Universidade de Aveiro
2015

Departamento de Eletrónica,
Telecomunicações e Informática

Flávio Silva Meneses

Desafios da Internet do futuro

Future network challenges

Universidade de Aveiro
Departamento de Eletrónica, Telecomunicações e Informática
2015

Flávio Silva Meneses

Desafios da Internet do futuro

Future network challenges

"*The greatest challenge to any thinker is stating the problem in a way that will allow a solution*"

— Bertrand Russell

Universidade de Aveiro
2015

Departamento de Eletrónica,
Telecomunicações e Informática

Flávio Silva Meneses

Desafios da Internet do futuro

Future network challenges

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Eletrónica e Telecomunicações, realizada sob a orientação científica do Doutor Rui L. Aguiar, Professor catedrático do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Daniel Corujo, investigador auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Professor Doutor Anibal Manuel de Oliveira Duarte

professor catedrático do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

vogais / examiners committee

Doutor Daniel Nunes Corujo

investigador auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

Doutor Pedro Miguel Naia Neves

consultor da PT Inovação

agradecimentos /
acknowledgements

A dissertação apresentada marca o término de um ciclo que não seria possível sem a colaboração e ajuda de várias pessoas e entidades, sendo com elas que partilho os méritos que dela possa receber. Agradeço a todos aqueles que contribuíram de forma decisiva para a concretização desta dissertação.

À Universidade de Aveiro e Instituto de Telecomunicações de Aveiro manifesto o meu apreço pela possibilidade de realização do presente trabalho e por todos os meios colocados à disposição, assim como a excelência da formação prestada e conhecimentos transmitidos, ambicionando que o trabalho desenvolvido dignifique ambas as instituições.

Aos orientadores desta dissertação, Professor Rui Aguiar e Doutor Daniel Corujo, agradeço não só toda a disponibilidade, colaboração e conhecimentos transmitidos, mas principalmente a capacidade de estímulo e confiança ao longo de todo o trabalho. Deixo também reconhecimento a Carlos Guimarães, agradecendo toda a partilha de conhecimentos.

Um especial e sentido abraço com profundo reconhecimento aos meus pais, Olga e Rui, pelo apoio e amor incondicional ao longo destes anos, pois sem eles esta meta não seria possível. Deixo também um agradecimento especial à minha namorada Sara Faria por me ter acompanhado sempre ao longo desta caminhada, com o seu ombro amigo e estabilidade emocional. À restante família, com especial atenção aos casais Fátima e António, e Rita e Marcelo, agradeço todo apoio e amizade partilhada.

Por último, a todas estas pessoas resta-me dizer: Muito obrigado...

Palavras Chave

OpenFlow; Redes sem-fios; Mobilidade; SDN; Handover.

Resumo

As capacidades de conetividade dos equipamentos móveis têm obrigado a constante mudança do modo de operação da rede, exigindo cada vez mais recursos. Visando as futuras redes 5G, existe a necessidade de evoluir as presentes redes móveis, melhorando as suas arquiteturas e mecanismos. Estas futuras redes, vistas como a próxima geração arquitetural das telecomunicações, tenta suportar a "explosão" do número de equipamentos ligados, serviços e tecnologias de acesso, apoiando-se fortemente nas redes definidas por *software* (do inglês, *Software Defined Networks, SDN*). Apesar de estas redes definidas por *software*, estarem a ser exploradas e implementadas no núcleo de rede, atualmente não contemplam o seu impacto em equipamentos sem-fios móveis, de forma a avaliar o possível suporte de controlo. Os desafios associados à extensão dos mechanismos e protocolos, como o OpenFlow, das redes definidas por *software* até aos equipamentos móveis, não só requerem o desenho de uma infra-estrutura capaz de suportar essa extensão, como também da sua avaliação e provenientes benefícios. Esta dissertação acompanha a tendência destas futuras redes, explorando a interação entre o equipamento móvel e a rede, em ambientes sem-fios heterogéneos, nos quais os mecanismos de SDN são extendidos até equipamentos móveis capazes de não só consumir, como também de produzir informação. Com isto, foi desenvolvida e implementada sobre uma rede sem-fios física uma arquitetura conceptual, na qual os mecanismos SDN são extendidos até ao terminal, suportando diferentes equipamentos móveis com múltiplos fluxos de dados. Os resultados obtidos, mostram a sua viabilidade em cenários de mobilidade sem congestionamento, visando benefícios em extender os mecanismos SDN para controlo de fluxos *end-to-end* em ambientes sem-fios.

Keywords OpenFlow; Wireless; Mobile Node; SDN; Mobility; Handover.

Abstract

The connectivity capabilities of mobile wireless devices have been forever changing how networks operate, increasingly demanding resources from the network. This places a need for novel mobile network architectures and mechanisms, targeting tomorrows challenges, as envisaged by 5G networks research efforts. This future network, seen as the next generation telecommunications architecture, aims to tackle the explosion of connected devices, services and access technologies, relying its architecture on Software Defined Networks (SDN) to compose its underlying mechanisms. Notwithstanding, despite the need for novel control procedures to support and optimize increasingly challenging wireless mobile scenarios, SDN has been being deployed at the core and backhaul sections of the network and is not actively considering its impact directly over the wireless mobile terminals themselves. The challenges associated with the extension of SDN protocols, such as OpenFlow, all the way to the terminal requires the design and evaluation of frameworks that not only provide such mechanisms, but actually evaluate them and their benefits.

This thesis shades a light on an important 5G trend, namely the interaction of the mobile node with the network, exploring a framework where SDN mechanisms are extended all the way to the mobile node, in heterogeneous wireless environments featuring different mobile nodes with multiple data flows, which act both as consumers and producers of information. In this way, flow-based mobility management becomes available to a network controller entity, via the OpenFlow protocol. The concept framework was implemented over a physical wireless testbed, validating its contribution in a mobile source-mobility use case, with results highlighting the promising benefits of extending SDN approaches for end-to-end flow control in wireless environments.

Contents

List of Figures

List of Tables

List of Acronyms

AP	Access Point
ESSID	Extended Service Set Identification
IEEE	Institute of Electrical and Electronics Engineers
IT	Instituto de Telecomunicações
IESG	Internet Engineering Steering Group
IETF	Internet Engineering Task Force
IRTF	Internet Research Task Force
IP	Internet Protocol
MAC	Media Access Control
MN	Mobile Node
OmniRAN	Open Mobile Network Interface for Omni-Range Area Networks
ONF	Open Networking Foundation
OvS	Open vSwitch
RTP	Real-time Protocol
SDN	Software Defined Networking
SDNRG	Software-Defined Networking Research Group
SDOs	Standards Development Organizations

CHAPTER 1

Introduction

In the last years we have been witnessing to a tremendous Internet growth supported by the simplicity of the network architecture, which allows a link between the different protocol solutions in the diverse layers of operation. This huge enlargement of the network, exponentially increasing not only the number of users, but also provided services, pushes the network to its limit and motivates pioneer architectural approaches, such as Software Defined Networking (SDN). These new architectural approaches promise to empower network infrastructures to better support upcoming challenges.

Towards this concern, upcoming 5G research has been guiding the development of the telecommunications architecture of tomorrow, addressing challenges such as massive traffic volumes, the proliferation of connected mobile devices and sustainable integration of heterogeneous networks in mobile environments. Software Defined Networking [1] has been one of the key building blocks of 5G network architectures, adding a greater degree of flexibility. Despite these contributions, SDN mechanisms were conceived with wired network infrastructures in mind, where new exploratory deployments have recently started to target its usage in cellular and wireless environments. However, these designs restrain SDN control to network edge nodes, disregarding the benefits towards mobile equipment control.

With the intend to contribute with new results in this particular area, this work addresses and extends SDN mechanisms, providing a new architecture where OpenFlow (the de-facto SDN protocol) is able to be used beyond the network edge and reach the mobile end-node. Unlike other solutions, the developed framework is detailed and supported by an experimental implementation and is conceived towards the support of different types of heterogeneous wireless technologies. To validate this concept, a prototype was implemented over a physical wireless testbed, allowing the study of the concept contribution and impact, in a mobile offloading source-mobility use case. Results highlight the promising benefits of extending SDN approaches for end-to-end flow control in wireless environments.

1.1 MOTIVATION AND OBJECTIVES

Although SDN mechanisms were developed keeping in mind the wired network infrastructures, the OpenFlow protocol was designed to allow networks to be (re)configured on the fly, and adding greater flexibility for accommodating new scenarios and applications. Extending SDN for end-to-end flow control in wireless environments allows the provision of an abstraction layer to the network nodes, and the capability of performing handover through OpenFlow mechanisms.

Keeping this in mind, the main objectives are: i) a whole new framework composed by a modified OpenFlow controller and several network nodes instantiating an Open vSwitch (OvS); ii) capability of performing a handover through OpenFlow mechanisms; iii) and a testing environment that evaluates the performance, overhead and scalability issues. With these contributions it is expected that the SDN controller entity is able to coordinate a handover when requested for, as quick as possible and with a minimum packet loss, providing an abstraction layer for the end-nodes involved. In order to accomplish the main objectives, several milestones were initially defined, enabling a gradual increase of complexity towards the main goal. The milestones defined were:

1. Identification of the state of the art:
   Identifying the state of the art of SDN implementations and its fields of study, allowed a more comprehensive vision of the overall SDN framework;
2. Study of OpenFlow and associated SDN mechanisms:
   An exhaustive study, exploration and testing of the OpenFlow specification and associated SDN mechanisms, had a major importance in the final experimental framework design;
3. Study of SDN, protocol implementation and software suites:
   A rigorous search for how to implement the SDN-enabling protocols and software enabled the evaluation of different solutions, performing a comparative term between used and available solutions.

1.2 METHODOLOGY

A proper beginning to design an experimental mobile OpenFlow-enabled framework, composes a study of not only the state of the art of the SDN framework, but also an exhaustive study of OpenFlow specification and SDN-enabled software. After acquiring the necessary knowledge in SDN mechanisms, this can be further capitalized by testing different SDN-enabled software, such as Mininet, which enables us to virtualize an OpenFlow network and test most of the SDN mechanisms.

The difficult part is the transitory step between the virtual emulation in Mininet and the deployment in real wireless environment, such as the AMazING[1] testbed, where external factors, such as wireless interferences and equipment restriction, need to be contemplated. In a bottom-up approach, the framework was taking shape, starting development from simple layer 2 forwarding rules to layer 3, and then extending the OpenFlow messages to the end-nodes.

With the framework fully functional, several experiences were made, exploring the handover mechanisms with high repeatability, where metrics such as handover delay, traffic throughput and

[1] http://amazing.atnog.av.it.pt/

packet loss were evaluated. Results highlighted the promising benefits of extending SDN approaches for end-to-end flow control in wireless environments.

1.3 CONTRIBUTIONS

This work explores the deployment of SDN mechanisms all the way to the mobile node, in heterogeneous wireless environments, allowing flow-based control to reach end-user devices, resulting in a contribution to an experimental implementation in a wireless environment, whose results were validated and submitted as a paper for IEEE GLOBECOM 2015, Extending SDN to End Nodes Towards Heterogeneous Wireless Mobility. Additionally, a paper focused in the study of the framework scalability, Multiple Flow in Extended SDN Wireless Mobility, was elaborated and submitted to the fourth edition of the European Workshop on Software Defined Networks.

Also, a contribution on Github[2] was made, distributing the developed code and creating a web page, which explains how to reproduce and configure a similar framework.

1.4 MASTER THESIS LAYOUT

The remainder of the master thesis is organized as follows: Chapter 2 presents a state of the art of SDN and OpenFlow mechanisms, as well as the related work on SDN in wireless environments, followed by Chapert 3 where the details of the developed framework are presented. The framework is evaluated in an end-to-end flow mobility scenario in Chapter 4, presenting the results of its deployment over a physical wireless testbed. A second scenario is presented in Chapter 5, were the framework was submitted to a multi-user environment, presenting its results. Finally, the master thesis concludes in Chapter 6, pointing out as well future work.

[2]http://atnog.github.io/of_mobilenode/

CHAPTER 2

State of the art

In the last years, the way how we use the network has changed. An increasingly amount of communication technologies and a large plethora of utilization scenarios, ranging from simple voice communications to real-time streaming of multimedia content, has contributed to an unprecedented boost in communications, number of connected devices and amount of traffic. An increasingly decisive contributor to this communications boost are mobile connections, particularly when considering mobile video [2], which explores access to content while on the move, able to take advantage of multiple access interfaces in mobile nodes (e.g., Wi-Fi, WiMax, UMTS, LTE, etc.) and overlapping networks.

Moreover, taking advantage of the powerful capabilities of modern smartphones and fast wireless networks, the users are evolving from information consumers to producers, generating different kinds of traffic (e.g., live video streams) with disparate requirements [2]. This raises unparalleled stringent requirements for upcoming telecommunication architecture innovations, going above the support of optimized mobility for devices, but actually into how to optimize the different traffic flows to and from a device, ensuring that they are used by the best network interface possible at each moment. Supporting new architectures capable of evolution and support for integrated services, 5G networks aim to come with a flexible, scalable and robust end-to-end smart integrated network, which is able to cope with the requirements imposed by both fixed and wireless accesses infrastructures. Besides, a great support for Software Defined Networking allows not only functional programmability and elasticity aspects, but also integrated virtualization of connectivity [3]. Despite the strong support of 5G networks as a motivation for this work, its major focus is in SDN mechanisms over wireless networks.

2.1 Software defined networking

Acknowledging the new challenges of the telecommunications architecture of tomorrow, upcoming 5G networks are starting to be defined, laying its reliance on Software Defined Networking (SDN) [1] principles as one of the key building blocks. By leveraging a separation from the data and control planes, SDN adds a greater degree of flexibility to the underlying operations of a network, allowing the network to better adapt to more dynamic environments, with a central high-level entity (i.e., the

controller) able to dynamically coordinate and program the network traffic forwarding entities behavior through a software API. To achieve this, the high-level entity deploys the rules in the forwarding entities (i.e., switches), allowing them to apply intelligent behavior to traversing packet flows. The interaction between the controller and forwarding entities (control and data plane, respectively) has been the subject of interface standardization, with ONF's OpenFlow [4], representing the de-facto open-source SDN instantiation. Due to its capabilities for (re)configuring networks on the fly and its flexibility for accommodating new scenarios and applications, allowing new protocols to be tested in real production networks, the OpenFlow protocol was originally designed for network researchers. From then, it has evolved as a core component of network virtualization mechanisms and cloud-based architectures [5], becoming a key cornerstone for the enablement of upcoming 5G architectures [6].

2.1.1 SDN ARCHITECTURE

Through provided open interfaces, SDN aims the development of software capable of controlling the network resources connectivity, performing traffic inspection and modification. By decoupling the control from the data plane, the SDN architecture centralizes the network intelligence and abstracts the network infrastructure from the applications, providing an increased programmability, automation, and network control. The enterprises and carriers are then capable of building highly scalable, flexible networks that readily adapt to changing business needs [7].

The ONF organization [7], sustains that the SDN architecture addresses the following requirements:

- Support for interoperability based upon open SDN controller plane interfaces;
- Independence from the characteristics of SDN controller implementation;
- Scalability and support for recursion to encompass all feasible SDN controller architectures;
- Applicability to a wide range of data plane resources;
- Policy and security boundaries related to information sharing and trust;
- Support for management interfaces, across which resources and policy may be established, as well as other more traditional management functions;
- Co-existence with existing business and operations support systems, and other administrative or control technology domains;

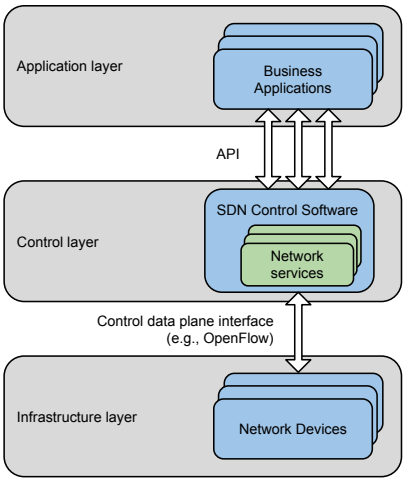


Figure 2.1: SDN architecture and respective layers

At a high level, the SDN architecture specifies the reference points and open interfaces, that can be managed to control the network resources connectivity. On the other hand, these resources can be tailored to a particular client or application, and can be interrogated and manipulated by those clients or applications. Regarding to the modeling of forwarding and processing behavior, a wild variety of media and connectivity types is supported, that includes any compute, storage or network functions and services that may cover all OSI Layers, and may be either physical or virtual.

