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

Dott.ssa Petrioli Chiara

Correlatore

Sasso Stefano

Candidato

Popa Pavel

Matricola: 1644389

Anno Accademico 2016/2017

Sasso Stefano

____

Firma

____

Data

Dott.ssa Petrioli Chiara

____

Firma

____

Data

Popa Pavel

____

Firma

____

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), a more mature technology in data centers, which is expected to enhance the current Long-Term Evolution (LTE) architecture. However, the integration of SDN components into the legacy network and the transition between phases need to be carefully defined.

This thesis aims to develop a Proof of Concept of a new Evolved Packet Core (EPC) SDN-based user plane architecture and to take into consideration the challenges that this technology might bring to the mobile core. The actual implementation consists of a GTP-enabled Open vSwitch [4] 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 product designed for computer networks. Our work extends Open vSwitch with an implementation of the GTP protocol. This extension enables Open vSwitch to be an excellent SDN component for 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 [15] 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 [7] 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 gathered operators and vendors to test a series of Proof of Concepts [9] with the aim to embrace NFV in their live networks.

The NFV technology allows operators to cope with the increasing traffic demands by employing a more flexible network management and better resource utilization. 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 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 in 2008 as a research project between Stanford University and University of California, Berkeley. 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]. Meanwhile vendors like Cisco and Brocade have released OpenFlow-friendly products, in addition to technology that will depend on alternate SDN approaches. 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, to increase the network capacity, it requires the deployment of new entities in specific network locations. Thus, the operation of such a static network is a costly, problematic and time-consuming process.

The NFV and SDN technologies aim to solve these issues, therefore a proper integration of these fairly new technologies in the existing EPC should be carefully analyzed. 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 [19]). 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 overprovisioning 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 [10]. 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 [22].

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. One line of research focuses on the development of a new architecture and further goes into the implementation details. For instance, Kempf *et al.* [20] present a study on the evolution of cloud-based EPC, where all the control functions of the S-GW, P-GW, and MME are lifted up into the cloud. The user plane is shifted into the OpenFlow switches, and these switches are extended to support GTP in a native fashion. OpenFlow 1.2 is extended with 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. The EPC network elements then run a simplified OpenFlow control plane, enhanced with GTP TEID routing extensions.

Another paper [14] 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 delagated 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 argue that the cellular data networks such as LTE / EPC lack the network visibility and control elasticity that enable the on-demand connectivity service. 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 [16] 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 approach is based on QoS routing where the routes of multimedia traffic are dynamically optimized to fulfill the required QoS. 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) [12] 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. The GTP protocol is a group of IP-based communications protocols used to carry GPRS within GSM, UMTS and 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 within the GPRS core network for signaling between gateway GPRS support nodes (GGSN) – the 3G equivalent of P-GW – and serving GPRS support nodes (SGSN) – the 3G equivalent of S-GW.

GTP-U is used for carrying user data within the GPRS core network and between the radio access network and the core network. The user data transported can be packets in any of IPv4, IPv6, or PPP formats.

GTP' (GTP prime) uses the same message structure as GTP-C and GTP-U, but has an independent function. It is used for carrying charging data.

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 presented.

Chapter 3 discusses the involved LTE procedures, such as *Attach Procedure* and *Default Bearer Establishment*, used to establish a data communication link with the PDN, the Internet, through the EPC. Knowledge of these procedures are useful in order to better understand the problems treated by the thesis. However, note that a detailed description of the setup operations for these discussed procedures is out of scope (the interested reader should check the 3GPP TS 23.401 specifications).

Finally, a design for a mobile network upon an SDN-based framework, at a user plane level, is proposed and analyzed.

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 a today's GTP-U maintained solution.

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

The exponential growth of cellular data services, mostly due to the smartphone usage explosion, triggered the demand for the mobile core network evolution, the current EPC. This chapter firstly introduces the EPC architecture and the operations of its main core entities in order to build the background for running NFV and SDN techniques on top of it. Therefore, a study of the benefits of employing these technologies is presented, as well as the interdependencies between them.

2.1 Evolved Packet Core (EPC)

Figure 2.1 shows the main components forming the EPC. The progression of the mobile core network is called System Architecture Evolution (SAE) which resulted in EPC, whereas the radio access component is called Evolved UMTS Terrestrial Radio Access Network (E-UTRAN). LTE and EPC together are called Evolved Packet System (EPS), where both the core network and the radio access infrastructure are fully packet-switched, contrary to the GSM, GPRS and HSPA, where all of them provide some support for the circuit-switched domain. The SAE offers many advantages over previous topologies and systems used for cellular core networks. Some of its key advantages are: improved data capacity, a flat and all IP architecture, reduced

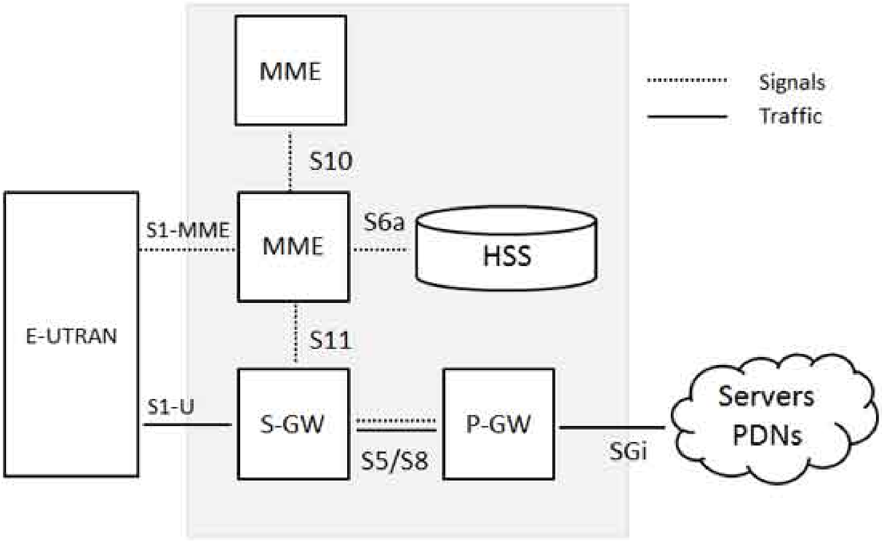


Figure 2.1: The EPC network entities, source [31]

latency, and reduced OPEX and CAPEX. The EPC architecture has been designed to provide IP connectivity between the end user, also called User Equipment (UE), and external PDNs, such as the Internet, private corporate networks or the IMS.

The EPC is mainly responsible for the control of the UE and the establishment of bearers. An *EPS bearer* [11] can be thought of as a bi-directional data pipe, which transfers data on the correct route through the network and with the correct quality of service. The basic service provided by the EPC is fundamental for an always-on support of the IP connectivity to mobile subscribers, i.e. to enable UE to get a valid IP address, to send and receive traffic to / from other IP hosts, all of this considering their mobility.

2.1.1 Serving Gateway (S-GW)

The *serving gateway* (S-GW) is a data plane element within the LTE SAE. Its main purpose is to manage the user plane mobility and it also acts as the main border between the Radio Access Network (RAN) and the core network. The S-GW also maintains the data paths between the eNodeBs, the LTE base stations, and the PDN Gateways. In this way the S-GW forms an interface for the data packet network at the E-UTRAN.
Also when UEs move across areas served by different eNodeBs, the S-GW serves as a mobility anchor ensuring that the data path is maintained.

2.1.2 PDN Gateway (P-GW)

The *packet data network gateway* (P-GW) is the EPC's point of contact with the outside world. Through the SGi interface, each P-GW exchanges data with one or more external devices or PDNs, such as the network operator's servers, the Internet or the IMS. Each PDN is identified by an *access point name* (APN). A network operator typically uses several different APNs; for example, one for the Internet and one for the IMS.

2.1.3 Mobility Management Entity (MME)

The *mobility management entity* (MME) controls the high-level operation of the mobile, by processing its signaling with the EPC via *S1-MME* interface. As with the S-GW, a typical network might contain several MMEs, each of which is responsible for a certain geographical region. Each mobile is assigned to a single MME, which is known as its *serving* MME, but that can be changed if the mobile moves too far. The MME also controls the other elements of the network, by means of signalling messages that are internal to the EPC.
In comparison with UMTS and GSM, the P-GW has the same role as the gateway GPRS support node (GGSN), while the S-GW and MME handle the data routing and signalling functions of the serving GPRS support node (SGSN). Splitting the SGSN in two makes it easier for an operator to scale the network in response to an increased load: the operator can add more S-GWs as the traffic increases, while adding more MMEs to handle an increase in the number of mobiles. To support this split, the S1 interface has two components: the S1-U interface carries traffic for the S-GW, while the S1-MME interface carries signalling messages for the MME.

2.1.4 Home Subscriber Server (HSS)

The *home subscriber server* (HSS) is a central database that contains information about all the network operator's subscribers. It is the concatenation of the Home Location Register (HLR) and the Authentication Center (AuC) – two functions that were present in GSM and UMTS networks. The HLR part is in charge of storing and updating, when necessary, the database containing all the user subscription information. The AuC part is in charge of generating security information from user identity keys. This security information is provided to the HLR and further communicated to other entities in the network.

2.1.5 Bearer Management

The EPS Bearer

LTE transports data packets using the same protocols that are used on the internet. However the transport mechanisms are very different and more complex, because LTE has to address two issues that the internet does not support. The first of these is mobility. On the internet, a device stays connected to the same access point, also technically called *default gateway*, for a long time and loses its connection to any external servers if the access point is changed. In LTE, a device expects to move from one base station (eNodeB, eNB) to another, and expects to maintain its connection to external servers when it does so. The second issue is the *quality of service* (QoS), a term that describes the performance of a data stream using parameters such as the guaranteed data rate, maximum error rate and maximum delay. The internet does not offer any QoS guarantees so that, for example, the performance of a VoIP call in a congested network is likely to be poor. In contrast, LTE can offer QoS guarantees and can assign different qualities of service to different data streams and to different users.

To address these issues, LTE transports data from one part of the system to another using *EPS bearers*. The bearer runs between the UE and the P-GW if the S5/S8 interface is GTP-based.

Default and Dedicated Bearers

One important issue is the distinction between default and dedicated bearers. The EPC sets up one EPS bearer, known as a *default bearer*, whenever a mobile connects to a PDN. As shown in Figure 2.2, a mobile receives one default bearer as soon as it registers with the EPC to provide it with always-on connectivity to a default packet data network such as the internet. At the same time, the mobile receives an IP address for it to use when communicating with that network. Later on, the mobile can establish connections with other PDNs. If it does so, then it receives an additional default bearer for every network that it connects to, together with an additional IP address.

After connecting to a PDN and establishing a default bearer, a mobile can also receive one or more *dedicated bearers* connecting to the same network. This does not lead to the allocation of any new IP addresses; instead, each dedicated bearer shares an IP address with its parent default bearer.

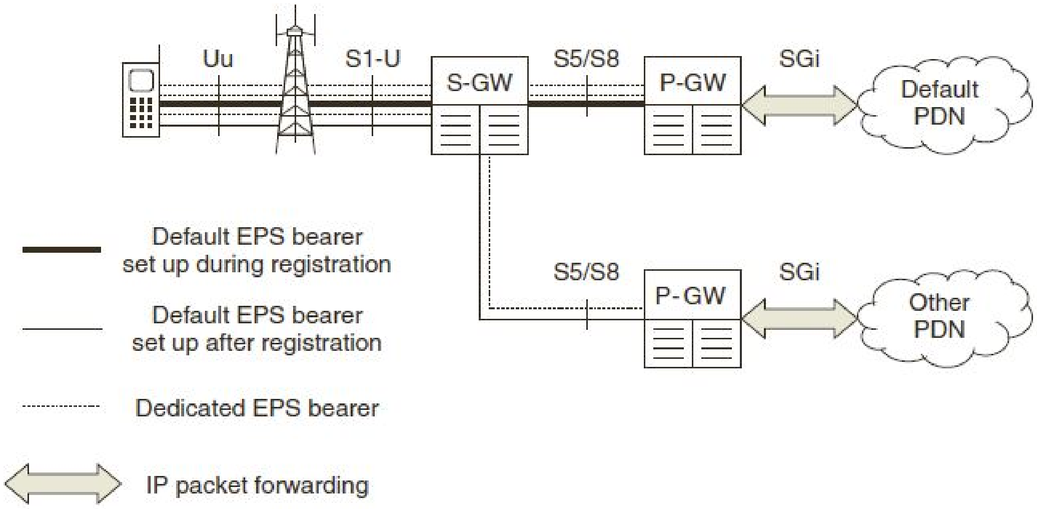


Figure 2.2: Default and dedicated EPS bearers, when using a GTP-based S5/S8 interface. Source [34]

Bearer Implementation Using GTP

The EPS bearer spans three different interfaces int the case of a GTP based S5/S8 interface. The EPS is broken down into three lower-level bearers (Figure 2.3), namely the *radio bearer*, the *S1 bearer* and the *S5/S8 bearer*. Each of these is itself associated with a set of QoS parameters and receives a share of the EPS bearer's maximum error rate and maximum delay.

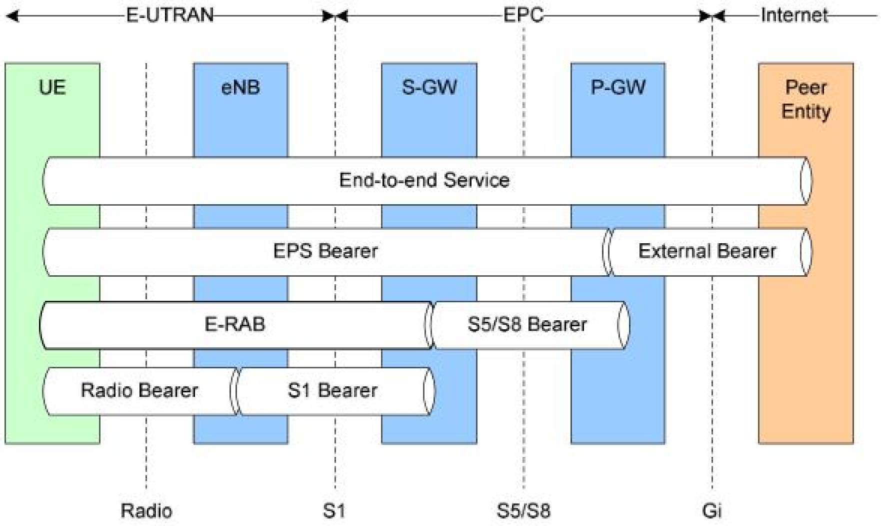


Figure 2.3: EPS Bearer Service Architecture, source [13]

The radio bearer is then implemented by a suitable 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. To show a very simple example of their operation, let us consider the downlink path from the server to the UE. Initially, the server

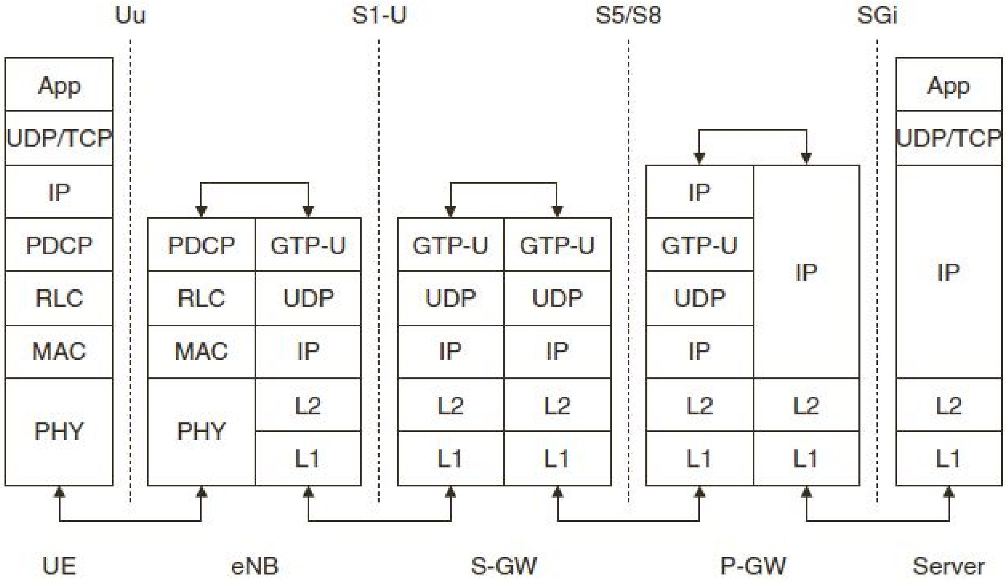
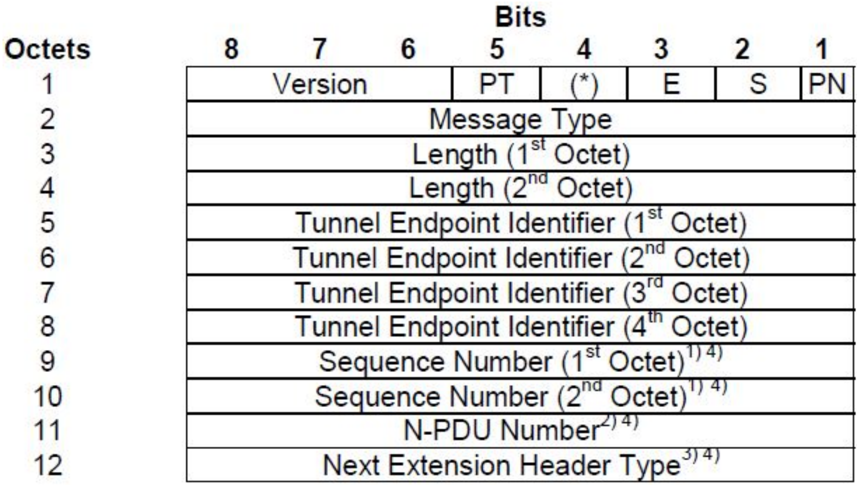


Figure 2.4: Protocol stacks used to exchange data between the UE and an external server, when a GTP based S5/S8 interface is used. Source [11]



	Bits							
Octets	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 Number2) 4)							
12	Next Extension Header Type3) 4)							

Figure 2.5: GTP-U (or GTPv1) header, source [12]

sends a packet to the UE using an IP header with the UE's IP address as the destination. The packet arrives at the P-GW, which inspects that address, identifies the target UE, and adds a second, *outer*, IP header that is addressed to the UE's S-GW. The network layer protocol then uses the S-GW's IP address to route the packet there. In turn, the S-GW uses the same mechanism to send the packet to the base station (eNodeB).

