Network Slicing in 5G Connected Data Network for Smart Grid Communications Using Programmable Data Plane

Munavar Harris Thottoli

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 12.10.2021

Supervisor

Prof. Raimo Kantola

Advisor

MSc. Mandana Ghasemi

Aalto University
School of Electrical Engineering

Copyright © 2021 Munavar Harris Thottoli

Aalto University
School of Electrical Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Munavar Harris Thottoli

Title Network Slicing in 5G Connected Data Network for Smart Grid Communications Using Programmable Data Plane

Degree programme Master's Programme in Computer, Communication and Information Sciences

Major Communications Engineering **Code of major** ELEC3029

Supervisor Prof. Raimo Kantola

Advisor MSc. Mandana Ghasemi

Date 12.10.2021 **Number of pages** 78+24 **Language** English

Abstract

Due to the technological advancements in communications, contemporary smartgrids have started to adopt Fifth Generation (5G) mobile networks for communications. Communication between Supervisory Control and Data Acquisition (SCADA) systems and Remote Terminal Units (RTUs) in smart grid environment utilizes the IEC 60870-5-104 protocol. It is a Transmission Control Protocol/Internet Protocol (TCP/IP) based protocol where data is transmitted in unencrypted form. Smart grids adopting 5G networking for communications are not isolated appropriately. Therefore, smart grids are still insecure against cyberattacks. With respect to recent developments in data plane programming, new networking paradigms can be realized including progressive ways of isolating smart grid traffic from normal traffic in a data plane.

The aim of the thesis is to explore the usage of data plane programming to isolate and secure smart grid traffic into a network slice in 5G networks. This thesis successfully develops a flexible and efficient 5G network slicing solution based on P4 (Programming Protocol-Independent Packet Processors) language framework. Slice isolation is achieved with varied packet rates in slices as well as blocking devices from one slice communicating to the devices in another slice. The network slicing solution enables 5G equipped RTUs to be connected with SCADA in the Data Network in an isolated manner. A P4-based packet tagging solution is also presented where smart grid packets are tagged with specific Differentiated Services Code Point (DSCP) in the Internet Protocol (IP) headers to aid network slicing. DSCP values in the IP headers are used by the P4-based slicing solution to classify smart grid packets appropriately and push them into network slices. Both the network slicing and the DSCP tagging solutions are implemented with P4 software switch known as the Behavioral Model version 2 (BMv2). The network slicing performance is assessed in an experimental 5G testbed, which is powered by an opensourced 5G core. Basestation and User Equipment (UE) elements for connecting the RTU are simulated using appropriate software. The network slices are examined carefully in this thesis as well as their ability to provide Quality of Service (QoS) for the services hosted in the slices.

Keywords P4, SDN, 5G, Network slicing, Smart grid, Data plane programming, QoS

Preface

First of all, I would like to thank my family for providing me with the means and emotional support to complete my Master's studies at Aalto University.

I would like to dedicate huge amounts of thanks to my thesis Supervisor, Prof. Raimo Kantola. He is the central pillar of this thesis and without his precise supervision and immense support throughout the duration of this thesis project, it would have been almost impossible to complete.

This thesis work was funded by Business Finland through the 5G VIIMA Project.

Furthermore, I would like to thank Jose Costa-Requena, Saimanoj Katta Rokkaiah and the employees of Cumucore Oy for providing me with advisory support and consultation, along with the provision of computing hardware and other necessary resources for the implementation of this project.

In addition, I would like to thank Juha Järvinen for the provision of the APU devices and Aalto University's ComNet IT for providing workstations, computer peripherals, and technical support throughout the duration of this thesis work.

Last but not least, I would like to thank Espresso House (non-sponsor) for providing me with lovely cups of coffee to sink my tears of despair throughout the thesis workload timeline.

Otaniemi, 12/10/2021

Munavar H. Thottoli

Contents

Abstract iii

Preface iv

Contents v

List of Figures vii

List of Tables x

Listings x

Symbols and Abbreviations xi

1 Introduction 1
- 1.1 Research Problem and Objectives . . . 2
- 1.2 Thesis Scope . . . 2
- 1.3 Structure of the Thesis . . . 3

2 Background 4
- 2.1 Quality of Service and Isolation Techniques in IP Network . . . 4
- 2.2 General Architecture of the Smart Grids . . . 6
- 2.3 IEC Transmission Protocols . . . 7
  - 2.3.1 IEC 60870-5 protocol stack . . . 8
  - 2.3.2 IEC 104 communications . . . 8
  - 2.3.3 The IEC 104 protocol . . . 8
  - 2.3.4 The APCI frame . . . 10
  - 2.3.5 Format of the ASDU frame . . . 12
- 2.4 P4 (Programming Protocol-Independent Packet Processors) . . . 16
  - 2.4.1 Traditional VS programmable switches . . . 18
  - 2.4.2 Programming the data plane with P4 . . . 20
- 2.5 P4 Architectural Model and the BMv2 Switch . . . 23
  - 2.5.1 The V1model . . . 23
  - 2.5.2 P4 extern objects - meters and counters . . . 28
  - 2.5.3 The BMv2 software P4 switch . . . 30
- 2.6 5G Core Network Architecture and Free5GC . . . 31
  - 2.6.1 The 5G Core . . . 32
  - 2.6.2 Free5GC . . . 34
  - 2.6.3 Network Slicing in 5G . . . 34

3 Design and Implementation 36
- 3.1 Design considerations for the 5G testbed . . . 36
- 3.2 System architecture for the 5G testbed and for network slicing . . . 37
- 3.3 Implementation of P4-based DSCP tagger . . . 41
- 3.4 Implementation of P4-based Network Slicer . . . 47

3.4.1 Ingress Processing for Network Slicer 48
3.4.2 Egress Processing for Slicing 50
3.4.3 Slice Implementation and Regulating Packet Rates 51

4 Results **55**

5 Discussion **63**
5.1 Performance of the BMv2 Software Switch 63
5.2 Networking Requirements for Smart Grids 68
5.3 What if SCADA-RTU Connection is Encrypted? 69
5.4 Technical Challenges 70
5.4.1 The P4-NetFPGA hardware 70
5.4.2 Other relevant challenges 72

6 Conclusion **74**
6.1 Future Work 75

References **77**

Appendix A **79**

A P4 Code for DSCP Tagger **79**

Appendix B **85**

B P4 Code for Network Slicer **85**

Appendix C **91**

C P4 Code for Network Slicer in the RAN Segment of the 5G Network **91**

Appendix D **99**

D Conceptualized 5G testbed with Network Slicer and DSCP Tagger **99**

Appendix E **100**

E P4 Pipeline for Network Slicing with only Strict Priority Queuing and P4 Counters **100**

Appendix F **101**

F P4 Pipeline for Network Slicing with Meters **101**

Appendix G **102**

G Network Slicing at the N6 Interface **102**

List of Figures

1 QoS Model for ITU/ETSI and IETF approaches. As seen in ([1], Fig. 1). . . . 4
2 General layout of smart grid communication between SCADA, RTU and IED. As seen in ([2], Fig. 2-2). . . . 6
3 IEC 104 protocol's reach in a smart grid environment. As seen in ([3], Fig. 1). . . . 9
4 Fixed and variable length APDU format for the IEC 104 protocol. As seen in ([4], Fig. 4). . . . 10
5 I-format APCI frame type. As seen in ([4], Fig. 5). . . . 11
6 S-format APCI frame type. As seen in ([4], Fig. 5). . . . 11
7 U-format APCI frame type. As seen in ([4], Fig. 5). . . . 12
8 ASDU frame format for the IEC 104 protocol. As seen in ([4], Fig. 8). 13
9 Traditional fixed-function network processor vs P4-based programmable network processor. Adapted from ([5], Fig. 1). . . . 19
10 P4 programming workflow, which consists of compilation and loading a P4 program into a target processor. As seen in [6] and [7]. . . . 20
11 P4 V1 Model architecture. As seen in [6]). . . . 23
12 Token buckets P and C. As seen in ([8], Fig 7-4). . . . 29
13 Typical BMv2 workflow for the simple_switch target, which runs myprog.p4 program. As seen in [6]). . . . 30
14 Typical standalone 5G cellular network which incorporates 5G RAN and 5G core. As seen in [9]). . . . 31
15 Elements in the 5G core system architecture and their interfaces as outlined by the 3GPP Release 15. As seen in [10]). . . . 32
16 E2E network slicing as outlined by the IETF. As seen in [11]). . . . 35
17 5G testbed with Free5GC and UERANSIM. . . . 36
18 The 5G network testbed implemented with APU box and TLSense mini PC. . . . 38
19 P4-based network slicer and DSCP tagger integrated with the experimental 5G testbed. . . . 40
20 Overview of the 5G testbed integrated with P4-based network slicing and DSCP tagging. Enlarged image can be viewed in Appendix D. . 41
21 Parsing logic for DSCP tagger. . . . 43
22 Ingress Processing logic for DSCP tagger. . . . 45
23 P4-based DSCP tagger processing the incoming IEC 104 packet to modify the IPv4 DSCP value in the packet accordingly. . . . 46
24 Modified DSCP field by the P4-based DSCP tagger for the ASDU header TypeID 45. DSCP Hex value 0x12 corresponds to DSCP value of 18. As captured in Wireshark. . . . 46
25 Modified DSCP field by the P4-based DSCP tagger for the ASDU header TypeID 45. DSCP Hex value 0x28 corresponds to DSCP value of 40. As captured in Wireshark. . . . 47
26 Parsing stage logic for the network slicer. . . . 48

27 Ingress stage logic for the network slicer. 49
28 Egress stage logic for the network slicer. 50
29 P4 pipeline for network slicer with only strict priority queuing and counters. Enlarged figure can be viewed in Appendix E. 53
30 P4 pipeline for network slicer with P4 meters, strict priority queuing and counters. Enlarged figure can be viewed in Appendix F. 53
31 Visualisation of the entire 5G testbed with virtual network slicing at the N6 interface. Enlarged figure can be viewed at Appendix G. . . . 54
32 Bar chart comparing the E2E delay for slices #1 and #2. 57
33 Bar chart comparing the TCP bandwidth delay for slices #1 and #2. 58
34 Combo chart displaying E2E delay and TCP bandwidth delay for slice #1. 58
35 Combo chart displaying E2E delay and TCP bandwidth delay for slice #2. 59
36 P4 counter for slice #1 displaying number of packets and bytes transmitted through the slice. 60
37 P4 counters for normal slices #1 and #2, as well as the DSCP slice #6. The slices are displaying the number of packets and bytes transmitted through the slices in the beginning of the scenario #3. 60
38 P4 counter for slice #1 displaying number of packets and bytes transmitted through the slices after the IEC 104 traffic were transmitted for scenario #3. 61
39 Packets captured at RTU interface with Wireshark displaying the tagged DSCP packets that corresponds to the packet count for DSCP slice #6 61
40 E2E TCP bandwidth between SCADA and the RTU without P4 switches and in network slicing (slice #1) scenarios. 63
41 E2E delay between SCADA and the RTU without P4 switches and in network slicing (slice #1) scenarios. 64
42 RTT delay comparison between SCADA-RTU connection and the UE2-RemoteHost 2 connection. 65
43 Approximate RTT delay for the SCADA-RTU connection with P4 switches in the path. 66
44 Approximate RTT delay for the Remote Host2-UE2 connection with P4 switch in the path. 66
45 Iperf test performed from a workstation connected to the internet through a P4 switch. 67
46 Iperf test performed from a workstation connected to the internet with a normal consumer-grade network switch. 68
47 Network latency requirements for the above network technologies for smart grid applications. As seen in [12]. 68
48 Network bitrate requirements for the above smart grid applications. As seen in [12]. 69
49 P4 DSCP tagger deployed in Kubernetes pod along with Nginx controller for TLS termination. 69

50 NetFPGA SUME NIC [13]. 70
D1 5G testbed with slicer 99
E1 P4 pipeline for network slicer with strict priority queuing and counters. 100
F1 P4 pipeline for network slicer 101
G1 Network slicing at the N6 interface of the 5G infrastructure. 102

List of Tables

1 EPA for IEC 60870-5 protocol stack. As seen in ([4], Tab. 1). 8
2 Protocol stack for the IEC 104 Protocol. As seen in ([4], Fig. 3). . . . 10
3 ASDU TypeIDs and their groups for the IEC 104 protocol. As seen in ([4], App. C.1). 14
4 Systems used for implementing the 5G testbed. 37
5 Systems used for implementing the network slicing and DSCP tagging utilizing P4 based BMv2 software switch. 39
6 End-to-End delay (ms) and TCP bandwidth (Mbps) for slices #1 and #2. 56

Listings

1 Header declarations for Ethernet and IPv4 protocols. As seen in [14]. 24
2 Parser block example for parsing incoming packets with Ethernet and IPv4 headers. As seen in [6] and [14]. 25
3 Checksum verification block declaration in a P4 program. As seen in [14]. 25
4 Ingress processing in a P4 program. As seen in [14]. 26
5 An empty egress processing block in a P4 program. As seen in [14]. . 27
6 An example checksum computation for IPv4 header. As seen in [14]. . 27
7 Deparser stage declared for Ethernet and IPv4 headers. As seen in [14]. 28
8 Header definitions for Ethernet; IPv4; TCP; APCI and ASDU. 41
9 Activating strict priority queuing in BMv2 switch for the V1Model architecture. 72
10 P4 program for DSCP tagging. 79
11 P4 program for Network Slicing. 85
12 P4 program for Network Slicing at the RAN segment. 91

Symbols and Abbreviations

Abbreviations

3GPP	3rd Generation Partnership Project
4G	Fourth Generation Mobile Network
5G	Fifth Generation Mobile Network
AF	Application Function
AMF	Access and Mobility Function
APCI	Application Protocol Control Information
APDU	Application Protocol Data Unit
API	Application Programming Interface
ARP	Address Resolution Protocol
ASDU	Application Service Data Unit
ASIC	Application-Specific Integrated Circuit
ATM	Asynchronous Transfer Mode
AUSF	Authentication Server Function
BMv2	Behavioral Model Version 2
CBS	Committed Burst Size
CIR	Committed Information Rate
CLI	Command Line Interface
CPU	Central Processing Unit
DevOps	Developer Operations
DN	Data Network
DPDK	Data Plane Development Kit
DRM	Digital Rights Management
DSCP	Differentiated Services Code Point
DSO	Distribution System Operator
DTLS	Datagram Transport Layer Security
E2E	End-to-End
EPA	Enhanced Performance Architecture
FGPA	Field Programmable Gate Array
Gbps	Gigabits per second
GOOSE	Generic ObjectOriented Substation Events
GTP	GPRS Tunnelling Protocol
HMI	Human-Machine Interface
IEC 104	IEC 60870-5-104
IEC	International Electrotechnical Commission
IETF	Internet Engineering Task Force
IED	Intelligent Electronic Device
IOA	Information Object Address
IoT	Internet of Things
IP	Internet Protocol
IPSec	Internet Protocol Security
IPv4	Internet Protocol Version 4
kbps	Kilobits per second
KPI	Key Performance Indicators
LAN	Local Area Network

MANO	Management and Orchestration
Mbps	Megabits per second
MNO	Mobile Network Operator
MPLS	Multi-Protocol Label Switching
ms	Millisecond
MU	Metering Unit
NEF	Network Exposure Function
NFV	Network Function Virtualization
NIC	Network Interface Card
NRF	Network Function Repository Function
NSSF	Network Slicing Selector Function
ONOS	Open Networking Operating System
OS	Operating System
OSI	Open Systems Interconnection
P4	Programming Protocol-Independent Packet Processors
PBS	Peak Burst Size
PC	Personal Computer
PCF	Policy Control Function
PCIe	PCI Express
PDU	Packet Data Unit
PIR	Peak Information Rate
PLC	Programmable Logic Controller
PSA	Portable Switch Architecture
QoS	Quality of Service
RAM	Random Access Memory
RAN	Radio Access Network
RPC	Remote Procedure Call
RTT	Round Trip Time
RTU	Remote Terminal Unit
SCADA	Supervisory Control and Data Acquisition
SDN	Software Defined Networking
SMF	Session Management Function
SSH	Secure Shell
SSTP	Secure Socket Tunneling Protocol
SV	Sample Values
TCP/IP	Transmission Control Protocol/Internet Protocol
TLS	Transport Layer Security
trTCM	Two Rate Three Color Marker
TypeID	Type Identification
UDM	Unified Data Management
UDP	User Datagram Protocol
UE	User Equipment
UPF	User Plane Function
VHDL	VHSIC Hardware Description Language
VNF	Virtualized Network Function
VM	Virtual Machine
VPN	Virtual Private Network
WAN	Wide Area Network

1 Introduction

The advent of Fifth Generation (5G) of mobile networks have contributed to novel use case scenarios that have not been addressed with prior generations of mobile networks. Newer and advanced use case scenarios in 5G have paved the way for new types of technologies and applications to emerge [15]. The smart grid is considered to be a part of a group of technologies that have been simultaneously developed along with new and emerging communication methods, standards, and protocols. When compared with traditional power grids, smart grid improves the efficiency and quality of power distribution; increases capacity and effectiveness of power networks; automates upkeep and processes while also increasing choices for consumers and enabling new goods, services, and markets [16].

A substation in a smart grid generally houses Remote Terminal Units (RTUs), Intelligent Electronic Devices (IEDs) and circuit breakers, in which interruptions in power systems are circumvented through the detection of faults by IEDs and the subsequent activation of circuit breakers to protect the electricity distribution network. Grid protection activities are automated using Supervisory Control and Data Acquisition (SCADA) systems hosted in control stations [17]. The SCADA systems are responsible for providing monitoring and controlling signals to substations for the operation of the RTUs. However, smart grids using Ethernet communication are susceptible to general Ethernet attacks along with other types of vulnerabilities exposed through its insecure nature of system architecture and software designs [18].

The International Electrotechnical Commission (IEC) 60870-5-104 protocol, also known as the IEC 104 protocol, formulates the communication between control or master stations and substations. The IEC 104 protocol utilizes Transmission Control/Internet Protocol (TCP/IP) to send relevant telecontrol messages between the SCADA and the RTU [4]. However, the messaging is performed in plain-text and is vulnerable to snooping and other attacks.

Also, the lack of authentication and encryption mechanisms in the application layer of the IEC 104 protocol, increases chances for unauthorized access to systems and cyber-attacks. Smart grid operators and the device manufacturers are mainly concerned with immediate service continuity, and therefore security in smart grid networks have become an afterthought, which has exposed exploitable security vulnerabilities in its architecture [19]. The design of IEC 104 is largely driven by harsh delay constraints, which are typical of, for example, differential line protection where detection, communication and the operation of circuit breakers must stay below 100 milliseconds (ms). Since the circuit breakers are electromechanical devices and must deal with high voltages, they take up most of the delay budget. Hence, quite low and predictable delay is expected of the IEC 104 communication.

The lack of encryption in IEC 104 communications has not been necessarily resolved with the adoption of 5G technology. A work in progress Internet Engineering

Task Force (IETF) draft explores the data plane optimization of the interface between the Data Network (DN) and the 5G core. The DN is considered to be a remote network, and therefore is not considered to be proximal to the user devices or User Equipment (UE). Due to this, there is a need to bring about new solutions for controlling, configuring, and upkeeping paths to the DN from the UE [20]. Furthermore, enabling data plane programmability has provided new approaches to dynamic and flexible solutions for computer networking. By combining the concept of data plane programmability and optimization of the network segment between the DN and the 5G core, this thesis proposes an isolating slicing solution with respect to securing the smart grid communications.

1.1 Research Problem and Objectives

The main research problem is to ponder whether Programming Protocol-Independent Packet Processors (P4) can provide efficiency and flexibility to create an isolating slicing solution for smart grid IEC 104 communications over the DN, which is enabled by connecting 5G equipped RTUs or IEDs. Therefore, the goal of this thesis is to develop a network slicing mechanism, which isolates smart grid communications between the RTU and the SCADA into network slices. The network slicing is implemented on the segment between the 5G core and the DN. The physical segment is virtually divided into several isolated virtual slices by accessing the programmable data plane using the P4 (Programming Protocol-Independent Packet Processors) language framework. In addition to securing the network slices from each other, the slice accommodating the smart grid traffic is tailored to satisfy the Quality of Service (QoS) requirement in order to mitigate the damage a possible malware attack on the RTU-SCADA connection could cause to the whole connection.

The following sub-objectives for this thesis are considered with respect to the research problem and the main goal:

1. Choice of experimentation platform for P4.
2. The choice of fields in the IEC 104 packets, which are used for slice classification.
3. The methodology for implementing a QoS mechanism that would be supported by P4 switches.
4. Functional verification of the solution.

1.2 Thesis Scope

This thesis is mainly focused on dataplane design, development and experimentation for the slicing implementation. Compared to the traditional End-to-End (E2E) network slicing solutions in 5G networks, this thesis's approach of network slicing is considered to be contained on the segment between the 5G Core and the DN. Therefore, the slicing is deemed to be in DN rather than the 5G network itself.

The concept is to use the modern programmable data plane to add a level of isolation between the different smart grid devices and functions. There are at least two optional approaches that could be pursued. Firstly, the IEC 104 traffic can be classified and controlled per device. This would require lots of policy and configuration information per device but the difficulty of not having reliable identities would easily defeat the controls if an attacker has managed to infiltrate either malicious hardware or malware into the smart grid. Therefore, second option of parsing the IEC 104 messages and creating a possibility of applying controls based on the type of smart grid traffic was chosen. The approach assumes the ability to assign controls to each type of traffic with a small amount of configuration information leading to a solution that would be easy to manage. Moreover, due to the programmable P4 data plane, there is the possibility of upgrading the precise controls by remote commands.

Optimization and performance tuning of the P4-based slicing mechanism is considered out of scope for this thesis mainly due to time constraints. In addition, the control plane implementation for the management of the network slices is not the primary focus. Also, some compromise is expected in terms of integration of the slicing solution with a real 5G network, mainly due to the limited amount of 5G equipment to test with. Also, there is uncertainty on whether a P4-based hardware would be compatible with the real world 5G equipment or not.

1.3 Structure of the Thesis

The thesis is structured into six chapters. Chapter 2 discusses the necessary background information related to QoS, the smart grid architecture and the IEC 60870-5-104 smart grid communications protocol. Also, the programming concepts of P4 language framework are explored in detail along with the architectural model of the P4 software switch and P4-based extern objects. Furthermore, the concept of 5G networking and core architecture is discussed at length. The chapter ends with the description of network slicing in 5G networks. Chapter 3 outlines the design and implementations of the 5G network, P4-based network slicing solution as well as the P4-based tagging solution for packet's QoS assignment. Chapter 4 describes the results obtained from performance measurements of the network slices and verifies the slicing in detail. Chapter 5 discusses the performance aspects of the P4-based software switch, and examines the results with the networking requirements for smart grids. The chapter ends with the discussion of the technical challenges that were encountered during the thesis execution process. The thesis is then concluded in Chapter 6, in which the objectives and the tasks are assessed. The chapter is then finished with the contemplation of future work which can be considered for improving the slicing solution as well as the entire P4 software development process.

2 Background

This chapter provides thorough explanations regarding the relevant topics for this thesis. Firstly, the concept of QoS and isolation techniques used in today's computer networks are discussed. Then smart grid architecture is examined in detail. In addition, the telecontrol messaging in smart grid communications with the IEC 104 protocol is studied. Furthermore, the concept of data plane programming with the P4 programming language framework is explained. This is followed by the discussion of the P4 architecture for P4-based targets, and the consecutive discussion regarding the P4 extern objects such as P4 meters and counters. Also, the operation of the P4-based software switch used for this thesis is examined in depth. The chapter is then concluded with the review of the concept of 5G mobile core networking architecture with respect to its implementation for this thesis with the open source project Free5GC.

2.1 Quality of Service and Isolation Techniques in IP Network

In computer networking, isolation of Quality of Service is one of the biggest challenges. QoS is a paradigm that gauges whether a service provided by a service provider satisfies the stated and implied needs to the user. However, the term itself is very broad and applies to many different aspects of the service.

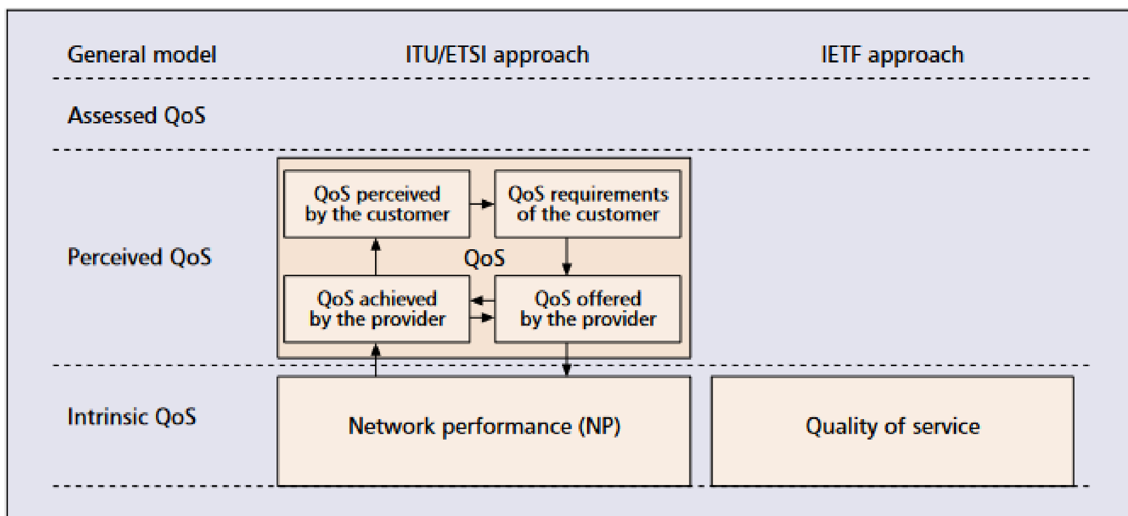

Figure 1: QoS Model for ITU/ETSI and IETF approaches. As seen in ([1], Fig. 1).

As seen in Figure 1, QoS is described by three models: assessed QoS, perceived QoS, and intrinsic QoS. The International Telecommunication Union (ITU) and European Telecommunications Standards Institute (ETSI) approaches QoS in a similar manner with perceived and intrinsic QoS models. They define QoS as the total effect of the performance of the service which controls the level of satisfaction of a user of the said service. Whereas, IETF perceives QoS with intrinsic QoS model,

in which it is a list of service conditions that must be fulfilled by a network while transmitting a flow [1].

One of the prominent, well defined and standardized architectures for implementing QoS is the Diffserv model. In this architecture, QoS is applied to predefined services which are chosen by individual packets or flows. The QoS assurance is provided by aggregating network packets to experience the same QoS level. The range of the QoS spans edge-to-edge. And they are administered based on the Differential Services Code Point (DSCP) carried in the Internet Protocol (IP) header.

When considering network slicing in mobile networks, slice isolation is also an important aspect. Each slice can be described as an independent set of resources which are set up utilizing the network environment along with a set of defined functions. The grade and power of network isolation may differ based on the requirements of use-case scenarios. For example, in a particular scenario, there would be a requirement for having communication between the slices. While in the other, the requirement would be strict isolation. Therefore, network slice isolation can be described in several methods and may consist of set of features which are selected based on implementation requisites [21]. Isolation of slices can be realized in the following areas:

- Traffic Isolation: Even though all the slices utilize the same network resources, there is guarantee that data flow in one slice does not go into another.
- Processing Isolation: Processing of network packets in a network slice is independent of all the other network slices. However, all the virtual slices utilize the same physical resources.
- Bandwidth Isolation: Each network slice is allocated with a specific bandwidth that is different from other network slices.
- Storage Isolation: Data pertaining to a network slice is stored independently from the other network slices.

Network slice isolation can also be realized with technologies such as Multi-Protocol Label Switching (MPLS), which enables tag-based network slice isolation. In addition, Virtual Local Area Networks (VLANs) can be used to partition computer network on the Etherney layer. Furthermore, Virtual Private Networks (VPNs) utilizing Internet Protocol Security (IPSec), Datagram Transport Layer Security (DTLS), Secure Socket Tunneling Protocol (SSTP), Secure Shell (SSH) could be used to ensure the confidentiality and validation for data transmission with network slices [21]. The network slicing implemented in this thesis mainly focuses on traffic and bandwidth isolation that is catered to the smart grid communications using IEC 104 protocol.

2.2 General Architecture of the Smart Grids

As briefly described in Chapter 1, from a control station, a SCADA system or control server centrally manages the RTUs and IEDs housed in different substations and secondary substations respectively. The IEDs and RTUs can be situated in geographically different sites, where they control actuators and monitor sensory equipment. The role of the SCADA system is to process the information gathered by the RTU, which controls the processes occurring locally in the substations. IEDs are special devices which act as a protective relay and directly interfaces with sensors and monitoring equipment. They can either transmit information directly to the SCADA or a local RTU, which polls the IEDs for collecting data and becomes an intermediary element by passing the collected information to the SCADA system. SCADA systems generally have in-built redundancy, and also are made to be fault tolerant.

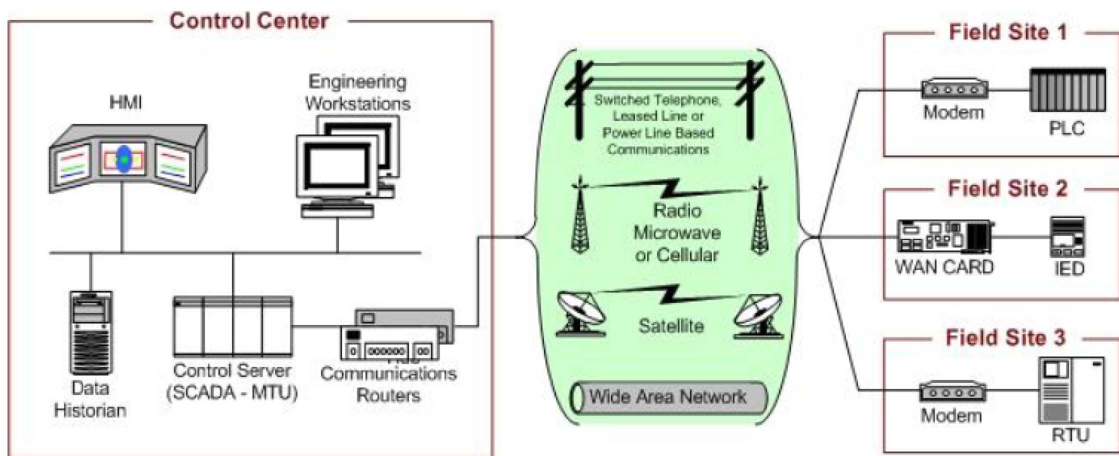

Figure 2: General layout of smart grid communication between SCADA, RTU and IED. As seen in ([2], Fig. 2-2).

Figure 2 displays the typical settings and communciation elements of the SCADA system. The control center consists of the Human-Machine Interface (HMI), workstations, storage servers, the main SCADA units as well as communication routers interfacing with the outside networks connecting the field sites. Most of the field sites are generally separated by a large geographical distance. Therefore, communication systems such as cellular and satellite networks are the norm. Few decades ago, switched telephone lines were standard for smart grid communication networks. However, due to advances in Wide Area Network (WAN) technologies, usage of telephone networks has discontinued, and newer iterations of Local Area Network (LAN) and WAN technologies have become the industrial standard for communication.

The control center is mainly responsible for monitoring and reporting. The field sites are tasked with localized control of sensors and monitoring equipment. In many cases, field sites are enabled to be remotely accessed by technicians, where they perform troubleshooting and repairs in case something goes wrong.