Further, architectural considerations and specifications include co-existence with non-SDN environments and migration issues. As ONF [7] claims, SDN needs to be deployable within the context of largely pre-existing multi-player environments, comprising many organizations or businesses, with the consequent need for policy and security boundaries of information sharing and trust. Real-world constraints include the need to co-exist with existing business and operations support systems, and other administrative or control technology domains.

The ONF [7], defines three basic principles for the SDN architecture:

1. Decoupling of controller and data planes;
   Despite the separation between the control and data planes, the controller must necessarily be exercised within data plane systems. In fact, the SDN controller can delegate significant functionality to the network elements, while remaining aware of its state.
2. Logically centralized control;
   In comparison to local control, a centralized controller has a broader perspective of the resources under its control, potentially allowing better decisions. Scalability is improved by decoupling and centralizing control, enabling for increasingly global but less detailed views of network resources.
3. Exposure of abstract network resources and state to external applications;
   Applications may exist at any level of abstraction or granularity.

In Fig. 2.1, the SDN architecture is shown comprising three layers: data and control plane, and SDN applications. However, the distinction between application and control, sometimes is a matter of perspective, since the same northbound interface can be seen differently by the various stakeholders. Regarding to a SDN controller, everything further south (i.e., southbound) is a data plane, and everything further north (i.e., northbound) is an application plane. Next, the three SDN architectural layers will be explained in more detail .

2.1.1.1 DATA PLANE

Traditional networking couples all three planes, which are implemented in the firmware of forwarding entities (i.e., routers and switches), however in order to develop the network into a more programmable and flexible state, SDN decoupled the data and control planes. With this clear separation between planes, the data plane (also known as forwarding plane) is in charge of carrying the user traffic, according to the control plane logic. So, the data plane is the part that comprises the network elements and exposes their capabilities towards the control layer (i.e., control plane), via the control-data plane interface. The communication between the two planes can be performed through the OpenFlow protocol.

The control plane implementation through software brought a more dynamic network access and administration, enabling a centralized control, in a very granular level, where the traffic can be monitored and managed without having to manually reconfigure individual switches.

2.1.1.2 CONTROL PLANE

The separation of the control plane from the data plane allowed a wide range of data plane resources to be managed by a logically centralized and scalable control plane entity (i.e., the controller). With

the packet forwarding being part of the data plane, the decisions about where traffic is sent belong to the control plane itself. Moreover, the control plane is also responsible for the system configuration, management, and keeping the routing table information updated, where routing protocols such as RIP, OSPF or BGP, can be used to help the controller maintain the topology updated.

Besides of monitoring the data plane, the SDN controller also translates the applications' requirements and exerts more granular control over the network elements, while providing relevant information up to SDN applications. Services are offered to applications via the application-controller plane interface by way of an information model instance that is derived from the underlying resources, management-installed policy, and local or externally available support functions. An SDN controller may orchestrate competing application demands for limited network resources [7].

2.1.1.3 SDN APPLICATIONS

Regarding to the SDN applications, these are at the application plane, communicating via application-controller plane interface, not only their network requirements toward the controller plane, but also some traditional management functions. In fact, management is required, at least, for initial network elements setup and to assign resources to the respective controller, in data plane, or to configure the SDN controller and the policies defining the scope of control given to each SDN application, and to monitor the performance of the system, when considering the controller plane. Moreover, if we look at the application plane, the management typically configures the contracts and service level agreements (SLAs), which are enforced by the controller plane. The security associations that allow distributed functions to safely intercommunicate, operate in the three planes.

2.1.2 STANDARDIZATION

The Software Defined Networking brought a new approach to the networks. By decoupling the control from the data forwarding function, the network evolved to an extremely dynamic, manageable, cost-effective, and adaptable architecture that gives administrators unprecedented programmability, automation, and control. Organizations such as Open Networking Foundation (ONF)[1], defend a SDN implementation via an open standard, stating an extraordinary agility while reducing service development and operational costs, and frees network administrators to integrate best-of-breed technology as it is developed.

In fact, ONF presents itself as a "user-driven organization dedicated to the promotion and adoption of Software-Defined Networking (SDN) through open standards development", emphasizing an open, collaborative development process that is driven from the end-user perspective. The OpenFlow Standard,which enables remote programming of the forwarding plane, is the ONF greater mark, being "the first SDN standard and a vital element of an open software-defined network architecture".

Notwithstanding, some other organizations and technical communities also analyze SDN requirements, evolve the OpenFlow Standard to address the needs of commercial deployments, and research new standards to expand SDN benefits. Next some of them will be explored.

[1]https://www.opennetworking.org

2.1.2.1 IETF & IRTF

The Internet Engineering Task Force (IETF)[2] is "a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet". Thus, the IETF is divided in several branches, with the working groups being organized by topic into different areas (e.g., routing, transport, security, etc.). Moreover, the IETF's standards development work is organized into 8 areas, where each area has one or more area directors, which together comprise the Internet Engineering Steering Group (IESG). Despite the technical management of IETF activities, the IESG is responsible for the Internet standards process and for the final approval of specifications as Internet Standards and publication as an RFC. In this matter, the IETF mission is to produce high quality and relevant technical documents, in order to improve the Internet in terms of design, use, and management.

While IETF addresses its major focus on shorter term issues of engineering and standardization, the Internet Research Task Force (IRTF)[3] promotes research of the Internet evolution, creating a long-term Research Group working focused on topics related to Internet protocols, applications, architecture and technology.

- Software-Defined Networking Research Group (SDNRG)[4]

Despite that SDN has been conceived keeping in mind the wired infra-structure, the SDNRG defends that SDN aims to benefit all types of networks, including wireless, cellular, home, enterprise, data centers, and wide-area networks. In fact, SDNRG investigates SDN identifying the approaches that can be defined, deployed and used in the near term as well the future research challenges. Additionally, the SDNRG provides objective definitions, metrics and background research, providing this information as input to protocol, network, and service design to Standards Development Organizations (SDOs) and other standards producing organizations such as the IETF, ETSI, ATIS, ITU-T, IEEE[5], ONF, MEF, and DMTF.

2.1.2.2 IEEE

The Institute of Electrical and Electronics Engineers (IEEE) defines itself as the world's largest technical professional society, dedicated to advancing innovation and technological excellence for the benefit of humanity. The IEEE is designed to serve professionals of several study fields, such as electrical, electronic, and computing, or even related areas of science and technology that underlie modern civilization. With the computers evolving from massive mainframes to portable devices, linked to global networks connected by copper wire, microwaves, satellites, or fiber optics, the IEEE's fields of interest expanded as well. Nowadays, IEEE maintain interest not only in electrical, electronics and computing engineering, but also in micro and nano-technologies, ultrasonics, bioengineering, robotics, electronic materials, and many others.

As part of the IEEE the Open Mobile Network Interface for Omni-Range Area Networks (OmniRAN)[6] addresses its efforts on IEEE 802 access technologies. The OmniRAN (i.e., 802.1CF)

[2]https://www.ietf.org
[3]https://irtf.org/
[4]https://irtf.org/sdnrg
[5]https://www.ieee.org
[6]http://www.ieee802.org/OmniRANsg/

specifies an access network, which connects terminals to their access routers, using technologies based on the family of IEEE 802 Standards. Additionally, it also specifies the functions of the IEEE 802 technologies for heterogeneous networks, which may include multiple network interfaces, network access technologies and network subscriptions. In some cases such heterogeneous functionality must be supported in a single user terminal.

Moreover, OmniRAN intends to unify the support of different interfaces, enabling shared network control and the use of SDN principles, thereby lowering the barriers to new network technologies, operators and service providers

2.2 SDN-ENABLED ENVIRONMENTS

A number of research efforts have focused on novel solutions for emulation/simulation of SDN network. The available solutions provide a reference and material to analyze and explore the concepts addressed along this thesis. Besides of presenting some possible technologies capable of implementing the designed framework, this section presents an overview of them, highlighting their architecture, features and limitations.

2.2.1 MININET

Mininet[7] is defined as a network emulator for SDN systems, with the capacity to generate OpenFlow networks that can be connected to an external SDN controller, without the need of hardware resources.

> "Mininet is a *network emulator* which creates a network of virtual hosts, switches, controllers, and links. Mininet hosts run standard Linux network software, and its switches support OpenFlow for highly flexible custom routing and Software-Defined Networking."
>
> Mininet

The fact that Mininet povides tools for automatically generate topologies, its hosts behave like real hosts and the Open vSwitch[8] is used as the default OF switch in Mininet, enabling a fast framework deployment before the real environment.

Despite its fast framework deployment and flexibility, Mininet lacks on the real world environment, especial in wireless, where the external factors are really important.

2.2.2 AMAZING TESTBED

Wireless Networks, and especially mobile networks, have been a hot-topic on the research community in the past years. Simulation tools have therefore been used to conduct such studies in a controlled manner with far less effort. However, with increasingly complex systems, the research community has

[7] http://mininet.org/overview/
[8] http://openvswitch.org/

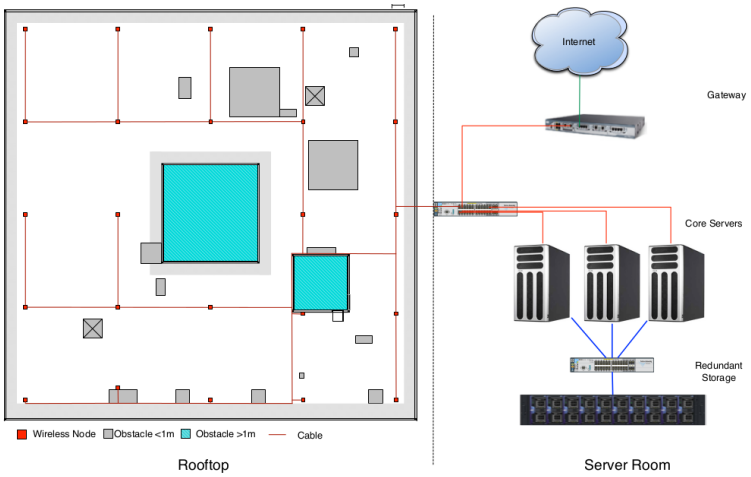


Figure 2.2: Architecture of the AMazING testbed with detail over the spatial deployment on the rooftop [8]

questioned in recent years the accuracy of such simulations, based on models of the wireless reality. This led to an increased interest in deploying testbeds were wireless studies could be conducted under more realistic (but still controllable) conditions. Such testbeds constitute an intermediate step between simulations/emulations and a full-scale prototype. The objective of such testbeds is to provide the means for the research community to validate their concepts and theories in an environment that better matches a real scenario [8].

As a wireless testbed, the AMazING[9] (Advanced Mobile wIreless Network playGround) testbed [8] is an outdoor system, Operating System agnostic, sited in the rooftop of Instituto de Telecomunicações (IT) in Aveiro, Portugal. Composed by 24 fixed nodes distributed across $1200m^2$ forming a grid with approximately 8m between each neighbor, it was deployed to support researches on next generation networks (NGN), with controllability (for the experimenter) and high reproducibility of the tests. Besides of providing to the users a full access to the node devices as the possibility to expand its capabilities by locating the core functions that eventually access the nodes' wireless interfaces, this testbed allows, through a management infrastructure, a coarse (by hardware) and/or a fine (by software modules) control of the testbed.

2.2.3 ENVIRONMENTS COMPARISON

As a powerful platform capable of developing custom topologies and virtualize hosts that run almost any program, Mininet comes recommended in several OpenFlow tutorials. Notwithstanding, Mininet deprives wireless networks support.

The use of the AMazING fulfills all the requirements, producing a framework under more realistic conditions than Mininet, yet the experimenter still has controllability and high reproducibility of the tests. Moreover, the AMazING offers the possibility to virtualize any needed network component.

[9]http://amazing.atnog.av.it.pt/

2.3 SDN ENTITIES

The SDN architecture comprises three planes (explored in section 2.1.1), with the connection of the data and control plane being performed via OpenFlow protocol. Towards this concern, the SDN entities (i.e., the SDN controller and the forwarding device) are those that enable the connection between planes. Belonging to the control plane the SDN controller performs the management of the forwarding devices (i.e., SDN switches), which remain at the data plane, while keeping the controller aware of the network state. In the following subsection these SDN entities will be explored, while comparing some of the technologies presented in the market.

2.3.1 SDN CONTROLLER

As mentioned before, the SDN concept separates the control plane from the data plane within the network, allowing the intelligence and state of the network to be managed centrally while abstracting the complexity of the underlying physical network. SDN controller, or OpenFlow controller, when the environment is based on OpenFlow protocol, provides the services that can realize a distributed control plane, as well as abet the concepts of ephemeral state management and centralization.

In this matter, an orchestration of the framework is expected from the SDN controller, by coordinating the number of interrelated resources, often distributed across subordinate platforms, and to assure transactional integrity as part of the process. In fact, a SDN controller has higher scope at a lower-level than an orchestrator, performing an orchestration across its own control domain. In the SDN controller, different agents, at the same time, may expose control over the network at different levels of abstraction or function sets. However, a network element or SDN controller have only one logical management interface, and therefore only one coordinator.

A quick search on the Internet shows several SDN/OpenFlow controller written in different programming languages (Table 2.1). Next, two of the most popular SDN/OpenFlow controllers will be described and compared.

2.3.1.1 OPENDAYLIGHT

Nowadays networks are evolving to a programmable state, chasing a potential improve on the level of functionality, flexibility and adaptability of the mainstream data centers. To achieve this goal Software Defined Networking (SDN) and Network Functions Virtualization (NFV) platforms appear, enabling network control and programmability.

Product of a "combination of open community developers and open source code and project governance that guarantees an open, community decision making process on business and technical issues", OpenDaylight[10] is built upon an open source SDN and NFV controller, enabling users to reduce operational complexity, extend the life of their existing infrastructure hardware and enable new services and capabilities available in SDN. So OpenDayLight can be seen as a flexible framework that allows organizations to deploy SDN and NFV as they please.

[10]http://www.opendaylight.org/

2.3.1.2 NOX / POX

NOX[11] as part of Software-defined Networking ecosystem, is a platform for building network control applications. With the growth of SDN academic projects, and the recognition of OpenFlow as one of the main SDN technologies, standardized by Open Networking Foundation[12] (ONF), NOX was initially developed at Nicira Networks side-by-side with OpenFlow, being the first OpenFlow controller.

As NOX's younger sibling appeared POX[13], bringing forward, through Python, a new rapid development and prototyping platform for the network control software, living up research as its primary target. Nowadays, besides of being one of a growing number of frameworks to write an OpenFlow controller, it is also an enabling framework to interact with OpenFlow switches.

POX claims the following advantages over NOX: [9]

- POX has a Phytonic OpenFlow interface.
- POX has reusable sample components for path selection, topology discovery, etc.
- POX runs anywhere and can be bundled with install-free PyPy runtime for easy deployment.
- POX specifically targets Linux, Mac OS, and Windows.
- POX supports the same GUI and visualization tools as NOX.
- POX performs well compared to NOX applications written in Python.

Both NOX and POX currently communicate with OpenFlow v1.0 switches and include special support for Open vSwitch.

2.3.1.3 CONTROLLERS COMPARISON

This section mitigates some open-source OpenFlow controllers presented out there, considering the clean code base, documentation existence beyond of the API reference and a supportive community, and how easy and fast is the deployment cycle (i.e., compile the code changes and start the controller). The major focus will be made on the OpenDayLight and POX controllers. Besides the open-source OpenFlow controllers already mentioned, we can find some others such as JAVA based Beacon[14] and its successor Floodlight[15], or the Ryu[16] and Trema[17] based on Python and Ruby respectively. Several other OpenFlow controllers can be found at Stanford University website[18].

As mentioned before POX and NOX (POX's older sibling), are maintained by the same organization. However, POX creates an OpenFlow controller more "developer-friendly" and due to the Python being an interpreted language the develop-and-deploy cycle was truly reduced, compared to NOX. POX also provides a web API (via JSON-RPC) and a good collection of manuals on its wiki.

On the other hand, OpenDaylight is an industry-supported Linux Foundation project, quite similar to Floodlight and with similar feature set, being both written in JAVA. Further, OpenDaylight follows a controller model that, in addition to OpenFlow, alternative south-bound protocols can be introduced.

