Sapienza
Università di Roma

Analysis and implementation of Software-Defined Network (SDN) techniques on core network nodes for next generation cellular networks

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Corso di Laurea Magistrale in Informatica (LM-18)

Cattedra di Reti Avanzate

Relatore	Candidato
Dott.ssa Petrioli Chiara	Popa Pavel
	Matricola: 1644389

Anno Accademico 2016/2017

Dott.ssa Petrioli Chiara	Popa Pavel
Firma	*Firma*
Data	*Data*

Abstract

The substantial rise in smartphone market penetration and the terrific growth of user-data traffic in mobile networks is facing big challenges in the way the traditional mobile network infrastructures operate. In the network evolution context, **5G** is the next telecommunications standard beyond the current **4G**. 5G is not necessarily going to look like a straightforward transition from the previous generations of mobile technology. As for the mobile core network, we might look beyond a simple *evolution*, and think more in terms of *revolution*. Speaking of revolution, a new technology which is foreseen to step in the coming years is Software-Defined Networking (SDN), which, due to its ability to automate core network operations and to decouple software components from a fixed-function hardware equipment, is expected to add great management flexibility and operations elasticity to the current Long-Term Evolution (LTE) architecture. There is, however, a catch. Almost all of today's mobile network operators have employed dedicated hardware into their core network architectures. This means that, although SDN would prove to be an excellent technology for being exploited, even potentially saving fortunes on network management operations and necessary hardware equipment, there still have to be defined several key transition stages in order to produce a smooth evolution of the architecture itself.

This thesis aims to develop a Proof of Concept of a new Evolved Packet Core (EPC) SDN-based user plane architecture, analyzing pros and cons of such approach when applied to a today's traditional cellular network technology. The actual implementation consists of a GTP-enabled Open vSwitch [2] in such a way that, being able to "speak" and "understand" GTP, it could be deployed on a user plane core network node, which makes it possible to leverage SDN techniques for user plane operations. Open vSwitch is an SDN-based open source software switch. The prototype implementation consists in extending Open vSwitch so it can understand GTP-related packet flows. In this way, having a GTP-capable Open vSwitch, provides us with the ability to use SDN paradigms in mobile networks, coping with their present scalability and flexibility issues.

This work focuses on the interfaces and protocols of the user plane only, which addresses a wide set of challenging scenarios that will be explained throughout the thesis.

Contents

1 Introduction

1.1 Motivation

The challenges of a 5G core network, when looking at the number of sessions that will be running over the mobile core network, are typically the proper dimensioning and dynamic scaling of the network functionalities. The mobile user data demand goes beyond the GSMA's predictions in terms of network scalability requirements and type of applications. As an example of the scaling issues that future networks will cope with, the statistics presented in [13] show that by 2020 there will be 10x more apps on 100x more devices running concurrently, with 1000x more bandwidth demand. Then, we note that sorting these numbers up translates into a million times more concurrent sessions than what today's mobile core network nodes can handle. Therefore, the first thing the network operators do realize is that it goes beyond the human control to actually manage and optimize each service over this shared network without some form of autonomous mechanism.

The statistics presented in [6] show that in 2015 the global mobile data traffic grew by 74% with respect to the end of 2014. Global mobile data traffic reached 3.7 Exabytes per month at the end of 2015, up from 2.1 Exabytes per month at the end of 2014. Moreover, the above mentioned forecast estimates that in 2020, the total amount of mobile traffic per month will be almost 30.6 Exabytes, an eightfold increase over 2015.

The emerging increase in data traffic has been one of the most concerning threats for mobile operators in the past years due to the multitude of bandwidth hungry consuming applications, especially in the area of real-time entertainment and video streaming. The exponential data growth triggered their interest to look beyond 3GPP proposals, which usually imply over-provisioning the network under high costs within long standardization periods. Recently, standardization bodies such as Network Function Virtualization (NFV) Industry Specification Group (ISG) under the European Telecommunications Standards Insitute

(ETSI) were introduced in order to focus operators' demands based on cost effective solutions, which can also bring deployment flexibility in the network architecture under very short time-to-market releases. The ETSI committee provided the big operators and vendors with several Proof of Concepts [7], in the hope that they could employ NFV concepts and techniques inside their production networks.

The NFV technology allows operators to cope with network flexibility and scalability issues, providing an improvement over the deployment of network function operations. However, the very first thing to explain about NFV should be its actual technical definition. What do network operators want to achieve by virtualizing network functions? Basically, a core network node is a piece of hardware that has some software running on it, and the NFV's purpose is to abstract that software out of the hardware itself and put that logic (software) typically on a server. Therefore the virtualized network functions, which are typically embedded in telecom network hardware, are now software available and capable of running on a virtual machine (VM) under the control of a hypervisor (e.g., vSphere, KVM, XEN). Applying NFV to mobile networks, 3G and 4G packet core components and IMS are the first to be virtualized. They are very software-oriented in the first place.

The NFV came to light at the OpenFlow / SDN World Congress in Darmstadt, Germany in October 2012, when seven of the largest operators (i.e. Deutsche Telekom, Orange, British Telecom, Telecom Italia, Telefonica, Verizon and China Mobile) published a white paper [33] on NFV and then ETSI board approved the creation of the NFV Industry Specification Group in November 2012.

In contrast to NFV, both OpenFlow protocol and SDN concept emerged in an academic environment back in 2008 as a research project. Several vendors and large enterprises started to produce the equipment and firstly implement SDN in 2011. Google, for example, built its own SDN switches and was the first to adopt a global software-driven network [30]. Cisco and Brocade, for example, entered the SDN world by producing OpenFlow-enabled hardware equipment. In the same year, 2011, it was decided to form Open Networking Foundation (ONF), and organization with the purpose of standardizing SDN in different group areas.

SDN is a new technology in the industry designed to make networks more agile. Today's networks are quite often "static", slow to change and dedicated to single services. SDN makes it possible to create many different services in a dynamic fashion, allowing to consol-

idate multiple services onto one common infrastructure. From a technological standpoint, SDN is a new network architecture paradigm made of three different layers. The bottom layer is the ***infrastructure layer***, where the network forwarding equipment is. Unlike today's architectures, that forwarding equipment relies on a new layer, the ***control layer***, to provide it with its configuration and its forwarding instructions. The middle layer, or control layer, is responsible for configuring the infrastructure layer, by receiving service requests from a third layer, the ***application layer***. The control layer maps the service requests onto the infrastructure layer in the most optimal manner possible, dynamically configuring the infrastructure layer. The third (application) layer is where cloud, management and business applications place their demands for the network resources onto the controller layer.
In SDN, each of these layers and their API among them are designed to be open. Thus, it is possible to have multiple vendor's equipment at the infrastructure layer, multiple vendors control plane components in the control layer, and multiple vendors applications at the application layer. However, the key concept of SDN is that it decouples the data / forwarding plane from the control plane. This means that the network hardware equipment is not tied to its software counterpart, the operating system. Therefore, the hardware equipment becomes just a generic network component, also called *forwarding element*, and the logic is moved onto a central server, also called *control element*. The network operating system becomes a control node deployed on a single physical hardware which instructs, in a central fashion as it plays the role of a server, a multitude of forwarding elements about how to forward the traffic through them.

In the next years, network operators will have to adopt a strategy in the evolution of their cellular core networks, as a transition towards 5G. The most conservative strategy is to preserve the current 3GPP protocols, with arguably successful deployment and limited capabilities to cope with the market demands. A more attractive and interesting solution would consist of the aforementioned technologies, NFV and SDN, which might allow operators to deploy more services in a more flexible manner with less costs and resources, assuring a long-term solution compared to the current 3GPP shortages. The main drawback for network operators to adopt such a transition technique is that it implies important changes in their architecture and a complete and permanent removal of some legacy hardware.

1.2 Problem Statement

Nowadays, the LTE / EPC architecture experiences a period of rapid and massive changes. An extensive network scaling is required as it connects more devices, applications and network traffic than ever. The current network architecture lacks service elasticity and dynamic features deployment on-demand. The reason is that the EPC's entities (i.e. Mobility Management Entity (MME), Serving Gateway (S-GW), Packet Data Network Gateway (P-GW), Home Subscriber Server (HSS), and Policy and Charging Rules Function (PCRF)) are based on custom specific hardware and need to be statically provisioned and configured. Therefore, a strategic deployment of new entities in specific network locations is normally required to get an increase of the network capacity. Such a network results in having a very static behaviour. Its operation costs, therefore, imply a huge amount of manual work (i.e. manpower).

There are two types of technologies which could be potentially exploited to improve a today's traditional cellular network architecture and, why not, infrastructure. And these are represented by NFV and SDN paradigms. One of the thesis goals is to investigate how SDN could shape the transition of today's traditional EPC towards an SDN-based user plane architecture. In order to be able to adopt such technologies, network operators need to make room for changes upon several studies and experiments on different use case scenarios. A full understanding of SDN and NFV benefits, along with their drawbacks, can provide enough proofs of the technological advances for specific needs, besides a careful focus on the required economic investment. The main issues in current EPC deployment are the high costs, the real time load balancing and the Quality of Service (QoS) control over different network flows and potentially service-level agreement (SLA) contracts.

To support the ever-growing demands for mobile traffic, operators try to provide more capacity and over-dimension their network size which leads to a huge increase in CAPEX and OPEX. The most expensive element is the telecom infrastructure, though a key strategy to reduce the costs is the virtualization of the legacy hardware, when possible. Virtualization allows to have multiple functions running on a single hardware platform and provides the required capacity in a flexible fashion. Carriers need to periodically upgrade their infrastructure in order to keep up with the fast technology evolution. The costly upgrades and legacy hardware can be easily replaced by virtualized systems, enabling the reuse of legacy software on newer hardware.

The SDN technology also provides better cost reduction in terms of the used hardware equipment and extra network "intelligence". For instance, the replacement of mobile gateways with SDN capable hardware might save a remarkable amount of economic resources. Though, the SDN's most attractive benefit is not mainly the cost reduction, but the ability of performing routing, controlling the load, resiliency and robustness in a smart way.

As an example, the current load balancing mechanisms in the mobile core network are done in a proactive manner, without considering the instantaneous load or capacity at the gateways, which might lead to overload network nodes of a cluster, while there is still remaining capacity on other nodes of the same cluster. The GPRS Tunneling Protocol Control Plane (GTP-C) load balancing technique based on Dynamic DNS updates as proposed in [1] requires a new standard interface for configuration and implies modifications in the operator's DNS infrastructure, which is very sensitive for operators. Although 3GPP standards do not specify how many Serving/Packet Data Network Gateways (S/PGW) should be deployed in LTE core network, in practice, the core network usually uses centralized and integrated S/PGW, which serves a large area of Evolved Node B (eNodeB)s. This results in intense back and forth IP traffic flowing through S/PGW.

By introducing an SDN control layer in the EPC infrastructure, the implementation of the user plane entities (S-GW and P-GW) as simple forwarding elements will bring more flexibility and visibility over the forwarded IP traffic, allowing user service optimizations (source [17]). This approach might be feasible to reduce the aforementioned back-and-forth traffic. The proposed solution will lead to a distributed S/PGW in the data plane with a centralized controller that uses SDN / OpenFlow to dynamically manage those S/PGW local/mobile IP traffic.

The radio spectrum is finite and does not provide too many options, thus, network over-provisioning to increase network capacity will lead to high costs and more complex deployments, affecting the network robustness and flexibility.

1.3 Related Work

One of the biggest challenges for mobile operators is how to efficiently integrate SDN and NFV technologies into the their current legacy hardware without compromising the network infrastructure and services. The recent ETSI NFV standardization body has defined several use cases for a virtualized EPC in [8]. This paper suggests the application of EPC deployments in both full or partial fashion along with the coexistence of virtualized functions and legacy entities. On the other hand, proposals to integrate the SDN architecture in the mobile core have been carried under the ONF Wireless & Mobile Group [21].

The academic research has performed several studies and made efforts in investigating both control and user plane of various mobile core network entities in order to find out how to optimize the existing mobile architecture given the certainty of a data exponential increase which impacts on the corresponding control traffic. Kempf *et al.* [18], for example, implemented a strategy of the EPC's evolution moving more towards a cloud-based fashion. Specifically, shifting the entire control plane of core network entities, such as S-GW, P-GW, and MME, into the cloud. After which, OpenFlow switches are used for implementing the corresponding user plane, extending them with GTP support in a native fashion. OpenFlow 1.2 is used for implementing two vendor extensions, one defining virtual ports to allow encapsulation and decapsulation and another one to allow flow routing using the GTP Tunnel Endpoint Identifier (TEID). The result enables an architecture where the GTP control plane can be extracted out of network elements such as the S-GW and the P-GW and moved into a central controller running in a VM in a data center.

Another paper [12] suggests a new encapsulation / decapsulation technique on top of IP referred to as "vertical forwarding" implemented in the Forwarding Elements (FE) and managed by a central controller. The encapsulation and decapsulation operations are delegated to a dedicated Network Interface Card (NIC) capable of recognizing GTP traffic. However, this solution does not seem to offer enough elasticity as the GTP traffic recognition is deferred to a dedicated hardware instead of an SDN GTP-enabled solution.

Further publications, like [28], highlight the today's cellular data networks lack of flexibility, which leads to consider several different approaches, most of them are SDN-based solutions. Specifically, they focus on the fact that the today's typical cellular network technologies static behaviour does not allow sufficient visibility and control elasticity over their networks. Hence, lacking a dynamic provision of on-demand connectivity services. The SDN is an emerging trend that should be considered to overcome the above drawbacks. Their proposed solution is based on an OpenFlow-based control plane for LTE / EPC architectures.

A relevant area of interest for the SDN research is the support for QoS routing. An interesting paper [14] focuses on the QoS mechanisms using an SDN controller. The authors propose a new OpenFlow controller, called "OpenQoS", designed for multimedia delivery with end-to-end QoS support. Their strategy is to exploit QoS routing so that the multimedia traffic routes could cover the required QoS being optimized with a dynamic approach. This optimization is accomplished with a separation of the data traffic from the multimedia flows, which are sent throughout QoS prioritized routes, while the data traffic follows the path computed by the Shortest-Path First (SPF) algorithm.

1.4 Contributions

The project focuses on the extension of the Open vSwitch, a software switch open source project, in order to make it aware of the GTP-User Plane (GTP-U) [10] tunnel traffic. Implementing GTP in a virtual / software switch offers the freedom of using general-purpose hardware as a way to embody an EPC network node, specifically the S-GW and P-GW which are the user plane forwarding entities. GTP is an IP-based tunneling protocol used to tunnel GPRS data within 2G (i.e. GSM), 3G (i.e. UMTS) and 4G (i.e. LTE) networks. As GTP carries three different types of traffic, which have three different scopes, its functions are split into three separate sub-protocols: GTP-C, GTP-U and GTP'.

GTP-C is used for carrying control plane traffic within the mobile core network entities, such as between P-GW and S-GW, as far as EPC is concerned.

GTP-U's purpose is to establish user plane tunnel sessions used to transport user data within the GPRS core network. The user data traffic carried inside a GTP-U tunnel is represented by any of IPv4, IPv6, or PPP data packets.

GTP' (GTP prime) has the same message structure of GTP-C and GTP-U, but is used for carrying charging data traffic only.

The purpose of the thesis is focused on the implementation of the GTP-U capability for Open vSwitch, to identify and define the main GTP tunnel parameters in order to forward / route user plane data traffic.

1.5 Structure of the thesis

The thesis outlines and further analyzes the main issues introduced, starting from a more detailed description of the key technologies involved in the realization of the project. Then an introduction to the context is presented, followed by a proposed system design, description of the actual implementation and measurement analysis. Hereby, the thesis is structured in five main chapters as follows:

Chapter 2 briefly describes the today's standardized EPC architecture, its key operations and main protocols. New technologies like NFV and SDN are discussed, reflecting on how their key strengths could be exploited in a cellular data network environment, like LTE. The OpenFlow protocol as well as the Open vSwitch architecture are briefly presented.

Chapter 3 analyzes and suggests an evolution design for a mobile network upon an SDN-based framework, at a user plane level.

Chapter 4 goes through the implementation steps of the Open vSwitch initialization process, describing some of its core functionalities, related computational problems, such as *Packet Classification*, and their implementation in Open vSwitch. At last, the actual GTP-U implementation in Open vSwitch is introduced in detail.

Chapter 5 consists of evaluation of the adopted GTP-U implementation, followed by a comparison with the performance obtained by simply generating user data traffic without involving any Open vSwitch processing at all.

Chapter 6 presents the final remarks and conclusions of the presented work. Some complementary future work are also discussed.

2 Background on Network Function Virtualization (NFV) and Software-Defined Networking (SDN) concepts for Evolved Packet Core (EPC) networks

This chapter firstly describes what the EPC architecture is comprised of, then a brief introduction about its main core entities is presented with the purpose of having a basic knowledge of which network nodes and which of their operations could be exploited by applying NFV and SDN techniques. At last, some of the NFV and SDN key technology concepts are described, so that the reader would have a clue of the potential that these technologies could bring on a modern mobile core network.

2.1 Evolved Packet Core (EPC)

The Evolved Packet System (EPS), as shown in Figure 2.1, is made of three key components:

- Radio access network, or enhanced Universal Terrestrial Radio Access Network (e-UTRAN) in 4G terminology;
- Evolved Packet Core (EPC), which is the 4G core network;
- Packet Data Networks that the entire system interfaces to, such as the Internet and/or IP Multimedia Subsystem (IMS) operated typically by the carrier.

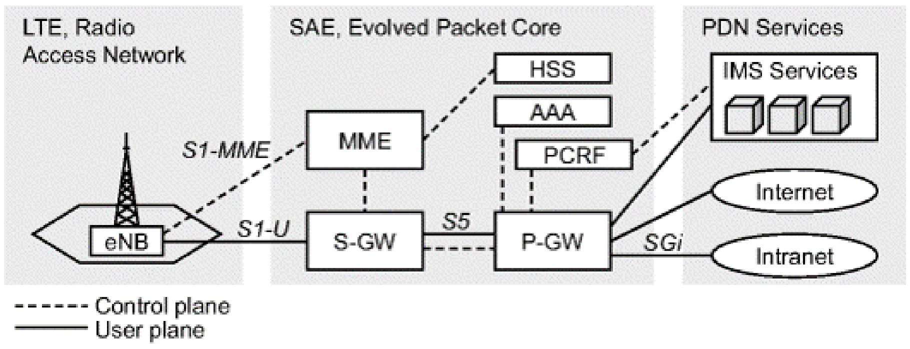


Figure 2.1: The EPS main entities, source [20]

As we can see, the EPC is the fundamental part of the whole system, as it lays in between the UEs requesting services and external Packet Data Networks (PDN) where these requested services are located. It is mainly responsible for the control of the UE and the establishment of bearers. An *EPS bearer* [9] can be thought of as a bi-directional data pipe, which transfers data on the correct route through the network.

2.1.1 Serving Gateway (S-GW)

The *serving gateway* (S-GW) is the core network entity that communicates user data to and from the eNodeBs, also abbreviated as eNB, using the S1-U interface. One S-GW may be connected to multiple eNBs, so as the eNB which may be connected to multiple S-GWs.

This node entity is used as a local mobility anchor point, for which a handover results in switching the user data path to the new elected eNB. S-GWs are also controlled by one or more MMEs (later described).

If the S-GW serving a certain user needs to be changed when a handover occurs, the corresponding MME replaces the current old S-GW with a new one and let the old one know when it can be released. When a UE is in idle mode, the S-GW buffers the downlink data, i.e. data destined to the UE, until it gets active and ready to receive them.

2.1.2 PDN Gateway (P-GW)

The *packet data network gateway* (P-GW) is the core network node that enables the forwarding of user data traffic to and from external packet data networks (PDN). The EPC network usually includes one P-GW for each type of external packet data network connection (e.g. one P-GW for the Internet, one P-GW for the IMS, etc.).

The P-GW is the entity that implements the allocation of IP addresses needed for the UEs.

It also maintains the <specific mobile ID (i.e. the IMEI), assigned IP address> pairing correspondence while the UE remains attached to the network.

In most LTE networks the GPRS Tunneling Protocol (GTP) is used for S5/S8 interface. In these cases, the QoS and EPS bearer management is done at this node entity.

2.1.3 Mobility Management Entity (MME)

The MME is the key control element within the EPC. The EPC network may include multiple MMEs. These are selected based on their geographical location, basically how far they are positioned from the UE, and according to their current load. After being selected, the MME chooses other network entities required for the accomplishment of the user service, such as S-GW and P-GW.

2.1.4 Home Subscriber Server (HSS)

This is the main subscriber database within the EPC network. For many aspects it resembles the Home Location Register (HLR) present in previous 3GPP networks.

This core network node implements a database storing permanent subscription details for each subscriber in the network, including the authentication key. It operates closely with the MME, obtaining from it updates for each subscriber the MME serves and storing them as temporary data. During the authentication procedures, the HSS provides the MME with authentication data based on the authentication key, which never leaves the HSS.

2.1.5 Bearer Management

The EPS Bearer

In order to manage the delivery of the user data packets in a differentiated manner, the packets are grouped into so-called *EPS bearers*. This bearer is able to identify packet flows that receive a common quality of service (QoS). A bearer may include packets of multiple services and is defined by combination of the QCI and destination IP address. There are two types of bearers:

- **Default Bearer**, which is setup when the UE attaches to the network. It provides basic connectivity and is maintained as long as the UE does not change its assigned IP address.