The packet forwarding process is supported by the GTP user plane (GTP-U), which labels each packet using a 32 bit *tunnel endpoint identifier* (TEID, see Figure 2.5) that identifies the overlying S1 or S5/S8 bearer. By inspecting 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 EPC, it uses UDP as the transport layer protocol and it basically adds a small header, in which, among other tunnel information, an identifier is inserted: the TEID, which univocally identifies the specific GTP tunnel for the given direction. This means that for a given (S-GW, P-GW) pair, a S-GW — P-GW GTP tunnel is different from a P-GW — S-GW GTP tunnel for that same S/P-GW pair. The reason why the user traffic is encapsulated into GTP tunnels within the EPC, is to easily handle mobility of the subscribers across base stations (the mobility within the coverage area of a single eNB is handled by the eNB itself and it does not impact the core).

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 protocol that supports establishment of tunnels for mobility management and bearers for QoS management. It is used for mobile core network specific signaling and uses UDP as the transport protocol.

2. **GPRS Tunneling Protocol User Plane (GTP-U) (GTP version 1 (GTPv1))** is a user plane protocol used for the forwarding of the user traffic via tunnels between mobile core network nodes acting as routers, specifically the S-GW, P-GW and the RAN side of the base station (eNB).

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

It is important to understand that it is the P-GW the entity which assigns the IP address to the UE: thus, because of the tunneling, the UE sees the P-GW as its first IP hop. What this means is that the P-GW is the only core network entity playing the role of default gateway 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. 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 term *network function* refers to some component of a network infrastructure that provides a well-defined functional behavior, such as intrusion detection, intrusion prevention or routing.

The Virtual Network Function (VNF) refers to the implementation of a network function using a software that is decoupled from the underlying hardware. This can lead to more agile

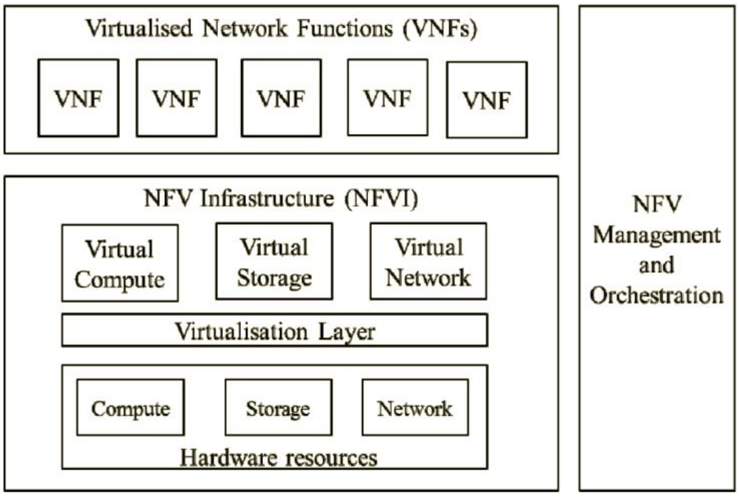


Figure 2.6: NFV architecture, source [10]

networks, with significant OPEX and CAPEX savings. The VNF is the software implementation of a network function which is capable of running 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 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 Virtualization Layer abstracts the hardware resources and decouples the software from the hardware. 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. VNFs can be connected or combined together as building blocks to offer 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)

This is the third component of the NFV architecture. 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 third and 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 the last component of the NFV architecture, and it is responsible for all the remaining management operations.

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.

The NFV might help optimizing network configuration and topology in real time based on the actual traffic / mobility patterns and service demands.

2.3 Software-Defined Networking (SDN)

Briefly mentioned in the previous sections, it was introduced with virtualization, enabling networking devices to be transformed into software. Associated with this definition, a new architecture has been defined, i.e. SDN: it decouples the data level from the control level. Up until now, forwarding tables have been computed in a distributed manner by each router or switch. In the SDN architecture, the computations for optimal control are performed by a different device, called the controller. Generally, the controller is centralized, but it can also be distributed. Before taking a closer look at this SDN architecture, let us examine the reasons for this new paradigm.

The limitations of traditional 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 objective of SDN is to reduce costs by virtualization, automation and simplification.

The architecture of SDN can be summarized with three key principles. The first is the decoupling of the physical and virtual layers (hardware and software). This enables virtual devices to be loaded on hardware machines. Therefore the network becomes hardware-independent. The second principle applies to the devices connected to the network, which

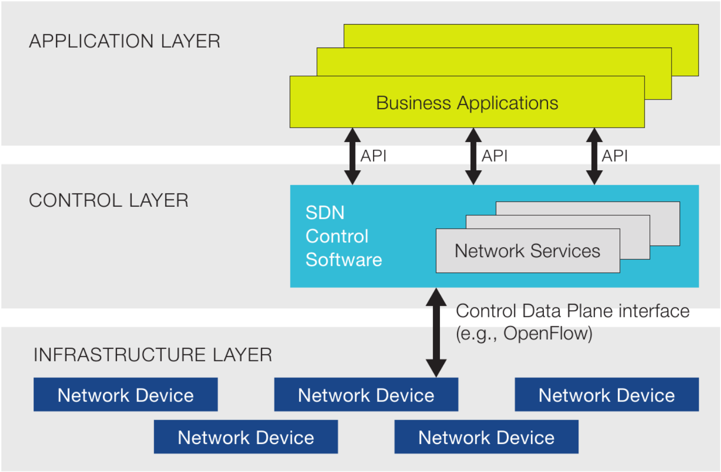


Figure 2.7: The ONF's SDN architecture, source [29]

must perceive no difference between a hardware or software machine. This new environment enables us to change the network without having to do anything to the host machines. Finally, the third principle is that of automation of the operations carried out on the network, whether for management or for control.

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 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.

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 summarise, 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

The OpenFlow protocol is a set of specifications maintained by the ONF. At the core of the specifications is a definition of an abstract packet processing machine, called a *switch*. The switch processes packets using a combination of packet contents and switch configuration state. A protocol is defined for manipulating the switch's configuration state as well as receiving certain switch events. Finally, a controller is an element that speaks the protocol to manage the configuration state of many switches and respond to events.

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

Switch Components

An OpenFlow Logical Switch, as shown in Figure 2.8, consists of one or more *flow tables* and a *group table*, which perform packet lookups and forwarding, and one or more *OpenFlow channels* to connect to external controllers.

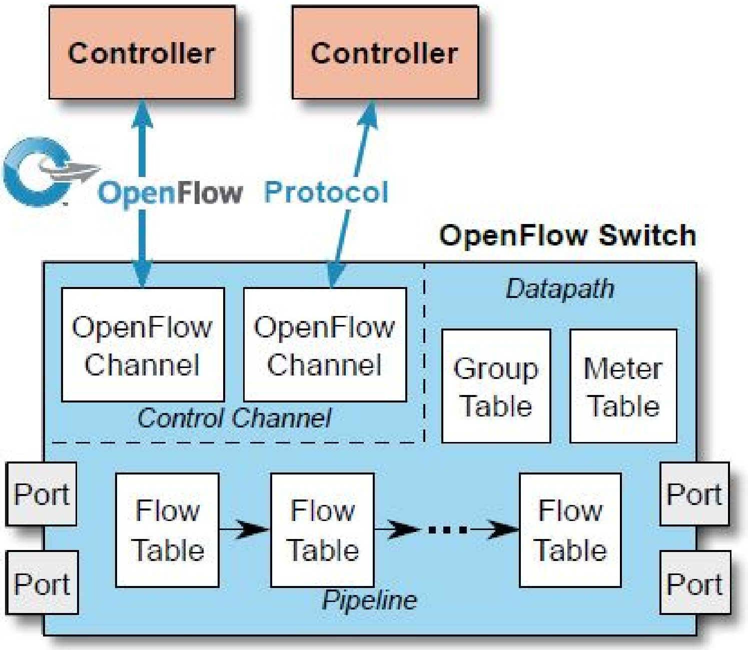


Figure 2.8: Main components of an OpenFlow switch, source [24]

Using the OpenFlow protocol, the controller can add, update, and delete *flow entries* in flow tables, both reactively (in response to packets) and proactively. Each flow table in the switch contains a set of flow entries; each flow entry consists of match fields, counters, and a set of instructions to apply to matching packets.

The matching process starts at the first flow table and may continue to additional flow tables of the pipeline. Flow entries match packets in priority order, with the first matching entry in each table being used. If a matching entry is found, the instructions associated with the specific flow entry are executed. If a flow table does not have a matching entry for the specific packet, the outcome depends on configuration of the *table-miss* flow entry: for example, the packet may be forwarded to the controllers over the OpenFlow channel, dropped, or may continue to the next flow table in the pipeline.

Instructions associated with each flow entry either contain actions or modify pipeline processing. Actions included in instructions describe packet forwarding, packet modification and group table processing. Pipeline processing instructions allow packets to be sent to subsequent tables for further processing and allow information, in the form of metadata, to be communicated between tables. Table pipeline processing stops when the instruction set associated with a matching flow entry does not specify a next table; at this point the packet is usually modified and forwarded.

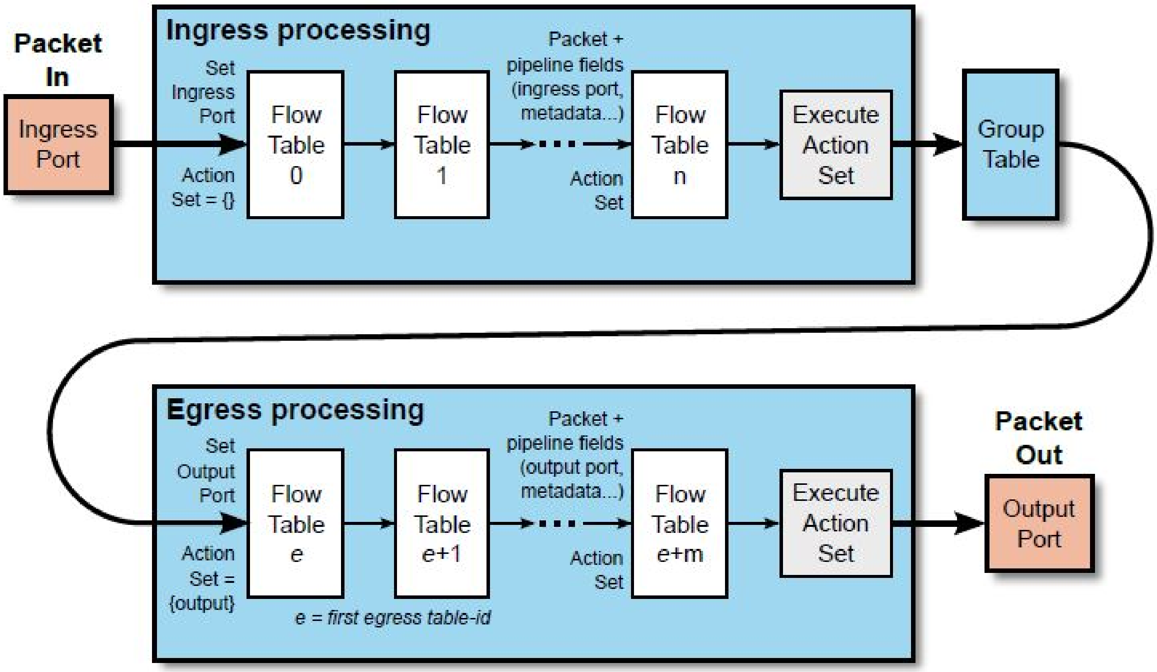


Figure 2.9: Packet flow through the processing pipeline, source [24]

OpenFlow Pipeline Processing

The are two types of OpenFlow-compliant switches: *OpenFlow-only*, and *OpenFlow-hybrid*. **OpenFlow-only** switches, as the name suggests, support only OpenFlow operation, in these switches all the packets are being processed by the OpenFlow pipeline.

OpenFlow-hybrid switches support both OpenFlow operation and normal Ethernet switching operation, i.e. traditional L2 Ethernet switching, L3 routing, ACL and QoS processing. Those switches should provide a classification mechanism outside of OpenFlow that routes traffic to either the OpenFlow pipeline or the normal pipeline.

The OpenFlow pipeline of every OpenFlow Logical Switch contains one or more flow tables, with each flow table containing multiple flow entries. The OpenFlow pipeline processing (Figure 2.9) defines how packets interact with the flow tables. The flow tables of an OpenFlow switch are numbered in the order they can be traversed by packets, starting from 0. Pipeline processing happens in two stages, *ingress processing* and *egress processing*. Pipeline processing always starts with ingress processing at the first flow table: the packet must be first matched against flow entries of flow table 0. Other ingress flow tables may be used depending on the outcome of the match in the first table. If the outcome of ingress processing is to forward the packet to an output port, the OpenFlow switch may perform egress processing in the context of that output port. However, egress processing is optional.

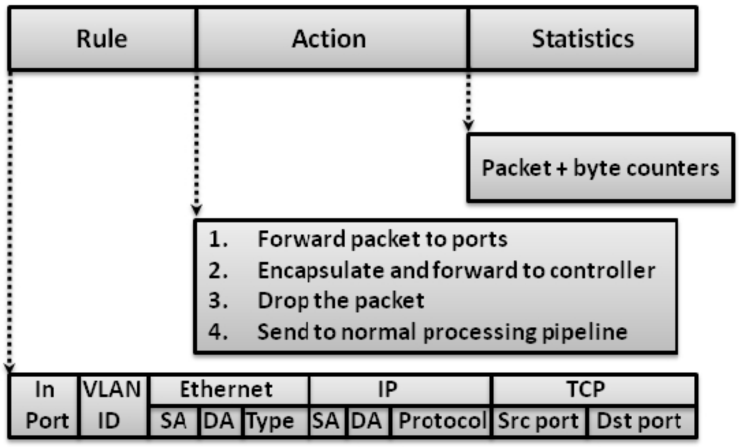


Figure 2.10: The OpenFlow v1.0 flow entry in a flow table, source [23]

When processed by a flow table, the packet is matched against the flow entries of the flow table to select a flow entry (see 5.3). If a flow entry is found, the instruction set included in that flow entry is executed. These instructions may explicitly direct the packet to another flow table, where the same process is repeated again. If the matching flow entry does not direct packets to another flow table, the current stage of pipeline processing stops at this table, the packet is processed with its associated action set and usually forwarded.

The event of a packet not matching any flow entry in a flow table is called **table miss**. The behavior at the occurrence of a table miss is defined by the table configuration. A table-miss flow entry uses several instructions in order to specify the processing which has to be applied on the unmatched packets, these include dropping them, passing them to a next table or sending them to the controllers over the control channel via *packet-in* messages. Figure 2.10 shows the concept of flow entry in OpenFlow 1.0, whereas Table 2.1 illustrates what the most updated flow entry looks like (at the time of writing the OpenFlow protocol version is at 1.5.1).

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 [24]

Each OpenFlow v1.5.1 flow entry (see Table 2.1) contains:

- **match fields**: to match against packets. These consist of the ingress port and packet headers.

- **priority**: matching order of the flow entry.

- **counters**: updated when packets are matched.
- **instructions**: to modify the action set or pipeline processing.
- **timeouts**: maximum amount of time or idle time before flow is expired by the switch.
- **cookie**: opaque data value chosen by the controller. Not used when processing packets.
- **flags**: flags alter the way flow entries are managed.

A flow table entry is identified by its match fields and priority: the match fields and priority taken together identify a unique flow entry in a specific flow table. The flow entry that wildcards all fields (all fields omitted) and has priority equal to 0 is called the table-miss flow entry.

Packet Matching

On receipt of a packet, an OpenFlow Switch performs the functions shown in Figure 2.11. The switch starts by performing a table lookup in the first flow table, and based on pipeline processing, may perform table lookups in other flow tables.

Packet header fields are extracted from the packet, and packet pipeline fields are retrieved. Packet header fields used for table lookups depend on the packet type, and typically include various protocol header fields, such as Ethernet source address or IPv4 destination address. The current state of a packet is represented by its header fields and pipeline fields.

A packet matches a flow entry if all the match fields of the flow entry are matching the corresponding header fields and pipeline fields from the packet. If a match field is omitted in the flow entry (i.e. value ANY), it matches all possible values in the corresponding header field or pipeline field of the packet. If the match field is present and does not include a mask, for a successful matching the match field has to match exactly the same value from the corresponding header field or pipeline field of the packet. If the switch supports arbitrary bitmasks on specific match fields, these masks can more precisely specify matches, the match field is matching if it has the same value for the bits which are set in the mask.

The packet is matched against flow entries in the flow table and only the highest priority flow entry matching the packet gets selected. When a successful matching occurs, the counters associated with the selected flow entry are updated and the instruction set included in the selected flow entry is executed.