[11] http://www.noxrepo.org/nox/about-nox/
[12] https://www.opennetworking.org/
[13] http://www.noxrepo.org/pox/about-pox/
[14] https://openflow.stanford.edu/display/Beacon/Home
[15] http://www.projectfloodlight.org/floodlight/
[16] http://osrg.github.io/ryu/
[17] http://trema.github.io/trema/
[18] http://yuba.stanford.edu/ casado/of-sw.html

This facet significantly differs OpenDaylight from the other controllers and lets you use switches employing non-OpenFlow proprietary control protocols. Table 2.1, shows the main features for the open-source OpenFlow controllers above mentioned.

Table 2.1: OpenFlow controllers features

Feature	**NOX**	**POX**	**Beacon**	**Floodlight**	**OpenDaylight**
Language(s) the controller is written in	C++	Python	Java	Java	Java
Language(s) supported by the controller	C, C++, Python	Python	Java	Java, Python	Java
Is actively developed?		✓	maintained	✓	✓
Has an active community?		✓	✓	✓	✓
Easy to install?		✓	✓	✓	✓
Easy to program?		✓	✓	✓	✓
Is documented?		✓	✓	✓	some
Provides a REST API?		✓	✓	✓	✓
Have utility functions?				✓	
Has a user interface	Python+QT4	Python+QT4, Web	Web	Java, Web	Web
Supports hosts with multiple attachment points?				✓	✓
Supports topologies with loops?		✓		✓	✓
Supports non-OF island connections?				✓	✓
Supports OF island connections with loops?					✓
Provides an abstraction layer above south-bound protocols?					✓
Supports OpenStack Quantum?				✓	✓

Adapted from http://vlkan.com/blog/post/2013/07/31/openflow-controllers/.

In order to avoid potential issues from controller limitation or core bugs, stability issues, or lack of documentation and community support, the choice of the OpenFlow controller that fulfills our needs can be quite challenging. On one hand POX has been progress on improving its documentation and components, however if more maturity is needed to deploy the system OpenDaylight (supported by Cisco, Juniper, Brocade, IBM) may be a better choice. To this specific framework, POX was the better choice, since the host equipment has a relative reduction of processing power, with POX's software being lighter than OpenDaylight's. Its fast development was also a point on its favor.

2.3.2 SDN SWITCH

In traditional networking the control plane (high level routing) and data plane (packet forwarding) are implemented in the same device (conventional switch). However, as mentioned before, with the SDN development the control plane was decoupled from the data plane, implementing through software a separate SDN controller capable of controlling the control plane and making high level routing decisions.

Although the data plane remains implemented in the switch, it performs a regular communication the SDN controller through OpenFlow controller. Hereupon, an OpenFlow switch can be seen as a "software program or hardware device that forwards packets in a software-defined networking (SDN) environment. OpenFlow switches are either based on the OpenFlow protocol or compatible with it".

2.3.2.1 OPEN VSWITCH

As mentioned in the previous section, OpenFlow switches can be either physical or virtualized. The Open vSwitch, as an open source software switch, belongs to the second group. Designed to be used in virtualized server environments, the OvS is capable of forwarding traffic between different virtual machines (VM) on the same physical host or between the virtual machines and the physical network. This capability besides of making the management of the VM network configuration easier, also allows to monitor state spread across physical hosts in dynamic virtualized environments. Other virtual switch applications as VMware, vNetwork distributed switch and Cisco Nexus 1000V, also offer this capability, however OvS makes the process easier by running on each physical host and supporting remote management.

Thinking about software defined networks and flow-based forwarding, OvS includes beyond of the standard management protocols (e.g., sFlow, NetFlow, IPFIX, RSPAN and CLI), two open protocols: OpenFlow and OVSDB, that expose the flow-based forwarding state and the switch port state, respectively.

Besides of being the most used and with the bigger community, Open vSwitch goes beyond others and implements a kernel module, enabling it to motorize the intern traffic, turning it crucial for the mobile node development.

2.3.3 OPENFLOW SWITCHES

In the previous sections, the Controller was explored, which communicates via OpenFlow protocol with OpenFlow switches. On the other hand, besides the OpenFlow channel to communicate with an external controller, an OpenFlow Switch consists of one or more flow tables and a group table, which perform packet lookups and forwarding (Fig. 2.3). Additionally, using the OpenFlow protocol, the controller can manage these flow tables, by adding, updating or even deleting a flow entry. This can be done both reactively (in response to packets) and pro-actively. Moreover, in a OpenFlow switch, each flow table contains a set of flow entries, and each flow entry consists of match fields, counters, and a set of instructions to apply to matching packets [10]. Fig. 2.4 illustrates the pipeline process of a packet flow and its intrinsic operations are explained in the following sub-sections.

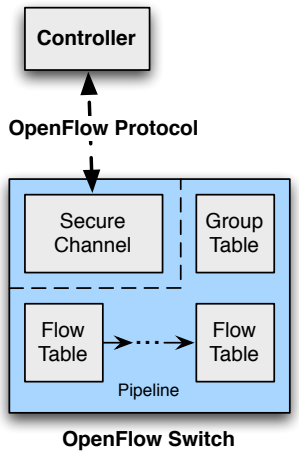


Figure 2.3: Main components of an OpenFlow switch [10]

2.3.3.1 MATCHING

The matching process starts at the first flow table and may continue to additional flow tables, with the flow entries matching the packets in priority order. When a matching entry is found, the instructions associated with the specific flow entry are executed, but if no match is found in a flow table, the outcome may vary. The table-miss flow entry can be configured to forward the packet to the controller over the OpenFlow channel, drop it, or continue to the next flow table [10].

2.3.3.2 INSTRUCTIONS

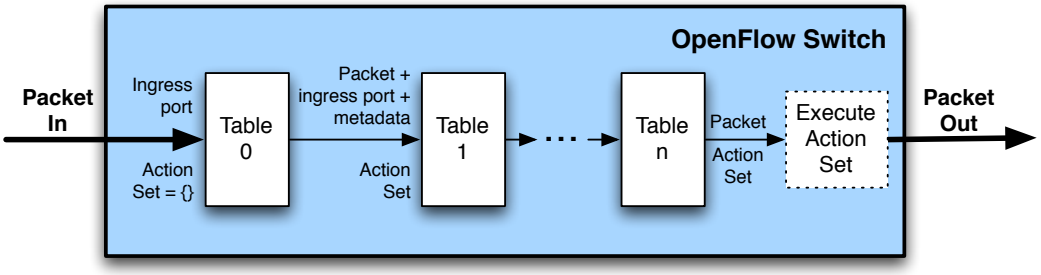
The instructions associated with each flow entry either contain actions or modify pipeline processing, where these actions describe packet forwarding, packet modification and group table processing. On the other hand, the pipeline processing instructions allow packets to be sent to subsequent tables for further processing and allow information, in the form of metadata, to be communicated between tables. When the instruction set associated with a matching flow entry does not specify a next table, the table pipeline processing stops. Then the packet is usually modified and forwarded [10].

2.3.3.3 FLOW ENTRIES

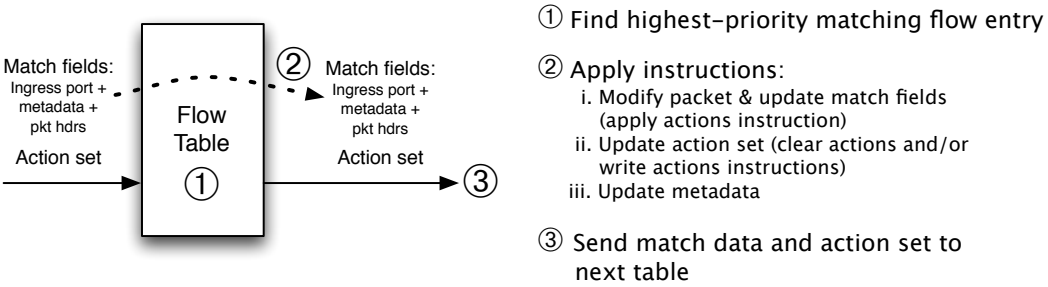
Flow entries may forward to a port, that is usually a physical port, but it may also be a logical port defined by the switch or a reserved port defined by the specifications. The reserved ports may specify generic forwarding actions such as sending to the controller, flooding, or forwarding using non-OpenFlow methods, such as “normal” switch processing, while switch-defined logical ports may specify link aggregation groups, tunnels or loopback interfaces [10].

2.3.3.4 ACTIONS

Actions associated with flow entries may also direct packets to a group, which specifies additional processing and represents sets of actions for flooding, as well as more complex forwarding semantics (e.g. multipath, fast reroute, and link aggregation). The group table contains group entries, and each group entry contains a list of action buckets with specific semantics dependent on group type. Lastly, the actions in one or more action buckets are applied to packets sent to the group [10].



(a) Packets are matched against multiple tables in the pipeline



(b) Per-table packet processing

Figure 2.4: Packet flow through the processing pipeline [10]

2.3.4 OPENFLOW-HYBRID

The OpenFlow-compliant switches can either be OpenFlow-only, where only OpenFlow operations are supported, with all packets being processed by the OpenFlow pipeline, or OpenFlow-hybrid. The OpenFlow-hybrid switches support both OpenFlow and normal Ethernet switching operations (i.e., traditional L2 Ethernet switching, VLAN isolation, L3 routing), and should provide a classification mechanism outside of OpenFlow that routes traffic to either the OpenFlow or normal pipeline. The switch may use the VLAN tag or input port of the packet to decide whether to process the packet using one pipeline or the other, or it may direct all packets to the OpenFlow pipeline. An OpenFlow-hybrid switch may also allow a packet to go from the OpenFlow pipeline to the normal pipeline through the "NORMAL" and "FLOOD" reserved ports.

2.4 OPENFLOW PROTOCOL

Added as a feature to commercial network equipment (e.g., switches, routers and wireless access points), and implemented by major vendors, OpenFlow provides a standardized hook to allow researchers to run experiments, without requiring vendors to expose the internal workings of their network devices. Unlike classical routers and switches, an OpenFlow Switch separates the data path from the control path (responsible for packet forwarding and routing decisions, respectively), with the data path portion still residing in the switch, while high-level routing decisions are moved to a separate controller. The communication between these two network elements (i.e., OpenFlow switch and controller) is performed via the OpenFlow protocol, which defines different action and messages (e.g., packet-received, send-packet-out, modify-forwarding-table, get-stats, etc.) [4].

2.4.1 OPENFLOW CHANNEL

The connection of an OpenFlow switch to a controller, is performed through an OpenFlow channel interface (Fig. 2.3), that allows the controller not only to communicate with the siwtch, but also to configure and manage it via OpenFlow messages. Despite that the datapath and OpenFlow channel interface are implementation-specific, all OpenFlow channel messages must be formatted according to the OpenFlow protocol. The OpenFlow channel is usually encrypted using TLS, but may be run directly over TCP [10].

Regarding to the OpenFlow messages types, the OpenFlow protocol defines the controller-to-switch messages that are initiated by the controller and used to directly manage or inspect the state of the switch; the asynchronous messages that are initiated by the switch and used to update the controller of network events and changes to the switch state; and finally, the symmetric messages are initiated by either the switch or the controller and sent without solicitation.

2.4.2 OPENFLOW MESSAGES

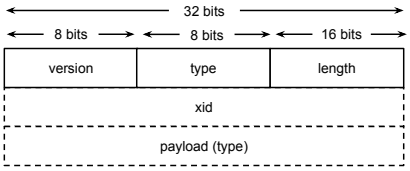


Figure 2.5: OpenFlow packet

Fig. 2.5 illustrates the header structure of an OpenFlow message. With a fixed structure, and common to all OpenFlow messages, it serves three roles that are independent of the version of OpenFlow being used (i.e., *version*, *length* and *xid*). The *version* field remains in the first 8 bits and indicates the version of OpenFlow which this message belongs to. The current OpenFlow version is 0x05 (v1.4). The *length* field indicates the end point of the message in the byte stream starting from the first byte of the header. The third version independent role is transaction identifier (i.e., *xid*), that is a unique value used to match requests to responses. Regarding to the *type* field, it indicates the message type and how to interpret the *payload*. These roles are version dependent. Next, the different types of OpenFlow messages will be described.

2.4.2.1 CONTROLLER-TO-SWITCH

The Controller-to-switch messages are initiated by the controller and may or may not require a response from the switch. Below, the different types of these messages are explained [10].

- Features: The controller may request the capabilities of a switch by sending a features request. Then the switch must respond with a features reply specifying its capabilities. This is commonly performed upon establishment of the OpenFlow channel.
- Configuration: The controller is able to set and query configuration parameters in the switch. The switch only responds to a query from the controller.
- Modify-State: This type of messages are sent by the controller to manage state on the switches. Their primary purpose is to add, delete and modify flow/group entries in the OpenFlow tables and to set switch port properties.
- Read-State: Read-State messages are used by the controller to collect various information from the switch, such as current configuration, statistics and capabilities.

- Packet-out: Used by the controller to send packets out of a specified port on the switch, and to forward packets received via Packet-in messages, packet-out messages must contain a full packet or a buffer ID referencing a packet stored in the switch. The message must also contain a list of actions to be applied in the order they are specified, or the packet will be dropped.
- Barrier: Barrier request/reply messages are used by the controller to ensure message dependencies have been met or to receive notifications for completed operations.
- Role-Request: Specially useful when the switch connects to multiples controllers, these messages are used by the controller to set the role of its OpenFlow channel, or query that role.
- Asynchronous-Configuration: The Asynchronous-Configuration messages are used by the controller to set an additional filter on the asynchronous messages that it wants to receive on its OpenFlow channel, or to query that filter. This is mostly useful when the switch connects to multiple controllers and commonly performed upon establishment of the OpenFlow channel.

2.4.2.2 ASYNCHRONOUS

Asynchronous messages are sent by the switch to the controller without its solicitation, denoting a packet arrival, a switch state change, or an error. The four main asynchronous message types are described below [10].

- Packet-in: Transfer the control of a packet to the controller. For all packets forwarded to the "CONTROLLER" reserved port using a flow entry or the table-miss flow entry, a packet-in event is always sent to controllers. Other processing, such as TTL checking, may also send packets to the controller using packet-in events. When a packet-in is generated by an output action in a flow entry or group bucket, it can be specified individually in the output action itself, for other packet-in it can be configured in the switch configuration.
  In some cases the Packet-in events can be configured to buffer packets, implying that the switch has sufficient memory to buffer them. In these cases the packet-in events contain only some fraction of the packet header and a buffer ID to be used by a controller when it is ready for the switch to forward the packet. In the other hand, if the Switches do not support internal buffering or have run out of internal buffering, the full packet has to be sent to the controllers as part of the event.
  Buffered packets will usually be processed via a Packet-out message from a controller, or automatically expired after some time. If the packet is buffered, the number of bytes of the original packet to include in the packet-in can be configured. By default, it is 128 bytes. For packet-in generated by an output action in a flow entry or group bucket, it can be specified individually in the output action itself, for other packet-in it can be configured in the switch configuration [10].
- Flow-Removed: Inform the controller about the removal of a flow entry from a flow table. Flow-Removed messages are only sent for flow entries with the OFPFF_SEND_FLOW_REM flag set. They are generated as the result of a controller flow delete requests or the switch flow expiry process when one of the flow timeout is exceeded.
- Port-status: Inform the controller of a change on a port. The switch is expected to send port-status messages to controllers as port configuration or port state changes. These events

include change in port configuration events, for example if it was brought down directly by a user, and port state change events, for example if the link went down.

- Error: The switch is able to notify controllers of problems using error messages.

2.4.2.3 SYMMETRIC

Symmetric messages are sent without solicitation, in either direction [10].

- Hello: Hello messages are exchanged between the switch and controller upon connection startup.
- Echo: Echo request/reply messages can be sent from either the switch or the controller, and must return an echo reply. They are mainly used to verify the liveness of a controller-switch connection, and may as well be used to measure its latency or bandwidth.
- Experimenter: Experimenter messages provide a standard way for OpenFlow switches to offer additional functionality within the OpenFlow message type space. This is a staging area for features meant for future OpenFlow revisions.

2.5 SDN IN WIRELESS NETWORKS

The flexibility introduced by SDN stemmed considerations on the application of SDN mechanisms into wireless and mobile networks, progressing its original deployment concept beyond wired-technologies, as identified by Open Network Foundation[19] (ONF) Wireless & Mobile Working Group (WMWG) [11]. In this sense, some contributions are already made and considered scenarios highlighted the possibilities provided by traffic steering under SDN principles, in use cases such as mobile offloading at a very granular level (i.e., including the session, user, device and application levels), contributing to the optimization of Radio Access Network (RAN) resources by operators, and improving the overall Quality of Experience (QoE) for data-intensive mobile applications. Contributions such as Mobileflow [12], where the traditional routing controller is complemented with a dedicated mobility one, and SoftCell [13], where data flows for mobile core networks are distributed according to a set of policies enforced by SDN, are at the core of different SDN-based mobile network architectures.