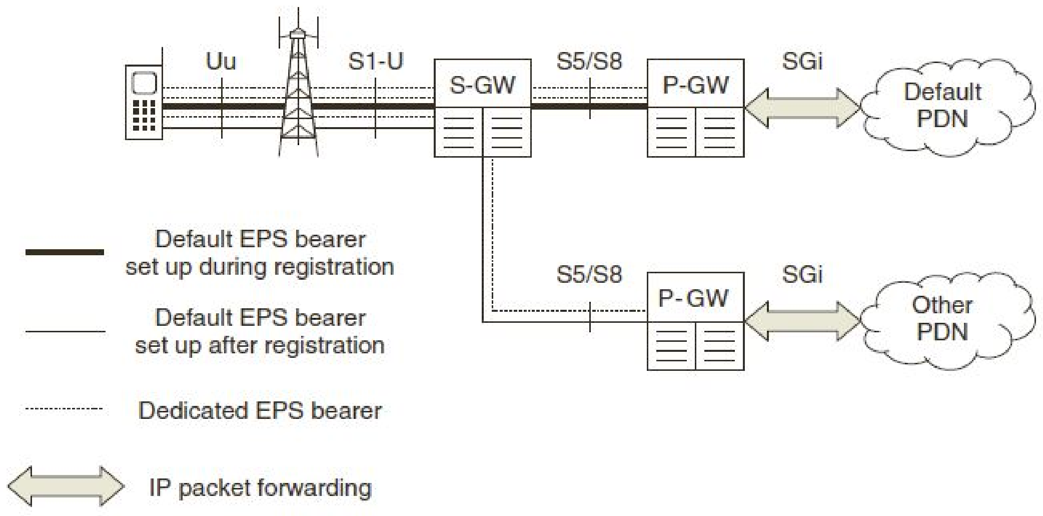


Figure 2.2: Default and dedicated EPS bearers, when using a GTP-based S5/S8 interface. Source [34]

- **Dedicated Bearer**, is setup for QoS sensitive applications. The Policy and Charging Rules Function (PCRF) network node determines which packet flows are mapped onto the dedicated bearers.

Figure 2.2 is an illustration of how a mobile receives a default bearer as soon as it registers with the EPC to provide it with always-on connectivity to a default packet data network.

Bearer Implementation Using GTP

In case of a GTP-based S5/S8 interface, the EPS bearer extends over three different interfaces. The EPS is split out into three lower-level bearers (Figure 2.3), namely the *radio bearer*, the

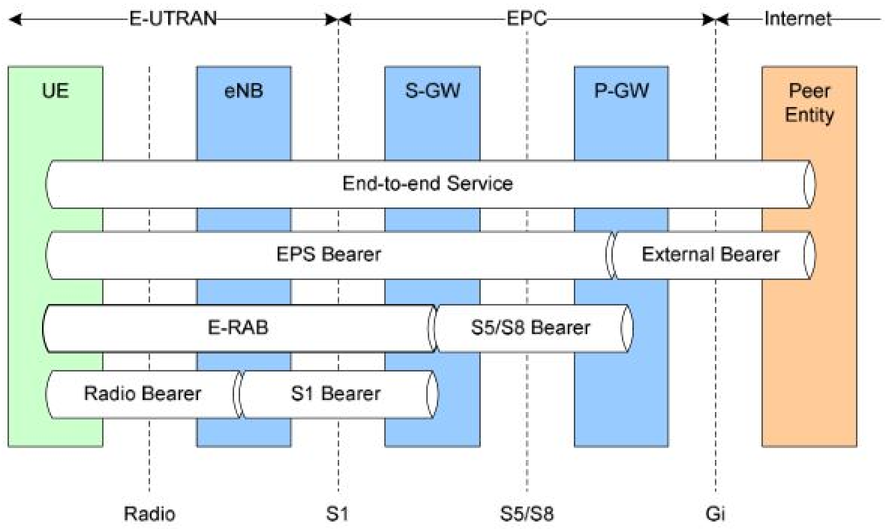


Figure 2.3: EPS Bearer Service Architecture, source [11]

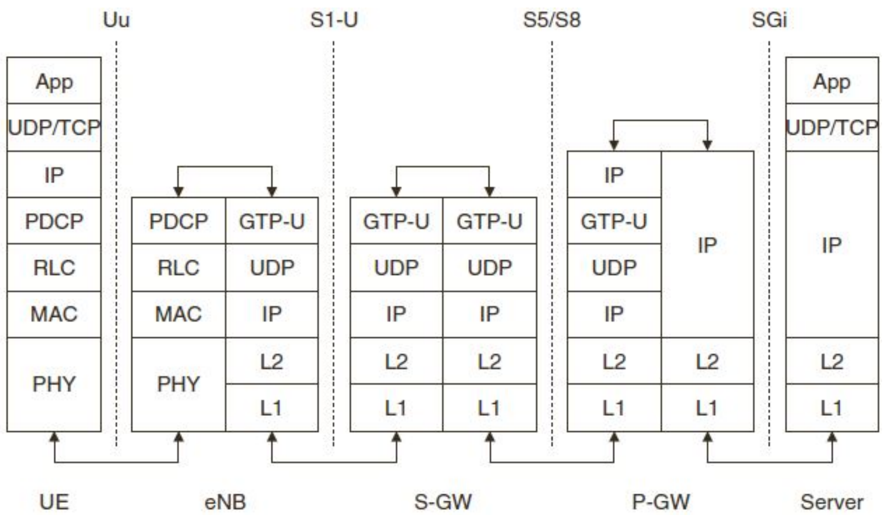


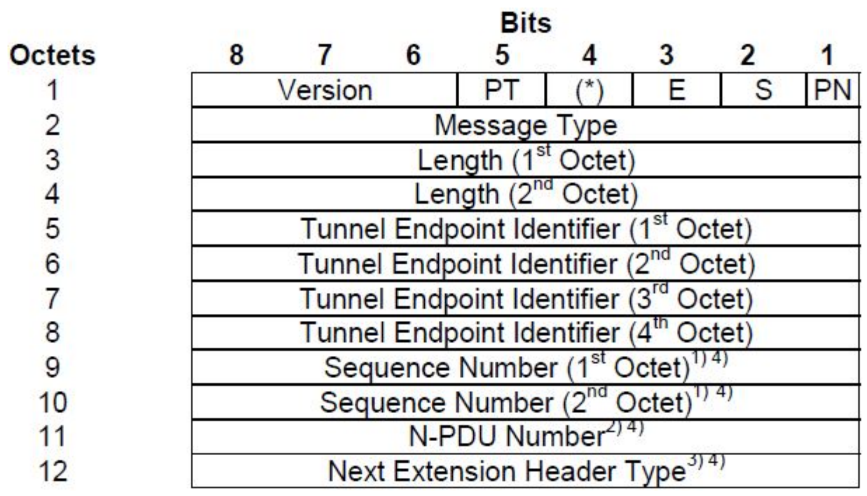
Figure 2.4: Protocol stacks used to exchange data between the UE and an external server, when a GTP based S5/S8 interface is used. Source [9]

S1 bearer and the *S5/S8 bearer*.

The radio bearer is implemented by a proper configuration of the air interface protocols, while the S1 and S5/S8 bearers are implemented using GTP user plane (GTP-U) tunnels. The resulting protocol stacks are shown in Figure 2.4. For a basic understanding of how they operate, let us consider the case when a packet is sent through the downlink path, sourced by a server located somewhere in the Internet, and destined to the UE. The IP packet the server generates has set the destination address as the UE's IP address. When the packet is sent, it first hits the P-GW node, which depending on where it is destined, the target UE is identified and a second, i.e. outer, IP header is added. This outer IP header is built with the P-GW set as the source IP address and the UE's S-GW set as the destination address. The P-GW's network stack then uses the S-GW's IP address to route the packet there. The S-GW, in turn, applies the same logic to send the packet to the proper eNodeB, and from there the packet will finally reach its target UE.

The packet forwarding process is performed by the GTP user plane (GTP-U) protocol, which labels each packet using a 32 bit *tunnel endpoint identifier* (TEID, see Figure 2.5) that in turn identifies the overlying S1 (eNB – S-GW) or S5/S8 (S-GW – P-GW) bearer. By fetching the TEID, the network can distinguish packets that belong to different bearers and can handle them using different qualities of service.

GTP is the main protocol within the EPC network, using mainly UDP at the transport level. Its key feature is to place a small header before the user IP packet. The most important field in the GTP header is represented by the tunnel key identifier, the TEID, enabling an



Octets	Bits 8	7	6	5	4	3	2	1
1	Version			PT	(*)	E	S	PN
2	Message Type							
3	Length (1st Octet)							
4	Length (2nd Octet)							
5	Tunnel Endpoint Identifier (1st Octet)							
6	Tunnel Endpoint Identifier (2nd Octet)							
7	Tunnel Endpoint Identifier (3rd Octet)							
8	Tunnel Endpoint Identifier (4th Octet)							
9	Sequence Number (1st Octet) 1) 4)							
10	Sequence Number (2nd Octet) 1) 4)							
11	N-PDU Number 2) 4)							
12	Next Extension Header Type 3) 4)							

Figure 2.5: GTP-U (or GTPv1) header, source [10]

unique style of identification of the specific GTP tunnel for the given direction (i.e. either uplink or downlink). This means that for a given (S-GW, P-GW) pair, an uplink GTP tunnel is different from a downlink GTP tunnel for that same S/P-GW pair. The main reason of using GTP tunnels within the EPC is to provide the mobiles with IP mobility when they move around. In other words, the IP address assigned to the UE remains the same even if the UE moves from one place to another, so the packets destined to the specific mobile are still forwarded and correctly received.

As previously mentioned, the GTP protocol has been standardized under three different types, each one carrying a different type of traffic:

1. **GPRS Tunneling Protocol Control Plane (GTP-C) (GTP version 2 (GTPv2))** is a control plane version of the GTP protocol which adds support for establishing tunnels required for mobility and QoS management. Uses UDP at the transport level and is mainly employed for carrying signaling traffic within the EPC.

2. **GPRS Tunneling Protocol User Plane (GTP-U) (GTP version 1 (GTPv1))** is the user plane version of the GTP protocol. It is used within the EPC to carry user data packets inside GTP tunnels. The EPC's network nodes operating as GTP routers, which enable the routing of the user data traffic back and forth between the UE and the Internet, are represented by the S-GW, P-GW and by the eNodeB when encapsulating in the uplink direction and decapsulating in the downlink direction.

3. **GPRS Tunneling Protocol Prime (GTP')** has the same message structure as of GTP-C, but is used for carrying charging data.

A key fact to understand is that the core network entity which assigns an IP address to the mobile is the P-GW: therefore, due to the tunneling, the UE sees the P-GW as its first IP hop, meaning that the P-GW embodies the default gateway functionality for the UEs.

2.2 Network Function Virtualization (NFV)

When asking "why virtualization?" we really need to look at what operator's challenges are. One of the challenges is that of the massive data demand. The use of mobile data is growing rapidly as it is well known, as well as the number of devices continues to grow, these devices being lead by the typical smartphone devices. The "call model" associated with these devices is also changing. Thus, the all predictable GSM and UMTS call model is already outdated.

As new services are being deployed by VoLTE and/or small cells, the call model is changing dramatically. An example would be when small cells are deployed, dramatically increasing the number of UEs or S1 connections that are required on the mobile core network, which changes the call model. All this leads to definitely unpredictable changes in signaling and data demand. Therefore the question now is what the new options and solutions are. One of the newest and most promising option is a fully functional, optimized, carrier-grade virtualized EPC solution.

A big problem for today's network operators is that of being populated with large hardware appliances, this means they require a large hardware equipment for the number of users so they can meet their growing demands. In order to launch the new services, the network operator requires: hardware variety, space for the new hardware, and enough power supply for the appliances. The three requirements have to be met in order to successfully accommodate all the new "boxes". Given the ever increasing user demand, this process becomes very difficult to achieve, it comes with an enormous amount of capital expenditure and assumes high maintenance costs. This "obsolete" process, as of today, has the following drawbacks:

- Increased cost of energy.
- Huge investments.
- Skills to design.
- Operation of complex hardware designs.

This is where NFV comes into play, it aims to address all of these issues. NFV is a virtualization technology technique which makes possible to consolidate many hardware equipment types, like servers, switches and storage. It is actually a subset of the already mentioned SDN; it decouples the hardware from the software, therefore moving network functions from specialized appliances to applications that run on commercial off-the-shelf (COTS) equipment. The fixed-defined functional behaviour of a component in a network infrastructure could potentially be considered as a *network function*, some examples could include an intrusion detection and prevention system, routing logic, etc..

Virtual Network Function (VNF) means implementing a network function employing software strictly decoupled from its hardware equipment counterpart. Such approach improves the network's flexibility leading to a more agile network, which further causes important OPEX and CAPEX savings. The VNF represents an extracted network function to be implemented in software, with the purpose of being installed and run over the NFV infrastructure (NFVI). The NFV Management and Orchestration controls the physical and software resources that support the NFVI and the lifecycle management of VNFs. It allows both horizontal and vertical scalability of the resources based on capacity demands.

2.2.1 NFV architecture

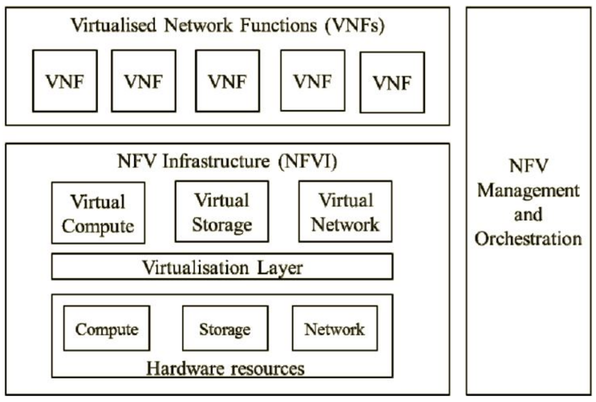


Figure 2.6: NFV architecture, source [8]

Figure 2.6 illustrates how the NFV architecture is organized and what it is comprised of. In this section we will first describe the high level NFV framework, and then explore each component in detail.

The NFV framework consists of four components: the NFV infrastructure (NFVI), Virtualized Network Functions (VNFs), the Management and Network Orchestration unit, and the

OSS/BSS layer sitting on top of VNFs.

NFV infrastructure (NFVI)

The *NFV infrastructure* is a kind of cloud data center containing hardware and virtual resources that build up the NFV environment, these include servers, switches, virtual machines and virtual switches.

The first part of the NFVI is the Hardware Resources, these include: computing resources such as servers and/or RAM, storage resources such as disk storage and/or NAS, and network resources such as switches, firewalls and routers.

The second part of the NFVI is the Virtualization Layer. The purpose of such layer is to perform an abstraction over the hardware resources, thus decoupling software from hardware components. This enables the software to progress independently from the hardware. For the Virtualization Layer we can use multiple open source and proprietary options such as KVM, QEMU, VMware, OpenStack.

The second part is the Virtualized Resources, these include virtual compute, virtual storage and virtual network.

Virtualized Network Functions (VNFs)

The VNFs are basic building blocks in the NFV architecture. They are software implementation of network functions. A set of connected VNFs is capable of offering a full scale network communication service, this is known as *service chaining*. Some of today's VNF examples include vIMS, vFirewall, vRouter.

Management and Network Orchestration unit (MANO)

MANO has three parts: Virtualized Infrastructure Manager, VNF Manager, and Orchestrator.

The Virtualized Infrastructure Manager controls and manages the interaction of the VNF with the NFVI compute, storage and network resources. It also has necessary deployment and monitoring tools for the Virtualization Layer.

The VNF Manager manages the lifecycle of VNF instances. It is responsible to: initialize, update, query, scale, and terminate VNF instances.

The last part of the NFV MANO is known as the Orchestrator. The Orchestrator manages the lifecycle of network services which includes: instantiation, policy management, performance management, and KPI monitoring.

OSS/BSS Layer

It is responsible for all the remaining management operations, which includes system monitoring, configuration, and billing procedures.

2.2.2 Benefits of NFV

Some of the NFV's main benefits are: lower equipment costs and reduced power consumption through virtualization. Apart from these, other benefits have been further identified which can be directly mapped to the EPC applied use case:

- Increased velocity of time-to-market by minimizing the typical network operator cycle of innovation.
- Network operators can share resources, using them in a more effective way.
- Lower risks in case of hardware failures.

Network configurations and topologies can be potentially optimized, depending on the type of service demanded and real-time traffic and mobility patterns, all of that by employing NFV technology concepts.

2.3 Software-Defined Networking (SDN)

Briefly mentioned in the previous sections, it was introduced with the growth of virtualization technologies, allowing to use networking devices as normal software components. SDN is an architecture paradigm that decouples data plane from the control plane. Up until now, the routing/forwarding tables have been computed locally, in a distributed way, by each router or switch equipment. In the SDN paradigm, the computations of the control plane operations are performed by a very specific device, called *controller*. Before taking a closer look at this SDN architecture, let us examine the reasons for this new paradigm.

The limitations of traditional network architectures are becoming significant: as of today, modern networks have a difficult time with optimizing the costs (i.e. the CAPEX and OPEX).

In addition, the networks are not agile. The time to market is way too long, and the provisioning techniques are not fast enough. All of this resulting for the networks to be pretty much unconnected to the services.

2.3.1 The objective

The purpose of SDN is to reduce costs by virtualizing and automating the traditional network operations as much as possible. The SDN architecture can be thoroughly described by introducing three key principles. The first one is defined by the separation of the physical (hardware) and virtual (software) components. This concept allows the storing of software devices onto the hardware equipment. Therefore the network becomes hardware-independent. The second principle specifies that the terminal devices should not distinguish any difference between a hardware or software entity. This results in a new setting allowing you to change the network without having anything to do with the host machines. The third and the last principle mentions the automation of the network operations, this usually implies greater elasticity.

2.3.2 ONF architecture

In order for this new world of SDN to have a chance of being successful, it has to be standardized. This standardization was carried out by the Open Network Foundation (ONF), which was set up under the sponsorship of large companies in California (USA), following the proposal of this new architecture by Stanford University and Nicira.

The architecture proposed by the ONF is shown in Figure 2.7. It comprises three layers. The bottom layer is an abstraction layer, which decouples the hardware from the software, and is responsible for packet forwarding. This level describes the protocols and algorithms which enable IP packets to advance through the network to their destination. This is called the infrastructure plane. The second layer is the control plane. This plane contains the controllers providing control data to the data plane so that the data traffic gets forwarded as effectively as possible. The ONF's vision is to centralize control in order to facilitate the recovery of a great deal of information on all the clients. The centralized controller enables obtaining a sort of intelligence. The infrastructure to be managed is distributed between the controllers.

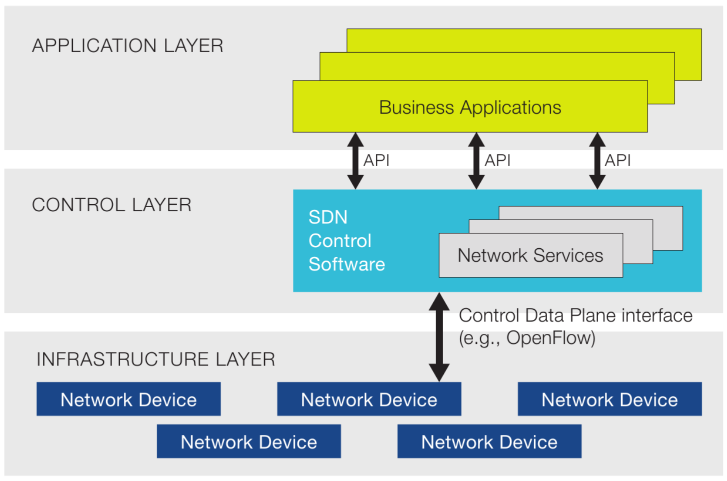


Figure 2.7: The ONF's SDN architecture, source [29]

Controllers carry out different functions, such as the provision of infrastructure, the distribution of loads on different network devices to optimize performances or reduce energy consumption. The controller is also in charge of the deployment of firewalls and servers necessary for the proper operation of the network and a management system. These different machines must be put in the most appropriate places.

Finally, the uppermost layer, the application plane, is responsible for the applications needed by the clients and for their requirements in terms of networking, storage, computation, security and management. This layer introduces the programmability of the applications, and sends the controller all of the necessary elements to open the software networks suited to the needs of the applications. Any new service can be introduced quickly, and will give rise to a specific network if it cannot be embedded on a pre-existing network.

So to summarize, the ONF architecture shown in Figure 2.7 is comprised of three layers: the application layer and programmability, the control layer with centralized intelligence, and abstraction at the infrastructure layer.

The ONF was formed for the purpose of standardizing a protocol between the controllers and the network devices in the infrastructure layer. This protocol is called *OpenFlow* and it runs on the *Southbound* interface, offering a means of communication between the OpenFlow network devices and the corresponding control layers.

2.3.3 OpenFlow

OpenFlow is the standard protocol used to communicate control data between SDN controllers and SDN-based switch components. This is the reason why a basic knowledge about OpenFlow is necessary for this thesis' topic. At the time of writing, the most recent OpenFlow standard is 1.5.1, thoroughly described in the ONF's OpenFlow specification [23]. This subsection serves as a brief explanation of the OpenFlow core logic, for a more complete description please refere to the 1.5.1 specification which can be found at the ONF's website https://www.opennetworking.org/.

The SDN-based switch represents the implementation of an abstract packet processing machine. Such a switch processes packets in a whole different way if compared to a normal traditional today's switch operations, as it keeps track of an internal dynamic configuration state in combination with a packet headers parser.

An OpenFlow controller is defined by a centralized server which speaks OpenFlow with OpenFlow-enabled switches, dictating their specific behaviour that should be performed when encountering certain packet flows. Therefore, the purpose of the OpenFlow controller is to update the dynamic configuration state of the connected OpenFlow switches. The updating type and rate is application-specific, it is actually the application implemented in the OpenFlow controller that decides what behaviour to update, in which OpenFlow switch node, and when.

Before delving into the details of how the OpenFlow engine works, let us see what the main components forming an OpenFlow switch are.

Switch Components

An OpenFlow logical switch, as shown in Figure 2.8, implements one or more *flow tables* and a *group table*. The packet lookups and their corresponding actions are performed based on the flow tables specific content. An OpenFlow channel is also implemented in the OpenFlow switch allowing it to connect to external controllers.

Being able to "speak" OpenFlow, the controller's operation is to add, update, and delete so-called *flow entries* which reside inside the flow tables. Each flow table consists of a set of flow entries, and each flow entry is comprised of match fields, counters, and a set of instructions which are to be applied on packets that match the given match fields.