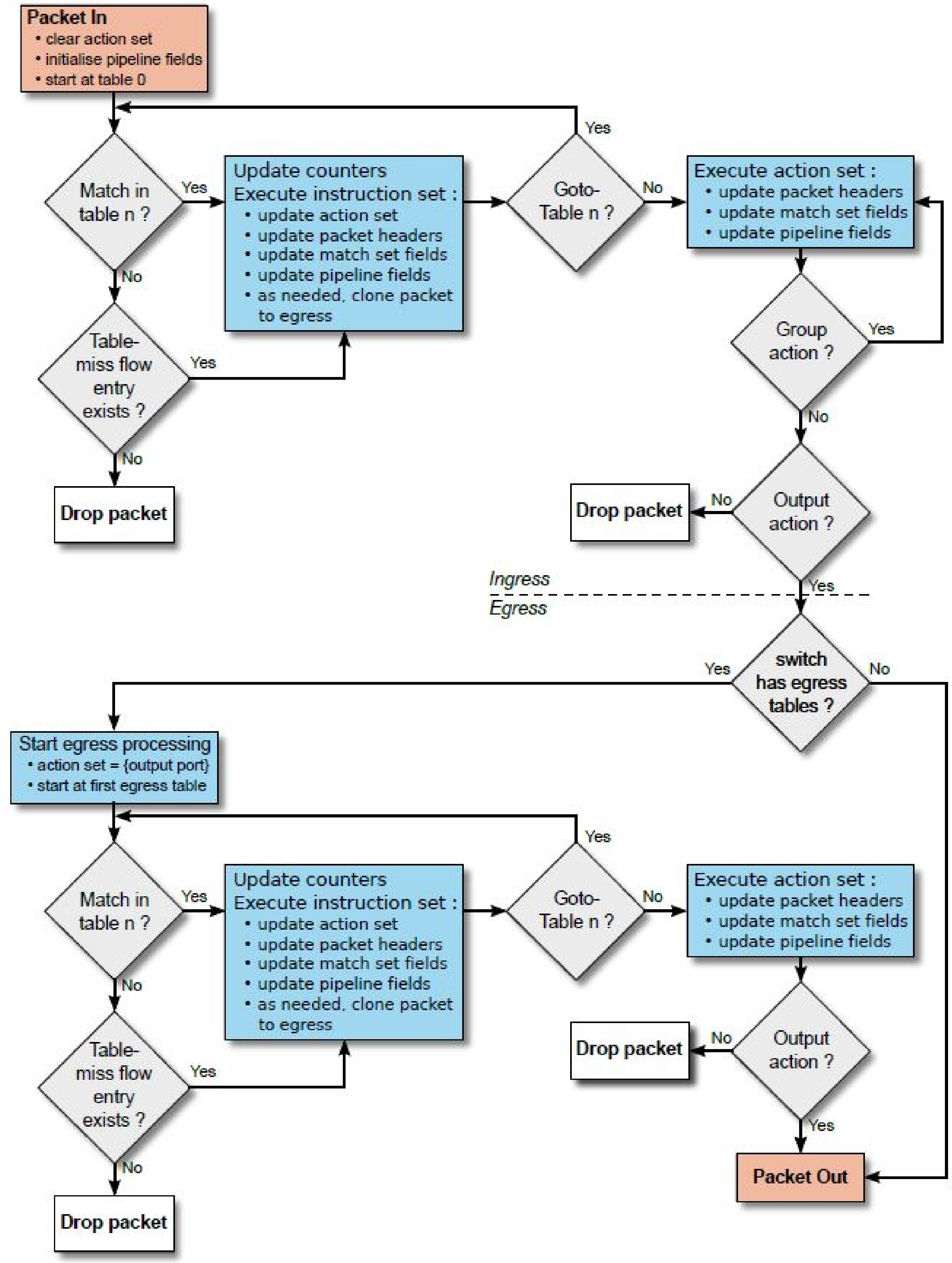


Figure 2.11: Packet flow through an OpenFlow switch, source [24]

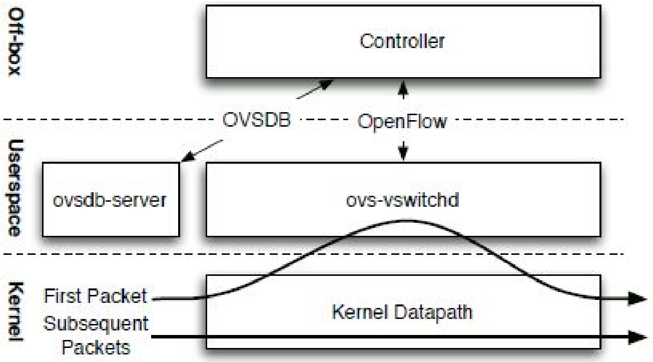


Figure 2.12: The components and interfaces of Open vSwitch. The first packet of a flow results in a miss, and the kernel module directs the packet to the userspace component, which caches the forwarding decision for subsequent packets into the kernel module. Source [4]

2.3.4 Open vSwitch design

Open vSwitch is a production quality, multilayer virtual switch licensed under the open source Apache 2.0 license. It is designed to enable massive network automation through programmatic extension, while still supporting standard management interfaces and protocols (e.g. NetFlow, sFlow, SPAN, RSPAN, CLI, LACP, 802.1ag). It is also part of the Linux kernel, since Linux kernel version 3.3 [25].

Open vSwitch has two major components, which represent the packet forwarding engine: `ovs-vswitchd` and a *datapath kernel module*. `ovs-vswitchd` is a userspace daemon that is essentially the same from one operating system and operating environment to another. The kernel module is usually written specially for the host operating system for performance reasons. Figure 2.12 shows how the two modules inter-operate to forward packets. The datapath kernel module is the first to receive the ingress packets. In case `ovs-vswitchd` has instructed the datapath what to do with packets of this type, the datapath module simply executes the given instructions, also called *actions*. Usually these actions specify at least one physical or logical port where to forward the matching packets. However, other than a simple packet forwarding, the actions may also specify packet modifications, packet sampling, or a straightforward packet dropping. Otherwise, if the datapath has not been told how to handle the specific packet type, it delivers the packet to the userspace component, `ovs-vswitchd`. In userspace, `ovs-vswitchd` decides the fate of the given packet, then it passes the packet back to the datapath with the desired handling. Usually, `ovs-vswitchd` also tells the datapath to cache the actions, in order to remember the behavior for similar future packets.

Open vSwitch is often used as an SDN switch, this is also the reason why it is being described in this section. OpenFlow is the protocol used by the SDN controllers to speak with the Open vSwitch data plane (see Figure 2.12). The Open vSwitch userspace component, `ovs-vswitchd`, simply receives OpenFlow flow tables from the SDN controller and matches any packets received from the datapath kernel module against these OpenFlow tables. If there are any successful matches, it gathers all the corresponding applied actions, and finally caches the result in the kernel datapath. From the OpenFlow controller's point of view, the caching and separation into user and kernel components are invisible implementation details: in the controller's view, each packet visits a series of OpenFlow flow tables and the switch finds the highest priority flow whose conditions are satisfied by the packet, and executes its OpenFlow actions.

Most of the packets are only processed in the kernel space module, thus it is referred as a *cache*. As previously mentioned, when Open vSwitch receives a packet from one of its virtual ports, the Open vSwitch kernel module first delivers the packet to the datapath. The datapath then generates a so-called *flow key*. This flow key contains all the information extracted from the packet headers, recognized by the Open vSwitch kernel datapath, along with other non header values such as the packet ingress port, the metadata value and other pipeline fields. The pipeline fields are a set of values attached to the packet during pipeline processing which are not header fields. These include the ingress port, the metadata value, the Tunnel-ID value and others. The metadata value is a maskable register that is used to carry information between flow tables.

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 2.13. We use "OVS" as a shorthand for Open vSwitch.

Journey of a packet through the Open vSwitch stack

With the Linux bridge, the source and destination MAC address of a received packet is inspected, learned and forwarded, after which the packet never leaves the kernel. OVS design and architecture is different although it is still connected to the same place in the Linux kernel where the Linux bridging code is nested. With OVS in place, the received network traffic is being hijacked to go through Open vSwitch, bypassing the Linux networking stack. From this point on, path determination and packet processing decisions are made entirely in the OVS user-space.

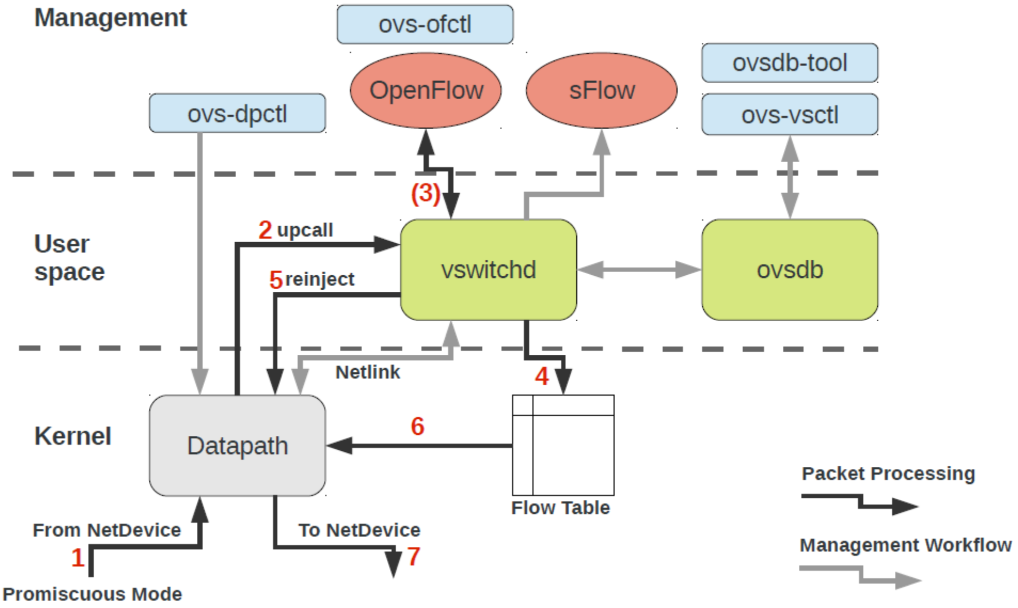


Figure 2.13: Open vSwitch's packet processing and management workflows, source [5]

The first received packet hits the OVS datapath (Figure 2.13 (1)), where a flow table lookup is being performed to determine whether the datapath has any knowledge on what actions to perform with the packet. The first packet(s) in a flow is sent up (2) to the `ovs-vswitchd` process in the user-space. Major tasks of `ovs-vswitchd` are to communicate with SDN controllers (3) using OpenFlow (TCP port 6633), interacting with the OVS database `ovsdb-server` via the management protocol (TCP port 6632) and exchanging information with the kernel module (i.e. OVS datapath) using Netlink sockets (described in Chapter 4). When a packet from the kernel is received, it is handed to the classifier which holds one or more OpenFlow tables and is a key part of `ovs-vswitchd`. For example, if a simple Layer 2 virtual switch is implemented, it makes assumptions based on L2 addresses and then decides to forward on a specific port(s). Whatever the decision is for packets belonging to that flow, `ovs-vswitchd` has to instruct the datapath about what to do with it. Because the datapath is implemented to do lightweight forwarding, it does not keep state about the packet just sent to the user-space and, therefore, does not wait for a response. A newly generated flow table entry is added (4) and `ovs-vswitchd` is forced to inject (5) the processed packet back into the datapath. The recently inserted packet is matched againt the new flow table entry of the datapath (6) and forwarded correspondingly (7) to the specified destination.

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

3 Proposed System Design

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. It is an illustration of how I expect this network architecture to evolve in the face of some of the major problems arising with the smartphone proliferation and the already in going IoT involvement. Then, in order to understand the proposed design implications, a detailed description of the main procedures used to establish a data communication link with the Internet is given.

3.1 Evolved Packet Core (EPC) Software-Defined Networking (SDN)-based User Plane Architecture

The proposed architecture, shown in Figure 3.1, aims to slightly change the existing 3GPP architecture in order to integrate the legacy core components into the OpenFlow network. The user plane functionality of the S-GW and P-GW is changed, specifically replaced with an Open vSwitch software / virtual 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.

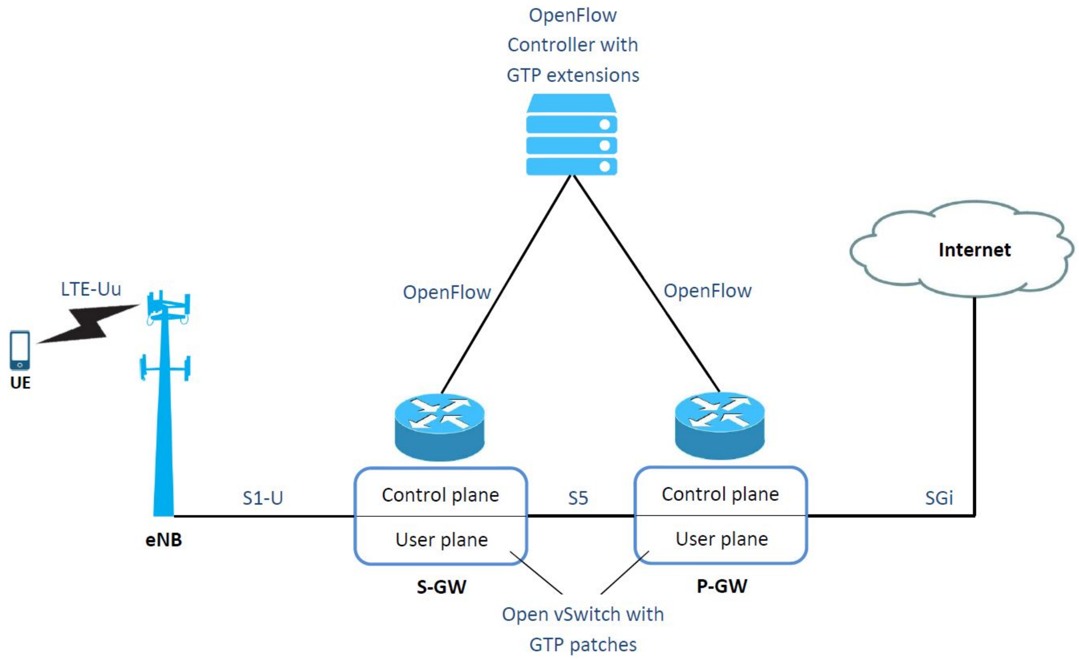


Figure 3.1: EPC SDN-based proposed architecture

3.2 The Protocol Stack Challenge

3.2.1 Challenge 1: GPRS Tunneling Protocol (GTP) encapsulation

GTP is the key protocol within EPC: in the user plane, the GTP tunnel is uniquely identified, in both directions (i.e. uplink and downlink) by a pair of TEIDs (corresponding to source and destination nodes) together with IP source and destination addresses and UDP port numbers. In the traditional EPC architecture, every bearer is identified by a tunnel and one UE can have multiple sessions corresponding to multiple bearers. Since the employed architecture is OpenFlow based, an Open vSwitch software switch is the platform used for processing all the traffic. However, the OpenFlow protocol most recent version (OpenFlow 1.5.1 [24]), at the time of writing this thesis, does not support GTP matching, neither TEID in the GTP header.

Solution 1-a)

There are two possible solutions to this problem. The first option, is to extend first Open vSwitch, and then the OpenFlow protocol in order to support GTP tunnel termination and TEIDs, respectively.

Solution 1-b)

Another possibility would be to keep Open vSwitch and OpenFlow untouched and add a separate entity (i.e. a line card) responsible for GTP tunneling termination. This should function as a *decapsulator* and it can be placed between eNodeB and the eNB-U switch. This entity main function is to remove the TEIDs. In this manner, the network between eNBs and the Internet is an IP core that could be easily managed by the controller.

3.2.2 Challenge 2: GPRS Tunneling Protocol (GTP) decapsulation

Nevertheless, there is no Ethernet header after the UDP header in the GTP stack. After GTP decapsulation, the packet will not have any Layer 2 information, thus, in order to implement the GTP protocol in the Open vSwitch, Layer 2 information is required. Hence, in case we want to extend Open vSwitch to support GTP protocol, the Layer 2 information has to be added.

Solution 2-a)

A possible solution to this problem is to create a simple Ethernet header and the packets will be forwarded based on Layer 3 information, since the Layer 2 information does not exist.

Solution 2-b)

Another option is to simply route the packets based on the available information. In this way, we take full advantage of the Open vSwitch capabilities by only specifying the input and output ports.

The performance of the first solution is expected to be better in terms of throughput than the second solution in which we enable Layer 3 mapping and modify the Ethernet header. The Layer 3 mapping and Ethernet header change will increase the overhead of the GTP-U tunnel. However, this increase in overhead should not have such of a big impact on the overall performance as the first solution for the GTP encapsulation challenge has been adopted. The reason for adopting a direct implementation of a GTP support in Open vSwitch is the need to preserve the LTE concepts of QoS, charging functions, and default and dedicated bearers in order to provide, most importantly, UE mobility.

We will now take a look at some of the major LTE procedures which, given the specific problem we want to solve (splitting the forwarding of the LTE user data traffic from the control plane), are useful to have a better understanding of how we got there in the first place.

3.3 LTE Attach Procedure

The EPS attach procedure is the most fundamental and almost certainly the most valuable, especially from a troubleshooting perspective, to know and understand. Realistically, if there is a problem with the network and subscribers are failing to establish an IP data session, the EPS attach procedure really is the place to dig in to try to understand what may be going wrong.

During the attach procedure, the mobile acquires an IP address (in our case we consider only IPv4), which it will subsequently use to communicate with the outside world. Before looking at the attach procedure itself, it is useful to discuss the methods that the network can use for *IP address allocation*.

IPv4 addresses are 32 bits long. In the usual technique, the P-GW allocates a dynamic IPv4 address to the mobile as part of the attach procedure. It can either allocate the IP address by itself or acquire a suitable one from a DHCP server. As an alternative, the mobile can itself use DHCP to acquire a dynamic IP address after the attach procedure has completed. To do this, it contacts the P-GW over the user plane, with the P-GW acting as a DHCP server towards the UE.

Because of the shortage of IPv4 addresses, the allocated address is usually a private IP address that is invisible to the outside world. Using network address translation (NAT) [17], the system can map this address to a public IP address that is shared amongst several UEs plus a UE-specific TCP or UDP port number. The P-GW can either do the task of network address translation itself or delegate it to a separate physical device.

Subsequently, the UE will use the same IP address for any dedicated bearers that it sets up with the same PDN. If it establishes communications with another PDN, then it will acquire another IP address using the same technique.

The attach procedure has four main objectives. The UE uses the procedure to register its location with a serving MME. The network gives the mobile an IPv4 address, using either or both of the previously described techniques, and sets up a default EPS bearer, which provides

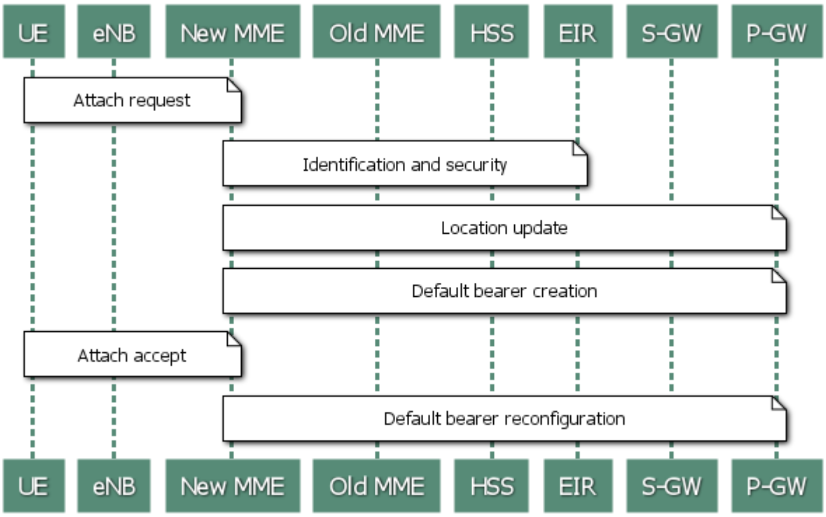


Figure 3.2: Overview of the attach procedure

the UE with always-on connectivity to a default PDN.

