port number 443, which is the default port of HTTP Secure (HTTPS). It also uses the HTTP/2 protocol. As part of the configuration for HTTPS, a self-signed certificate was created using *OpenSSL* [47]. *OpenSSL* is a software that includes tools and functions to generate SSL/TLS certificates. The self-signed certificate can be installed in the browser to avoid security warning messages. The IP address of the server and the path for the certificate related files can be seen in Figure 4.18. Also present in the figure is the creation of the *quart* object, named *app*. This object comprises the configurations of the *quart* application defined in Figure 4.17, as well as the static files folder and the template folder. The difference between these is that static folder specifies the path for static files, like Cascading Style Sheets (CSS) [48] files or JavaScript files, while template folder provides the path for dynamic files. As an example, the majority of files served in the *NWDAF*

server are static, the only exception being an *HTML* [49] page to which two extra variables were provided. These variables were then processed by JavaScript code inside the *HTML* file.

```
IP_address = "192.168.70.140:443"
certfile_path = 'openssl/certificate.crt'
keyfile_path = 'openssl/private.key'

app = Quart(__name__,static_folder='/', template_folder = '/UPF_NWDAF/')
```

Figure 4.18: Setting of the IP address of the *NWDAF* server, as well as the definition of the self-signed certificate related files and creation of the *quart* object

In order to launch the web server the following command is used:

```
docker exec oai-NWDAF python3 NWDAF_server.py
```

4.3.6.1 Route definitions

The main structure of the quart web server is composed of *@app.route* definitions. These route definitions define the Uniform Resource Locator (URL) patterns and the corresponding functions that handle the requests for those URLs. When a client makes a request to a specific URL, the quart application matches the URL to the appropriate route definition and executes the corresponding function. Figure 4.19 exemplifies the route that handles a client request for the root URL of the web server. When a GET request is received the *home* function is executed. The *subprocess_instance* global variable verifies if a process is running on the server. If no process is running, the block of code after *else* executes. This is the expected behavior every time the server starts. As a response, the index file *index.html* is presented in the client's browser. Also part of the response are headers responsible for disabling browser caching. The rationale behind no-caching the index file is to allow the refresh of the page when going back on the browser. If a process is running on the server, that means the *NWDAF* was monitoring flows, and the index page was requested, either by going back on the browser or by typing the root URL. If this is the case, the block of code under the *if* statement is executed. The process that was running is stopped and a redirection page is defined. This page will redirect the user to the index page (*index.html*) after displaying for two seconds, a simple page with a message. This message informs the user that they have unsubscribed from a subscription with a specific Correlation ID. This ID is defined in the *NnwdaF_AnalyticsSubscription_Subscribe* operation and saved in the server on the Subscription class. More details about it will be given later. Essentially this is equivalent to the *NnwdaF_AnalyticsSubscription_Unsubscribe* operation. Note that the subscription is deleted by setting the *sub* object to none.

```

@app.route('/')
async def home():
    global subprocess_instance
    if subprocess_instance is not None:
        subprocess_instance.terminate()
        subprocess_instance = None
    global sub
    Subscription_Correlation_ID = sub.Subscription_Correlation_ID
    message = "Successfully unsubscribed from Subscription = %s" % (Subscription_Correlation_ID)
    sub = None
    return """
    <html>
        <body>
            <h2>{message}</h2>
            <script>
                setTimeout(function() {{
                    window.location.href = "/";
                }}, 2000); // Redirect after 2 seconds
            </script>
        </body>
    </html>
    """.format(message=message)
    else:
        response = await send_file('index.html')
        response.headers['Cache-Control'] = 'no-cache, no-store, must-revalidate'
        response.headers['Pragma'] = 'no-cache'
        response.headers['Expires'] = '0'
        return response

```

Figure 4.19: Root route of the *NWDAF* web server

4.3.6.2 Subscribe service

The web server implements the *Nnwdaf_AnalyticsSubscription_Subscribe* operation at the *handler* URL route. The *handler* URL is defined in the *index.js* file, the *JavaScript* file associated with the *index.html* homepage. The URL will be composed of multiple fields, the first of them being *handler*. How this URL is defined will be explained in the web page section. When the server receives a request for the */handler* URL the handler function is executed. This function extracts the other fields that compose the complete URL. It first obtains the *action* parameter. Then it gets the names of the two video stream flows of interest. Or in the case of a custom stream, it extracts the filter attached to it. The two *if* statements related to both streams, verify if the value has more than one space. If that is the case, then it is a filter instead of just a name. Also, right before the function evaluates the *action* value, the flag indicating the completion of the subprocess is set to False. The code to execute the described steps can be seen in Figure 4.20.

```

@app.route('/handler', methods=['GET'])
async def handler():
    # Get the value of the 'action' query parameter from the URL
    action = request.args.get('action', '')
    global stream_A
    global stream_B
    global subprocess_completed
    global subprocess_instance
    global Custom_A
    global Custom_B

    stream_A = unquote(request.args.get('stream1', ''))
    stream_B = unquote(request.args.get('stream2', ''))

    if len(stream_A.split()) > 1 :
        Custom_A = True
    if len(stream_B.split()) > 1 :
        Custom_B = True

    subprocess_completed = False

```

Figure 4.20: First snippet of the */handler* route of the *NWDAF* web server

If the *action* provided in the URL is *Nnwdaf_AnalyticsSubscription_Subscribe*, then the function stops any process that might be running and initiates the Subscription Class object (*sub*). The URL provides the necessary field values that define a subscription. The definition of this class can be observed in Figure 4.21.

```
class Subscription:
    def __init__(self, Notification_Correlation_ID, Stream_A, Stream_B, Analytics_ID,
                  Subscription_Correlation_ID, Target_Period):
        self.Notification_Correlation_ID = Notification_Correlation_ID
        self.Stream_A = Stream_A
        self.Stream_B = Stream_B
        self.Analytics_ID = Analytics_ID
        self.Subscription_Correlation_ID = Subscription_Correlation_ID
        self.Target_Period = Target_Period
```

Figure 4.21: Definition of the Subscription class in *Python*

After the creation of the *sub* Subscription object, the server defines a command to call the *captures.sh* shell script from within the *UPF* container. *Docker exec* is used to execute the script inside the *UPF* container. Four command line arguments are passed to the script: 1) *stream_A*, the name of the first flow to monitor; 2) *stream_B*, the name of the second flow to monitor; 3) *subprocess_completed*, this variable essentially tells the script if it needs to run just once or run for the time provided in the fourth argument; 4) This corresponds to the time that the script will run for. Here, 0 means it will run just once. Then, the command is executed with *subprocess.run*. This function blocks the code progress until the completion of the command. After its completion, the flag *subprocess_completed* is set to *True*. Note that a blocking function is used, so that the server has any information to display about the video stream flows. Therefore, after clicking on the submit button, the process will run, and once it is finished (5 seconds), the new page will load with the pertinent content. The code that implements the described procedures is depicted in Figure 4.22.

```
# Check the value of 'action' and return the appropriate response
if action == 'Nnwdaf_AnalyticsSubscription_Subscribe':
    if subprocess_instance is not None:
        subprocess_instance.terminate()
        subprocess_instance = None

    ## start subscription object ##

    Notification_Correlation_ID = request.args.get('Notification_Correlation_ID', '')
    Analytics_ID = request.args.get('Analytics_ID', '')
    Subscription_Correlation_ID = str(secrets.token_hex(16))
    Target_Period = request.args.get('Target_Period', '')
    global sub
    sub = Subscription(Notification_Correlation_ID, stream_A, stream_B, Analytics_ID,
                       Subscription_Correlation_ID, Target_Period)

    command = shlex.split('docker exec vpp-upf /bin/bash -c \' /UPF_NWDAF/captures.sh "{}" "{}" "{}" "{}"; '
                           '$SHELL &\''.format(stream_A, stream_B, subprocess_completed, 0))
    first_instance = subprocess.run(command)
    subprocess_completed = True
```

Figure 4.22: Second snippet of the */handler* route of the *NWDAF* web server

If the user specified a *Target_Period* or *averaging window* of more than five seconds, the block of code presented in Figure 4.23 is executed. The command is defined as similar to the one in Figure 4.22, but the last argument is the *Target_Period* field. This will tell the *captures.sh*

script for how long it will run the captures. It is important to notice that this is capture time and not actually real-time. In other words, the *Target_Period* defines the number of times *tshark* will capture packets. For example, if *Target_Period* is 15, the *tshark* commands responsible for the captures will be executed 3 times. Another key difference is that instead of *subprocess.run*, *subprocess.Popen* will be used. This function is not blocking, which allows the server to accept requests while running the process in the background.

```
if (int(Target_Period) > 5) :
    command = shlex.split('docker exec vpp-upf /bin/bash -c \'/UPF_NWDAF/captures.sh "{}" "{}" "{}" "{}"; '
    '$SHELL &\''.format(stream_A, stream_B, subprocess_completed, Target_Period))
    subprocess_instance = subprocess.Popen(command)
```

Figure 4.23: Third snippet of the */handler* route of the NWDAF web server

In both cases, the response from the server will be the *HTML* page, *analytics.html*. This page will be served using the template folder as the root folder. As talked earlier, this is the only dynamic page served on the server. The *Subscription_Correlation_ID* and the *Target_Period* variables will be given as arguments for *JavaScript* code in the *HTML* to process. Additionally, browser caching is disabled, with the use of the *headers* options. Figure 4.24 shows the code to execute the explained procedures.

```
html = await render_template('analytics.html', Subscription_Correlation_ID =
    Subscription_Correlation_ID, Target_Period = Target_Period)

response = Response(html)

response.headers['Cache-Control'] = 'no-cache, no-store, must-revalidate'
response.headers['Pragma'] = 'no-cache'
response.headers['Expires'] = '0'
return response
```

Figure 4.24: Final snippet of the */handler* route of the NWDAF web server

4.3.6.3 Unsubscribe service

The unsubscribe operation is very similar to the block of code executed for the *if* statement of Figure 4.19. If the *action* specified on the requested URL is *Nnwdafe_AnalyticsSubscription_Unsubscribe*, the function stops any subprocess running. The *subprocess_instance* variable refers to the process started by *subprocess.Popen* in the *Nnwdafe_AnalyticsSubscription_Subscribe* operation. Note that in Figure 4.23, the process created by *subprocess.Popen* command is associated with the *subprocess_instance* variable. After that, the *Subscription* object *sub* is removed and a redirecting message is set up, just like the one from the root route (4.19). The *Nnwdafe_AnalyticsSubscription_Unsubscribe* is illustrated in Figure 4.25.

```

elif action == 'Nnwdaf_AnalyticsSubscription_Unsubscribe':
    if subprocess_instance is not None:
        subprocess_instance.terminate()
        subprocess_instance = None
    sub = None
    Subscription_Correlation_ID = request.args.get('Subscription_Correlation_ID', '')
    message = "Successfully unsubscribed from Subscription = %s" % (Subscription_Correlation_ID)
    return """
    <html>
        <body>
            <h2>{message}</h2>
            <script>
                setTimeout(function() {{
                    window.location.href = "/";
                }}, 2000); // Redirect after 2 seconds
            </script>
        </body>
    </html>
    """.format(message=message)
else:
    return "Error while trying to subscribe"