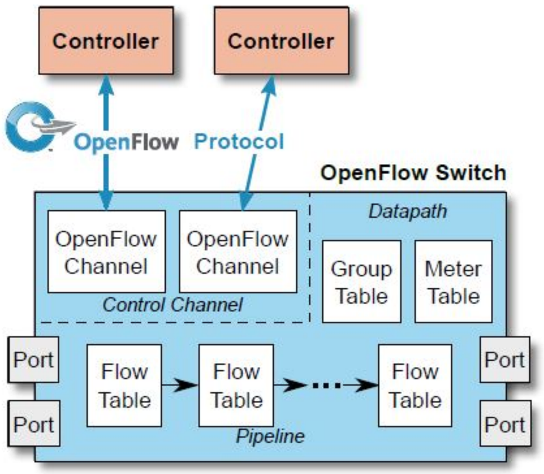


Figure 2.8: Main components of an OpenFlow switch, source [23]

The process of matching packets starts at the very first flow table and may continue to further flow tables present in the pipeline. The packets are matched against flow entries, whereas the selection of which flow entries to choose in which order depends entirely on their priority, therefore flow entries with higher priority are chosen first. If a flow entry is found to match the packet in the pipeline, the flow entry's corresponding instructions are fetched and applied to the packet. In case a flow entry is not found to match the packet in a given flow table, the so-called *table-miss* flow entry is selected as the default instruction set to be applied on the triggering packet. This *table-miss* flow entry may specify any action supported by OpenFlow, like forwarding the match-missing packet to an OpenFlow controller, sending it to one of the next flow tables in the pipeline, or even dropping it.

Thus, each flow entry specifies a corresponding instruction set which has to be executed whenever it encounters a packet that matches its matching part. The instruction set may contain supported OpenFlow actions or may modify the order the packets traverse the flow tables in the pipeline. The OpenFlow actions may dictate where the packet should be forwarded and/or editing some packet's header fields. The latter can be thought of some sort of DPI (Deep Packet Inspection).

At the moment of this writing, the hardware equipment industry implements two types of OpenFlow-enabled switches: *OpenFlow-only*, and *OpenFlow-hybrid*. An OpenFlow-only switch, as the name suggests, implements only the OpenFlow packet processing engine, and therefore all the packets entering such type of switch are being processed by the OpenFlow engine only. The other type of switch equipment (OpenFlow-hybrid), on the other hand, has the support of both OpenFlow engine and normal Ethernet switch processing engine. These

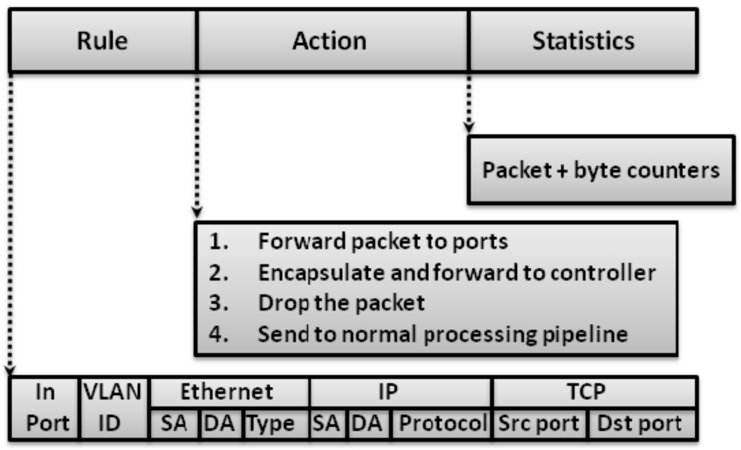


Figure 2.9: The OpenFlow v1.0 flow entry in a flow table, source [22]

type of switches usually provide a way to specify the type of traffic that should be processed by either the OpenFlow engine or the normal Ethernet engine.

Every OpenFlow switch implements an OpenFlow engine, which in turn implements one or more flow tables, and each flow table is represented by multiple flow entries.

In a flow table, the packet is matched against its flow entries so that a matching entry is selected and its associated action set executed. If instructed by this action set, the packet may be sent to any subsequent flow table, repeating the same process. The packet is not sent to any other subsequent flow table whenever the matching entry in the current table does not explicitly say so, in this case the packet is usually either forwarded or dropped, depending on the specified OpenFlow actions.

When a packet does not match a flow entry in a table, this case is called **table miss**. This corresponds to a table state configuration specifying a default action set to be performed on unmatched packets. The default action set contains normal OpenFlow supported actions, such as sending the unmatched packet to a subsequent table in the pipeline, dropping or encapsulating it in an OpenFlow *packet-in* message in order to be sent to an OpenFlow controller and further processed by an OpenFlow application executing on top of the controller itself. The notion of flow entry in OpenFlow 1.0 was somewhat different from the today's flow entry concept (at the time of writing the OpenFlow protocol version is at 1.5.1). We can see this by analyzing the differences between the Figure 2.9 and Table 2.1.

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags

Table 2.1: Main components of an OpenFlow v1.5.1 flow entry in a flow table, source [23]

Follows a brief description list of the most important entries in a typical OpenFlow 1.5.1 flow entry, whereas Table 2.1 is a full representation with all of its components.

- **match fields**: set of fields to match against packets. Comprised of metadata information, such as the ingress port number, and packet header fields.
- **priority**: entries with high values are selected first in the matching process.
- **instructions**: specifying the set of OpenFlow actions and/or sending to a subsequent table in the pipeline.

Inside a specific flow table, the values represented by match fields and priority are sufficient to identify a flow entry in an unique manner. There is only one flow entry in a flow table with a specific set of match fields and priority value. The table-miss flow entry corresponds to an entry for which all of its fields are wildcarded (i.e. masked with "don't care bits") and its priority value set to 0.

Packet Matching

When a packet is received by the OpenFlow switch, it first selects the 0 indexed flow table where a table lookup is performed against the just received packet. Depending on the matching entry instruction set, further table lookups may be performed if a "sent to subsequent table" action is specified.

The header fields parser, as its name suggests, extracts all the header fields from the packet that are supported by the current OpenFlow version, retrieving also the pipeline fields associated with the packet. The extracted packet header fields correspond to those of the most popular protocols that are used today, e.g. Ethernet source/destination address, IPv4 source/destination address, etc.. The set of header and pipeline fields serves as the current image of a packet being processed.

When all of the match fields in a flow entry are matching the packet header and pipeline fields extracted by the parser, bit by bit, then that packet results in having a successful matching entry. If the flow entry specifies a fully wildcarded match field (i.e. value ANY), then this field will successfully match any value that is set in the same corresponding header or pipeline field in the processing packet. Whenever a bitmask is specified on a given match field, the bitmask is applied on the corresponding header/pipeline field of the packet via a bitwise AND operation. If the resulting value matches the one specified in the match field coupled with that bitmask, then we have a successful matching entry.

2.3.4 Open vSwitch

Open vSwitch is an SDN-based open source project, introduced for the first time by Pfaff *et al.* in [2]. It implements an SDN-based multilayer software switch which is also able to "speak" OpenFlow with the connected OpenFlow controllers, if there are any. Its main purpose is to enable network design flexibility and massive network automation via programmatically implemented extensions. Because it also implements a kernel module, it was merged in the Linux kernel tree since the kernel version 3.3 [24].

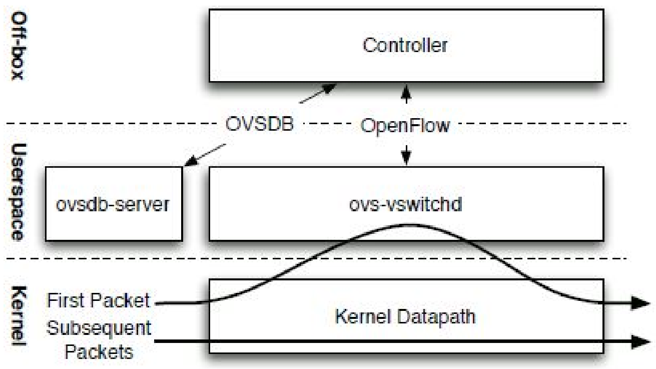


Figure 2.10: Components and interfaces of Open vSwitch. There is a miss when processing the flow's first packet, so the kernel module sends the match-missing packet to the userspace daemon. After which, the kernel module stores in its cache the instructions received from the userspace daemon, which will be applied on subsequent packets of the same flow. Source [2]

Open vSwitch (often abbreviated as OvS) consists of two major components, which combined together form the packet processing logic. These are `ovs-vswitchd` and its kernel module `openvswitch.ko`. `ovs-vswitchd` is one of the core components, implemented as a userspace daemon, which defines the logic of packet processing engine. It does this by adding a simple abstraction on top of the kernel module component (`openvswitch.ko`). It also implements a control over all the Open vSwitch datapaths present on the system (sitting in kernelspace).

`openvswitch.ko` as just mentioned, is the OvS's kernel module. It implements the switching logic complying with the SDN paradigm. Therefore, its main purpose is to implement datapaths in kernelspace. Datapath is a concept defining a set of physical or virtual ports. This means that when a packet enters a physical or virtual port belonging to an OvS datapath, then this packet is logically captured for processing by the Open vSwitch kernel engine, skipping in this way the legacy Linux network stack. Figure 2.10 is a sketch of the OvS architecture, showing the normal packet processing pattern.

We can picture the processing of Open vSwitch as a two layer logic. The bottom layer corresponds to the kernel module, which is the first to receive the ingress packets as they are actually captured when the ingress physical port is linked to a virtual port which, in turn, is embedded in an OvS datapath. The top layer is represented by the userspace daemon, which usually receives packets from the kernel module whenever the kernel module does not know what to do with packets of this specific type of flow. After which the userspace component will instruct the kernel datapath how to behave with the next packets of the same flow, in order to reduce processing latency due to the fact that from now on all of the received packets belonging to the same flow will be processed in the OvS kernel datapath only. The "what to do" part refers to what actions should be executed on packets matching the associated flow. These actions usually tell the matching packet to what physical or logical port it should go.

At this point, the more careful readers may have guessed the use policy of the OvS kernel module, which corresponds to a cache-like operation. Well, that is in fact the key purpose of an OvS kernel datapath. The kernel module is most efficiently used as a cache of the most recent packet flows.

In Figure 2.10 we can also see that OpenFlow is the protocol used to exchange information between OvS and OpenFlow controllers. The OpenFlow controller simply sends flow tables to the OvS userspace daemon. This daemon installs these OpenFlow flow tables and, when receiving a packet from the OvS kernel datapath, it performs table lookups starting from the first flow table in order to find a matching entry for the packet. If a matching entry is found, `ovs-vswitchd` fetches this matching flow entry and caches it in the kernel module. The OpenFlow controller is abstracted from these caching details, so the userspace along with the kernelspace components are seen as one OpenFlow logical switch.

Usually most of the packets are only processed in the kernelspace module, thus it is referred as a *cache*. As previously mentioned, a packet gets captured by the Open vSwitch datapath whenever it enters one of the datapath's virtual ports. The datapath's flow key parser catches the packet and extracts all the values of the packet header fields that are recognized by OvS, storing this set of values inside a, so-called, *flow key*. This flow key is a container holding all of the information extracted by the flow key parser, which may consist of header

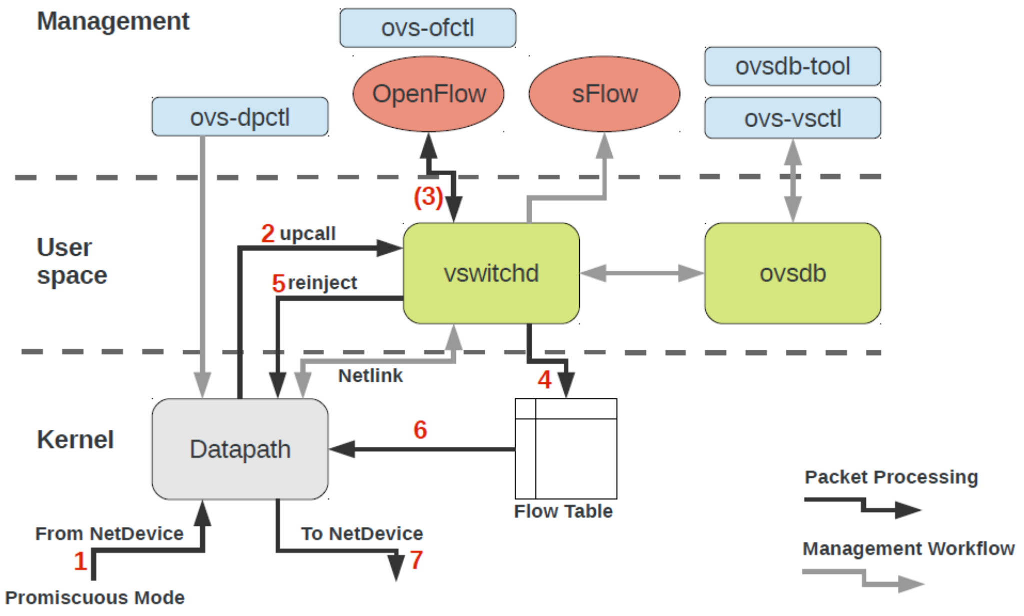


Figure 2.11: Open vSwitch's packet processing and management workflows, source [3]

fields values, non header values such as the ingress port number and other metadata values.

In order to better explain how Open vSwitch processes network packets, let us join a *packet in a journey through the Open vSwitch stack*, illustrated in Figure 2.11. We use "OvS" as a shorthand for Open vSwitch.

Journey of a packet through the Open vSwitch stack

Suppose you have a Linux box with two installed NICs and Open vSwitch properly installed and configured. An OvS datapath is allocated in kernelspace with two OvS ports, each linked to one of the two physical port interfaces. Then, every packet received on one of the two NICs is automatically hijacked to go through the Open vSwitch packet processing logic, thus bypassing the entire Linux networking stack. From this point on, the fate of all the packets received by the two physical ports depends entirely on the Open vSwitch engine.

The received packet enters first the OvS kernel datapath (Figure 2.11 (1)). The OvS engine dictates the execution of a flow table lookup operation in order to find out if it knows what to do with this type of packet flows. If this is the first type of packet ever received, then the datapath will almost surely not know what to do with it. Thus, the packet results in a *cache-miss* so the datapath executes an upcall for sending it up (2) to the userspace daemon via a generic netlink socket. At this point, `ovs-vswitchd` attempts to find a matching entry in any of its installed OpenFlow tables. If this matching also results in a miss, the triggering packet is encapsulated in an OpenFlow packet-in message and sent to an OpenFlow controller (3).

The OvS userspace will then wait to be instructed by the controller on what to do with such type of packets. However, whether the daemon knows what to do or not, it will eventually have knowledge of the specific actions to be applied on such type of packets. So, either way, `ovs-vswitchd` will have the packet processing decision saved in one of its flow tables and further cached in the datapath (4), after which it will give the processed packet back to the OvS kernel datapath. From now on, packets of the same flow will match the cached flow entry (6) and forwarded (7) to the specified OvS vport, and from there to the associated physical port interface.

Some of the Open vSwitch's key implementation details, such as the *packet classification* algorithm, are described in Chapter 4.

3 EPC Possible User Plane Evolution

This chapter first presents the design of a possible evolution, if an SDN-based solution is adopted, for the user plane of a mobile core network architecture, specifically of the LTE's S-GW and P-GW network node entities. It is an illustration of a very plausible evolution of the current EPC's user plane due to some major issues arising with the smartphone proliferation and the already in going IoT involvement. Then, in order to understand the proposed design implications, a detailed description is given about how user data packets are being forwarded within the EPC network.

3.1 SDN-based User Plane Architecture

The proposed architecture, shown in Figure 3.1, aims to slightly change the existing EPC network embodying SDN paradigms in the mobile core network itself. The user plane functions of the S-GW and P-GW is changed, specifically replaced with an Open vSwitch software switch able to perform user data forwarding.

As already said, this thesis project extends only Open vSwitch with GTP-U capability, it does not apply the appropriate implementations in order to have also a GTP-capable OpenFlow controller. The reason is because the GTP-U extension has been implemented *natively* (more in the next chapter) in Open vSwitch. Thus, to have a GTP-enabled OpenFlow controller also requires a GTP extension of the OpenFlow protocol itself. As OpenFlow is an actual real world protocol, extending it requires long implementation times, and once that is done, an OpenFlow controller has to be chosen, applying the corresponding GTP support to its core API sticking to the GTP extension of the OpenFlow protocol.

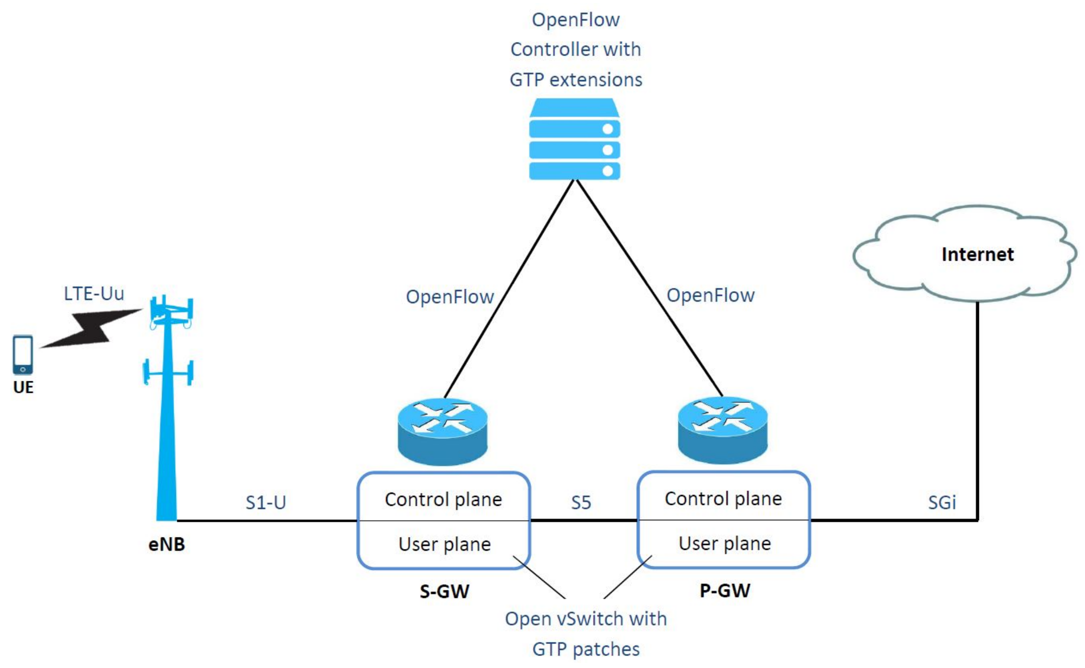


Figure 3.1: EPC SDN-based proposed architecture

3.2 Protocol Stack Dilemma

The GTP protocol is one of the most important protocols used in an EPC core network. This is mostly due to the fact that it provides the UEs with mobility, with no need to change its already assigned IP address (as long as it remains covered by the same P-GW). The GTP's function is to establish <eNB – S-GW> and <S-GW – P-GW> tunnels in uplink, and viceversa in downlink. A GTP tunnel is uniquely identified by its tunnel key ID, the TEID, along with its tunnel destination IP address.

To be able to apply SDN techniques, we must employ some kind of SDN-based hardware and/or software solution. Thus, we will use Open vSwitch as the implementation solution for the thesis project, since Open vSwitch implements a SDN-based software switch. Though there are two nontrivial problems with using OvS:

1. One of them is due to the fact that the protocol used to communicate with SDN controllers is OpenFlow, and at the time of writing, the protocol's most recent version, 1.5.1 [23], does not support a single header field in the GTP header structure.

2. The other one is due to the GTP tunnel protocol stack, which does not require the presence of an Ethernet header.

In order to support tunnel encapsulation and decapsulation along with a GTP match and header recognition, we have two options:

- implementing an Open vSwitch GTP extension, or
- leaving alone Open vSwitch and get a modular dedicated hardware network card board explicitly supporting GTP operations.

A GTP tunnel packet does not have specified any Layer 2 Ethernet information. Therefore, if we want to extend Open vSwitch with a GTP support in an IP-based network environment, an Ethernet header has to be added to each GTP tunnel packet and properly populated with source and destination Ethernet addresses along with the two-byte ethertype field. The addition of a properly crafted and populated Ethernet header on each GTP tunnel packet, brings a slight increase in overhead that should, however, not have such of a big impact on the overall performance since the GTP extension in Open vSwitch is chosen as solution to be implemented.

Given that the implementation choice is to implement the forwarding behaviour of the GTP tunnel, we will now see, with the help of a detailed example, how user data packets are being forwarded via GTP tunnels within a today's traditional EPC.

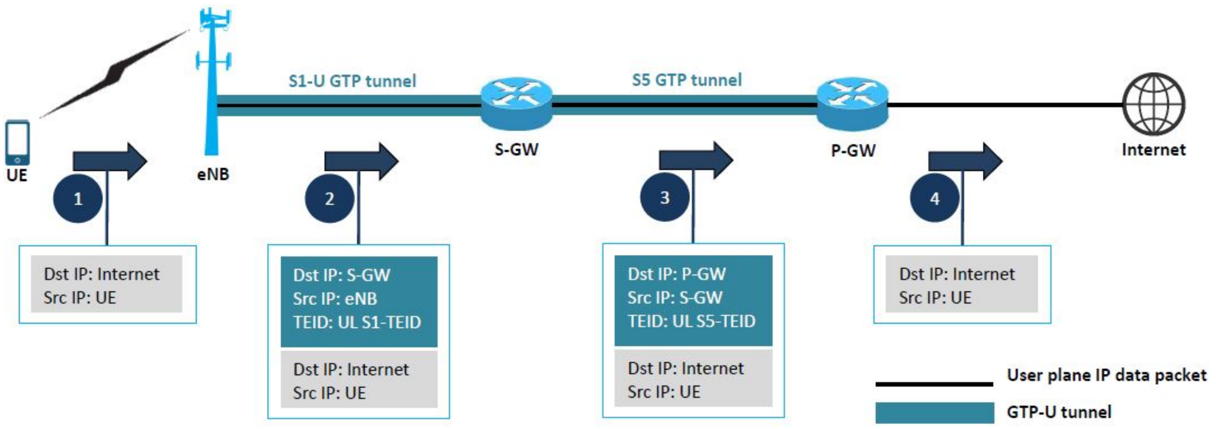


Figure 3.2: An IP packet sent by the UE to the Internet

3.3 GTP-U Tunneling

We will now see, with the help of a detailed example, how the user plane data traffic generated by an UE reaches the Internet and vice versa, all way through the EPC. As shown in the example Figure 3.2, IP packets sent by an UE are delivered from an eNodeB (eNB) to a P-GW through GTP user plane tunnels. This means that the P-GW is actually seen as the UE's default gateway within the EPC.