The sequence diagram in Figure 3.2 is a summary of the attach procedure.

3.3.1 Attach Request

It all starts with an *attach request*. The UE sends an attach request to the eNodeB, literally telling "I want to connect to the network" which is basically the message it sends. The UE includes in this message: its identifier (IMSI [1] for initial attach), supported capabilities, and whether or not it wants to request any specific IP service. The eNodeB then determines the MME and forwards to it the mobile's attach request.

3.3.2 Identification and security

The MME receives the messages from the base station and can now run some procedures that relate to identification and security.

If the mobile has moved to a new MME since it was last switched on, then the MME has to find out the mobile's identity. To do this, it extracts the identity of the old MME from the mobile's GUTI [2] and sends the GUTI to the old MME in a GTP control plane message. The old MME's response includes the IMSI and the mobile's security keys.

[1] The International Mobile Subscriber Identity or IMSI is used to identify the UE and is a unique ID associated with all cellular networks. It is stored as a 64 bit field and is sent by the phone to the network.

[2] The Globally Unique Temporary ID or GUTI is changed on a frequent basis and used instead of the IMSI in most air interface messages for security reasons.

The network can now run two security procedures: in authentication and key agreement, the UE and network confirm each other's identities and set up a new set of security keys; in NAS security activation, the MME activates those keys and initiates the secure protection of all subsequent EPS session and management functions related messages.

The MME then retrieves the IMEI (a number, usually unique, used for UE identification). If the mobile sets a specific EPS Session Management (ESM) flag in its attach request message, then the MME can now send it an ESM message requesting configuration information. The UE sends its protocol configuration options in response; for example, any access point name that the mobile would like to request. Now that the network has activated NAS security, the mobile can send the message securely.

3.3.3 Location Update

The MME can now update the network's record of the UE's location. If the UE is re-attaching to its previous MME without having properly detached (for example, if its battery ran out), then the MME may still have some EPS bearers that are associated with the mobile. If this is the case then the MME deletes them by performing the *detach procedure*.

If the MME has changed, then the new MME sends the mobile's IMSI to the HSS. The HSS updates its record of the mobile's location and tells the old MME to forget about the mobile. If the old MME has any EPS bearers that are associated with the mobile, then it deletes these as before.

Then, the HSS answers the new MME, including the user's *subscription data* [2]. The subscription data lists all the access point names (APNs) that the user has subscribed to, each of them is defined using an *APN configuration* where one of the APN configurations is identified as the default. In turn, each APN configuration identifies the access point name, states whether the corresponding PDN supports IPv4, IPv6 or both, and defines the default EPS bearer's QoS.

3.3.4 Default Bearer Creation

The MME now has all the information that it needs to set up the default EPS bearer. It begins by selecting a suitable P-GW, using the UE's preferred APN if it specified one, or the default APN otherwise. It then selects an S-GW and sends it a GTP control plane message, including the relevant subscription data and identifying the UE's IMSI and the destination P-GW.

The S-GW receives the message and forwards it to the P-GW. In this message, the S-GW includes a GTP-U TEID, which the P-GW will eventually use to label the downlink packets that it sends across the S5/S8 interface.

The P-GW now acknowledges the S-GW's request by means of a GTP control plane message, including in it any IP address that has been allocated to the UE, as well as the QoS of the default EPS bearer. The P-GW also includes a TEID of its own, which the S-GW will eventually use to route uplink packets across S5/S8. The S-GW forwards the message to the MME, except that it replaces the P-GW's tunnel endpoint identifier with an uplink TEID for the base station to use across S1-U.

3.3.5 Attach Accept

The MME can now reply to the mobile's attach request. It first initiates an ESM procedure which starts with a message that includes the EPS bearer identity, the APN, the QoS and any IP address that the network has allocated to the mobile.

The MME embeds the ESM message in to an EPS Mobility Management (EMM) message, which is a response to the UE's original attach request. The message includes a list of tracking areas in which the MME has registered the mobile and a new globally unique temporary ID.

In turn, the MME embeds both messages into a message request. This is the start of a procedure which tells the eNodeB to set up an S1 signalling connection for the UE, and S1 and radio bearers that correspond to the default EPS bearer. The message includes the bearers' QoS, the uplink TEID that the MME received from the S-GW and a key for security purposes. The MME sends all three messages to the eNodeB.

On receiving the message, the eNodeB can then compose a signalling message for setting up a radio bearer that will carry the default EPS bearer. It sends this message to the UE, along with the EMM and ESM messages that it just received from the MME.

The mobile reconfigures then its Radio Resource Control (RRC) [3] connection as instructed and sets up the default EPS bearer. It then sends its acknowledgements to the network in two stages, where the first one triggers a response control message to the MME, which includes a downlink TEID for the S-GW to use across the S1-U interface.

The UE can now send uplink data as far as the P-GW. However, we still need to tell the S-

[3] The Radio Resource Control (RRC) protocol is used in UMTS and LTE on the air interface.

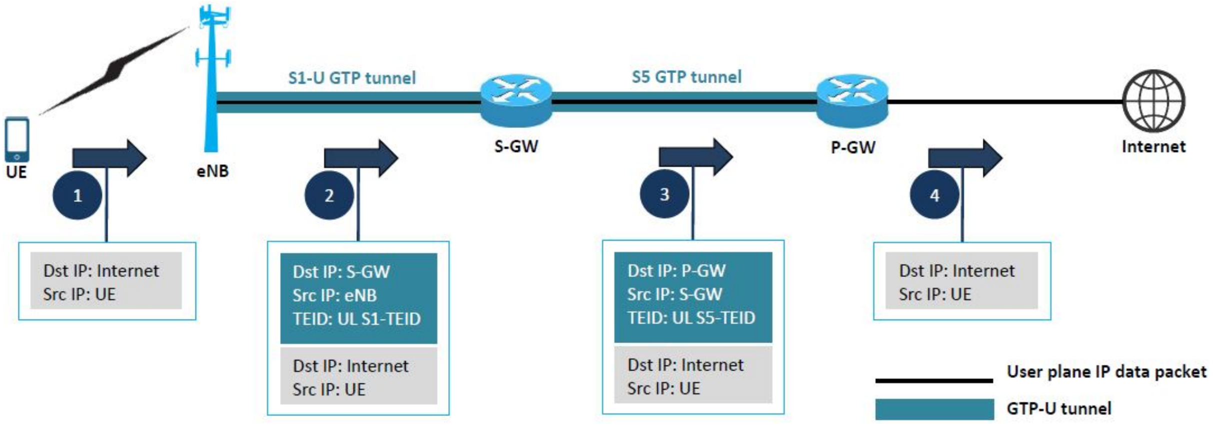


Figure 3.3: An IP packet sent by the UE to the Internet

GW about the identity of the selected eNodeB and send it the TEID that the eNodeB has just provided. To do this, the MME sends a GTP control plane message to the S-GW and the S-GW responds. From this point on, downlink data packets can flow to the mobile and vice versa.

3.4 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 3.3, IP packets sent by an UE are delivered from an eNodeB (eNB) to a P-GW through GTP user plane tunnels. What this means is that "*all IP packets that an UE sends are always sent through an eNB to a P-GW regardless of their specified destination IP addresses (i.e., the P-GW is the UE's default gateway)*".

Taking Figure 3.3 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 original packet sent by the UE will look something like shown in Figure 3.4.

   Notice the lack of Ethernet header, that is because the UE sends the packet on an LTE-Uu interface (i.e. LTE radio link), not on an Ethernet one!

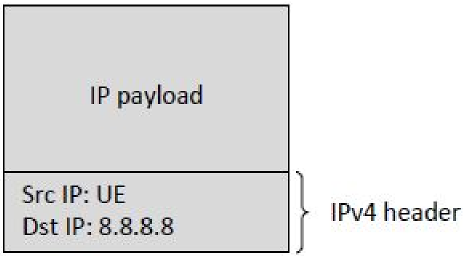
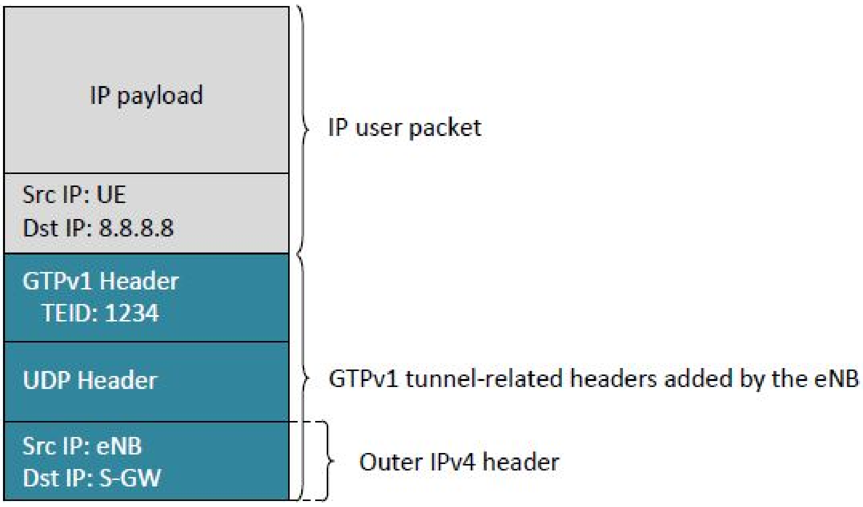


Figure 3.4: UE generated packet

2. **eNB to S-GW**

   Upon receiving the IP packet from the UE, the eNB adds a GTPv1 tunnel header, consisting of three individual headers — a GTPv1 header, UDP header, and an IPv4 header for the GTP tunnel — in front of the UE's IP packet. Then, the resulting IP packet (sent by the eNB to an S-GW) will be as follows:



   So, if only an IP routing network exists between the eNB and the S-GW, the routing network performs routing based on the destination IP address of the packet (in this example, the IPv4 address of the S-GW, which is the destination IP address specified in the outer IPv4 header), and then delivers the IP packet to the 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.

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

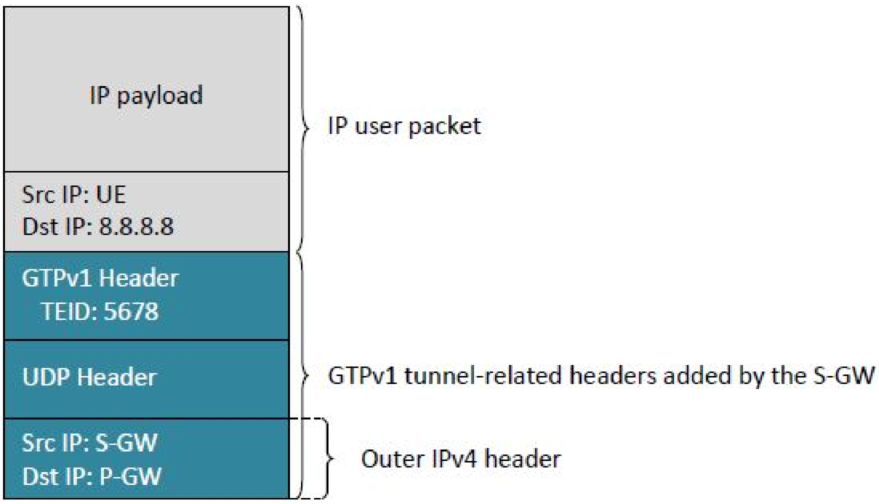


Figure 3.5: The IP user packet encapsulated in a GTP-U tunnel

packet (see Figure 3.4) sent by the UE to the Internet.

Now it is the right time to give a full explanation about these GTP TEIDs. Let us say there are 10 UEs that are connected to the same <S-GW, P-GW> pair. Since one GTP tunnel is generated per UE (though more than one can be practically generated), 10 GTP tunnels will be created. Now, the LTE network has to be able to distinguish which GTP tunnel belongs to which UE. For this purpose, a TEID is assigned to each UE. So, for example, the TEID is marked as TEID = UL S1-TEID (ex. 1234) for the link from the eNB to the S-GW and as TEID = UL S5-TEID (ex. 5678) for the link from the S-GW to the P-GW, in reference to Figure 3.3.

Now that TEIDs specific to UEs are used, the LTE network can distinguish its subscribers (UEs) from one another by checking their TEIDs instead of IP addresses (P-GWs check both TEIDs and IP addresses of UEs, eNBs and S-GWs check TEIDs only).

TEIDs are also unidirectional. That is, they can only serve for one direction, either uplink or downlink. Meaning that for the reverse traffic, from the Internet to the UE, a new TEID is assigned and used for the links from the P-GW to the S-GW and from the S-GW to the eNB.

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.

The GPRS tunneling protocol (GTP), as already mentioned, is a very central protocol for carrying subscriber's IP packets through the EPC. It is essential to implement GTP in the SDN switch that we will use, specifically in Open vSwitch which has been adopted in this project. 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 has been applied to the kernel module only. At last, the actual GTP implementation will be briefly described, with some minor code references.

4.1 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 kernel module implements multiple *datapaths* (analogous to bridges), each of which can have multiple *vports* (analogous to ports within a bridge). Each datapath also has associated a *flow table* which the user space populates with *flows* that map from keys based on packet headers and metadata to sets of actions. The most common action forwards the packet to another vport; other actions are also implemented.

When a packet arrives on a vport, the kernel module processes it by extracting its *flow key* and looking it up in the flow table. If there is a matching flow, it executes the associated actions. If there is no match, it queues the packet to user space for processing (as part of its processing, user space will likely set up a flow to handle further packets of the same type entirely in-kernel).

A flow key is passed over a Netlink socket as a sequence of Netlink attributes. Some attributes represent packet metadata, defined as any information about a packet that cannot be extracted from the packet itself, e.g. the vport on which the packet was received. Most attributes, however, are extracted from headers within the packet, e.g. source and destination addresses from Ethernet, IP, or TCP headers.

The `openvswitch.h` header file defines the exact format of the flow key attributes. For informal explanatory purposes here, we write them as comma-separated strings, with parentheses indicating arguments and nesting. For example, the following could represent a flow key corresponding to a TCP packet that arrived on vport 1:

```
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)
```

A wildcarded flow is described with two sequences of Netlink attributes passed over the Netlink socket. A flow key, exactly as described above, and an optional corresponding *flow mask*.

A wildcarded flow can represent a group of exact match flows. Each ’1’ bit in the mask specifies an exact match with the corresponding bit in the flow key. A ’0’ bit specifies a "don’t care" bit, which will match either a ’1’ or ’0’ bit of an incoming packet.

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 background of what is happening when a packet enters OVS datapath, we can focus onto each of its major packet processing steps, studying their corresponding main problems and looking at the OVS implementation approach.

Algorithmic packet classification is expensive on general purpose processors, and packet classification in the context of OpenFlow is especially costly due to the generality of the form of the match, which may test any combination of Ethernet addresses, IPv4 and IPv6 addresses, TCP and UDP ports, and many other fields, including packet metadata such as the ingress port.

Open vSwitch uses a *tuple space search* classifier [32] for all of its packet classification, both kernel and user space.

4.1.1 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 computations possible. That is what the packet classification problem is.

4.1.2 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.

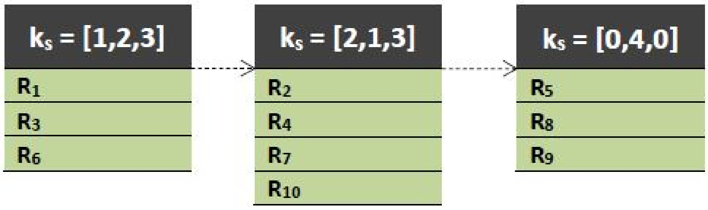
Consider, for example, the following set of rules:

	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]

Notice 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.

4.2 Some Critical Data Structures

This section describes some of the most fundamental data structures used in the OVS kernel module as a prerequisite for understanding how to evolve the datapath logic and how to read its code.

4.2.1 Linked Lists

To use the list mechanism, we must include the "linux/list.h" header file. This file defines a simple structure of type `list_head`:

```
struct list_head
{
    struct list_head *next, *prev;
};
```

Linked lists used in real kernel code are almost invariably made up of some type of structure, each one describing one entry in the list. To use the Linux list facility in your kernel code, it is necessary to only include a `list_head` inside the structures that make up the list. Suppose your kernel code has to maintain a list of things to do, say, its declaration would look something like the following:

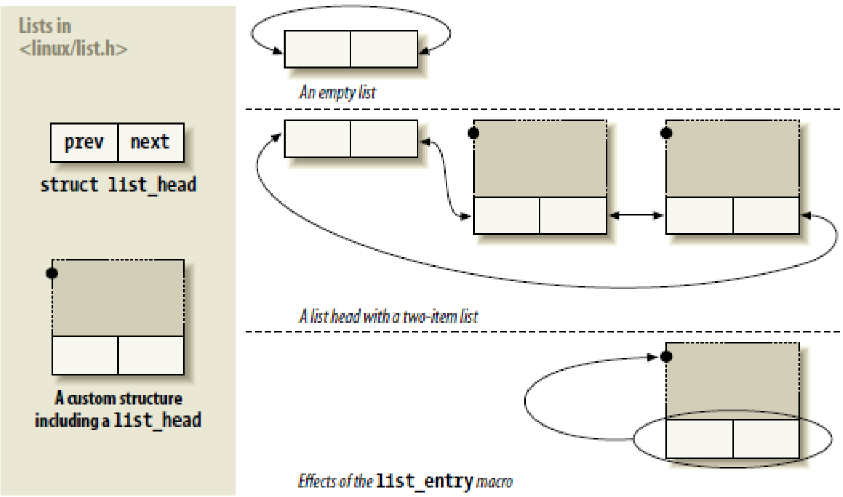


Figure 4.1: The `list_head` data structure, source [3]

```
struct todo_struct
{
    struct list_head list;
    int val;
    ...
};
```

The head of the list is usually a standalone `list_head` structure. Figure 4.1 shows how the simple `struct list_head` is used to maintain a list of data structures. List heads must be initialized prior to use with the `INIT_LIST_HEAD` macro. A list head can be declared and initialized with:

```
struct list_head todo_list;
INIT_LIST_HEAD(&todo_list);
```

Alternatively, lists can be initialized at compile time:

```
LIST_HEAD(todo_list);
```

The `list_head` structures are good for implementing a list of like structures, but the invoking program is usually more interested in the larger structures that make up the list as a whole. A macro, `list_entry`, is provided that maps a `list_head` structure pointer back into a pointer to the structure that contains it. It is invoked as follows:

```
list_entry(struct list_head *ptr, type_of_struct, field_name);
```

where `ptr` is a pointer to the `struct list_head` being used, `type_of_struct` is the type of the structure containing the `ptr`, and `field_name` is the name of the list field within the structure.