```

Figure 4.25: '/unsubscribe' route of the NWDAF web server

4.3.6.4 Metrics routes

To respond and display the metrics for the video streams, two routes were defined. Route '/metrics_A' for the first stream and route '/metrics_B' for the second. Figure 4.26 showcases the code that handles the requests for metrics of the first stream. It starts by defining the file to be served based on the type of stream. Then, if the file path is correct and the the command executed by *subprocess.run* in the subscribe operation is completed, it serves the file. Additionally, the web browser caching is disabled to prevent the wrong display of information on the web page. The route that handles requests for the second stream, '/metrics_B' applies the same logic, the only differences being *stream_B* and *Custom_B* are used instead of *stream_A* and *Custom_A*.

```

@app.route('/metrics_A')
async def metrics_A():
    global subprocess_completed
    global stream_A
    global Custom_A

    if Custom_A == True:
        file_path = '/UPF_NWDAF/metrics_custom_A.txt'
    else:
        file_path = "/UPF_NWDAF/metrics_" + stream_A + ".txt"

    if subprocess_completed and os.path.exists(file_path):
        response = await send_file(file_path)
        response.headers['Cache-Control'] = 'no-cache, no-store, must-revalidate'
        response.headers['Pragma'] = 'no-cache'
        response.headers['Expires'] = '0'
        return response
    else:
        return "Please wait until the subprocess completes."

```

Figure 4.26: '/metrics_A' route of the NWDAF web server

4.3.6.5 Histograms/Graphs routes

When it comes to the histogram and graph images, the server handles the requests in a similar fashion to the previously described metric routes, but with a few differences. First difference is in the route name. At the end of the route name definition, `<stream_name>` was added. This is what is known as a dynamic route [50]. A dynamic route responds to requests with URLs that can have an arbitrary value at the end. For example, this route will respond to a request with an URL, such as `/delay_histogram_Stream_A/default`. A static route would only be able to handle requests for `/delay_histogram_Stream_A`. Second difference is on the function associated with the route. To handle the dynamic behavior, the function accepts the value at the end of the URL as an argument. If there are no errors, the `png` file is served, however, in this case, web browser caching is not disabled. Web browser caching in conjunction with dynamic routing allow the web page to update the `png` images being displayed. When a file is cached by the browser, it associates the cached version with a timestamp from the server's Last-Modified header or other caching-related headers. This timestamp helps the browser determine whether to use the cached version or request a fresh copy from the server based on its validity. The last difference is that, in the case where the file does not exist, a placeholder image is served instead. The placeholder image is just an empty picture designed with the same size as the histogram or graph would have. Figure 4.27 depicts the implementation of the route for the `/delay_histogram_Stream_A/<stream_name>` URL. The histogram for Stream_B and the throughput graphs for both streams. follow the same logic and the code is almost identical.

```
@app.route('/delay_histogram_Stream_A/<stream_name>')
async def get_delay_Stream_A_histogram(stream_name):
    global stream_A
    global Custom_A

    if Custom_A == True:
        file_path = "/UPF_NWDAF/delay_histogram_custom_A.png"
    else:
        file_path = "/UPF_NWDAF/delay_histogram_" + stream_A + ".png"

    global subprocess_completed
    if subprocess_completed and os.path.exists(file_path):
        return await send_file(file_path, mimetype='image/png')
    elif subprocess_completed and not os.path.exists(file_path):
        return await send_file('/UPF_NWDAF/empty_placeholder.png', mimetype='image/png')
    else:
        return "Please wait until the subprocess completes."
```

Figure 4.27: `/delay_histogram_Stream_A/<stream_name>` dynamic route of the *NWDAF* web server

4.3.7 NWDAF Web Page

4.3.7.1 Home page

The home page of the *NWDAF* is defined in the `index.html` file. The user can access the page through a web browser by typing in the IP address of the *NWDAF* (192.168.70.140). The first time entering the IP address it is likely that the browser will throw a security warning message.

To disable it, the user can install the self-signed certificate in the browser options. As long as the browser cache is not cleared, the warning message will not appear again. Figure 4.28 illustrates the home page of the *NWDAF* service. The user of the page can select two flows. The drop-down menu allows the selection of four pre-defined streams or the option to customize a stream. “*RTP*”, *YOUTUBE*, *SIC*, and *TVI* are the four default video stream options. To define a custom stream, the user shall enter a source IP address, select the protocol (*HTTP/2 TCP* or *HTTP/3 UDP*), and enter a destination IP address. The text boxes to enter both IP addresses and the drop-down menu for protocol selection, only become active after selecting the custom stream option. Originally the *NWDAF* was designed with video streaming in mind, so *HTTP/2* and *HTTP/3* were chosen as the protocols. However, *HTTP/2* traffic consists of *TCP/TLS* packets and *HTTP/3* consists of *UDP* packets. Therefore, the final designation of the protocols provides a more generic name convention and it also makes the *NWDAF* a more complete tool. By default, the destination IP address of a custom flow is 12.1.1.2. This is the IP address of the *UE* defined by *UERANSIM*. After the user chooses two video stream flows with all the fields properly formatted, the averaging window has to be chosen. As stated previously, the developed *NWDAF* works in intervals of five seconds. So, the averaging window has to be a multiple of 5. Then, the user clicks the submit button and the *NWDAF* server interprets the URL and starts processing the information.

← → ↻ ☆ https://192.168.70.140

NWDAF video stream flow monitoring

Flow 1

Select a stream ▼

Custom source IP address: Protocol ▼

Custom destination IP address:

Flow 2

Select a stream ▼

Custom source IP address : Protocol ▼

Custom destination IP address:

Averaging window: s

Figure 4.28: Homepage of the *NWDAF* web page

The *index.js* file handles the user selections and sends the appropriate request for the server. The submit button is associated with the *Nnwdaf_AnalyticsSubscription_Subscribe* service. The *JavaScript* function *handleSubscribeClick* is responsible for extracting the field values from the *HTML* file and constructing the URL to send to the server. The following figures provide snippets of *JavaScript* code of that function. Figure 4.29 depicts the beginning of the *handleSubscribeClick* function. Using *document.getElementById* *JavaScript* extracts from the document the stream names provided by the user associated with the pre-defined streams. The *Target_Period* is also obtained through the same method. Note the value provided in the *Averaging* box will be the *Target_Period* value. It also creates a random *Notification_Correlation_ID* and declares the variables to handle the fields of a custom-defined stream.

```
function handleSubscribeClick() {
    const stream1 = document.getElementById("stream1").value;
    const stream2 = document.getElementById("stream2").value;
    const Notification_Correlation_ID=Math.random().toString(36).substr(2);
    const Analytics_ID = "Network Performance";
    const Target_Period = document.getElementById("target_period").value;
    let CustomInputIP_A_src;
    let CustomInputIP_B_src;
    let CustomInputIP_A_dst;
    let CustomInputIP_B_dst;
    let Protocol_A;
    let Protocol_B;
```

Figure 4.29: First snippet of the *handleSubscribeClick* function

Figure 4.30 showcases how the code handles a case where the first stream is custom. It first extracts the custom source and destination IP addresses and validates them, using the *checkValidity* *JavaScript* method. Then, it extracts the protocol of the custom stream and verifies the validity of all the fields using the *CheckCustomOption* function. This function will go through them and return an alert message if any of them are incorrect. It will also return -1 , signaling an error. As a result, the execution of the code will be stopped.

```
if(stream1 === "Custom" && stream2 !== "Custom"){
    CustomInputIP_A_src = document.getElementById("Custom_stream_A").querySelector('input');
    CustomInputIP_A_dst = document.getElementById("Custom_stream_A_dst").querySelector('input');
    let valid_src = CustomInputIP_A_src.checkValidity();
    let valid_dst = CustomInputIP_A_dst.checkValidity();
    Protocol_A = document.getElementById("Protocol_A").value;
    let CheckCustom1 = CheckCustomOption(stream1, CustomInputIP_A_src, CustomInputIP_A_dst,
    valid_src, valid_dst ,Protocol_A, 1);
    if (CheckCustom1 === -1){
        Protocol_A.disabled = false;
        return;
    }
}
```

Figure 4.30: Second snippet of the *handleSubscribeClick* function

To construct the URL, the *buildUrl* function is used. This function takes as arguments the stream-related fields, and based on the type of stream (pre-defined or custom) builds the URL,

concatenating to the base URL the value of the fields. The base URL is defined in Figure 4.31, as the *url* variable.

```
function buildUrl(stream1, stream2, CustomInputIP_A_src, CustomInputIP_A_dst, protocolA, CustomInputIP_B_src,
CustomInputIP_B_dst, protocolB, notificationCorrelationID, analyticsID, targetPeriod) {
  let url = '/handler?action=Nnwdaf_AnalyticsSubscription_Subscribe';

  // Stream 1 and 2 are not custom
  if (stream1 !== "Custom" && stream2 !== "Custom") {
    url += '&stream1=' + encodeURIComponent(stream1);
    url += '&stream2=' + encodeURIComponent(stream2);

    // Stream 1 is custom, Stream 2 is not
  } else if (stream1 === "Custom" && stream2 !== "Custom") {
    const custom1 = `${CustomInputIP_A_src.value} && ${CustomInputIP_A_dst.value} && ${protocolA}`;
    url += '&stream1=' + encodeURIComponent(custom1);
    url += '&stream2=' + encodeURIComponent(stream2);

    // Stream 2 is custom, Stream 1 is not
  } else if (stream1 !== "Custom" && stream2 === "Custom") {
    const custom2 = `${CustomInputIP_B_src.value} && ${CustomInputIP_B_dst.value} && ${protocolB}`;
    url += '&stream1=' + encodeURIComponent(stream1);
    url += '&stream2=' + encodeURIComponent(custom2);

    // Both Streams are custom
  } else if (stream1 === "Custom" && stream2 === "Custom") {
    const custom1 = `${CustomInputIP_A_src.value} && ${CustomInputIP_A_dst.value} && ${protocolA}`;
    const custom2 = `${CustomInputIP_B_src.value} && ${CustomInputIP_B_dst.value} && ${protocolB}`;
    url += '&stream1=' + encodeURIComponent(custom1);
    url += '&stream2=' + encodeURIComponent(custom2);
  }

  url += '&Notification_Correlation_ID=' + encodeURIComponent(notificationCorrelationID);
  url += '&Analytics_ID=' + encodeURIComponent(analyticsID);
  url += '&Target_Period=' + encodeURIComponent(targetPeriod);

  return url;
}
```