Taking Figure 3.2 as a simple example, let us now take a step by step look into what happens with the IP packets sent by the UE to the Internet (the situation in the reverse direction is specular but with different GTP TEIDs):

1. **UE to eNB**

   Suppose an UE sends an IP packet with its destination IPv4 address set to, say, 8.8.8.8 (IPv4 address of a Google DNS server) to an eNB through the radio link. Then the UE's generated IP packet is something like shown in Figure 3.3 below.

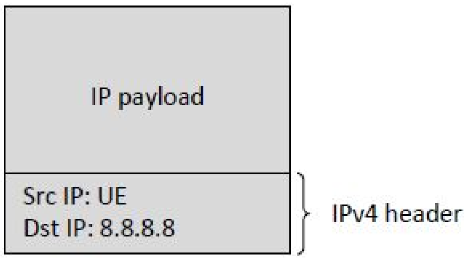
   

   Figure 3.3: UE generated packet

   Notice the lack of Ethernet header, that is because the UE sends the packet over an LTE-Uu interface (i.e. LTE radio link), instead of an Ethernet-based one!

2. **eNB to S-GW**

   The eNB receives the UE's IP packet, establishes a GTPv1 tunnel session with the S-GW, and finally encapsulates the just received IP packet inside this GTP tunnel. The GTP-U tunnel header consists of three individual headers — a GTPv1 header, UDP header, and an IPv4 header for the GTP tunnel itself — in front of the UE's IP packet. Then, the resulting IP packet (sent by the eNB to an S-GW) will be as illustrated in Figure 3.4.

   Thus, we have an IP routing network between the UE's and its associated S-GW. This means that the packet's destination IP address (in this example, the IPv4 address of the S-GW, which is the destination IP address specified in the outer IPv4 header) is parsed and the IP packet is forwarded to the proper S-GW accordingly.

3. **S-GW to P-GW**

   The S-GW, upon receiving the IP packet from the eNB, modifies its GTP and outer IPv4 headers as shown in Figure 3.5.

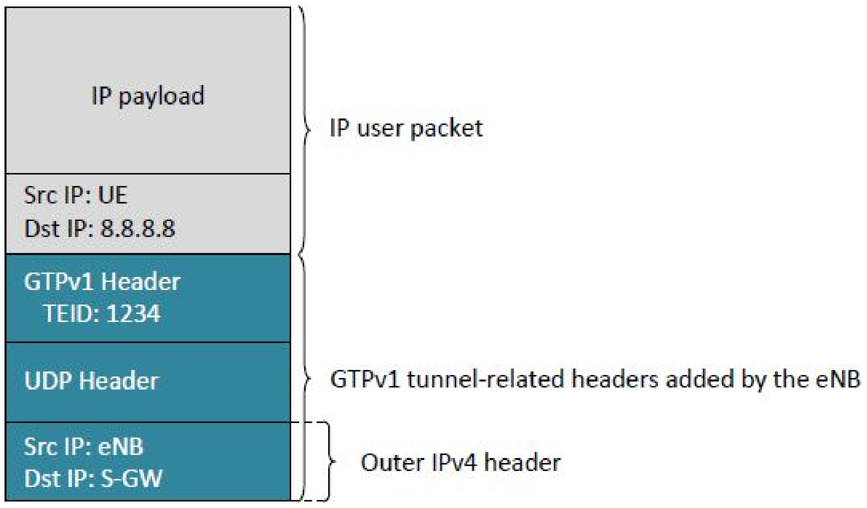


Figure 3.4: The IP user packet encapsulated by eNB in a GTP-U tunnel

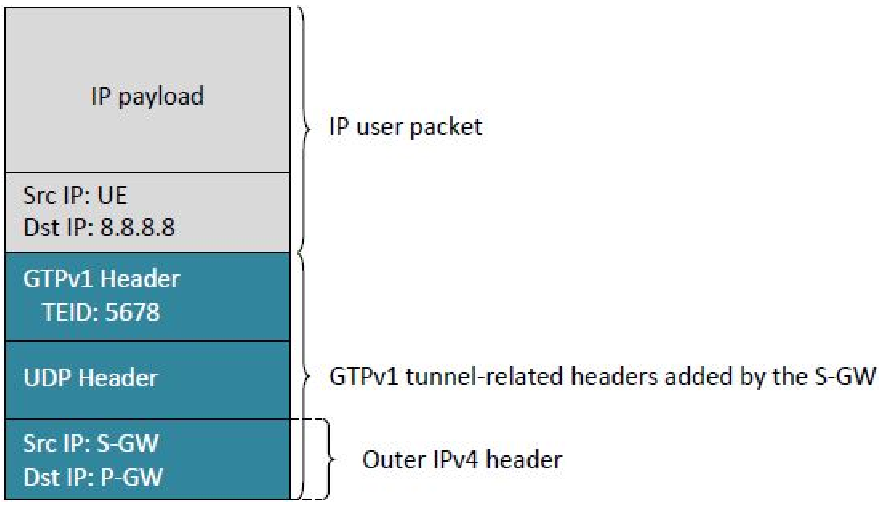


Figure 3.5: The IP user packet encapsulated by S-GW in a GTP-U tunnel

4. **P-GW to PDN (the Google DNS server in this case)**

   The packet is then delivered to the P-GW accordingly. The P-GW then removes all three headers (the GTP tunnel related headers) from the packet and delivers the original packet (see Figure 3.3) sent by the UE to the Internet.

Now it is the right time to give a full explanation about these GTP TEIDs. Suppose there are 10 UEs that are connected to the same <S-GW, P-GW> pair. If for each UE a GTP tunnel is allocated, at the end the core network will result in having created a total of 10 GTP tunnels. At the allocation time of each tunnel, the EPC core network will generate and, therefore, assign the GTP tunnel a unique label key, the so-called TEID.

Let us take Figure 3.2 as a simple example reference. It clearly shows how the UE-generated packets are being tunneled through GTP tunnel sessions in order to reach the Internet. The uplink <eNB – S-GW> tunnel has assigned TEID = UL S1-TEID (say 1234), whereas the immediately next uplink tunnel session, the one with S-GW and P-GW as endpoints, has assigned TEID = UL S5-TEID (say 5678). From now on, because unique labels specific to each UE are being used, it is sufficient to check the tunnel's TEID in order to identify the subscriber that is using the specific GTP tunnel. Practically, however, P-GWs check TEIDs along with the UE's IP addresses, whereas eNBs and S-GWs check only the assigned TEIDs.

A generated TEID can be used only in one single direction. Meaning that if the core network has assigned a freshly generated TEID for the <S-GW – P-GW> uplink tunnel, the same label key can not be used for the same <S-GW, P-GW> pair in the reverse direction (i.e. downlink). This implies that in the reverse direction, i.e. traffic destined to UE and sourced from the Internet, the core network will generate and assign a different label key for each tunnel pair up to the UE's eNB. Specifically, a new TEID for <P-GW – S-GW> GTP tunnel and another one specific for <S-GW – eNB> GTP tunnel. Finally, the eNB will extract the data packet and forward it to its proper UE.

4 Implementation

In the previous chapter, an overview of the design of an SDN framework for the user plane evolution of a mobile core network was presented. In this chapter, the details of this design and the experimental SDN implementation is given; which is what the thesis project's main goal consists of.

The GPRS tunneling protocol (GTP), as already mentioned, is a very central protocol for carrying subscriber's IP packets through the EPC. It is the key enabler for expanding EPC with SDN concepts. This was done by adopting, specifically, Open vSwitch. We will first see some of the main implementation details in the Open vSwitch's datapath kernel module. For performance reasons and time required for its implementation, the GTP support was implemented in the kernel module only. At last, the actual GTP implementation will be briefly described, with some minor code references.

4.1 Implementation steps

First of all, let us explore in more detail the thesis project's main goal, which is "**implementing the forwarding of user data traffic in an EPC network with an SDN fashion**". The goal statement is clear and simple, however, it yields some not so straightforward consequences. The first issue relates with what type of SDN technology should be employed, and the proposed solution here is a pretty obvious one, that is OpenFlow. The reason is because it is the standard chosen protocol allowing the communication between any OpenFlow controller and the "controlled" OpenFlow nodes. OpenFlow is the common ground on which the controller and the controlled node entities operate.

The forwarding logic of user data between the eNodeB (the LTE base station) and external networks, such as the Internet, relies upon carrying subscriber's IP packets inside the so-called GTP tunnels. This means that the employed SDN technology solution, in this case being OpenFlow, has to support the GTP tunnel protocol. As already stated in the previous chapter, the ugly truth is that at the time of writing, the OpenFlow's most recent specification, that is 1.5.1 [23], does not support the GTP protocol at all. All of this meaning that any OpenFlow entity would not understand a single thing about what a GTP tunnel means, which is the first core feature to be implemented if we want to apply SDN concepts on an EPC network infrastructure. Therefore, the current OpenFlow standard can not be used for the purpose of the thesis project's main goal!

As at the time of writing the current SDN industry did not offer any *open* technology explicitly supporting GTP tunneling, the only solution was to check for another way. So the final solution consisted in focusing attention on the open source realm, therefore, the next implementation stage was that of searching, finding, and finally adopting, the most appropriate open source implementation of SDN paradigm. It did not take a lot of time to realize that Open vSwitch [2] represented the best fit for the thesis project's purpose. The reason [24] is that currently it is the most widely recognized and supported SDN implementation project in the open source SDN community. Thus, by adopting Open vSwitch the thesis project's main goal could be achieved only by developing a Proof of Concept using such open source SDN platform, i.e. Open vSwitch. There was, however, only one problem with Open vSwitch, which is the fact that it does not support GTP tunneling. The only thing possible was to study and learn the Open vSwitch internals, how it operates and how it behaves when capturing a packet, with the purpose of implementing the missing GTP tunneling feature, thus extending the OvS logic which enriches its capabilities.

The key idea was that a GTP capable Open vSwitch could be used in an EPC network for implementing the forwarding of user data traffic in an SDN fashion, in this case specifically on the S-GW and P-GW node entities. Open vSwitch is also able to exchange SDN information with SDN controllers via OpenFlow channels, thus allowing the implementation of a full SDN, GTP-enabled, architecture for the EPC's user plane. The thesis project's purpose, therefore, becomes that of extending Open vSwtich with the GTP tunneling support.

Now let us explore the implementation stages that were followed in order to obtain a GTP enabled Open vSwitch. The following is a list of related implementation stages, some of them are described here, others are described later in the chapter:

1. **GTP flow key**

   The idea of flow key is described later in the chapter; the flow key definition was extended with GTP tunnel related parameters, specifically with the tunnel's IPv4 destination address and its 32-bit TEID. The pair of these two parameters was specifically chosen because it uniquely identifies a GTP tunnel instance from another. Thus, the extended flow key definition will look as showed in the following code snapshot:

```
struct sw_flow_key
{
    ...
    /* GTP tunnel key */
    struct
    {
        /* big endian */
        __be32 ipv4_dst;  // IPv4 dst address
        __be32 teid;    // GTP Tunnel ID
    } gtp_u;
};
```

2. **GTP flow key parser**

   The idea of flow key parser is described later in the chapter; the flow key parser logic was extended with the ability of reading and scanning GTP packets, so that in can fetch the GTP tunnel key, i.e. tunnel's IPv4 destination address and its corresponding TEID field value, inside the flow key instance. We know, from the previous step, that the flow key was extended with enough room space where the GTP tunnel key parameters could be stored. The GTP flow key parser was implemented with a function called `check_extract_gtp()`, which, as its name suggests, checks if the passed skb is a GTP tunneled packet, and extracts at the same time the GTP tunnel key.

```
int check_extract_gtp(struct sk_buff *skb, struct sw_flow_key *key)
{
        struct iphdr *iph;
        struct gtpv1hdr *gtph;
        unsigned short ethtype, dst_port;

        if (key)
                /* zero out the gtp tunnel key */
                memset(&key->gtp_u, 0, sizeof(key->gtp_u));

        ethtype = eth_hdr(skb)->h_proto;
        if (ethtype != htons(ETH_P_IP)) // if next header is not IPv4
                return 0;

        iph = ip_hdr(skb); // skip the possible gtp ethernet dummy header
        if (iph->protocol != IPPROTO_UDP) // if next header is not UDP
                return 0;

        dst_port = udp_hdr(skb)->dest;
        if (dst_port != htons(GTPv1_PORT)) // if next header is not GTPv1
                return 0;

        gtph = GTPv1_HDR(skb); // get a ptr to the gtp header
        if (gtph->type != GTP_MSG_TYPE_GPDU) // if gtp-u msg type is not G-PDU
                return 0;

        if (key)
        {
                /* in case of GTP tunneling we don’t care about ethernet */
                memset(&key->eth, 0, sizeof(key->eth));
                /* populate key with the tunnel’s IPv4 dst address */
                key->gtp_u.ipv4_dst = iph->daddr;
                /* populate key with the gtp Tunnel ID */
                key->gtp_u.teid = gtph->teid;
        }

        return 1;
}
```

Following the ETSI specification standard [10], in the `check_extract_gtp()` routine, `GTPv1_PORT` is a constant defining the 2152 UDP port serving the GTPv1 protocol. `GTP_MSG_TYPE_GPDU` is another constant set to 255 which checks the GTP packet type, in this case we care only about GTP packets that are carrying user data packets, such GTP packets are also called G-PDUs (GTP Packet Data Unit).

At this point we have that Open vSwitch is able to read and fetch GTP packets, but it is still unable to apply matches on GTP flows. To do this we would need to add some API support via an userspace OvS utility tool, so that a network administrator could specify GTP tunnel related flow entries / rules to be installed. Hence, the next step consisted of choosing the appropriate OvS administrative utility tool and extending it correspondingly.

3. **API support for GTP flows**

The OvS administrative utility tool that was chosen to be extended with the capability of specifying and installing GTP rules is called `ovs-dpctl`. `ovs-dpctl` allows the installation of generic flow rules via an already implemented command that is called `add-flow`. There are two implementation choices here, either add the new API support inside `add-flow`, or implement a new command dedicated specifically for installing only GTP tunnel related flows. The chosen option was to dedicate a separate command for its own logic of installing GTP flows only, it was named `add_gtpu_flow`. The reason for choosing this option was to avoid mixing the addition of GTP flows logic with the generic flows logic, as the GTP flow extension adds some radical changes which do not comply with the generic flows behaviour, like being forced to specify L2 information for the command's matching part. An example of `add_gtpu_flow` could be as follows:

```
# ovs−dpctl add−gtpu−flow "ipv4(dst=8.8.8.8,frag=no)"
    "push_gtp(src=192.168.1.1,dst=192.168.1.2,teid=1234), 1"
```

This adds a GTP flow that matches the entire IPv4 traffic destined to 8.8.8.8, and encapsulates it inside a GTP-U tunnel set to be sourced from 192.168.1.1 and destined to 192.168.1.2, with '1234' set as the corresponding tunnel label (i.e. GTP TEID field). At last, this GTP-encapsulated traffic ends to be forwarded over the OvS vport '1'.

All the information specified with `ovs-dpctl` is passed into the OvS kernel module over a netlink socket instantiated specifically for sending information, like adding a new flow entry in this example, to the kernel module. Netlink sockets were implemented in Linux with the main purpose of instantiating communication channels between userspace processes and the kernel [31]. Therefore, once the `ovs-dpctl` utility builds the appropriate netlink message, it is handed over to the OvS kernel module via a netlink socket. The OvS kernel module, in turn, checks if there are any errors in the received netlink message, and in case it is correctly formatted, the corresponding flow entry is extracted and finally added in the hash table mapping flow keys to flow entries. Thus, the OvS kernel module's definition of generic netlink socket operations was extended with a new dedicated handler, processing the addition of GTP flows only, as showed below:

```
static struct genl_ops dp_flow_genl_ops[] = {
    ...
    {
        .cmd = OVS_FLOW_CMD_NEW_GTPV1, // add new gtp flow table entry
        .flags = GENL_ADMIN_PERM,
        .policy = gtp_flow_policy, // attributes validation
        .doit = ovs_flow_cmd_new_gtp
    },
};
```

This `dp_flow_genl_ops[]` array represents the generic netlink operations defining the allowed flow related requests from userspace processes. `ovs_flow_cmd_new_gtp()` was implemented to operate as the handler routine to be executed whenever a generic netlink message is received with `OVS_FLOW_CMD_NEW_GTPV1` set as the `cmd` field value in the `genlmsghdr` header within the netlink message - in other words, it is the implemented generic netlink callback receiving requests to install new GTP flow table entries. `ovs_flow_cmd_new_gtp()` eventually installs the GTP flow entry, meaning that when a packet matching this flow entry is received, the mapped actions are executed as a result of the OvS packet processing logic (this is also what makes Open vSwitch to be seen as an SDN implementation technology). The next step, therefore, was to implement the appropriate GTP tunnel related actions, specifically encapsulating and decapsulating IPv4 user data traffic.

4. **GTP encap and decap actions**
   The `push_gtp(src=..., dst=..., teid=...)` and `pop_gtp` action directives specified within `ovs-dpctl add-gtpu-flow` commands have to be implemented by their corresponding routines inside the OvS kernel module. `push_gtpv1()` and `pop_gtpv1()` are the two functions that were implemented to encapsulate and decapsulate, respectively, IPv4 user data packets in/from GTP tunnels. Their implementation were fundamental for the thesis project's purpose, so both of them are greatly described in the "Encapsulation and Decapsulation in/from GTP" dedicated section at the end of this chapter.

5. **Transmission of GTP encapsulated/decapsulated packets**
   At this point the only information missing, i.e. not specified by the admin via `ovs-dpctl add-gtpu-flow` utility tool, are the appropriate L2 addresses. The reason for this is because we are in an Ethernet network environment, thus we need the correct and full Ethernet information for a successful L2 packet delivery. The only solution possible was to employ an explicit ARP request in case the local ARP cache did not have the <L2, L3> entry for the tunnel's IPv4 destination address. More implementation details can be found at the "Packet Transmission" dedicated section at the end of this chapter.

The rest of this chapter describes in more detail those concepts that so far were briefly introduced, the Open vSwitch core packet processing logic and its internals, its major addressed problems (like the "packet classification problem"), and finally a brief description of some of the kernel's most critical data structures which knowledge have to be thought of in order to have an idea of how to work with the OvS kernel module internals.

4.2 The Packet Classification Problem

Before describing the actual Packet Classification problem, let us first familiarize with the major procedures that occur when a packet enters the Open vSwitch datapath.

The OvS kernel module's main purpose it to provide the userspace core daemon with a cache-like capability, mainly due to performance reasons. But how is the kernel module actually able to do that? Well, the key idea is a pretty clever one, which consists in having the ability of allocating at least one, so-called, *datapath*. A datapath is the core concept logic implemented in kernelspace by the OvS kernel module, it can be seen as a similitude of a Linux bridge. Its implementation allows it to embody one or more OvS ports (or vports in OvS terminology). Usually, a vport is allocated with the purpose of being linked to a physical network port interface (NIC) present on the system, so that all the traffic received on its directly linked NIC will be captured and processed by the Open vSwitch packet processing logic. Generally speaking, a datapath is implemented as a collection of data structures, such as vports, flow table, flow keys, flow entries, etc.. A datapath consists mainly of a flow table, which in turn consists of multiple flow entries populated by the user admin and/or `ovs-vswitchd` userspace daemon. A flow entry is defined by a flow key, mapping packets of the same type of flow to a set of specified actions. The flow key is a notion of a big data structure that stores values of header fields and metadata extracted from a packet captured by the OvS datapath. This extraction is performed by a flow key parser, implemented as a set of functions invoked whenever a packet is received on a vport. Finally, the set of actions may have specified several actions to be executed on the flow matching packet, such as forwarding to a vport, modifying some header fields, encapsulating or decapsulating in/from a tunnel (e.g. MPLS, GRE, etc.), deliver to userspace, or simply dropping it.

The packet that is received on a vport gets captured by the Open vSwitch networking stack. Then, the flow key parser comes into the picture. Its function is to scan the entire packet's header stack, in an attempt to extract all the values assumed by the header fields (those recognized by Open vSwitch) and to store them in a container data structure called flow key. The extracted flow key is then used as the search key indispensable for the successful outcome of the flow table lookup process. In case the extracted flow key is found in the flow table (i.e. we have a successful flow match, call it *flow cache-hit*), its mapped set of actions gets fetched and each stated action is executed one by one in the order they are specified. Otherwise, if a matching flow is not found (i.e. we have a *flow cache-miss*), the unmatched packet along with its extracted flow key and flow mask key (identical to its associated flow key but specifying a sequence of bitmasks) are sent to userspace over a generic netlink socket as a sequence of netlink attributes.

The flow key netlink attributes employ an exact and predefined format, the more curious would check the `openvswitch.h` header file. For a better understanding of these flow key netlink attributes, we can employ some very specific and strict rules for how to write them down. The flow key netlink attributes belonging to a specific flow entry gets written as one single string separated with commas, making also use of parenthesis whenever you want to specify arguments, and eventually nesting, associated with the header stack parameter stated just outside the corresponding opening parenthesis. Take for example the following:

```
in_port(1), eth(src=00:11:22:33:44:55, dst=66:77:88:99:aa:bb),
eth_type(0x0800), ipv4(src=192.168.1.1, dst=192.168.1.2, proto=17,
frag=no), tcp(src=49163, dst=80)
```

This entire string describes a flow key matching TCP packets requesting http service that were received on vport 1.

As mentioned before, a flow key may be sent to userspace along with corresponding flow mask, which is optional. If the flow key is sent without a corresponding flow mask, then it is assumed to result with a successful match only if each of its specified argument matches exactly the same value of the equivalent packet header field. Such type of flow key is called exact match flow. On the other hand, if the flow key is sent coupled with a flow mask then it is referred as a wildcarded flow key.

The wildcarded flow key, contrary to the exact match counterpart, may allow different values for a header field depending on its corresponding specified bitmask. So, long story short it is the exact same thing of a subnet mask when it is applied to a host's IP address. You specify `/24` to exactly match (i.e. bit by bit) the three most significant bytes assumed by the host's IP address for the purpose of finding the subnet it sits on, whereas the less significant byte may assume any value as we care only about finding its subnet. A common way of how to call the bits assumed in the bitmask is a "care" bit for the '1' bits, and a "don't care" bit for the '0' bits, as we actually care about the corresponding bit in the flow key if its associated bit in the bitmask is '1', and we do not care if the corresponding flow key bit is set to either '1' or '0' in case the associated bit in the bitmask is '0'.