When first encountering this, most people get confused because they have been thaught to implement linked lists just by adding a pointer in a structure which points to the next similar structure in the linked list. The drawback of this approach, and the reason for which the kernel implements linked lists differently, is that you need to write code to handle the adding / removing / etc of elements specifically for that data structure. Here, we can add a `struct list_head` field to any other data structure and make it a part of a linked list.

In our `todo_struct` structure from before, we called the list field simply `list`. Thus, we would turn a list entry into its containing structure with a line such as:

```
struct todo_struct *todo_ptr = list_entry(listptr, struct todo_struct, list);
```

The `list_entry` macro takes a little getting used to, so let us see first how it is defined:

```
/**
 * list_entry - get the struct for this entry
 * @ptr: the &struct list_head pointer.
 * @type: the type of the struct this is embedded in.
 * @member: the name of the list_struct within the struct.
 */
#define list_entry(ptr, type, member) \
        container_of(ptr, type, member)
```

As we can see, the `list_entry` macro just calls another `container_of` macro, this one deserves a careful explanation. So, what does this `container_of` macro do? Well, precisely what its name suggests. It takes three arguments - a pointer, type of the container, and the name of the member the pointer refers to. The macro will then expand to a new address pointing to the container which accommodates the respective member.

For a better explanation, the Figure 4.2 below illustrates the principle of the `container_of` macro.

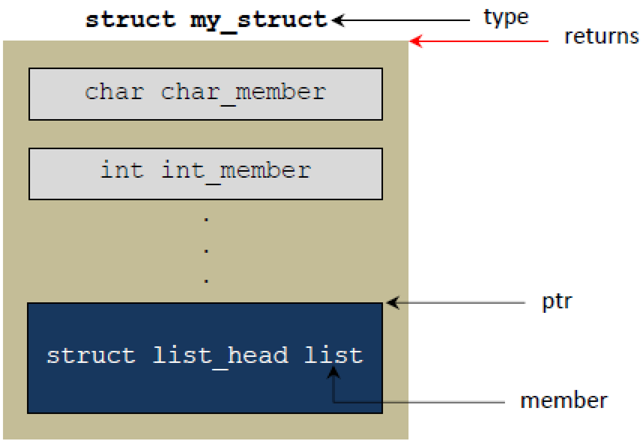


Figure 4.2: The `container_of(ptr, type, member)` macro

Below is the actual implementation in the Linux kernel:

```
#define container_of(ptr, type, member) ( \
        { \
           const typeof(((type *)0)->member) *__mptr = (ptr); \
           (type *)((char *)__mptr - offsetof(type, member)); \
        })
```

At a first glance, this might look like a whole lot of magic, but it is not quite so, obviously. Let us take it step by step. The first thing to focus our attention on is `typeof()`. This is a non-standard GNU C extension. It takes one argument and simply returns its data type back.

But what about the zero pointer dereference? Well, it is a little "pointer magic" to get the type of the member. It will not crash, because the expression itself will never be evaluated. All the compiler cares about is its type. The same occurs when we ask back for the address. The compiler again does not care about the value, it simply adds the offset of the member to the base address of the structure, in this particular case `0`, and returns the new address.

The second line we can see is also using a macro, the `offsetof` macro, which actually makes use of that same aforementioned pointer magic. This macro will return a byte offset of a member to the beginning of the structure. It is even part of the standard library, available in "stddef.h". Here is how it is defined inside the kernel:

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

It returns an address of a member called `MEMBER` of a structure of type `TYPE` that is stored in memory from address `0`, which happens to be the offset we're looking for.

Finally, when looking more closely at the original definition of `container_of`, we may start wondering if the first line is really good for anything. Well, that is a very legitimate question to ask as it is not intrinsically important for the end result of the macro, but is there for type-checking reasons. We want to make sure that the pointer passed in is really a pointer that has the type of `&(type.member)`. And what the second line really does? It subtracts the offset of the structure's member from its address yielding the address of the container structure. This is it!

For a full explanation of the linked lists API, please check "include/linux/list.h".

4.2.2 Hash Tables

Hash tables are implemented inside the Linux kernel with the help of the following two data structure types: `hlist_head` and `hlist_node`. The table's buckets are defined as type `hlist_head`, and the actual elements that are added to the table embed an element of type `hlist_node` that is used to link them to the table. The only difference between the types is

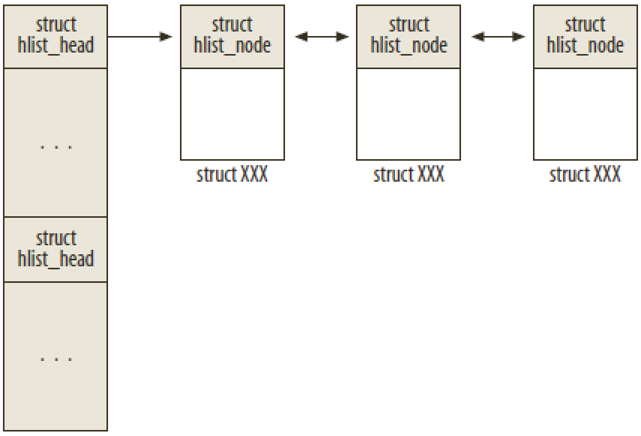


Figure 4.3: A kernel hash table

that `hlist_head` includes only a forward pointer, whereas `hlist_node` has both forward and backward pointers. Thus, the lists in hash table buckets are bidirectional. However, the list is not circular, as the head has only a forward pointer. The reason for having only one pointer per bucket is because, compared to a hash table with buckets that have two pointers (a forward and a backward one), reducing the size of the bucket by 50% results in doubling the number of buckets this "one pointer per bucket" hash table implementation can store with the same amount of memory with respect to the "two pointers per bucket" implementation.

Figure 4.3 above shows an example of a kernel-style hash table, where three items are hashed (collide) to the same bucket, therefore, forming a bidirectional list. However, a good hash function should make sure to get *O(1)* elements into every bucket.

Let us take a look at the kernel's hash table API with an example. First of all we have to define the structure of the items that will be hashed inside our hash table, it can simply be something like this:

```
struct mystruct
{
    int data;
};
```

To be able to link each element of type `struct mystruct` to others, we need to add a `struct hlist_node` member:

```
struct mystruct
{
    int data;
    struct hlist_node my_hlist;
};
```

The hash table is actually an array of `struct hlist_head` pointers, where each one points to a different list, and each one of those lists holds all elements that are hashed to the same bucket. So every element is essentially part of a linked list and the hash table holds only the head of these lists.

Back to the example, let us create the first variable representing an item that will be hashed:

```
struct mystruct item_1 = {
    .data = 10,
    .my_hlist = 0 // initialized when added to the hash table
};
```

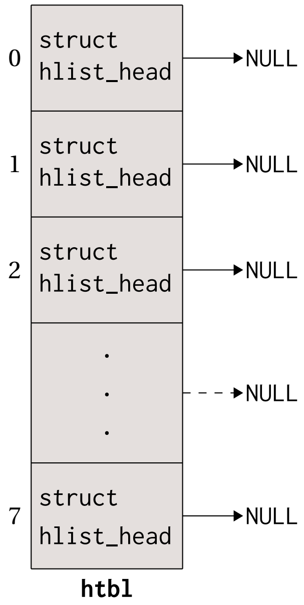
Now that we have an element that can be added to a hash table, let us define one as follows:

```
DEFINE_HASHTABLE(htbl, 3);
```

`DEFINE_HASHTABLE` is a macro defined in kernel as:

```
#define DEFINE_HASHTABLE(name, bits) \
        struct hlist_head name[1 << (bits)] = \
           { [0 ... ((1 << (bits)) - 1)] = HLIST_HEAD_INIT } \
```

The `HLIST_HEAD_INIT` macro initializes only a NULL pointing of the bucket pointer called `first`. Notice also that we use the number of bits to define a hash table and not the size, so the hash table we declared with 3 bits has an actual size of 8 buckets. At this point, our just initialized `htbl` looks as follows:



So, now we have a hash table called `htbl` with 8 buckets each consisting of a list (pointing to `NULL`), and an element that we want to add to the hash table. But first we need to decide about a key for the element. For simplicity, let us use the item's data as a hash key.

An example of a macro to be used in order to add an object to a specified hash table is `hash_add` defined in "include/linux/hashtable.h" as:

```
/**
 * hash_add - add an object to a hashtable
 * @hashtable: hashtable to add to
 * @node: the &struct hlist_node of the object to be added
 * @key: the key of the object to be added
 */
#define hash_add(hashtable, node, key) \
        hlist_add_head(node, &hashtable[hash_min(key, HASH_BITS(hashtable))])
```

As we can see, the hash_min macro plays the role of the hash function, meaning that it is the one who decides into which bucket the element will go. It is using *golden ratio* constants to calculate, deterministically, a random number from any key.

Let us add the item_1 element to the htbl hash table:

```
hash_add(htbl, &item_1.my_hlist, item_1.data);
```

This will add the element item_1 to the hash table. Let us add another two elements:

```
struct mystruct item_2 = {
    .data = 20,
    .my_hlist = 0
};

struct mystruct item_3 = {
    .data = 44,
    .my_hlist = 0
};

hash_add(htbl, &item_2.my_hlist, item_2.data);
hash_add(htbl, &item_3.my_hlist, item_3.data);
```

Now we have three items in the hash table. For iterating over all elements we can use the hash_for_each macro which translates to a for loop that loops the lists in all buckets.

```
/**
 * hash_for_each - iterate over a hashtable
 * @name: hashtable to iterate
 * @bkt: integer to use as bucket loop cursor
 * @obj: the type * to use as a loop cursor for each entry
 * @member: the name of the hlist_node within the struct
 */
#define hash_for_each(name, bkt, obj, member) \
        for ((bkt) = 0, obj = NULL; obj == NULL && (bkt) < HASH_SIZE(name); (bkt)++) \
             hlist_for_each_entry(obj, &name[bkt], member)
```

The most inner macro hlist_for_each_entry is implemented in the "include/linux/list.h" header file which simply expands to another for loop iterating over all members of the list. The outer for loop iterates over all buckets in the hash table, i.e. all indices of the array.

Now we can see, on a 64-bit machine, that item_1 and item_3 are hashed to bucket number 1, and item_2 is hashed to bucket number 2.

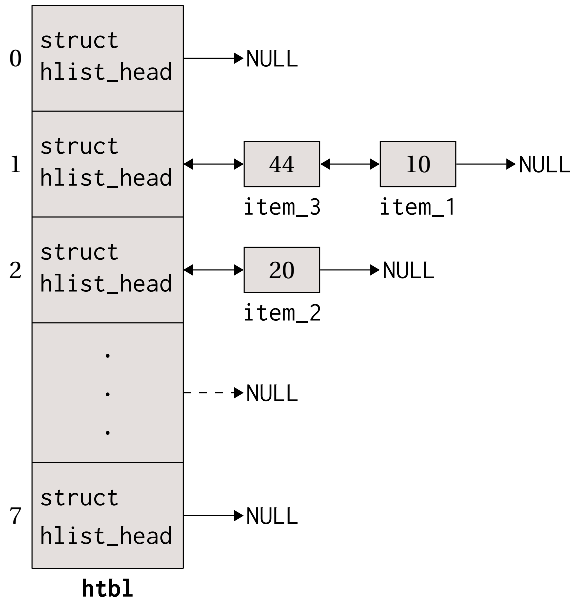


Figure 4.4: The `htbl` hash table populated with three elements

Figure 4.4 above is a visual representation of what we have in memory.

4.2.3 The Socket Buffer: `sk_buff` structure

`struct sk_buff` is where a packet is stored. It is the most fundamental data structure in the Linux networking code. Every packet sent or received is handled using this data structure. The structure is used by all the network layers to store their headers, information about the user data (the payload), and other information needed internally for coordinating their work.

Defined in the "include/linux/skbuff.h" header file, it consists of a tremendous amount of data. Its fields can be classified roughly into the following categories:

- Layout
- General
- Feature-specific
- Management functions

`sk_buff` structure consists of three segments:

- `sk_buff` structure itself
- Linear block of data

- Non-linear data portion represented by `struct skb_shared_info`

This structure is used by several different network layers (MAC or another link protocol on the L2 layer, IP on L3, TCP or UDP on L4), and various fields of the structure change as it is passed from one layer to another. L4 appends a header before passing it to L3, which in turn puts on its own header before passing it to L2. One of the first things done by each protocol, as the buffer passes down through layers, is to call `skb_reserve()` to reserve space for the protocol's header.

When the buffer passes up through the network layers, each header from the old layer is no longer of interest. The L2 header, for instance, is used only by the device drivers that handle the L2 protocol, so it is of no interest to L3. Instead of removing the L2 header from the buffer, the pointer to the beginning of the payload is moved ahead to the beginning of the L3 header, requiring fewer CPU cycles.

From this point on, a brief description follows for each of the core members of the `sk_buff` structure. The structure definition starts as:

```
struct sk_buff
{
    /* These two members must be first. */
    struct sk_buff *next;
    struct sk_buff *prev;
    ...
    struct sock *sk;
    struct net_device *dev;
    ...
}
```

The first two members are part of the layout fields, specifically they implement list handling. Packets can exist on several kinds of lists and queues. For example, a TCP socket send queue. The sk pointer is where we record the socket associated with this packet buffer. When a packet is sent or received for a socket, the memory associated with the packet must be charged to the socket for proper memory accounting.

The `dev` pointer is part of the general fields, describing a network device. Specifically, it identifies the device either where to send the packet or from which one it was received on. Other interesting fields are:

`unsigned int len`

This is the size of the block of data, including protocol headers as well, in the buffer. This length includes both the data in the main buffer (i.e. the one pointed to by head) and the data in the fragments. Its value changes as the buffer moves up and down the network stack layers, because headers are discarded while moving up in the stack and are added while moving down the stack.

`unsigned int data_len`

Unlike `len`, `data_len` accounts only for the size of the data in the fragments.

`__u16 mac_len`

This stands for the size of the MAC header.

`unsigned int truesize`

This field represents the total size of the buffer, including the `sk_buff` structure itself. It is initialized by the `__alloc_skb()` function to `len + sizeof(sk_buff) + sizeof(struct skb_shared_info)` when the buffer is allocated for a requested data space of `len` bytes.

```
#define SKB_TRUESIZE(X) ((X) + \
                        SKB_DATA_ALIGN(sizeof(struct sk_buff)) + \
                        SKB_DATA_ALIGN(sizeof(struct skb_shared_info)))

struct sk_buff *__alloc_skb(unsigned int size, gfp_t gfp_mask, int flags, int node)
{
   ... ... ...
   skb->truesize = SKB_TRUESIZE(size);
   ... ... ...
}
```

From the code snapshot we can see that it is the `SKB_TRUESIZE` macro that is actually calculating `truesize`, taking into account the appropriate memory alignment.
The field gets updated whenever `skb->len` is increased.

`unsigned char *head`

`unsigned char *data`

`unsigned char *tail`

`unsigned char *end`

These four pointers provide the core management of the linear packet data area of an `sk_buff`. `head` and `end` point to the beginning and end of the space allocated to the buffer, and `data` and `tail` point to the beginning and end of the actual data. For a visual illustration see Figure 4.5.

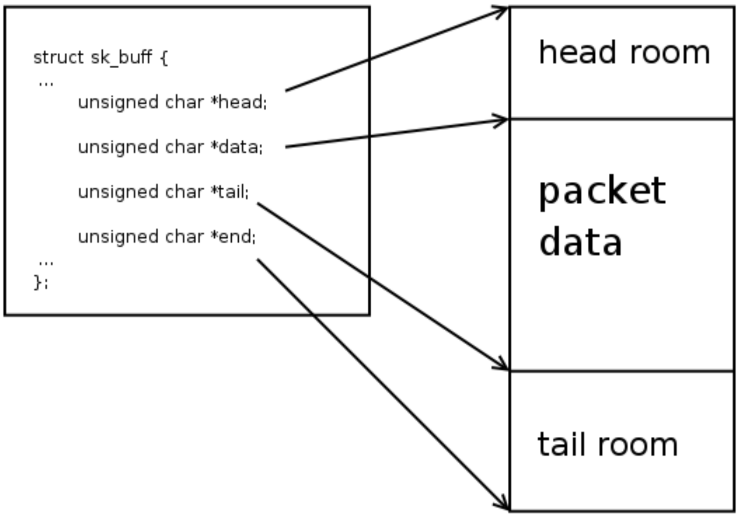
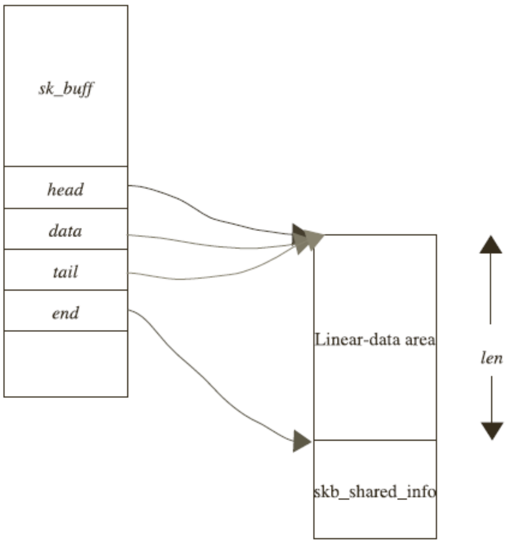


Figure 4.5: The layout of the sk_buff data area and where in that area the various pointers in struct sk_buff point, source [8]

Whenever a new sk_buff is allocated, the size of the linear-data area needs to be provided. At the same time, these four fields of sk_buff are initialized to point to linear-data area in appropriate positions. The figure below shows the position of the four fields when a new sk_buff is allocated as just returned by alloc_skb(). Notice the memory space reserved for struct skb_shared_info at the end of the linear data area. This structure is shared across the sk_buff clones.



union {...} h

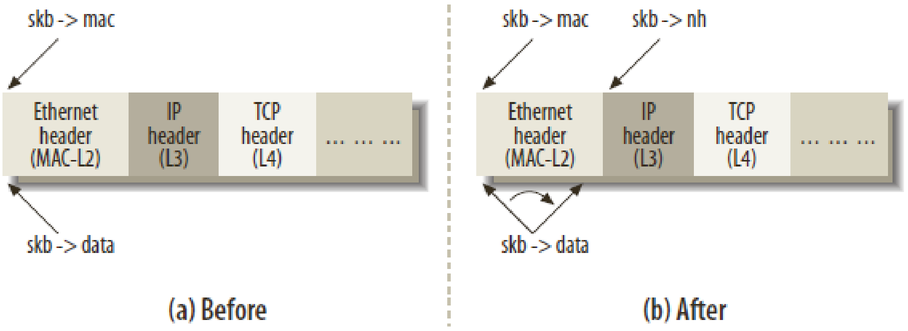
union {...} nh

union {...} mac

Here we store the location of the various protocol layer headers as we build outgoing packets, and parse incoming ones. One member of each union is called `raw` and is used for initialization; all later accesses are through the protocol-specific members. For example, `skb->mac.raw` is set by `eth_type_trans()` when an ethernet packet is received. Later, we can use this to find the location of the MAC header.