Figure 4.31: First snippet of the *buildUrl* function

Following that, the function verifies the type of stream defined for flow 1 and flow 2 and acts accordingly. In Figure 4.32 the rest of the *buildUrl* function is shown. It handles the other two cases, and at the end appends the *Notification_Correlation_ID*, the *Analytics_ID*, and the *Target_Period* to the URL.

```
    } else if (stream1 !== "Custom" && stream2 === "Custom") {
      const custom2 = `${CustomInputIP_B_src.value} && ${CustomInputIP_B_dst.value} && ${protocolB}`;
      url += '&stream1=' + encodeURIComponent(stream1);
      url += '&stream2=' + encodeURIComponent(custom2);

      // Both Streams are custom
    } else if (stream1 === "Custom" && stream2 === "Custom") {
      const custom1 = `${CustomInputIP_A_src.value} && ${CustomInputIP_A_dst.value} && ${protocolA}`;
      const custom2 = `${CustomInputIP_B_src.value} && ${CustomInputIP_B_dst.value} && ${protocolB}`;
      url += '&stream1=' + encodeURIComponent(custom1);
      url += '&stream2=' + encodeURIComponent(custom2);
    }

    url += '&Notification_Correlation_ID=' + encodeURIComponent(notificationCorrelationID);
    url += '&Analytics_ID=' + encodeURIComponent(analyticsID);
    url += '&Target_Period=' + encodeURIComponent(targetPeriod);

    return url;
  }
}
```

Figure 4.32: Second snippet of the *buildUrl* function

Finally, after the URL is constructed the web redirects the current web page to the new URL specified by the *url* variable. Essentially, it redirects the user to a different web page based on the URL constructed using the *buildUrl* function. The *NWDAF* server then handles the request for the URL, as explained earlier.

```
const url = buildUrl(stream1, stream2, CustomInputIP_A_src, CustomInputIP_A_dst, Protocol_A,
CustomInputIP_B_src, CustomInputIP_B_dst, Protocol_B, Notification_Correlation_ID,
Analytics_ID, Target_Period);
window.location.href = url;
```

Figure 4.33: Final snippet of the *buildUrl* function

4.3.7.2 Analytics page

Figure 4.34 depicts the general overview of the *analytics.html* page. This page is loaded after the user submits the two flows defined in the *index.html* page. In other words, this page shows the analytics for the two subscribed video stream flows. The page is divided into two sections. The left section and the right section. In *HTML* terms, the page is divided into two `<div>`. The packet delay histogram, the throughput graphic, and the metrics for the first flow are shown on the left. On the right, the packet delay histogram, the throughput graphic, and the metrics for the second flow are shown. At the bottom, there is a button for the unsubscribe operation and an indication of the *Averaging window* selected for analysis. The Figure was taken right after the averaging window elapsed. As such, the histogram displayed takes into account all the packets captured during that time period. The metrics shown at the bottom, are also averages calculated taking into account the results for the averaging window, in this case, 15 seconds. Note that in this figure, none of the streams suffered any intervention from the *ISP* part, as this is just an illustrative image of the web page. Furthermore, the screenshot was captured with the intention of presenting the entire page rather than providing a good view of the results.

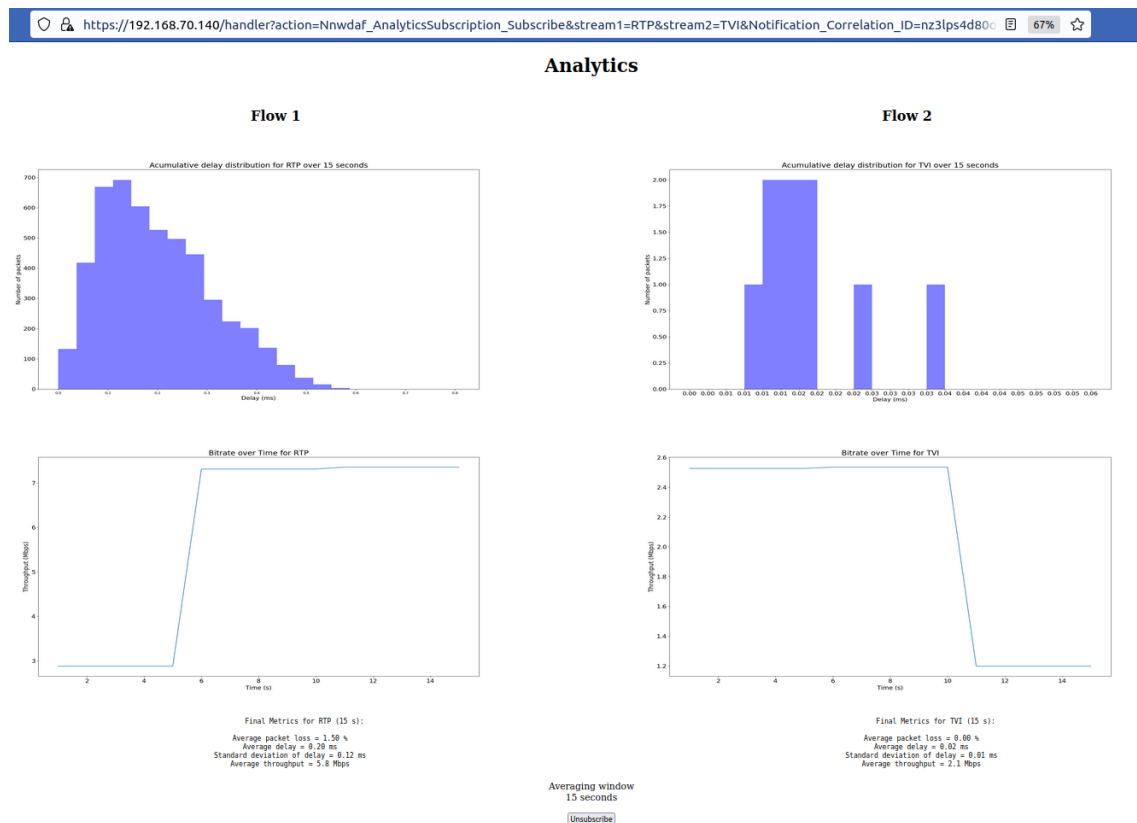


Figure 4.34: Overview of the *analytics.html* page

To update the metrics being displayed at the bottom of the page, the *updateFileContents* function is defined. This function will be executed every four seconds and request the metric files for both streams. Figure 4.35 depicts the *updateFileContents* function, as well as the auxiliary code to

call this function periodically. The *updateContentsPeriodically* function calls the *updateFileContents* two times, with the URL and the *elementId* associated with each stream. This function was configured to be executed every four seconds with the *setInterval JavaScript* method. The *updateFileContents* function uses Asynchronous JavaScript And XML (AJAX) [51] to perform various asynchronous GET requests to the server with the specified URL. AJAX is a technique that allows making asynchronous requests to the server without requiring a page reload. A callback function is defined to obtain the response from the server and update the document with the new metric values. To update the images a different technique was used. This technique is called *cache-busting* [52] and takes advantage of dynamic routes in the server and web browser caching. *Cache-busting* is deployed by attaching a timestamp to the image URLs. This will ensure that the browser will fetch the latest version of a file, instead of relying on cache. The *updateImage* function is called four in four seconds to fetch any new version of the image files available on the server.

```
// Function to fetch and update the file contents //
function updateFileContents(url, elementId) {
    const xhr = new XMLHttpRequest();
    xhr.open("GET", url);
    xhr.responseType = "text";
    xhr.onload = function() {
        document.getElementById(elementId).textContent = xhr.response;
    };
    xhr.send();
}

// Function to periodically update the file contents //
function updateContentsPeriodically() {
    updateFileContents("/metrics_A", "file-content_A");
    updateFileContents("/metrics_B", "file-content_B");
}

// Call the updateContentsPeriodically function initially //
updateContentsPeriodically();

// Set the interval to periodically update the contents //
const Metrics_interval = setInterval(updateContentsPeriodically, 4000);
```

Figure 4.35: *JavaScript* code to handle the update of the metric values displayed in the *analytics.html* page

The *JavaScript* code that handles the updates of the histogram and graph image files can be observed in Figure 4.36. The *updateImage* function is called four times, two times to fetch the images associated with the first stream and two other times to update the images of the second stream. As stated previously, each function call has a four-second frequency. Inside the *updateImage* function, the *elementId* is stored and the current time is concatenated to the URL of the image source (*image.src*).

```
// Function to update images //
function updateImage(url, elementId) {
    const image = document.getElementById(elementId);
    const timestamp = new Date().getTime();
    image.src = url + timestamp;
}

updateImage("/delay_histogram_Stream_A/image.png?cache=", "Histogram_Stream_A");
updateImage("/delay_histogram_Stream_B/image.png?cache=", "Histogram_Stream_B");
updateImage("/bitrate_graph_Stream_A/image.png?cache=", "Bitrate_graph_Stream_A");
updateImage("/bitrate_graph_Stream_B/image.png?cache=", "Bitrate_graph_Stream_B");

// Set the interval to periodically update the images //
const imageUpdateInterval = setInterval(function() {
    updateImage("/delay_histogram_Stream_A/image.png?cache=", "Histogram_Stream_A");
    updateImage("/delay_histogram_Stream_B/image.png?cache=", "Histogram_Stream_B");
    updateImage("/bitrate_graph_Stream_A/image.png?cache=", "Bitrate_graph_Stream_A");
    updateImage("/bitrate_graph_Stream_B/image.png?cache=", "Bitrate_graph_Stream_B");
}, 4000);
```

Figure 4.36: JavaScript code to handle the update of the images displayed in the *analytics.html* page

Handling the unsubscribe button click is much simpler than handling the submit/subscribe button click. Unlike, the *subscribe* operation, the URL defined for the *unsubscribe* operation is not complex. All that is needed is the *Subscription_Correlation_ID*. This value will be displayed in the browser whenever the user clicks on the unsubscribe button. Figure 4.37 depicts the code to handle the unsubscribe button click.

```
// Function to handle the unsubscribe button click //
function handleUnsubscribeClick() {
    let Subscription_Correlation_ID = '{ Subscription_Correlation_ID }';
    const url = '/handler?action=Nnwdaf_AnalyticsSubscription_Unsubscribe&Subscription_Correlation_ID='
        + encodeURIComponent(Subscription_Correlation_ID);
    window.location.href = url;
}
```

Figure 4.37: *handleUnsubscribeClick* JavaScript function

Chapter 5

System and *NWDAF* Validation

In this chapter, the system and the developed *NWDAF* will be validated using a series of tests. In section 5.1, the methodology used for the validation process is defined. In section 5.2, the components of the system are tested to ensure that the system is operational. In section 5.3, the use case proposed for this dissertation is tested. The last section, 5.4, discusses and analyzes the results obtained. Overall, this chapter's objective is to demonstrate that the developed *NWDAF* can be used to support the assessment of traffic differentiation by ISPs, with a particular focus on the practice of throttling.