2.3 IEC Transmission Protocols

A technical report by Matoušek (2017) [4] provides the analysis of the IEC 104 protocol in detail. The report describes that the IEC 60870 standards for telecontrol in electrical engineering and power system automation applications for SCADA units are defined by the IEC. The part 5 of the IEC 60870 standard stipulates a transmission outline for transmitting essential telecontrol messages between the primary telecontrol station and the substations. The 60870-5 transmission protocol outlines the following parts. They are:

- IEC 60870-5-1 Transmission Frame Formats
  - It details the function of the physical and the Ethernet layers.
- IEC 60870-5-2 Link Transmission Procedures
  - Two types of service primitives and transmission procedures are defined here: unbalanced and balanced. It also outlines if messaging can be commenced only by a master station or by any other station.
- IEC 60870-5-3 General Structure of Application Data
  - It defines the overall structure of data at the application level, as well as the rules for generating application data units.
- IEC 60870-5-4 Definition and Coding of Application Information Elements
  - It establishes a standard description of the different information items used in telecontrol applications and defines the information elements. Common elements such as signed or unsigned integers, fixed or floating point numbers, bit-strings, and time elements are examples of these.
- IEC 60870-5-5 Basic Application Functions
  - Top level functionalities such as station startup, data acquisition techniques, clock synchroniziation, command transmission, totalizer counts, and file transfer are defined here.
- IEC 60870-5-6 Guidelines for conformance testing for the IEC 60870-5 companion standards

With respect to these guidelines, the IEC has developed companion standards for generic telecontrol functions with IEC 60870-5-101 transmission protocol standard. IEC 60870-5-104 transmission standard was later then released adding network access with TCP/IP. Security extensions for both IEC 60870-5-101 and IEC 60870-5-104 protocol standards were defined with IEC 62351-3. However, it is found that the security measurements implemented in this standard for IEC 104 has increased complexity and reduces backwards compatibility for certain communication protocols. Therefore, smart grid operators need to consider architectural changes and will have to update their hardware to highly performing ones adding to unwarranted operational and capital expenditures [22].

2.3.1 IEC 60870-5 protocol stack

The protocol stack for the IEC 60870-5 is established on an attenuated version of the Open Systems Interconnection (OSI) model. This is known as the Enhanced Performance Architecture (EPA). It includes the three layers of the OSI model namely: Application Layer (Layer 7), Ethernet Layer (Layer 2) and Physical Layer (Layer 1). In the following Table 1, the parts of the IEC 60870-5 corresponds to the following layers of the EPA:

<table>
<tr><th colspan="2">Enhanced Performance Architecture (EPA)</th></tr>
<tr><td>Selected application functions of IEC 60870-5-5</td><td>User process</td></tr>
<tr><td>Selected application information elements of IEC 60870-5-4</td><td rowspan="2">Application Layer (L7)</td></tr>
<tr><td>Selected application service data units of IEC 60870-5-3</td></tr>
<tr><td>Selected link transmission procedures of IEC 60870-5-2</td><td rowspan="2">Link Layer (L2)</td></tr>
<tr><td>Selected transmission frame formats of IEC 60870-5-1</td></tr>
<tr><td>Selected ITU-T recommendations</td><td>Physical Layer (L1)</td></tr>
</table>

Table 1: EPA for IEC 60870-5 protocol stack. As seen in ([4], Tab. 1).

2.3.2 IEC 104 communications

As described earlier, IEC 60870-5-101 provides a transmission outline for sending generic telecontrol signals between a centralized station and substation in a master-slave manner. IEC 104 iteration improves upon IEC 60870-5-101 by introducing network access with TCP/IP. The main smart grid station elements in IEC 104 communication are explained as follows:

- Controlling station or master station, which consists of SCADA units. This is where control of substations are carried out.
- Controlled station, slave station or substation, where devices such as RTU or IEDs are controlled by the SCADA in master stations.

The IEC 104 protocol also defines modes of directions in communication:

- **Control Direction** - Messages are transmitted from SCADA to the RTUs.
- **Monitor Direction** - Messages are transmitted from the RTUs to the SCADA.
- **Reverse Direction** - Slave stations transmit control signals to master stations or master stations transmit data to RTUs in the slave station.

2.3.3 The IEC 104 protocol

The IEC 104 protocol allows for the transmission of monitoring and controlling information over geographically widespread operations with the help of TCP/IP protocol.

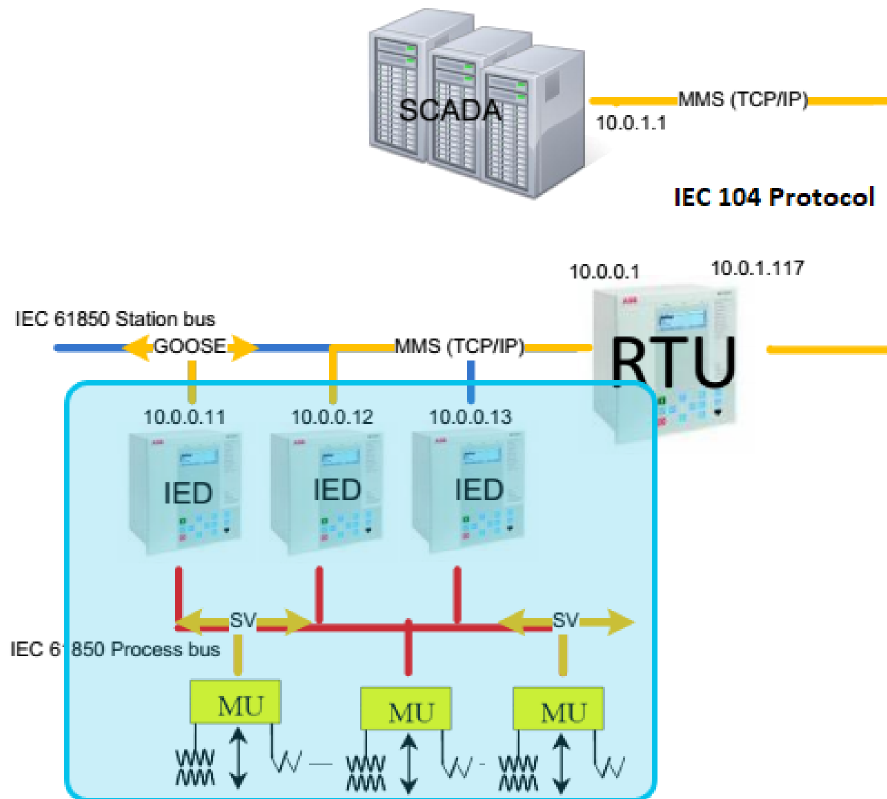

Figure 3: IEC 104 protocol's reach in a smart grid environment. As seen in ([3], Fig. 1).

The protocol's application uses equivalent station articles as the IEC 101 protocol. In the report [4], it is described that the TCP/IP suite allows multiple network types to be utilized such as Frame Relay, Ethernet, Asynchronous Transfer Mode (ATM) and serial point-to-point. Figure 3 demonstrates that the IEC 104 protocol mainly spans between the SCADA unit, the RTU and even the IEDs (yellow communication line). However, the communication between the IEDs and the monitoring elements are specified by the IEC 61850 protocol (red and blue line). The 61850 protocol specifies the standard to send data back and forth between the IEDs and the Metering Units (MU), which are basically monitoring equipment and relays or actuators. The IEC 61850 protocol standardises data transmission through Generic Object Oriented Substation Events (GOOSE) and Sampled Values (SV) network packet types. But, this protocol is considered out of scope for this thesis. Therefore, the thesis is mainly concerned with the IEC 104 protocol.

The protocol stack for IEC 104 protocol with consideration to the OSI model is displayed in Table 2. The IEC 104 protocol allows transmission of data between the SCADA and the RTU. This data is housed in Application Service Data Unit (ASDU), which is situated in the Application Layer from the OSI model perspective. ASDU combined with Application Protocol Control Information (APCI) constitutes

<table>
<tr><th>Selected Application Functions</th><th>User Process</th></tr>
<tr><td>Selection of Application Service Data Units (ASDU) of IEC 104</td><td rowspan="2">Application Layer (L7)</td></tr>
<tr><td>Application Protocol Control Information (APCI)</td></tr>
<tr><td rowspan="4">Selection of TCP/IP Protocol Suite (RFC 2200)</td><td>Transport Layer (L4)</td></tr>
<tr><td>Network Layer (L3)</td></tr>
<tr><td>Ethernet Layer (L2)</td></tr>
<tr><td>Physical Layer (L1)</td></tr>
</table>

Table 2: Protocol stack for the IEC 104 Protocol. As seen in ([4], Fig. 3).

the Application Protocol Data Unit (APDU). The APDU can either contain only the APCI or an APCI with ASDU.

2.3.4 The APCI frame

The APCI frame begins with a start byte of the hex value 68, which is followed by four 8 bit control fields. This constitutes the APDU with fixed length. As for APDU with variable length, it consists of the general APCI field format comprising of the four 8 bit control fields followed by the ASDU. The APCI field constitutes 6 bytes entirely.

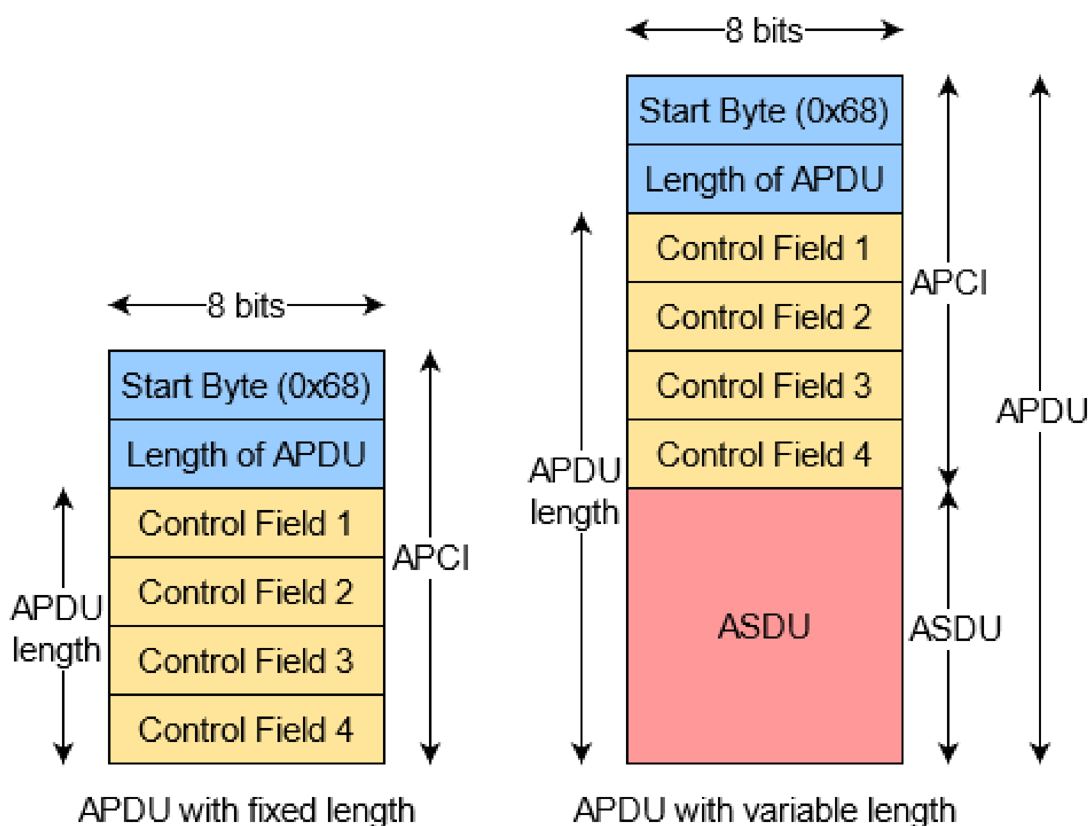

Figure 4: Fixed and variable length APDU format for the IEC 104 protocol. As seen in ([4], Fig. 4).

The frame format of the entire APDU is determined by the control fields that are present in the APCI header. There are three APDU frame formats for the IEC 104 protocol. They are: I-format, S-format and the U-format.

- **I-format**
  Between the controlling station and the controlled station, the I-format is

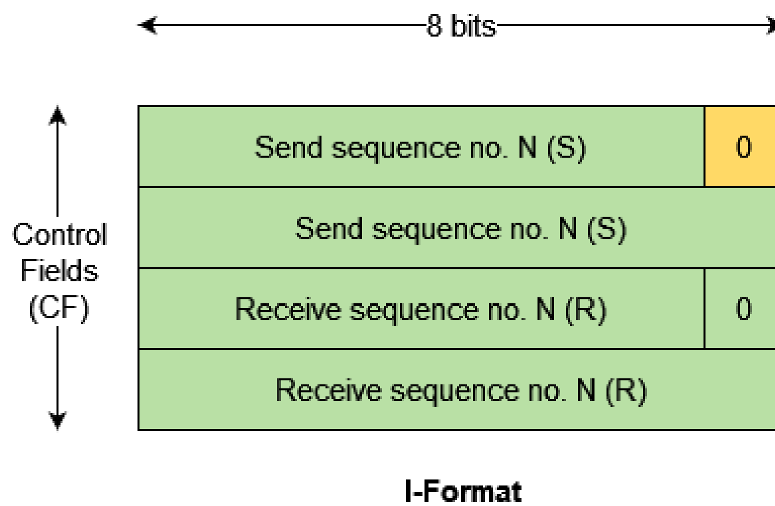

Figure 5: I-format APCI frame type. As seen in ([4], Fig. 5).

utilized to convey numerical information. It is variable in length and it always guarantees an ASDU header within the APDU frame. Furthermore, its control fields determine the communication direction. The last bit of the control field 1 is set to 0.

- **S-format**

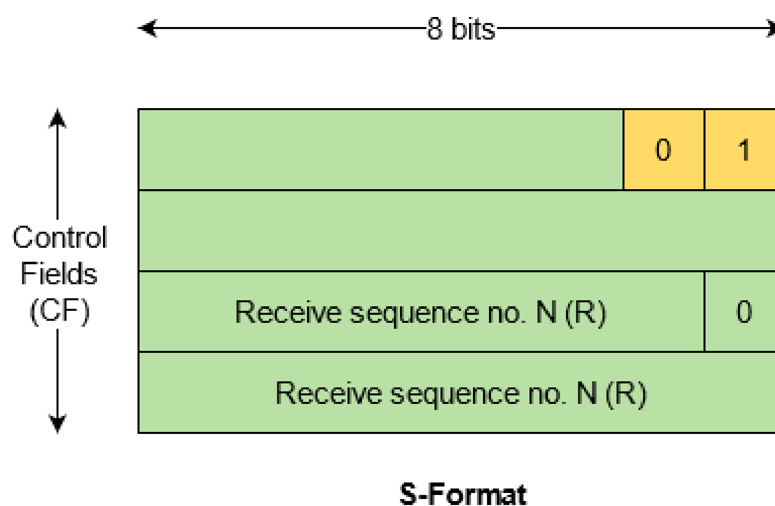

Figure 6: S-format APCI frame type. As seen in ([4], Fig. 5).

The S-format is of fixed length. Numbered supervision functions are carried out using this frame. The data transfer with S-format APDU is unidirectional. It is transmitted during buffer overflows, timeouts and during instances where the communication grid has passed the maximum numerical threshold of sending I-format APDUs without sending or receiving acknowledgements. The final bits of the control field 1 are set to 01.

- **U-format**

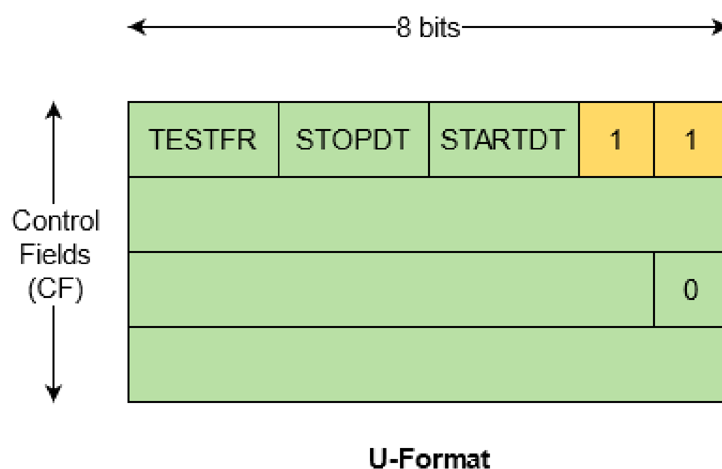

Figure 7: U-format APCI frame type. As seen in ([4], Fig. 5).

The U-format of the APDU is utilized to execute unnumbered control function. This frame too has a fixed length. It comprises of only APCI. Its function, test frame (TESTFR) is activated along with the stop data transfer (STOPDT) or start data transfer (STARTDT) simultaneously. TESTFR function is utilized to verify the established connections on a regular basis in order to discover any issues in communication. The last 2 bits of the control frame 1 are always set to 11.

2.3.5 Format of the ASDU frame

The ASDU frame is divided into two main parts. The first part is the data unit identifier, which is fixed in length of 6 bytes. The second part is the data, which can be made of at least one information object. The particular category of data is specified by the data unit identifier. It also offers addressing to identify the data's unique identity, as well as extra information such as the reason of transmission [4]. At most, an ASDU can transmit a maximum of 127 objects. The frame format of the ASDU is as shown in Figure 8.

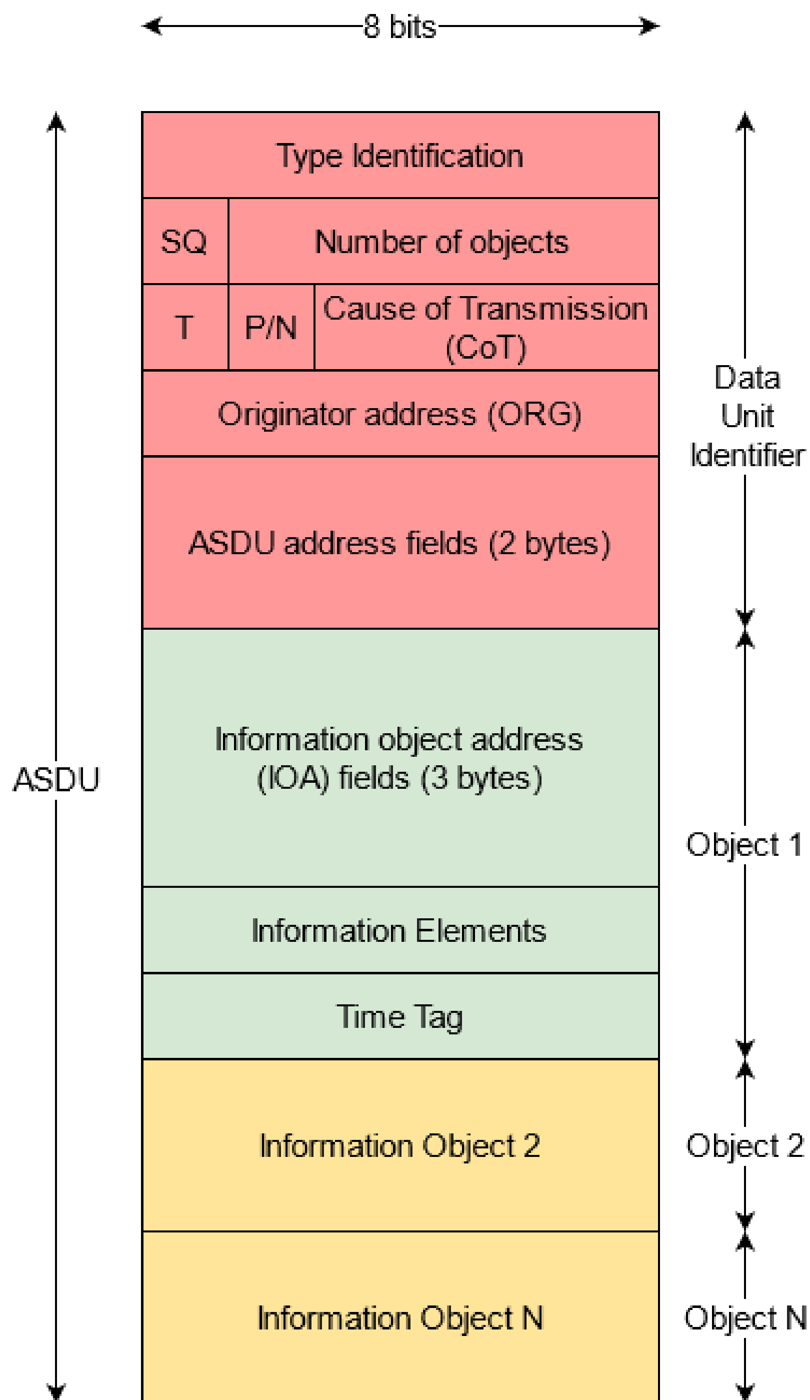

Figure 8: ASDU frame format for the IEC 104 protocol. As seen in ([4], Fig. 8).

The following are the important fields of the ASDU frame:

1. **Type Identification (TypeID) - 1 byte**

   The TypeID determines the type of information carried by the ASDU frame. It also determines whether the ASDU frames are being transmitted in the monitor or control direction.

   The following table outlines the TypeID groups and the type of data they represent in the IEC 104 ASDU frames:

TypeID	Group
1-40	Process information in monitoring direction
45-51	Process information in control direction
58-64	Command telegrams in control direction
70	System information in monitor direction
100-107	System information in control direction
110-113	Parameter in control direction
120-127	File transfer

Table 3: ASDU TypeIDs and their groups for the IEC 104 protocol. As seen in ([4], App. C.1).

2. **Structure Qualifier (SQ) - 1 bit**

   This field determines the way data elements are addressed. When the bit value is 0, it denotes that there are consecutive data elements or information objects embedded in the ASDU frame. When the bit value is 1, then the ASDU frame only consists of a single information element per frame.

3. **Number of objects/elements - 7 bits**

   This field basically denotes the amount of information objects contained by the ASDU frame. The value ranges from 0 to 127.

4. **Test (T) - 1 bit**

   This field determines whether the ASDU frame is generated during test conditions or not. Value 0 denotes that there is no test. Value 1 denotes that the ASDU frame is a test frame.

5. **Positive/Negative (P/N) - 1 bit**

   This field in the ASDU frame specifies the verification of positive or negative activation demanded by the primary application function. Value 0 states that the verification is positive and value 1 verifies that the response is negative. It is generally established to 0 when it is not appropriate.

6. **Cause of Transmission (CoT) - 6 bits**

   This field in the ASDU frame is used to route messages on a communication network and within a station by directing them to the right program or task for execution. The ASDUs in the control direction have been validated as application services, and they may be replicated in the monitor direction, albeit with different transmission mechanisms. Typical values are between 1 and 47. Values between 48-63 are used for special cases.

7. **Originator Address (ORG) - 1 byte**

   This field is used by the controlling or master station for explicitly informing the slave station about its identity. This is particularly useful when multiple master stations are involved. This allows the slave station to selectively send information back to the master station where the control ASDU frames originated from.

8. **ASDU Address Field - 2 bytes**

   This field basically consist of a normal station address. But it can be framed to be a sector address where multiple logical units can be represented by breaking a singular station. This ASDU address field can also be used for global address. A global address is used as a broadcast address where the messages are broadcasted to all the stations situated in a specific sector. The global address is generally utilized when a specific operation must be started at the same time. For example, ASDU with TypeID 103 is sent to all the logical units simultaneously for synchronizing clocks to a conventional time.

9. **Information Objects - N bytes**

   The data utilized by the smart grid communication is embedded in the information objects fields within the ASDU frame. The information object is identified by the Information Object Address (IOA), which is 3 bytes in length. The IOA is utilized to identify which data belongs to which specific station. It is also used to acknowledge the control and monitor directions of smart grid communication with the help of destination and source addresses respectively.

10. **Information Elements**

    Information elements fields in the ASDU header are the building components that are utilized to transfer data. The standard specifies the format and length of each information element. Furthermore, it specifies how the values included in the fields are decoded once they have been encoded.

The most relevant ASDU header field for this thesis is the ASDU TypeID. The TypeID is used to determine the DSCP value, that would be added by the P4 switch when IEC 104 protocol based packet traverses through the switch from the SCADA.

2.4 P4 (Programming Protocol-Independent Packet Processors)

Data plane programming as a concept initiated when Software Defined Networking (SDN) reinforced the separation of the network into control plane and forwarding or data plane. Over the years since the advent of SDN, vendor agnostic control plane applications such as OpenFlow gained huge traction. It enabled network operators to control multiple data plane devices with a single control plane. But due to development of new and upcoming networking headers, protocols, services and applications, it became increasingly difficult for network operators to wait for specific updates and patches for their physical and virtualized networking hardware.

Due to this aspect, researchers from Barefoot Networks, Intel, Stanford University and other institutions introduced a networking paradigm that would allow flexible configuration of the data plane. This flexibility allows a network programmer to program their networking hardware in whatever manner they see fit, for unique networking applications instead of waiting for the network vendors and manufacturers to release new features and zero-day security updates [23].

The authors in the paper [23], introduced the design of P4 programming language for data plane programming with the following main objectives:

- **Reconfigurability**
- **Protocol Independence**
- **Target Independence**

As explained in [23], the reconfigurability aspect empowered by the P4 programming enables re-defining of packet parsing and processing. The protocol independence prevents the networking hardware from being locked to a particular format for the networking packets. Rather, there is flexibility in defining the packet parsing as well as implementing match-action tables to process network packets. Finally, target independence skips the need for a network programmer to know the underlying processor architecture of the networking hardware. In lieu of this, a programming compiler converts a "taget-independent" P4-based programmatic description into a "target-dependant" one that is understood by the underlying processor of the networking hardware.

The advantages of data plane programming are numerous. A presentation [6] created by the members of the P4 consortium outlines the following advantages of general data plane programming.

Data plane programming:

1. Enables support of new, upcoming, and custom protocols,

2. Enables elimination of obsolete protocols, which minimizes complexity,
3. Allows for Developer Operations (DevOps) style of development, which enables quick bug fixes, rapid innovation and faster programming design cycle,
4. Facilitates systematic use of resources, in which usage of tables are scales as required.
5. Provides increased visibility of the network and its devices by allowing new methods for network telemetry and diagnosis.

The specifications of the P4 programming were first outlined in August 2014. At that time, the programming specification called the language framework as $P4_{14}$. Later in 2016, a new iteration of the P4 programming language framework was released. It is known as $P4_{16}$, which is named after the release year. In this thesis project, the slicing implementation as well as the DSCP tagger were implemented with the latest $P4_{16}$ framework. From this point onward, whenever P4 language framework is mentioned, $P4_{16}$ is indicated instead of the older network programming framework $P4_{14}$.

Before delving deep into the topic of data plane programming of P4, some relevant definitions must be presented with. As specified in [7], they are:

1. **Architecture**: It consists of a group of P4-programmable elements and defines the way the data plane communicates with them.
2. **Control Plane**: It consists of a category of programs which are used to supply the data plane with the related input and output information.
3. **Data Plane**: The data plane consists of a group of programs which explains how the packets are processed and modified (forwarding logic).
4. **Metadata**: It is described as the transitional information which is constructed during the P4 program runtime.
5. **Packet**: It is also known as network packet. which is a structured element of data which is transmitted within a computer network.
6. **Packet Header**: It is the structured information at the forefront of the packet. A single packet may consists of one or several packet headers depicting various computer network protocols.
7. **Packet Payload**: It consists of the raw data after the packet headers are defined.
8. **Packet-Processing System**: It consists of data handling methods for handling packets. It brings together both the control plane and data plane programs to form the network plane.

9. **Target**: It is the packet-processing system that is able to recognize the functions and constructs defined in the P4, and run them to execute network functions for implementing the forwarding logic in the data plane.

2.4.1 Traditional VS programmable switches

In the specification document [7] for $P4_{16}$, a traditional switch is compared with P4 programmable switch. While most network switches utilizes both the control plane and the data plane, P4 allows modifying the data plane characteristic of the programmable network unit. P4 programs can be used to determine the link or the interface by which the control plane can interact with the data plane. However P4 programming cannot be used like OpenFlow for control plane management. Generally, the control plane is used to control the P4 programmable data plane by administering table entries, handling network protocol packets, setting up unique objects (P4 counters or meters), and processing unsynchornized events such as link-state notifications.

With respect to a traditional network switch, P4-based programmable switch is distinguished in the following ways:

- The P4 program specifies the functionalities of the data plane. It is configured when the P4 programmable switch is started. Before the initialization, the programmable switch does not have prior knowledge of the prevailing network protocols.
- Interaction with the data plane by the control plane is performed by utilizing the same way (channels) as the fixed-function network switches. However, the network tables and data plane objects can be specified in the P4 program in different ways as well as any number of times unlike the fixed function switches. The P4 compiler will create the Application Programming Interface (API) that will allow the control plane to interact with the P4 defined tables, objects and program functions in the data plane.

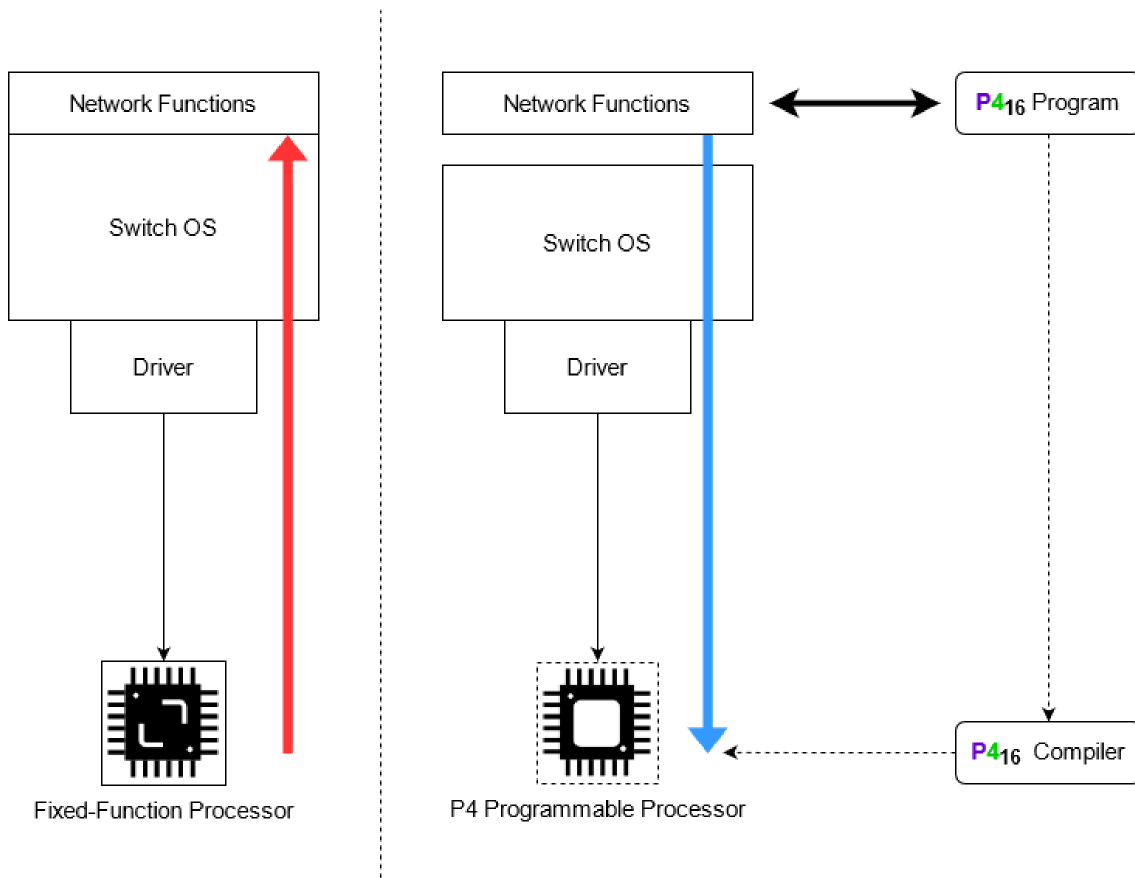

Figure 9: Traditional fixed-function network processor vs P4-based programmable network processor. Adapted from ([5], Fig. 1).