The implementation of the flow table is realized as a hash table data structure where the key part is represented by the extracted flow key, and the mapped value is the *flow entry*. A flow entry is a data structure containing the optional flow mask and the corresponding actions to be performed on the matching packet.

Now that we have a basic knowledge of what is happening when a packet enters OvS datapath, we can focus onto each of its major packet processing stages, studying their corresponding main problems and looking at the OvS implementation approach.

In a non dedicated hardware, the packet classification problem usually consumes a lot of CPU resources, and in case of Open vSwitch even more attention has to be dedicated as it is actually executed on behalf of a general-purpose CPU. The reason is that OpenFlow provides a matching process that is generic, causing a potentially exponential number of possible test combinations of all the recognized fields extracted from a processing packet, including metadata such as the ingress vport number. Thus, Open vSwitch uses a *tuple space search* classifier [32] for all of its packet classification logic, in both kernel and user space. Hence, in order to understand the complexities related with the flow table lookup processing logic, this *classifier* is briefly described.

4.2.1 Problem Definition

Given a sequence of N_k header fields, referred to as a "full key", and a table of N_k columns on which to perform lookup, find the matching flow key of M_k header fields ($M_k \leq N_k$) which is a subset of the full key, which hash value results in a hit in the flow table. For example, suppose the full key K is $[F_1, F_2, F_3, F_4]$, then through all the $2^{N_k} = 16$ combinations of these $N_k = 4$ header fields at least one of them results in a hash table hit. The definitive problem is to find that exact combination of header fields with the least number of hash calculations possible. That is what the packet classification problem is.

4.2.2 Defining Tuple Space

Tuple Space Search is the algorithm adopted by Open vSwitch in order to cope with the packet classification problem. The term "key signature" has a major role to help us understand how this algorithm works. As a simple example, suppose the full key has a total of three 4-bits header fields. Then, a rule $R = [F_1 = 0***, F_2 = 01**, F_3 = 100*]$ (the $*$ symbol stands for an "I don't care" bit) has the following $k_s = [1, 2, 3]$ as its associated key signature. Each number in the k_s key signature represents the number of the most significant bits to match with the corresponding value in that header field. In this case, the most significant bit for values in the F_1 header field, the two most significant bits for F_2, and the three most significant bits for F_3.

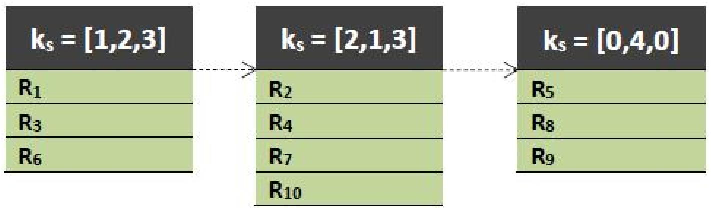
Consider, for example, the following set of rules:

	F_1	F_2	F_3	k_s
R_1	0∗∗∗	01∗∗	100∗	[1,2,3]
R_2	01∗∗	1∗∗∗	010∗	[2,1,3]
R_3	1∗∗∗	01∗∗	110∗	[1,2,3]
R_4	00∗∗	0∗∗∗	100∗	[2,1,3]
R_5	∗	0101	∗	[0,4,0]
R_6	1∗∗∗	10∗∗	101∗	[1,2,3]
R_7	00∗∗	1∗∗∗	000∗	[2,1,3]
R_8	∗	1101	∗	[0,4,0]
R_9	∗	1001	∗	[0,4,0]
R_{10}	10∗∗	1∗∗∗	010∗	[2,1,3]

Note the rules R_n can be grouped into three subsets based on the associated key signatures:

$$\{(R_1, R_3, R_6 \in [1,2,3]),\ (R_2, R_4, R_7, R_{10} \in [2,1,3]),\ (R_5, R_8, R_9 \in [0,4,0])\}$$

Therefore, the overall structure of the flow table, from the tuple space search point of view, results to be a list of three lookup tables as shown below:



Each lookup table in this list, referred as a *tuple* in tuple space search terminology, represents a hash table associated with an unique key signature. Thus, in this case, to find a match we need a maximum of three hash table lookups, which means that the lookup complexity of tuple space search performs linearly in the number of tables in the list.

4.3 Open vSwitch

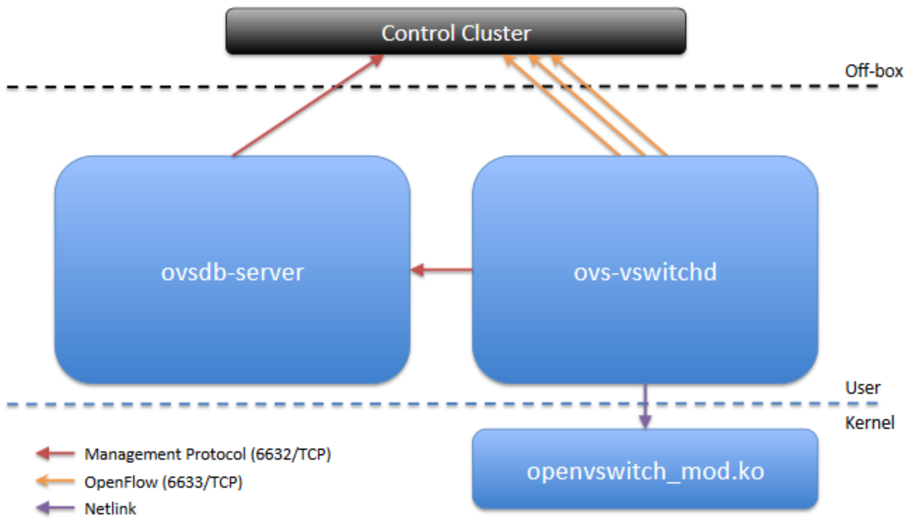


Figure 4.1: Main Open vSwitch components, source [19]

4.3.1 Kernel Datapath

Overview

From an architectural point of view, there are three main components started by the open-vswitch service:

- `ovs-vswitchd`, the core implementation of the virtual switch;
- `ovsdb-server`, manipulates the database of the switch configuration and flows;
- and `openvswitch_mod.ko`, which operates as a fast path cache for flow matching packets.

Figure 4.1 is a simple illustration of their relationship.

One of the most important components implemented by Open vSwitch is represented by its userspace daemon, `ovs-vswitchd`, instructing the OvS kernel module how to behave when encountering data packets belonging to some specified type of flows. The key fact to understand is that `ovs-vswitchd` in no circumstances may be seen as an SDN or OpenFlow controller. It may, however, be thought as some kind of pseudo control plane agent as it controls local data plane modules, i.e. OvS kernel datapath.

ovsdb-server is the Open vSwitch Database server. ovs-vswitchd saves and changes the switch configuration into this database; it communicates with ovsdb-server using Unix sockets with the purpose of fetching and saving the Open vSwitch datapath configuration. Because of this, the Open vSwitch configuration does survive after a reboot.

Let us now take a look at the kernel part. For simplification purposes, we may think of the OvS kernel module as a fixed-function ASIC (Application Specific Integrated Circuit). Essentially, it is the place where all of the packet processing logic is executed upon capturing a packet.

The OvS kernel datapath communicates with the userspace daemon via netlink sockets, the generic netlink family [27] is used in this case. A datapath implements a bunch of generic netlink (genl) commands in order to get, set, add, and delete a specified datapath, flow, or vport. There are also netlink commands used to populate a flow entry with specified actions, which will eventually be executed whenever the flow entry matches a packet. Those used to refer to a datapath, and defined in the "openvswitch.h" header file, are:

```
enum ovs_datapath_cmd
{
    OVS_DP_CMD_UNSPEC, // just a placeholder
    OVS_DP_CMD_NEW,
    OVS_DP_CMD_DEL,
    OVS_DP_CMD_GET,
    OVS_DP_CMD_SET
};
```

The kernel module defines several functions each implementing the logic of creating, deleting, dumping, and modifying a dapath, such as ovs_dp_cmd_[new() | del() | get() | set()]. And not just datapath related, but also applied to vports and flow entries.

Before describing the involved data structures, it is fundamental to understand how a packet is sent out or received from an OvS bridge port. When a packet is sent out, passing through the OvS bridge, it is first sent to an internal device defined by the kernel module, a struct vport instance. The packet is finally passed to internal_dev_xmit(), which actually receives the packet. Now, the kernel needs to perform a flow table lookup, to see if there is any "cache entry" specifying the forwarding for this packet. This is done by the ovs_flow_tbl_lookup_stats() function, which needs a key (the flow key). The flow key extraction is performed by ovs_flow_key_extract() which briefly collects the details of the packet (L2, L3 and L4 header values) and constructs an unique key for this flow. Assume this is the first packet going out after the OvS bridge has been created, thus, we have a "cache-miss" in the kernel, meaning the kernel module does not know how to handle this type of packet. The packet is then passed to the user space module, ovs-vswitchd, with the ovs_dp_upcall() function which also uses genl. At this point, the user space daemon will check the flow entries table to retrieve the destination port for this packet, and will send to the kernel datapath a genl message with OVS_ACTION_ATTR_OUTPUT set as the corresponding command to tell kernel what is the port it should forward to, say eth0. At last, the datapath would execute the specified action in case a OVS_PACKET_CMD_EXECUTE command is found. This normally happens in the do_execute_actions() routine, where the kernel would find this genl command and, thus, execute it causing the forwarding of the packet to the eth0 port via do_output().

The receiving side is very similar. The OvS kernel module registers an rx_handler for the underlying (non-OvS-internal) devices, that is netdev_frame_hook(), so once the underlying device receives a packet on wire, the datapath will forward it to user space to check where it should go and what actions it needs to execute on it.

There are several actions defined by the OvS datapath, other than the aforementioned OVS_ACTION_ATTR_OUTPUT, those implemented in OvS v2.5.0 (defined in "openvswitch.h") are the following:

```
enum ovs_action_attr
{
    OVS_ACTION_ATTR_UNSPEC, // just a placeholder
    OVS_ACTION_ATTR_OUTPUT,
    OVS_ACTION_ATTR_USERSPACE,
    OVS_ACTION_ATTR_SET,
    OVS_ACTION_ATTR_PUSH_VLAN,
    OVS_ACTION_ATTR_POP_VLAN,
    OVS_ACTION_ATTR_SAMPLE,
    OVS_ACTION_ATTR_RECIRC,
    OVS_ACTION_ATTR_HASH,
    OVS_ACTION_ATTR_PUSH_MPLS,
    OVS_ACTION_ATTR_POP_MPLS,
    OVS_ACTION_ATTR_SET_MASKED
};
```

So far, we discussed how the packets are handled by the datapath. However, there are much more details to discuss about, especially about the flows and datapath. Long story short, a flow in the kernel module is represented as struct sw_flow, a datapath is defined as struct datapath, and the flow actions are implemented as struct sw_flow_actions. The implementation details about these data structures can be found in "flow.h" and "datapath.h" header files.

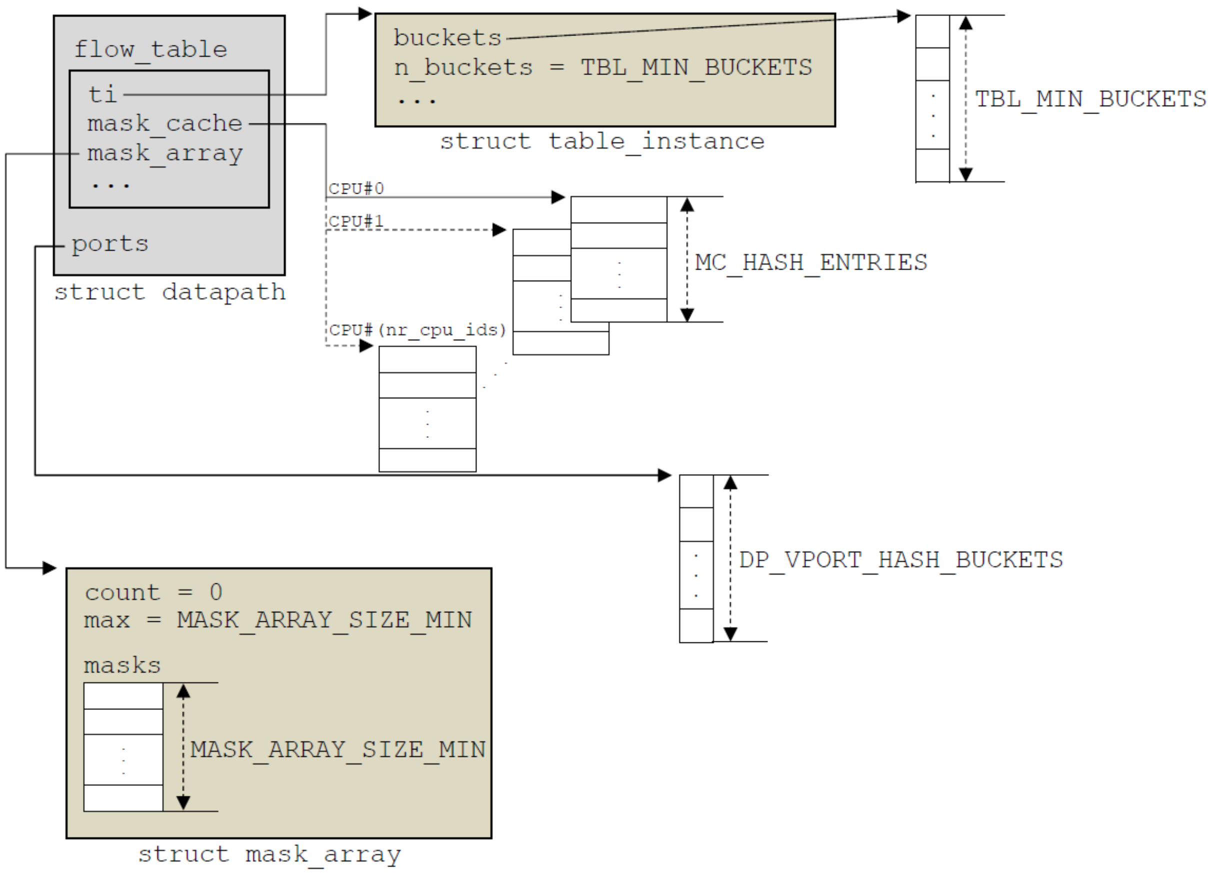


Figure 4.2: The `datapath` data structure and its main relationships

Datapath Initialization

The GTP support has been implemented using Open vSwitch v2.5.0. Figure 4.2 is an in-memory view of the datapath's core data structures when it is first created and initialized, i.e. loaded in the kernel with the "`modprobe openvswitch`" command in Linux terminal.

The four constants shown in Figure 4.2 are defined as follows:

```
#define TBL_MIN_BUCKETS 1024
#define MC_HASH_SHIFT 8
#define MC_HASH_ENTRIES (1u << MC_HASH_SHIFT)
#define DP_VPORT_HASH_BUCKETS 1024
#define MASK_ARRAY_SIZE_MIN 16
```

where:

- `TBL_MIN_BUCKETS` is the size or number of hash buckets in the `buckets` hash table used as the flow lookup table.
- `DP_VPORT_HASH_BUCKETS` is the size or number of hash buckets in the `ports` hash table mapping vport (called *virtual port* in Open vSwitch terminology) numbers to their corresponding vport data structure instances, i.e. `struct vport` instances.

- `MASK_ARRAY_SIZE_MIN` is the size of the `masks` array containing flow mask references, i.e. pointers to `struct sw_flow_mask` instances.

- `MC_HASH_ENTRIES` is the size (`1 << 8 = 256`) of the `mask_cache` per-cpu array. Suppose the CPU of the system you're working on has 4 cores, a normal-performing CPU at the moment of this writing. Then, executing the OvS kernel module, there would be allocated a total of 4 `mask_cache` arrays, one per each CPU core of the executing system.

Data Structures

Here we will see what the major data structures forming the datapath are. The knowledge about how these data structures are used is mandatory as they represent the building blocks of an OvS kernel module instance. The OvS kernel module implementation can be found in the "datapath" directory of the OvS root tree.

datapath

The datapath data structure is the most important one, as it stands for the virtual switch entity for flow-based packet switching. The following is its definition from the "datapath.h" header file:

```c
struct datapath
{
    struct rcu_head rcu;
    struct list_head list_node;

    /* Flow table. */
    struct flow_table table;

    /* Switch ports. */
    struct hlist_head *ports;

    /* Stats. */
    struct dp_stats_percpu __percpu *stats_percpu;

    /* Network namespace ref. */
    possible_net_t net;

    u32 user_features;
};
```

It consists of two important components: flow table and vport.

vport

The vport data structure represents the implementation of the OvS switch port, it can be thought of as a port within a datapath instance. The following is its definition from the "vport.h" header file:

```
struct vport
{
    struct net_device *dev;
    struct datapath *dp;
    struct vport_portids __rcu *upcall_portids;
    u16 port_no;

    struct hlist_node hash_node;
    struct hlist_node dp_hash_node;
    const struct vport_ops *ops;

    struct list_head detach_list;
    struct rcu_head rcu;
};
```

Note the `struct vport_ops *ops` pointer, it represents the type of the virtual port. Follows its definition:

```
struct vport_ops
{
    enum ovs_vport_type type;

    /* Called with ovs_mutex. */
    struct vport *(*create)(const struct vport_parms *);
    void (*destroy)(struct vport *);

    int (*set_options)(struct vport *, struct nlattr *);
    int (*get_options)(const struct vport *, struct sk_buff *);

    int (*get_egress_tun_info)(struct vport *, struct sk_buff *,
                               struct dp_upcall_info *upcall);
    netdev_tx_t (*send)(struct sk_buff *skb);

    struct module *owner;
    struct list_head list;
};
```

`vport->`dp points to the datapath to which the vport belongs to, and all of the virtual ports belonging to the same datapath are mapped into a hash table using their corresponding vport number as key.

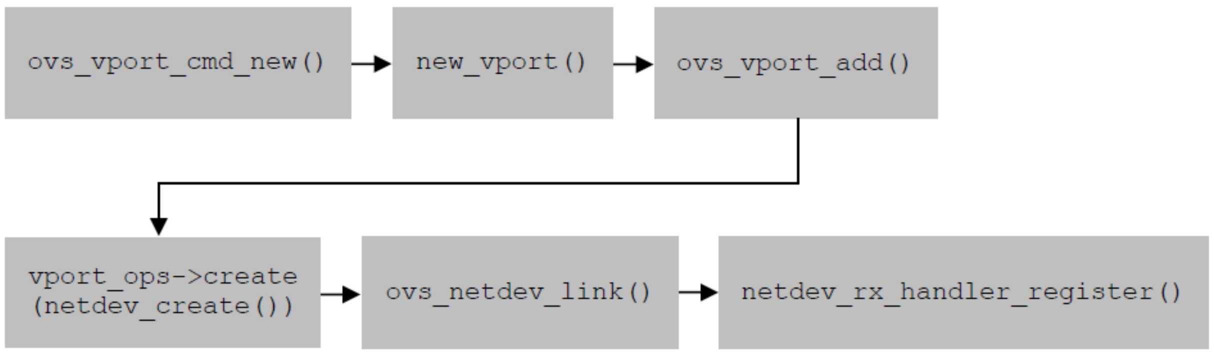
There are 5 types of virtual ports, and the following is how they are defined in "openvswitch.h":

```
enum ovs_vport_type
{
    OVS_VPORT_TYPE_UNSPEC, // just a placeholder
    OVS_VPORT_TYPE_NETDEV, /* network device */
    OVS_VPORT_TYPE_INTERNAL, /* network device implemented by datapath */
    OVS_VPORT_TYPE_GRE, /* GRE tunnel. */
    OVS_VPORT_TYPE_VXLAN, /* VXLAN tunnel. */
    OVS_VPORT_TYPE_GENEVE /* Geneve tunnel. */
};
```

`OVS_VPORT_TYPE_NETDEV` is the most popular one which we may use as an example to demonstrate how the packets are redirected into OvS datapath. This vport type is instantiated and set as the `type` enum parameter in "vport-netdev.c" as follows:

```
static struct vport_ops ovs_netdev_vport_ops = {
    .type   = OVS_VPORT_TYPE_NETDEV,
    .create  = netdev_create,
    .destroy = netdev_destroy,
    .send   = dev_queue_xmit,
};
```

In `struct vport` definition, dev points to the physical network device. When we attach the physical netdev to the datapath, the first and most fundamental thing it does is to register an `rx_handler` for it. The whole processing diagram is shown in the Figure below.


ovs_vport_cmd_new()
new_vport()
ovs_vport_add()
vport_ops->create (netdev_create())
ovs_netdev_link()
netdev_rx_handler_register()

With `rx_handler` registered, the packets are redirected into the datapath without entering the Linux legacy network stack.

flow

OvS maintains a flow cache table in kernel space, where the `sw_flow` data structure is the representation of a specific flow table entry. The following is its definition from the "flow.h" header file:

```
struct sw_flow
{
    ...
    struct sw_flow_key key;
    struct sw_flow_mask *mask;
    struct sw_flow_actions __rcu *sf_acts;
    ...
};
```

`struct sw_flow_key` collects the details of the packet including L2-L4 header information, while `struct sw_flow_actions` contains the actions to perform on the packet matching the `key` and its corresponding `mask`.

The following is the structure of the flow table inside the kernel, as defined in "flow_table.h":

```
struct flow_table
{
    ...
    struct table_instance __rcu *ti;
    struct mask_cache_entry __percpu *mask_cache;
    struct mask_array __rcu *mask_array;
    unsigned int count; // nr of flow entries in the flow table
    ...
};
```

Inspecting the `table_instance` data structure, also defined in "flow_table.h", we can see the `flex_array` is used as the hash buckets implementation. It is meant to replace cases, like this one, where an array-like structure has gotten too big to fit into `kmalloc()`. For more details about the flex_array API see [16].

Datapath Processing

In this section we will take a picture of the whole kernel datapath processing scheme. When a packet comes into NIC, it will be handled by `netif_receive_skb()`.