5.1 Methodology

The first set of tests conducted during the evaluation of the system involved Component Testing. This phase aimed to assess the proper functioning of each individual component within the system. To achieve it, the ping utility was utilized. The final set of tests was conducted to validate the specific use cases defined for the proposed system. These tests involved subjecting the *5G Core Network* to a range of diverse network conditions in order to assess and validate the functionality of the developed *NWDAF*. The methodology adopted is slightly different for the two sets of tests. In the case of Component Testing, the objective was to validate the functionality of the components. This involved defining the test purpose and then conducting a functional validation using the ping tool. On the other hand, for Use Case Testing, the objective was to evaluate the performance of the *NWDAF*. Similar to Component Testing, the test purpose was defined, but the evaluation focused on assessing the performance of the *NWDAF* rather than validating individual components.

5.2 Component Testing

The goal of this set of tests is to verify the connectivity of the components of the system. To accomplish it, the connectivity between the *5G Core Network Docker* containers, including the *NWDAF*, is evaluated. Furthermore, the routing of the *UERANSIM* container is tested to guarantee that the packets sent to the Internet go through the *5G Core Network*.

5.2.1 Core Network and NWDAF

The ping utility was used to demonstrate that all containers in the *5G Core Network* were working. They must be able to communicate with the host PC to function correctly. Several pings were executed from the *demo-oai* bridge interface. This interface connects the host PC to the Core Network (192.168.70.0/24). If all pings are successful, this means that all the components are reachable. In Figure 5.1 is possible to observe the results of the ping messages sent to NRF, AMF, SMF, *UPF*, UDR, UDM, AUSF. All of them were successful, meaning that all components are behaving as expected.

```
PING 192.168.70.130 (192.168.70.130) from 192.168.70.1 demo-oai: 56(84) bytes of data.
64 bytes from 192.168.70.130: icmp_seq=1 ttl=64 time=0.038 ms
64 bytes from 192.168.70.130: icmp_seq=2 ttl=64 time=0.025 ms

--- 192.168.70.130 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1003ms
rtt min/avg/max/mdev = 0.025/0.031/0.038/0.006 ms

=====
PING 192.168.70.132 (192.168.70.132) from 192.168.70.1 demo-oai: 56(84) bytes of data.
64 bytes from 192.168.70.132: icmp_seq=1 ttl=64 time=0.028 ms
64 bytes from 192.168.70.132: icmp_seq=2 ttl=64 time=0.036 ms

--- 192.168.70.132 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1022ms
rtt min/avg/max/mdev = 0.028/0.032/0.036/0.004 ms

=====
PING 192.168.70.133 (192.168.70.133) from 192.168.70.1 demo-oai: 56(84) bytes of data.
64 bytes from 192.168.70.133: icmp_seq=1 ttl=64 time=0.029 ms
64 bytes from 192.168.70.133: icmp_seq=2 ttl=64 time=0.033 ms

--- 192.168.70.133 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1022ms
rtt min/avg/max/mdev = 0.029/0.031/0.033/0.002 ms

=====
PING 192.168.70.134 (192.168.70.134) from 192.168.70.1 demo-oai: 56(84) bytes of data.
64 bytes from 192.168.70.134: icmp_seq=1 ttl=64 time=0.037 ms
64 bytes from 192.168.70.134: icmp_seq=2 ttl=64 time=0.030 ms

--- 192.168.70.134 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1022ms
rtt min/avg/max/mdev = 0.030/0.033/0.037/0.003 ms

=====
PING 192.168.70.136 (192.168.70.136) from 192.168.70.1 demo-oai: 56(84) bytes of data.
64 bytes from 192.168.70.136: icmp_seq=1 ttl=64 time=0.025 ms
64 bytes from 192.168.70.136: icmp_seq=2 ttl=64 time=0.027 ms

--- 192.168.70.136 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1022ms
rtt min/avg/max/mdev = 0.025/0.026/0.027/0.001 ms

=====
PING 192.168.70.137 (192.168.70.137) from 192.168.70.1 demo-oai: 56(84) bytes of data.
64 bytes from 192.168.70.137: icmp_seq=1 ttl=64 time=0.028 ms
64 bytes from 192.168.70.137: icmp_seq=2 ttl=64 time=0.032 ms

--- 192.168.70.137 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1022ms
rtt min/avg/max/mdev = 0.028/0.030/0.032/0.002 ms

=====
PING 192.168.70.138 (192.168.70.138) from 192.168.70.1 demo-oai: 56(84) bytes of data.
64 bytes from 192.168.70.138: icmp_seq=1 ttl=64 time=0.037 ms
64 bytes from 192.168.70.138: icmp_seq=2 ttl=64 time=0.023 ms

--- 192.168.70.138 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1022ms
rtt min/avg/max/mdev = 0.023/0.030/0.037/0.007 ms
```

Figure 5.1: Successful pings from host PC to each Core component

The same procedure was executed for *NWDAF*. As can be observed in Figure 5.2 the *NWDAF* replied successfully to the Internet Control Message Protocol (ICMP) ping packets.

```
root@ruben-HP-Pavilion-Gaming-Laptop-15-ec2xxx:/# ping -I demo-oai -c 2 192.168.70.140
PING 192.168.70.140 (192.168.70.140) from 192.168.70.1 demo-oai: 56(84) bytes of data.
64 bytes from 192.168.70.140: icmp_seq=1 ttl=64 time=0.080 ms
64 bytes from 192.168.70.140: icmp_seq=2 ttl=64 time=0.033 ms

--- 192.168.70.140 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1031ms
rtt min/avg/max/mdev = 0.033/0.056/0.080/0.023 ms
```

Figure 5.2: Successful ping from host PC to *NWDAF*

5.2.2 UERANSIM

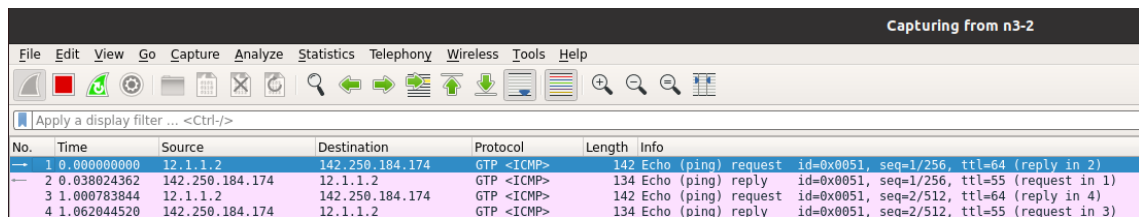
For the system to work as intended, the UE (represented by *UERANSIM*) needs to access the Internet through the 5G Core Network. The next test was performed to assess if the system respected this condition. During the test, a ping command was executed to the Internet, as depicted in Figure 5.3. The expected behavior is for the packets to be forwarded to the UPF for proper routing.

```
root@ruben-HP-Pavilion-Gaming-Laptop-15-ec2xxx:/# docker exec ueransim ping -c 2 google.com
PING google.com (142.250.184.174) 56(84) bytes of data.
64 bytes from mad07s23-in-f14.1e100.net (142.250.184.174): icmp_seq=1 ttl=55 time=17.2 ms
64 bytes from mad07s23-in-f14.1e100.net (142.250.184.174): icmp_seq=2 ttl=55 time=92.9 ms

--- google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 17.256/55.124/92.992/37.868 ms
```

Figure 5.3: Ping from *UERANSIM* to the Internet (google.com)

The *wireshark* screenshot of Figure 5.4 displays the network traffic observed at the *n3-2* interface during the execution of the ping commands. Four packets can be seen, representing the request and reply packets of the ICMP protocol. Furthermore, the packets are encapsulated inside the GTP-U tunnel that connects the access network to the core network. As such, it is possible to conclude that *UERANSIM* has a well-configured connection to the Internet.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	12.1.1.2	142.250.184.174	GTP <ICMP>	142	Echo (ping) request id=0x0051, seq=1/256, ttl=64 (reply in 2)
2	0.038024362	142.250.184.174	12.1.1.2	GTP <ICMP>	134	Echo (ping) reply id=0x0051, seq=1/256, ttl=55 (request in 1)
3	1.000783844	12.1.1.2	142.250.184.174	GTP <ICMP>	142	Echo (ping) request id=0x0051, seq=2/512, ttl=64 (reply in 4)
4	1.062044520	142.250.184.174	12.1.1.2	GTP <ICMP>	134	Echo (ping) reply id=0x0051, seq=2/512, ttl=55 (request in 3)

Figure 5.4: *Wireshark* capture showing ping packets sent from *UERANSIM* going through the UPF

5.3 Use Case Validation

5.3.1 eMBB

The following set of tests aimed at validating the *eMBB* use case defined as part of the work for this dissertation. The setup, as described in the Chapter 4, involved the monitoring of two video streams, one of which was negatively influenced by the *ISP*. The primary objective of this evaluation was to assess whether the implemented *NWDAF* is capable of capturing metrics that effectively showcase the detrimental effects caused by an *ISP* on the traffic flow of a video stream.

5.3.1.1 Delay Test

To simulate the *ISP* intervention on a traffic flow, the *tc* tool described in Chapter 4 is used. The first set of tests introduced packet delay into one of the video stream flows. Typically, an *ISP* does not directly add delay to a specific data flow. However, packet delay occurs as an indirect result

of the traffic shaping techniques employed by *ISPs*. Following the convention defined in Chapter 4, the affected video stream flow is known as the *tCAP*, while the unaffected flow is known as the *bCAP*. Using this procedure three tests were executed. The first focused on HTTP/2 traffic, the second on HTTP/3, and the third on the custom stream option. In all these test scenarios involving packet delay, a constant delay of 50 ms was introduced.

HTTP/2: For this test, TVI was the *tCAP* and SIC was the *bCAP*. For the TVI traffic flow, 50 ms of delay was added. The delay histograms depicted in Figure 5.5 clearly demonstrate that all TVI (*tCAP*) packets encountered approximately 50 ms of delay, consistent with the imposed conditions.