When receiving a data packet, the function responsible for the layer n header processing receives a buffer from layer $n-1$ with `skb->data` pointing to the beginning of the layer n header. The function that handles layer n initializes the proper pointer for this layer (for instance, `skb->nh` for L3 handlers) to preserve the `skb->data` field, because the contents of this pointer will be lost during the processing at the next layer, when `skb->data` is initialized to a different offset within the buffer. The function then completes the layer n processing and, before passing the packet to the layer $n+1$ handler, updates `skb->data` to make it point to the end of the layer n header, which is the beginning of the layer $n+1$ header (see the figure below).

Sending a packet reverses this process, with the further complexity of adding a new header at each layer.



Header's pointer initializations before and after L2 processing, source [6]

`char cb[48]`

This is a "control buffer", or storage for private information, maintained by each layer for internal use. It is statically allocated within the `sk_buff` structure (currently with a size of 48 bytes) and is large enough to hold whatever private data is needed by each layer. In the code for each layer, access is done through macros to make the code more readable. TCP, for example, uses that space to store a `tcp_skb_cb` data structure containing TCP sequence numbers and header flags.

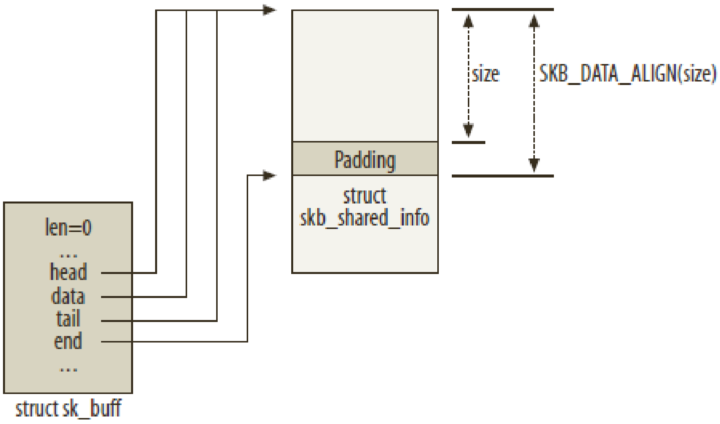
`__be16 protocol`

This field specifies the protocol used at the next-higher layer from the perspective of the device driver at L2. Typical protocols listed here are IP, IPv6, and ARP; the "include/uapi/linux/if_ether.h" header file defines all the next-protocols to use as *ethertype* (i.e. values for the 2 byte Protocol ID in Ethernet header). Since each protocol has its own function handler for the processing of incoming packets, this field is used by the driver to inform the layer above it what handler to use. Each driver calls `netif_rx()` to invoke the handler for the upper network layer, so the protocol field must be initialized before that function is invoked.

Management Functions

`alloc_skb()`

This function allocates a new `sk_buff`. We pass on the length of the data area and the mode of memory allocation. Data area is the block of memory allocated for the `sk_buff` where the packet is constructed. End of the linear data area is reserved for structure that keeps information of the paged-data area and fragments associated with the `sk_buff`. So, we allocate a `sk_buff` head and the data area of length "size" bytes. The position of head, `data`, `tail`, and end pointers are shown in the figure below when the `alloc_skb()` returns.



`alloc_skb()` function, source [6]

We can see that the *tail room* [1] is equal to the length of the data block requested for `sk_buff` just after allocation. *Head room* [2] and data length are zero.

[1] Tail room is the space between the `tail` and end pointers.

[2] Headroom is the space between the head and `data` pointers

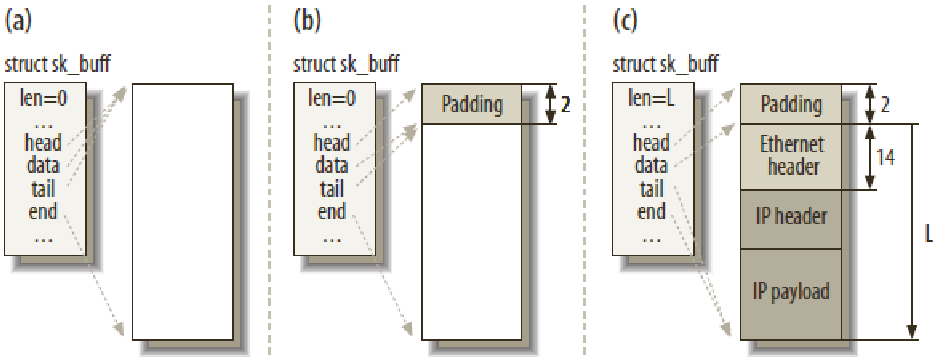
skb_reserve()

This function changes head and tail room for the sk_buff. It is called mostly to reserve space for the protocol headers. We pass length of the headroom we need to reserve for the protocol headers.

Usually the receive function of all the Ethernet drivers use the following command before storing any data in the buffer they have just allocated:

```
skb_reserve(skb, 2); /* Align IP on 16 byte boundaries */
```

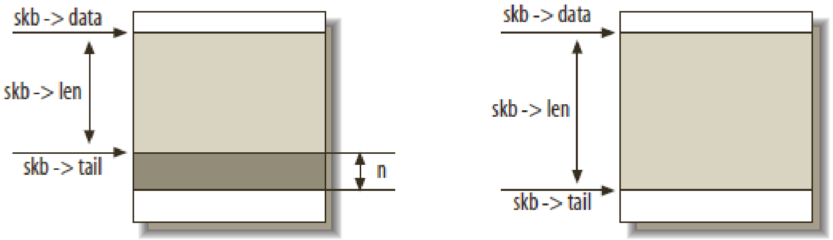
Because they know that they are about to copy an Ethernet frame that has a header 14 bytes long into the buffer, the argument of 2 shifts the head pointer by 2 bytes more. This keeps the IP header, which follows immediately after the Ethernet header, aligned on a 16-byte boundary from the beginning of the buffer, as shown in figure below.



skb_reserve(): (a) before, (b) after, and (c) after copying the frame on the buffer. Source [6]

skb_put()

Routine used to manipulate sk_buff's linear data area. The function reserves space for the segment data at the end of the linear data area, i.e. skb->tail. It advances the tail pointer by the specified number of bytes, it also increments the len field by that number of bytes as well.



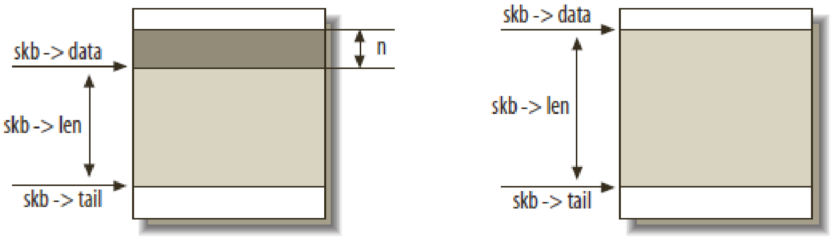
skb_put(): before and after calling the function (from left to right). Source [6]

It does not add any data to the buffer, it simply moves the `tail` pointer and updates `len` accordingly.

`skb_push()`

This function manipulates the `data` pointer and acts only on linear data area. It pushes the `data` pointer closer to the `head` pointer by the number of bytes provided as an argument to the function. The head room is reduced by the number of bytes that data length has increased. This shift of `data` pointer toward `head` causes overall `sk_buff` length to expand by the requested length.

This routine is mainly called when sending a packet. The packet contains data and protocol headers. We need to add data, and each protocol layer will add its header as it passes through different layers. So, the topmost layer adds data and then its header. We have seen functions that will create headroom and the room for the user data. We create headroom by calling `skb_reserve()` and then room for user data by calling `skb_put()`. We then copy user data in the data area pointed to by `skb->data`. Now it is the chance to add the protocol header just before the start of user data.



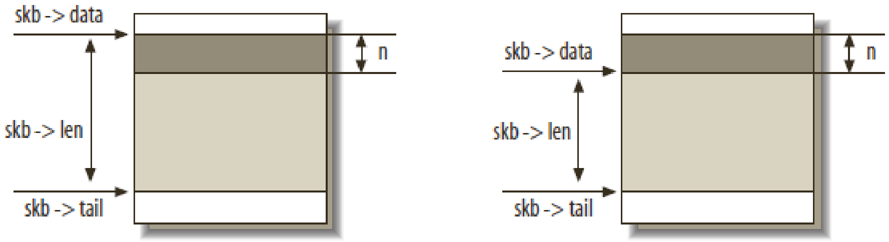
`skb_push()`: before and after calling the function (from left to right). Source [6]

Like `skb_reserve()` and `skb_put()`, this function does not really add any data to the buffer, it simply moves the `data` pointer updating `len` accordingly.

`skb_pull()`

The routine pulls down the `data` pointer by the number of bytes specified as an argument to the function and returns the new `data` pointer. This manipulates `sk_buff`'s linear data area by modifying its `data` field. It reduces `skb->len` by the number of bytes requested hence increasing headroom for `sk_buff`'s linear data area.

The function is mostly used to access protocol headers when a packet arrives.



`skb_pull()`: before and after calling the function (from left to right). Source [6]

`skb_clone()`

When the same buffer needs to be processed independently by different consumers, and they may need to change the content of the `sk_buff` descriptor, the kernel does not need to make a complete copy of both the `sk_buff` structure and the associated data buffers. Instead, to be more efficient, the kernel can clone the original, which consists of making a copy of the `sk_buff` structure only and playing with the reference counts to avoid releasing the shared data block prematurely. Buffer cloning is done with the `skb_clone()` function.

The `sk_buff` clone is not linked to any list and has no reference to the socket owner. The field `skb->cloned` is set to 1 in both the clone and the original buffer. `skb->users` is set to 1 in the clone so that the first attempt to remove it succeeds, and the number of references (`dataref`) to the buffer containing the data is incremented (since now there is one more `sk_buff` data structure pointing to it). Figure 4.6 shows an example of a cloned buffer. The `skb_clone()` function can be used to check the cloned status of an `sk_buff` buffer.

There is also the `skb_share_check()` function that can be used to check the reference count `skb->users` and clone the `skb` buffer when the `users` field says the buffer is shared.

4.3 Open vSwitch

4.3.1 Microflow Caching

Initially, only the packet forwarding could achieve good performance, so the initial implementation put all the OpenFlow processing in a kernel module. The module received a packet, classified through the OpenFlow table, modified it as necessary, and finally sent it

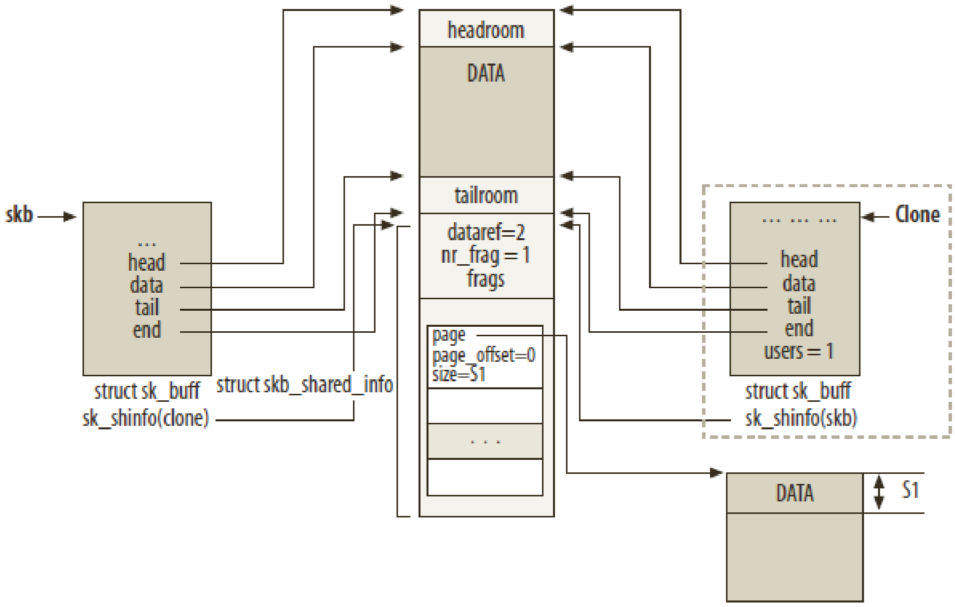


Figure 4.6: `skb_clone()` function, source [6]

to another port. However, this approach soon became impractical because of the difficulty of developing in the kernel and distributing and updating kernel modules.

The solution was to reimplement the kernel module as a *microflow cache* in which a single cache entry matches exactly with all the packet header fields supported by OpenFlow. In this design, cache entries are extremely fine-grained and match at most packets of a single transport connection. Thus, even for a single transport connection, a change in network path and hence in IP TTL field would result in a miss, and would sent the packet to user space which consulted the actual OpenFlow flow table to decide what to do with it.

4.3.2 Megaflow Caching

The only drawback of the microflow cache is that of the exact match requirement, which means that in order to install a flow entry you had to specify values for all the supported header fields. Therefore, soon the microflow cache got replaced with a *megaflow cache*. The megaflow cache is a single flow lookup table that supports generic matching, i.e., it supports caching forwarding decisions for larger aggregates of traffic than connections. While it more closely resembles a generic OpenFlow table than the microflow cache does, due to its support for arbitrary packet field matching, it is still strictly simpler and lighter in runtime for two main reasons. First, it does not have priorities, which speeds up packet classification: the in-kernel tuple space search implementation can terminate as soon as it finds any match, instead of continuing to look for a higher priority match until all the mask-specific hash tables are inspected. Second, there is only one megaflow classifier, instead of a pipeline of them, so user space installs megaflow entries that collapse together the behavior of all relevant OpenFlow tables.

The cost of a megaflow lookup is close to the general purpose packet classifier, even though it lacks support for flow priorities. Searching the megaflow classifier requires searching each of its hash tables until a match is found, as seen in the example of "Defining Tuple Space" section. Assuming that each hash table is equally likely to contain a match, matching packets require searching $(n+1)/2$ tables on average, and non-matching packets require searching all n of them. Therefore, for $n > 1$, which is usually the case, a classifier-based megaflow search requires more hash table lookups than a microflow cache.

Open vSwitch addresses the costs of megaflows by retaining the microflow cache as a first-level cache, consulted before the megaflow cache. This microflow cache is a hash table that maps from a microflow to its matching megaflow. Thus, after the first packet in a microflow passes through the kernel megaflow table, requiring a search of the kernel classifier, this exact-match cache allows subsequent packets in the same microflow to get quickly directed to the appropriate megaflow. This reduces the cost of megaflows from per-packet to per-microflow.

4.3.3 Kernel Datapath

Overview

From an architectural point of view, there are three main components started by the openvswitch 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 4.7 is a simple illustration of their relationship. The most important component is the user space daemon, `ovs-vswitchd`, that dictates how the kernel module will be programmed – think of it as a locally significant control plane. This daemon runs on every physical host. It

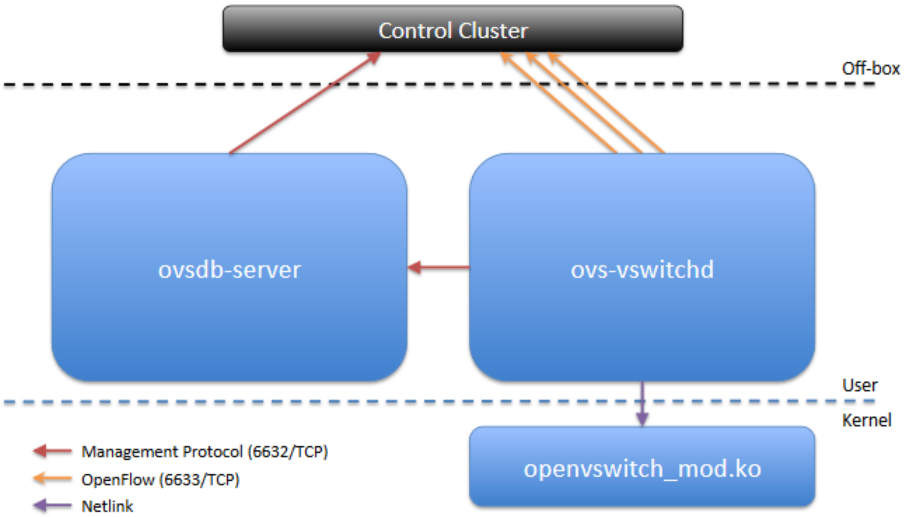


Figure 4.7: Main Open vSwitch components, source [21]

is important to understand that this is not an SDN or OpenFlow controller by any means, it is just a controller in the sense that it controls a local data plane module, or rather an *extension* of the data plane.