Still, the enhancements of such designs stop at the network edge and do not explore any integrated vision including the wireless accesses themselves. Nevertheless, initiatives regarding the use of OpenFlow into wireless environments were introduced in [14] with its implementation over OpenWrt, allowing small office/home wireless routers to act as an OpenFlow-enabled switch. In other solutions, such as [15] and [16], an OpenFlow Controller uses IEEE 802.21 [17] not only to control and acquire information of several properties of the wireless link interfaces but also manages and controls the handover procedures of mobile devices. Moreover, the current OpenFlow specification (v1.4) sees an IEEE 802.11 interface as an unknown interface connected to the switch, with unknown properties [18], evidencing a lack of mechanisms for fully supporting wireless interfaces. In [19] and [20], besides of mitigating some of these issues, they explore the configuration of wireless datapath elements (such as control power, data rate, SSID, etc.) through SNMP, in order to acquire properties and to capture events in wireless environments, such as the association of a host with an Access Point (AP). Projects

[19] `https://www.opennetworking.org/`

such as OpenRadio [21], SoftRAN [22] and OpenRF [23], study the SDN and wireless interactions for wireless-based operations handled centrally at a controller

However, wireless communications are not just limited to the infrastructural component, and also involve the receiving node entities, (e.g., the user terminal). Independently of all the control enhancement contributions to the operation of the network that SDN may have, the increasing heterogeneous environment provided by different access technologies, user services and preferences, place the user terminal at an important spot for further optimization. In [24] and [25] a conceptual architecture where SDN programmability is extended to the mobile node is presented, enabling and simplifying mobility-based scenarios and device-to-device possibilities. However, neither provides experimental results of these approaches. Conversely, [26] experimentally evaluates a set of proposed mobile extensions for SDN, but aims for a user-to-cloud connectivity scenario and focuses on the control of specific technology-dependent wireless aspects (i.e., TDMA slicing for 802.11).

This is where this work contributes. It explores the deployment of SDN mechanisms all the way to the mobile node, in heterogeneous wireless environments, allowing flow-based control to reach end-user devices, providing what such a networking paradigm really aims for: controlling data flows. It exploits a generic SDN deployment, able to operate independently of the underlying wireless technology or the connectivity scenario, providing feasibility guidelines for taking SDN mechanisms into the mobile terminal domain.

2.6 CHAPTER CONSIDERATIONS

This chapter explored the state of the art of SDN, addressing its focus on SDN over wireless environments. Besides the SDN architecture, where its layers were explained, the available solutions of softwares and SDN/OpenFlow entities for a mobility framework in an OpenFlow environment deployment were studied. The OpenFlow protocol and its type of messages were also considered.

Moreover, it was noticeable different meanings for the SDN/OpenFlow controller, but a simple way to be described is making a comparison to a sort of operating system for the network, where the OpenFlow protocol connects the controller software to the network devices, performing communication between the server software and switches, allowing the controller to configure network devices and choose the best path for application traffic. By implementing a network control plane through software, rather than the firmware of hardware devices, network traffic can be managed more dynamically and at a much more granular level.

CHAPTER 3

An End-to-End SDN-enabled Framework for Mobile Devices

This chapter will explore the designed framework and its requirements, not only presenting the chosen technologies and the framework architecture, but also exploring and justifying the configuration of each framework's network component. Regarding to the framework, its architecture, signaling and involved network nodes will be detailed. The framework here described is capable of performing a handover, through SDN mechanisms, when triggered by an end-node. Furthermore, optimization scenarios are allowed through the addiction of a traffic monitor mechanism, added to wireless access points, that is capable of triggering the handover if a certain threshold of load is reached.

3.1 INTRODUCTION

The deployed framework aims to explore the SDN mechanisms, by extending it all the way to the mobile end-node. This SDN extension enables the network controller entity not only an end-to-end data flow management, but also the capability of performing a handover through OpenFlow mechanisms. In this sense, handover can be seen as the process of transferring an ongoing data session from one channel connected to the core network to another with the minimum packet loss possible.

Towards this concern, the presented framework, explores a wireless dual-interfaced Mobile Node (MN) attached to the remained network through two different access points. Supporting SDN mechanisms, the MN has a established connection with the network controller (i.e., OpenFlow controller), while streaming towards the listener node through one of attached access points. For experimental proposals, it was considered that the MN has the capability of detecting the poor link quality and the packet loss, motivating a handover solicitation through the OpenFlow messages (i.e., packet_in). Due to the controller be centralized, it remains aware of not only the network state, but also the mobile nodes attached points, enabling it to handovering the MN, by redirecting the its data flows through flow modifications (i.e., flow_mod) messages.

3.2 FRAMEWORK REQUIREMENTS

In order to develop the above mentioned system framework it was used an AMazING node for each emulated network component (i.e., access points, router, controller, mobile nodes and listener node). The hardware of these network nodes needed to support the used application (i.e., OvS, POX and VLC) and also supporting a remote control. To run POX, it officially requires Python 2.7, and should run under Linux, Mac OS, and Windows. Additionally, the mobile node and access points require wireless physical interfaces hardware. Each access point requires at least one wireless interface, monitored by hostapd[1], while the mobile node requires at least two. Despite the mobile node that is attached to the network via wireless, the remaining nodes require to be interconnected through an Ethernet physical interface.

3.3 FRAMEWORK ARCHITECTURE

In the past years, networks have been evolving to a software defined concept, where the network control plane and data plane are separated, being the OpenFlow protocol responsible for the connecting of the two planes. The framework here presented explores a fundamental evolution aspect when compared with the well known software defined networks, implementing an OpenFlow control signaling extension all the way to end nodes. Fig. 3.1 illustrates how the OpenFlow controller entity directly interacts with the end nodes (i.e., the mobile node and the listener), besides its ability to control network endpoints that may connect to different access technologies. By receiving these OpenFlow messages, the end node is able to realize packet and flow-level actions, in a similar way to OpenFlow switches, such as redirecting packets to other entities, or moving traffic flows between different mobile interfaces, allowing the network to enforce traffic optimization strategies.

The end nodes can further contribute to the signaling itself, assisting the network decisions and policy enforcement by generating signaling, in the form of events and handover triggers, due to, e.g., changes in the link quality, the discovery of alternative links or purely on user device preferences. This can be supported by the usage of *Packet_in* messages sent towards the controller, which then, after deciding changes to flows affecting the involved node, responds with a *Flow_mod* message deploying such changes. However, despite the focus on OpenFlow and the thorough evaluation of its signaling impact in end devices, this framework is flexible enough to be integrated with other signaling mechanisms for triggering (or commands), such as IEEE 802.21 [17]. Additionally, in chapter 5, the impact caused by different numbers of mobile node flows under the influence of a single controller, was studied.

[1]hostapd is a user space application that runs as a background process for wireless access points. More properly it is an IEEE 802.11 AP and IEEE 802.1X/WPA/WPA2/EAP/RADIUS Authenticator

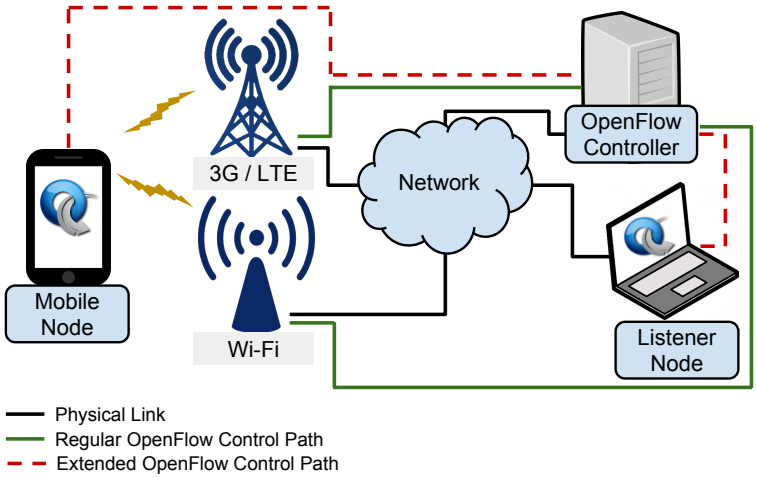


Figure 3.1: Wireless Offloading Scenario

3.3.1 SIGNALING

In order to properly assess the extensions provided by this framework, an utilization scenario is first illustrated, where the SDN signaling exchanged between the controller and mobile end-node is presented. This scenario features a dual-interfaced MN that is connected to two different access networks simultaneously, and was extended with OpenFlow support as already mentioned. This node, despite being mobile, acts as a mobile content source, streaming a live video towards a listener node, connected through another access. The signaling associated to this scenario is shown in Fig. 3.2.

The scenario (dubbed *case A*) begins with the MN already connected to both APs, with the controller connection already established, sending video stream towards the listener node through AP1. By detecting a degradation in the wireless link quality (e.g. due to load, high bandwidth consumption from other nodes or a decrease in signal strength), the MN sends an OpenFlow *Packet_in* message (Fig. 3.2 - 2a) to the OpenFlow controller.

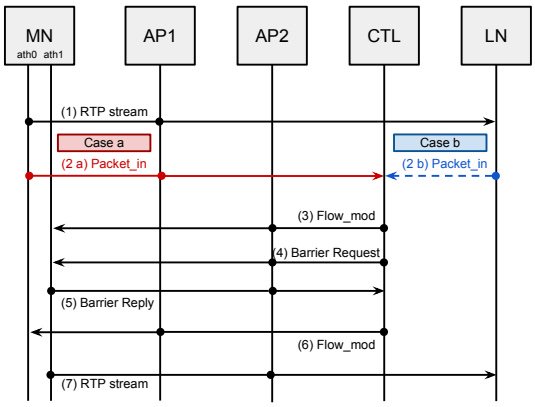


Figure 3.2: Signaling flow diagram

This *Packet_in* message is used as a handover trigger, and, upon its reception by the Controller, it generates a handover decision for moving the video flow from the MN's interface connected to AP1, to its interface connected to AP2. This handover decision is carried by the Controller which sends an OpenFlow *Flow_mod* message (Fig. 3.2 - 3) for each MN interface involved in the handover. However, after the first *Flow_mod*, a *Barrier_request* is sent, ensuring that the second *Flow_mod* (Fig. 3.2 - 4 and 6, respectively) is sent only when the first is already executed, avoiding packet loss or *ping-pong* effects during the handover. Therefore, mimicking a *make-before-break* approach, the first *Flow_mod* is sent through the interface connected to AP2 (i.e., *ath1*), preemptively implementing a rule enabling it to forward (to the outside) the traffic coming from the interface connected to AP1 (i.e., *ath0*). The second *Flow_mod* message implements the forwarding from *ath0* to *ath1*. In order to perform this forwarding, the rule for the second interface needs to implement the *"MOD_SRC_ADD"*

and *"MOD_DST_ADD"* actions, to fulfill the wireless handshake. Both actions change the source and destiny Media Access Control (MAC) Address respectively.

A secondary scenario for evaluation (dubbed *case B*), where the trigger (i.e., the OpenFlow *Packet_in* message) is sent by the listener node, as indicated in blue in Fig. 3.2 - 2b, was also explored. This situation mimics the listener device detecting that the performance of the video decreased below a previously established threshold, originating a trigger prompting for network optimization procedures. This allows the evaluation of the OpenFlow signaling impact in both end-points of the communication path.

Focusing on the specific OpenFlow aspect of this interaction between the MN and the OpenFlow Controller, this framework does not explore specification aspects such as how the controller is aware that the MN is connected to AP2, or even how to execute link establishment procedures therein. This is also applied to how the handover decision is taken by (or reaches) the controller. In this respect, the framework can exploit northbound interfaces between the controller and a mobility management entity, or even southbound interfaces for preparing and establishing handover candidate links, for different kinds of access technologies (i.e., IEEE802.11 and 3GPP).

3.3.1.1 OPENFLOW MESSAGES

The handover signaling of the developed framework considers three types of OpenFlow messages. Next, its importance for the framework will be explained.

- Packet_in (Fig. 3.2-2a/2b):
  The packet_in is an asynchronous message sent by the switch to the controller without its solicitation. In this case the packet_in was generated by an output action in a flow entry with a specific matching, which was done through destination Internet Protocol (IP) address;
- Flow_mod (Fig. 3.2-3/6):
  The "modify flow entry message" (i.e., flow_mod) is a modify-state message, that belonging to the Controller-to-switch messages group, is sent by the controller to manage switches state. Its primary purpose is to add, delete and modify flow entries in the OpenFlow switch;
  In this framework, the first flow_mod message (i.e., Fig. 3.2-3) adds in the MN OvS bridge a flow entry, which redirects the matching traffic (i.e., the video stream) to a specific port (i.e., the patch port). Th second flow_mod (i.e., Fig. 3.2-3), also adds a flow entry, however before redirecting the matching traffic (i.e., the traffic coming from the patch port), it modifies the source and destiny MAC addresses;
- Barrier_request (Fig. 3.2-4) and Barrier_reply (Fig. 3.2-5):
  As in the previous case, the Barrier request/reply messages belong to the Controller-to-switch messages group, being sent, in this framework, by the controller to receive notifications for completed operations. Thus, these messages are used to ensure that the first flow_mod is completed before sending the second one.

3.4 FRAMEWORK DEPLOYMENT

The scenario was deployed a network topology with 6 nodes (i.e., 2 access points, router, network controller, mobile node and a terminal), exploring a mobile node as the source content, streaming a video through the network towards the listener node. Keeping that in mind, the mobile node has a wireless connection with both APs, whereas all the other nodes have a physical link connection. All nodes can ping each other.

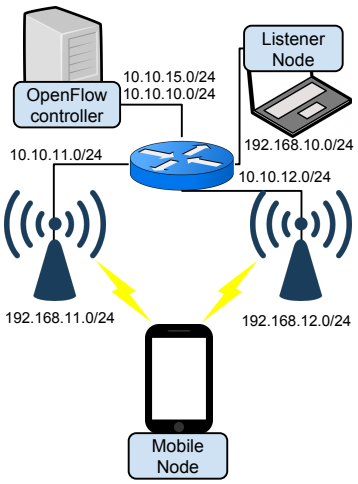


Figure 3.3: System framework configuration

This work required new configurations of the used software. Therefore, rather than presenting the software installation, the configuration used to reproduce the experience will be explained.

With the purpose of extending the OpenFlow control path up to the mobile node, the OvS with its dependencies was installed in every network node, except on the controller node, where the POX was used as the OpenFlow agent in that node. With the mobile node being the source content, its OvS configuration is somehow different from the remaining network nodes, enabling it to apply new rules over the kernels forwarding, allowing the packet redirection to the other wireless interface. To accomplish that, both OvS bridges attached to each mobile nodes's wireless interface, were interconnected by a patch port that acts as a patch cable. The next subsections explain the nodes configuration, while some examples are shown.

3.4.1 OPENFLOW CONTROLLER

By default, when a OpenFlow Controller connects to a new OpenFlow switch, it erases all forwarding tables therein. To accomplish the specification this feature was disabled, so that the connection between the MN and the Controller does not shut-down. This event is caused by the wireless handshaking protocol, since if the MN does not have any rule that allows to carry out the wireless handshaking between itself and the Access Point, the Mobile Node and Controller communication cannot be established. In order to perform those requirements and for a faster development, the chosen controller was POX[2], Fig. 3.4 e).

As explored in chapter 2, the development programing language of POX is Python, that despites requiring Python 2.7 to run, reduces the POX develop-and-deploy cycle, since Python is an interpreted language. The POX controller brings some interesting scripts as the *l3_learning*, that acts as layer 3 switch, by keeping a table that maps IP to MAC addresses and switch ports. In this sense, it was not necessary, once each network node was capable of handling its own ARP request, by the OvS bridge layer 3 configuration. In fact, OvS cannot handle the ARP's request, in this framework these messages are carried by kernel.

In addition to the original POX's library, a new Python script was created enabling POX to handle the handover solicitation from the end-devices. Also, some changes in the core POX's code were necessary, in order to avoid the controller erasing the OvS forwarding table, when a new connection is established. This requisite was mandatory for the connection with the mobile node, since if the MN's OvS bridges do not have any rule, the handover handshake cannot be performed.