As depicted in the Figure 9, the network functions that can be used by the network administrators for their networks are pre-defined by the manufacturers of the fixed-function network devices. The network functions can be accessed through the device's Operating System (OS) with the utilization of the device drivers, which is a gateway to the processor chip that executes these network function programs. The network functions are mostly fixed and they would not allow the network administrators to modify the network functions as required. Modification of the network functions usually require firmware or OS upgrade deployments and processor redesign. This would add operational and capital expenses for both the manufacturer and the enterprise consumers.

Whereas, this is not the case for the devices with programmable processors. The programmable processor allows the network programmer to define the network functions with P4. The P4 code can be compiled into a set of programs with the help of the P4 compiler, which the programmable processor can recognize to execute them. Unlike the network functions in fixed-function processors, network functions in programmable processors can be modified as required without the intervention of the processor manufacturer or network hardware vendor. The processor firmware could be updated as required to add more feature-sets to the P4 language

compiler rather than redesigning or reprogramming the entire processor itself. This enables more flexibility and adaptability for the network programmer to modify their computer networks in an elastic and scalable manner. It also reduces expenditure by minimizing hardware obsolescence as well as mitigating long wait times for the network programmers to deploy upgrades or plug security vulnerabilities and bugs in the network functions without delay and extra involvement of the network processor manufacturers, hardware vendors, and software providers.

2.4.2 Programming the data plane with P4

In relation to other data plane programming frameworks, P4 is considered "protocol independent" which allows network programmers to define various different sets of protocols that are standard and customized as well as implement diverse data plane operations [7]. However, in order to ensure that the network hardware such as a switch is P4 programmable, the manufacturer of the network processor must design the compiler appropriate for specific target processor. The target processor can be a Field Programmable Gate Array (FPGA), Application-Specific Integrated Circuit (ASIC), BMv2 and other physical or virtualized network hardware that are enabled to support P4 programming with the utilization of P4-based compilers.

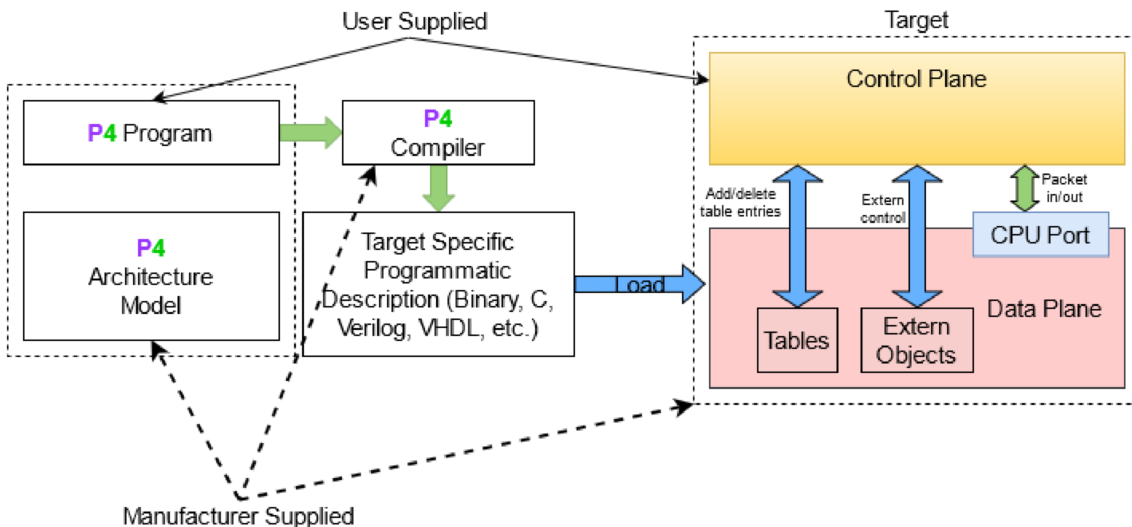

Figure 10: P4 programming workflow, which consists of compilation and loading a P4 program into a target processor. As seen in [6] and [7].

As visualized in the Figure 10, the compilation of P4 program with the P4 compiler can generate the following [7]:

- Structure of the data plane, which is applied by the forwarding logic outlined in the input P4 program.
- An API for the control plane, which can be used to administer objects and configurations in the data plane, that are defined in the P4 program.

Apart from the general advantages of data plane programming, there are also other merits to data plane programming with P4 as described in [7]:

- **Elasticity**: Compared to traditional networking hardware, P4 can help design new methods for packet forwarding which are describable as program functions. Traditional hardware only discloses fixed-functions for network administrators to work with.
- **Expressiveness**: Multipurpose processes and table retrieval can be utilized with P4 to articulate complex and hardware independent network packet handling algorithms. These programs can be used between multitude of different hardware targets that implement the same architecture.
- **Resource Mapping and Management**: The P4 compiler automatically plots custom fields defined by the user to vacant hardware resources and handles resource and time allotment in an abstract manner.
- **Software Engineering**: P4 programming enables multiple usage of the software as well as type-checking and concealment of information within the program.
- **Component Libraries**: Manufacturers provide component libraries to encapsulate manufacturer defined functions into transferable high-level P4 constructs.
- **Decoupling hardware and software evolution**: Target or processor manufacturers can define new abstract architectures, allowing them to additionally separate high-level computation from low-level design.
- **Debugging**: Software archetypes of the model provisioned by the manufacturer can help in the P4 program development and debugging.

P4 programming is quite similar to programming with the C language, albeit without the for and while loop functions. However, there are some fundamental concepts for defining forwarding logic in the data plane with the P4 language framework. The fundamental concepts as defined by the P4 specification [7] are:

1. **Header Types**: This abstraction is used to define the structure of the network packets.
2. **Parsers**: It is used to define the order of the headers and header fields within the ingress packets. It also describes the means of recognizing the packet as well as the information contained in the headers and fields that are to be derived from the network packets.
3. **Tables**: This is utilized to link operations with the keys defined by the network programmer. The tables are used to apply access control lists, flow tables, forwarding information base, routing tables as well as custom-built tables for invoking multifaceted networking decisions.

4. **Actions**: Actions in code block form, are used to define the manner of modification of packet header fields and metadata. It can also incorporate information provided by the control plane while the network processing is taking place.

5. **Match-Action Unit**: It is generally utilized to execute a specific order of processes. Firstly, the lookup keys are generated from the packet header fields or calculated metadata. Then, by utilizing the generated lookup keys, table lookup is carried out. During this process, an action is also selected to be carried out. Finally, at the end of the match-action process, the chosen action code blocks are processed.

6. **Control Flow**: It describes the method each packet must be handled within the P4 programmable target. It includes the order of calling match-action units. The reconstruction (deparsing) of the network packet is also executed with the help of the control flow abstraction.

7. **Extern Objects**: These are specified within the architecture that is controlled by the P4 program via clearly expressed APIs. They are not modifiable with P4, and their characteristics are fixed (packet header checksum function is one such example).

8. **User-Defined Metadata**: These are data constructs, which are linked to each ingress or egress network packets. They are defined by the user.

9. **Intrinsic Metadata**: It is the metadata that is supplied by the model linked with each ingress or egress packet. For example, the port at which the packet is incoming is considered as an intrinsic metadata.

2.5 P4 Architectural Model and the BMv2 Switch

The P4 architecture is an important aspect for programming with the P4 language framework. It defines the relevant P4-programmable blocks in the appropriate order:

- Parser
- Ingress Control Flow
- Egress Control Flow
- Deparser

The P4 architecture binds the P4 program with the target that must be programmed. The P4 architecture is outlined in the P4 compiler by the target manufacturer. The P4 architecture determines the way P4 targets behave when processing a network packet. The architectural model uses machine language to map appropriate bit values to intrinsic metadata. An example is, during initialization of a P4 counter with a relevant identification value, that value is stored in a control register by the P4 architecture. However, the manner the intrinsic metadata is stored and understood, is determined by which P4 architecture is implemented by the manufacturer.

Different P4 architectures cannot support the same P4 program. However, any P4 targets that support the specific P4 architecture model can support the P4 program written for the model. The P4 architectural model used for this thesis is known as the V1model.

2.5.1 The V1model

The V1model implements the following P4 programming blocks for applying its architecture:

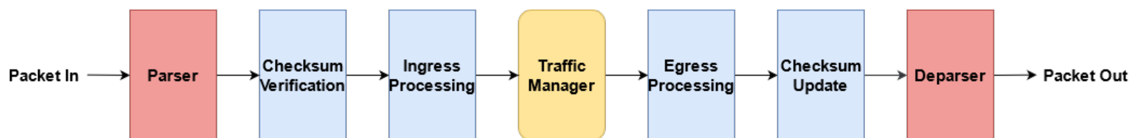

Figure 11: P4 V1 Model architecture. As seen in [6]).

As observed in the Figure 11, the P4 target that runs the P4 program, performs the processing of the network packet by starting with the parsing. Once the packet is parsed, the necessary checksum header fields in the packet are checked for validation. If the checksum verification fails, the packet is dropped by default. After this stage, the ingress processing begins. When the ingress processing finishes, the packet is pushed into the traffic manager, in which the packet is kept in queue if necessary. After the traffic management, right before the packets are pushed through the output port of the network hardware, the packet undergoes egress processing, which is very similar to ingress processing stage. After the egress processing, the checksum fields

in the packets are updated. If this is not done, the consecutive network device will reject the packets since the checksums are not in order due to the packet processing in the programmable P4 target. The packets are then restructured back to their correct forms or restructured with modified or with additional data (such as tunnel entries) and pushed out of the respective network port as specified in the P4 program.

Before the P4 programmatic blocks are implemented, the packet header fields that must be extracted, must be declared. For example, if the network programmer wants to extract information contained in the fields in an IPv4 header of an incoming packet, then the P4 program must declare the following header definitions:

```
header ethernet_t {
    bit<48> dstAddr;
    bit<48> srcAddr;
    bit<16> etherType;
}

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    dscp;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    bit<32>   srcAddr;
    bit<32>   dstAddr;
}

struct metadata {
    /* empty */
}

struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
}
```

Listing 1: Header declarations for Ethernet and IPv4 protocols. As seen in [14].

In the above Listing 1, the header fields for Ethernet and IPv4 protocols are declared in an abstract manner. The variable name for each bit block can be named in any specific manner as per the network programmer's needs. However, the bit lengths for each header fields are declared as per the standards required for the communication protocols to operate. Otherwise, incorrect information would be parsed into the header field declarations.

The parser block in a P4 program is implemented as the following:

```
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            0x800: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }

}
```

Listing 2: Parser block example for parsing incoming packets with Ethernet and IPv4 headers. As seen in [6] and [14].

The parser is declared as seen in Listing 2. The parser block in the P4 program can have predetermined declarations, namely: Start, Accept and Reject. The Start declaration begins the parsing process. Accept declaration allows the program to move to the next parsing stage, while the Reject declaration prompts the P4 target to reject the packet out of the P4 pipeline. The parsing stage allows the network programmer to chart information in the network packets into headers and metadata declared in the P4 program. While transitioning from one state to another, the programmer is free to implement greater than or equal to zero number of programmatic statements in the parser block. The parser block contains the select statement, which is used to create branches in the parsing algorithm. This is identical to using case statements in C and Java programming. However, there is no need for break statements. Branching is required to parse the bit information in the packets. In the above Listing 2, the hex value 0x800 (line 13) establishes the structure of the consecutive headers that must be extracted from the packet.

After the parsing stage, then the checksum verification stage begins. The checksum verification is an optional P4 programming block that can be used as needed in the P4 program. In order to use it, the checksum fields in the packet headers are declared here. The checksum verification block is declared in the following Listing 3:

```
control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
```

```
    apply {  }
}
```

Listing 3: Checksum verification block declaration in a P4 program. As seen in [14].

The necessary header checksum verification declarations are specified in the apply statement in this (Listing 3) P4 block.

After the declaration of the checksum verification block, ingress processing begins. In ingress processing, new variables can be declared, along with the generation of lookup tables and initializing extern objects. The network functions are basically specified in both ingress and egress processing blocks. In addition, match-action pipelines are declared here for specifying the way the P4 programmable target must process the network packets.

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    action ipv4_forward(bit<32> dstAddr, bit<9> port) {
        .......
        .......
        .......
        /* Implement your network function here */
    }

    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
        }
    }

    apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
        }
    }
}
```

Listing 4: Ingress processing in a P4 program. As seen in [14].

As observed in Listing 4, the way the packets should be handled in the target is described here. Firstly, an action statement is defined, with the declaration of temporary variable header fields that are used in the statement for defining the network function (line 5). Subsequently, a lookup table is also defined, in which the key declaration is used to activate the network function, once the key is triggered along the entire network processing. The lookup table can be called from the control plane to insert the key that must trigger the network function. Finally, an apply statement is declared, where the lookup table is instantiated inside this statement

(line 22). After the ingress processing, the packet gets transferred through the traffic manager. The traffic manager consists of traffic queuing pipeline leading to egress processing. But the queuing is optional and be customized by needed by activating strict priority function in the P4 target.

The egress processing works in a very similar way as the ingress process, except the processing is performed at the egress stage. The declaration of action statements and lookup tables are done in the same manner. The following Listing 5 depicts the manner egress processing is declared in the P4 program:

```
 control MyEgress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    apply {  }
}

```

Listing 5: An empty egress processing block in a P4 program. As seen in [14].

After the egress processing, the packets which had their header fields modified must undergo checksum calculation again. If this is not done, then the modified packets with incorrect checksum will get dropped at the interfaces of other devices in the network. The checksum calculation block is depicted in the P4 program as below:

```
control MyComputeChecksum(inout headers  hdr, inout metadata meta)
   {
    apply {
  update_checksum(
          hdr.ipv4.isValid(),
            { hdr.ipv4.version,
            hdr.ipv4.ihl,
              hdr.ipv4.diffserv,
              hdr.ipv4.totalLen,
              hdr.ipv4.identification,
              hdr.ipv4.flags,
              hdr.ipv4.fragOffset,
              hdr.ipv4.ttl,
              hdr.ipv4.protocol,
              hdr.ipv4.srcAddr,
              hdr.ipv4.dstAddr },
            hdr.ipv4.hdrChecksum,
           HashAlgorithm.csum16);
    }
}
```

Listing 6: An example checksum computation for IPv4 header. As seen in [14].

In the above Listing 6, the checksum calculation is depicted for the IPv4 header of the network packets. The checksum is updated in case the header fields are modified in some way by the network functions declared in the P4 program. In this checksum

calculation declaration, the hash algorithm extern object provided by the V1 model is used (line 17). The calculated checksum is 16-bit in length, which is the standard length for IPv4 checksum field.

After the checksum update block in the P4 program, the deparser block is next. The deparser block is necessary in order to rebuild the packet back to its original form. With consideration of the parsing stage for Ethernet and IPv4 headers, the following code block declares deparsing stage for these headers:

```
control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
    }
}
```

Listing 7: Deparser stage declared for Ethernet and IPv4 headers. As seen in [14].

In the Listing 7 referencing the deparser block, both the Ethernet and IPv4 headers that were parsed in the beginning of the entire networking process are constructed back to their original form and then pushed through the output port. From this point onwards, the target has performed the network processing as described by the P4 program that was loaded into it.

2.5.2 P4 extern objects - meters and counters

As described earlier in Section 2.4.2, P4 extern objects are specific functions which are defined in the P4-based architecture. They are not modifiable and are hard-corded by the vendor or the manufacturer. The P4 externs are accessed through the control plane APIs. Apart from the checksum extern object, there are other extern object which are relevant for this thesis. They are: P4 meters and P4 counters.

P4 meter extern is based on the concept of a two rate three color marker (trTCM) [24]. It is defined by default in the V1Model P4-based architecture for the P4 programmable targets (such as BMv2). As per the concept of the trTCM, it marks incoming IP network packets with green, yellow or red mark. When a packet exceeds the Peak Information Rate (PIR), the trTCM marks the packet red. Generally, the packet is marked green or yellow based on whether it fails to be under or exceeds the Committed Information Rate (CIR). The trTCM is generally used for policing incoming traffic of a service, whenever the peak rate has to be imposed independently from the committed rate.

The P4 meter marks each packet and transmits the packet and the relevant metering outcome to the marker. The trTCM meter in P4 is set up by allocating values to four parameters: PIR and its relevant Peak Burst Size (PBS), and CIR and its relevant Committed Burst Size (CBS). The units measured for both the PIR and CIR are packets per second. The PIR should generally be same as CIR or

greater than it. Whereas both the PBS and CBS are measured in packet sizes or byte sizes. This can be configured in the P4 program when the P4 meter is invoked. The behavior of the P4 meter is defined in regards to two token bucket P and C. The PIR and CIR rates are also considered. The maximum size of the token bucket P corresponds to the PBS, while the token bucket C corresponds to CBS.

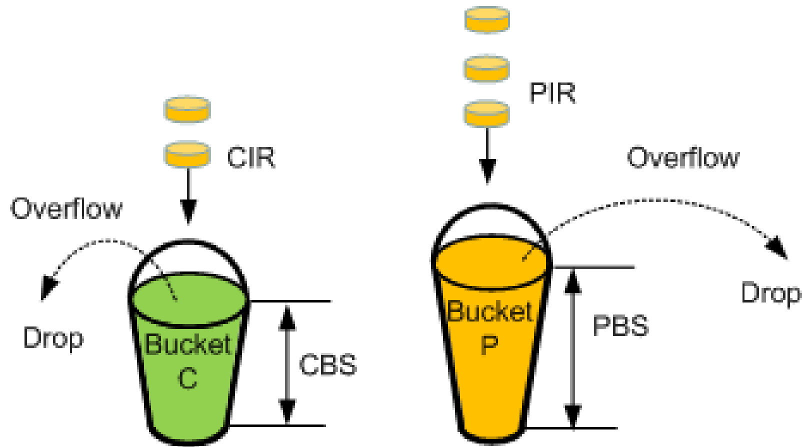

Figure 12: Token buckets P and C. As seen in ([8], Fig 7-4).

As shown in Figure 12, the tokens are placed in P and C buckets with respect to the PIR and CIR rates. When one of the bucket is filled with tokens, the consecutive tokens are dropped. However, if the other bucket is not full, the tokens are transferred to that bucket. The trTCM operates on the basis of checking whether the transmission rate is following the specifications. Hence the traffic rate is calculated based on the bucket P first, and the bucket C is taken into consideration for measurement [8].

Let Tp and Tc attribute to the number of tokens in buckets P and C correspondingly. The preliminary values for Tp and Tc are PBS and CBS respectively. For Color-Blind mode, when a packet of size B reaches at time t [8]:

- If Tc(t) $- B \geq 0$, both Tp and Tc are decreased by B and the packet is marked green.
- If Tp(t) $- B \geq 0$ but Tc(t) $-$ B $<$ 0, Tp is decreased by B, and the packet is marked yellow.
- If Tp(t) $-$ B $<$ 0, the Tp and Tc values remain unchanged, and the packet is marked red.

P4 counters are classified as extern objects that are used to keep network packet statistics. They support both packet and byte counting. Byte counts are incremented by measuring packet size. However, the implementation of this can vary based on the P4-based architecture.

2.5.3 The BMv2 software P4 switch

As per the official GitHub repository for the BMv2 switch, it is the second iteration of the software switch that supports P4 [25]. C++ was used to develop this software switch. The P4 compiler for this switch target creates a JSON file from the P4 compiler. The JSON file is used by the software switch to apply the programmatic functions in the the BMv2 target, which enables it to handle the network packets transferred through it.

The BMv2 has several model targets. However the most recommended one to be used is the simple_switch target. The simple_switch target utilizes the V1Model architecture and it supports $P4_{16}$. The control plane of the simple_switch target can be accessed with the Thrift server running in the target during runtime. The Thrift server is programmed to be accessed by a network admin with the help of APIs. This manner of interacting with the P4 BMv2's control plane, enables the P4 program to be loaded into the BMv2 target without the control plane program recompiled again for use. Hence, the Thrift API (runtime CLI) is considered program independent. This is why the BMv2 target is used for this thesis.

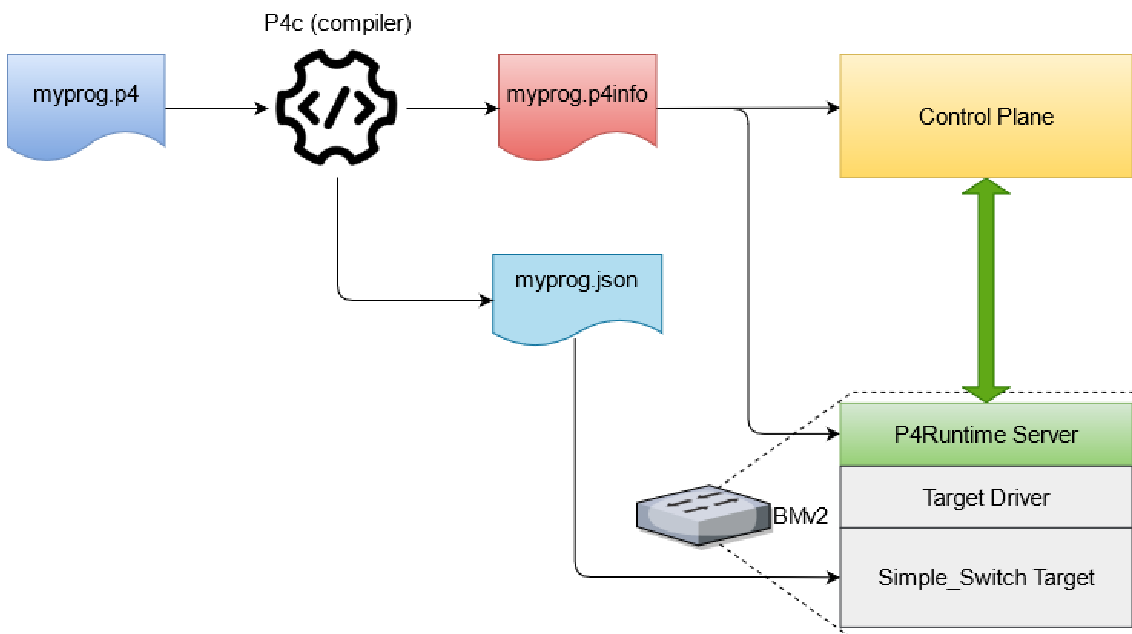

Figure 13: Typical BMv2 workflow for the simple_switch target, which runs myprog.p4 program. As seen in [6]).

The second model target is the simple_switch_grpc. It has all the features of simple_switch, but it allows interaction with the controller through TCP communications utilizing Remote Procedure Call (RPC), however this target model is not used for this thesis. The final model target is the psa_switch, which utilizes the Portable Switch Architecture (PSA) instead of the V1model. But the implementation for the

PSA is not finished yet and is work in progress.

As observed in Figure 13, the P4 compiler compiles the myprog.p4 P4 program to generate two files: myprog.p4info and myp4rog.json. The myprog.p4info file is used by the P4Runtime server, which interacts with the control plane of the BMv2 software switch. The runtime control uses the outline of the pipeline provided by myprog.p4info file. All the characteristics of the P4 programs such as actions, lookup tables, extern objects, and many others are described in this file. During the BMv2 switch runtime, the runtime controller will access the myprog.info file to understand the P4 program's structure and provides an interface for the network administrator to interact with the P4 program's functions. The format for the .P4info file is based on protocol buffers, which is an open source library developed by Google. The compiler output is always target independent as long as the target model is supported by the hardware provided by the manufacturers.

2.6 5G Core Network Architecture and Free5GC

Currently, 5G networks are being deployed due to the increasing number of applications such as Internet of Things (IoT), smart home, smart grid, industrial IoT and many others. 5G is being adopted widely, because it significantly reduces delay and energy consumption compared to 4G (Fourth Generation Mobile Networks). Furthermore, it facilitates very high throughput, faster development and deployment of services or applications; in addition to elasticity, improved security, privacy and connectivity to large number of devices amounting to billions.

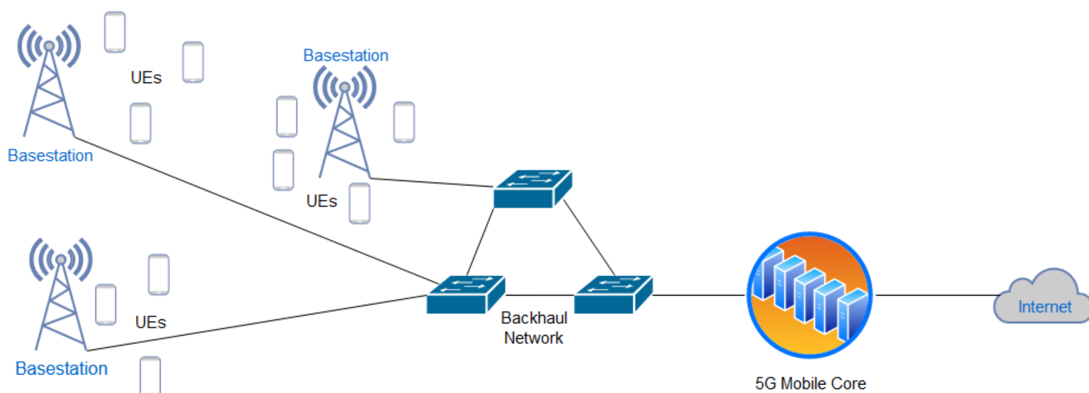

Figure 14: Typical standalone 5G cellular network which incorporates 5G RAN and 5G core. As seen in [9]).

5G is an evolving system. It is restricted by Key Performance Indicators (KPIs) and other standards. One of the main aspects of a 5G network is the 5G core. It describes the way mobile user devices are connected to each other as well as devices on the internet. The 3rd Generation Partnership Project (3GPP), has defined the outline for the 5G mobile core network to adopt a microservice based architecture.

Microservice based architecture allows to break down the implementation into different functional blocks. Apart from the core, the Radio Access Network (RAN) is also very important for the mobile cellular networks.

Current mobile networks are deployed in different ways. First way is the non-standalone method. There are two approaches to deploy a non-standalone mobile network. First approach is deploying 4G and 5G RAN with 4G's Evolved Packet Core. Second approach is deploying both 4G and 5G RAN over a 5G core. The second method of deployment is the standalone 5G, where the entire architecture is 5G, including the the RAN, which provides wireless interfaces for the UEs to connect to the mobile network [9].

As displayed in the Figure 14, the RAN, also known as the basestation or gNodeB (5G-New Radio) is connected to the 5G core through a backhaul network. The RAN ensures that the radio spectrum is effectively used by the UEs to fulfil the QoS prerequisites.

2.6.1 The 5G Core

The 5G core consists of the following elements in its system architecture:

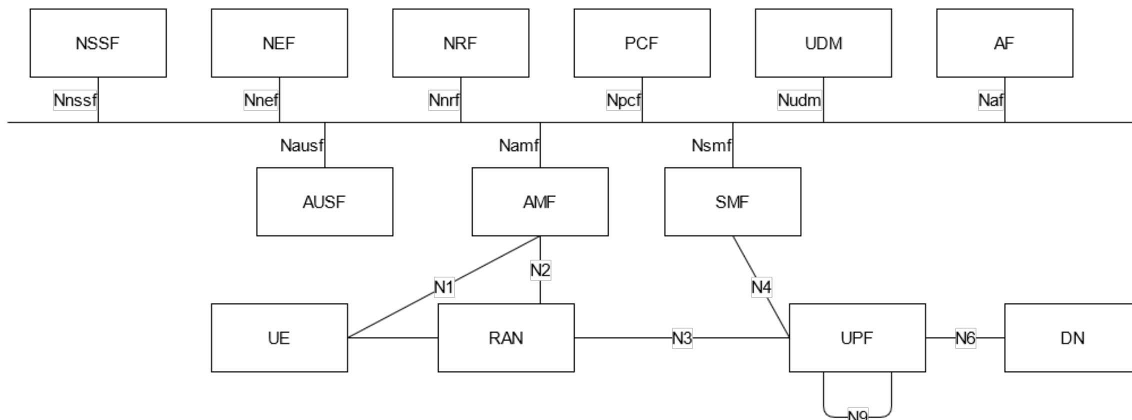

Figure 15: Elements in the 5G core system architecture and their interfaces as outlined by the 3GPP Release 15. As seen in [10]).

As seen in the above Figure 15, the following are the elements in the 5G architecture [9]:

- **Network Slicing Selector Function (NSSF)** - It is responsible for selecting a network slice for the UE.
- **Network Exposure Function (NEF)** - It is used to expose the abilities of the network functions to outside services in a secure manner.
- **Network Function Repository Function (NRF)** - It is used to expose network functions that are active. It provides a discovery service.

- **Policy Control Function (PCF)** - It provides the control to create or modify policy rules, which are imposed by other network functions.
- **Unified Data Management (UDM)** - It provides the function to manage user identity, as well as creation of authentication keys.
- **Application Function (AF)** - The AF provides access for the NEF and session linked data to the PCF for policy control. It is considered as a logical element in the 5G core architecture.
- **Authentication Server Function (AUSF)** - The AUSF acts as the authentication server.
- **Access and Mobility Function (AMF)** - Considered as the control plane function. Its important operations are reachability management, mobility management, connection management and registration management. Basically, these operations allow the UE to register or unregister to the 5G network, as well as establish control plane signalling, while ensuring the UE is always available and is located appropriately.
- **Session Management Function (SMF)** - It is responsible for assignment of IP addresses to UE, manage QoS, and also control session context with the UPF.
- **User Plane Function (UPF)** - It is the gateway between the RAN and the DN. It performs packet forwarding for the UEs to reach the devices in the DN. Furthermore, it provides network statistics and provides control over QoS.

In addition to the elements the following are the interfaces used by the network functions to operate [10]:

- **Nnssf** - It is the interface exposed by the NSSF.
- **Nnef** - It is the interface exposed by the NEF.
- **Nnrf** - It is the interface exposed by the NRF.
- **Npcf** - It is the interface exposed by the PCF.
- **Nudm** - It is the interface exposed by the UDM.
- **Naf** - It is the interface exposed by the AF.
- **Namf** - It is the interface exposed by the AMF.
- **Nsmf** - It is the interface exposed by the SMF.

Apart from the interfaces, the reference points are also depicted in the Figure 15 [10]:

- **N1** - It is the point between the AMF and the UE.
- **N2** - It is the point between the AMF and the RAN.
- **N3** - It is the point between the RAN and the UPF.
- **N4** - It is the point between the SMF and the UPF.
- **N6** - It is the point between the UPF and the DN.
- **N9** - It is the point between two UPFs. Multiple UPFs are used either for redundancy or for connecting two different DNs.

2.6.2 Free5GC

The Free5GC is the open source project for implementing the 5G core for the 5G mobile network. The project implements the 5G core network as per the requirements of the 3GPP Release 15. There are other open source projects for configuring the 5G core network. The Free5GC was chosen for this thesis based on the extensive guides available to understand its installation and configuration process. In addition, the UE and basestation simulator, UERANSIM is designed to work with Free5GC [26]. The 5G core network that is implemented for this thesis with Free5GC is the same as depicted in Figure 15. However, the NSSF features are not used and is not relevant for this thesis.

2.6.3 Network Slicing in 5G

As mentioned in the Introduction Chapter 1, network slicing allows for the isolation of a physical network into different logical networks (slices) dedicated to different services tailored to their needs, in order to provide adaptable and diverse applications. Each slice conforms to specific KPIs and standards as defined by the mobile operator. It is also an enabler for isolating one end-to-end communication from others and hence, securing them. Usually, slices can be generated on request and are given adequate isolation with independent control and management [27]. Network slicing as a concept for operating the smart grids have been examined carefully, in which new and efficient business models have been conceptualized [28].

In Figure 16, an E2E network slicing is depicted over the 5G network. It constitutes of network slicing on the RAN, transport network, core network and the network where the app servers are operating. Various slices which are portrayed in the figure, are conformed to different standards for different application. The slices on each component are managed by the same or different controllers, which could be managed by a network slice orchestrator. This can provide consistency in their management as well as in the provision of QoS, high reliability and compliance to explicit KPIs.

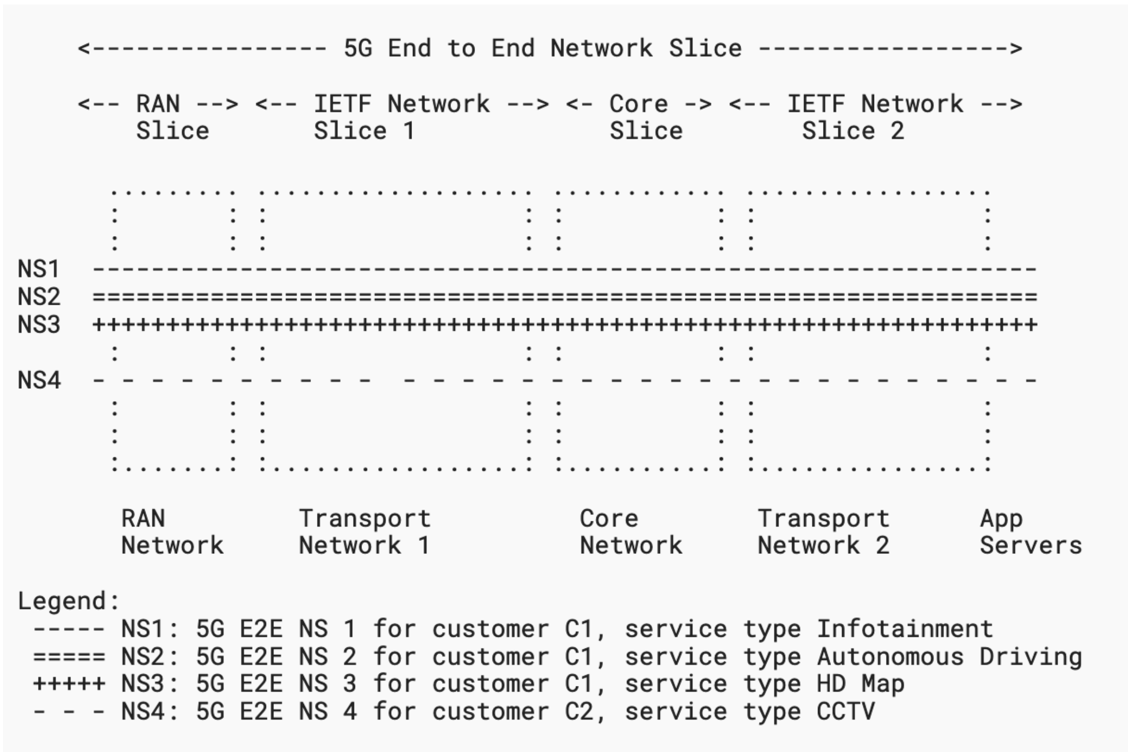

Figure 16: E2E network slicing as outlined by the IETF. As seen in [11]).

There are several technologies that enable network slicing. They are mainly hypervisors, virtual machines, containers, Software Defined Networking (SDN) and Network Function Virtualization (NFV). SDN and NFV technologies in particular have had large impact on the way today's conventional network slices are created and managed. For example, SDN allowed the separation of the control plane from the data plane where centralized network management has been consolidated in the control plane to provide vital attributes, such as elasticity, expandability, service-oriented conformability as well as sturdiness in networking systems. These features are required to maintain network slices through the SDN controllers residing in the control plane of the networks. Popular SDN solutions such as Open Networking Operating System (ONOS) and OpenDaylight are some examples, which provide a bird's eye view of the network for easier network management. Furthermore, NFVs have been also utilized heavily to implement network slicing in which, Virtualized Network Functions (VNFs) are service chained in conjunction with NFV infrastructure, and Management and Orchestration (MANO) [29]. The network slicing in this thesis is restricted to the transport network connecting the DN to the UPF of the 5G core as previously stated in the thesis research problem 1.1. There is no network slice orchestrator, instead the network slicer's (BMv2)'s runtime Command Line Interface (CLI) is used to add, delete and ban slices as required.

3 Design and Implementation

The main idea for the design of the testbed is controlling the IEC 104 traffic by isolating it from other network traffic transmitted through the 5G network. It is important to consider that less important traffic, which may also include malicious traffic should not be able to take the smart grid down without any difficulty. The system should be able to upkeep the QoS of different categories of network traffic. With respect to the IP traffic, the possibility of controlling the QoS comes with the Differential Services architecture, where different QoS parameters corresponds with different DSCP code values in the IP header field.

The first section discusses the design considerations of the experimental testbed for testing the virtual slicing framework at the segment between the 5G core and the DN. Then, the next section will discuss the system architecture along with the equipment that was used for implementing the P4-based network slicer and the P4-based DSCP tagger. Finally, the implementation of slicing and DSCP tagging with P4 will be discussed more thoroughly.

3.1 Design considerations for the 5G testbed

In the beginning of this thesis project, a testbed that includes a proprietary 5G core as well as a Nokia basestation was considered and tested. The 5G network was configured in the standalone mode. The UE consisted of a Raspberry Pi 2 device equipped with Quectel's 5G modem. However, in standalone mode, there were connection reliability issues with the 5G modem equipped on the Raspberry Pi, the assembly and setup of this testbed was dropped. Throughout the testing of the Raspberry Pi's connection to the 5G network, it could not establish the Packet Data Unit (PDU) session with the 5G core most of the time. Moreover, a sufficient number of Quectel modems for further testing and for establishing concurrent UE connections to the DN was not available.

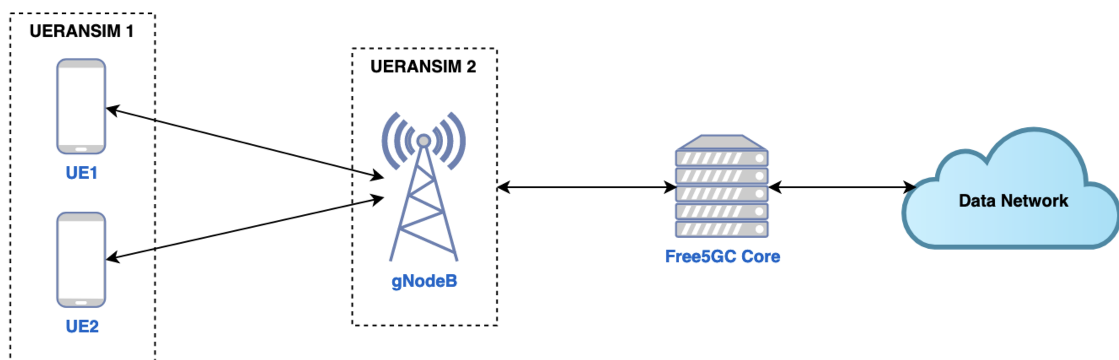

Figure 17: 5G testbed with Free5GC and UERANSIM.

Due to these complications, the plan for the 5G testbed with Cumucore 5G core

and the Nokia basestation was withdrawn. Instead, a 5G testbed with Free5GC 5G core and a UE/RAN simulator was devised. The UE and RAN simulator was obtained from the open source UERANSIM GitHub repository created by Ali Güngör [30].

In the Figure 17, the main idea is to separate the UE and RAN elements from each other rather than keep them together with UERANSIM. Therefore, in this case, three instances of UERANSIM software are running separately in separate systems. The core is also run separately in a standalone workstation, so that its performance will not be affected by the other components.

3.2 System architecture for the 5G testbed and for network slicing

The following systems were used to implement the 5G testbed, network slicing and DSCP tagging:

System	**TLSense Mini PC**	**APU Box 1 (Model apu4d4)**	**APU Box 2 (Model apu4d4)**
Purpose	Hosting 5G Core with Free5GC	RTU & UE2 as UEs with UERANSIM SCADA in Data Network	Hosting basestation with UERANSIM
System OS	Ubuntu Desktop 20.04 LTS	VMware ESXi 6.5	VMware ESXi 6.5
Guest Host OS (VMware Only)	-	2 Hosts - Ubuntu Server 18.04.5 LTS	Ubuntu Server 18.04.5 LTS
CPU Model	Intel Celeron Processor N3160 @2.24 GHz; 4 cores	64-bit AMD Embedded GX-412TC @1 GHz; 4 cores	64-bit AMD Embedded GX-412TC @1 GHz; 4 cores
Memory (RAM)	8 GB	4 GB (2 GB per host)	4 GB

Table 4: Systems used for implementing the 5G testbed.

Table 4 outlines the system used for implementing the 5G testbed. The 5G testbed consists of the Free5GC 5G core installed in the TLSense mini Personal Computer (PC). Due to limitations in obtaining additional workstations with several network interfaces, the APU boxes were used. As described in the official manufacturer website [31], APU boxes are single board computers that are used for computer networking. Even though they do not have high processing power, they have multiple network interfaces to work with. Instead of installing standalone OSes on the APU

box, VMware ESXi 6.5 hypervisor was installed. VMware ESXi is a purpose-build baremetal hypervisor, which can be installed in the baremetal server. It provides the user with a centralized management system to efficiently dividing hardware resources to applications and reduce expenses. Standalone installation of the linux based OS in the APU box is extremely cumbersome. This is why VMware hypervisor was chosen instead, which allows multiple isolated virtual machines (VMs) to be run on the same APU box, allowing easier installation of the OSes and their management.

The 5G network testbed with the DN was configured with the aforementioned systems as follows:

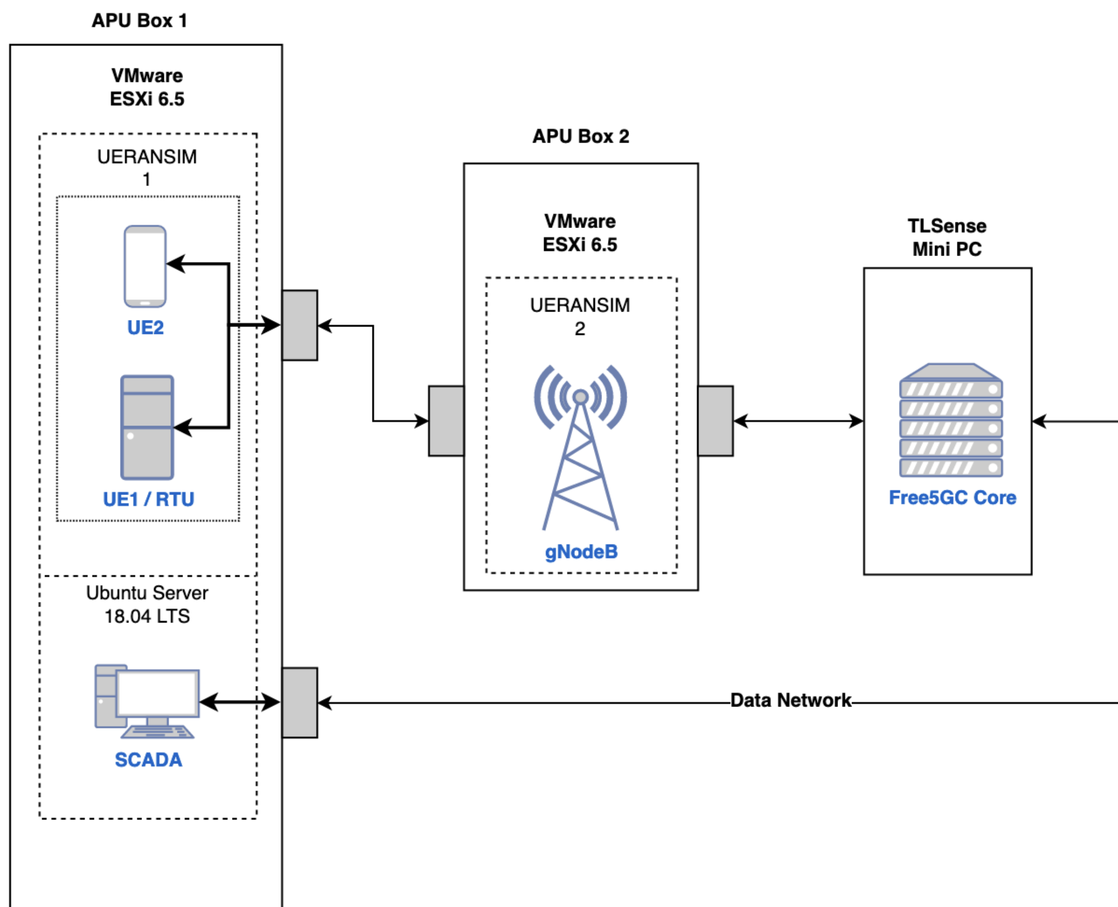

Figure 18: The 5G network testbed implemented with APU box and TLSense mini PC.

Figure 18 displays that the 5G network was configured with two APU boxes and a mini PC. Running UERANSIM did not require high amount of processing power and RAM memory. Furthermore, UERANSIM allows the user to create concurrent UE profiles to be connected with the Free5Gcore through the gNodeB basestation element without any issues. For demonstration of the slicing technique, it was determined to have two UEs: one for RTU and the other as standalone UE for slice number 2. As for the SCADA, it is a normal VM running on Ubuntu server 18.04 LTS in the

VMware hypervisor.

In order to simulate smart grid traffic that consists of sending and receiving ASDU frames, an open source python based IEC 104 client and server by GitHub user, RocyLuo [32] was used. The simulator is based on Python programming language, and utilizes socket streaming based on the TCP standard to send IEC 104 traffic between the server and the client. The simulator supports the I, S and U frames and only supports the following ASDU frames with the TypeIDs: 45, 46, 47, 48, 49, 50, 51, 58, 59, 60, 61, 62, 63, 64, 101 and 103. These TypeIDs conform to the IEC 104 traffic transmission in the controlling direction (SCADA to RTU) of the smart grid communication.

The simulator utilizes the Scapy tool, which is a Python-based tool that allows to forge and send packets. Since this simulator was published six years ago, it only supports an older version of Python programming language (Python 2.7) and Scapy (version 2.2). Therefore, both the UE1 and the RTU machines were installed with the outdated Python and Scapy libraries in order to run the IEC 104 smart grid traffic simulation.

As for the implementation of the P4-based network slicer and DSCP tagger, the following systems were used:

System	**HP EliteDesk 800 Workstation**	**APU Box 3 (Model apu3d4)**
Purpose	P4-based Network Slicer	P4-based DSCP Tagger
System OS	Ubuntu Desktop 16.04 LTS	VMware ESXi 6.5
Guest Host OS (VMware Only)	-	Ubuntu Server 16.04 LTS
CPU Model	Intel Core i5-6500 CPU @3.20 GHz; 4 cores	64-bit AMD Embedded GX-412TC @1 GHz; 4 cores
Memory (RAM)	8 GB	4 GB

Table 5: Systems used for implementing the network slicing and DSCP tagging utilizing P4 based BMv2 software switch.

As outlined in the Table 5, a standalone workstation and an APU box with VMware ESXi 6.5 hypervisor was used to implement the slicing technique as well as the DSCP tagger. Both systems utilize Ubuntu 16.04 LTS OS, since BMv2 and the P4 compiler do not support Ubuntu versions above 16.04. In addition, installation of the P4 compiler and the BMv2 takes up significant hard drive storage space due

to decompression of relevant software packages and libraries. Hence, custom Ubuntu 16.04 installation images with P4 compiler and BMv2 software switch were created beforehand for easier and modular installation.

The P4 network slicer and the DSCP tagger were integrated with the 5G network testbed as follows:

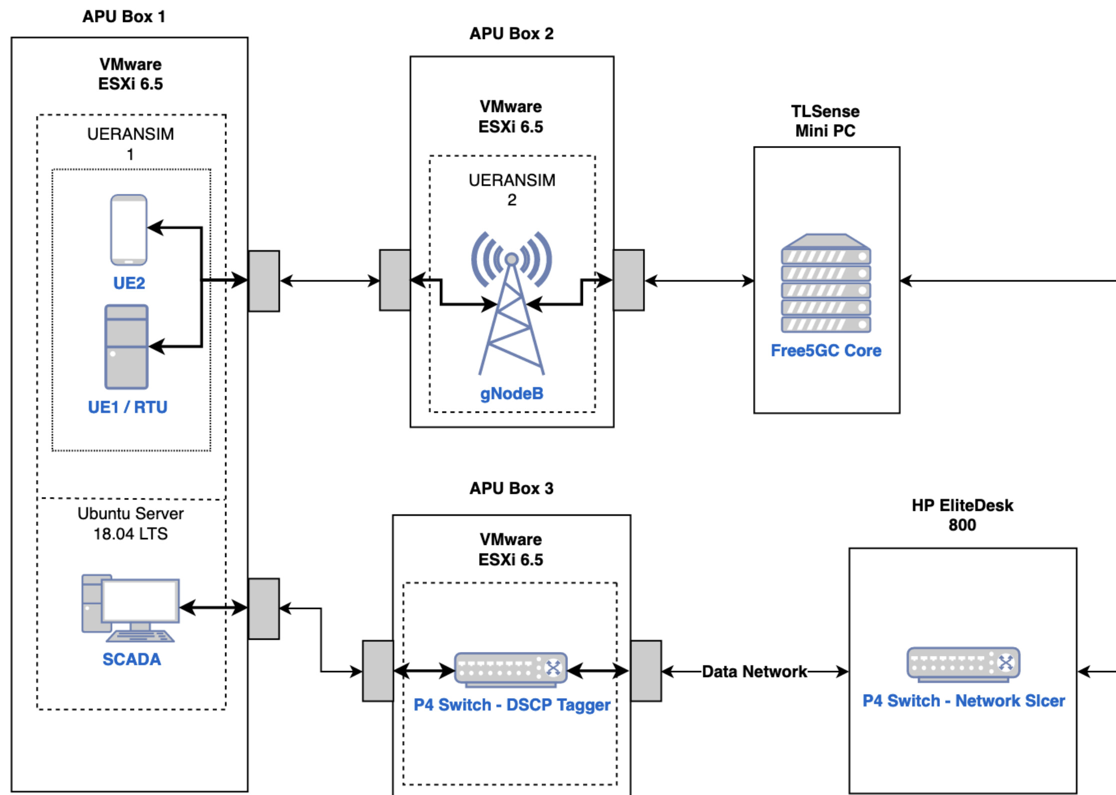

Figure 19: P4-based network slicer and DSCP tagger integrated with the experimental 5G testbed.

Figure 19 displays that, the P4 slicer was placed in proximity to the 5G core, whereas the P4 DSCP tagger is placed in proximity to the SCADA in the DN. Considering this setup, it can be surmised that the smart grid Distribution System Operator (DSO) has made an agreement with the Mobile Network Operator (MNO), which owns the 5G networking infrastructure. Based on their deal, the DSO tags their smart grid traffic with appropriate DSCP values, which the MNO will use to push the valid DSCP tagged traffic into a specified network slice through the network slicer on the DN segment. The DSCP based slice provides the appropriate isolation as well as compliance to the QoS requirements in accordance to the agreements with the DSO.

While taking the aforementioned concept into consideration, the 5G network testbed along with the slicing implementation can be mapped to a business environment as shown below:

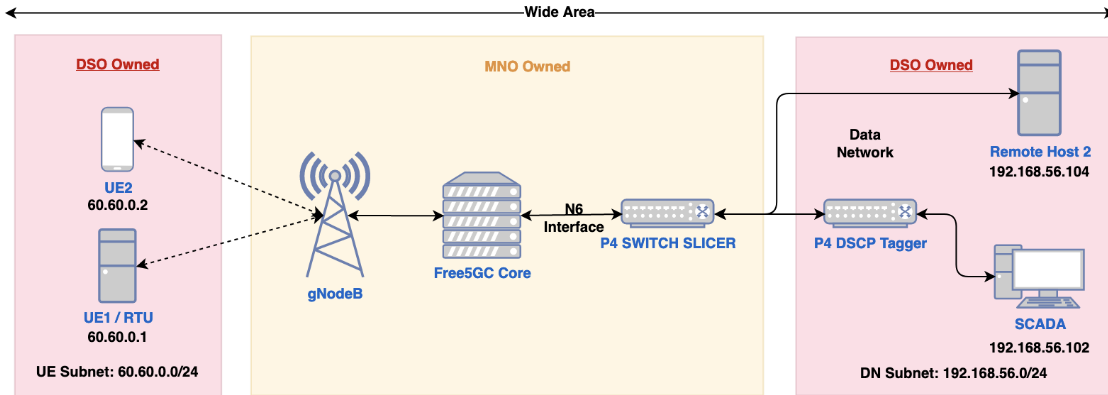

Figure 20: Overview of the 5G testbed integrated with P4-based network slicing and DSCP tagging. Enlarged image can be viewed in Appendix D.

In the above Figure 20, it can be observed that the UEs (RTU and UE2) are owned by the DSO. Whereas, the MNO owns the basestation, the 5G core and the P4-based network slicer. The P4-based DSCP tagger, the Remote Host 2, and the remote SCADA are owned by the DSO. The Remote Host 2 is a high performance Linux powered laptop for testing purposes. Therefore, this scenario is tailored for smart grid communications powered by 5G network infrastructure. In addition, the UE subnet is set to 60.60.0.0/24. This is configured in the Free5GC configuration files. Whereas the IP subnet for the DN is set to 192.168.56.0/24.

3.3 Implementation of P4-based DSCP tagger

In order to establish network slices based on DSCP values, the smart grid traffic must first be tagged with relevant DSCP values before the network slicer can utilize them. If both endpoints are owned by the DSO, DSCP tagging could be implemented directly in the applications. This would however spread the issues of QoS and slicing into each and every application. By using a P4-based tagger, we can separate the concerns of QoS and traffic isolation from the applications and create a more manageable overall architecture. Therefore, the implementation of the P4-based DSCP tagger is discussed first. The DSCP tagger was first compiled using a P4 compiler, which compiles a P4 program into a JSON file. The JSON file is loaded into the BMv2 switch installed in APU box 3.

Firstly, in the P4 program, all the important headers are defined for parsing as follows:

```
1
2 header ethernet_t {
3     macAddr_t dstAddr;
```

```
4     macAddr_t srcAddr;
5     bit<16>   etherType;
6 }
7
8 header ipv4_t {
9     bit<4>    version;
10    bit<4>    ihl;
11    bit<8>    diffserv;
12    bit<16>   totalLen;
13    bit<16>   identification;
14    bit<3>    flags;
15    bit<13>   fragOffset;
16    bit<8>    ttl;
17    bit<8>    protocol;
18    bit<16>   hdrChecksum;
19    ip4Addr_t srcAddr;
20    ip4Addr_t dstAddr;
21 }
22
23 header tcp_t {
24    bit<16> srcPort;
25    bit<16> dstPort;
26    bit<32> seqNo;
27    bit<32> ackNo;
28    bit<16> tcp_flags;
29    bit<16> wndw_size;
30    bit<16> tcp_checksum;
31    bit<16> urgentPtr;
32    bit<8>  tcp_option_kind;
33    bit<8>  tcp_option_kind2;
34    bit<8>  tcp_option_kind3;
35    bit<8>  tcp_option_len;
36    bit<32> tcp_option_tval;
37    bit<32> tcp_option_tsecr;
38 }
39
40 header apci_t {
41    bit<8> StartByte;
42    bit<8> apdu_len;
43    bit<8> type_h;
44    bit<16> rx;
45    bit<8> tx;
46 }
47
48 header asdu_t {
49    bit<8> TypeId;
50    bit<8> sq;
51    bit<8> numix;
52    bit<8> cot;
53    bit<8> nega;
54    bit<8> test;
55    bit<8> oa;
56    bit<16> addr;
57 }
```

```
58
59 struct metadata {
60     /* empty */
61 }
62
63 struct headers {
64     ethernet_t   ethernet;
65     ipv4_t       ipv4;
66     tcp_t        tcp;
67     apci_t       apci;
68     asdu_t       asdu;
69
70 }
```

Listing 8: Header definitions for Ethernet; IPv4; TCP; APCI and ASDU.

After the header definition, the parsing stage is executed. The entire P4 program for DSCP tagging can be viewed in Appendix A. However, the following logic graph for the parser is presented:

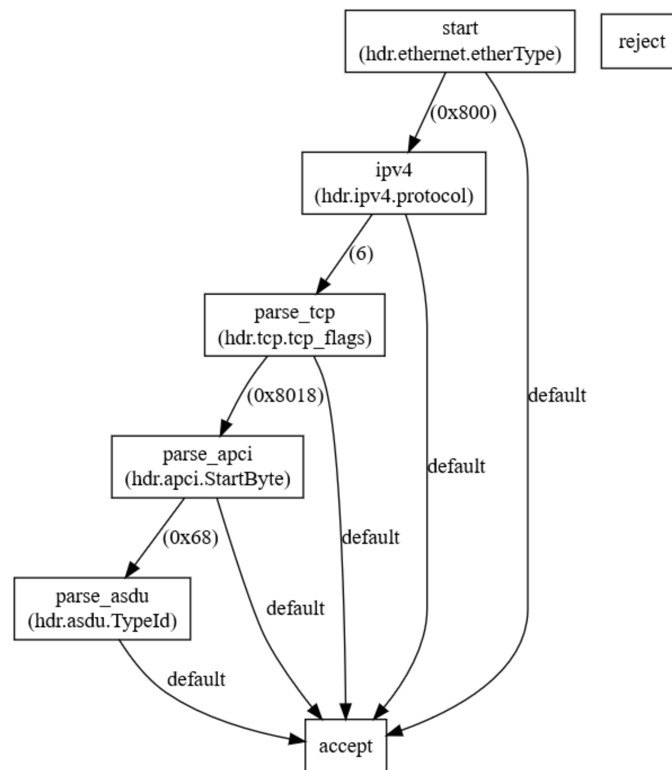

Figure 21: Parsing logic for DSCP tagger.

As seen in the above Figure 21, the parser logic in the BMv2 switch initiates at the Ethernet header. Once the required value is found in the etherType field (0x800) of the Ethernet header, the parsing stage continues to IPv4 then to TCP, from TCP to APCI header of the IEC 104 protocol and finally the ASDU header. The parsing occurs as specified in the P4 program and the required information is extracted as outlined in the header definitions of the P4 program. By default, the parsing stages

are accepted and the packets are not dropped. However, P4 allows the parsing logic to be programmed in such a way that, certain parsing stages can be rejected by the P4 switch. But, there was no need to implement this, since the P4 hardware acts as a network switch forwarding different types of network packets, whether IEC 104 protocol is present or not.

After the parsing stage, the BMv2 switch moves onto the checksum verification. However, checksum verification was not applied for DSCP tagging. And therefore, the ingress processing takes place.

In the Figure 22, the ingress processing starts with the the validation of the IPv4 header. When it is comfirmed to be true, the lookup of the values entered in the desp_dest table is performed. While the P4 program is running on the BMv2 switch, the values for the table are entered by the network admin through the runtime CLI. The values entered in the table are the IP addresses of the devices involved in the smart grid communication. In this case, they are the IP address of the RTU on the UE side of the mobile network.

Once the table lookup hits the entered key values, the TypeID field in the ASDU header is checked. The process of TypeID checkup is divided into two. Smart grid traffic corresponding to the TypeIDs between 45 and 51, as well as 101 and 103 are deemed to contain system or process information. Therefore, they are configured with a DSCP value of 18 (as seen in Figure 23). Whereas, ASDU packets with TypeIDs between 58 and 64 have the DSCP field configured to be 40, which is considered to be the appropriate values for sending signalling based traffic. The tagging is done by set_dscp_mon() and set_dscp_cont() functions for tagging DSCP values of 18 and 40 respectively. After the tagging is performed, the ingress processing logic ultimately ends with switch_forward() function, which enables the BMv2 acting as a network switch forwarding any packets from one interface to another. Throughout the packet processing, there is no rejection of packets. Any incoming packets, which are not relevant to the smart grid application are normally switched to the output port and will have their DSCP value set to 0, which is the default value. There are no processes defined for the egress processing, and therefore the deparsing is performed for the Ethernet, IPv4, TCP, APCI and ASDU headers.

The P4 program for DSCP tagging is compiled and loaded into the BMv2 switch as a JSON file in the APU box 3. During the P4 program's runtime, the program elements outlined in the program can be interacted with the runtime CLI provided by the simple_switch. The runtime CLI can be invoked by:

```
1   #Invoke the runtime CLI initialized on thrift port 9090
2   simple_switch_CLI --thrift-port 9090
```

After the runtime CLI has been invoked, the following commands are entered for activating the network processes defined by the P4 program in the network switch:

```
1   #Program the network hardware to switch packets from ingress
    port to egress port
```

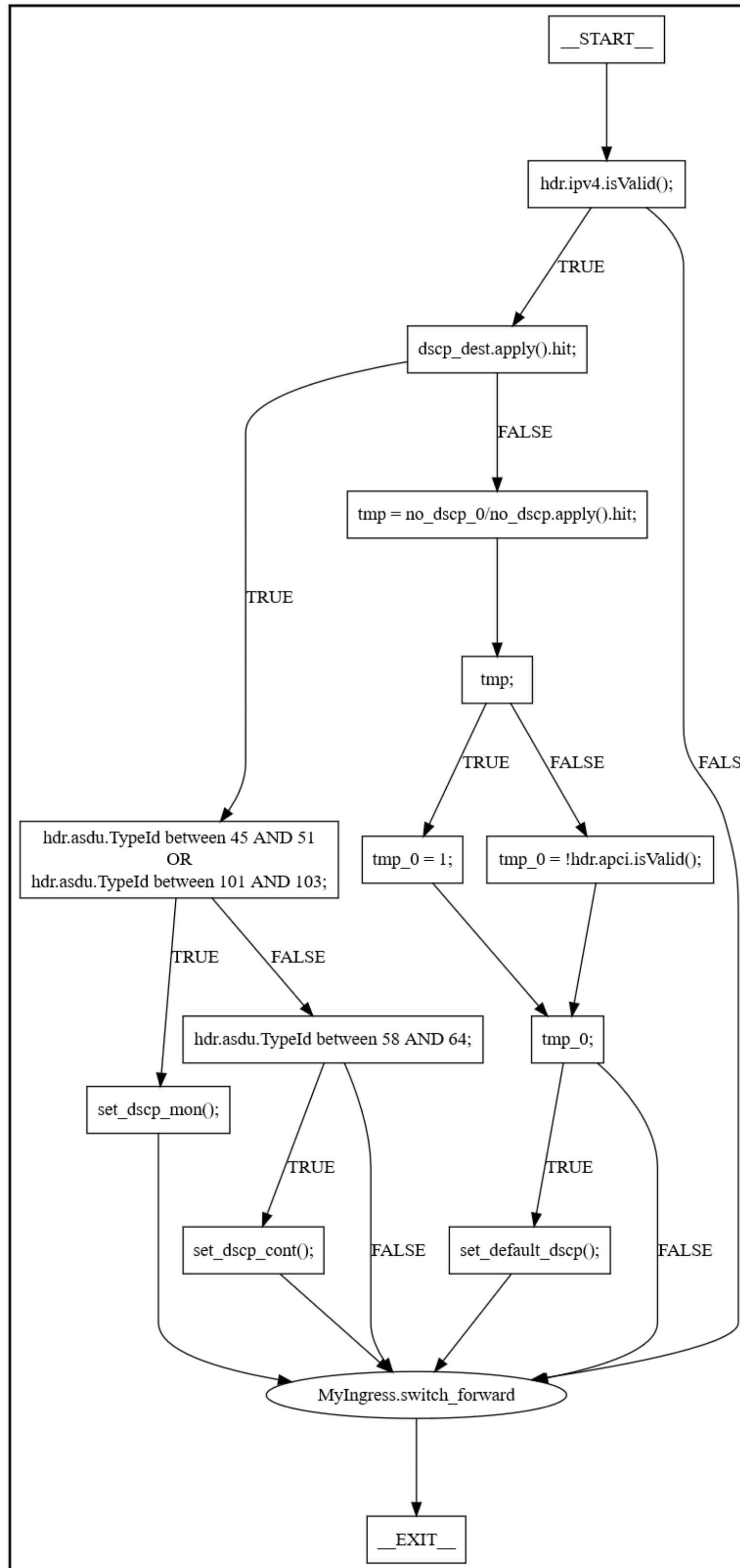

Figure 22: Ingress Processing logic for DSCP tagger.

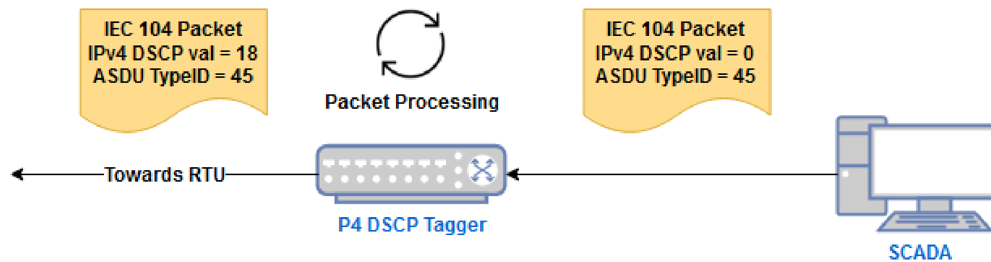

Figure 23: P4-based DSCP tagger processing the incoming IEC 104 packet to modify the IPv4 DSCP value in the packet accordingly.