As mentioned before, OvS registers an `rx_handler` for the netdev which hijacks entering packets into the OvS datapath instead of the Linux legacy network stack. We then get the whole processing scheme shown in Figure 4.3.

netdev_frame_hook()

This is the hook function the OvS kernel module registered as the `rx_handler`. Its only processing is to call `netdev_port_receive()` via the `port_receive()` macro defined in "vport-netdev.c" as follows:

```
#define port_receive(skb) netdev_port_receive(skb, NULL)
```

netdev_port_receive()

Achieves the information of `vport` from `skb->dev`, after which it makes a copy of the packet represented by skb.

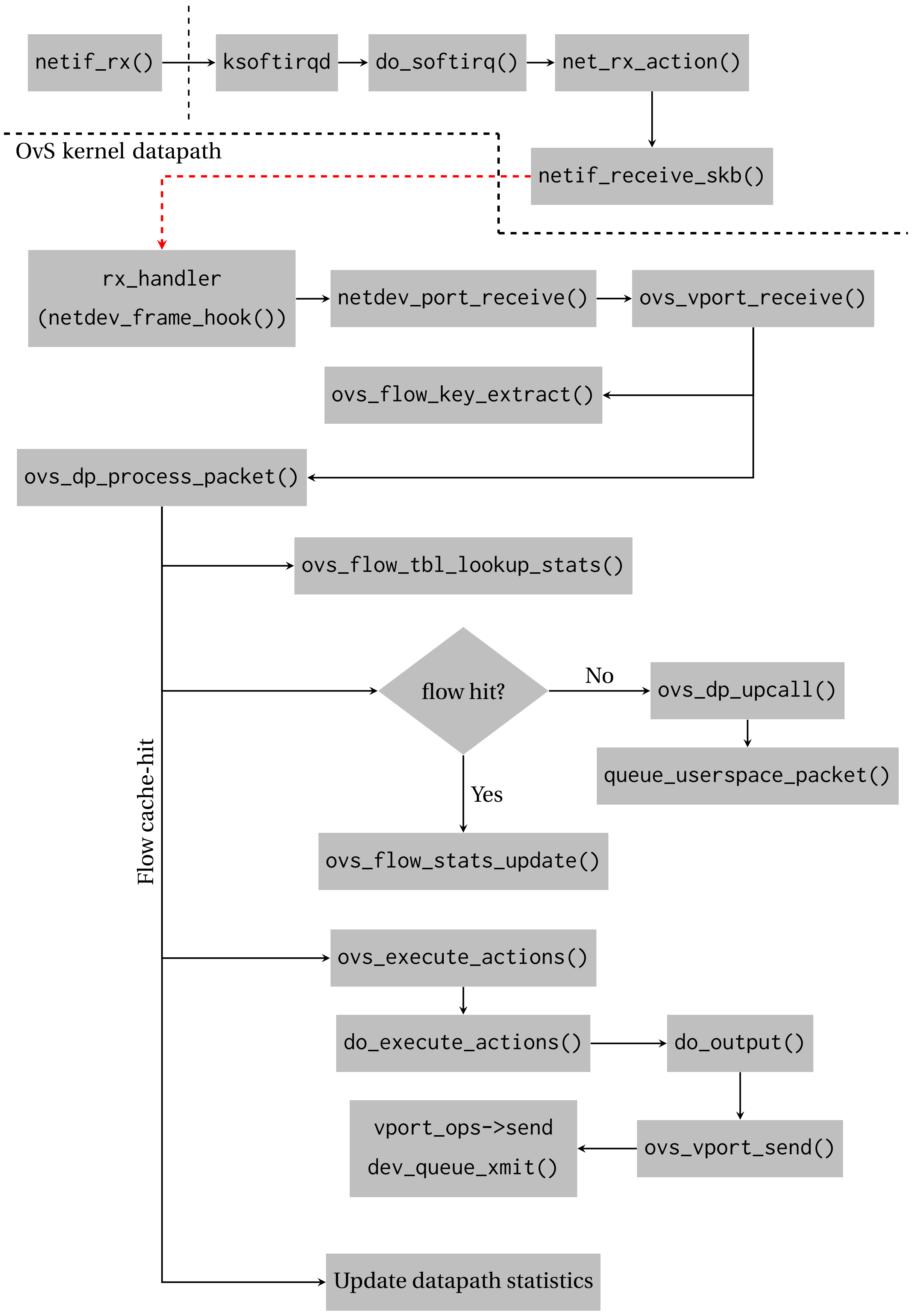


Figure 4.3: OvS kernel module packet processing main sequence

`ovs_vport_receive()`

After extracting the flow key from the received packet via `ovs_flow_key_extract()`, it passes up the packet to the datapath for processing.

The input vport reference is saved here, specifically in the *skb control block*, which in this case is used to save the input vport reference.

The following are the function's main lines of code as implemented in "vport.c":

```
{
    // Save the input 'vport' reference inside the skb's control block
    OVS_CB(skb)->input_vport = vport;
    ovs_flow_key_extract(tun_info, skb, &key); // Extract flow from 'skb' into 'key'
    /* Pass up the received packet and its extracted flow key
     * to the datapath's main packet processing routine
     */
    ovs_dp_process_packet(skb, &key);
}
```

`ovs_flow_key_extract()`

This is the datapath's *flow key parser*. It extracts the datapath's notion of a flow key, scanning the L2-L4 headers of the received packet and storing the extracted information in a data structure container represented by a `struct sw_flow_key` instance.

It first extracts the metadata information, after which the headers scanning takes place.

Now, we are given with two options for how to implement the new GTP flow parsing:

1. either *natively*, i.e. extending `struct sw_flow_key` with GTP tunnel parameters recognition and extracting the tunnel information directly into this extended flow key container;

2. or via a dedicated vport GTP tunnel implementation (also called *logical port* in OpenFlow terminology), i.e. extracting the tunnel information into the Linux kernel's `ip_tunnel_key` data structure defined in the "include/net/ip_tunnels.h" header file as follows (excluding the IPv6 related parameters).

```
struct ip_tunnel_key
{
    __be64 tun_id;
    struct
    {
        __be32 src;
        __be32 dst;
    } ipv4;
    __be16 tun_flags;
    u8 tos;
    u8 ttl;
    __be16 tp_src;
    __be16 tp_dst;
};
```

The GTP TEID should be stored, as for this option, into the 64-bit field `tun_id` parameter.

Because of a more intuitive and consistent approach, the *native* option has been chosen as the implementation for the new flow key GTP tunnel parser. Therefore, the `sw_flow_key` data structure has been extended consequently as highlighted in Table 4.1.

phy	eth	ip	tp	ipv4		gtp
priority	src	proto	src	addr	arp	ipv4_dst
in_port	dst	tos	dst	src	src_ha	teid
	tci	ttl	flags	dst	trgt_ha	
	type	frag				

Table 4.1: The datapath's flow key representation (excluding IPv6)

The last, "gtp", column is the component added to `struct sw_flow_key`, where the tunnel's IPv4 destination address and its TEID are stored in `ipv4_dst` and `teid` parameters, respectively.

The tunnel's destination address together with its TEID represent an unique way for identifying a GTP tunnel, i.e. an EPS bearer.

`ovs_dp_process_packet()`

This routine embodies the datapath's core processing engine. With the flow key extracted by the datapath's flow key parser, it performs the flow table lookup for which actions should be applied on the packet. In case of a successful lookup, the actions associated with the corresponding flow entry are executed, i.e. `ovs_execute_actions()` gets called. Otherwise, the packet is sent to `ovs-vswitchd` along with its extracted flow key, i.e. the `ovs_dp_upcall()` routine is invoked.

The following is a snapshot of its main implementation from "datapath.c":

```
{
    const struct vport *p = OVS_CB(skb)->input_vport;
    struct datapath *dp = p->dp;
    struct sw_flow *flow;
    struct sw_flow_actions *sf_acts;
    u32 n_mask_hit;
    ...
    struct sw_flow *flow;
    ...
    /* Look up flow. */
    flow = ovs_flow_tbl_lookup_stats(&dp->table, key, skb_get_hash(skb), &n_mask_hit);
    if (unlikely(!flow)) // if flow table cache-miss
    {
        struct dp_upcall_info upcall;

        memset(&upcall, 0, sizeof(upcall));
        upcall.cmd = OVS_PACKET_CMD_MISS;
        ...
        ovs_dp_upcall(dp, skb, key, &upcall);
        ...
    }

    /* Flow table cache-hit. */
    ovs_flow_stats_update(flow, key->tp.flags, skb);

    // get actions from the matching flow entry
    sf_acts = rcu_dereference(flow->sf_acts);
    ovs_execute_actions(dp, skb, sf_acts, key);
    ...
}
```

`ovs_execute_actions()`

Invokes `do_execute_actions()` which is the function actually executing the actions specified in the matching flow entry. It simply iterates over the `sw_flow_actions` buffer instance, scanning and executing all the specified actions one by one.

The two most fundamental and powerful function implementations, as this thesis project contribution, are the ability to encapsulate and decapsulate in/from GTP tunnels. Both of them are implemented in the "actions.c" source file, same as the `do_execute_actions()` implementation.

Let us see how it iterates over all the actions specified in the matching actions buffer, along with the GTP encapsulation and decapsulation function calls:

```
/* Execute a list of actions against 'skb'. */
static int do_execute_actions(struct datapath *dp, struct sk_buff *skb,
                              struct sw_flow_key *key, const struct nlattr *attr,
                              int len)
{
    int prev_port = -1; // vport number
    const struct nlattr *a;
    int rem;

    // iterate over the actions buffer
    for (a = attr, rem = len; rem > 0; a = nla_next(a, &rem))
    {
        if (unlikely(prev_port != -1))
        {
            struct sk_buff *out_skb = skb_clone(skb, GFP_ATOMIC);
            if (out_skb)
                // forward packet to the specified vport
                do_output(dp, out_skb, prev_port, key);

            prev_port = -1;
        }

        switch (nla_type(a))
        {
            case OVS_ACTION_ATTR_OUTPUT:
                prev_port = nla_get_u32(a); // vport number where to forward the packet
            break;

            case OVS_ACTION_ATTR_PUSH_MPLS:
                err = push_mpls(skb, key, nla_data(a)); // perform MPLS encapsulation
            break;

            case OVS_ACTION_ATTR_POP_MPLS:
                err = pop_mpls(skb, key, nla_get_be16(a)); // perform MPLS decapsulation
            break;

            case OVS_ACTION_ATTR_PUSH_VLAN:
                err = push_vlan(skb, key, nla_data(a)); // perform VLAN encapsulation
            break;

            case OVS_ACTION_ATTR_POP_VLAN:
                err = pop_vlan(skb, key); // perform VLAN decapsulation
            break;
            .
            .
            .
            case OVS_ACTION_ATTR_PUSH_GTPV1:
                err = push_gtpv1(skb, key, nla_data(a)); // perform GTP encapsulation
            break;

            case OVS_ACTION_ATTR_POP_GTPV1:
                err = pop_gtpv1(skb, key); // perform GTP decapsulation
            break;
        }
        ...
    }
    ...
}
```

4.3.2 Encapsulation and Decapsulation in/from GTP

The ovs-dpctl is the administrative command controlling the creation, modification, and deletion of Open vSwitch datapaths, including the routines we're interested in, i.e. insertion and removal of flow entries directly in OvS kernel datapath. As project concerns, that OvS admin command has been extended with the GTP support as an API for the admin which wants to specify GTP flow entries.

As of this point, we already know that GTP is a very special and simple kind of tunneling protocol. However, there is a not so trivial problem with the GTP API extension in ovs-dpctl. That is, the mandatory matching of the Ethernet header fields. That is a problem because GTP does not require an L2 header, so the mandatory Ethernet header matching makes no sense in the GTP context. We will see the adopted solution soon.

ovs-dpctl implements the insertion of flow entries with the add-flow command, for example:

```
root# ovs-dpctl add-flow "in_port(1), eth(src=00:11:22:33:44:55,dst=11:22:
33:44:55:66), eth_type(0x0800), ipv4(src=192.168.1.2,frag=no)" "2"
```

adds a flow entry matching packets entering the OvS vport numbered "1", having the specified Ethernet and IPv4 headers, and forwarding them to the OvS vport numbered "2" as the corresponding action. Note the matching on the Ethernet header, that is eth to match on MAC addresses, and eth_type to match the 2-bytes ethertype header field. This ovs-dpctl utility requires a mandatory matching of at least one MAC address, and ethertype in case of L3-L4 header match. This mandatory L2 header matching is the first issue we have to solve.

From Chapter 2, we know that a flow entry is comprised of two main components: a "matching" part, and an "actions" part. The actions part is optional, meaning that if it is not specified, then all the packets that match the matching part will be dropped by default. The solution for skipping the mandatory L2 header matching was to implement a separated command dedicated to GTP flow entries only, i.e. flow entries with the matching part on GTP tunnel parameters and/or with the actions part specifying GTP encapsulation / decapsulation. This dedicated command implementation was chosen especially as a strategy for treating GTP flow entries as a special case. This way, it is possible to tell the datapath if the flow entry it has to add is GTP-related (call it *GTP flow entry*) or not. Simply adding a boolean flag, called is_gtp, to the struct sw_flow definition in "flow.h" header file, allows to identify the GTP flow entries from the other ones. It is set in a generic netlink callback routine, called `ovs_flow_cmd_new_gtp()`, implemented specifically for the installation of new GTP flow table entries.

The adopted solution allows you to add a GTP flow entry as follows:

```
root# ovs−dpctl add−gtpu−flow "ipv4(dst=8.8.8.8,frag=no)"
"push_gtp(src=192.168.10.1,dst=192.168.10.2,teid=1234), 1"
```

The GTP flow entry added from above is matching non-fragmented IPv4 packets destined to `8.8.8.8`, encapsulating them in a GTP tunnel sourced from `192.168.10.1`, `192.168.10.2` as tunnel endpoint, and with `1234` as its corresponding TEID, and finally forwarding the encapsulated packet to the specified OvS vport identified by its vport number "`1`".

The decapsulation counterpart can be specified as:

```
root# ovs−dpctl add−gtpu−flow "gtp(teid=1234,dst=192.168.10.2)"
"pop_gtp, 2"
```

Where "`gtp`" stands for matching GTP traffic which TEID is `1234` and `192.168.10.2` as the corresponding tunnel endpoint; "`pop_gtp`" simply tells the datapath to extract the packet carried inside this GTP tunnel; and, at last, "`2`" indicates the OvS vport where to forward the GTP decapsulated user traffic.

Now that we are familiar with the GTP flow entry adopted solution, i.e. the implementation of a new `add-gtpu-flow` command in the `ovs-dpctl` utility tool, we are ready to see how the GTP encapsulation and decapsulation are actually implemented through the `push_gtpv1()` and `pop_gtpv1()` routines, respectively. We know, from the previous section, that these two functions are invoked from `do_execute_actions()`, which in turn is called from `ovs_execute_actions()`.

GTP encapsulate: push_gtpv1() routine

This function simply takes the UE generated traffic and places before the user packet (usually IP packet) the set of headers forming a GTP tunnel in the following order, starting from byte zero: Ethernet||IPv4||UDP||GTPv1||UE_Packet. "||" stands for concatenation, literally meaning that what is on its right side immediately follows what is specified on its left side.

The function's task, then, is to expand the headroom space in the UE's packet buffer sufficiently enough to contain the maximum possible size the set of GTP tunnel-related headers could assume. After which it will fill up correctly, and as specified by the admin, the all four blue-colored tunnel headers. Specifically:

- the TEID field in the GTPv1 header is set to the admin-specified value.
- the UDP destination port is set to 2152.
- the admin-specified source and destination IPv4 addresses are set in the (blue-colored) IPv4 header.
- the Ethernet header is populated in do_output(), which is the datapath's packet transmission routine.

The function signature is defined as follows:

```
static int push_gtpv1(struct sk_buff *skb, struct sw_flow_key *key,
                      const struct ovs_action_push_gtpv1 *gtpv1)
```

ovs_action_push_gtpv1 is a data structure storing the mandatory tunnel parameters that are used for populating the GTP tunnel-related headers.

```
struct ovs_action_push_gtpv1
{
    struct
    {
        __be32 src;
        __be32 dst;
    } ipv4;
    __be32 teid;
};
```

The snapshot below captures the push_gtpv1() main implementation:

```
static int push_gtpv1(struct sk_buff *skb, struct sw_flow_key *key,
                      const struct ovs_action_push_gtpv1 *gtpv1)
{
    // length in bytes comprising all the gtp tunnel related headers
    const size_t gtp_tun_len = ETH_HLEN + IPV4_HLEN_PLAIN + UDP_HLEN + GTPU_HLEN_PLAIN;
    struct gtpv1hdr *gtphdr;  // GTPv1 header
    ...
    /* reserve a maximum amount of headroom for containing the GTP tunnel headers */
    if (skb_cow(skb, LL_MAX_HEADER + gtp_tun_len) < 0)
        return -ENOMEM;

    /* Update the 'data' ptr in such a way that the old ethernet header gets perfectly
     * overwrited, zeroing out the space required by the gtp tunnel headers
     * (including the "old" ethernet header).
     */
    memset(__skb_push(skb, gtp_tun_len - ETH_HLEN), 0, gtp_tun_len);
    ...
    /* fill the tunnel's IP header */
    outer_iphdr->protocol = IPPROTO_UDP;   // follows UDP tunnel header
    outer_iphdr->saddr = htonl(gtpv1->ipv4.src); // tunnel's src address
    outer_iphdr->daddr = htonl(gtpv1->ipv4.dst); // tunnel's dst address
    ...
    /* fill the tunnel's UDP header */
    outer_udphdr->dest = htons(GTPv1_PORT);
    ...
    /* fill the GTPv1 header */
    gtphdr->version = 1;     // GTPv1
    gtphdr->type = GTP_MSG_TYPE_GPDU;  // G-PDU msg type
    gtphdr->teid = htonl(gtpv1->teid);  // Tunnel ID
    ...
}
```

There are, obviously, some sanity checks – not included in the snapshot for the sake of simplicity – performed before the main processing, checking if the packet to be encapsulated is an IPv4 packet, and ensuring it is not already GTP tunneled.

GTP decapsulate: `pop_gtpv1()` routine

The decapsulation routine, compared with the encapsulation one, has a very simple and straightforward implementation. Its key task is to just strip off all the GTP tunnel-related headers from the GTP-encapsulated packet.

It performs some sanity checks before proceeding to the main processing. These include checking if the packet buffer is GTPv1-encapsulated, carrying a G-PDU [1], specifically an IPv4 packet.

```
static int pop_gtpv1(struct sk_buff *skb, struct sw_flow_key *key)
{
    size_t gtp_tun_len;
    ...
    /* remove all the gtp tunnel related headers, leaving exact room to hold the
     * Ethernet header */
    gtp_tun_len = skb->len - (ntohs(GTPv1_HDR(skb)->tot_len) + ETH_HLEN);
    __skb_pull(skb, gtp_tun_len);
    ...
}
```

The call to `__skb_pull()` is really enough to accomplish the task, as it moves the `sk_buff`'s `data` pointer in a way such that it perfectly skips all the tunnel-related headers. The headroom and tailroom of a socket buffer are completely ignored as they are not part of the payload.

[1] G-PDU is a vanilla user plane message, which carries the original packet (T-PDU). In a G-PDU message, GTP-U header is followed by a T-PDU. [10]

4.3.3 Packet Transmission

The `do_output()` function, before sending out the packet, first checks if its Ethernet header is zeroed all out. If that is the case, as it is with a GTP encapsulated packet, then the source MAC address is set to that of the network device associated with the OvS output vport specified by the admin, and the 2-bytes ethertype header field is statically set to `0x0800`, as we assume to encapsulate only IPv4 traffic. The destination MAC address is, however, a whole different "beast" to deal with.

Remember, the OvS kernel datapath is replacing the legacy Linux network stack! Meaning that, once a packet is captured by the OvS datapath, it is up to the datapath logic to correctly complete all of the packet's headers before actually sending, in the form of a socket buffer, over a network device. Moreover, the `dev_queue_xmit()` function (see Figure 4.3), which is the actual sending routine called by the datapath, requires the packet buffer to be completely populated. Basically, this means that when `dev_queue_xmit()` is called, all the information required to transmit the frame, such as the outgoing device, the next hop, and its link layer address, has to be known and specified. Otherwise, the frame will be dropped because there is not enough forwarding information for a successful L2 transmission.

So, in case of a GTP encapsulated or decapsulated packet on behalf of the datapath, there is only one, fundamental, forwarding information missing. That is, the destination MAC address. It is found based on the destination IPv4 address of the socket buffer, which means we have to use ARP. However, we have to know first whether that IP address is on the same network as the host machine executing Open vSwitch or not. That is because in case the IP destination address is on a different network, then it is the IPv4 address of the gateway router that has to be used in the ARP. Finding out whether a certain IP address is on the same network or not, that is the purpose of the IP routing lookup. Therefore, we implement a function that, given the IPv4 destination address of the packet buffer, finds out the destination MAC address. Based on the IP routing lookup, it could be either of the destination host itself (meaning it is on the same network), or of the gateway router (meaning it is on a different network).

Now that we have a full knowledge about the values to use in the L2 header fields, i.e. the source and destination Ethernet address and ethertype, we can finally populate the packet's (socket buffer) Ethernet header and transmit it via a call to `ovs_vport_send()`.