[2] `https://openflow.stanford.edu/display/ONL/POX+Wiki`

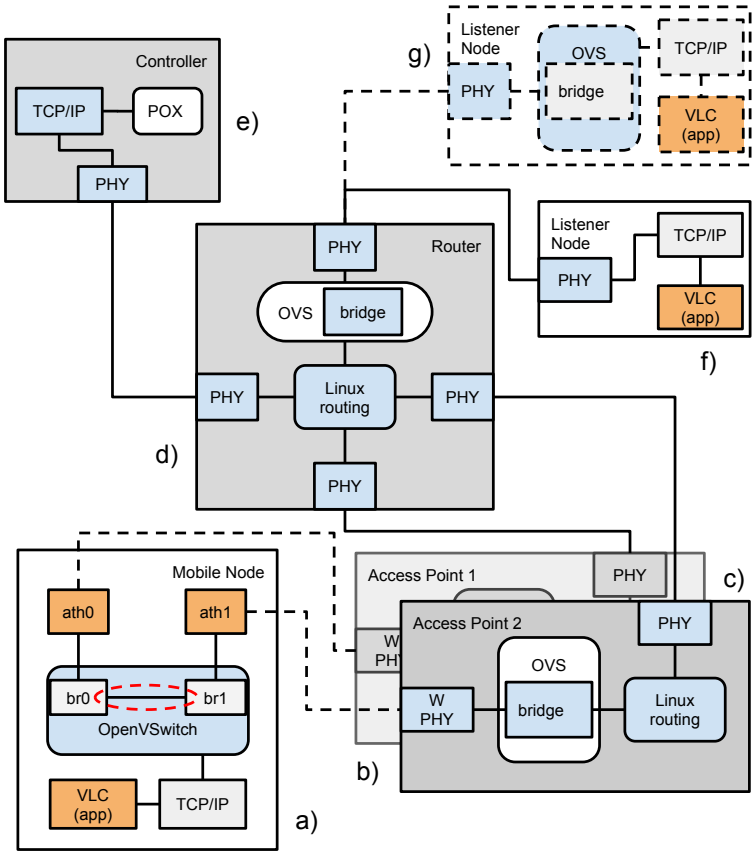


Figure 3.4: Network Modules

The Python script that gives to the controller the capability to handle the handover solicitation and to deploy the required rules is triggered when the handover solicitation (i.e., *Packet_in* message, from the source or listener) is received, as shown in Fig. 3.2. In this matter, when the Controller receives the *Packet_in* massage, it generates an interruption, and after analyzing the source of the packet, starts the procedures to perform the handover. Looking for a *make before brake* approach, the first rule is sent to the new transmission bridge (i.e., *br1* attached to *ath1*), and immediately after, a *Barrier_request* message is sent. This last message ensures that the first rule is implemented before sending the second one. Similarly to the *Packet_in*, the *Barrier_reply* message, sent with the same ID as the *Barrier_request*, will cause an interruption that motivates the sending of the second rule, addressed to OvS bridge *br0*.

In order to make the Controller accessible to the network nodes, it was also necessary, as in the previous nodes, to load the IEEE 802.1Q and configure two VLANs, allowing the controller to be accessible through two different IP addresses. The two VLANs requirement stands for the two wireless physical network interfaces of the mobile node. Then, each MN's bridge has a different IP address associated to its controller.

3.4.2 MOBILE NODE

To perform the extension of OpenFlow up to the MN, Fig. 3.4 a), Open vSwitch[3] (v2.3.1) was installed and configured with two layer 3 bridges, one for each wireless network interface. In order to interconnect both bridges for flow mobility, a pair of virtual devices (i.e., patch ports) were used, acting as patch cable. The two bridges requirement stands by the fact that the MN is a content source, implying the need for a module between the kernel TCP/IP stack and the physical network interface, since the OvS ability for layer 3 routing relies on the routing functionality that is built into the Linux kernel.

Hereupon when the physical network interface (PHY) of mobile node receives a packet, it passes it to the kernel network stack through the OvS "LOCAL" port. On the other hand, when the network node needs to send a packet, the kernel will send it through the OvS "LOCAL" port to the respective PHY. With the OvS being the intermediary between the network stack and the PHY, it is possible to set routing rules over the static rules imposed by the kernel routing table.

Regarding to the bridges configuration, both are initially configured with a low priority "NORMAL" action, allowing the communication without an external OpenFlow controller connection, and a higher priority action to allow the trigger for the handover. The "NORMAL" action allows the OvS to process the packet using traditional Layer 2 or Layer 3 processing. With the purpose of making the OvS responsible for forwarding packets in case of controller fail, the OvS bridges were configured in fail-safe-mode "standalone".

For the mobile node configuration, it was not necessary the use of IEEE 802.1Q, since it is connected to the remainder network via wireless (the use of IEEE 802.1Q will be explore in the next section). However, the IEEE 802.11g was needed, with the purpose of attaching the MN to both APs. To perform the connection between the MN and AP, first it is necessary to setup the AP's respective node as access point (through *hostapd* software, explored in next section). Then through *iwconfig* it is possible to attach the MN to AP.

As in the previous cases and for the same reasons, also the WPHY of the mobile node must be set in promiscuous mode. All the MN's configuration can be easily seen in the respective appendix (Listing. 1).

3.4.3 ACCESS POINTS, ROUTER AND LISTENER NODE

Despite of addressing several network nodes, this section shows and explains the configuration of the router. The remain nodes, can be easily extrapolated from the router's configuration script (i.e., Listing. 2). Notwithstanding, some changes, such as the respective network interfaces and IP addresses, are needed. Moreover, it is also important to set the connection mode of the controller to out-of-band. The configuration of the mobile nodes shows how this can be done.

As in the MN, it was necessary to add an OvS layer 3 configuration to the remaining network nodes, where a single bridge, with an assigned IP address, is created and attached to the physical network interface (Fig. 3.4)). Whereas in the Access Points (APs) and Router the OvS layer 3 aims to allow the layer 3 routing through the Linux kernel, in the Listener Node it aims to enable it, as in the MN case, to trigger the OpenFlow controller (i.e., Fig. 3.2 *case B*). With this configuration, the OpenFlow communication between the Listener Node and the Controller becomes possible, enabling a

[3] `http://openvswitch.org/`

handover solicitation (by sending a *Packet_in* message) from the content receiver to the OpenFlow controller.

It is important to note that with the purpose of reducing the physical requirements of the network nodes, the framework has the control and data paths sharing the same network interface. Concretely, in order to reduce generated OpenFlow traffic (specifically, support control traffic such as controller-link *keep-alives*), in our experiment we reduce as well the number of bridges. Instead of having the conventional single OvS bridge for a single interface on each network node, we create a single bridge just for one interface of the Access Points (Fig. 3.4 b) and Fig. 3.4 c)) and Router nodes (Fig. 3.4 d)). However, the whole process remains similar.

Besides of the OvS configuration, it was also needed to set some others conditions, such as creating VLANs not only because of the lack of physical interfaces in the AMazING nodes, but also to set up the environment independently of the network control interface of AMazING. In order to allow the router and the access points to forward the received packets, it was also necessary to enable the kernel's packets forwarding.

Regarding to the VLANs, initially its necessary to load the kernel's module 8021q (i.e., modprobe *8021q*). This module enables the use of IEEE 802.1Q, that stands for a networking standard that supports virtual LANs (VLANs) on an Ethernet network. This same standard defines a system of tagging for Ethernet frames and the accompanying procedures to be used by bridges and switches in handling such frames. With the module loaded, its possible to create (or add) a new VLAN through the *vconfig* command. After creating the VLAN it remains the need to set a new IP address, and set the interface up. It is also recommended to set a new MAC, in order to avoid conflicts, although it is not mandatory. These last setups can be done using the *ifconfig* command.

To start Open vSwitch software, it is also necessary to load its kernel module (i.e., modprobe *openvswitch*) and to start its database server (i.e., *ovsdb-server*). Then an OvS bridge its created attached to an Ethernet port. As already mentioned, it must be a layer 3 bridge, implying to set the Ethernet port in *promiscuous mode* and give to the bridge the Ethernet desired IP address. Setting up the *promiscuous mode* in the physical network interface, causes the controller to pass all traffic it receives to the central processing unit (CPU) rather than passing only the frames that the controller is intended to receive, forcing all packets to pass through the OvS bridge.

3.4.3.1 ACCESS POINTS

Beyond of the previous requirements, both network nodes that act as APs, use the IEEE 802.11g standard. With the purpose of configuring the node to act as AP, it was used the *hostapd* software, that can be easily installed on Linux OS, more specifically Ubuntu distribution, through the package tool *apt-get*.

3.4.4 RUNNING A DEMO

Before starting the demo, the mobile node (MN) not only requires a "NORMAL" action and rule that forwards the handover solicitation as a *packet_in* message, but also an already established link with both APs. For the second *case* (i.e., *case b*), the OvS forwarding rule that sends the handover solicitation through a *packet_in* message needs to be installed in the listener node (LN), rather than

in MN. Moreover, the static rules that perform the forwarding between the network nodes, need to be implemented in the kernel's routing table. Below, it is shown the experience running process:

1. Video stream starts:
   cvlc -dvvv BigBuckBunny1080p.mp4 –sout '#duplicatedst=rtpmux=ts,dst=192.168.10.1,port=5004'
   IP = 192.168.10.1, Real-time Protocol (RTP), port = 5004;
   The MN sends the RTP packets towards LN through the interface attached to the AP2 (i.e., *ath0*);
2. After 15 seconds the signal quality decrease:
   tx-power = 1, bitrate = 5 Mbps;
3. After 10 seconds of bad signal quality its sent the handover solicitation:
   case a: MN sends *packet_in* - UDP packet sent to the IP addess 11.11.11.11;
   case b: LN sends *packet_in* - UDP packet sent to the IP addess 11.11.11.11;
   In both cases, the OvS has implemented a forwarding rules that sends to the Controller the packets with *destination IP 11.11.11.11*, through a *packet_in* message;
4. Controller receives and analyzes the *packet_in* message;
5. Controller sends the first *flow_mod* message, and promptly sends the *barrier_request* message;
6. MN implements the flow modification and responds the *barrier_request* with a *barrier_reply* message;
7. Controller *barrier_reply* message, analyzes it and sends second *flow_mod* message;
8. MN receives the second *flow_mod* message and starts redirecting flows.

The time values used in both cases are illustrative, with the framework being flexible enough to integrate mechanisms that allow dynamic configuration of different triggering opportunities.

3.5 CHAPTER CONSIDERATIONS

This chapter addressed the designed framework capable of performing a handover, through SDN mechanisms, when triggered by an end-node. It explored not only the framework requirements, but also its architecture, exploring its signaling and the used technologies. In the end, a deployment over a physical testbed was explored, explaining the configuration of each network node and identifying each step of the demo.

CHAPTER 4

Extending SDN into Wireless Mobility Evaluation

In order to evaluate in this chapter the impact of OpenFlow in end-to-end flow mobility, the scenarios described in Fig. 4.2 (case A and case B) were implemented, analyzing the impact in terms of performance and signaling overhead, as well as the number of OpenFlow rules in the switch and/or controller.

The evaluation scenarios were deployed over the AMazING testbed [8] sited on the rooftop of the Instituto de Telecomunicações (IT) of Aveiro. The AMazING testbed is composed by 24 remotely managed wireless nodes. Each node is composed by a VIA Eden 1GHz processor with 1GB RAM and two wireless interfaces (an 802.11a/b/g/n Atheros 9K and a 802.11a/b/g Atheros 5K), running Ubuntu 12.04 LTS. Even though the experimental evaluation was done using two 802.11 APs, the framework design is generic enough to support different heterogeneous technologies.

In both study cases the MN is initially connected to both APs with the controller connection already established, while streaming, using the VLC application[1], the *Big Buck Bunny*[2] video (1080p) through AP1 towards the listener node. After 15 seconds, the signal quality at AP1 starts decreasing, and after 10 seconds of bad signal quality the MN notifies the OpenFlow Controller, which, in turn, initiates the procedures for optimizing the network. The second case (i.e., case B) is similar to the previous with the difference that it is the listener node that notifies the OpenFlow Controller when video quality decreases below a previously established threshold. The time values used in both cases are illustrative, with the framework being flexible enough to integrate mechanisms that allow dynamic configuration of different triggering opportunities. The experiments for evaluating the impact in terms of performance and signaling overhead, and the impact of the number of OpenFlow rules in the switch and/or controller, were run 10 times, showing here average results with 95% of confidence.

[1] `http://www.videolan.org/vlc/`

[2] `https://www.youtube.com/watch?v=XSGBVzeBUbk`

4.1 EVALUATION AND RESULTS

The experience controllability was taken through remote control using Secure Shell (i.e., SSH). Initiating text-based shell sessions on remote machines, SSH provides to the user a remote command-line login and a remote command execution on a machine without being physically present near it. With all the experiment being controlled remotely, the data results were captured and saved in the proper nodes using the paket analyzer tcpdump[3]. The tcpdump software allows to save (or display) the packets being transmitted or received over a network in the respective machine. After, the packet captures were analyzed in Wireshark[4]. Being similar to tcpdump, Wireshark adds a graphical front-end, and integrated sorting and filtering options.

In order to evaluate the impact of OpenFlow in end-to-end flow mobility, the impact in terms of performance and signaling overhead was analyzed, as well as the number of OpenFlow rules in the switch and/or controller, with the experience being run 10 times. For the data treatment and manipulation, MATLAB[5] was used, showing here average results with 95% of confidence.

4.1.1 PERFORMANCE

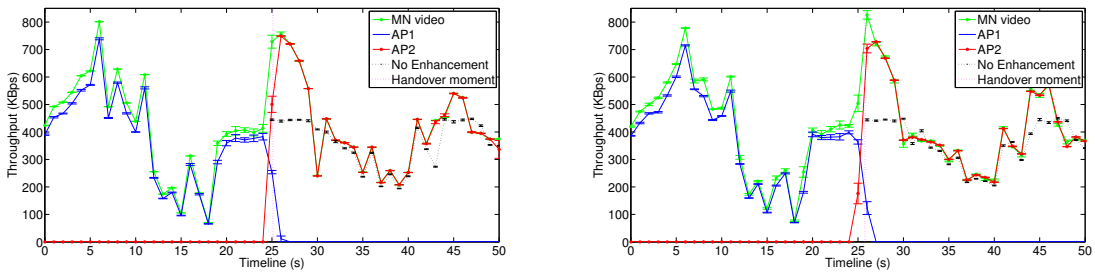
To evaluate the impact caused by the framework in terms of performance, the throughput of the RTP flow in the last node (Router, in Fig. 3.4) before the receiver was registered. Obtained values are shown in Fig. 4.1 a), for the scenario when the trigger is sent by the Mobile Node and in Fig. 3.4 b), when the Listener Node sends the trigger. The green line represents the throughput of the video, while the blue and red lines represent the same value on AP1 and AP2, respectively. The dotted black line represents the RTP flow throughput without using the framework, and the dotted pink line represents the handover moment.

As can be observed in both scenarios, before the handover of the video stream flow, the link conditions of the MN could not accommodate the requirements of the video stream, resulting in packet loss before the handover. This issue was not observed when using our framework, where the handover allowed the flow to switch to a link with better bandwidth. It can be seen that, for both scenarios, the developed framework allows for a considerable gain in terms of RTP throughput, when compared with the regular approach.

In this mobility scenario, when the handover occurs, there are still pending packets to be sent in the queue of the network interface connected to the old point of attachment. As these packets may reach the listener node in the wrong order, their impact was also accounted, since this means as well that both interfaces send packets simultaneously. In the *Case A* scenario (Fig. 4.1a), the video stream was sent through both interfaces for 679.78 (± 31.58) ms, with 143 (± 3) packets sent through the interface connected to AP1. In the *Case B* scenario (i.e., when the handover trigger was sent by the listener node), Fig. 4.1b, it was verified that packets were sent through both interfaces during 689.33 (± 41.35) ms, resulting in about 141 (± 2) overlapped packets. This is related with the fact that this design employs a *make-before-break* approach.

Table 4.1 presents reported values on flow activation time and handover delay, which are directly correspondent with the times illustrated in the signaling flow diagram in Fig. 4.2. Concretely, *Flow*

[3]http://www.tcpdump.org/
[4]https://www.wireshark.org/
[5]http://www.mathworks.com/products/matlab/



(a) Handover trigger from the source (b) Handover trigger from the listener

Figure 4.1: RTP throughput

Activation Time matches the t_{1_mn} time for *ath1* and t_{4_mn} for *ath0*, for the MN, and stands for the time between sending the handover solicitation (i.e., *packet_in*, Fig. 4.2-2a/2b) and the arrive and implementation of the flow modification that enables the handover. The second flow activation time regards for the time between MN acknowledging the implementation of the first flow modification (i.e., *barrier_reply*, Fig. 4.2-5) and implementing the second handover rule (i.e., *flow_mod*, Fig. 4.2-6) *MN Barrier* matches the time $t_{2\_mn} + t_{3\_mn}$. In the MN case, *Handover Delay* indicates the time from the handover trigger (i.e., *Packet_in* message) to the flow being moved (t_{1-5_mn}), while in the Controller case it indicates the time between the reception of the *Packet_in* message and sending the second *Flow_mod* message (t_{1-4_ctl}). The difference registered in the flow activation time for both *Flow_mod* messages is caused by the event handler, triggered by the *Packet_in* or *Barrier_reply* messages (Fig. 4.2 - 2a/b and 5, respectively).