```
2     table_add switch_forward 1 => 2
3     table_add switch_forward 2 => 1
4
5     #Tag DSCP values to packets based on TypeID
6     table_add dscp_dest NoAction 60.60.0.1 =>
```

The above commands enables the network admin to invoke lookup tables and enter key values into them. In the case of switch_forward table, the key value is the ingress port and the action value is egress port. Therefore, the switch is tasked to transfer packets from the ingress port to egress port and vice-versa. As for activating the DSCP tagging, the IP address of the RTU (UE 1) is provided as the destination address.

```
> Frame 184: 82 bytes on wire (656 bits), 82 bytes captured (656 bits) on interface 0
> Ethernet II, Src: Vmware_be:79:a1 (00:0c:29:be:79:a1), Dst: SuperMic_6a:6c:c0 (3c:ec:ef:6a:6c:c0)
v Internet Protocol Version 4, Src: 192.168.56.102, Dst: 60.60.0.1
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  > Differentiated Services Field: 0x12 (DSCP: Unknown, ECN: ECT(0))
    Total Length: 68
    Identification: 0x485d (18525)
  > Flags: 0x4000, Don't fragment
    ...0 0000 0000 0000 = Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
    Header checksum: 0xbcf9 [validation disabled]
    [Header checksum status: Unverified]
    Source: 192.168.56.102
    Destination: 60.60.0.1
> Transmission Control Protocol, Src Port: 2404, Dst Port: 48606, Seq: 1, Ack: 17, Len: 16
> IEC 60870-5-104-Apci: -> I (1,0)
v IEC 60870-5-104-Asdu: ASDU=3 C_SC_NA_1 Act     IOA=45000 'single command'
    TypeId: C_SC_NA_1 (45)
```

Figure 24: Modified DSCP field by the P4-based DSCP tagger for the ASDU header TypeID 45. DSCP Hex value 0x12 corresponds to DSCP value of 18. As captured in Wireshark.

```
> Frame 714: 91 bytes on wire (728 bits), 91 bytes captured (728 bits) on interface 0
> Ethernet II, Src: Vmware_be:79:a1 (00:0c:29:be:79:a1), Dst: SuperMic_6a:6c:c0 (3c:ec:ef:6a:6c:c0)
v Internet Protocol Version 4, Src: 192.168.56.102, Dst: 60.60.0.1
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  > Differentiated Services Field: 0x28 (DSCP: AF11, ECN: Not-ECT)
    Total Length: 77
    Identification: 0x669d (26269)
  > Flags: 0x4000, Don't fragment
    ...0 0000 0000 0000 = Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
    Header checksum: 0x9e9a [validation disabled]
    [Header checksum status: Unverified]
    Source: 192.168.56.102
    Destination: 60.60.0.1
> Transmission Control Protocol, Src Port: 2404, Dst Port: 48738, Seq: 1, Ack: 26, Len: 25
> IEC 60870-5-104-Apci: -> I (67,66)
v IEC 60870-5-104-Asdu: ASDU=3 C_SE_TA_1 Deact   IOA=48000 'set point command, normalized value with time tag CP56Time2a'
    TypeId: C_SE_TA_1 (61)
```

Figure 25: Modified DSCP field by the P4-based DSCP tagger for the ASDU header TypeID 45. DSCP Hex value 0x28 corresponds to DSCP value of 40. As captured in Wireshark.

In the Figures 24 and 25, the modified DSCP fields are displayed. In normal smart grid communications over TCP, the DSCP values are not changed at the application level and remain to be 0. The P4-based DSCP tagger changes the value as required by the DSO, so that the MNO can provide application specific mobile networking services (such as slicing) based on the DSCP value.

3.4 Implementation of P4-based Network Slicer

The P4 program that enables network slicing allows for the creation of two types of slices. First slice is the basic slice which isolates E2E connection between two network devices. The second slice is the DSCP based network slice, which spans the same E2E connection. Both slices for the same E2E connection can be provided with QoS and packet queuing parameters that are different from each respective slice for QoS based isolation. In addition, network slices for other E2E connections can be deployed in such a way that they are either isolated or devices in a slice can be connected to device in the other slice, provided that the devices belong to the same IP subnet.

The P4 program for the network slicer once again starts with header definitions for Ethernet and IPv4 headers. In addition, header definition for the User Datagram Protocol (UDP) is also added. After the headers are defined, the next stage is the parsing.

In the Figure 26, the parsing stage starts with the parsing of the Ethernet header, which then continues to IPv4 parsing, and then finishing after UDP header is parsed. By default all the packets are parsed appropriately and not rejected even when the above parsing sequence does not conform to the packet structure. There is no checksum verification implemented in this P4 program, so the program skips to the next stage.

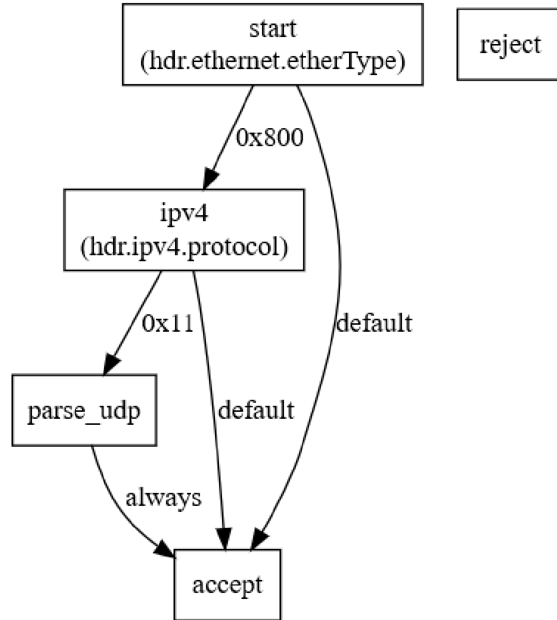

Figure 26: Parsing stage logic for the network slicer.

3.4.1 Ingress Processing for Network Slicer

The network slicing implementation is incorporated throughout the ingress and egress stages of the P4 pipeline, and is not limited to a single pipeline. The entire code for the P4-based network slicer can be viewed in Appendix B. In the ingress processing, the network process for slicing is performed as below:

As depicted in the Figure 27, the ingress stage starts with the invocation of the switch_table with the forward action function responsible for switching the incoming packets from ingress port to the egress port. Once the necessary actions are called, the process invokes the slicein table, where the table's key is the Ipv4 source address. The slice_in table invokes the add_slice action, where the slice ID is associated with the key entered into the table. After this, the DSCP fields are checked. The DSCP values for verification can be configured by the network admin. If the approved DSCP values are detected, the DSCP slice is created with the Ipv4 destination address as the key for the dspc_slicein table. Along with the DSCP check, strict priority queuing (highest priority) is applied to the network packets. Considering the hardware used, it was decided to use highest priority for both the DSCP values (18 and 40). In the table, the same add_slice action is invoked, where the network admin can provide a different slice ID to activate the DSCP slice. Apart from this, if the packets are not detected with the appropriate DSCP value, then the default priority queuing (priorty 0) is applied. That is, normal slices are applied with lowest priority, while DSCP slices are given highest priority.

Subsequently, the P4 extern known as the P4 meter is expressed through the meter_table. Here, the slice ids that have been assigned for the slices are used as

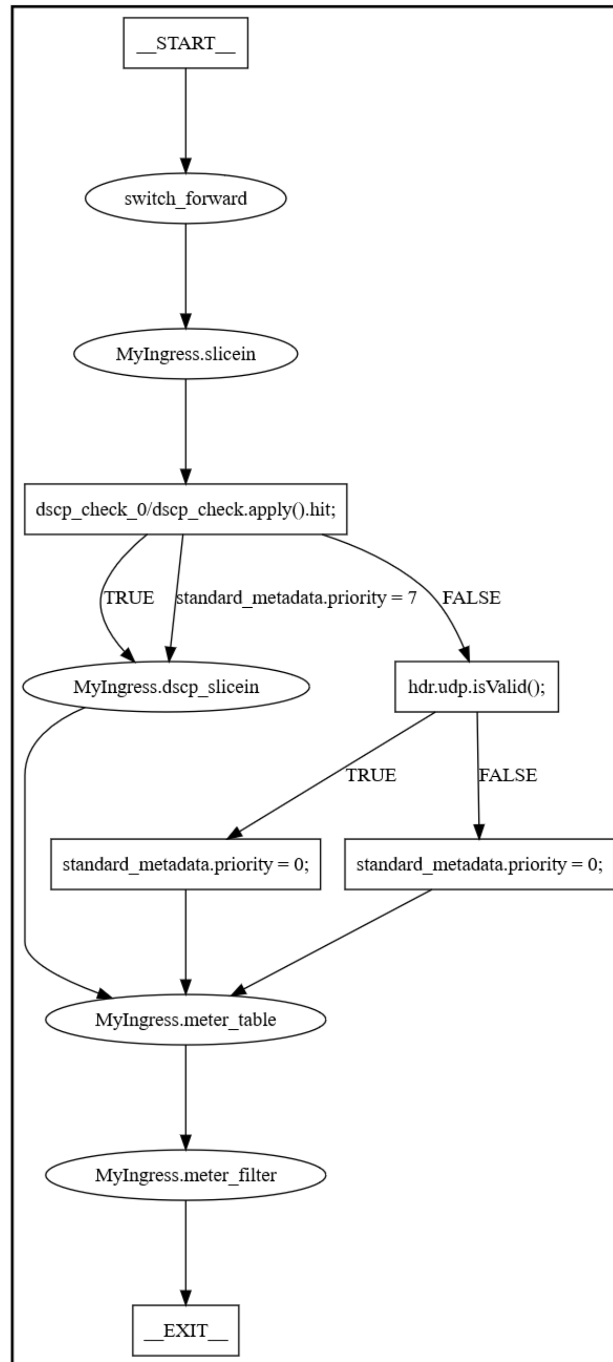

Figure 27: Ingress stage logic for the network slicer.

keys to apply the P4 meter. The meter_table invokes the m_action function, where the P4 meter is initialised and has slice ID assigned to it. The P4 meter extern used in this implementation of network slicing is based on trTCM as outlined by the IETF [24]. The meter extern is used to control the packet rate for each slice. Therefore, slices can be tailored with different queuing and packet rate for applying QoS.

3.4.2 Egress Processing for Slicing

The P4 program then moves onto the egress stage. In the egress stage, each slice is provisioned with a P4-based counter, where this extern object can be used to keep count of the packets and bytes transmitted through the slice. The logic for the egress processing is as follows:

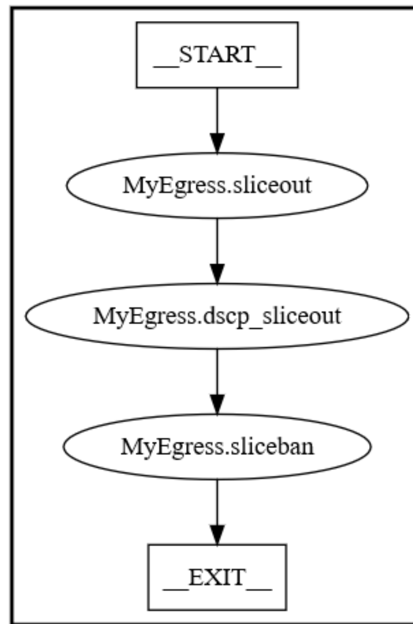

Figure 28: Egress stage logic for the network slicer.

As depicted in the above Figure 28, the egress stage mainly invokes the network functions through the following tables: sliceout, dscp_sliceout and sliceban. Sliceout table consists of IPv4 destination address and the slice ID as a key-pair. When the key-pair is detected in the packets, the table invokes the function known as slice_action() which initializes a counter which is associated to the slice. The dscp_sliceout table is very similar to the sliceout table. The key-pairs are IPv4 destination address and the slice ID assigned for the dscp-based slice.

After the application of the counter for slices, the sliceban table is applied. The sliceban table takes the slice ID as well as IPv4 source addresss as the keys. When the values are entered into the table, the table invokes the drop function, where the BMv2 drop the packets and do not forward them. The sliceban is an optional feature that was implemented during the development. The P4 pipeline then continues with IPv4 header checksum computation and finally ends in the deparsing of the Ethernet, IPv4 and UDP headers.

3.4.3 Slice Implementation and Regulating Packet Rates

The normal and the DSCP slices are applied through the runtime CLI while the P4 program is running on the BMv2 switch. The CLI commands for normal slices were implemented as follows:

```
#Slice ID 1 between RTU (60.60.0.1) and the SCADA (192.168.56.102)
table_add slicein add_slice 60.60.0.1 => 1
table_add slicein add_slice 192.168.56.102 => 1

#Slice ID 2 between UE2 (60.60.0.2) and the Remote Host2
    (192.168.56.104)
table_add slicein add_slice 60.60.0.2 => 2
table_add slicein add_slice 192.168.56.104 => 2

#Assigning P4 counter 1 to Network Slice 1
table_add sliceout slice_action 1 60.60.0.1 => 1
table_add sliceout slice_action 1 192.168.56.102 => 1

#Assigning P4 counter 2 to Network Slice 2
table_add sliceout slice_action 2 60.60.0.2 => 2
table_add sliceout slice_action 2 192.168.56.104 => 2

#Isolating network slices #1 and #2
table_add sliceout Drop 1 60.60.0.2 =>
table_add sliceout Drop 1 192.168.56.104 =>
table_add sliceout Drop 2 192.168.56.102 =>
table_add sliceout Drop 2 60.60.0.1 =>
```

The above commands implement the network slices with slice ids 1 and 2. Slice isolation is also ensured in such a way that devices in one slice cannot connect to devices in the other. As for DSCP slice, it is implemented with the following runtime CLI commands:

```
#Check for DSCP values 18 and 40 and apply highest traffic queuing
    priority
table_add dscp_check dscp_priority 18 => 7
table_add dscp_check dscp_priority 40 => 7
#Rest of the slices have default queuing priority (level 0)

#Assign Slice ID 6 for the DSCP based slice between SCADA and RTU
table_add dscp_slicein add_slice 60.60.0.1 => 6
table_add dscp_slicein add_slice 192.168.56.102 => 6

#Assign P4 counter #6 to Slice ID 6 with respect to SCADA and RTU
table_add dscp_sliceout slice_action 6 60.60.0.1 => 6
table_add dscp_sliceout slice_action 6 192.168.56.102 => 6

#Isolate DSCP slice from Slice #2 devices
table_add dscp_sliceout Drop 6 60.60.0.2 =>
table_add dscp_sliceout Drop 6 192.168.56.104 =>
```

As previously mentioned, DSCP check is performed first and the traffic with the assigned DSCP values have higher priority compared to other traffic. The slice ID assigned for the DSCP slice is 6. And the slice with the slice ID 2 is isolated from

the DSCP slice. However, the slice is not isolated from the slice ID 1, since the DSCP slice is applied to the same connection between the RTU and the SCADA. Additionally, slices can be banned immediately from use in case of security attacks or breaches with the following commands:

```
#Ban A Slice
table_add sliceban Drop 02 60.0.0.2 =>
table_add sliceban Drop 02 192.168.56.104 =>

#Remove Slice Ban
table_clear sliceban
```

In addition to the strict priority queuing for the normal and DSCP slices, the packet rates in the slices can be controlled by the usage of P4 meters. The following commands are applied to implement the control for packet rates:

```
#Initializing P4 meter extern object slicerate_meter
table_set_default meter_filter drop
table_add meter_filter NoAction 0 =>

#Provisioned Packet Rate for Slice #1
table_add meter_table m_action 1 => 1
meter_set_rates slicerate_meter 1 0.01:1 0.05:1
#If packet size = 1000 bytes,
#Packet rate = 0.01 x (1 packet/1 us) x 1000 bytes x 8 bits = 80
   Mbps

#Provisioned Packet Rate for Slice #2
table_add meter_table m_action 2 => 2
meter_set_rates slicerate_meter 2 0.00001:1 0.00005:1
#Packet rate = 0.00001 x (1 packet/1 us) x 1000 bytes x 8 bits = 80
    kbps

#Provisioned Packet Rate for DSCP Slice #6
table_add meter_table m_action 6 => 6
meter_set_rates slicerate_meter 6 0.01:1 0.05:1
#Packet rate = 80Mbps
```

The runtime CLI commands entered during P4 program's runtime allows the network admin to control the packet rates in the network slices. The same commands were also used during the actual implementation and testing phases of this thesis. The visualization of the entire P4 pipelines for slicing is depicted in the Figures 29 and 30.

The figures outline the ingress and egress processing of the P4 program for implementing the network slicing. Figure 29 depicts the P4 pipeline, where the packet transmission rates are not controlled for each slice. Therefore, this pipeline did not assign the P4 meters for each network slice yet. As seen in the ingress stage of the pipeline, packets are classified based on the DSCP values that are configured to be checked as per the network admin. Once the necessary packets are identified with specific DSCP values, they are assigned to the DSCP slice while the rest are assigned to their respective slices based on their IPv4 destination ad-

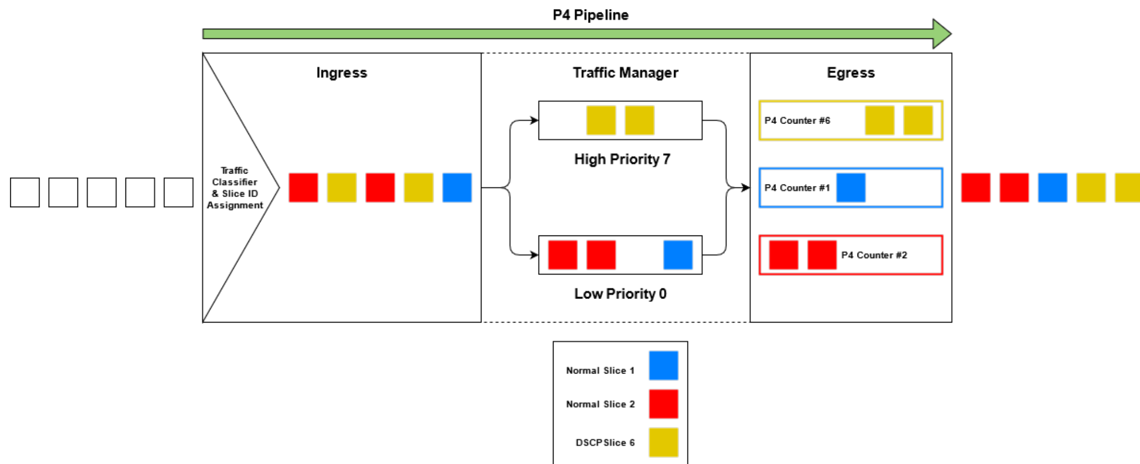

Figure 29: P4 pipeline for network slicer with only strict priority queuing and counters. Enlarged figure can be viewed in Appendix E.

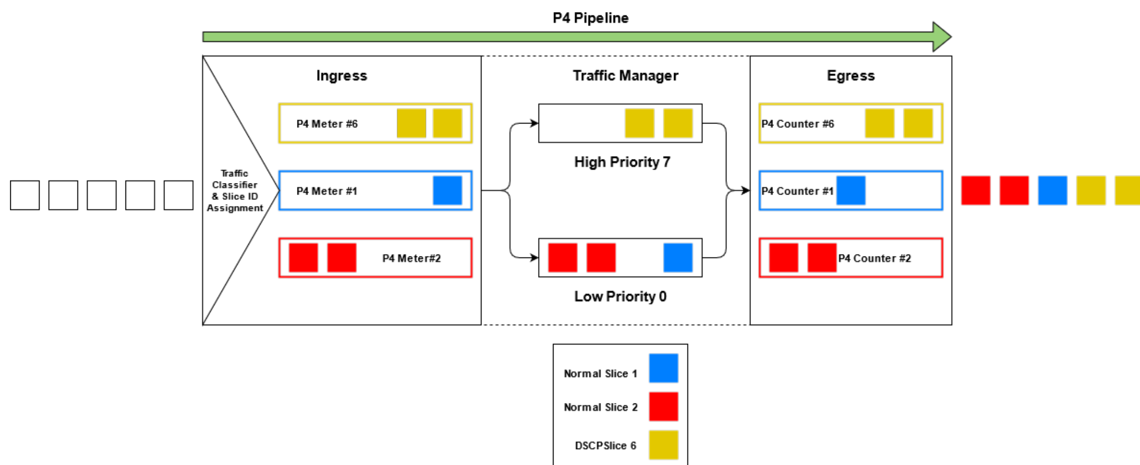

Figure 30: P4 pipeline for network slicer with P4 meters, strict priority queuing and counters. Enlarged figure can be viewed in Appendix F.

dresses as configured by the network admin. During the movement of the packets between the ingress and the egress stage of the pipeline, each the packets with the appropriate DSCP value gets the highest priority. Other traffic with DSCP value of 0 (default) gets the lowest priority. When the packets arrive in egress, each packet tagged with specific slice id are pushed into a P4 counter assigned to the respective slices. The P4 counters counts the packet and byte count while the packets traverses through them. In this manner, the number of packets transmitted can be obtained for each slice. The packets then exit the interface through the egress port.

Figure 30 depicts a very similar P4 pipeline, however this time the P4 meters are activated. The incoming packets gets classified in the same manner as described before. But each packet assigned to a slice id gets allocated a P4 meter with specific configuration as prescribed through the runtime CLI. The P4 meter controls the

rate the packets gets transmitted and in this manner, bit rate for each slice can be allocated by the network admin as required. Apart from this, the rest of the process are the same. However, the P4 switch requires high amount of Central Processing Unit (CPU) and memory resources to perform this kind of network processing.

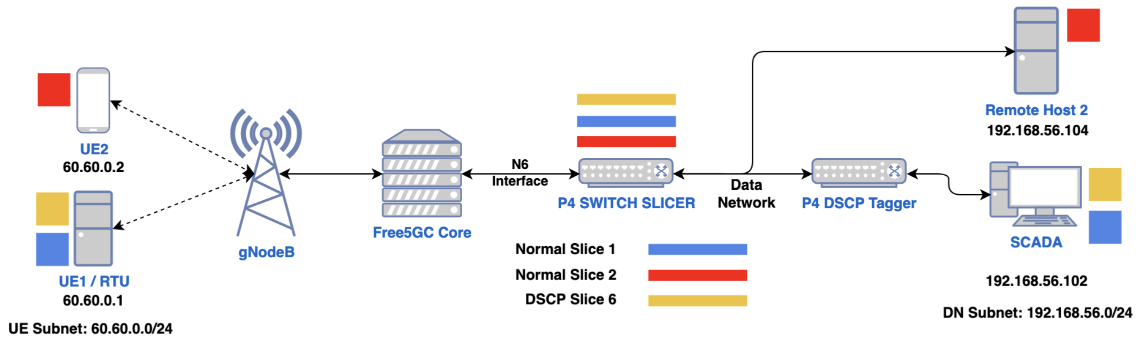

Figure 31: Visualisation of the entire 5G testbed with virtual network slicing at the N6 interface. Enlarged figure can be viewed at Appendix G.

Figure 31 depicts an overview of the network slicing in the 5G testbed and shows where exactly this takes place. In hindsight, the network slicing is limited to the N6 interface between the 5G core and the DN. The slice id that packets are tagged with, only exists within the P4 switch and is not communicated to the SDN controller. Therefore, this makes it impossible to implement an E2E network slicing. As mentioned before, the P4 program for slicing allows the slices to be isolated from each other, so that they are secure. In addition, a slice ban functionality is also implemented for immediate reaction to security breaches and attacks on the UE devices or the E2E connection between the UE and the remote devices in the DN.

4 Results

This chapter firstly provides an overview of the types of measurements taken and then briefly outlines the scenarios that were considered for the 5G testbed and the slicing setup that was discussed in the previous sections. Later in the chapter, an analysis of the measurements will be provided.

The main measurements that were taken for the analysis of the effectiveness of the network slices are mainly Round Trip Time (RTT) and TCP bandwidth with the help of the ping and iperf tools respectively. Measurements based on the UDP protocol were not taken into consideration, mainly due to the fact that they gave inconsistent measurement values. Throughout the measurement process, IEC 104 traffic was also transmitted between the RTU and the SCADA in the DN. This is done in order to ensure that there is consistent transmission of IEC 104 traffic even though other types of network traffic are being transmitted through the medium.

The following non-slicing and slicing scenarios were considered for the measurement and analysis of the 5G testbed and the network slicer:

- **Scenario #1: Normal Slicing (Slices #1 and #2)**

  In this first scenario, network slices #1 and #2 are deployed for communications between RTU and SCADA along with UE2 and Remote Host 2 respectively. As depicted in Figures 29 and 31.

- **Scenario #2: Normal Slicing + Packet Rate Control for Each Network Slice (#1 and #2)**

  In the second scenario, normal slicing is deployed. In addition, packet transmission rates for each slice are controlled as well. Packet rates are restrained to around 80 Megabits per second (Mbps) for normal slice #1. Whereas, packet rates are controlled to 80 Kbps for slice #2. As depicted in Figures 30 and 31.

- **Scenario #3: Normal Slicing along with DSCP Slice (Slices #1, #2 and #6)**

  In the third scenario, both normal slices #1 and #2, as well as the DSCP slice #6 are deployed. But the packet transmission rates are not restrained for these slices.

- **Scenario #4: Normal Slicing + DSCP Slice + Packet Rate Control for Each Slice**

  In this final scenario, slices are deployed similarly as the previous scenario. That is, normal slices #1 and #2 are deployed along with the DSCP slice #6. But each slice has their packet transmission rate restrained. Slice #1 and

DSCP slice #6 have their bitrates restrained to 80 Mbps. Whereas, slice #2 has its bitrate controlled to 80 Kbps.

It is important to consider the fact that the measurements were taken for slices #1 and #2, while the IEC 104 traffic was being transmitted from the SCADA to the RTU and vice-versa. This applies to all the scenarios. Another important aspect of this network slicing is that, the slice #1 overlaps with DSCP slice #6 for the communication between the SCADA and the RTU. That is, it is not possible to obtain RTT and TCP measurements for the DSCP slice #6, when there are two types of slices for the same communication endpoints (SCADA and RTU) and the DSCP slice allocation is mainly dependant on whether the chosen DSCP values are present in the DSCP field of the IP headers of the transmitted packets or not.

Scenario	**Description**	**RTT Slice 1 (ms)**	**TCP BW Slice 1 (Mbps)**	**RTT Slice 2 (ms)**	**TCP BW Slice 2 (Mbps)**
#1	Normal Slicing	10.61	0.97	7.64	1.04
#2	Normal Slicing and Slice Bitrate Control	10.20	1.01	8.13	0.06
#3	Normal Slicing and DSCP Slicing	10.40	0.82	7.33	1.05
#4	Normal Slicing, DSCP Slicing, and Slice Bitrate Control	10.74	1.21	7.84	0.05

Table 6: End-to-End delay (ms) and TCP bandwidth (Mbps) for slices #1 and #2.

The Table 6 showcases the results obtained for E2E delay for the SCADA and the RTU communications in Slice #1 as well as their TCP bandwidths for each slicing scenarios that were previously mentioned. In addition, E2E delay and TCP bandwidth are provided for the UE 2 and Remote Host 2 devices in Slice #2. The E2E delay is in ms, while the TCP bandwidths are obtained in Mbps.

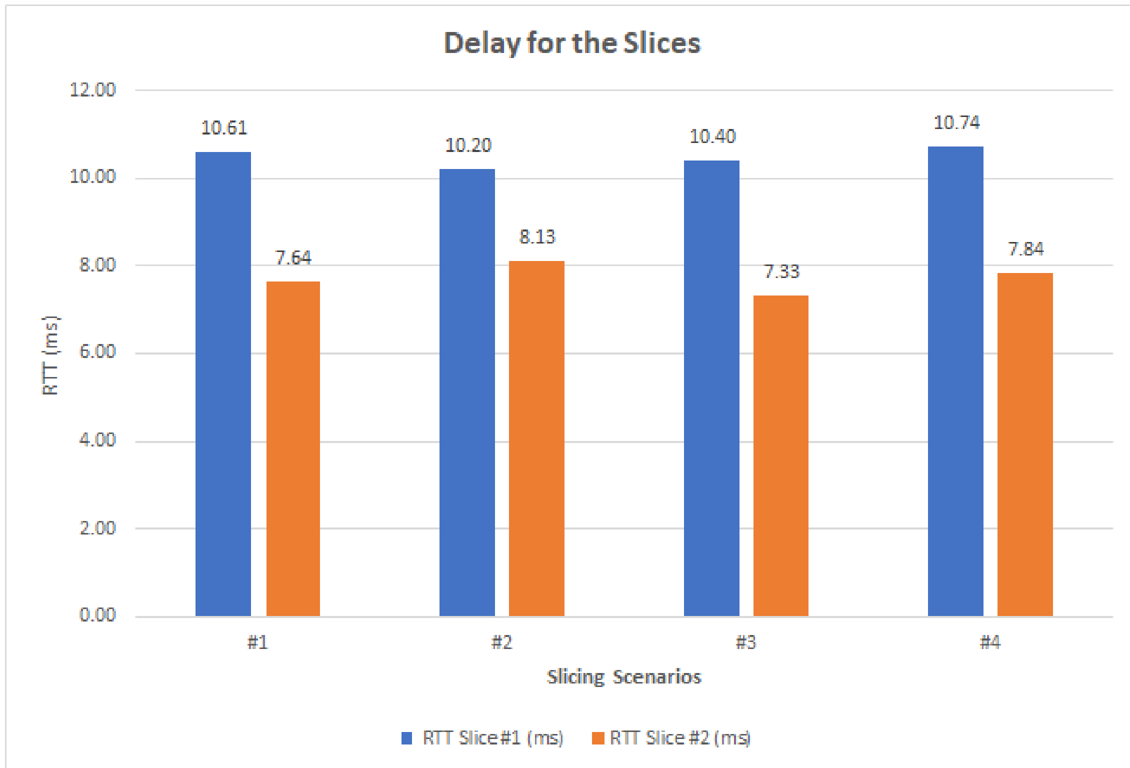

Figure 32: Bar chart comparing the E2E delay for slices #1 and #2.

The Figure 32 showcases the differences in the E2E delays between slices #1 and #2 with respect to each slicing scenarios that were outlined before. All of the E2E delay measurements are averages of the RTT measurements, where the number of ping requests were at least more than ten. On average, the E2E latencies for slice #1 is higher than the E2E latencies for slice #2 for all the slicing scenarios. The difference is around 2.75 ms on average. This difference is mainly due to the fact that, the SCADA device in slice #1 is connected to the P4-based DSCP tagger, while the Remote Host 2 is not connected to it in the DN and is directly connected to the P4-based network slicer (as seen in 20). The DSCP tagger contributes to the extra latency when compared to the E2E latencies obtained for slice #2. This was due to the fact that, all the devices were connected directly and did not have any type of P4-based hardware in between. Apart from this, there is a noticeable increase in latency for slice #2 in scenarios #2 and #4. This is caused by limiting the bitrate of the slice to 80 Kilobits per second (Kbps).

In the Figure 33, the TCP throughput performances for the slice #1 and #2 have been outlined. In scenarios #1 and #3, slice #2 has higher TCP bandwidth, due to the fact that there is only one P4-based device (network slicer) between the Remote Host 2 and the UE2. However, in scenarios #2 and #4, the TCP bandwidths for slice #2 are comparitively very low, with values being around 0.06 and 0.05 Mbps respectively. This is because, the bitrate for the slice #2 has been set to 0.08 Mbps. Apart from this, both the slices show consistent TCP bandwidth of around 1 Mbps

in normal slicing cases.

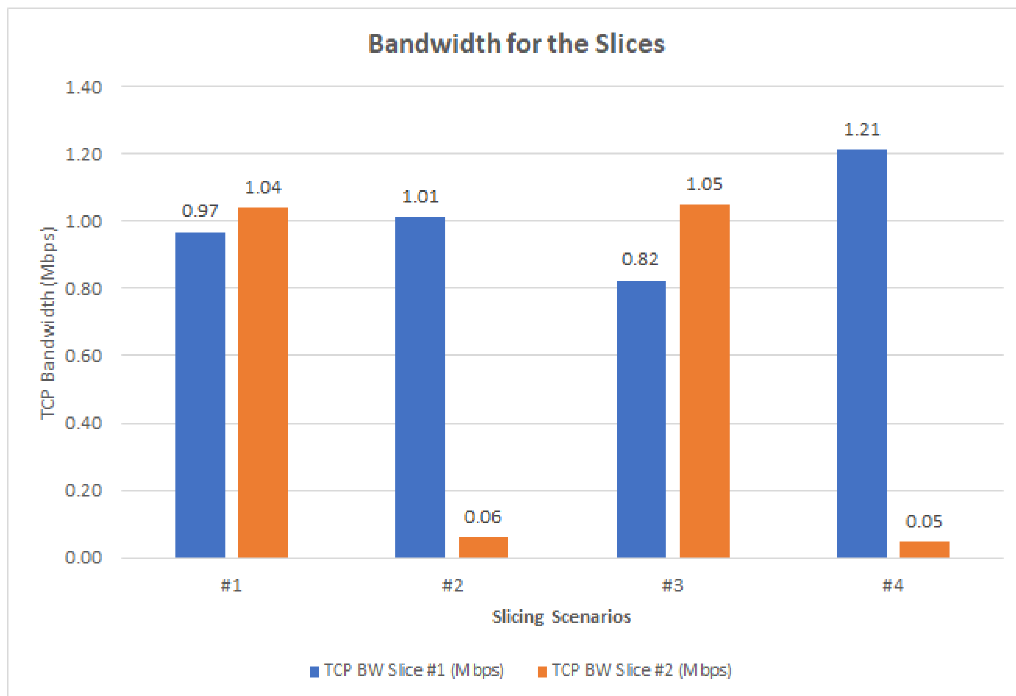

Figure 33: Bar chart comparing the TCP bandwidth delay for slices #1 and #2.

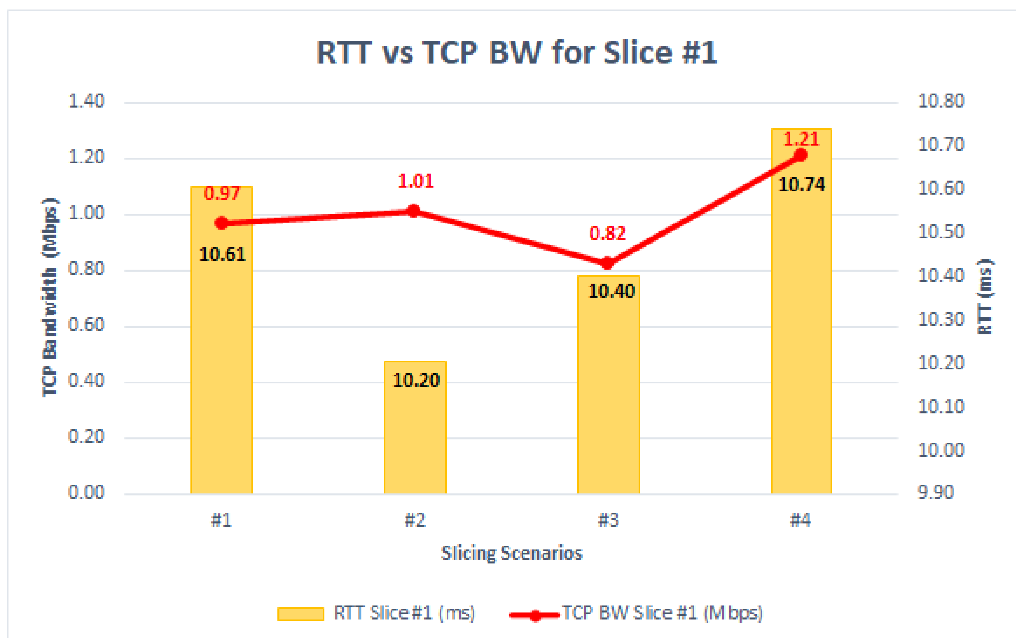

Figure 34: Combo chart displaying E2E delay and TCP bandwidth delay for slice #1.

Figure 34 outlines the RTT and the TCP throughput measurements for slice #1 in a chart consisting of both bar and line graph. For each scenario, the bar graph

represents the E2E delay, while the line graph represents the TCP bandwidth or throughput measurements. The lowest latency was observed in scenario #2 (10.20 ms) while the highest measurement was observed in scenario #4 (10.74). With respect to the TCP bandwidth measurements, highest value was observed in scenario #4, while the lowest was observed in scenario #3.

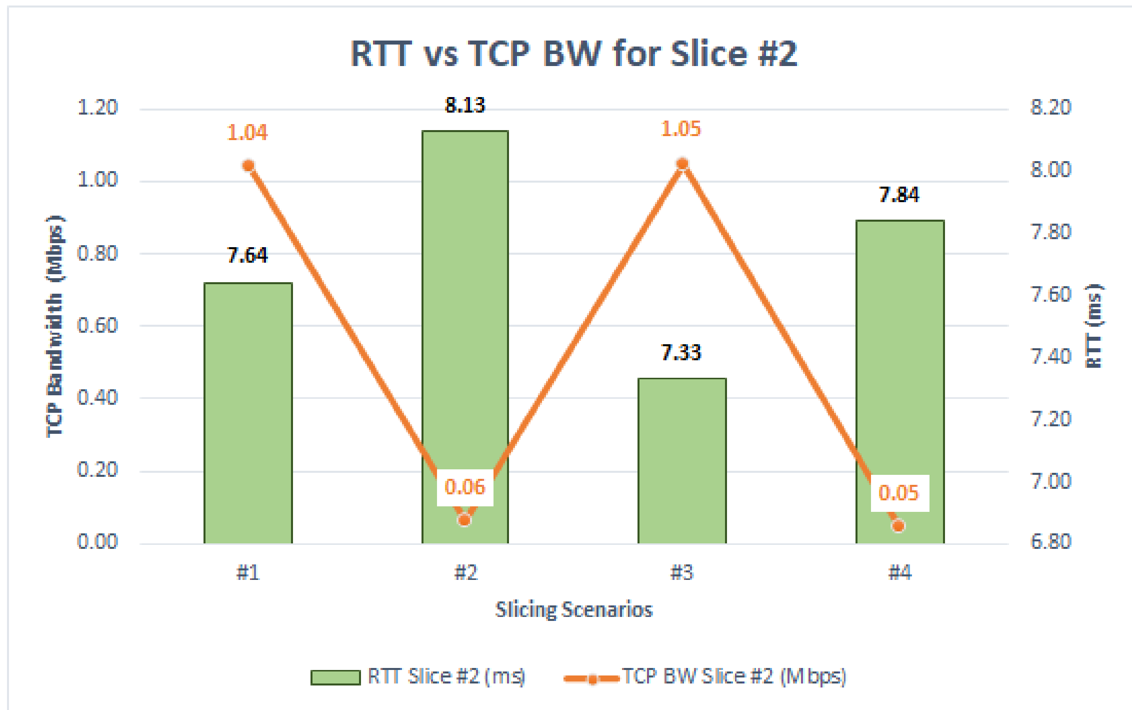

Figure 35: Combo chart displaying E2E delay and TCP bandwidth delay for slice #2.

The combo chart shown in Figure 35 depicts the E2E delay and the TCP throughput measurements for slice #2. In this chart, the low throughput measurements that were observed in the slicing scenarios #2 and #4 are 0.06 Mbps and 0.05 Mbps. Highest TCP bandwidth value, 1.05 Mbps was obtained in slicing scenario #3.

As for the P4 counters, it allows to display the number of packets and bytes that were transferred through the slice. In the scenario #1, the following information was obtained from the P4 counter linked to slice #1:

```
ubuntu@P4switch:~/Desktop/Network Slicer$ simple_switch_CLI
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: counter_read MyEgress.slice_counter 1
MyEgress.slice_counter[1]= (10287470 bytes, 11310 packets)
RuntimeCmd: 
```

Figure 36: P4 counter for slice #1 displaying number of packets and bytes transmitted through the slice.

The Figure 36 displays the amount of bytes transferred through the network slice #1. The P4 counter for slice #1 was invoked after performing the measurement tests for scenario #1.

In scenario #3, the P4 counters were invoked to display the bytes transmitted through the slice before any IEC 104 traffic were transmitted back and forth from the SCADA to the RTU.

```
ubuntu@P4switch:~/Desktop/Network Slicer$ simple_switch_CLI
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: counter_read MyEgress.slice_counter 1
MyEgress.slice_counter[1]= (392 bytes, 4 packets)
RuntimeCmd: counter_read MyEgress.slice_counter 2
MyEgress.slice_counter[2]= (0 bytes, 0 packets)
RuntimeCmd: counter_read MyEgress.slice_counter 6
MyEgress.slice_counter[6]= (0 bytes, 0 packets)
RuntimeCmd: 
```

Figure 37: P4 counters for normal slices #1 and #2, as well as the DSCP slice #6. The slices are displaying the number of packets and bytes transmitted through the slices in the beginning of the scenario #3.

Figure 37 outlines the P4 counters for each slice. They provide the statistics of the byte and packet count for each network slice in scenario #3. But since not many packets were sent, the counters are empty.

```
ubuntu@P4switch:~/Desktop/Network Slicer$ simple_switch_CLI
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: counter_read MyEgress.slice_counter 1
MyEgress.slice_counter[1]= (118398 bytes, 1650 packets)
RuntimeCmd: counter_read MyEgress.slice_counter 2
MyEgress.slice_counter[2]= (0 bytes, 0 packets)
RuntimeCmd: counter_read MyEgress.slice_counter 6
MyEgress.slice_counter[6]= (71554 bytes, 946 packets)
RuntimeCmd:
```

Figure 38: P4 counter for slice #1 displaying number of packets and bytes transmitted through the slices after the IEC 104 traffic were transmitted for scenario #3.

In the Figure 38, the P4 counters displays the byte and packet counts in normal slice #1 and DSCP slice #6. The packets that were tagged with the DSCP values 18 and 40 traversed through slice #6 (946) while the rest of the packets traversed through slice #1 (1650). This can be proven in this Wireshark capture:

```
uesimtun0_scene3.pcap
File  Edit  View  Go  Capture  Analyze  Statistics  Telephony  Wireless  Tools  Help
(frame.number>=1 && frame.number<=2596) && (ip.dsfield==18 || ip.dsfield==40)    Expression...  +
No.  Time      Source          Destination  Protocol  Length
 5  0.025130  192.168.56.102  60.60.0.1    TCP       52
 6  0.025197  192.168.56.102  60.60.0.1    104asdu   68
10  0.040784  192.168.56.102  60.60.0.1    TCP       60
13  0.055668  192.168.56.102  60.60.0.1    TCP       52
14  0.055743  192.168.56.102  60.60.0.1    104asdu   68
18  0.073704  192.168.56.102  60.60.0.1    TCP       60
21  0.092596  192.168.56.102  60.60.0.1    TCP       52
22  0.092663  192.168.56.102  60.60.0.1    104asdu   68
26  0.105970  192.168.56.102  60.60.0.1    TCP       60
29  0.120850  192.168.56.102  60.60.0.1    TCP       52
30  0.120918  192.168.56.102  60.60.0.1    104asdu   68
34  0.136758  192.168.56.102  60.60.0.1    TCP       60
> Frame 5: 52 bytes on wire (416 bits), 52 bytes captured (416 bits)
  Raw packet data
> Internet Protocol Version 4, Src: 192.168.56.102, Dst: 60.60.0.1
0000  45 12 00 34 48 5c 40 00  3f 06 be 0a c0 a8 38 66
0010  3c 3c 00 01 09 64 bd de  9c 3c 8f 97 5c 41 45 12
0020  80 10 01 fd 8a 65 00 00  01 01 08 0a 64 05 06 51
0030  44 96 71 b8
uesimtun0_scene3.pcap    Packets: 68450 · Displayed: 946 (1.4%)    Profile: Default
```

Figure 39: Packets captured at RTU interface with Wireshark displaying the tagged DSCP packets that corresponds to the packet count for DSCP slice #6 .

As seen in Figure 39, the Wireshark packet capture displays the packets that were captured on on the network interface of the RTU device. The Wireshark packet capture was filtered to obtain the IEC 104 traffic, that were tagged with the relevant DSCP values. The filter expression that was used in Wireshark is the following:

```
1 (frame.number>=1 && frame.number<=2596) && (ip.dsfield==18 || ip.
    dsfield==40)
```

The entire IEC 104 traffic exchange between the SCADA and the RTU using the simulator from [32] comprises of 2596 packets. Before other traffic was sent through the network for scenario #3, the IEC traffic was sent first. Therefore, the initial packet count was 2596 packets in total. As observed in the packet count in each slice #1 and #6, combining the number of packets sent through the slices, the total packet count of 2596 is observed. In addition, the the total number of DSCP tagged packets is 946. These packets have been clearly separated by the P4-based network slicer and put into DSCP slice #6, while the rest of the IEC traffic from RTU which are not DSCP tagged are pushed into normal slice #1. Apart from this aspect, the isolation of the slices have been established with the P4-based network slicer. In a way, it acts as a network firewall, that blocks ingress or egress network packets based on the policies administered by the network admin. The network slicer was able to establish slice isolation by blocking the incoming packets that were destined to the devices in different slice. That is, devices in slice #1 cannot communicate with devices in slice #2.

5 Discussion

This chapter first explains the details of the performance of the BMv2 software switches used for network slicing and DSCP tagging. It also provides the reason how thesis implementation meets the requirements of networking for smart grids in general. The chapter then ponders over the usage of encryption along with P4-based BMv2 switch for DSCP tagging application in the IEC 104 smart grid communications. Finally, technical challenges pertaining to the implementation of this thesis are also described.

5.1 Performance of the BMv2 Software Switch

Huge performance degradation for the slicing implementation was observed, since the P4-based BMv2 software switch does not provide very high throughput performance when compared to a virtual network switch such as an Open vSwitch [25]. Moreover, low performance APU boxes installed with VMware ESXi hypervisor were used to partition its hardware resources for hosting P4 switch, remote host (SCADA), basestation and the UEs. Therefore, this too adds to the performance degradation when compared to employing the aforementioned network elements in standalone baremetal nodes.

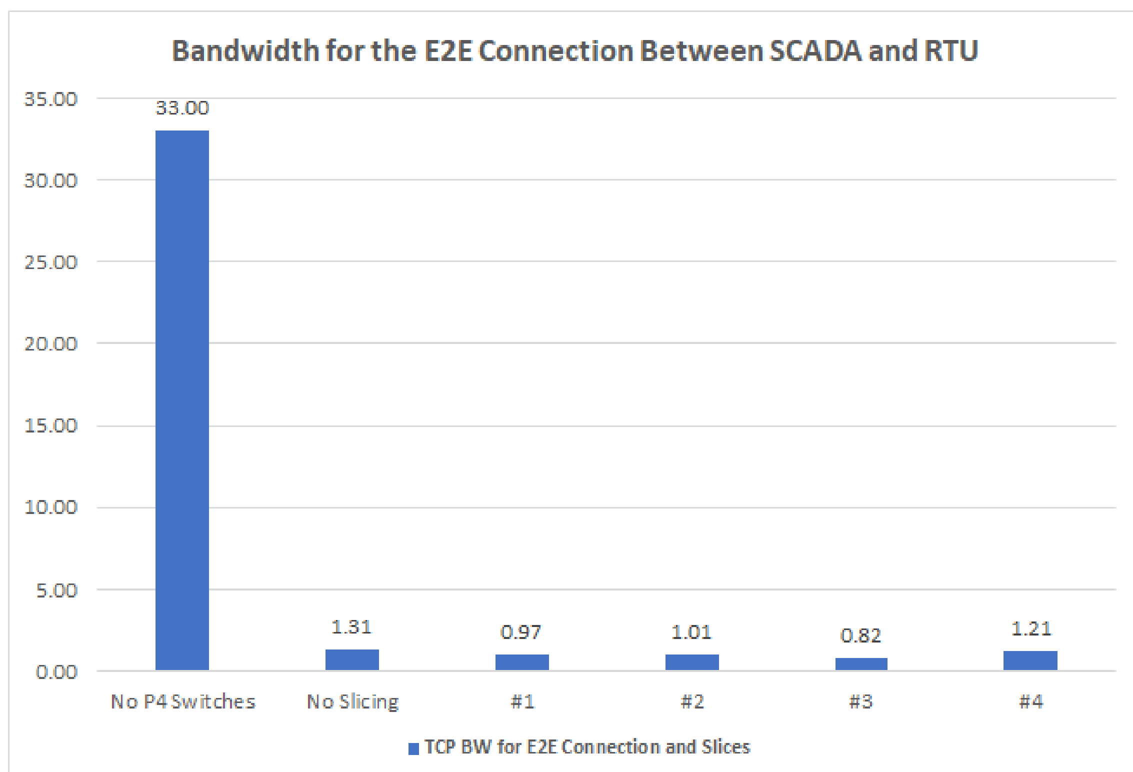

Figure 40: E2E TCP bandwidth between SCADA and the RTU without P4 switches and in network slicing (slice #1) scenarios.

As observed in Figure 40, the 5G network performs better in terms of TCP throughput without the P4 switches. Around 33 Mbps throughput is achieved. It can be concluded that the more network functions are added into the P4 program, the more there is performance overhead for packet processing. For example, it has been observed that the processing of the DSCP tagged packet in the network slicer takes around 1 ms. Whereas, it takes about few hundred microseconds for non-DSCP tagged packets to get processed. This is observed from the logs of the P4-based network slicer. In addition, compared to the network slices generated by the P4-based network slicer, without the BMv2 switch integration, the 5G network displayed higher throughput and better latency performance compared to the network slices. Furthermore, the connection between the Remote Host 2 and the UE2 displayed lower E2E latency compared to the connection between RTU and the SCADA.

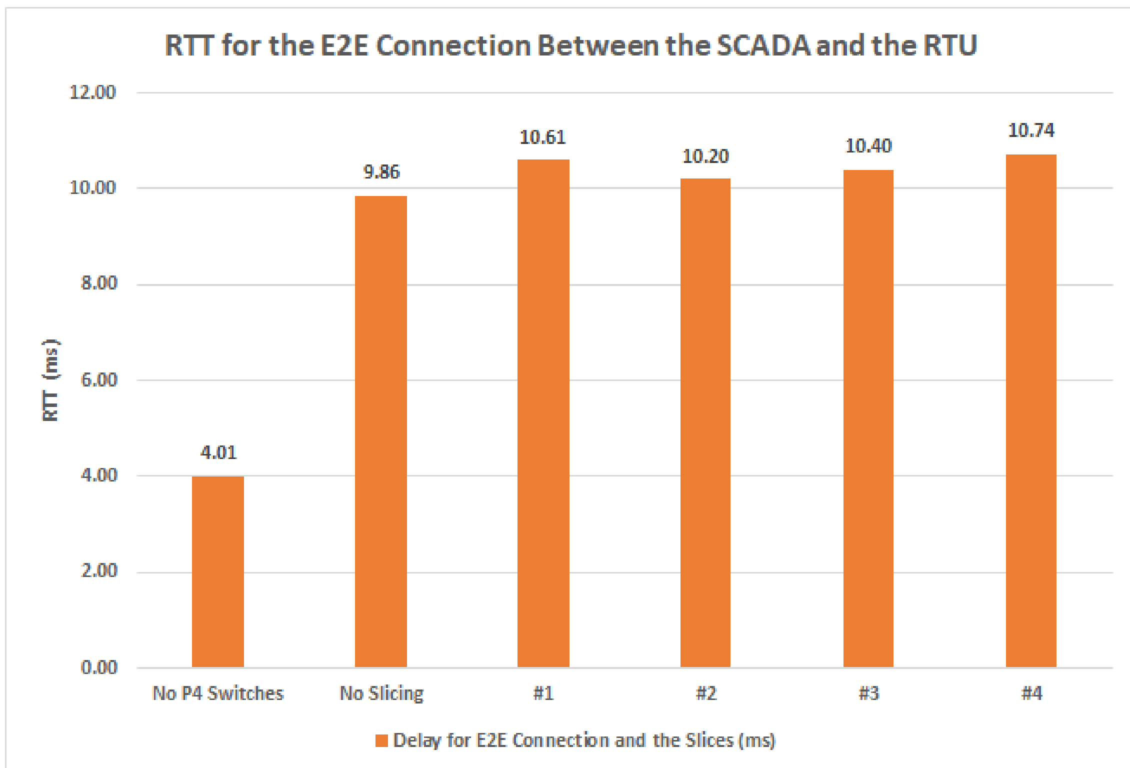

Figure 41: E2E delay between SCADA and the RTU without P4 switches and in network slicing (slice #1) scenarios.

In Figure 41, the E2E delay for the scenario without P4 switches is comparatively lesser for the scenario without P4 switches. With the P4 switches, but without the network slicing, there is a significant increase in E2E delay. However, with respect to the slicing scenarios, this delay is lower. In no slicing and the subsequent slicing #1, #2, #3, and #4 scenarios, two P4 switches (P4-based network slicer and tagger) were connected in the path between the SCADA and the RTU.

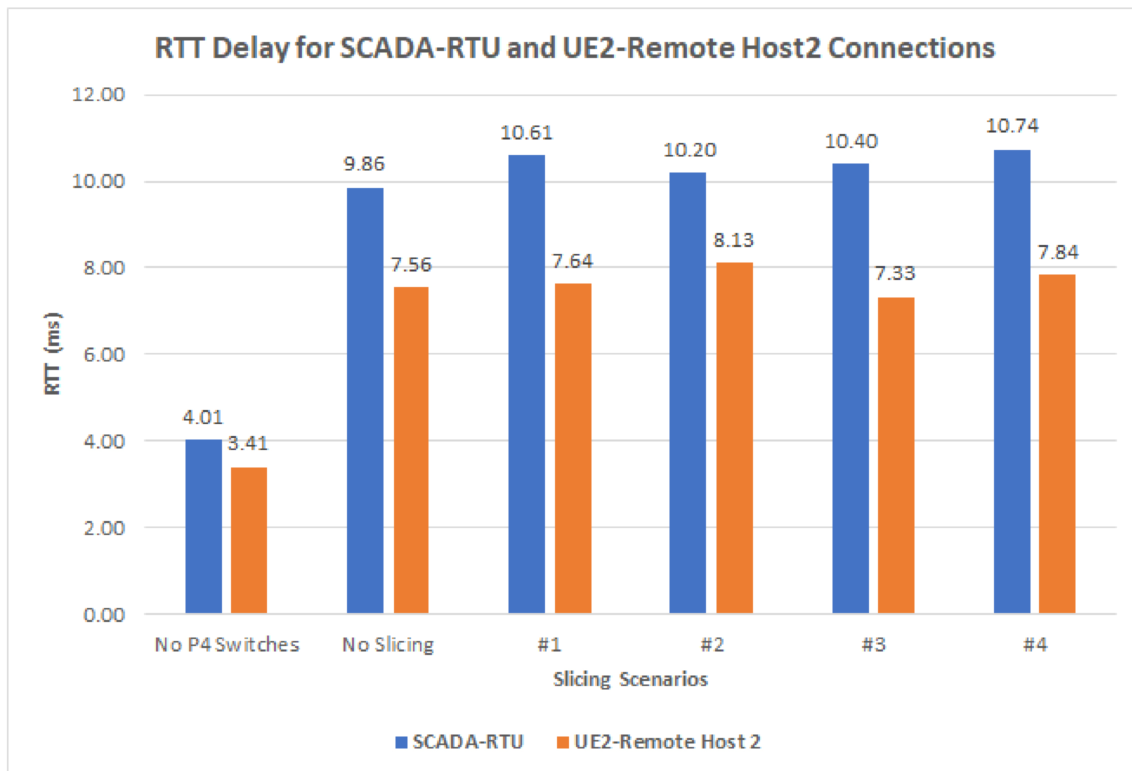

Figure 42: RTT delay comparison between SCADA-RTU connection and the UE2-RemoteHost 2 connection.

As seen in the Figure 42, there is a difference in the RTT delays between the SCADA-RTU and the UE2-Remote Host 2 connections. The following factors contributes in the perceived increase in RTT delay between the SCADA-RTU connection compared to UE2-Remote Host 2 connection:

- There are two BMv2 P4-based software switches (P4-based network slicer and DSCP tagger) in the SCADA-RTU connection path.
- Both the SCADA and the RTU devices are VMs hosted in the same APU box sharing its resources. Therefore, performance degradation is expected.
- There is only one BMv2 P4-based software switch (P4-based network slicer) in the UE2-Remote Host 2 connection path.
- Remote Host 2 in the Data Network is a high performance Linux based laptop equipped with physical network interface and therefore, displays lower latency performance. It is not emulated or running as a VM. This is why there is around 600 microseconds difference in the no P4 switches scenario.

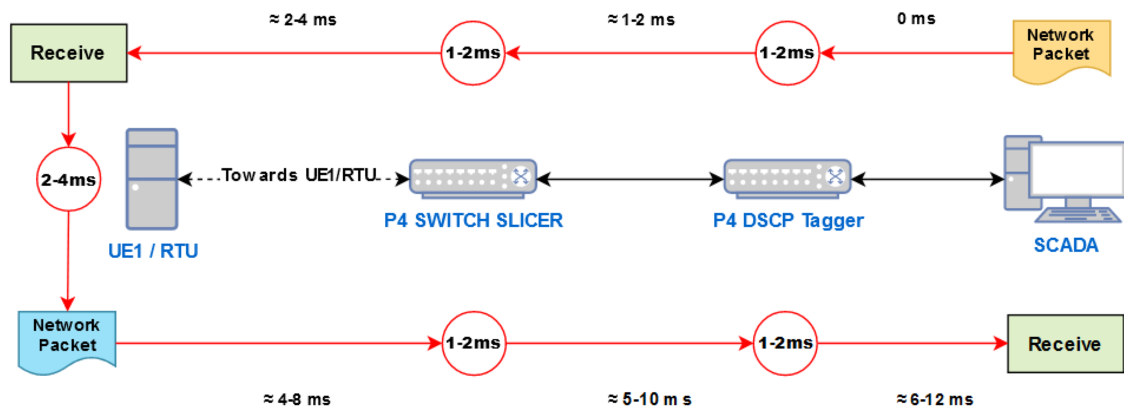

Figure 43: Approximate RTT delay for the SCADA-RTU connection with P4 switches in the path.

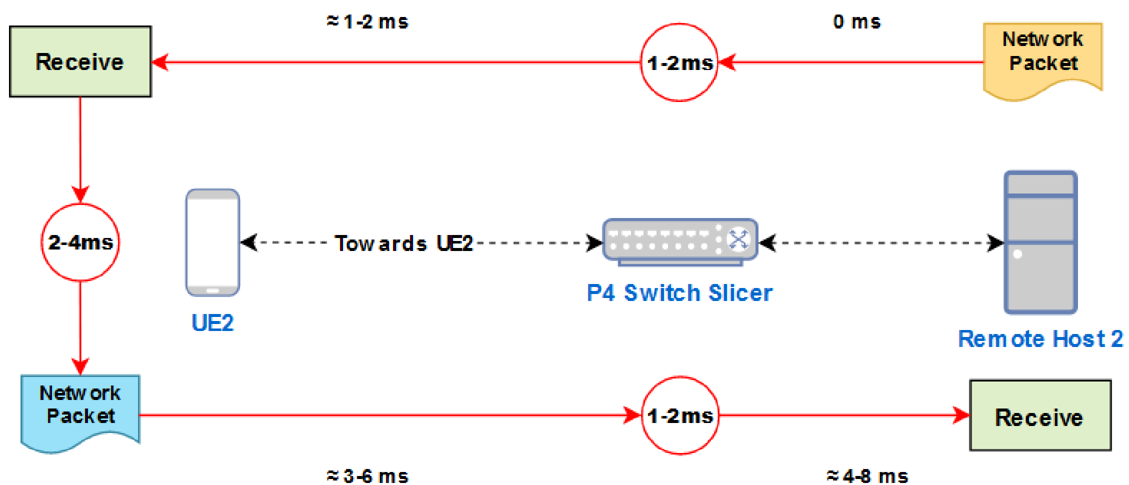

Figure 44: Approximate RTT delay for the Remote Host2-UE2 connection with P4 switch in the path.

The Figures 43 and 44, demonstrate how the P4 switches used for network slicing and DSCP tagging add latency to the SCADA-RTU and the Remote Host2-UE2 connections. Each P4 switch adds around 1-2 ms of latency, while 2-4 ms of latency by each P4 switch is counted when a network packet is transmitted back and forth (round-trip). Additional 2-4ms of latency is added by processing of the packet in the 5G mobile network which consists of the 5G core, basestation and the UE in round-trip transmissions.

In round-trip transmission for SCADA-RTU connection, the latency becomes approximately 6-12ms in total. This is because in round-trip transmission, the network packet gets transmitted through both the P4 switches two times which adds around 4-8ms of latency along with the latency of 2-4ms due to the packet processing

in the 5G network. Whereas in the Remote Host2-UE2 connection, there is only one P4 switch. Therefore in a round-trip transmission for Remote Host2-UE2 connection, the total latency is approximately 4-8ms due to the processing in the P4 switch and in the 5G network.

Additional tests were also performed to gauge the performance of the P4 switch by connecting a workstation over the internet. An iperf test that was performed provided the following results:

```
thottom1@tt-lab12:~$ iperf3 -c speedtest.serverius.net -p 5002 -4
Connecting to host speedtest.serverius.net, port 5002
[  4] local 192.168.9.162 port 44428 connected to 178.21.16.76 port 5002
[ ID] Interval           Transfer     Bandwidth       Retr  Cwnd
[  4]   0.00-1.00   sec  36.8 KBytes   301 Kbits/sec   17   2.83 KBytes
[  4]   1.00-2.00   sec  55.1 KBytes   452 Kbits/sec   13   2.83 KBytes
[  4]   2.00-3.00   sec  74.9 KBytes   614 Kbits/sec   11   2.83 KBytes
[  4]   3.00-4.00   sec  46.7 KBytes   382 Kbits/sec    9   4.24 KBytes
[  4]   4.00-5.00   sec  87.7 KBytes   718 Kbits/sec   14   5.66 KBytes
[  4]   5.00-6.00   sec  96.2 KBytes   788 Kbits/sec   16   4.24 KBytes
[  4]   6.00-7.00   sec  63.6 KBytes   521 Kbits/sec   15   1.41 KBytes
[  4]   7.00-8.00   sec  69.3 KBytes   568 Kbits/sec   15   4.24 KBytes
[  4]   8.00-9.00   sec  96.2 KBytes   788 Kbits/sec   12   5.66 KBytes
[  4]   9.00-10.00  sec   119 KBytes   973 Kbits/sec   14   2.83 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-10.00  sec   745 KBytes   610 Kbits/sec  136             sender
[  4]   0.00-10.00  sec   721 KBytes   591 Kbits/sec                  receiver

iperf Done.
```

Figure 45: Iperf test performed from a workstation connected to the internet through a P4 switch.

The above iperf test in the Figure 45 displays the average TCP bandwidth that was obtained when the workstation was connected to the internet through the BMv2 P4 switch installed in the HP EliteDesk 800 system. The average throughput that was successfully obtained from the sender end (workstation) was around 610 Kbps, while the remote server with the address speedtest.serverius.net observed a TCP throughput of 591 Kbps. A normal consumer-grade switch functions with higher performance compared to the BMv2 switch, albeit without dataplane customizability as observed in the Figure 46. Disabling the macro logger for the BMv2 switch reduces the performance impact immensely. That is, it reduces latency by 1-1.5 ms depending on the length and complexity of the P4 program. But, logging was an important aspect for troubleshooting issues with the generation of the network slices in the P4-based network slicer and therefore was enabled. Apart from this, the usage of the APU with VMware Esxi 6.5 hypervisor for hosting linux based workstations and the P4-switch also impacted the performance. CPU performance for the APU box is clocked at 1GHz, which is quite low.