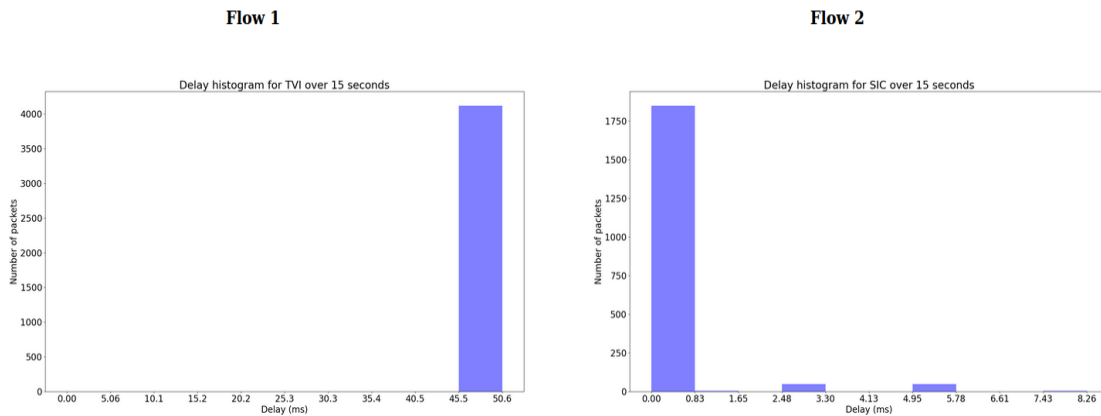


Figure 5.5: Delay histograms for HTTP/2 delay test

The metrics obtained as part of the test, illustrated in Figure 5.6, further demonstrate that the TVI stream was conditioned to 50 ms of packet delay. Additionally, it is observed that the packet loss ratio metric remains close to zero for both streams, indicating that there was no significant change in this metric. Furthermore, it is worth noting that the averaging window for this test was set to 15 seconds.

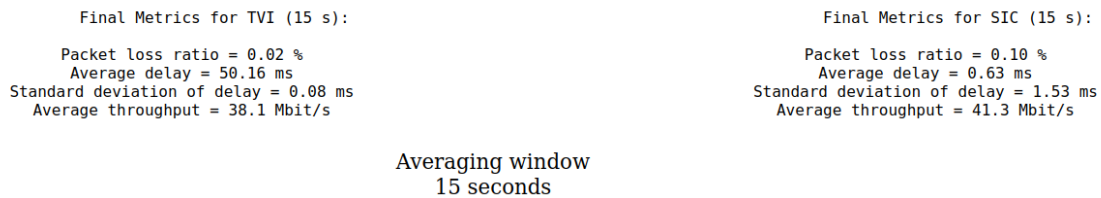


Figure 5.6: NWDAF metrics for HTTP/2 delay test

HTTP/3: This test was similar to the previous one, but it specifically targeted HTTP/3 video streams. The *tCAP* was “RTP” and the *bCAP* was YOUTUBE. The *tCAP* flow was affected with a constant packet delay of 50 ms. The delay histograms in Figure 5.7 provide evidence that the *ISP* introduced around 50 ms of packet delay to the “RTP” stream.

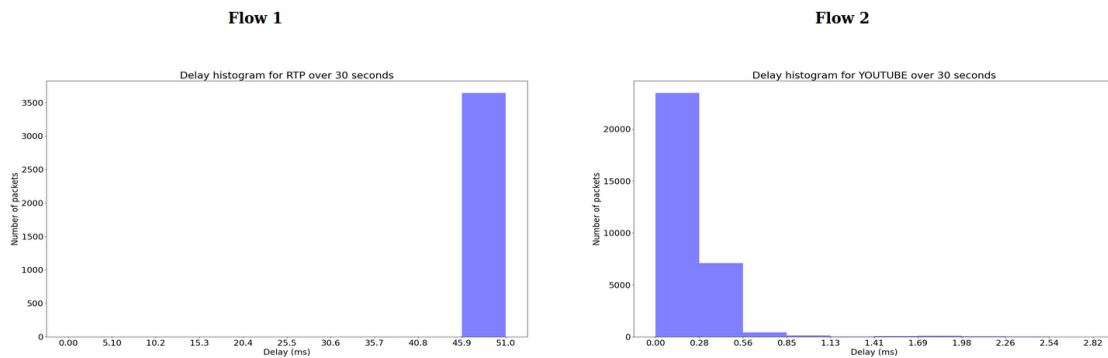
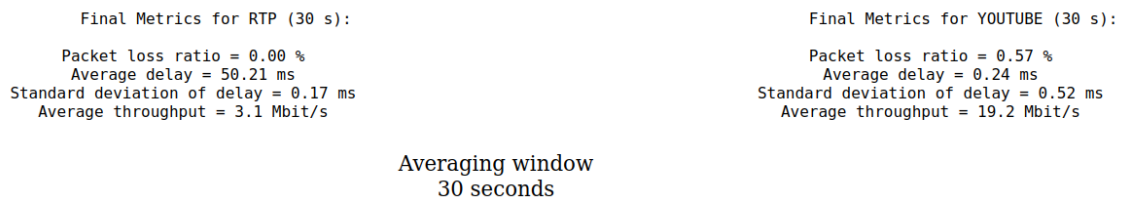


Figure 5.7: Delay histograms for HTTP/3 delay test

The metrics obtained by the *NWDAF*, showcased in Figure 5.8, support the claim that the *tCAP* stream packets were delayed by 50ms. Additionally, it demonstrates that the packet loss ratio metric was not changed by the *ISP*. Moreover, it should be noted that this test ran for twice the duration of the HTTP/2 test (30 s).

Figure 5.8: *NWDAF* metrics for HTTP/3 delay test

Custom stream: The final delay metric test utilized the custom stream option. With this option, the traffic can be either HTTP/2 or HTTP/3. So the comparison can be made between two flows using the same protocol or between two flows using different protocols. In this scenario, the decision was made to compare a custom HTTP/2 stream with the pre-defined HTTP/3 “RTP” stream. The custom stream was the *tCAP* and 50 ms of delay were added to it. To obtain the source IP address that defines a custom stream, one would have to analyze the desired video stream flow in the browser using the web developer tools. To access this information, the user can right-click on the browser and select ‘inspect’. The web developer tool will show up at the bottom of the browser, as seen in Figure 5.9. The ‘network’ tab will present all the HTTP requests generated by the browser. To filter just the requests related to video activity, ‘media’ can be chosen. If the user clicks on an HTTP request, a window showing the contents of that request will show up on the right side. The summary of the Headers will display the version of HTTP used.

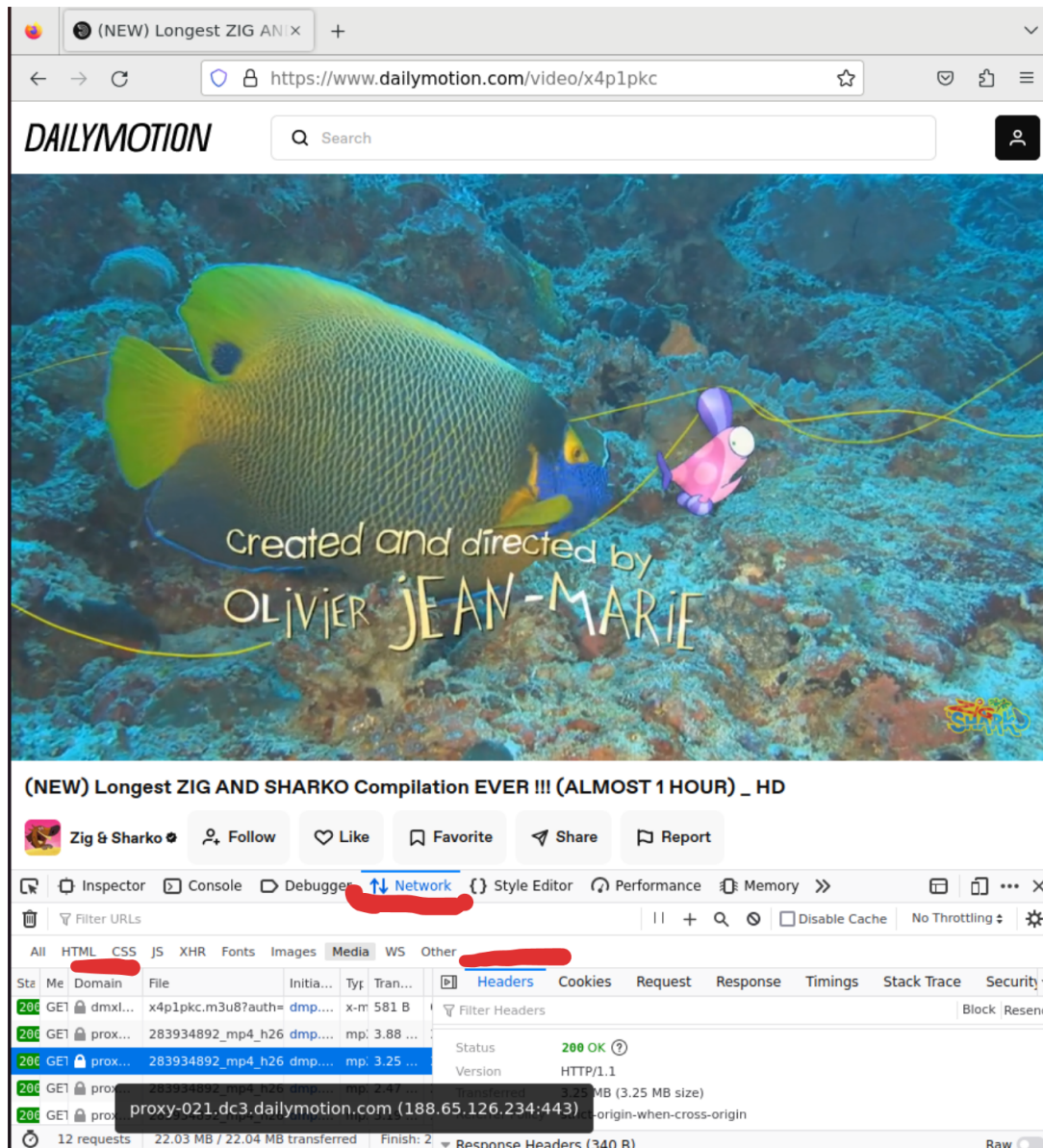


Figure 5.9: Retrieval of the source IP address and the protocol name for a custom stream

In this example, HTTP/1 is shown. Since this is a secure website that uses HTTPS, the HTTP/2 option of the *NWDAF* will still work. To obtain the source IP address of the requested content, one can hover the mouse over the domain name column on the left side. For the rest of the *eMBB* tests, the custom stream used is the same as the one shown in 5.9, *Dailymotion*. Note that the complete IP address might not always be the same for the same stream. One might have to use a network as the source IP address instead of a complete IP. For example, in this case, it might be feasible to use 188.65.126.0/24 as the custom source IP address. The delay histograms in Figure 5.10 confirm that the custom stream packets were indeed delayed by 50 ms.