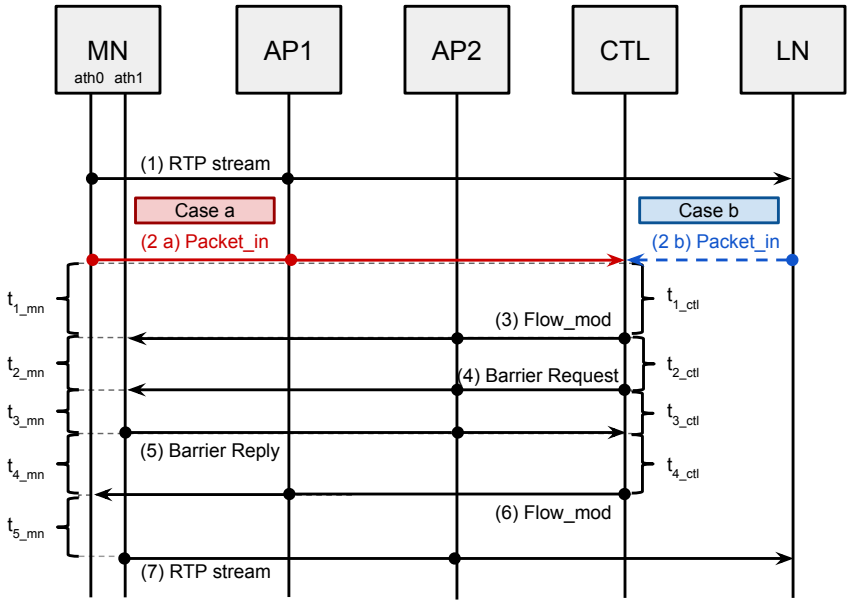


Figure 4.2: Signaling flow diagram with respective times

Results show that both cases present close performances, where, for *Case B*, the *Flow Activation Time* in *ath1* increased its time in 21.86%, the *MN Barrier* increased by 2.92%, and the *Handover Delay* experienced in the Controller and MN, increased by 18.92% and 18.24% respectively.

Table 4.1: Handover Data

	Flow Activation Time (ms)		MN Barrier (ms)	Handover Delay (ms)	
	ath0 (AP1)	ath1 (AP2)		Controller	Mobile Node
Case A	27.24± 8.41	142.56 ± 7.95	28.72 ± 5.96	78.57 ± 14.03	184.69 ± 13.71
Case B	27.10 ± 10.37	173.73 ± 23.46	29.56 ± 10.94	92.90 ± 14.94	219.63 ± 34.76

4.1.2 OVERHEAD

In order to evaluate the overall impact of OpenFlow traffic in the framework, the different types of traffic in the three experiments (i.e., trigger form the source or from the listener, and no enhancement) were registered in Table 4.2, showing its value at different network points. In Table 4.2 the *OF* stands for the OpenFlow traffic impact, while the *OF+TCP* is the same traffic, however accounting its synchronization impact. The *RTP* traffic is regarded to the video stream.

Starting with APs overhead, and despite that both AP1 and AP2 have the same transmission time (i.e., half of the experience time each), for both handover experiments (i.e., *source trigger* and *listener trigger*), AP2 is responsible for the major traffic transmission. In terms of listener node receive traffic, both handover cases have similar performance, allowing to consider that the SDN extension to the listener node has minor OpenFlow impact (0.01% of the total traffic). A comparison between

Table 4.2: Overhead table

			Controller		Access Points					Mobile Node		Listener Node	
					Total	AP 1		AP 2					
Source Trigger	Packets	OF	185.20 ± 3.96	59.54 ± 1.08%	15485.8 ± 94.9	47.60 ± 0.25%	0.61 ± 0.01%	52.40 ± 0.25%	0.56 ± 0.06%	16070.5 ± 76.9	0.30 ± 0%	15431.9 ± 167.9	-
		OF+TCP		98.39 ± 0.44%			1.03 ± 0.03%		0.88 ± 0.04%		0.51 ± 0.02%		-
		RTP		-			98.83 ± 0.03%		99.06 ± 0.04%		99.38 ± 0.02%		99.85 ± 0.09%
	MBytes	OF	0.01 ± 0	63.12 ± 0.97%	20.03 ± 0.12	47.55% ± 0.25%	0.035 ± 0%	52.75% ± 0.75%	0.030 ± 0%	20.87 ± 0.10	0.018 ± 0%	20.13 ± 0.20	-
		OF+TCP		98.87 ± 0.31%			0.06 ± 0%		0.05 ± 0%		0.03 ± 0%		-
		RTP		-			99.93 ± 0%		99.95 ± 0%		99.96 ± 0%		99.99 ± 0%
Listener Trigger	Packets	OF	222.8 ± 5.6	59.70 ± 1.16%	15446.4 ± 170.0	49.10 ± 0.47%	0.58 ± 0.01%	50.90 ± 0.47%	0.55 ± 0.01%	16073.1 ± 80.3	0.29 ± 0.01%	15412.3 ± 62.8	0.14 ± 0%
		OF+TCP		98.74 ± 0.28%			0.99 ± 0.03%		0.91 ± 0.02%		0.52 ± 0.02%		0.22 ± 0.01%
		RTP		-			98.95 ± 0.21%		99.02 ± 0.02%		99.35 ± 0.03%		99.72 ± 0.06%
	MBytes	OF	0.02 ± 0	63.44 ± 1.07%	19.98 ± 0.22	49.06 ± 0.47%	0.03 ± 0%	50.94 ± 0.47%	0.03 ± 0%	20.87 ± 0.10	0.02 ± 0%	20.08 ± 0.08	0.01 ± 0%
		OF+TCP		99.11 ± 0.20%			0.05 ± 0%		0.05 ± 0%		0.03 ± 0%		0.01 ± 0%
		RTP		-			99.93 ± 0%		99.95 ± 0%		99.96 ± 0%		99.98 ± 0%
No Enhanc.	Packets	RTP	-							15083 ± 221	99.88 ± 0.01%	13951 ± 0	99.87 ± 0.01%
	MBytes									19.68 ± 0.29	99.99 ± 0%	18.21 ± 0.28	99.99 ± 0%

the handover cases and *no enhancement*, shows a considerable throughput gain, about 10%, in both handover cases, stating the performance improvement when the handover is performed.

A depthless study of OpenFlow overhead and its impact in the develop framework will be explored in the next section.

4.1.3 OPENFLOW OVERHEAD

Table 4.3 shows, for both scenarios, the OpenFlow overhead introduced by the handover signaling presented in Fig. 4.2 and the total OpenFlow signaling exchanged during the experiments, including not only the handover signaling but also the *keep-alive* messages exchanged between OpenFlow entities. For this study, it was not considered the overhead introduced by Layer 2, since the developed framework is intended to operate independently of the underlying access technology.

Table 4.3: OpenFlow Overhead

		Controller		Mobile Node		Listener Node	
		Total	Handover	Total	Handover	Total	Handover
Source Trigger	Packets	110 ± 2	5 ± 0	48 ± 0	5 ± 0	-	
	Bytes	6901 ± 126	609 ± 0	3166 ± 48	609 ± 0		
Listener Trigger	Packets	140 ± 2	5 ± 0	47 ± 1	4 ± 0	21 ± 0	1 ± 0
	Bytes	8285 ± 129	609 ± 0	3072 ± 42	472 ± 0	1335 ± 24	137 ± 0

In terms of overhead, the handover signaling for both cases corresponds to about 609 bytes, regarding to the five OpenFlow messages exchanged in Fig. 4.2. The OpenFlow Controller is involved in all five messages of the handover signaling due to its centralized operation, while the MN (and listener node) accounted for 100% (and 0%) or 80%(and 20%), in respect to *Case A* and *Case B*.

The total OpenFlow overhead increases significantly the *keep-alive* messages also considered, since, in the experiments, these messages were exchanged each 5 seconds by the OpenFlow enabled entity. This value is doubled in the case of the MN due to each bridge sending the keep-alive messages. Nevertheless, this overhead could be reduced by using a higher period for exchanging *keep-alive* messages with OpenFlow-enabled end-points.

4.1.4 OPENFLOW RULES IMPACT

Another study was the impact of the amount of OpenFlow rules introduced by the framework in the MN and OpenFlow Controller. Fig. 4.3 shows how the number of rules affect the handover delay. To simulate additional rules on the OpenFlow Controller side (blue line), an additional cycle where a different number of rules are verified, ensures that the worst case is always analyzed. In the case of the MN, additional rules were implemented on the OvS bridge with an higher priority than the rule that performed the handover trigger and the packets forwarding, allowing also an evaluation of the worst case, enabling a fair comparison with the previous case. In both cases, with the purpose of performing the handover rule (first case) or forwarding the packets (second case), the header of the *Packet_in* message (handover trigger) is parsed and the source and/or destination IP addresses are checked.

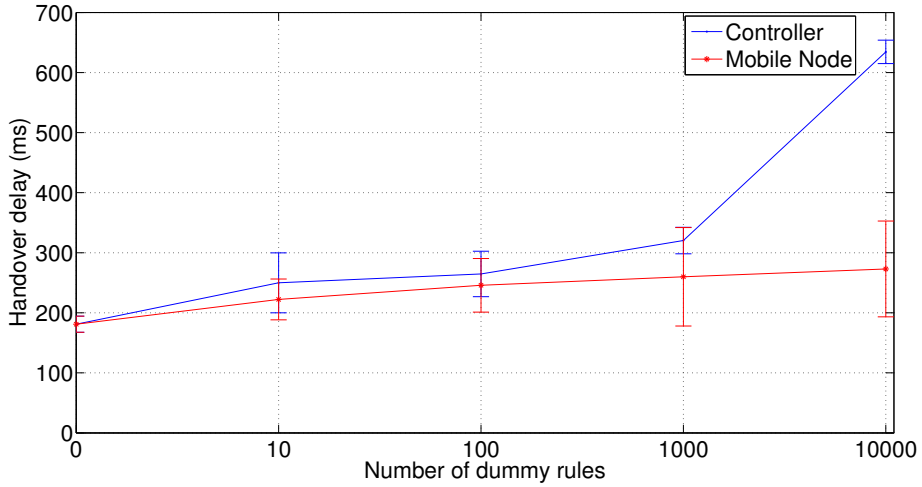


Figure 4.3: Dummy rules

It is visible from Fig. 4.3 that the increase in number of rules has higher impact on the flow activation time in the OpenFlow Controller than on the MN. Since the MN is using OvS (even though it is not specifically written for mobile terminals), it reflects the capability in terms of performance that such software must have in network forwarding equipment.

4.2 CHAPTER CONSIDERATIONS

In this chapter, the impact of OpenFlow over an heterogeneous mobility framework was studied, where SDN procedures were able to reach and control wireless mobile end-nodes. An implementation in a physical wireless testbed allowed for its evaluation, with results showcasing the feasibility and benefits of the proposed mechanisms towards a source mobility offloading scenario, without requiring additional mobility protocols.

CHAPTER 5

Multiple Flow in Extended SDN Wireless Mobility

The previous chapter deploys an experimental SDN architecture capable of performing a handover when triggered by an end-node, by extending SDN mechanisms all the way to the mobile node. Aiming to extend it and explore its behavior in a multiple flow scenario, the number of network devices was increased, and a traffic monitor, in the access point 2 (Fig. 5.1bd), is able to trigger the handover when a certain threshold is reached. In the deployed framework[1], the OpenFlow protocol reaches the end-nodes (i.e., the mobile nodes), enabling the network controller to provide mobility management via SDN mechanisms, by handovering flows from one wireless interface to the other, optimizing the MN's connectivity to the network. As in the previous study, the evaluation scenario was deployed over the physical wireless tested AMazING [2], sited in the rooftop of the Instituto de Telecomunicações of Aveiro. The scenario starts with both mobile nodes (MNs) initially connected to two different APs each, and with the controller connection already established. A third node, which only purpose is to generate traffic, is already attached to the AP2. Then, MN1 starts a new connection through AP2 towards the listener node (LN), increasing the AP's load. When MN2 starts its data stream, through the same AP, the AP2 overloads its bandwidth capacity and notifies the OpenFlow controller, that in its turn initiates the procedures for optimizing the network.

Further in this chapter, the architecture and involved signaling process will be explained and evaluated in an end-to-end multiple flow mobility scenario, presenting and discussing the results of its deployment over a physical wireless testbed.

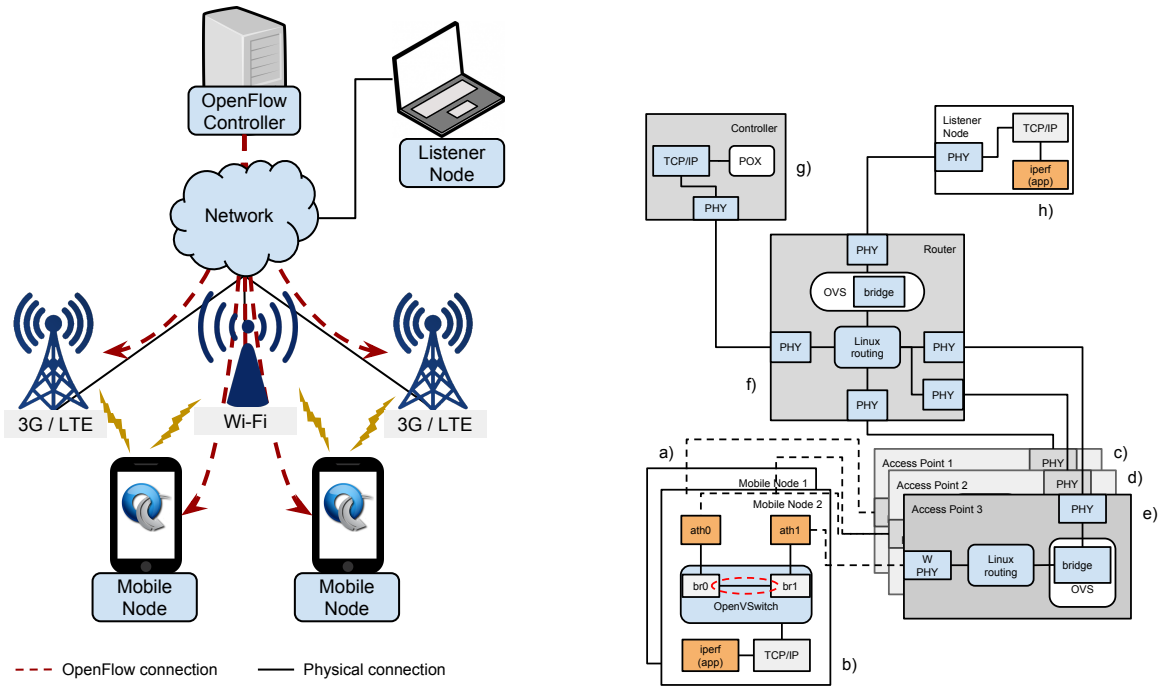
5.1 FRAMEWORK ARCHITECTURE

As in the previous scenario (i.e., Chapter 4), the end-to-end multiple flow mobility scenario, was deployed over the concept framework explored in Chapter 3, considering some required changes. Despite the number of network nodes involved, the main difference between the frameworks configuration

[1] Available as open-source: https://github.com/ATNoG/of mobilenode.git
[2] http://amazing.atnog.av.it.pt/

(Fig. 3.4 and Fig. 5.1b) is the change of the OvS bridge attached interface, that now is the physical network interface (PHY) (in the access points, Fig. 5.1b-c/d/e) instead of the wireless physical network interface (WPHY) (Fig. 3.4-b/c). Notwithstanding, the OvS configuration is maintained, which implements a layer 3 bridge in all network nodes, enabling a layer 3 routing using the kernel routing table. The OvS bridge attached interface adjustment was imposed by the necessity of the AP2 sending the handover solicitation through a *Packet_in* message to the controller. However, as in the previous scenario (i.e., Chapter 4), this configuration still allows to set a routing rule over the kernel routing table decision.