```
thottom1@tt-lab12:~$ iperf3 -c speedtest.serverius.net -p 5002 -4
Connecting to host speedtest.serverius.net, port 5002
[  4] local 192.168.9.162 port 44296 connected to 178.21.16.76 port 5002
[ ID] Interval           Transfer     Bandwidth       Retr  Cwnd
[  4]   0.00-1.00   sec  10.5 MBytes  88.2 Mbits/sec    0    403 KBytes
[  4]   1.00-2.00   sec  11.2 MBytes  94.2 Mbits/sec    0    403 KBytes
[  4]   2.00-3.00   sec  11.2 MBytes  94.1 Mbits/sec    0    403 KBytes
[  4]   3.00-4.00   sec  11.2 MBytes  94.1 Mbits/sec    0    403 KBytes
[  4]   4.00-5.00   sec  11.2 MBytes  94.1 Mbits/sec    0    403 KBytes
[  4]   5.00-6.00   sec  11.2 MBytes  94.1 Mbits/sec    0    403 KBytes
[  4]   6.00-7.00   sec  11.2 MBytes  94.2 Mbits/sec    0    403 KBytes
[  4]   7.00-8.00   sec  11.2 MBytes  94.1 Mbits/sec    0    403 KBytes
[  4]   8.00-9.00   sec  11.2 MBytes  94.2 Mbits/sec    0    403 KBytes
[  4]   9.00-10.00  sec  11.2 MBytes  94.1 Mbits/sec    0    403 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-10.00  sec   112 MBytes  93.6 Mbits/sec    0             sender
[  4]   0.00-10.00  sec   111 MBytes  93.2 Mbits/sec                  receiver

iperf Done.
```

Figure 46: Iperf test performed from a workstation connected to the internet with a normal consumer-grade network switch.

5.2 Networking Requirements for Smart Grids

The ICT requirements specification which is outlined by the Smart Net initiative provides the following requirements adopting the networking technologies for the appropriate operation of smart grids [12]:

Type/application	Latency requirements	Network technology
Type 1A - Trip (fault isolation & protection)	3-10ms (P1/P2)	Fiber optic communication, Metro-Ethernet, High performance microwave
Type 1B - Other IED automation	20 ms (P3)	Fiber optic, Metro-Ethernet, High performance microwave, LTE-advanced
Type 2 - Medium speed control	100ms (P4)	LTE, WiMAX and above
Type 3 - Low speed control	500 - 1000 ms (P5/P6)	PLC, RF-mesh, 3G, LTE, WiMAX and above
Type 4 - Continuous Raw IED data messages	3-10ms (P7/P8)	Fiber optic, Metro-Ethernet, High performance microwave, future LTE-advanced
Type 5 - File transfer functions	≥ 1000ms (P9)	PLC, RF-mesh, 3G, LTE, WiMAX and above

Figure 47: Network latency requirements for the above network technologies for smart grid applications. As seen in [12].

The Figures 47 and 48 describe the latency and bitrate requirements for the smart grid application. The expected lowest latency for the smart grid application is between 3ms and 10ms. Whereas the maximum expected bitrate is around 2.3 Mbps. However, this is for the synchrophasor application. While the expected bitrate for distribution IEDs of around 15 in number is between 18 and 60 kbps.

Considering these requirements, the performance of the P4-based network slicer almost meets the requirement with respect to the latency, while the bitrate require-

Application	Data source (units)	Message size (bytes)	Latency (seconds)	Data rate (kbps)
On-demand meter reading	625 meters	100	5	100
Multi-interval meter reading	625 meters	1600-2400*	10**	800-1200
Load management	500 customers	64	5	51
Distribution automation	15 IEDs***	150-500	1	18-60
Synchrophasor	100 PMUs	48	0,0167	2300

* 100 bytes x 4 messages per hour x 4-6 hours/reading interval
** From Table 3.1. Higher latencies could also be valid
*** 15 field devices per 1000 meters is a typical value according to [8]

Figure 48: Network bitrate requirements for the above smart grid applications. As seen in [12].

ment is definitely met. However, in terms of meeting the 5G KPIs, the 5G network testbed has not met the requirements since the E2E latency is expected to be below 1ms while the downlink bandwidth is expected to be more than 1 Gigabits per second (Gbps) and the Uplink bandwidth around 100 Mbps.

5.3 What if SCADA-RTU Connection is Encrypted?

It is possible for manufacturers to adopt Transport Layer Security (TLS) for sending IEC 104 traffic between the SCADA devices and RTUs or IEDs. In this scenario, the BMv2 P4-based software switch could be deployed as a containerized application, which uses Nginx for TLS termination to the external networks connecting RTUs and IEDs. The TLS encryption mechanism could be programmed in RTU or IED by the device manufacturer or the DSO may use Kubernetes based controller such as Nginx to interface with external networks supporting TLS encryption.

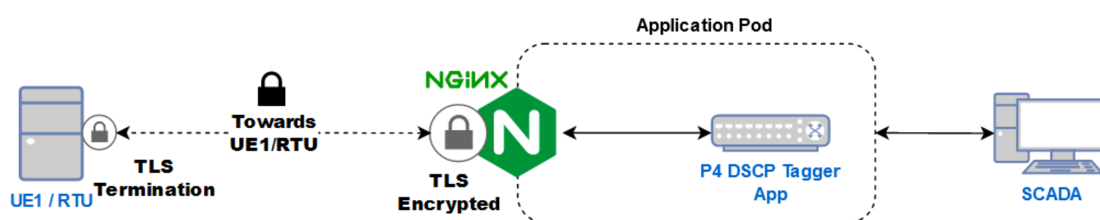

Figure 49: P4 DSCP tagger deployed in Kubernetes pod along with Nginx controller for TLS termination.

As observed in Figure 49, the BMV2 software switch could be deployed as a Kubernetes pod application, where it is programmed to tag DSCP value based on the IEC 104 traffic. An Nginx controller deployed in the pod can be configured to be TLS termination for external connections. The Nginx interface acts as "TLS gateway", where it encrypts and decrypts TLS traffic that is being transmitted through it. With this method, the lack of TLS encryption could be resolved and the BMv2 switch can

easily parse and modify the relevant information in the packets without any difficulty before the packets are sent through the controller to the RTU device on the UE side.

5.4 Technical Challenges

There were several technical challenges that were encountered throughout the duration of this thesis. Main issue was trying to find a suitable hardware that supports P4 programming. Other than that, there were challenges in finding a suitable hardware to host various different Linux based hosts and the BMv2 P4 software switch. Due to this, compromises with the network performance pertaining to the latency and bandwidth had to be made.

5.4.1 The P4-NetFPGA hardware

The P4-NetFPGA hardware, also known as the NetFPGA SUME platform is an FPGA programmable Network Interface Card (NIC) that has been designed and created on the initiative of Xilinx Labs, Stanford University and University of Cambridge.

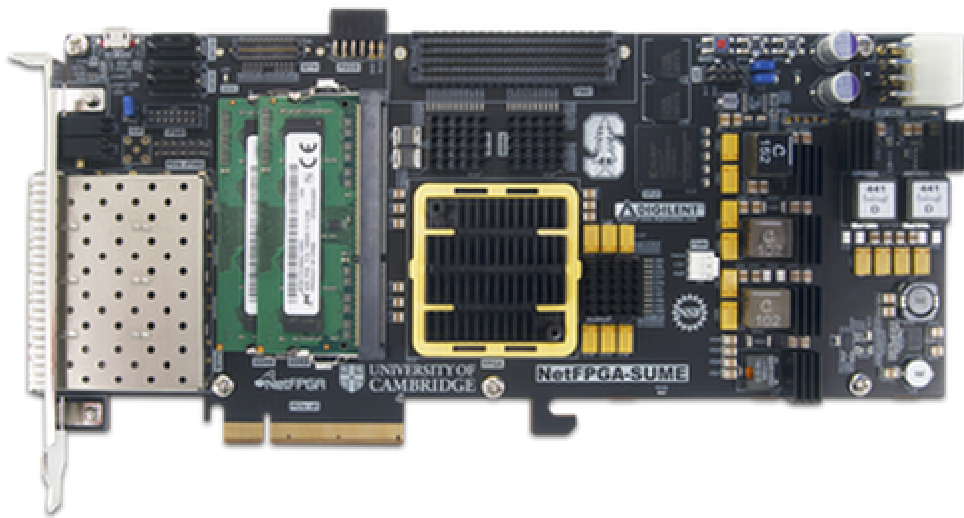

Figure 50: NetFPGA SUME NIC [13].

The NetFPGA-SUME, as seen in Figure 50 consists of four small 10 Gbps form-factor pluggable plus (SFP+) interfaces. The card can be plugged into a workstation motherboard using the PCI Express (PCIe) slot, which can hold considerable amount of high-speed data streams. The NetFPGA SUME platform has been developed for providing a state-of-the art platform for researchers and developers to work with in the realm of computer networking. The NetFPGA SUME board utilizes the SimpleSumeSwitch P4-based architecuture instead of the V1Model. Therefore, some functionalities available in V1Model may not be necessarily available in SimpleSumeSwitch and vice-versa.

However, the platform is not completely open to everyone. In order to program the NetFPGA with P4 language framework, it requires Xilinx's proprietary SDNet module which powers the P4 compiler to turn P4 program into VHSIC Hardware Description Language (VHDL) and Verilog bin files to program the NetFPGA SUME. Gaining access to the proprietary module is quite difficult and requires dealing with the sales support from Xilinx. In addition, usage of the board requires the installation of Xilinx's proprietary IDE tool known as Vivado. The installation and the activation of Vivado is quite cumbersome due to the presence of Digital Rights Management (DRM). After the installation of the SDNet, Vivado modules and relevant tool chains, it was later discovered that the NetFPGA SUME board did not support the motherboard it was connected with the PCIe slot in the workstation. In fact the NetFPGA SUME only supports few computer motherboards that were manufactured in the years 2015, 2016 and 2017. Due to hardware deprecation and discontinued manufacturing, it is difficult to get hold of the motherboards recommended by the NetFPGA community [33]. During the initial stage of this thesis, two NetFPGA SUME boards were obtained for use in order to use them for high speed P4 packet processing. But due to the aforementioned issues, they were not put into use for this thesis.

A master's thesis that worked on high speed Network Address Translation (NAT) with P4 describes the usability issues of the P4-based NetFPGA NIC [34]:

- The P4 based toolchains and modules for NetFPGA SUME is not developed enough.
- Too many steps for the code compilation.
- Scripts for lookup table entries must be invoking tables that are not used for the network processing. Invoking only the required parameters crashes the program.
- The documentations outlining the usage of the P4 based NetFPGA SUME card is spread and there is no straightforward instructions to operate the card properly.
- Depending on the size of the P4 program, the compilation takes upto more than 6 hours preventing faster development process.
- Syntax or other critical errors may not constitute as acceptable errors during compilation. At times, non-critical errors can be shown as critical errors. This makes troubleshooting very difficult.
- Resulting logfiles due to improper compilation and crashes are 5 MB or more in size, preventing the user to troubleshoot easily.
- Even though the NetFPGA kernel module provides access to the NetFPGA's SFP+ interfaces, tools such as Wireshark or tcpdump are not able to detect

packets. Instead they are captured from a workstation connected physically to the NIC on the other end.

Considering all the statements with regards to the usage of the P4-based NetFPGA hardware, this makes it difficult to use it properly with the given time-frame for the thesis project. Therefore, there was a shift in focus from high performance to functionality and try to verify a P4-based solution for smartgrid. And the decision was made to utilize BMv2 P4 software switch in a standalone server instead of the NetFPGA.

5.4.2 Other relevant challenges

Apart from the challenges with the NetFPGA SUME NIC, there were other challenges faced with the BMv2 P4 software switch. Firstly, there were no direct instructions to map BMv2's software interfaces with the physical NIC interfaces on the workstation. This was figured out through trial and error. In addition, packets processed by the P4 switch are not necessarily accepted by other NICs connected to it. Therefore, it was recommended by a researcher to disable NIC offloading in the NICs connected to the P4 switches, which worked and prevented other devices connected to the P4 switch from dropping the packets.

Furthermore, strict priority queuing functionality in the BMv2 P4 software switch is not enabled by default. In the initial stages of the installation of the BMv2 software switch, the v1model.p4 file must be modified to include the following lines:

```
/// set packet priority
@alias("intrinsic_metadata.priority")
bit<3> priority;
@alias("queueing_metadata.qid")
bit<5> qid;
```

Listing 9: Activating strict priority queuing in BMv2 switch for the V1Model architecture.

In addition to this, the following line must be added to simple_switch.h file:

```
define SSWITCH_PRIORITY_QUEUEING_ON
```

After the modification, the files must be compiled and installed. Moreover, BMv2 switch is not considered a production grade P4 switch. Therefore, care must be taken that it must be configured to have macro logging disabled. Unfortunately this was not performed for the P4-based network slicer, because logging was required for troubleshooting issues with the slice and slice rate configuration.

In order to disable macro logging for performance increase, the BMv2 must be installed with the following configuration:

```
git clone https://github.com/p4lang/behavioral-model.git bmv2
cd bmv2
./install_deps.sh
./autogen.sh
```

```
5 ./configure 'CXXFLAGS=-g -O3' 'CFLAGS=-g -O3' --disable-logging-
    macros --disable-elogger
6 make -j8
```

This configuration ensures that the BMv2 switch performs around 1 Gigabits per second (Gbps), albeit there is sufficient amount of hardware resources for the BMv2 switch to operate. In addition, 1 Gbps throughput performance was observed in mininet simulations rather than an actual implementation of the BMv2 in a baremetal server. However, there could be potential solutions for improving the throughput performance of the BMv2 software through several means, apart from allocating higher resources and disabling macro logging.

Apart from the challenges faced during optimization of the BMv2 performance, some challenges were faced when trying to implement an actual 5G network for the integration of the slicing network. First of all, there were limited number of 5G devices to operate with. Secondly, there were issues with connectivity of the UE to the 5G core through a 5G basestation (gNodeB), since the network was operating in standalone mode. PDU sessions were not established for the UEs as expected, because only a few modems and devices in the market support connecting to standalone 5G networks. Therefore, a quick decision was made to circumvent this issue by utilizing open source Free5GC 5G core and UERANSIM simulator for simulating basestation and the UE in linux based workstations. With respect to actual carrier-grade 5G hardware, the type of hardware used for the implementation of the 5G core were consumer-grade at best. This has contributed to reduction in performance for the 5G network that was implemented.

6 Conclusion

This thesis focused on implementing a network slicing solution on the segment between the 5G core and the DN. The main objective was to implement the network slicing to secure the communication between the SCADA and the RTU in the smart grid environment utilizing the IEC 104 protocol to secure their communications. The slicing mechanism was implemented in the dataplane using the P4 programming framework. The implementation of the network slicing was successful in terms of isolation that was achieved between the slices. In addition to the main goal, one of the sub-objectives were to implement a solution to tailor the smart grid traffic to support the imposition of the QoS requirements. This was performed by modifying the DSCP value in the IP header fields of the smart grid traffic based on the IEC 104 protocol ASDU header's TypeID field values. The parsing of the smart grid traffic and the subsequent modification of the DSCP values were achieved with dataplane programming utilizing the P4 language. Furthermore, the QoS requirements are imposed by utilizing strict priority queuing and P4 meters, which is based on two rate three colour marker.

The sub-objective was to find a platform for running P4 programs. The P4-based slicing mechanism was applied in the BMv2 P4 software switch. The BMv2 switch provides access to the dataplane through the P4 language framework. It can be programmed with P4 with custom network functions as per the wishes of the network programmer. As for interacting with the dataplane in BMv2, this is performed through the runtime CLI which is considered to be interfacing with the control plane of the BMv2 switch. The runtime CLI allows the network admin to interact with the network functions outlined in the P4 program. The network slices were created by entering the necessary information such as IPv4 addresses in the lookup tables corresponding to the network functions defined in the P4 program. The information that constitutes the network slice are IPv4 address, which are tied to the slice id. As for the tagging of the relevant DSCP values, this too was performed with the utilization of P4 program that was running in another instance of BMv2 software switch connected to the SCADA device in the DN. Therefore, smart grid traffic outgoing from the SCADA device to the RTU was tagged with appropriate DSCP values, which are utilized by the network slicer to push the smart grid traffic into a separate DSCP slice distuinguished from the main slices for the same E2E connection. The entire network slicing and the DSCP tagging platform were implemented on a 5G network that was powered by open source Free5GC 5G core. The basestation and the UEs comprising of the RTU and a second UE device were simulated using UERANSIM program.

For this thesis' demonstration purposes, a total number of three network slices were created. Two of the slices were normal slices, whereas the third slice was generated based on the DSCP values in the IP header fields and the IPv4 address of the packets in the smart grid traffic. By default, network traffic in each slice undergoes strict priority queuing in the network slicer's P4 pipeline. The packets

pushed into the DSCP slice gets the highest priority, while the packets in other slices get a lower priority. In addition to the queuing, the bit rate in each slice can also be controlled as required by the network administrator using P4 meters implemented in the network slicing program.

In terms of meeting the performance requirements for 5G networks, the combination of both the 5G testbed and the network slicer did not meet performance requirements in terms of latency (less than 1 ms) as well as having a network throughput of more than 1 Gbps in at least the downlink direction. Furthermore, the standalone 5G network could only display a maximum throughput of around 33 Mbps. The average round trip latency with respect to the usage of the P4 two switches were around 10.36 ms, whereas the TCP throughput performance was around 1.06 Mbps. This kind of performance is quite terrible with respect to the current performance of the 5G communications and network slicing paradigms.

However, the performance of the slices has met the requirements of the smart grid networking, where both the data rates and the latency fall into the required parameters for sustaining smart grid operations in good condition. The slicing implementation also displayed that QoS requirements can be imposed by controlling the bitrates in the slices.

6.1 Future Work

There are some additional studies that must be performed to improve the performance of P4 switches. Throughout the process of this thesis several P4-based solutions were considered. The initial solution was to use the NetFPGA SUME NIC as the P4 hardware. However, due to various technical issues encountered along with the complexity of its usage, the plan for its usage for this thesis was scrapped. Therefore, it is recommended to perform more research on finding appropriate methods and hardware to fully utilize the NetFPGA's potential. If this is realized, we expect to obtain around 100 or more times the throughput performance of the BMv2 switch acheived for this project.

Furthermore, another P4-based solution for running P4 programs is the T4P4S compiler that allows the compilation of P4 programs into Data Plane Development Kit (DPDK) based application. T4P4s is a retargetable compiler for the P4 language. The compiler mainly creates DPDK applications that can be utilized by DPDK supported NICs. The T4P4S compiler also utilizes the V1Model of P4-based architecture and therefore, the P4 program for both the network slicer and the P4-based DSCP tagger developed for this thesis will be compatible with the T4P4S compiler without any changes to the code. Therefore, this thesis suggests for looking into implementing the network slicing and the DSCP tagging with the T4P4S compiler, through which adequate improvement in latency and throughput performances are expected to be observed.

Additionally, in the beginning planning stage of the thesis, the scope of the network slicing covered E2E. That is, the network slicing was supposed to be implemented for the RAN network, Core network and the transport network. However, this makes E2E slicing with P4 to be very complex. Furthermore, there will be a need for an intricate control plane application that can manage slices in all the three elements without any issues. Hence, it was decided to implement network slicing only at the transport network segment connecting the DN of the 5G network. But during the developmental phase of the network slicing, relevant parsing of the GPRS Tunnelling Protocol (GTP) was achieved, and therefore the P4 code for network slicing at the segment between the RAN and the 5G core was developed. However, due to time constraints not much tests could be done with this. In addition, GTP relies on UDP protocol for its operation, and this requires UDP checksum computation for the GTP packets to be transmitted reliably. The P4 code has not implemented the UDP checksum, and therefore the network slicing on the RAN segment cannot be achieved. Accordingly, it is highly suggested to implement the UDP checksum computation, so that the slicing mechanism can also be implemented at the RAN segment. The code for the network slicing in the RAN segment is included in Appendix C.

The slicing mechanism relies on BMv2's runtime CLI control plane application, which is running on a thrift server. There is a gRPC iteration of the BMv2 software switch, which allows network admins to administer P4 configurations remotely. This can help in improving centralized network administration. Future work could focus on implementing a control plane application for the centralized remote management of network slices in P4 hardware such as the BMv2 software switch.

Moreover, the performance tests that were done for gauging the performance of the P4-based network slicer and the DSCP tagger were not adequate enough due to the time contraints. It is highly suggested that rigorous testing must be performed, which takes TCP and UDP protocols into consideration. With respect to the devices used for the implementation of the P4-based network slicers, 5G core network and the basestation/UE simulators, it is suggested to implement the entire infrastructure in Docker or Linux based containers. Containers allow to port applications in a portable and granular manner. Container technology skips hardware incompatibility while deploying applications in the OS environment. It is assumed that the network performance would be better when the applications are deployed in containers hosted in a powerful workstation or server.

Finally, the V1Model architecture model for the P4 does not have default priority queuing functionalities. The strict priority functionality had to be activated manually in a non-straightforward manner as discussed in section 5.4.2. It is recommended to implement a queuing mechanism for network slicing. The new queuing mechanism could be integrated with the slicing mechanism developed for this thesis to obtain a more robust and centralized network slicing system.

References

[1] J. Gozdecki, A. Jajszczyk, and R. Stankiewicz, “Quality of service terminology in ip networks,” *IEEE Communications Magazine*, vol. 41, no. 3, pp. 153–159, 2003.

[2] K. Stouffer, V. Pillitteri, S. Lightman, M. Abrams, and A. Hahn, “Guide to industrial control systems (ics) security,” *NIST Special Publication 800-82*, vol. Revision 2, May 2015.

[3] Typhoon-HIL, “Iec 61850 sampled values protocol,” tech. rep.

[4] P. Matoušek, “Description and analysis of iec 104 protocol,” tech. rep., 2017.

[5] P. Parol, “P4 network programming language—what is it all about?,” Apr 2020.

[6] P4.org, “P4 language tutorial,” 2018.

[7] T. P. L. Consortium, “P416 language specification,” Oct 2019.

[8] Huawei, “Token bucket,” *Huawei*.

[9] L. L. Peterson and O. Sunay, *5G mobile networks: A systems approach*. Morgan and Claypool Publishers, 2020.

[10] ETSI, “Ts 123 501 - v15.3.0 - 5g; system architecture for the 5g,” Sep 2018.

[11] R. Rokui, S. Homma, X. d. Foy, L. M. Contreras, P. Eardley, K. Makhijani, H. Flinck, R. Schatzmayr, A. Tizghadam, C. Janz, and et al., “Ietf network slice for 5g and its characteristics,” Nov 2020.

[12] R. Rodriguez, A. Gil de Muro, J. Lopez, and K. Bañuelos, “Ict requirements specifications - smartnet project,” tech. rep., Nov 2016.

[13] Digilent, “Netfpga-sume virtex-7 fpga development board.”

[14] p4lang, “Tutorials/basic.p4 at master · p4lang/tutorials,” Apr 2019.

[15] J. Navarro-Ortiz, P. Romero-Diaz, S. Sendra, P. Ameigeiras, J. J. Ramos-Munoz, and J. M. Lopez-Soler, “A survey on 5g usage scenarios and traffic models,” *IEEE Communications Surveys Tutorials*, vol. 22, no. 2, pp. 905–929, 2020.

[16] X. Fang, S. Misra, G. Xue, and D. Yang, “Smart grid — the new and improved power grid: A survey,” *IEEE Communications Surveys Tutorials*, vol. 14, no. 4, pp. 944–980, 2012.

[17] H. Zeynal, M. Eidiani, and D. Yazdanpanah, “Intelligent substation automation systems for robust operation of smart grids,” in *2014 IEEE Innovative Smart Grid Technologies - Asia (ISGT ASIA)*, pp. 786–790, 2014.

[18] M. Z. Gunduz and R. Das, "Cyber-security on smart grid: Threats and potential solutions," *Computer Networks*, vol. 169, p. 107094, 2020.

[19] P. Radoglou-Grammatikis, P. Sarigiannidis, I. Giannoulakis, E. Kafetzakis, and E. Panaousis, "Attacking iec-60870-5-104 scada systems," in *2019 IEEE World Congress on Services (SERVICES)*, vol. 2642-939X, pp. 41–46, 2019.

[20] M. Liebsch and U. Fattore, "Control-/data plane aspects for n6 traffic steering," Mar 2019.

[21] Z. Kotulski, T. Nowak, M. Sepczuk, M. Tunia, R. Artych, K. Bocianiak, T. Ośko, and j.-p. Wary, "On end-to-end approach for slice isolation in 5g networks. fundamental challenges," pp. 783–792, 09 2017.

[22] R. Schlegel, S. Obermeier, and J. Schneider, "A security evaluation of iec 62351," *Journal of Information Security and Applications*, vol. 34, p. 197–204, 2017.

[23] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, p. 87–95, July 2014.

[24] "rfc2698," *A Two Rate Three Color Marker*, Sep 1999.

[25] p4lang, "P4lang/behavioral-model: The reference p4 software switch."

[26] Free5GC, "Free5gc/free5gc: Open source 5g core network based on 3gpp r15."

[27] R. F. Olimid and G. Nencioni, "5g network slicing: A security overview," *IEEE Access*, vol. 8, pp. 99999–100009, 2020.

[28] P. Hovila, S. Horsmanheimo, S. Borenius, Z. Li, M. Uusitalo, H. Kokkoniemi-Tarkkanen, and P. Syväluoma, "5g networks enabling new smart grid protection solutions," 06 2019.

[29] I. Afolabi, T. Taleb, K. Samdanis, A. Ksentini, and H. Flinck, "Network Slicing I& Softwarization," *IEEE Communications Surveys and Tutorials*, 2018.

[30] Aligungr, "aligungr/ueransim: Open source 5g ue and ran (gnodeb) implementation.."

[31] P. Engines, "Pc engines website."

[32] RocyLuo, "Rocyluo/iec104tcp: Iec104 client and server simulator," Sep 2015.

[33] NetFPGA, "Motherboard information · netfpga/netfpga-sume-public wiki," Mar 2019.

[34] N. Schottelius, "High speed nat64 with p4," Master's thesis, ETH Zürich, 2019.

A P4 Code for DSCP Tagger

```
/* -*- P4_16 -*- */

#include <core.p4>
#include <v1model.p4>

const bit<16> TYPE_IPV4 = 0x800;

const bit<8> ASDU_TYPEID_45 = 0x2d; //process info - cont
const bit<8> ASDU_TYPEID_46 = 0x2e; //process info - cont
const bit<8> ASDU_TYPEID_47 = 0x2f; //process info - cont
const bit<8> ASDU_TYPEID_48 = 0x30; //process info - cont
const bit<8> ASDU_TYPEID_49 = 0x31; //process info - cont
const bit<8> ASDU_TYPEID_50 = 0x32; //process info - cont
const bit<8> ASDU_TYPEID_51 = 0x33; //process info - cont
const bit<8> ASDU_TYPEID_58 = 0x3a; //command telegram
const bit<8> ASDU_TYPEID_59 = 0x3b; //command telegram
const bit<8> ASDU_TYPEID_60 = 0x3c; //command telegram
const bit<8> ASDU_TYPEID_61 = 0x3d; //command telegram
const bit<8> ASDU_TYPEID_62 = 0x3e; //command telegram
const bit<8> ASDU_TYPEID_63 = 0x3f; //command telegram
const bit<8> ASDU_TYPEID_64 = 0x40; //command telegram
const bit<8> ASDU_TYPEID_101 = 0x65; //system info - cont
const bit<8> ASDU_TYPEID_103 = 0x67; //system info - cont


const bit<8> ASDU_COT = 0x06;