`ovsdb-server` is the Open vSwitch Database server. `ovs-vswitchd` saves and changes the switch configuration into this database; it talks to `ovsdb-server` via Unix domain socket in order to retrieve or save the configuration information. Because of this, the openvswitch configuration does survive after a reboot.

Let us now take a look at the kernel part. For a simple understanding, I would equate the kernel module to ASICs (Application Specific Integrated Circuit) on a hardware equipment switch. It is where all of the packet processing takes place, i.e., the data plane.

The user space module communicates with the kernel space module via Netlink protocol, the generic netlink family [27] is used in this case. So there are several groups of generic netlink (genl) commands defined by the OVS kernel module, they are used to get / set / add / delete some datapath / flow / vport and execute some actions on matching packets. The ones 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
};
```

There are also kernel functions doing the corresponding job:

```
ovs_dp_cmd_new()

ovs_dp_cmd_del()

ovs_dp_cmd_get()

ovs_dp_cmd_set()
```

Similar functions are defined for vport and flow commands too.

Before describing the involved data structures, let us see 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" for how to forward 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 database 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. Finally, an `OVS_PACKET_CMD_EXECUTE` command tells the kernel to execute the action we just set. That is, the kernel will execute this genl command in the `do_execute_actions()` function and will finally forward 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. In 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.

Initialization

The GTP support has been implemented using Open vSwitch v2.5.0. Figure 4.8 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 4.8 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.

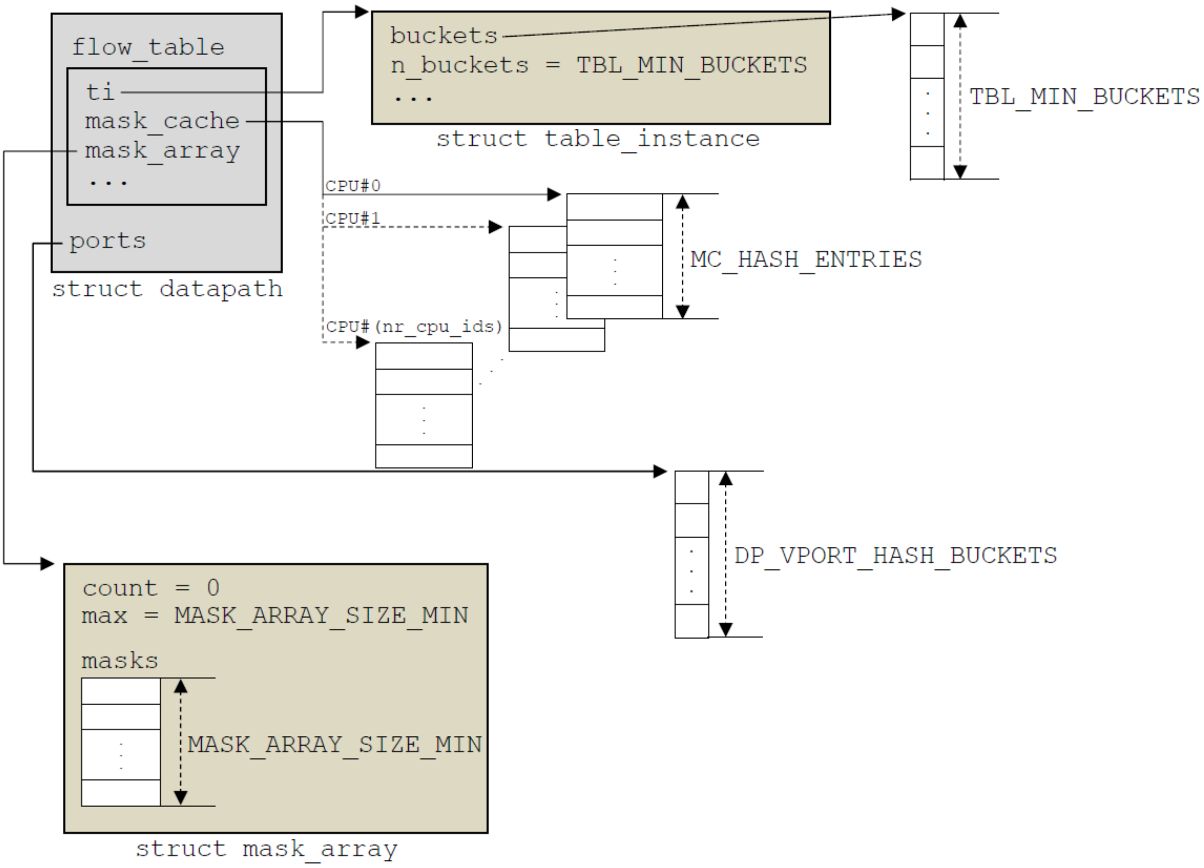


Figure 4.8: The `datapath` data structure and its main relationships

- `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 OVS kernel module implementation can be found in the "datapath" directory of the OVS root tree.

<u>datapath</u>

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:

```
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;
};
```

Notice 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 to 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 can be seen in Figure 4.9.

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

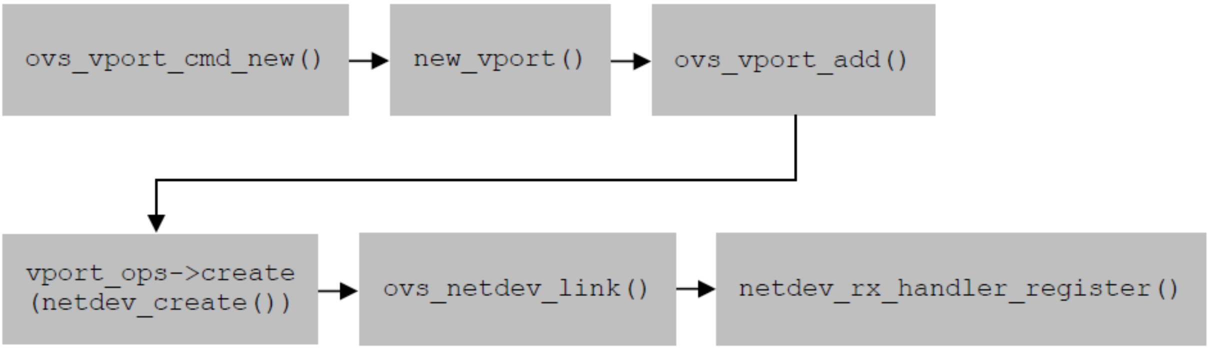


Figure 4.9: The processing sequence for `rx_handler` registration

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 rcu_head rcu;
    struct
    {
            struct hlist_node node[2];
            u32 hash;
    } flow_table, ufid_table;
    int stats_last_writer;  /* NUMA-node id of the last writer on 'stats[0]' */
    struct sw_flow_key key;
    struct sw_flow_id id;
    struct sw_flow_mask *mask;
    struct sw_flow_actions __rcu *sf_acts;

    /* One for each NUMA node. First one is allocated at flow creation time,
     * the rest are allocated on demand while holding the 'stats[0].lock'.
     */
    struct flow_stats __rcu *stats[];
};
```

`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 table_instance __rcu *ufid_ti;
    struct mask_cache_entry __percpu *mask_cache;
    struct mask_array __rcu *mask_array;
    unsigned long last_rehash;
    unsigned int count; // nr of flow entries in the flow table
    unsigned int ufid_count;
};
```

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 [18].

The kernel space implementation of a hash table and linked list data structures is somewhat different from the user space implementation style, where the latter is known and used by the most programmers as there are much more developers working in user space than in kernel space. Check the "Some important Linux kernel data structures" section for a detailed explanation of these data structures kernel space implementation.

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 4.10.

`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`. It makes then a copy of the packet represented by `skb`, otherwise OVS would mangle the packet for anyone who came before it.

`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*. This is an

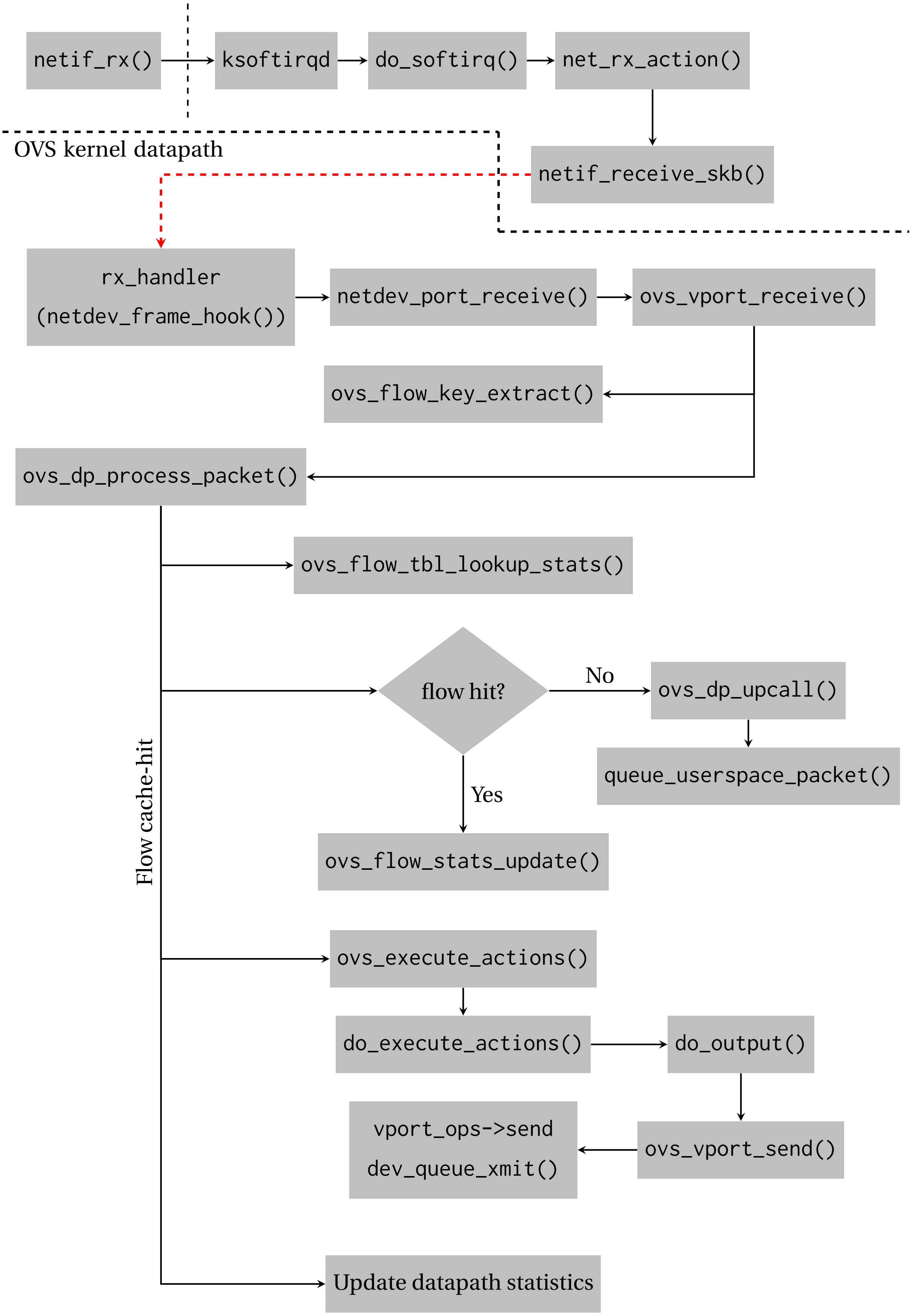


Figure 4.10: OVS kernel module main processing sequence

opaque storage area usable by protocols, and even some drivers, to store private per-packet information. TCP uses it, for example, to store sequence numbers and retransmission state for the frame. However, in this case it 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);

    return 0;
}
```

`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 is 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 are an unique way to identify 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;
            }