The following is the function implementing all of what was explained in this section so far:

```
static int fill_ethdr(struct sk_buff *skb, const struct vport *vport)
{
    const __be32 ipv4_dst = ip_hdr(skb)->daddr;
    ...
    rtbl = ip_route_output(dev_net(vport->dev), ipv4_dst, 0, 0, 0); // routing lookup
    dst = &(rtbl->dst);
    skb->dev = dst->dev; // net_device used to reach the neighbour
    /* set the correct next-hop IP depending on the lookup */
    nexthop_ip = rtbl->rt_uses_gateway ? rtbl->rt_gateway : ipv4_dst;
    ...
    nbr = __ipv4_neigh_lookup_noref(dst->dev, nexthop_ip); // ARP cache lookup
    if (unlikely(!nbr)) // ARP cache-miss
    {
        // create ARP entry
        nbr = __neigh_create(&arp_tbl, &(nexthop_ip), dst->dev, false);
        ...
        /* Send ARP request to solve the MAC address associated
         * with 'nexthop_ip' IPv4 address.
         */
        nbr->nud_state = NUD_INCOMPLETE; // first ARP discovery for this neighbour
        arp_send(ARPOP_REQUEST, ETH_P_ARP, nexthop_ip, dst->dev,
                 ip_hdr(skb)->saddr, NULL, dst->dev->dev_addr, NULL);
    }

    ethdr = eth_hdr(skb);
    ethdr->h_proto = htons(ETH_P_IP);  // follows IPv4
    memcpy(ethdr->h_source, vport->dev->dev_addr, ETH_ALEN); // src MAC

    if (nbr->nud_state & NUD_VALID)
        memcpy(ethdr->h_dest, nbr->ha, ETH_ALEN); // dst MAC
    else
    {
        memcpy(ethdr->h_dest, broadcast_mac, ETH_ALEN); // broadcast dst MAC
        ...
    }
    ...
}
```

It is invoked inside `do_output()`, after which the frame is transmitted, as stated before, with a call to `ovs_vport_send()`.

```
static void do_output(struct datapath *dp, struct sk_buff *skb,
                       int out_port, struct sw_flow_key *key)
{
    struct vport *vport = ovs_vport_rcu(dp, out_port);
    struct ethhdr *ethdr = eth_hdr(skb);
    ...
    if (ethdr->h_proto == 0 && ETH_ADDR_IS_ZERO(ethdr->h_source) &&
        ETH_ADDR_IS_ZERO(ethdr->h_dest))
    {
        if (fill_ethdr(skb, vport) != 0)
            return kfree_skb(skb);
    }
    ovs_vport_send(vport, skb);
    ...
}
```

The book [4] is a good source to learn about "Neighboring Subsystem" and "Routing Lookup" inside the Linux kernel.

5 Testing and Evaluation

This chapter describes the network performance of the GTP-U tunnel implementation in the Open vSwitch kernel module. The tested solution is compared with the normal plain traffic, i.e. non GTP encapsulated/decapsulated and not processed by Open vSwitch, with regard to three aspects: throughput, jitter, and average latency. Finally, the analysis of these tests' results is also presented.

5.1 Test Environment

The test environment is made up of three deployed Linux-based physical machines. A machine simulates the S-GW node and UE component. Another one is playing the role of the Internet endpoint (SGi gateway). The last one, finally, is a VMware Linux guest playing at being the P-GW entity with the GTP-extended Open vSwitch installed.

We can picture the environment as illustrated in Figure 5.1 below, where the P-GW virtual machine comes installed with Debian 7, kernel image 3.16.0-0.bpo.4-amd64; has allocated 4 CPU cores of an Intel Xeon E5-2650 operating at 2.2GHz, 4GB of RAM, 50GB of disk storage, and x2 10 Gigabit Ethernet network cards (one of which fakes the S5 interface, and the other one the SGi interface).

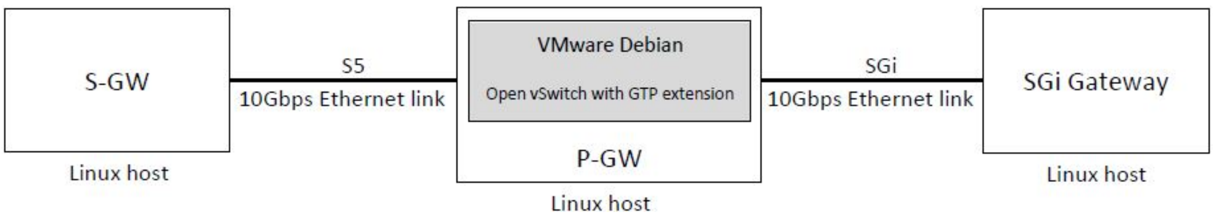


Figure 5.1: The test environment setup

Remember from the *GTP-U Tunneling section* of Chapter 3, the S5 interface carries GTP-U traffic, whereas the SGi flows genuine, i.e. original, user data traffic. Therefore, the Open vSwitch kernel module is charged with two tasks:

- Decapsulate GTP-U packets sent by the S-GW through the S5 line, and forward the decapsulated data packets to the host faking the SGi gateway via the SGi link.
- Encapsulate in a GTP-U tunnel the packets coming through the SGi which are destined to the UE faked by the host operating as an S-GW, and forward the encapsulated data packets to the S-GW simulated host via the S5 link.

The tests are performed checking the obtained UDP throughput, TCP throughput, latency, and UDP jitter. For UDP and TCP throughput tests, sender SGi Gateway offers different packet loads to receiver S-GW. The TCP/UDP throughput considered is that obtained at the receiving side. Latency is measured with a ping test, whereas the jitter for UDP packets is taken into account.

The tool used to stress the testbed with TCP/UDP traffic is Iperf 2.5 (software details can be found at iperf.fr website). The SGi Gateway machine runs the Iperf client, whereas the S-GW host runs the Iperf server side. The GTP-U tunnel extension, as implemented in this project, was implemented using Open vSwitch 2.5.0.

Before showing the tests' results, let us see how to properly configure Open vSwitch so that it can capture all the incoming traffic via both NICs playing as S5 and SGi connected interfaces.

5.1.1 Open vSwitch Configuration

We have two NICs, say the one connected to the S5 link is called `eth2`, and the other one is called `eth4` which connects with the SGi link. Open vSwitch has to embed both network devices, `eth2` and `eth4`, inside its kernel module so that it will capture all of the traffic flowing through them. The two interfaces are configured on two different subnets, as expected. As explained in [25], when you have multiple ports on different networks that have to be used by Open vSwitch, the correct way to configure such a behaviour is to allocate an Open vSwitch bridge per network. Thus, in our case, we have to create two bridges, say `S5_bridge` and `SGi_bridge`, as follows:

```
root# ovs-vsctl add-br S5_bridge
root# ovs-vsctl add-br SGi_bridge
```

After which, `eth2` and `eth4` have to be connected to their corresponding bridge, resulting in a <`eth2` – `S5_bridge`> and <`eth4` – `SGi_bridge`> mapping:

```
root# ovs−vsctl add−port S5_bridge eth2
root# ovs−vsctl add−port SGi_bridge eth4
```

Now we have to disable the IPv4 addresses used by `eth2` and `eth4`:

```
root# ifconfig eth2 0
root# ifconfig eth4 0
```

At this point, the IP addresses originally used by each port are moved to the bridge they are connected with:

```
root# ifconfig S5_bridge up
root# ifconfig S5_bridge <eth2_IPv4> netmask <eth2_netmask>
root# ifconfig SGi_bridge up
root# ifconfig SGi_bridge <eth4_IPv4> netmask <eth4_netmask>
```

Finally, all of the routing configuration using `eth2` and `eth4` are replaced with `S5_bridge` and `SGi_bridge`, respectively.

One last thing to do is to add the flow rules for encapsulation and decapsulation:

```
root# ovs−dpctl add−gtpu−flow "ipv4(dst=172.26.131.177, frag=no)"
"push_gtp(src=172.26.128.241, dst=172.26.128.177, teid=177), 3"
root# ovs−dpctl add−gtpu−flow "gtp(teid=177, dst=172.26.128.241)"
"pop_gtp, 2"
```

Figure 5.2 below shows how a proper Open vSwitch configuration should look like (in our specific test environment), where `ovs-system` is the datapath name, and 3 and 2 are the OvS virtual ports connecting `eth2` to `S5_bridge` and `eth4` to `SGi_bridge`.

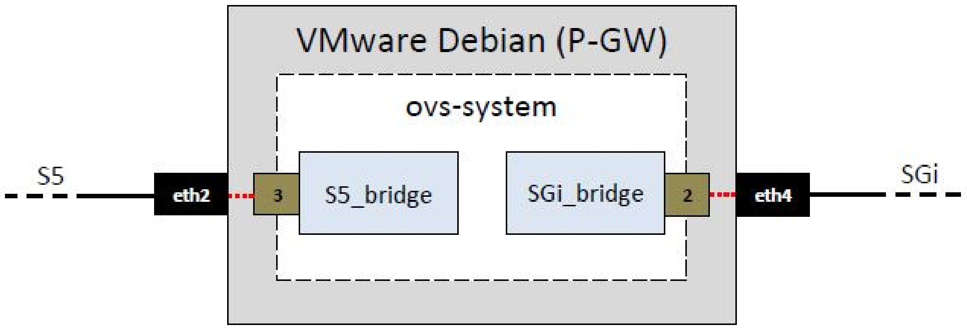


Figure 5.2: Our testbed's Open vSwitch configuration

5.2 GTP-U tunnel performance as a function of packet length and load

We examine the different behaviours of the implemented GTP tunnel solution in terms of throughput, latency, and jitter, comparing them with the results obtained using plain user data traffic.

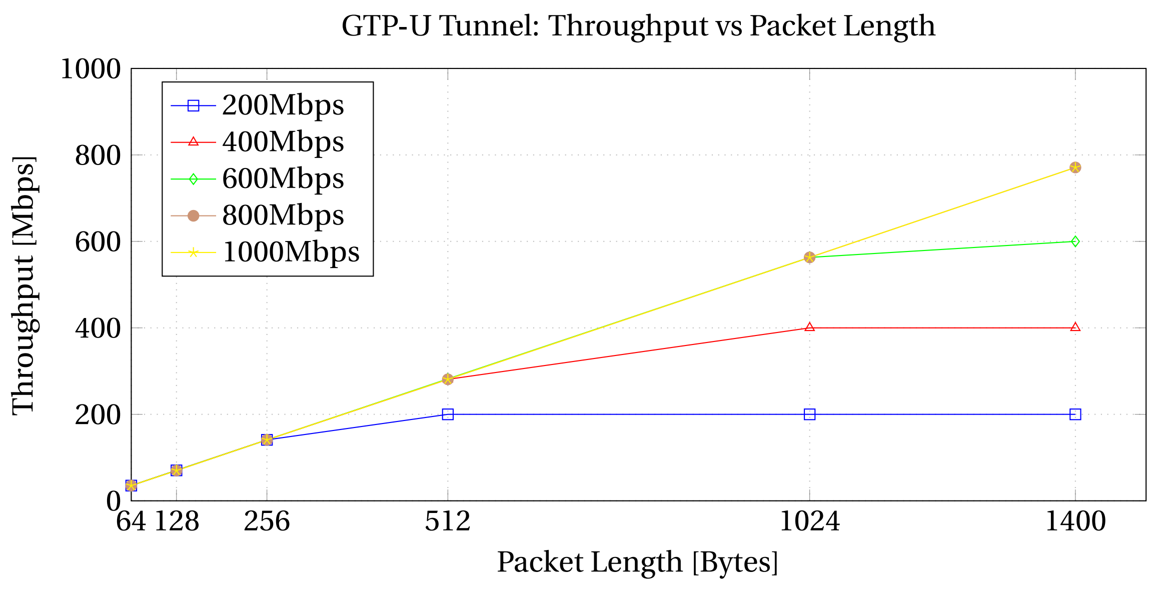


Figure 5.3: Throughput vs Packet Length (UDP) - Various offered loads, from 200Mbps to 1Gbps

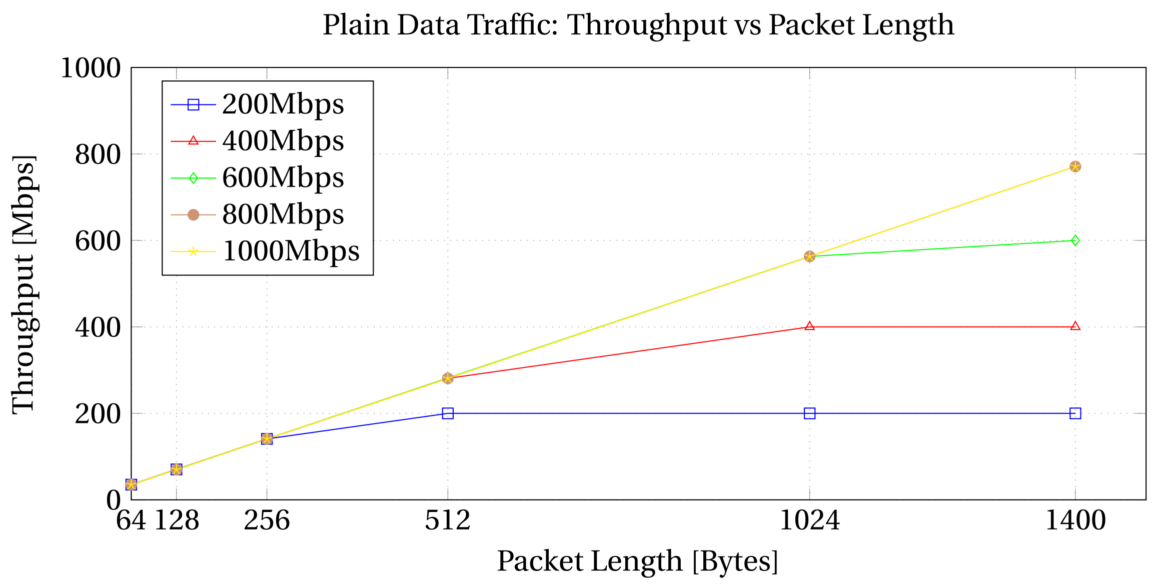


Figure 5.4: Throughput vs Packet Length (UDP) - Various offered loads, from 200Mbps to 1Gbps

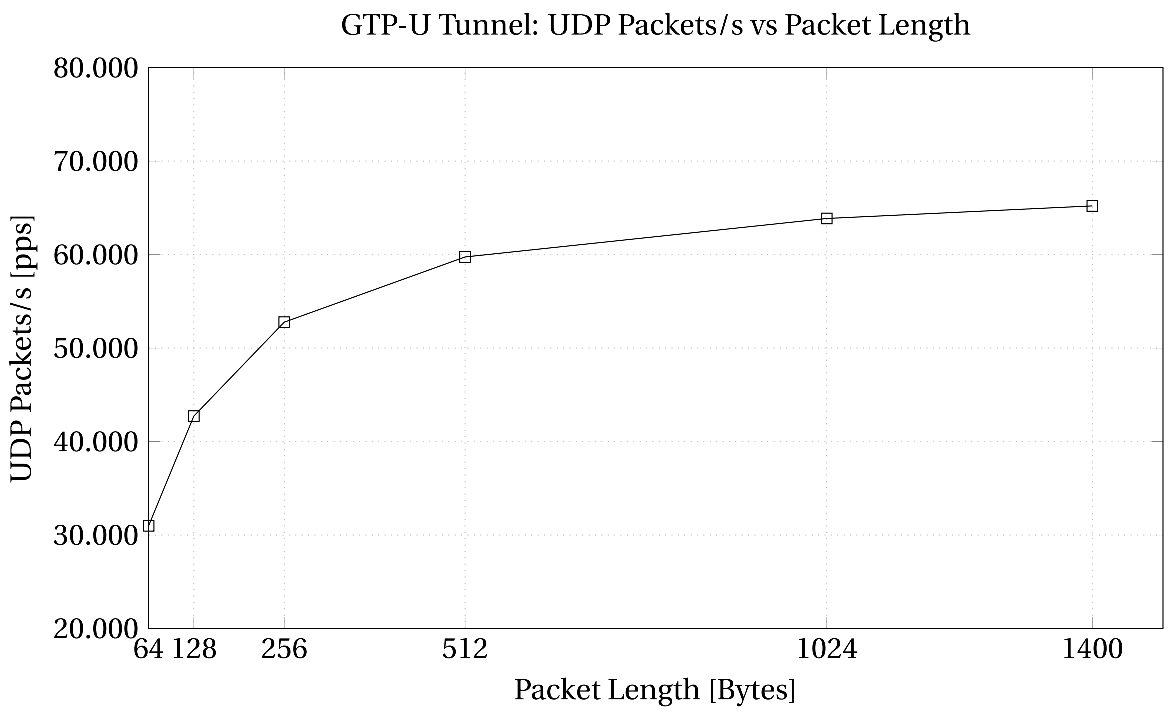


Figure 5.5: Packets/s vs Packet Length (UDP, Offered Load 1Gbps)

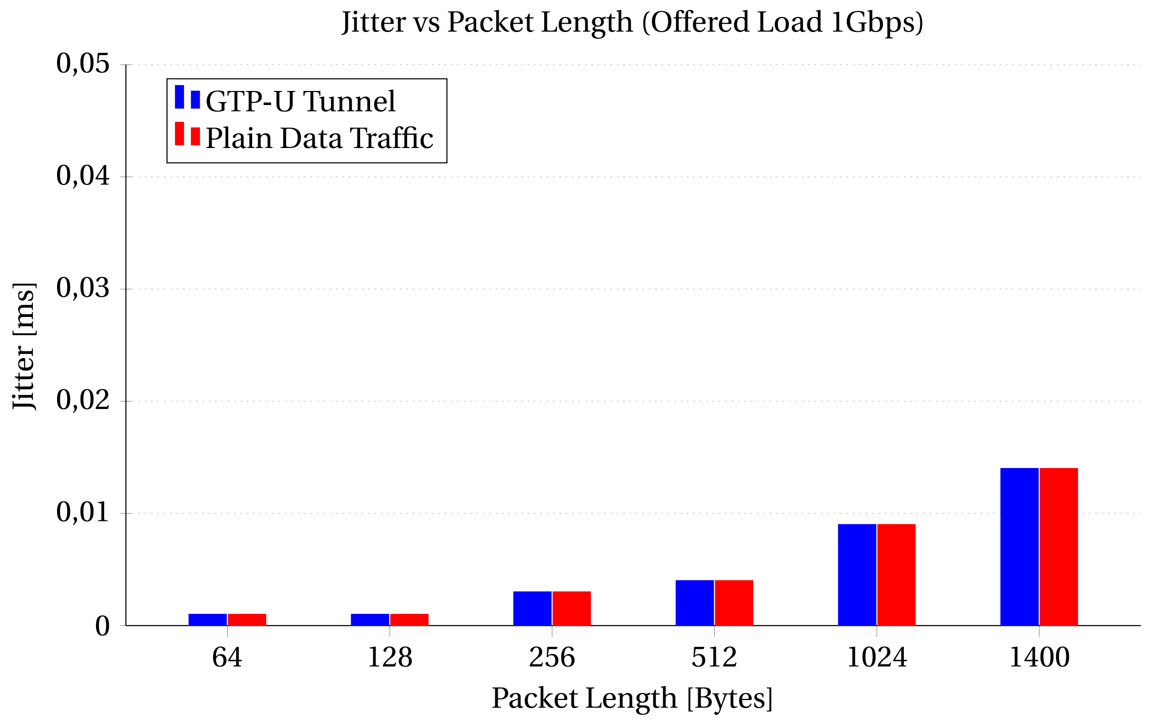


Figure 5.6: Jitter vs Packet Length (UDP, Offered Load 1Gbps)

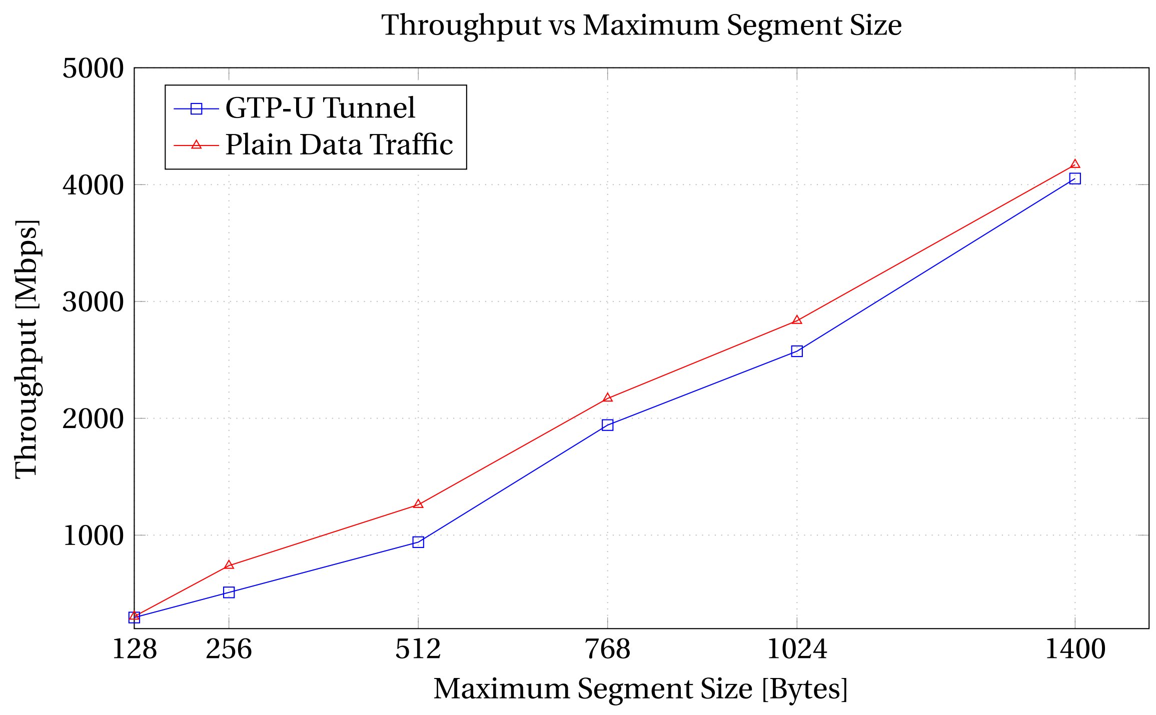


Figure 5.7: Throughput vs Maximum Segment Size (TCP)