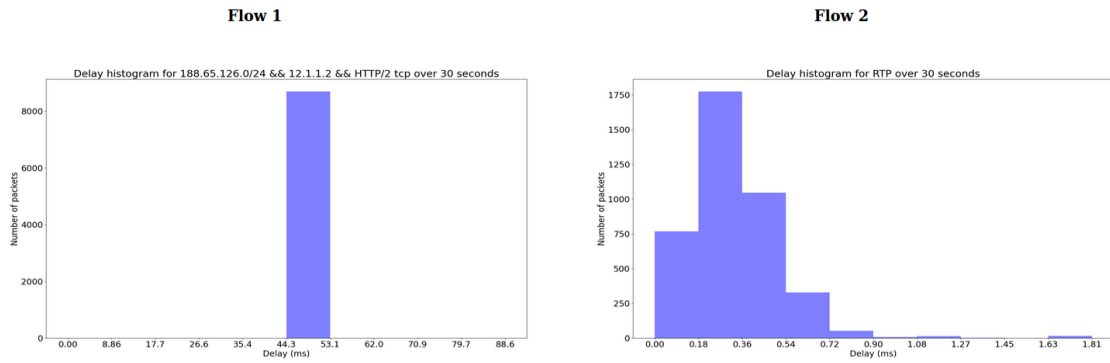
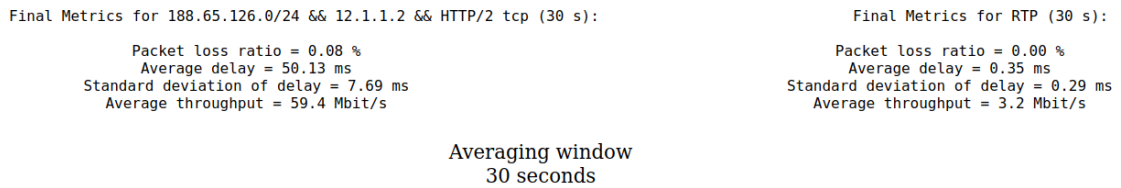


Figure 5.10: Delay histograms for custom stream delay test

Figure 5.11 depicts the metrics collected by the *NWDAF* for this test. The average delay metric confirms the values provided by the delay histograms.

Figure 5.11: *NWDAF* metrics for custom stream delay test

5.3.1.2 Packet Loss Test

In this set of tests, packet loss was injected into one of the video stream flows using the *tc* tool. The objective was to evaluate if the *NWDAF* can detect when a flow is experiencing packet loss. As in the case of the delay test, it is worth noting that packet loss is not directly introduced by the *ISPs*, but rather is a byproduct of the *ISP*'s traffic shaping techniques. The set of tests consisted of three distinct evaluations. The initial test concentrated on examining traffic utilizing the HTTP/2 protocol, while the second test specifically assessed the behavior of traffic using HTTP/3. Lastly, the third test focused on analyzing the custom stream option. The introduced packet loss ratio value was standardized at 20% for all the tests involving packet loss.

HTTP/2: In the given scenario, the *tCAP* stream (SIC) was subjected to a significant packet loss of 20%. The metrics provided by *NWDAF*, 5.12, indicate that SIC was experiencing approximately 20% of packet loss. Additionally, the average delay was reported to be low, although there was a notable standard deviation associated with it.

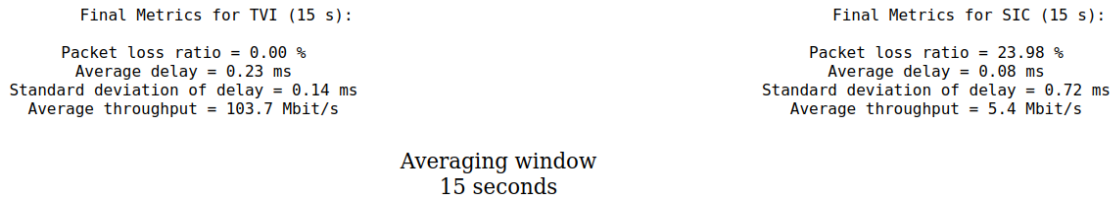


Figure 5.12: NWDAF metrics for HTTP/2 loss test

HTTP/3: A packet loss of 20% was added to YOUTUBE. Figure 5.13 depicts the metrics obtained by the NWDAF. The YOUTUBE stream has 21.73% of packet loss, which is close to the intended value.

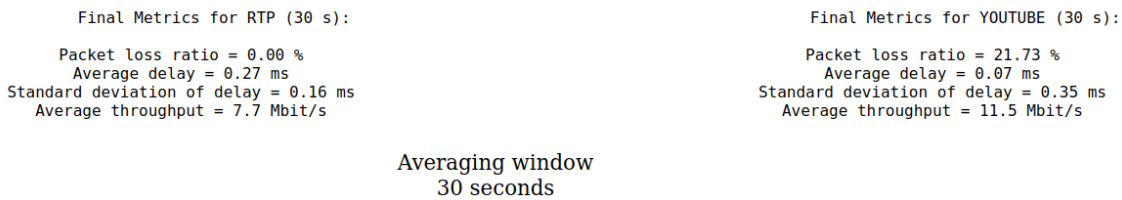


Figure 5.13: NWDAF metrics for HTTP/3 loss test

Custom stream: As the custom stream, a Dailymotion stream was utilized, and 20% of packet loss was introduced. The metrics connected to the custom stream are presented on the left side of Figure 5.14, and a number relative near to 20% (18.40%) verifies the test.

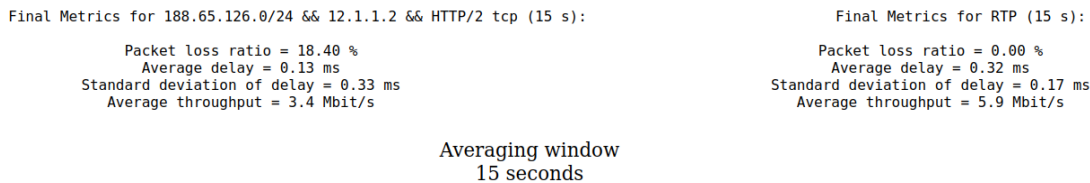


Figure 5.14: NWDAF metrics for custom stream loss test

5.3.1.3 Packet delay and Packet Loss Test

In the final set of tests, packet loss and packet delay were introduced into the *tCAP* video flow stream. The goal was to verify whether the NWDAF was capable of identifying instances of packet loss and packet delay in the target network flow. Like the other set of tests, three different evaluations were performed. The first test focused on traffic using the HTTP/2 protocol, whereas the second test particularly analyzed the behavior of traffic using HTTP/3. Finally, the final test examined the custom stream option.

HTTP/2: For this test 20% of packet loss and 50 ms of delay was added to the TVI stream. In Figure 5.15, the delay histograms showcase that the TVI packets are experiencing a delay of approximately 50 ms.

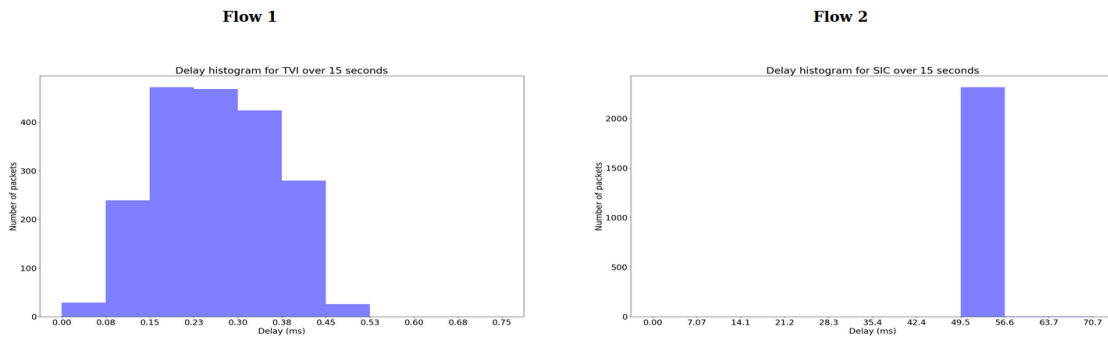
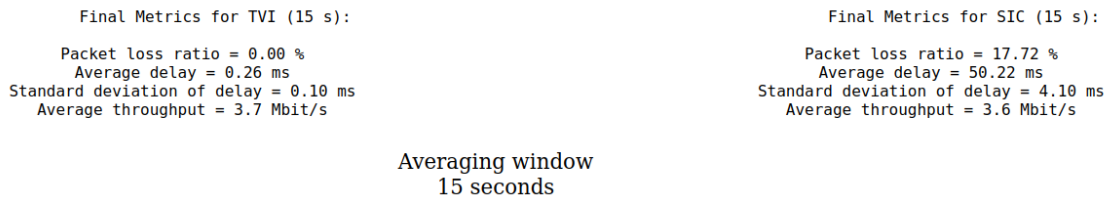


Figure 5.15: Delay histograms for HTTP/2 delay and loss test

Based on the metrics presented in Figure 5.16, it can be inferred that the *NWDAF* metrics aligned with the delay values depicted in the histograms. Additionally, the metrics indicated that the TVI stream experienced approximately 17.22% packet loss, which is close to the intended value set for the test.

Figure 5.16: *NWDAF* metrics for HTTP/2 delay and loss test

HTTP/3: For this test, the “RTP” stream was affected with 20% of packet loss and 50 ms of packet delay. The delay histograms of Figure 5.17 confirmed that the “RTP” stream was experiencing packet delay values of around 50 ms.

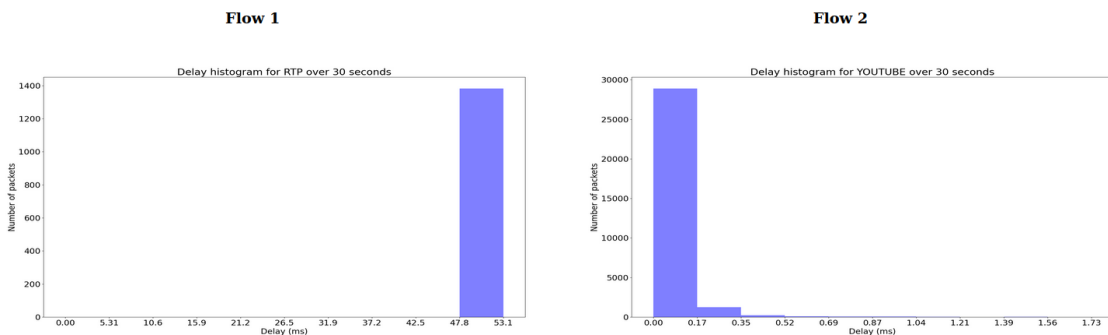


Figure 5.17: Delay histograms for HTTP/3 delay and loss test

The *NWDAF* metrics displayed in Figure 5.18 further confirmed the accuracy of the packet delay values and revealed that the “RTP” stream experienced approximately 23% packet loss.