            ...
        }

        ...
    }
```

4.3.4 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. Notice 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() function

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()` function

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 [3], 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. Remember that the headroom and tailroom of a socket buffer are completely ignored, they are not part of the payload, as stated in "The Socket Buffer: `sk_buff` structure" section.

[3] 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. [12]

4.3.5 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 4.10), 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 [6] is a good source to learn about "Neighboring Subsystem" and "Routing Lookup" in 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. For each obtained result, a detailed description is given about how the corresponding test was performed. 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).

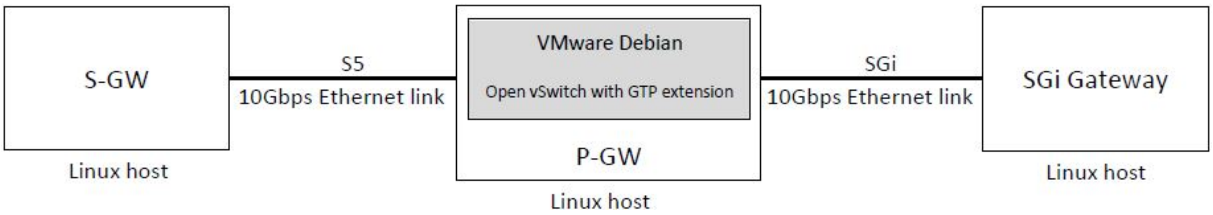


Figure 5.1: The test environment setup

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 using four criteria to evaluate the network's performance, these are 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 received data rate is considered as the UDP or TCP thoughput. The latency is measured with a ping test. Jitter is measured for UDP packets.

Iperf version 2.5 is the tool used as TCP and UDP network traffic generator (details of this software 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 Open vSwitch version 2.5.0 was used with the GTP-U tunnel extension (as implemented in this project).

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 [26], 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.

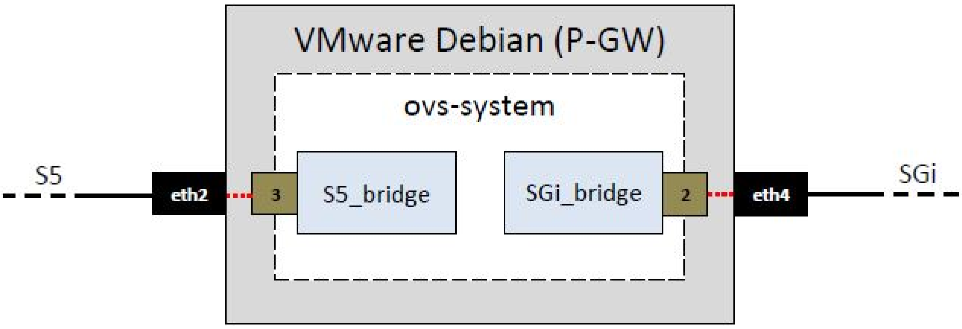


Figure 5.2: Our testbed's Open vSwitch configuration

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.

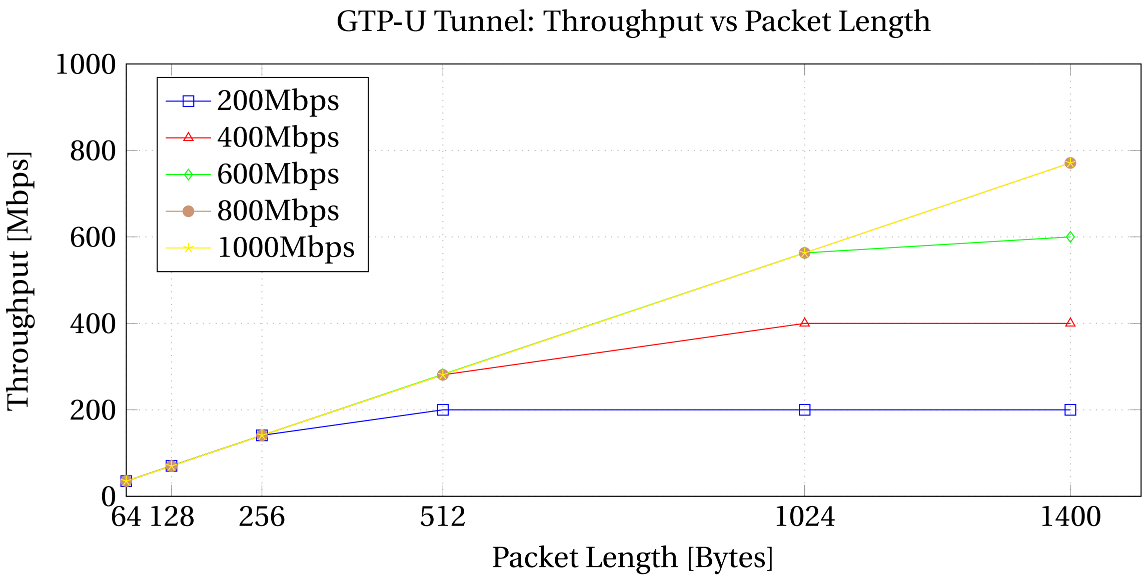


Figure 5.3: Throughput vs Packet Length (UDP) - Various offered loads, from 200Mbps to 1Gbps

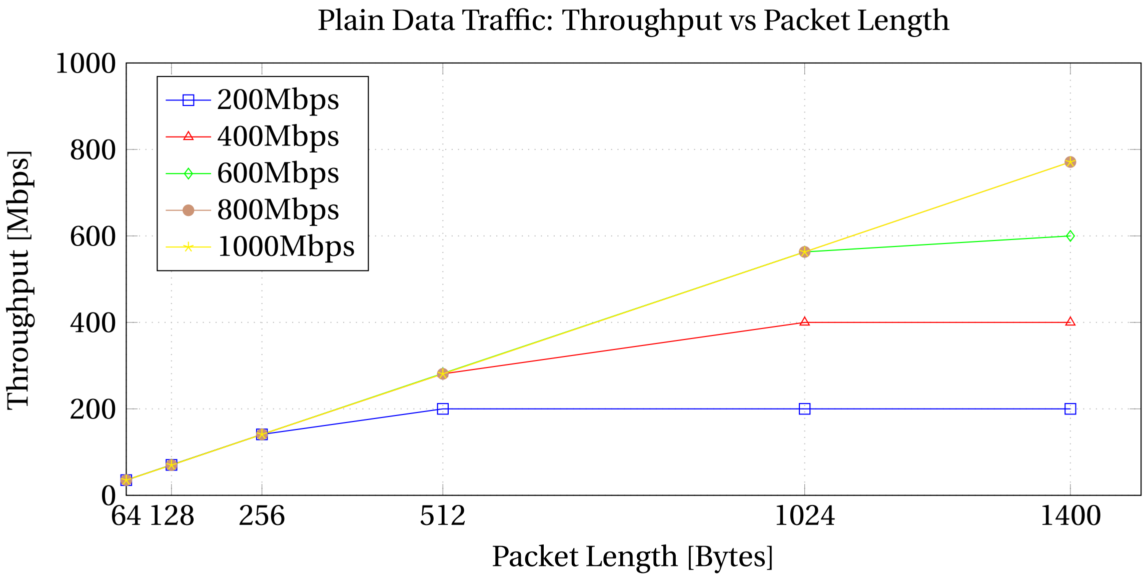


Figure 5.4: Throughput vs Packet Length (UDP) - Various offered loads, from 200Mbps to 1Gbps

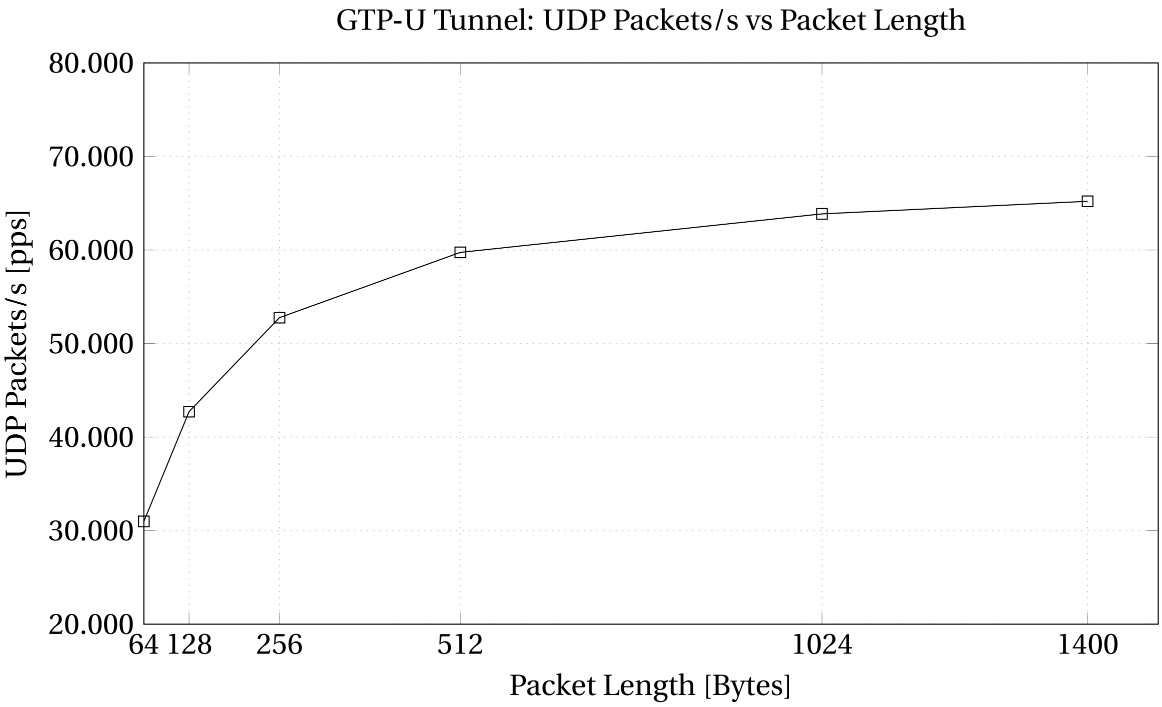


Figure 5.5: Packets/s vs Packet Length (UDP, Offered Load 1Gbps)

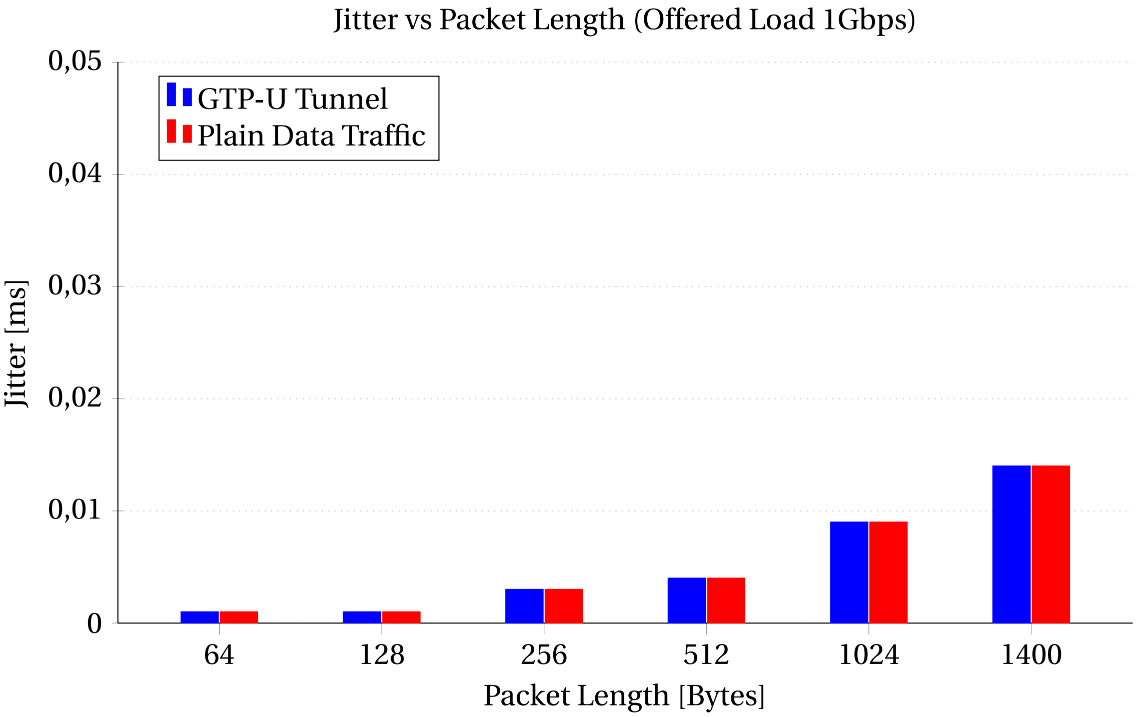


Figure 5.6: Jitter vs Packet Length (UDP, Offered Load 1Gbps)

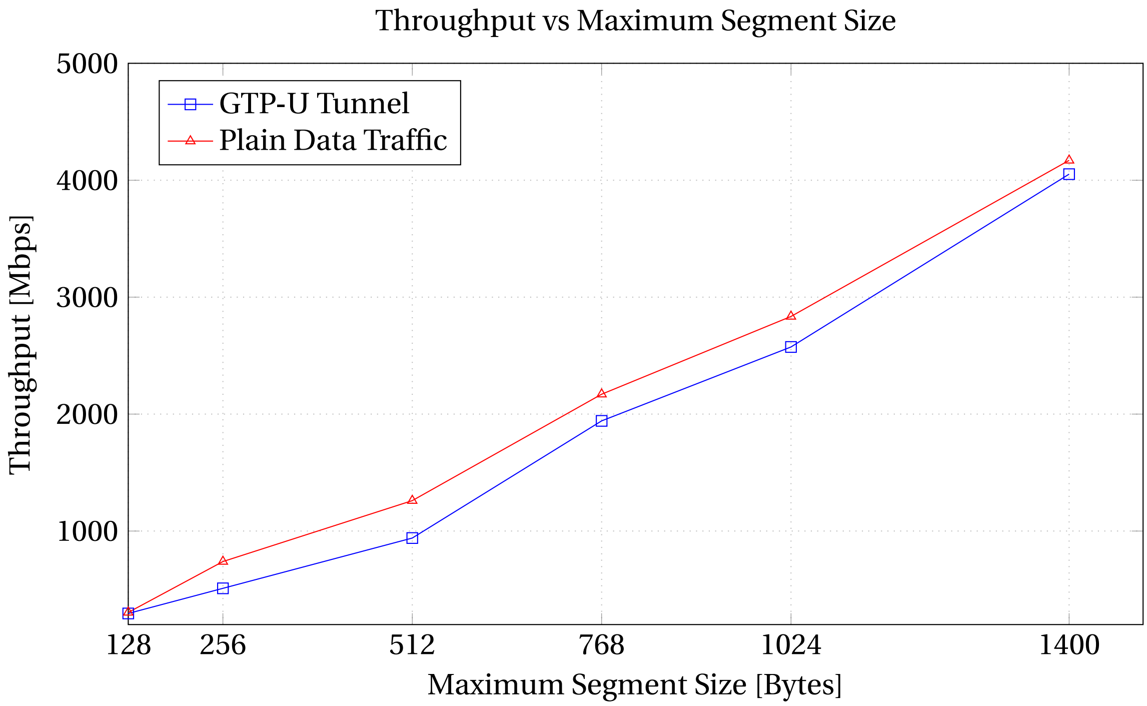


Figure 5.7: Throughput vs Maximum Segment Size (TCP)

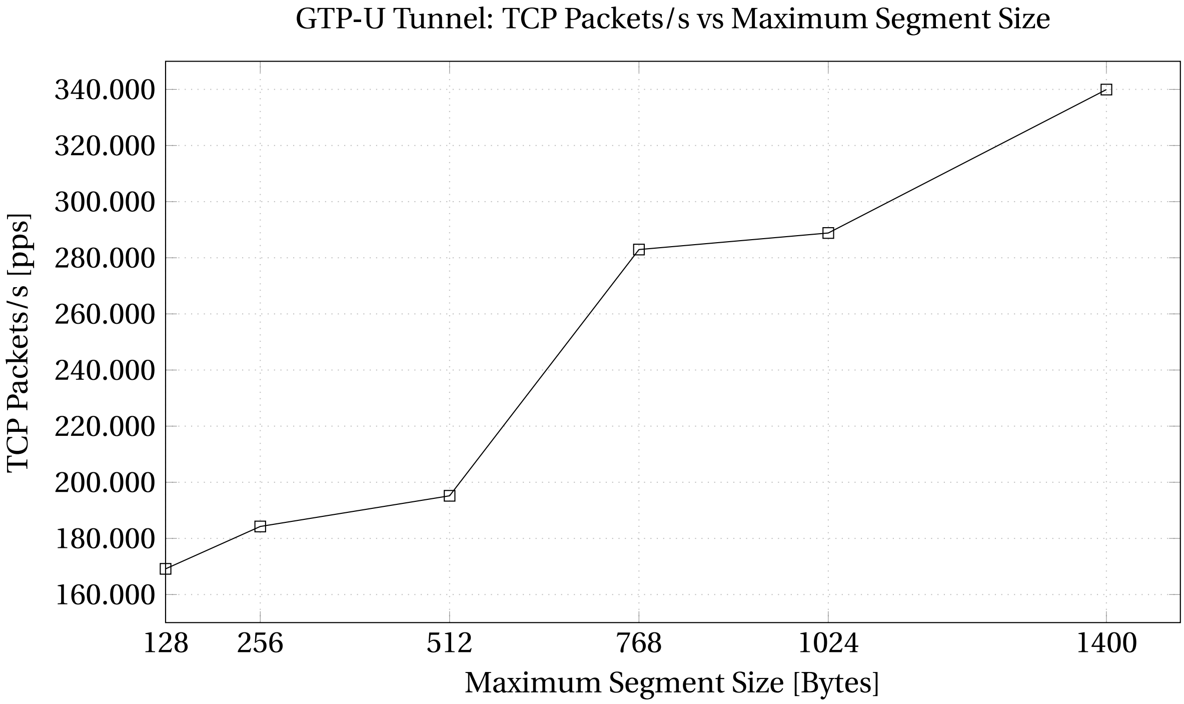


Figure 5.8: Packets/s vs Maximum Segment Size (TCP)

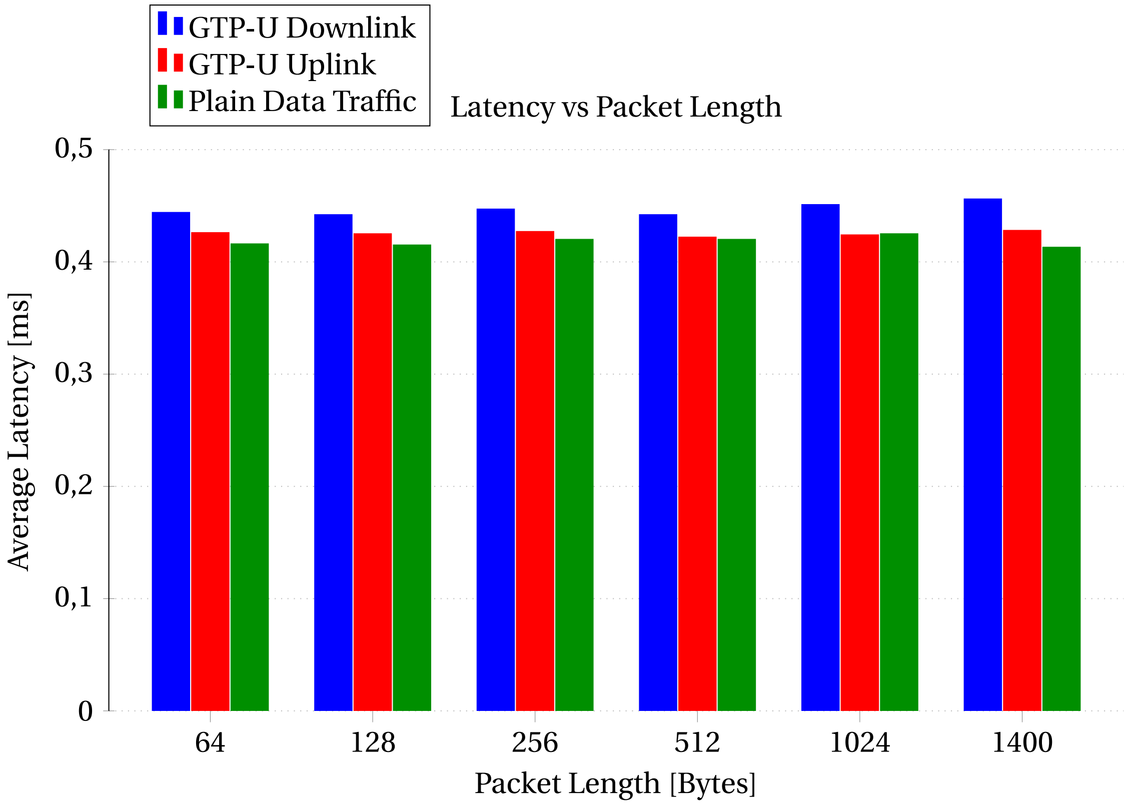


Figure 5.9: Latency vs Packet Length (ICMP)

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 little 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 by 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 in case of 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.

Overall, the GTP-U tunnel implementation was successful as it shows a quite good performance, i.e., it is comparable to the performance obtained using the plain data traffic (without involving Open vSwitch).

6 Conclusions and Future Work

In this chapter we will state our conclusions and insights gained as result of this thesis project. The chapter ends with some future work suggestions.

6.1 Conclusions

In this thesis we have described a potential evolutionary path for the user plane of a modern mobile core network. Based upon some primary design principles for SDN and mobile networks, the project designed a user plane architecture for a mobile core network and presented a set of core entities that could be utilized to realize the proposed network architecture. The user plane evolution can be viewed as a revolutionary approach in the packet core network for future mobile networks. The experimental implementation of the user plane phase of this evolution was also demonstrated and evaluated. By exploring the implementation possibilities and the design alternatives, we gained knowledge of SDN and the flexibility and power of the SDN-based software switch, i.e., Open vSwitch. As a part of the implementation process, the key enabler for the evolution of a mobile core network is represented by the GTP tunneling protocol. The user plane of the GTP protocol was implemented in the Open vSwitch software switch. As shown in Chapter 5, the performance of our GTP-U tunnel implementation was very promising, as its 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.

It is expected that the research for SDN and GTP tunneling mechanisms will be considered as building blocks for the overall design of future mobile core network evolution. A lot of work remains to be done from an engineering perspective to realize the proposed architecture and user plane entities. However, as already stated, SDN and NFV are significant in defining the direction of the evolution of mobile networks.

6.2 Future Work

In this section we suggest some future work with regard to our GTP implementation. We describe which of the parts of the existing implementation have to be improved in order to make the proposed solution work in a practical environment.

6.2.1 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.

6.2.2 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.

6.2.3 OpenFlow GTP Header Match

The optimal solution to match the GTP TEID field is to do it directly, however, the OpenFlow protocol does not yet support this matching. We should extend OpenFlow to support this matching, which further takes us to implement a proper API support in an OpenFlow controller.

6.2.4 IPv6 Support

For experimental purposes, the implementation described in this project was based on IPv4. However, IPv6 is inevitable, especially in mobile networks. Naturally, a future implementation should include support for IPv6.

6.2.5 Latency Evaluation

In the evaluation chapter, the latency test is simply performed using ping. However, it may not be accurate enough. A better solution would be using UTP synchronize clock with wireshark to do latency measurements.

6.2.6 Consider of Heterogeneous Wireless Access Networks

The design presented in this thesis was on a modern simulated LTE network core. However, LTE is not the only option for a broadband wireless access network. It is expected that future access networks will consist of a variety of heterogeneous network environments. These access networks may have different protocol stacks. How to best communicate via these heterogeneous networks is an issue that needs to be considered in future research.

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] 3GPP TS 29.272 (2013). Evolved Packet System (EPS); Mobility Management Entity (MME) and Serving GPRS Support Node (SGSN) Related Interfaces Based on Diameter Protocol, Release 11, Sections 7.3.2, 7.3.34, 7.3.35.

[3] A. Rubini and J. Corbet (2001). *Linux Device Drivers, 2nd Edition.* O'Reilly.

[4] 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).*

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

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

[7] 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].

[8] D. S. Miller. How SKB's work. http://vger.kernel.org/~davem/skb_data.html. [Online; accessed 22 December 2016].

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

[10] 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].

[11] 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].

[12] 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].

[13] 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].

[14] 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.

[15] 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].

[16] 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.

[17] IETF RFC 2663 (1999). IP Network Address Translator (NAT) Terminology and Considerations. https://tools.ietf.org/html/rfc2663. [Online; accessed 6 December 2016].

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

[19] 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.

[20] 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.

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

[22] 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].

[23] 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].

[24] 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].

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

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

[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] tutorialspoint.com. The Evolved Packet Core (EPC) (The core network). https://www.tutorialspoint.com/lte/lte_network_architecture.htm. [Online; accessed 25 November 2016].

[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].