Fig. 5.1 compares the designed framework, where the OpenFlow controller is able to communicate and coordinate the mobile nodes through OpenFlow messages, with its configuration over the AMazING tested.



(a) Framework overview (b) Network modules

Figure 5.1: System framework

5.1.1 SIGNALING

The deployed multiple flow mobility scenario considers two dual-interfaced mobile nodes connected simultaneously to two different access networks each, while sharing one of them (Fig. 5.1a). Seen as source of content, each mobile node streams data packets towards a listener node through the same access point. This behavior increases the data load on the AP, which after reaching a certain threshold, sends a handover solicitation towards the controller. As in the previous scenario (Chapter 4), the handover solicitation is implemented through a *packet_in* message. The signaling processes involving the mobile node handover start with the *packet_in* arrival at the OpenFlow controller. Fig. 5.2 presents all the signaling involved in the handover process, explained as follows.

Initially, UDP streams from both MN1 and MN2 are sent towards the listener node through AP2 (Fig. 5.2-1/2), causing an overload in AP2. Motivated by the overloading, AP2 sends a handover solicitation (i.e., *packet_in* message) towards the network controller (Fig. 5.2-3), which after its reception, analyzes the *packet_in* header and sends to the MN1 a *Flow_mod* message and consecutively a *barrier_request* (Fig. 5.2-4/5, respectively), to verify the correct implementation of the first rule. Sent to the new transmission interface (i.e., *ath1*), the first *flow_mod* implements not only the forwarding of the received packets from the br0 (i.e., *ath0*), but also the actions "MOD_SRC_ADD" and "MOD_DST_ADD", which modify the source and destiny MAC addresses to accomplish the wireless handshake. When MN1 receives the *barrier_request*, a *barrier_reply* (Fig. 5.2-6) with the same *xid* is sent, indicating the installation of the first rule (implemented through the *flow_mod*). The controller, after receiving and verifying the *barrier_reply*, sends the second rule, through a *flow_mod* message (Fig. 5.2-7). This last rule redirects the packets to the new interface (i.e., *ath1*). The subsequent packets sent from MN1 to listener node IP, with the correspondent destiny port and protocol, are sent through AP1 (Fig. 5.2-7). The next time that AP2 overloads and sends a handover solicitation, the controller will process the handover of MN2, with a similar process.

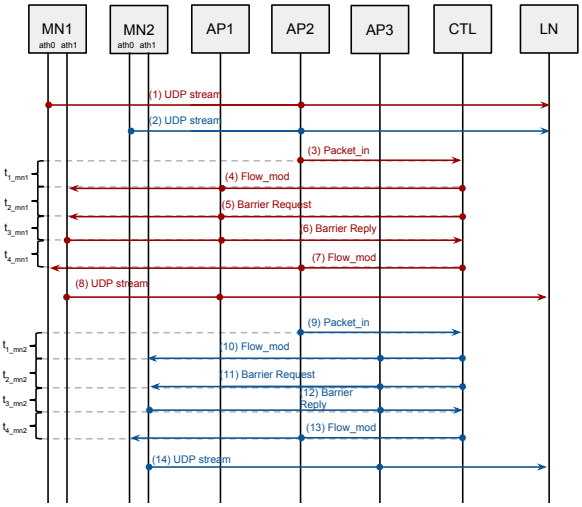


Figure 5.2: Network Signaling

5.1.1.1 OPENFLOW MESSAGES

The designed framework considers three types of OpenFlow messages signaling to support the handover, explained as follows.

- Packet_in (Fig. 5.2-3/9):
  In this scenario the packet_in is generated by an output action in a flow entry with IP address matching, by sending an UDP packet when AP2 reaches the load threshold;
- Flow_mod (Fig. 5.2-4/7/10/13):
  As in the previous scenario, the first "modify flow entry message" of each MN handover signaling (i.e., Fig. 5.2-4/10) adds in both MN1's and MN2's OvS bridge, respectively, a flow entry to redirects the matching traffic (i.e., the video stream) to the patch port. Regarding to the flow entry motivated by the second flow_mod of each MN handover signaling (i.e., Fig. 5.2-7/13, respectively), it redirects the matching traffic (i.e., the traffic coming from the patch port) after modifying the source and destiny MAC addresses;
- Barrier_request (Fig. 5.2-5/11) and Barrier_reply (Fig. 5.2-6/12):
  In the developed framework, these messages are used to ensure the installation of the first flow_mod before sending the second one, in each MN handover signaling.

5.1.2 HANDOVER TRIGGERING

Extending SDN up to the end-nodes, allows mobility management though OpenFlow messages without requiring additional mobility protocols. In this scenario, the handover is solicited by the access point, which sends a *packet_in* message to the controller, when its load reaches a pre-established threshold. In oder to automatize the handover solicitation, a bash script that monitors the received traffic in the access point (AP) was developed. The main reason for the bash as the language of the script was for the simplicity of getting the necessary information. While C requires libraries to access the physical network interface (PHY) information (e.g., libpcap[3]), in bash a simple command call (i.e., ifconfig) and a parsing command (i.e. grep) is sufficient. Getting these commands together allows the needed information (i.e. the amount of received packets) to be captured. The final command should be:

```
read RX_B <<< $(ifconfig $IFACE | grep 'RX bytes'| awk '{ print $2}' | grep -o
    [0-9] | tr -d '\n')
```

As mentioned before, in order to trigger the *packet_in* message, the access point is able to monitor the bitrate in its physical wireless interface. This value is obtained through a configurable pre-established time period, enabling the bitrate calculation through equation 5.1, where Rx_{new} stands for the received bits after t seconds of the Rx_{old} value.

$$bitrate = \frac{Rx_{new} - Rx_{old}}{t} \tag{5.1}$$

This monitor feature stores the bitrate values and triggers the *packet_in* message when the average measure of the last three bitrate values are above of the threshold (i.e., 90% maximum load) for three consecutive times. This allows to avoid *false positive* or even *ping-pong* affects. With these requirements it is possible to find a maximum time value for the handover solicitation to be sent, after the threshold is reached. Equation 5.2 shows how this value can be calculated, where t stands for the period between measures. In order to study the impact of the reading frequency over the framework performance, this value was ranged from 0.5 to 2 seconds.

$$t_{hand.sol.} \leq 5 \times t \tag{5.2}$$

5.2 FRAMEWORK DEPLOYMENT

Exploring the capability of using the OpenFlow protocol in different types of network nodes and studying the possibility of providing a handover through SDN mechanisms, the scenario was deployed in a tree network topology with 9 nodes (i.e., 3 access points, router, network controller, listener node, 2 mobile node and a wireless node), where the mobile nodes are seen as a source of content, emulating a voice streaming through the network towards the listener node. Keeping that in mind, the mobile nodes have a wireless connection with two APs each (i.e., MN1 attach to AP1 and AP2, and MN2 attach to AP2 and AP3, Fig. 5.3). On the other hand, the third wireless node is attached only to AP2, whereas all the remaining nodes have a physical link connection. All nodes can ping each other.

[3]http://www.tcpdump.org/

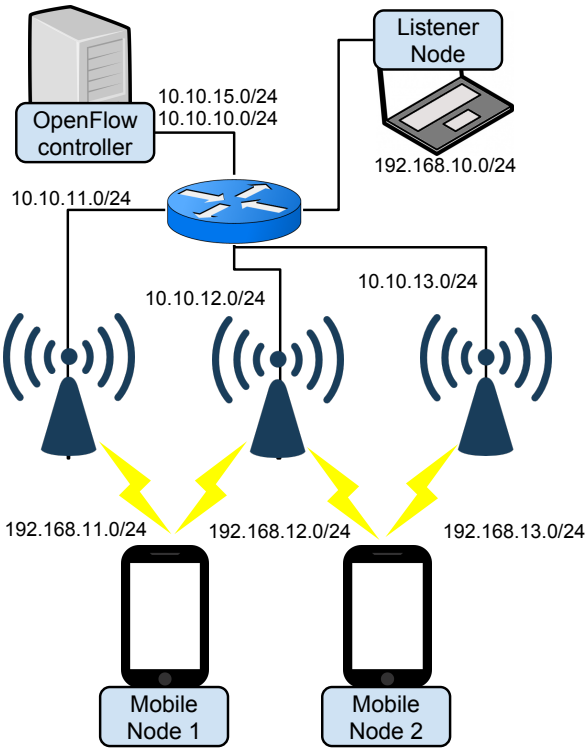


Figure 5.3: Framework configuration

With the purpose of extending the OpenFlow control path up to the Mobile Nodes (Fig. 3.4a/b), the OvS v2.3.1 with its dependencies was installed in every network node, except on the controller node, listener and wireless node. While the listener and wireless node does not have any OpenFlow agent, the POX was used in the controller node (Fig. 3.4g), enabling it to perform a connection between any node of the framework with an OpenFlow agent. With the mobile node being the source content, its OvS configuration is somehow different from the remaining network nodes, enabling it to apply new rules over the kernels forwarding, allowing the packet redirection to the other wireless interface. To accomplish that, both OvS bridges attached to each mobile nodes's wireless interface, were interconnected by a patch port that acts as a patch cable.

As explained in Chapter 3, an OpenFlow switch is not capable of handling ARP messages, so an external entity to handle them (eg., the controller) is required. As in the previous scenario, the nodes' OvS configuration gives the task to the Linux kernel, enabling each network node to handle its own ARP messages. Moreover, the data and control path share the same link, whereby instead of using the usual OvS bridge per network interface, we used just one OvS bridge in each network node (Fig. 3.4). With this configuration, less control traffic is generated, and still allows us to set a new forwarding rule over the kernel's routing table. In this way, the OvS bridge in each network node was configured as layer 3, enabling each network interface to handle its own ARP messages.

Finally, to enable the MNs to establish a wireless connection and perform the wireless handshaking a prior setup of a "NORMAL" action is also required. This "NORMAL" action allows the OvS to send the packet to the kernel, to be processed as layer 2 or layer 3. With the purpose of making the OvS responsible for the packets forwarding in case of controller fail, the OvS bridges were configured in fail-save-mode "standalone".

5.2.1 RUNNING A DEMO

Before starting to run the demo, and since the control and data path are in the same interface, the mobile nodes (MNs) and access points (APs) require not only a "NORMAL" action and rule which forwards the handover solicitation as a *packet_in* message, but also an already established link with APs and MNs. Contrary to the previous scenario, the handover solicitation (i.e., *packet_in* messages) will be sent by the AP2, so it requires an OvS forwarding rule that sends the handover solicitation through a *packet_in* messages towards the controller. Moreover, the static rules that perform the forwarding between the network nodes, need to be implemented in the kernel's routing table. Below, it is shown the experience running process:

1. Iperf stream from wireless node starts:
   iperf -c 10.10.12.101 -u -b 8m -t 120
   IP dst = 10.10.12.101, User Datagram Protocol (UDP), port = 5001;
   The wireless node sends the udp packets towards AP2 through at 8 Mbps;

2. After 5 seconds MN1 starts its own iperf stream:
   iperf -c 192.168.10.1 -u -b 8m -t 100
   IP dst = 192.168.10.1, User Datagram Protocol (UDP), port = 5001;
   The MN1 sends the udp packets towards LN through interface attached to AP2 (i.e., ath0) at 8 Mbps;

3. After 15 seconds MN2 starts its own iperf stream:
   iperf -c 192.168.10.1 -u -b 8m -t 100
   IP dst = 192.168.10.1, User Datagram Protocol (UDP), port = 5001;
   The MN2 sends the udp packets towards LN through interface attached to AP2 (i.e., ath0) at 8 Mbps;

4. The load of AP2 reaches the threshold (i.e., 18 Mbps), forcing the AP2 to send the handover solicitation:
   AP2 sends *packet_in* - UDP packet sent to the IP addess 11.11.11.11;
   The OvS of AP2 has implemented a forwarding rules that sends to the Controller the packets with *destination IP 11.11.11.11*, through a *packet_in* message;

5. Controller receives and analyzes the *packet_in* message;

6. Controller performs the MN1 handover:
   It sends the first *flow_mod* message, and promptly sends the *barrier_request* message;
   MN1 implements the flow modification and responds the *barrier_request* with a *barrier_reply* message;
   Controller *barrier_reply* message, analyzes it and sends a second *flow_mod* message;
   MN1 receives the second *flow_mod* message and starts redirecting flows.

7. With the handover of MN1 to the AP1, the load of AP2 decreases below to the threshold;

8. After 10 seconds, the wireless node requires more bandwidth, by adding a new iperf stream:
   iperf -c 10.10.12.101 -u -b 8m -t 100
   IP dst = 10.10.12.101, User Datagram Protocol (UDP), port = 5001;
   The wireless node sends the udp packets towards AP2 at 8 Mbps;

9. This new stream causes a load increase at AP2 and reaches the threshold (i.e., 18 Mbps), forcing the AP2 to send the a second handover solicitation:
   AP2 sends *packet_in* - UDP packet sent to the IP addess 11.11.11.11;

10. Controller receives and analyzes the *packet_in* message;

11. Controller performs the MN2 handover to AP3 with a similar process to MN1:

12. The handover of MN2 to the AP3, causes load decrease at AP2, and stays below the threshold till the end;

The time values used in both cases are illustrative, with the framework being flexible enough to integrate mechanisms that allow dynamic configuration of different triggering opportunities.

5.3 EVALUATION AND RESULTS

As in the previous scenario, the experience control was taken remotely using Secure Shell (i.e., SSH). The data results were captured and saved in the proper nodes using the packet analyzer tcpdump[4] and analyzed in Wireshark[5].

In order to evaluate the impact of the framework mobility management capabilities in a wireless environment containing multiples flows, the scenario illustrated in Fig. 5.1b was deployed over the AMazING[6] testbed. Despite the fact that the evaluation testbed is composed of Wi-Fi nodes, the framework is flexible enough to be deployed in mobile network environments, with the necessary network end-points adaptations.

In the evaluated scenario, the mobile nodes, besides being content consumers, are also a source of content, streaming data (eg., using *iperf*). Enabling OpenFlow in the mobile nodes allowed the communication between the OpenFlow controller and mobile nodes, giving to the controller the possibility to monitor the network state and impact the end-nodes connectivity to the network.

The scenario starts with AP2 having already a load of baseline 8Mbps UDP traffic (Fig. 5.4a, blue line), and the MNs connections to APs and controller already established. After the first 5 seconds, the MN1 starts sending 8Mbps of UDP traffic through AP2 towards LN, increasing the AP2 load. To originate these UDP streams, the *iperf* tool was used, configuring the desired bandwidth (-b 8m) and protocol (-u or --udp). When the MN2 starts streaming (at 20 seconds of experience time), also towards LN and through AP2, the AP2 starts overloading, sending a handover solicitation to the controller. Upon its reception, the controller decides to perform the handover of that flow to AP1 (Fig. 5.4a, red line). At about 35 seconds of experience, the initial UDP stream requires more bandwidth, simulating dynamic quality requirements present in a real application, overloading the AP2, that sends another handover solicitation to the controller, this time performing the handover of MN2 to AP3 (Fig. 5.4a black line).

The experiments to evaluate the performance of the multiple flow mobility were run 5 times, for different load reading time values, showing here the average results with a 95% of interval confidence.

[4]http://www.tcpdump.org/
[5]https://www.wireshark.org/
[6]http://amazing.atnog.av.it.pt/

For the data treatment and manipulation, MATLAB[7] was used, showing here average results with 95% of confidence.

5.3.1 PERFORMANCE

To evaluate the performance, the average throughput at several network points was considered. Fig. 5.4 shows these throughputs, where the performance loss before the handover in both MN1 and MN2 can be noted. A throughput loss is visible in MN1 when MN2 starts streaming, increasing again after performing the handover (Fig. 5.4b/c). This effect is especially noticeable in MN2 (Fig. 5.4c), which initially cannot transmit at its maximum throughput, and immediately after the MN1 handover, its throughput grows up to 8Mbps. When AP2 starts overloading at 35 seconds, the throughput in MN2 (Fig. 5.4c) decreases, which grows back to its original value after its handover to AP3.

The throughput difference between MN1 and MN2 before each handover (i.e., 7.5Mbps for MN1 and 4Mbps for MN2), is attributed to the wireless interferences and signal strength between each MN and AP2, with the MN2 having a weaker signal (physically farther from the AP).