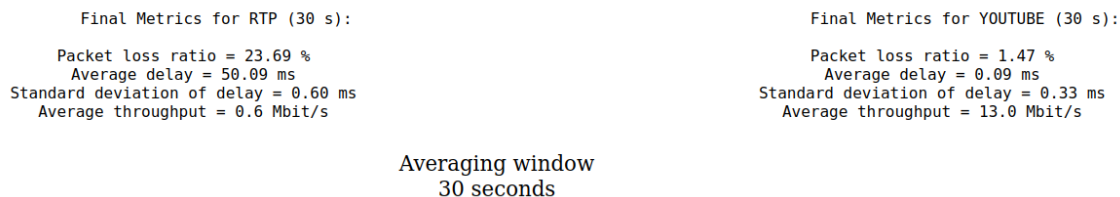


Figure 5.18: NWDAF metrics for HTTP/3 delay and loss test

Custom stream: This final test subjected the custom stream (Dailymotion) to 50 ms of delay and 20% of packet loss. There was a key difference in this test compared to the others. The *bCAP* stream (SIC) was not active during the time period of the test, therefore no histogram was created for *bCAP*. As such, Figure 5.19 only displays the delay histogram for the *tCAP*. It is demonstrated that the custom stream was affected by a packet delay of around 50 ms.

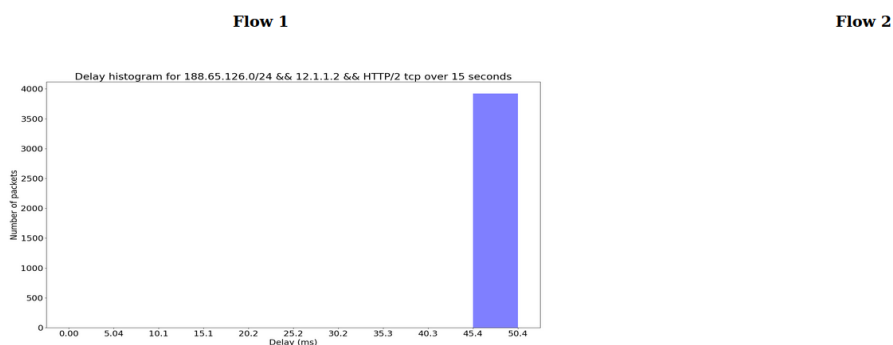


Figure 5.19: Delay histograms for custom stream delay and loss test

Figure 5.20 depicts the metrics collected by the NWDAF for this test. On the left, the provided average delay metric verifies that the custom stream experienced a packet delay of around 50 ms and a packet loss close to 20%. On the right, the NWDAF displayed a message stating that no network activity occurred for the *bCAP* stream.

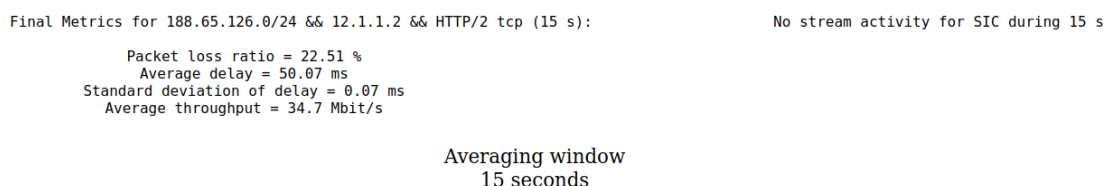


Figure 5.20: NWDAF metrics for custom stream delay and loss test

5.4 Discussion

The tests carried out allowed the evaluation of the basic functionality of the system and the validation of the developed NWDAF function. The first testing phase, Component Testing, sought to evaluate and validate the individual performance of each component of the system. The second set

of tests, Use Case Validation, considered an experimental scenario focused on a 5G network slice such as *eMBB*. Taking all the test findings into consideration, it is possible to infer that the created *NWDAF* can appropriately provide metrics that may be utilized for future traffic discrimination assessment.

Chapter 6

Conclusions

The main objective of this dissertation was to develop an *NWDAF* function capable of detecting traffic differentiation (*TD*) inside the 5G Core Network. The *NWDAF* should obtain standardized metrics from the network and expose that information to the telecom regulator. A specific use case was defined, in which the *NWDAF* focused on detecting *TD* for *eMBB* traffic. The proposed solution consisted of a system designed to allow the connection of a *UE* to two video stream *CAPs* on the Internet through the 5G Core Network (*5GC*). The *NWDAF* would then calculate metrics based on the captured traffic on the *UPF*.

The open-source software *OpenAirInterface* (*OAI*) was used for implementing the *5G Core Network* (*5GC*), while the *UERANSIM* simulator was employed for simulating the User Equipment (*UE*) and the Radio Access Network (*RAN*).

The developed *NWDAF* provides two web pages, which are accessible through a web browser. The first serves as the video stream selection page and the second displays the information gathered via histograms, graphics, and metrics. The *HTTP/2* protocol was used, as defined by the standard. Additionally, a self-signed certificate was created to enable traffic encryption using the TLS protocol.

An implementation challenge encountered was the need to become familiar with *Python* web frameworks. Initially, the *Flask* web framework was considered, but it was later discovered, through testing and research, that it did not support the *HTTP/2* protocol. Therefore, the *quart* web framework was chosen as an alternative. This framework is built upon the *flask* framework, which means that the syntax and structure are quite similar. As a result, the transition from *flask* to *quart* was relatively smooth, leveraging the existing familiarity with *flask*.

The learning curve of *JavaScript* presented additional challenges during the development process. Certain functionalities on the web pages, such as handling button clicks or drop-down menus, required *JavaScript* code to be implemented and executed. This required familiarity with *JavaScript* and understanding its syntax and concepts in order to effectively incorporate the desired interactive elements into the web pages.

Furthermore, some challenges were found when developing the *python* script to calculate network performance metrics and the implementation of the commands to manipulate traffic with

the *tc* tool. When dealing with *HTTP/2* traffic over TLS, the TCP protocol issued an unexpected problem that required some modifications to the configurations of the *ext-oai-dn* container. A considerable amount of time was dedicated to researching and finding the appropriate solutions to address the encountered problem. Regarding traffic shaping, the challenge lied in constructing the appropriate filter. As the packets were encapsulated within a GTP tunnel, filtering for a desired source IP address was not as straightforward as expected. A more complex filter had to be devised, in which the intended source IP address had to be matched within the packets. This required a deeper understanding of the packet structure and the GTP encapsulation protocol to accurately identify and manipulate the desired traffic.

A battery of tests was conducted to validate the proposed solution. The initial set of tests aimed to evaluate the functionality of the system modules. It involved verifying that each component performed its intended tasks correctly and that they could communicate and interact with each other effectively. The second set of tests focused on assessing the performance of the developed *NWDAF* function. In particular, a specific *eMBB* scenario was tested. This scenario involved monitoring two video streams originating from two different *CAPs*.

Despite the potential for improvement in the proposed solution, the testing results indicate the successful accomplishment of the primary objectives of this dissertation. Hence, the main original contributions are as follows: 1) Creation of an *NWDAF* function that can act as a tool to assess *TD*; 2) Enabling the process by which a regulator can gather performance data on the 5G network to assess traffic differentiation by *ISPs*. These contributions represent a significant breakthrough, considering the unprecedented application of *NWDAF* in assessing net neutrality compliance.

6.1 Limitations and Future work

Even though the *NWDAF* developed for this dissertation was validated and achieved the proposed objectives, there is still room for improvement. The main limitation of this work is that it can only obtain metrics from the *UPF*. An *ISP* might perform traffic discriminatory actions on traffic on other components of the 5G Network, more specifically on the New Generation Radio Access Network (NG-RAN). Therefore, monitoring the traffic between the interfaces of the *UPF* will not suffice to conclude if an *ISP* negatively influences a traffic stream.

NG-RAN comprises a group of *gNBs* that are linked to the 5G Core Network (*5GC*) via the NG interface. Interconnection between *gNBs* is facilitated through the Xn interface. A *gNB* is typically composed of a *gNB* Central Unit (*gNB-CU*) and one or multiple *gNB* Distributed Units (*gNB-DUs*). The connection between a *gNB-CU* and a *gNB-DU* is established through the F1 interface [53].

The diagram presented in Figure 6.1 illustrates an example of the overall architecture of NG-RAN. It highlights that the *ISP* may impact the performance of a traffic stream at various points throughout the NG-RAN. These points are denoted by dotted lines between network interfaces. Displayed in green is the connection between the two interfaces of the *UPF*. This point of observation was the primary focus of this dissertation. In blue is the connection between the NG-U

interface and the F1-U interface, and in red is the link between the N6 interface and the F1-U interface. An *ISP* may intervene at these points of the system rather than just at the *UPF*. If that is the case, then using the metrics obtained at the *UPF* to assess traffic differentiation will provide unreliable results.

In future work, one can apply the same method employed in this dissertation to develop an *NWDAF* capable of monitoring traffic between two other interfaces of the 5G Network, such as the ones depicted in this example.

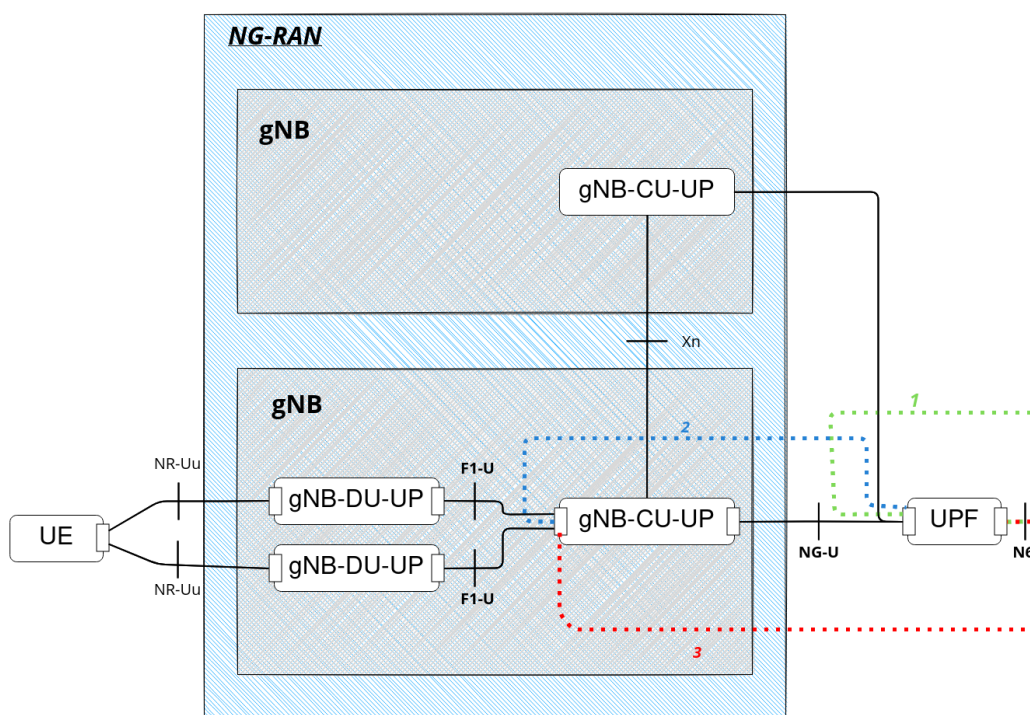


Figure 6.1: Overall architecture of NG-RAN

Furthermore, there is room for improvement in both the web server and the web page of the *NWDAF*. Enhancements can be made to the visual aspect of the web page to provide better flexibility in terms of zooming. However, the most significant area for improvement lies in the Subscription service. Currently, the *NWDAF* server supports only one Subscription at a time. In future work, it would be very useful to scale this feature to allow various Subscriptions at a time and options to add, remove, and edit them. Moreover, there is the potential of exploring other use cases such as *URLLC* (particularly concerning cloud gaming) or even IoT scenarios.