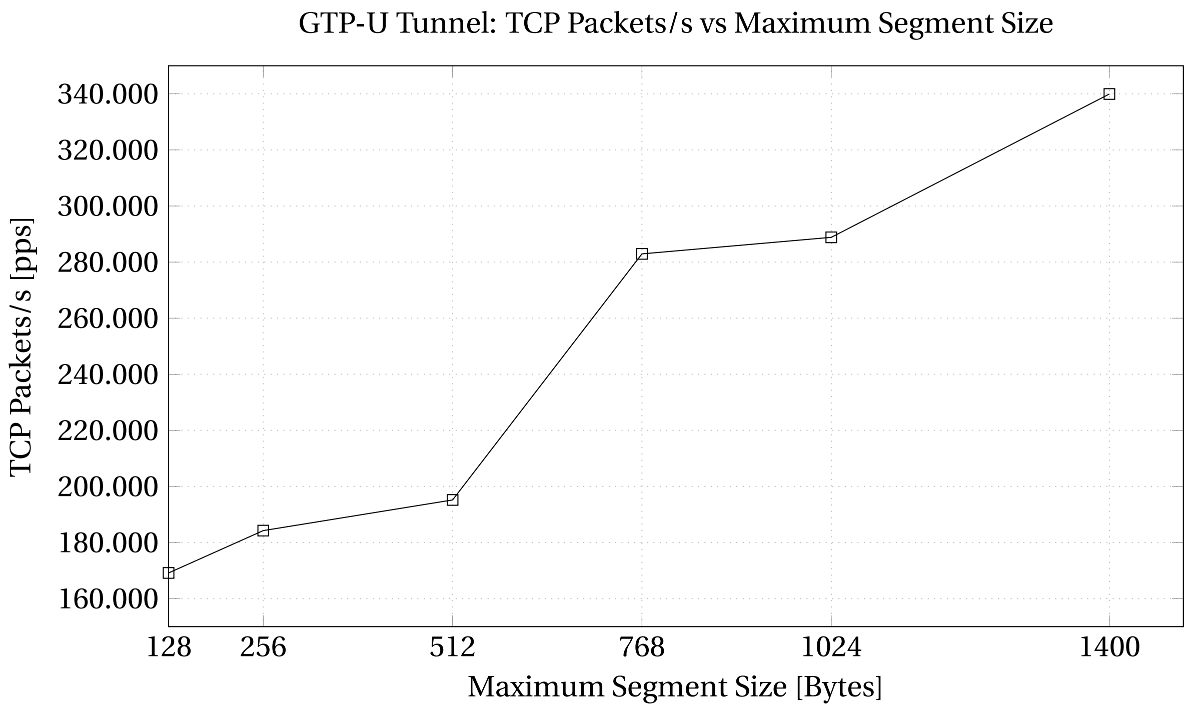


Figure 5.8: Packets/s vs Maximum Segment Size (TCP)

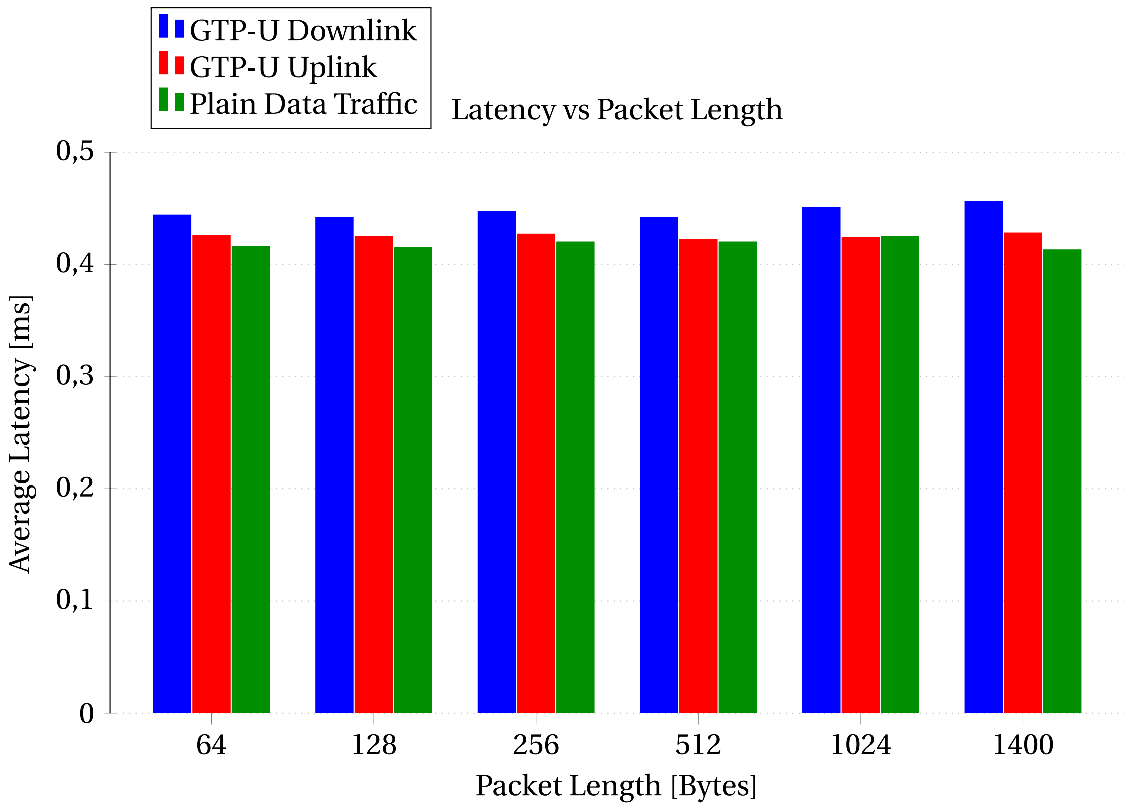


Figure 5.9: Latency vs Packet Length (ICMP)

5.3 Analysis

The previous section presents several charts for different loads and with different UDP packet sizes and TCP maximum segment sizes. The performance obtained using the GTP-U tunnel implementation is compared with the case using plain, genuine, user data traffic. The chart in Figure 5.5, however, does not plot the "plain data traffic" use case, as the UDP throughput results are identical to the GTP-U tunnel implementation with Open vSwitch.

As just stated, the two charts in Figures 5.3 and 5.4 reveal a pretty interesting fact: in case of UDP, the GTP-U solution does not show any performance degradation at all with respect to the normal, plain, data traffic use case. The same applies when testing jitter, which is why the bar chart in Figure 5.6 shows the same results for both cases, either with or without using Open vSwitch in GTP-U tunnel mode.

With TCP, as can be seen from the chart in Figure 5.7, the *GTP-U Tunnel – Plain Data Traffic* comparison is slightly different from what we have with UDP. We can see that the performance of the GTP-U tunnel is very similar when not using Open vSwitch, yet sufficiently underperforming to observe a small gap between the two lines. This is very reasonable as the Open vSwitch kernel module has to encapsulate and decapsulate GTP traffic, not to mention the overhead due to the TCP control mechanisms.

A similar behaviour is observed in the bar chart in Figure 5.9, with one difference: this time we diversify the GTP traffic between downlink and uplink flows. The downlink flow (traffic destined to the UE) is GTP-encapsulated, whereas the uplink flow (traffic sourced from the UE) is GTP-decapsulated. This chart proves that the GTP-encapsulation case is the heaviest one, the reason for this is because the OvS (Open vSwitch) datapath has to first allocate, for each packet to be encapsulated, memory required for all the GTP tunnel-related headers, and to correctly populate each header afterwards. There is obviously a penalty in latency for doing all that. In the uplink direction we have less latency with respect to the downlink counterpart, that is because the only thing the OvS datapath has to do with each to be decapsulated packet is to move the `data` offset pointer in the socket buffer by the exact amount of bytes required to skip all the GTP tunnel-related headers. Finally, we can conclude that, overall, the GTP-U tunnel processing time has a minimal impact over the average latency with respect to a non OvS-processed traffic.

The charts in Figures 5.5 and 5.8 are just a translation from their corresponding throughput results in packets per second as unit of measurement.

One thing the charts do not show is that the 10 gigabit ethernet link gets fully saturated when using either multiple TCP/UDP flows or jumbo frames over the GTP-U tunnel, proving that the implemented solution is capable of using the network cards' maximum capacity.

The performance obtained using the GTP-U tunnel implementation solution is comparable to the one obtained using plain data traffic (i.e. without involving Open vSwitch).

6 Conclusions and Future Work

This chapter introduces and elaborates some of the conclusions as intuitive outcomes and observations realized after working on the thesis project implementation and its further obtained results observing a performance comparison. As the thesis project creates a working prototype, there obviously has to be a lot of key features left for future work. The thesis, therefore, ends up with a description about what these key missing features are.

6.1 Conclusions

Throughout the thesis we have studied, analyzed, and implemented a new and evolutionary paradigm called SDN, applied within the user plane of a modern mobile core network. We proposed an evolution of the today's user plane EPC, analyzing pros and cons of the adopted approach and further identifying the most suited core network nodes on which the evolutionary approach could be applied. We also witnessed the actual implementation details of the prototype, noticing impressive flexibility and power of the SDN-based software switch, i.e., Open vSwitch. A careful study about how the today's EPC user plane works allowed us to identify the key feature which could be heavily exploited in our SDN approach. That is the GTP protocol, which we adopted as the core implementation extension in Open vSwitch. We then tested the implemented solution in a simulated EPC environment, noticing that the performance of our GTP-U tunnel implementation was very promising. We can defend this statement by showing that the prototype's performance was comparable to the normal flowing of the genuine data traffic, i.e., not touched by any other processing external to the Linux network stack.

SDN is an excellent piece of technology that has not yet been thoroughly studied to be employed in a traditional cellular core network infrastructure. However, there already is a lot of research in place studying how to best exploit SDN techniques in several other networks, not just cellular network environment. SDN can definitely become a huge player in the evolution, and revolution of the next generation mobile networks.

6.2 Future Work

We will introduce here some of the key features left for future work. Implementing these features would mean to have a completely dynamic behaviour exploiting the best from Open vSwitch and OpenFlow.

Tunnel Flow Aging

The Open vSwitch statistics include all the packet information that has been collected about the packets flowing through the switch. We could use this information to trigger a *tunnel flow aging* process from the Open vSwitch kernel module, whenever a tunnel has not been in use for a while.

GTP Userspace Support

The thesis project adds the GTP-U support in the Open vSwitch kernel module only. Thus, the kernel module's notion of flow key is different, or better yet, richer from that of the user space module. We should extend also the Open vSwitch user space to support GTP-U, in order to have a consistent view of the same flow key notion.

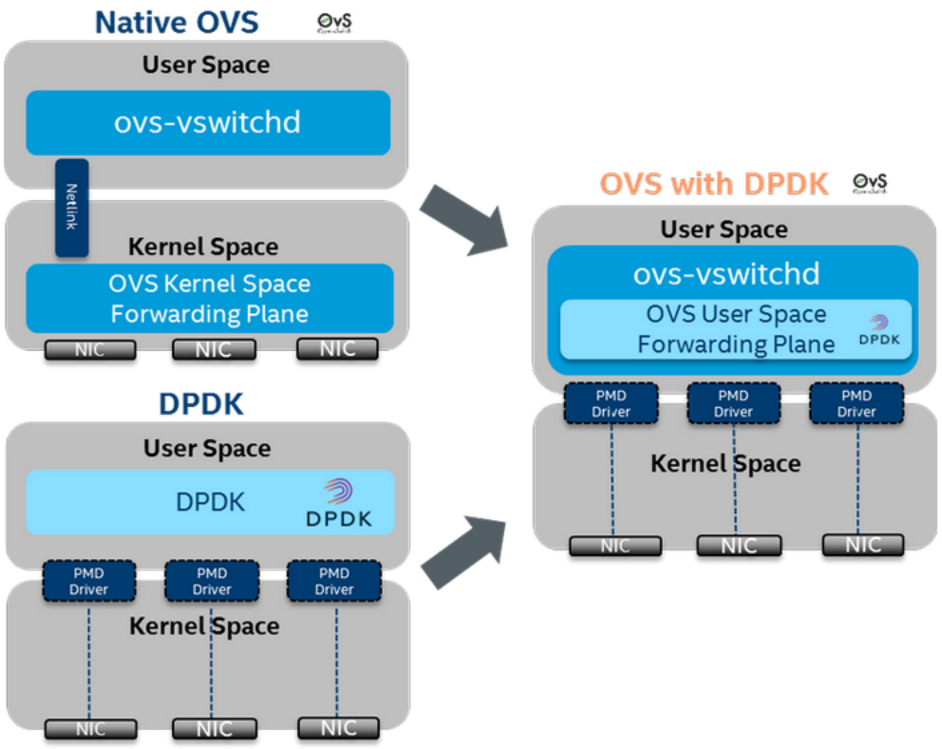
OpenFlow with GTP

A fully dynamic behaviour implies having an SDN controller capable of instructing the OpenFlow data planes with installing, removing, and modifying GTP-related flow entries. There is a nontrivial issue though, as at the time of writing the most recent OpenFlow version (i.e. 1.5.1) does not support GTP. Therefore, to get a GTP support in OpenFlow, we would have to explicitly implement a GTP extension in the OpenFlow protocol itself, which further takes us to implement a proper API support in an OpenFlow controller.

Open vSwitch with DPDK

Data Plane Development Kit (DPDK), developed by Intel along with 6WIND, is a set of libraries and NIC drivers implementing fast packet processing which entirely skips the Linux network stack, thus avoiding its forwarding bandwidth limitations [15]. Open vSwitch with DPDK underneath promises a tenfold increase over a native Open vSwitch implementation for the obtained packet processing performance.

A possible future continuation of this thesis project should definitely employ DPDK, given the potential performance enhancement, although employing DPDK would also mean that all of the changes applied to the OvS kernel module should be moved into `ovs-vswitchd`, i.e. the core Open vSwitch userspace daemon. However, writing code in user space is a lot easier than working through the kernel space, thus this should also be a big plus.
The figure below is a simple illustration of how Open vSwitch with DPDK integration outcome looks like.



Integration of DPDK with Open vSwitch. Source [15]

Open vSwitch with P4

P4 is a domain-specific programming language used to compile Programming Protocol-Independent Packet Processor program objects. Programming Protocol-Independent Packet

Processor [26] is a rather new paradigm, developed by the same research teams that invented the OpenFlow protocol standard. P4 was created to comply with SDN control protocols, which allows it to work together with, e.g., OpenFlow.

P4 allows developers to implement the packet processing logic, as what protocol headers to recognize and how to parse them, the matching logic, and the actions to be performed on matching packets. In [5] it was proved that Open vSwitch can be used with P4, the key benefit being the ease of addition of new protocols, which was the main concern of this thesis project (i.e. adding GTP tunnel support in Open vSwitch). Therefore, it would be interesting to see future implementations of Open vSwitch extensions, where the new protocols, like GTP in this case, are defined with P4.

IPv6

The implemented prototype works on IPv4 traffic only. In the real world, however, IPv6 support should be definitely covered.

Ping Latency Test

The latency evaluation experiment was performed using ping, which does not have an extreme reliability. Other more accurate tools for latency measurement should be used if what we seek is an almost 100% of latency accuracy measurements.

A Acronyms

SDN	Software-Defined Networking
EPC	Evolved Packet Core
EPS	Evolved Packet System
LTE	Long Term Evolution
GSMA	GSM Association
3GPP	3rd Generation Partnership Project
NFV	Network Function Virtualization
VNF	Virtual Network Function
ETSI	European Telecommunications Standards Institute
PoC	Proof of Concept
IMS	IP Multimedia Subsystem
MME	Mobility Management Entity
S-GW	Serving Gateway
P-GW	PDN Gateway
PDN	Packet Data Network
HSS	Home Subscriber Server
IMSI	International Mobile Subscriber Identity
GUTI	Globally Unique Temporary ID

IMEI	International Mobile Equipment ID
DNS	Domain Name System
GTP	GPRS Tunneling Protocol
ONF	Open Networking Foundation
API	Application Programming Interface
TEID	Tunnel Endpoint Identifier
QoS	Quality of Service
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communications
UMTS	Universal Mobile Telecommunications System
HSPA	High Speed Packet Access
UE	User Equipment
SAE	System Architecture Evolution
APN	Access Point Name
RAN	Radio Access Network
RAM	Random Access Memory
NAS	Network Attached Storage
KPI	Key Performance Indicator
ACL	Access Control List
IoT	Internet of Things
APN	Access Point Name
genl	Generic Netlink Family Protocol

Bibliography

[1] 3GPP TR 23.843 V12.0.0 (2013). Study on Core Network Overload (CNO) solutions.

[2] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado (2009). The Design and Implementation of Open vSwitch. *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*.

[3] blog.csdn.net (2014). Open vSwitch architecture. http://blog.csdn.net/egarle/article/details/22005177. [Online; accessed 5 December 2016].

[4] C. Benvenuti (2006). *Understanding Linux Network Internals*. O'Reilly.

[5] Cian Ferriter. Open vSwitch with P4. https://dpdksummit.com/Archive/pdf/2016Userspace/Day02-Session12-CianFerriter-Userspace2016.pdf. [Online; accessed 9 May 2017].

[6] Cisco (2016). Cisco visual networking index: Global mobile data traffic forecast. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html. [Online; accessed 22 November 2016].

[7] ETSI (2014). PoCs overview. http://nfvwiki.etsi.org/index.php?title=PoCs_Overview. [Online; accessed 23 November 2016].

[8] ETSI GS NFV 001 (2013). Network Functions Virtualisation (NFV): Use Cases. http://www.etsi.org/deliver/etsi_gs/nfv/001_099/001/01.01.01_60/gs_nfv001v010101p.pdf. [Online; accessed 24 November 2016].

[9] ETSI TS 123 401 V13.5.0 (2016). LTE; General Packet Radio Service (GPRS) enhancements for Evolved Universal Terrestrial Radio Access Network (E-UTRAN) access (3GPP TS 23.401

version 13.5.0 Release 13). http://www.etsi.org/deliver/etsi_ts/123400_123499/123401/13.05.00_60/ts_123401v130500p.pdf. [Online; accessed 28 November 2016].

[10] ETSI TS 129 281 V13.2.0 (2016). Universal Mobile Telecommunications System (UMTS); LTE; General Packet Radio System (GPRS) Tunneling Protocol User Plane (GTPv1-U) (3GPP TS 29.281 version 13.2.0 Release 13). http://www.etsi.org/deliver/etsi_ts/129200_129299/129281/13.02.00_60/ts_129281v130200p.pdf. [Online; accessed 25 November 2016].

[11] ETSI TS 136 300 V13.2.0 (2016). LTE; Evolved Universal Terrestrial Radio Access (E-UTRA) and Evolved Universal Terrestrial Radio Access Network (E-UTRAN); Overall description; Stage 2 (3GPP TS 36.300 version 13.2.0 Release 13). http://www.etsi.org/deliver/etsi_ts/136300_136399/136300/13.02.00_60/ts_136300v130200p.pdf. [Online; accessed 28 November 2016].

[12] G. Hampel, M. Steiner and T. Bu (2013). Applying Software-Defined Networking to the Telecom Domain. *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 133–138.

[13] GSMA (2014). Understanding 5g: Perspectives on future technological advancements in mobile. https://www.gsmaintelligence.com/research/?file=141208-5g.pdf. [Online; accessed 22 November 2016].

[14] H. E. Egilmez, S. T. Dane, A. M. Tekalp, and K. T. Bagci (2012). OpenQoS: An OpenFlow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks. *Signal & Information Processing Association Annual Summit and Conference, 2012 Asia-Pacific*, pages 1–8.

[15] Intel. Open vSwitch with DPDK Overview. https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview. [Online; accessed 7 May 2017].

[16] J. Corbet (2009). Flexible arrays. https://lwn.net/Articles/345273/. [Online; accessed 15 December 2016].

[17] J. Kempf, B. Johansson, S. Pettersson, H. Lüning, and T. Nilsson (2012a). Implementing EPC in a Cloud Computer with OpenFlow Data Plane. US Patent App. 13/536,838.

[18] J. Kempf, B. Johansson, S. Pettersson, H. Lüning, and T. Nilsson (2012b). Moving the mobile Evolved Packet Core to the cloud. *Proceedings of the 2012 IEEE 8th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 784–791.

[19] J. Pettit and J. Gross (2011). Open vSwitch. http://openvswitch.org/slides/LinuxSummit-2011.pdf. [Online; accessed 13 December 2016].

[20] Jyrki T. J. Penttinen (2011). *The LTE / SAE Deployment Handbook*. John Wiley & Sons.

[21] Open Networking Foundation. ONF Wireless & Mobile Working Group. https://www.opennetworking.org/images/stories/downloads/working-groups/charter-wireless-mobile.pdf. [Online; accessed 30 December 2016].

[22] Open Networking Foundation (2009). OpenFlow Switch Specification Version 1.0 (*Protocol version 0x01*). http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf. [Online; accessed 30 November 2016].

[23] Open Networking Foundation (2015). OpenFlow Switch Specification Version 1.5.1 (*Protocol version 0x06*). https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf. [Online; accessed 30 November 2016].

[24] openvswitch (2016). Why Open vSwitch? https://github.com/openvswitch/ovs/blob/master/WHY-OVS.rst. [Online; accessed 1 December 2016].

[25] openvswitch.org. Common Configuration Issues. http://docs.openvswitch.org/en/latest/faq/issues/. [Online; accessed 27 January 2017].

[26] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker (2014). P4: Programming Protocol-Independent Packet Processors.

[27] P. Moore (2006). Generic Netlink HOW-TO based on Jamal's original doc. https://lwn.net/Articles/208755/. [Online; accessed 13 December 2016].

[28] S. B. H. Said, M. R. Sama, K. Guillouard, L. Suciu, G. Simon, X. Lagrange, and Jean-M. Bonnin (2013). New control plane in 3GPP LTE/EPC architecture for on-demand connec-

tivity service. *Cloud Networking (CloudNet), 2013 IEEE 2nd International Conference on Cloud Networking*, pages 205–209.

[29] SDxCentral. Understanding the SDN Architecture. https://www.sdxcentral.com/sdn/definitions/inside-sdn-architecture/. [Online; accessed 29 November 2016].

[30] SearchSDN. Openflow protocol guide: SDN controllers and apps. http://searchsdn.techtarget.com/guides/OpenFlow-protocol-tutorial-SDN-controllers-and-applications-emerge. [Online; accessed 23 November 2016].

[31] The Linux Foundation. netlink. https://wiki.linuxfoundation.org/networking/netlink. [Online; accessed 4 May 2017].

[32] V. Srinivasan, S. Suri, and G. Varghese (1999). Packet Classification using Tuple Space Search.

[33] White Paper (2012). Network Functions Virtualisation, An Introduction, Benefits, Enablers, Challenges and Call for Action. https://portal.etsi.org/nfv/nfv_white_paper.pdf. [Online; accessed 23 November 2016].

[34] Y. Zanjireh (2015). LTE, System Architecture Evolution. http://www.slideshare.net/YousefZanjireh/lte-system-architecture-evolution-53641557. [Online; accessed 28 November 2016].