/********************** H E A D E R S *********************/

typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
```

```
}

header tcp_t {
    bit<16> srcPort;
    bit<16> dstPort;
    bit<32> seqNo;
    bit<32> ackNo;
    bit<16> tcp_flags;
    bit<16> wndw_size;
    bit<16> tcp_checksum;
    bit<16> urgentPtr;
    bit<8>  tcp_option_kind;
    bit<8>  tcp_option_kind2;
    bit<8>  tcp_option_kind3;
    bit<8>  tcp_option_len;
    bit<32> tcp_option_tval;
    bit<32> tcp_option_tsecr;
}

header apci_t {
    bit<8> StartByte;
    bit<8> apdu_len;
    bit<8> type_h;
    bit<16> rx;
    bit<8> tx;
}

header asdu_t {
    bit<8> TypeId;
    bit<8> sq;
    bit<8> numix;
    bit<8> cot;
    bit<8> nega;
    bit<8> test;
    bit<8> oa;
    bit<16> addr;
}

struct metadata {
    /* empty */
}

struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
    tcp_t        tcp;
    apci_t       apci;
    asdu_t       asdu;

}

/*************************** P A R S E R  *****************/


```

```
107 parser MyParser(packet_in packet,
108                 out headers hdr,
109                 inout metadata meta,
110                 inout standard_metadata_t standard_metadata) {
111
112     state start {
113
114         packet.extract(hdr.ethernet);
115         transition select(hdr.ethernet.etherType){
116
117             TYPE_IPV4: ipv4;
118             default: accept;
119
120         }
121
122     }
123
124     state ipv4 {
125
126         packet.extract(hdr.ipv4);
127         transition select(hdr.ipv4.protocol){
128             6: parse_tcp;
129             default: accept;
130         }
131     }
132
133     state parse_tcp {
134         packet.extract(hdr.tcp);
135         transition select (hdr.tcp.tcp_flags){
136             {0x8018}: parse_apci;
137             default: accept;
138         }
139     }
140
141     state parse_apci {
142         packet.extract(hdr.apci);
143         transition select (hdr.apci.StartByte){
144             0x68: parse_asdu;
145             default: accept;
146         }
147     }
148
149     state parse_asdu {
150         packet.extract(hdr.asdu);
151         transition select (hdr.asdu.TypeId){default:accept;}
152
153         }
154 }
155
156
157
158
159 /****   C H E C K S U M    V E R I F I C A T I O N   ****/
160
```

```
161
162
163 control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
164
165     apply {  }
166
167 }
168
169 /*****  I N G R E S S   P R O C E S S I N G   *****/
170
171 control MyIngress(inout headers hdr,
172                   inout metadata meta,
173                   inout standard_metadata_t standard_metadata) {
174
175     action drop() {
176         mark_to_drop(standard_metadata);
177     }
178
179     action forward(bit<9> port) {
180         standard_metadata.egress_spec = port;
181     }
182
183     action set_dscp_cont() {
184         // for 58-64
185         hdr.ipv4.diffserv = 40;
186     }
187
188
189
190     action set_dscp_mon() {
191         //for 101 and 103, 45-51
192         hdr.ipv4.diffserv = 18;
193     }
194
195     action set_default_dscp() {
196         hdr.ipv4.diffserv = 0;
197     }
198
199     table switch_forward {
200         key = {
201             standard_metadata.ingress_port: exact;
202         }
203
204         actions = {
205             forward;
206             drop;
207         }
208
209         size = 1024;
210         default_action = drop();
211     }
212
213     table dscp_dest {
214         key = {
```

```
215             hdr.ipv4.dstAddr: exact;
216         }
217         actions = {
218             NoAction;
219         }
220     }
221 
222         table no_dscp {
223             key = {
224                 hdr.ipv4.dstAddr: exact;
225             }
226             actions = {
227                 NoAction;
228             }
229         }
230 
231     apply {
232 
233         if (hdr.ipv4.isValid()){
234             if (dscp_dest.apply().hit){
235                 if((hdr.asdu.TypeId >= ASDU_TYPEID_45 && hdr.asdu.
    TypeId <= ASDU_TYPEID_51) || hdr.asdu.TypeId >= ASDU_TYPEID_101
    && hdr.asdu.TypeId <= ASDU_TYPEID_103) {
236                     set_dscp_mon();
237                 }
238                 else if ((hdr.asdu.TypeId >= ASDU_TYPEID_58 && hdr.
    asdu.TypeId <= ASDU_TYPEID_64)) {
239                     set_dscp_cont();
240                 }
241             }
242             else if (no_dscp.apply().hit || !hdr.apci.isValid()) {
243                 set_default_dscp();
244             }
245 
246     }
247         switch_forward.apply();
248     }
249 }
250 
251 /*****  E G R E S S   P R O C E S S I N G   *****/
252 
253 control MyEgress(inout headers hdr,
254                  inout metadata meta,
255                  inout standard_metadata_t standard_metadata) {
256 
257     apply {  }
258 
259 }
260 
261 /*****   C H E C K S U M    C O M P U T A T I O N   *****/
262 
263 control MyComputeChecksum(inout headers  hdr, inout metadata meta)
    {
264 
```

```
    apply {
  update_checksum(
      hdr.ipv4.isValid(),
            { hdr.ipv4.version,
            hdr.ipv4.ihl,
              hdr.ipv4.diffserv,
              hdr.ipv4.totalLen,
              hdr.ipv4.identification,
              hdr.ipv4.flags,
              hdr.ipv4.fragOffset,
              hdr.ipv4.ttl,
              hdr.ipv4.protocol,
              hdr.ipv4.srcAddr,
              hdr.ipv4.dstAddr },
            hdr.ipv4.hdrChecksum,
            HashAlgorithm.csum16);
    }
}

/*****  D E P A R S E R  *****/

control MyDeparser(packet_out packet, in headers hdr) {

    apply {

        //parsed headers have to be added again into the packet.
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.tcp);
        packet.emit(hdr.apci);
        packet.emit(hdr.asdu);
    }
}

/*****  S W I T C H  *****/

V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

Listing 10: P4 program for DSCP tagging.

B P4 Code for Network Slicer

```
/* -*- P4_16 -*- */

#include <core.p4>
#include <v1model.p4>

const bit<16> TYPE_IPV4 = 0x800;

/********************** H E A D E R S **********************/

typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;
typedef bit<8> value_t;


header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}


header udp_t {
    bit<16> srcPort;
    bit<16> dstPort;
    bit<16> plength;
    bit<16> checksum;
}

struct slice_meta_t {
    bit<8> slice;
}

struct metadata {
slice_meta_t slice_meta;
bit<32> meter_tag;
}
```

```
53
54 struct headers {
55     ethernet_t   ethernet;
56     ipv4_t       ipv4;
57     udp_t        udp;
58 }
59
60 /******************** P A R S E R  **********************/
61
62 parser MyParser(packet_in packet,
63                 out headers hdr,
64                 inout metadata meta,
65                 inout standard_metadata_t standard_metadata) {
66
67     state start {
68
69         packet.extract(hdr.ethernet);
70         transition select(hdr.ethernet.etherType){
71
72             TYPE_IPV4: ipv4;
73             default: accept;
74
75         }
76
77     }
78
79     state ipv4 {
80
81         packet.extract(hdr.ipv4);
82         transition select(hdr.ipv4.protocol){
83             0x11  : parse_udp;
84             default      : accept;
85         }}
86
87     state parse_udp {
88          packet.extract(hdr.udp);
89          transition accept;
90     }
91     }
92
93 /****   C H E C K S U M    V E R I F I C A T I O N   ****/
94
95 control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
96
97     apply {  }
98
99 }
100
101 /*****  I N G R E S S   P R O C E S S I N G   *****/
102
103 control MyIngress(inout headers hdr,
104                   inout metadata meta,
105                   inout standard_metadata_t standard_metadata) {
106
```

```
107     action drop() {
108         mark_to_drop(standard_metadata);
109     }
110 
111     action forward(bit<9> port) {
112         standard_metadata.egress_spec = port;
113     }
114 
115     table switch_forward {
116         key = {
117             standard_metadata.ingress_port: exact;
118         }
119 
120         actions = {
121             forward;
122             drop;
123         }
124 
125         size = 1024;
126         default_action = drop();
127     }
128 
129     action add_slice (value_t value) {
130         meta.slice_meta.slice = value;
131     }
132 
133     table slicein {
134         key = {
135             hdr.ipv4.dstAddr: exact;
136         }
137         actions = {
138             add_slice;
139             NoAction;
140         }
141     }
142     action dscp_priority (bit<3>prio){
143         standard_metadata.priority = prio;
144     }
145 
146     table dscp_check {
147         key = {
148             hdr.ipv4.diffserv: exact;
149         }
150         actions = {
151             NoAction;
152             dscp_priority;
153         }
154     }
155 
156     table dscp_slicein {
157         key = {
158             hdr.ipv4.dstAddr: exact;
159         }
160         actions = {
```

```
161             add_slice;
162             NoAction;
163         }
164     }
165
166     meter(10, MeterType.packets) slicerate_meter;
167
168     action m_action(bit<32> meter_id) {
169         slicerate_meter.execute_meter((bit<32>)meter_id, meta.
    meter_tag);
170     }
171
172     table meter_table {
173         key = {
174             meta.slice_meta.slice: exact;
175         }
176
177         actions = {
178             m_action;
179             NoAction;
180         }
181
182         size = 1024;
183         default_action = NoAction();
184
185     }
186
187
188
189     table meter_filter {
190         key = {
191             meta.meter_tag: exact;
192         }
193
194         actions = {
195             drop;
196             NoAction;
197         }
198
199         size = 1024;
200         default_action = drop();
201
202     }
203
204
205     apply {
206         switch_forward.apply();
207         slicein.apply();
208         if (dscp_check.apply().hit) {
209         dscp_slicein.apply();
210         }
211         else if (hdr.udp.isValid()) {
212         standard_metadata.priority = (bit<3>)0;
213         }
```

```
214         else {
215         standard_metadata.priority = (bit<3>)0;
216         }
217 
218         meter_table.apply();
219         meter_filter.apply();
220     }
221 
222 }
223 
224 /*****  E G R E S S   P R O C E S S I N G   *****/
225 
226 control MyEgress(inout headers hdr,
227                  inout metadata meta,
228                  inout standard_metadata_t standard_metadata) {
229 
230     counter (16384, CounterType.packets) slice_counter;
231 
232     action Drop() {
233         mark_to_drop(standard_metadata);
234     }
235 
236     action slice_action(bit<32> index) {
237         slice_counter.count(index);
238     }
239 
240     table sliceout {
241         key = {
242             meta.slice_meta.slice: exact;
243             hdr.ipv4.srcAddr: exact;
244         }
245         actions = {
246             slice_action;
247             NoAction;
248             Drop;
249         }
250     }
251 
252     table dscp_sliceout {
253         key = {
254             meta.slice_meta.slice: exact;
255             hdr.ipv4.srcAddr: exact;
256         }
257         actions = {
258             slice_action;
259             NoAction;
260             Drop;
261         }
262     }
263 
264     table sliceban {
265         key = {
266             meta.slice_meta.slice: exact;
267             hdr.ipv4.srcAddr: exact;
```

```
268         }
269         actions = {
270             Drop;
271         }
272     }
273 
274     apply {
275     sliceout.apply();
276     dscp_sliceout.apply();
277     sliceban.apply();
278 }
279 
280 }
281 
282 /*****   C H E C K S U M    C O M P U T A T I O N   *****/
283 
284 control MyComputeChecksum(inout headers  hdr, inout metadata meta)
    {
285      apply {
286     }
287 }
288 
289 /*****  D E P A R S E R  *****/
290 
291 control MyDeparser(packet_out packet, in headers hdr) {
292     apply {
293         packet.emit(hdr.ethernet);
294         packet.emit(hdr.ipv4);
295         packet.emit(hdr.udp);
296     }
297 
298 }
299 
300 /*****  S W I T C H  *****/
301 
302 V1Switch(
303 MyParser(),
304 MyVerifyChecksum(),
305 MyIngress(),
306 MyEgress(),
307 MyComputeChecksum(),
308 MyDeparser()
309 ) main;
```

Listing 11: P4 program for Network Slicing.

C P4 Code for Network Slicer in the RAN Segment of the 5G Network

```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

const bit<16> ETHERTYPE_IPV4 = 0x0800;
const bit<8>  IPPROTO_UDP    = 0x11;
const bit<16> GTP_UDP_PORT      = 2152;


/***** H E A D E R S  *****/

typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<48> mac_addr_t;
typedef bit<32> ip4Addr_t;
typedef bit<32> ipv4_addr_t;
typedef bit<8> value_t;

header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

header udp_t {
    bit<16> srcPort;
    bit<16> dstPort;
    bit<16> plength;
    bit<16> checksum;
}
/* GPRS Tunnelling Protocol (GTP) common part for v1 and v2 */

header gtp_common_t {
    bit<3> version; /* this should be 1 for GTPv1 and 2 for GTPv2
   */
```

```
50     bit<1> pFlag;   /* protocolType for GTPv1 and pFlag for GTPv2
       */
51     bit<1> tFlag;   /* only used by GTPv2 - teid flag */
52     bit<1> eFlag;   /* only used by GTPv1 - E flag */
53     bit<1> sFlag;   /* only used by GTPv1 - S flag */
54     bit<1> pnFlag;  /* only used by GTPv1 - PN flag */
55     bit<8> messageType;
56     bit<16> messageLength;
57     bit<32> teid;
58 }
59
60 header gtpv1_extension_hdr_t {
61     bit<24> nodata;
62     bit<8> nextExtHdrType;
63     bit<32> plength;
64 }
65
66
67 struct slice_meta_t {
68     bit<8> slice;
69 }
70
71 struct metadata {
72     slice_meta_t slice_meta;
73     bit<32> meter_tag;
74 }
75
76 struct headers {
77     ethernet_t   ethernet;
78     ipv4_t       ipv4;
79     ipv4_t       inner_ipv4;
80     gtp_common_t gtp_common;
81     gtpv1_extension_hdr_t gtpv1_extension_hdr;
82     udp_t udp;
83     udp_t inner_udp;
84 }
85
86
87
88 /***** P A R S E R  *****/
89
90 parser MyParser(packet_in packet,
91                 out headers hdr,
92                 inout metadata meta,
93                 inout standard_metadata_t standard_metadata) {
94
95     state start {
96         transition parse_ethernet;
97     }
98
99     state parse_ethernet {
100        packet.extract(hdr.ethernet);
101        transition select(hdr.ethernet.etherType) {
102            ETHERTYPE_IPV4: parse_ipv4;
```

```
103             default: accept;
104         }
105     }
106 
107     state parse_ipv4 {
108         packet.extract(hdr.ipv4);
109         transition select(hdr.ipv4.protocol) {
110             IPPROTO_UDP   : parse_udp;
111             default       : accept;
112         }
113     }
114 
115     state parse_udp {
116         packet.extract(hdr.udp);
117         transition select(hdr.udp.dstPort) {
118             GTP_UDP_PORT : parse_gtp;
119             default      : accept;
120         }
121     }
122 
123     state parse_gtp {
124         packet.extract(hdr.gtp_common);
125         transition select(hdr.gtp_common.version, hdr.gtp_common.
    eFlag) {
126             (1,0) : parse_inner;
127             (1,1) : parse_gtpv1_extension_hdr;
128             default : accept;
129         }
130     }
131 
132     state parse_gtpv1_extension_hdr {
133         packet.extract(hdr.gtpv1_extension_hdr);
134         transition select (hdr.gtpv1_extension_hdr.nextExtHdrType)
135         {
136             0x85 : parse_inner;
137             default : accept;
138         }
139     }
140 
141     state parse_inner {
142         packet.extract(hdr.inner_ipv4);
143         transition accept;
144     }
145 
146 }
147 
148 
149 
150 /******  C H E C K S U M    V E R I F I C A T I O N   ******/
151 
152 
153 
154 control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
155 
```

```
156     apply {  }
157
158 }
159
160 /******  I N G R E S S   P R O C E S S I N G   ******/
161
162 control MyIngress(inout headers hdr,
163                   inout metadata meta,
164                   inout standard_metadata_t standard_metadata) {
165
166     action drop() {
167         mark_to_drop(standard_metadata);
168     }
169
170     action forward(bit<9> port) {
171         standard_metadata.egress_spec = port;
172     }
173
174     table switch_forward {
175         key = {
176             standard_metadata.ingress_port: exact;
177         }
178
179         actions = {
180             forward;
181             drop;
182         }
183
184         size = 1024;
185         default_action = drop();
186     }
187
188     action add_slice (value_t value) {
189         meta.slice_meta.slice = value;
190     }
191
192     table slicein {
193         key = {
194             hdr.ipv4.dstAddr: exact;
195         }
196         actions = {
197             add_slice;
198             NoAction;
199         }
200     }
201     action dscp_priority (bit<3>prio){
202         standard_metadata.priority = prio;
203     }
204
205     table dscp_check {
206         key = {
207             hdr.ipv4.diffserv: exact;
208         }
209         actions = {
```

```
210             NoAction;
211             dscp_priority;
212         }
213     }
214
215     table dscp_slicein {
216         key = {
217             hdr.ipv4.dstAddr: exact;
218         }
219         actions = {
220             add_slice;
221             NoAction;
222         }
223     }
224
225     meter(10, MeterType.packets) slicerate_meter;
226
227     action m_action(bit<32> meter_id) {
228         slicerate_meter.execute_meter((bit<32>)meter_id, meta.
    meter_tag);
229     }
230
231     table meter_table {
232         key = {
233             meta.slice_meta.slice: exact;
234         }
235
236         actions = {
237             m_action;
238             NoAction;
239         }
240
241         size = 1024;
242         default_action = NoAction();
243
244     }
245
246
247
248     table meter_filter {
249         key = {
250             meta.meter_tag: exact;
251         }
252
253         actions = {
254             drop;
255             NoAction;
256         }
257
258         size = 1024;
259         default_action = drop();
260
261     }
262
```

```
263
264     apply {
265         switch_forward.apply();
266         slicein.apply();
267         if (dscp_check.apply().hit) {
268         dscp_slicein.apply();
269         }
270         else if (hdr.udp.isValid()) {
271         standard_metadata.priority = (bit<3>)0;
272         }
273         else {
274         standard_metadata.priority = (bit<3>)0;
275         }
276
277         meter_table.apply();
278         meter_filter.apply();
279     }
280
281 }
282
283 /*****  E G R E S S   P R O C E S S I N G   *****/
284
285 control MyEgress(inout headers hdr,
286                  inout metadata meta,
287                  inout standard_metadata_t standard_metadata) {
288
289     counter (16384, CounterType.packets) slice_counter;
290
291     action Drop() {
292         mark_to_drop(standard_metadata);
293     }
294
295     action slice_action(bit<32> index) {
296         slice_counter.count(index);
297     }
298
299     table sliceout {
300         key = {
301             meta.slice_meta.slice: exact;
302             hdr.ipv4.srcAddr: exact;
303         }
304         actions = {
305             slice_action;
306             NoAction;
307             Drop;
308         }
309     }
310
311     table dscp_sliceout {
312         key = {
313             meta.slice_meta.slice: exact;
314             hdr.ipv4.srcAddr: exact;
315         }
316         actions = {
```

```
317             slice_action;
318             NoAction;
319             Drop;
320         }
321     }
322 
323     table sliceban {
324         key = {
325             meta.slice_meta.slice: exact;
326             hdr.ipv4.srcAddr: exact;
327         }
328         actions = {
329             Drop;
330         }
331     }
332 
333     apply {
334     sliceout.apply();
335     dscp_sliceout.apply();
336     sliceban.apply();
337 }
338 
339 }
340 
341 /*****   C H E C K S U M    C O M P U T A T I O N   *****/
342 
343 control MyComputeChecksum(inout headers  hdr, inout metadata meta)
    {
344      apply {
345          /* add UDP checksum computation here */
346     }
347 }
348 
349 /*****  D E P A R S E R  *****/
350 
351 control MyDeparser(packet_out packet, in headers hdr) {
352     apply {
353         packet.emit(hdr.ethernet);
354         packet.emit(hdr.ipv4);
355         packet.emit(hdr.udp);
356         packet.emit(hdr.gtp_common);
357         packet.emit(hdr.gtpv1_extension_hdr);
358         packet.emit(hdr.inner_ipv4);
359     }
360 
361 }
362 
363 /*****  S W I T C H  *****/
364 
365 V1Switch(
366 MyParser(),
367 MyVerifyChecksum(),
368 MyIngress(),
369 MyEgress(),
```

```
370 MyComputeChecksum(),
371 MyDeparser()
372 ) main;
```

Listing 12: P4 program for Network Slicing at the RAN segment.

D Conceptualized 5G testbed with Network Slicer and DSCP Tagger

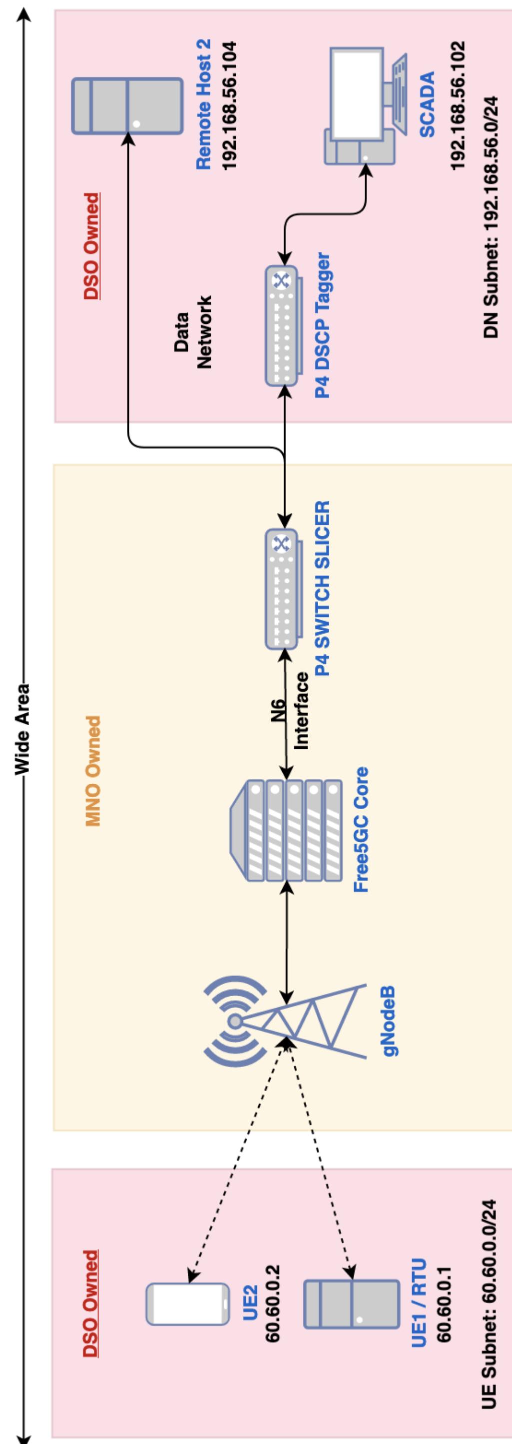

Figure D1: 5G testbed with slicer

E P4 Pipeline for Network Slicing with only Strict Priority Queuing and P4 Counters

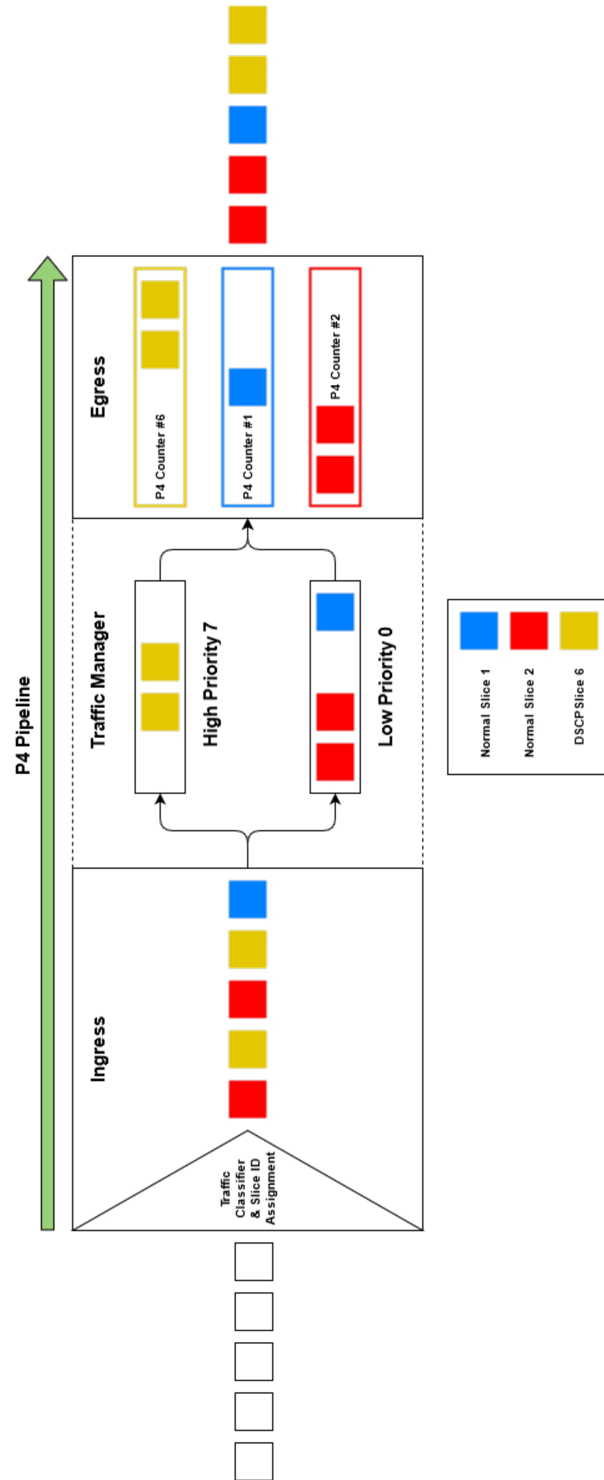

Figure E1: P4 pipeline for network slicer with strict priority queuing and counters.

F P4 Pipeline for Network Slicing with Meters

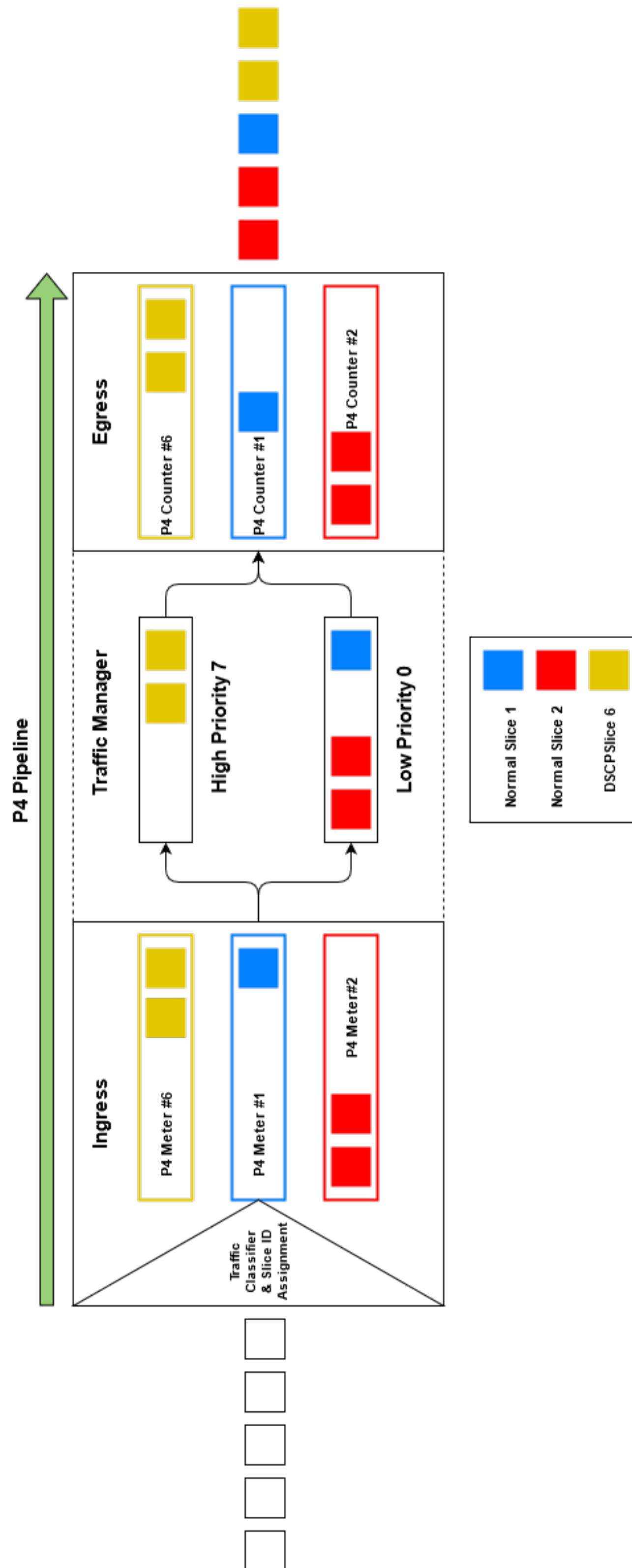

Figure F1: P4 pipeline for network slicer

G Network Slicing at the N6 Interface

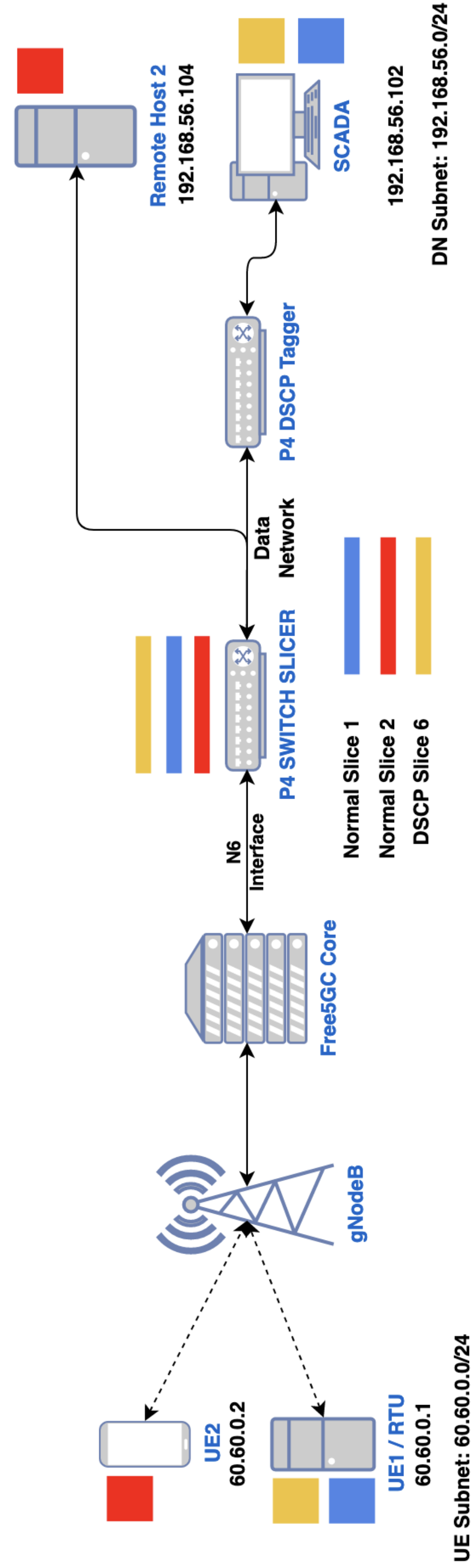

Figure G1: Network slicing at the N6 interface of the 5G infrastructure.