References

- [1] Riccardo Ravaoli, Guillaume Urvoy-Keller, and Chadi Barakat. Towards a general solution for detecting traffic differentiation at the internet access. In *2015 27th International Teletraffic Congress*, pages 1–9, 2015. doi:10.1109/ITC.2015.8.
- [2] T. Garrett, L.E. Setenareski, L.M. Peres, L.C.E. Bona, and E.P. Duarte. Monitoring network neutrality: A survey on traffic differentiation detection. *IEEE Communications Surveys and Tutorials*, 20:2486–2517, 2018. doi:10.1109/COMST.2018.2812641.
- [3] B. Van Schewick and D. Farber. Point/counterpoint: Network neutrality nuances. *Communications of the ACM*, 52:31–37, 2009. doi:10.1145/1461928.1461942.
- [4] David Miguel de Almeida Coimbra Maia. *Control and Positioning of a 5G Radio Access Node Deployed in a Mobile Robotic Platform*. PhD thesis, FEUP, 2022.
- [5] 3GPP. 5G;Architecture enhancements for 5G System (5GS) to support network data analytics services. (23.288), 03 2022. Version 17.4.0. URL: <https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3579>.
- [6] J. Walrand and S. Parekh. *Communication Networks: A Concise Introduction, Second Edition*, volume 10. 2017. doi:10.2200/S00804ED2V01Y201709CNT020.
- [7] Itu-t an architectural framework for support of quality of service in packet networks, 2004.
- [8] Docker. What is a container? | docker. Available from <https://www.docker.com/resources/what-container/>, 2023. Accessed: June 2, 2023.
- [9] 5G Core Network – OpenAirInterface. Accessed: 2022-12-24. URL: <https://openairinterface.org/oai-5g-core-network-project/>.
- [10] Free5GC. Accessed: 2022-12-24. URL: <https://www.free5gc.org/>.
- [11] Open5GS - Open source project of 5GC and EPC. Accessed: 2022-12-24. URL: <https://open5gs.org/>.
- [12] Magma linux foundation project. Accessed: 2022-12-24. URL: <https://magmacore.org/>.
- [13] aligungr/ueransim: Open source 5G UE and RAN (gNodeB) implementation. Accessed: 2022-12-24. URL: <https://github.com/aligungr/UERANSIM>.
- [14] free5G/free5GRAN: free5GRAN is an open-source 5G ran stack. the current version includes a receiver which decodes MIB & SIB1 data. it also acts as a cell scanner. free5GRAN

- works in SA mode. Accessed: 2022-12-24. URL: <https://github.com/free5G/free5GRAN>.
- [15] The srslte project is evolving. Accessed: 2022-12-24. URL: <https://www.srslte.com/srslte-srsran>.
- [16] Robert Beverly, Steven Bauer, and Arthur Berger. The internet is not a big truck: Toward quantifying network neutrality. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4427 LNCS:135–144, 2007. doi:10.1007/978-3-540-71617-4_14.
- [17] Ying Zhang, Zhuoqing Morley Mao, and Ming Zhang. Detecting traffic differentiation in backbone isps with netpolice. *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC*, pages 103–115, 2009. doi:10.1145/1644893.1644905.
- [18] Marcel Dischinger, Krishna P. Gummadi, Massimiliano Marcon, Ratul Mahajan, Saikat Guha, and Stefan Saroiu. Glasnost: Enabling end users to detect traffic differentiation. *Proceedings of NSDI 2010: 7th USENIX Symposium on Networked Systems Design and Implementation*, pages 405–418, 2010.
- [19] Glasnost: Test if your isp is shaping your traffic. Accessed: 2022-12-21. URL: <https://broadband.mpi-sws.org/transparency/bttest.php>.
- [20] G. Lu, Y. Chen, S. Birrer, F.E. Bustamante, and X. Li. Popi: A user-level tool for inferring router packet forwarding priority. *IEEE/ACM Transactions on Networking*, 18:1–14, 2010. doi:10.1109/TNET.2009.2020799.
- [21] Partha Kanuparth and Constantine Dovrolis. Diffprobe: Detecting isp service discrimination. *Proceedings - IEEE INFOCOM*, 2010. doi:10.1109/INFOCOM.2010.5461983.
- [22] Enrico Gregori, Valerio Luconi, and Alessio Vecchio. Neutmon: Studying neutrality in european mobile networks. pages 523–528. Institute of Electrical and Electronics Engineers Inc., 7 2018. doi:10.1109/INFOCOMW.2018.8407022.
- [23] Fangfan Li, Arian Akhavan Niaki, David Choffnes, Phillipa Gill, and Alan Mislove. A large-scale analysis of deployed traffic differentiation practices. *SIGCOMM 2019 - Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication*, pages 130–144, 8 2019. doi:10.1145/3341302.3342092.
- [24] Wehe: Check your isp for net neutrality violations. Accessed: 2022-12-22. URL: <https://wehe.meddle.mobi/index.html>.
- [25] Docker. Docker compose overview | docker documentation. Available from <https://docs.docker.com/compose/>, 2023. Accessed: June 11, 2023.
- [26] Docker. docker images | docker documentation. Available from <https://docs.docker.com/engine/reference/commandline/images/>, 2023. Accessed: June 11, 2023.
- [27] Docker. Volumes | docker documentation. Available from <https://docs.docker.com/storage/volumes/>, 2023. Accessed: June 11, 2023.
- [28] Nanyang Technological University (NTU). Environment Variables in Windows/macOS/Linux PATH, CLASSPATH, JAVA_HOME. Available from https://www3.ntu.edu.sg/home/ehchua/programming/howto/Environment_Variables.html, 2023. Accessed: June 11, 2023.

- [29] Docker. Dockerfile reference | docker documentation. Available from <https://docs.docker.com/engine/reference/builder/>, 2023. Accessed: June 11, 2023.
- [30] Tutorialspoint. Shell scripting tutorial. Available from https://www.tutorialspoint.com/unix/shell_scripting.htm, 2023. Accessed: June 11, 2023.
- [31] X.Org. Xauth(1) manual page. Available from <https://www.x.org/archive/X11R6.8.1/doc/xauth.1.html>, 2023. Accessed: June 11, 2023.
- [32] Wireshark. Wireshark · go deep. Available from <https://www.wireshark.org/>, 2023. Accessed: June 11, 2023.
- [33] Wireshark. tshark(1). Available from <https://www.wireshark.org/docs/man-pages/tshark.html>, 2023. Accessed: June 11, 2023.
- [34] Matplotlib. Matplotlib — visualization with python. Available from <https://matplotlib.org/>, 2023. Accessed: June 11, 2023.
- [35] Pandas. pandas - python data analysis library. Available from <https://pandas.pydata.org/>, 2023. Accessed: June 11, 2023.
- [36] Python. statistics — mathematical statistics functions. Available from <https://docs.python.org/3/library/statistics.html>, 2023. Accessed: June 11, 2023.
- [37] Wireshark. capinfos(1). Available from <https://www.wireshark.org/docs/man-pages/capinfos.html>, 2023. Accessed: June 11, 2023.
- [38] Manpages ubuntu. Ubuntu manpage: tc - show / manipulate traffic control settings. Available from <https://manpages.ubuntu.com/manpages/xenial/man8/tc.8.html>, 2023. Accessed: June 14, 2023.
- [39] Ieee standard for ethernet. *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, pages 1–5600, 2018. doi:10.1109/IEEESTD.2018.8457469.
- [40] Wikipedia. Maximum transmission unit - wikipedia. Available from https://en.wikipedia.org/wiki/Maximum_transmission_unit, 2023. Accessed: June 11, 2023.
- [41] Wikipedia. Tcp offload engine - wikipedia. Available from https://en.wikipedia.org/wiki/TCP_offload_engine, 2023. Accessed: June 11, 2023.
- [42] Wikipedia. Network interface controller - wikipedia. Available from https://en.wikipedia.org/wiki/Network_interface_controller, 2023. Accessed: June 11, 2023.
- [43] Flask. Welcome to flask — flask documentation (2.3.x). Available from <https://flask.palletsprojects.com/en/2.3.x/>, 2023. Accessed: June 11, 2023.
- [44] Quart. An async python web microframework to build web applications. Available from <https://github.com/pallets/quart>, 2023. Accessed: June 11, 2023.
- [45] Docker. docker run | docker documentation. Available from <https://docs.docker.com/engine/reference/commandline/run/>, 2023. Accessed: June 11, 2023.

- [46] Docker. dockerd | docker documentation. Available from <https://docs.docker.com/engine/reference/commandline/dockerd/>, 2023. Accessed: June 11, 2023.
- [47] Openssl. The openssl project develops and maintains the openssl software - a robust, commercial-grade, full-featured toolkit for general-purpose cryptography and secure communication. Available from <https://www.openssl.org/>, 2023. Accessed: June 11, 2023.
- [48] W3schools. Css tutorial. Available from <https://www.w3schools.com/css/>, 2023. Accessed: June 11, 2023.
- [49] W3schools. Html tutorial. Available from <https://www.w3schools.com/html/>, 2023. Accessed: June 11, 2023.
- [50] Pgjones. Routing — quart 0.17.0 documentation. Available from https://pgjones.gitlab.io/quart/how_to_guides/routing.html, 2023. Accessed: June 11, 2023.
- [51] W3schools. Ajax introduction. Available from https://www.w3schools.com/js/js_ajax_intro.asp, 2023. Accessed: June 11, 2023.
- [52] KeyCDN Support. What is cache busting? Available from <https://www.keycdn.com/support/what-is-cache-busting>, 2023. Accessed: June 11, 2023.
- [53] 3GPP. Ts 138 401 - v15.2.0 - 5g; ng-ran; architecture. (38.401), 2018. Version 15.2.0. URL: <https://portal.etsi.org/TB/ETSIDeliverableStatus.aspx>.