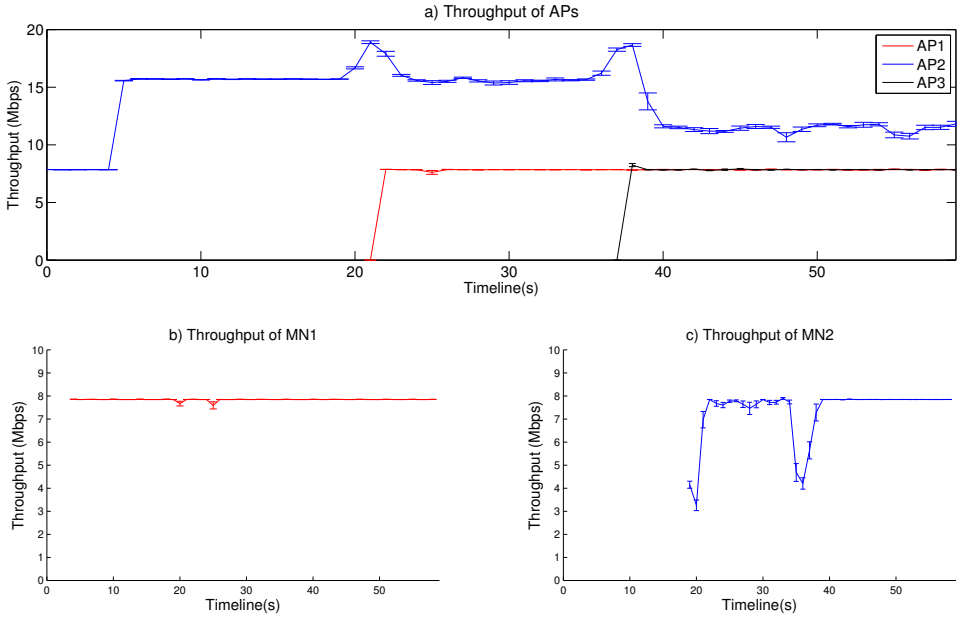


Figure 5.4: UDP throughputs at different network points

5.3.2 HANDOVER DETECTION

In the experiment the handover detection performance was also evaluated to different time interval values. Equation 5.3 enables the calculation of the number of required CPU readings in the AP to

[7]http://www.mathworks.com/products/matlab/

detect the overload and trigger the handover solicitation.

$$N_{reads} = \frac{t_{hand.sol.}}{t} \tag{5.3}$$

With the purpose of evaluating the impact of the time interval load readings, several experiments were made ranging the interval from 500ms to 2s. The choice of these values relies on the fact that values below 500ms have a significant error when extrapolated to bit per second (i.e., bps), and values greater than 2s have a considerable increase of overload detection time. Fig. 5.5 shows the throughputs of AP2 for the different time intervals.

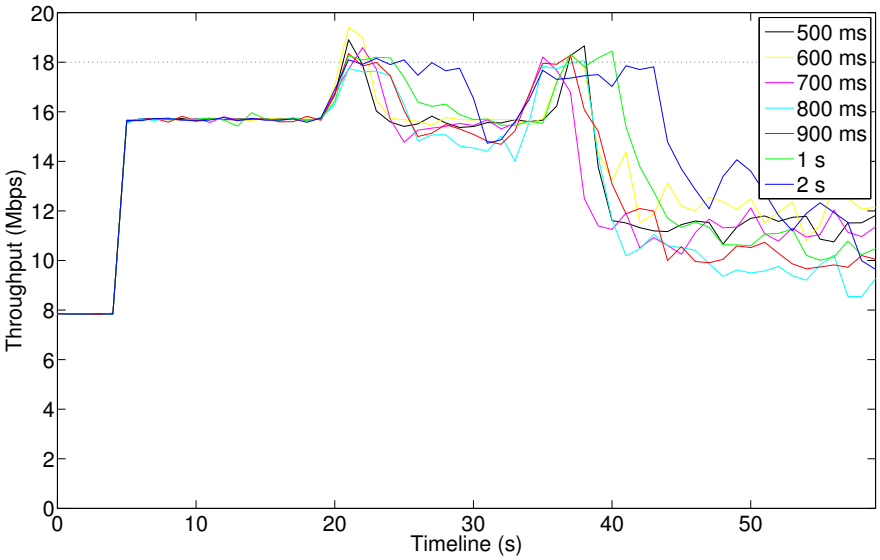


Figure 5.5: UDP throughputs of AP2 for different time interval values

In TABLE 5.1, the times of the occurrence of each handover are shown. As expected, for the first handover (i.e., *handover 1*, MN1 handover), its moment delays with the time interval increment. On the other hand, this is not always true, and if we observe the second handover moment (i.e., *handover 2*, MN2 handover), the first handover to happen was for the 700ms interval, since it is also the first to start the AP overload. Notwithstanding, looking at the handover time detection, in TABLE 5.1 we note that the quickest was the 500ms interval.

Despite that the shortest interval is the one that gives the faster handover detection, this also implies more CPU load. For example, if the experience runs over 60s and the reading time period was set to 500ms, the CPU will read the received bytes at least 120 times (equation 5.3). On the other hand, if the time period was set to 2s, the CPU will read the same value only 30 times. This behavior, translates into a delay when concerning the overload detection.

Regarding to the handover delay, it remains constant with the 14.18 ($\pm$ 1.89) ms for MN1 and 362.18 ($\pm$ 48.41) ms for MN2. These values directly correspond with the times in the signaling flow diagram (Fig. 5.2), with the MN1 time being the sum of t_{2_mn1} and t_{4_mn1}. The t_{2_mn2} and t_{4_mn2} correspond to MN2.

Table 5.1: Handover Timeline

Time interval	Timeline (s)		Handover triggering $t_{hand.sol}$ (s)
	Handover 1	Handover 2	
500 ms	22.68 ± 0.26	39.00 ± 0.50	2.01 ± 0.13
600 ms	22.92 ± 0.13	39.35 ± 1.11	2.34 ± 0.18
700 ms	24.02 ± 0.41	37.54 ± 0.71	3.08 ± 0.29
800 ms	24.97 ± 0.60	38.97 ± 0.37	4.25 ± 0.26
900 ms	25.03 ± 0.52	39.64 ± 1.85	4.33 ± 0.41
1 s	26.22 ± 1.56	42.13 ± 1.44	5.03 ± 1.11
2 s	30.41 ± 0.79	48.35 ± 5.00	9.60 ± 0.43

5.3.3 OPENFLOW IMPACT

The handover OpenFlow signaling exchanged between controller and MN, showed an overhead of 609 bytes, regarding to the five OpenFlow messages exchanged. The OpenFlow overhead introduced by the handover signaling presented in Fig. 5.2 in each MN was 80% of this value (i.e., four OpenFlow messages), with the remaining 20% (first message, i.e., *packet_in*) being ensured by AP2. Since the OpenFlow Controller has a centralized operation, it is involved in all five messages of the handover signaling. As a framework that aims to operate independently of the underlying access technology, in this study we do not consider the overhead introduced by Layer 2.

The total OpenFlow overhead is significantly increased by the keep-alive messages (exchanged between the controller and OvS by default), since these messages were exchanged each 5 seconds by the OpenFlow enabled entities. In a 60 seconds experiment, 24 (± 1) OpenFlow keep-alive messages are exchanged. This value is duplicated in the case of the MN due to each bridge sending the keep-alive messages, causing a total (i.e, keep-alive and handover signaling messages) of 3483 (± 151) bytes OpenFlow overhead for each MN. Nevertheless, this overhead could be reduced by using a higher period for exchanging keep-alive messages with OpenFlow-enabled end-points.

5.4 CHAPTER CONSIDERATIONS

In this chapter, it was studied the impact of applying SDN principles all the way to the end-nodes, allowing the OpenFlow to be used as a flow-based mobility management in an heterogeneous mobility framework. An implementation in a physical wireless testbed allowed for its evaluation, with results showcasing a handover performance without packet loss, with the overhead of the proposed mechanisms being minimal. The involved software configurations were made available as open-source.

CHAPTER 6

Conclusion

Along this thesis, as a main goal and aiming to improve 5G network research, an heterogeneous mobility framework, where SDN mechanisms were able to reach and control wireless mobile end-nodes, was developed, exploring OpenFlow and associated SDN mechanisms. The developed framework addresses future challenges such as massive traffic volumes, the proliferation of connected mobile devices and sustainable integration of heterogeneous networks in mobile environments. Thus, SDN has been one of the key building blocks of 5G network architectures, adding a greater degree of flexibility.

In chapter 2 an exhaustive exploration of the SDN architecture and its mechanisms shows some SDN advantages, such as the complexity reduce in today's networks, enabling network administrators to manage network services from a central management tool by virtualizing physical network connectivity into logical network connectivity. Moreover, SDN adoption provides to a company the ability to model its physical networking environment into software, reducing the overall CAPEX and OPEX. Also, in physical environments a modification at a physical networking device, would often take a considerable amount of time to substitute and (re)configure the equipment, while in a SDN environment the ability to control the virtual and physical networking is provided by using a central management tool. This ability not only, improves the packet forwarding, but also facilitates the network equipment update. Additionally, SDN gives extensibility with management APIs to allow vendors to extend the capabilities of an SDN solution by developing applications to simplify the control of networking traffic behavior. This same chapter also explored some SDN-enabled software that could be used to deploy mobility framework in an OpenFlow environment, where SDN/OpenFlow controller is simplified as an operating system for the network, with the OpenFlow protocol connecting the controller software to the network devices, allowing the controller to configure network devices.

Despite the possibilities allowed by the coupling of Software Defined Networking (SDN) mechanisms in upcoming network control and management frameworks towards the enablement of 5G network architectures, there is yet little experimental work exploiting the application of these aspects into end-nodes in mobile wireless environments. Towards this concern, chapter 3 addressed the main objective, exploring a whole new framework composed by a modified OpenFlow controller and several network nodes instancing OvS, with the purpose of enabling the mobile nodes handovering.

In chapter 4 and chapter 5 two evaluation scenarios implemented in a physical wireless testbed were considered. A first scenario which explores an extension of SDN mechanisms in heterogeneous networks, by enabling the use of the OpenFlow protocol in the mobile end-node, shows its feasibility with minor

OpenFlow overhead impact. A second scenario where the OpenFlow was used as a flow-based mobility management in mobile wireless environments, provided results showcasing not only the feasibility and benefits of the proposed mechanisms towards a source mobility offloading scenario, without requiring additional mobility protocols, but also the benefits of extending SDN approaches for end-to-end flow control in wireless environments.

To conclude, this thesis contributed towards upcoming 5G effort by studying the impact of applying SDN principles all the way to the end-nodes, allowing the OpenFlow to be used as a flow-based mobility management in mobile wireless environments. Both scenarios and respective evaluation were validated and submitted as a paper to IEEE GLOBECOM 2015 and to European Workshop on Software Defined Networks. Additionally, a contribution on Github[1] was made, distributing and explaining the framework entities.

6.1 FUTURE WORK

As it was previously referred, there are several issues that still exist and need to be overcome to make OpenFlow a reality in Mobile Nodes. Some of them will be mentioned below:

- Handover time improvements: Although the handover latency results are acceptable, it is possible to improve it by developing a proper OpenFlow switch to be used in Mobile Nodes, where the nodes instead of being a network forwarding equipment are not only a data consumer, but also a source of content.;
- OpenFlow Controller: the framework controller can also be improved, by developing a script capable of discovering the best route from the source to the destiny;
- Improving the cooperation between the MN and the controller;
- Machine-to-Machine scenarios: to evaluate the application of this concept applied to different kinds of mobile devices.

[1] http://atnog.github.io/of_mobilenode/

References

[1] B. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks", *Communications Surveys Tutorials, IEEE*, vol. 16, no. 3, pp. 1617–1634, Third 2014.

[2] C. W. Paper, "Cisco visual networking index: Global mobile data traffic forecast update, 2014-2019", Feb. 2015. [Online]. Available: http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.pdf.

[3] 5.-P. W. Paper, "Future networks and management", Jun. 2011. [Online]. Available: http://www.networks-etp.eu/fileadmin/user_upload/Publications/Position_White_Papers/White_Paper_Future_Network_Management.pdf.

[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks", *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008, ISSN: 0146-4833.

[5] R. Jain and S. Paul, "Network virtualization and software defined networking for cloud computing: A survey", *IEEE Communications Magazine*, vol. 51, no. 11, pp. 24–31, 2013.

[6] W. H. Chin, Z. Fan, and R. Haines, "Emerging technologies and research challenges for 5g wireless networks", *Wireless Communications, IEEE*, vol. 21, no. 2, pp. 106–112, Apr. 2014, ISSN: 1536-1284. DOI: 10.1109/mwc.2014.6812298.

[7] O. N. Foundation, *SDN architecture [online]*, 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf.

[8] J. Barraca, D. Gomes, and R. L. Aguiar, "Amazing – advanced mobile wireless playground", English, in *Testbeds and Research Infrastructures. Development of Networks and Communities*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, T. Magedanz, A. Gavras, N. Thanh, and J. Chase, Eds., vol. 46, Springer Berlin Heidelberg, 2011, pp. 219–230, ISBN: 978-3-642-17850-4.

[9] T. Nadeau and K. Gray, *Software Defined Networks*, 1st ed. O'Reilly Media, Inc., Aug. 2013, ISBN: 978-1-449-34230-2.

[10] O. N. F. Documentation, "Openflow switch specification", Jun. 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf.

[11] O. N. F. W. M̃. W. Group, *WMWG Charter Application [online]*, 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/working-groups/charter-wireless-mobile.pdf.

[12] K. Pentikousis, Y. Wang, and W. Hu, "Mobileflow: Toward software-defined mobile networks", *Communications Magazine, IEEE*, vol. 51, no. 7, pp. 44–53, Jul. 2013.

[13] X. Jin, L. E. Li, L. Vanbever, and J. Rexford, "Softcell: Scalable and flexible cellular core network architecture", in *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '13, Santa Barbara, California, USA: ACM, 2013, pp. 163–174, ISBN: 978-1-4503-2101-3.

[14] Y. Yiakoumis, J. Schulz-Zander, and J. Zhu, *Pantou : OpenFlow 1.0 for OpenWRT*, 2011. [Online]. Available: `http://www.openflow.org/wk/index.php/OpenFlow_1.0_for_OpenWRT`.

[15] C. Guimaraes, D. Corujo, R. Aguiar, F. Silva, and P. Frosi, "Empowering software defined wireless networks through media independent handover management", in *Global Communications Conference (GLOBECOM), 2013 IEEE*, Dec. 2013, pp. 2204–2209.

[16] P. Dely, A. Kassler, and N. Bayer, "Openflow for wireless mesh networks", in *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, Jul. 2011, pp. 1–6.

[17] "Ieee standard for local and metropolitan area networks - media independent handover services", *IEEE Std 802.21-2008*, pp. 1–, Jan. 2009.

[18] P. Congdon, L. Contreras, S. Manning, R. Marks, A. de la Oliva, and J. C. Zuniga, *Ieee 802 tutorial wireless sdn in access and backhaul*, Nov. 2011. [Online]. Available: `https://mentor.ieee.org/802-ec/dcn/13/ec-13-0055.pdf`.

[19] K.-K. Yap, M. Kobayashi, R. Sherwood, T.-Y. Huang, M. Chan, N. Handigol, and N. McKeown, "Openroads: Empowering research in mobile networks", *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 1, Jan. 2010.

[20] K.-K. Yap, R. Sherwood, M. Kobayashi, T.-Y. Huang, M. Chan, N. Handigol, N. McKeown, and G. Parulkar, "Blueprint for introducing innovation into wireless mobile networks", in *Proceedings of the Second ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures*, ser. VISA '10, New Delhi, India: ACM, 2010, pp. 25–32, ISBN: 978-1-4503-0199-2.

[21] M. Bansal, J. Mehlman, S. Katti, and P. Levis, "Openradio: A programmable wireless dataplane", in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12, Helsinki, Finland: ACM, 2012, pp. 109–114, ISBN: 978-1-4503-1477-0.

[22] A. Gudipati, D. Perry, L. E. Li, and S. Katti, "Softran: Software defined radio access network", in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13, Hong Kong, China: ACM, 2013, pp. 25–30, ISBN: 978-1-4503-2178-5.

[23] S. Kumar, D. Cifuentes, S. Gollakota, and D. Katabi, "Bringing cross-layer mimo to today's wireless lans", *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 387–398, Aug. 2013, ISSN: 0146-4833.

[24] C. Bernardos, A. De La Oliva, P. Serrano, A. Banchs, L. Contreras, H. Jin, and J. Zúniga, "An architecture for software defined wireless networking", *Wireless Communications, IEEE*, vol. 21, no. 3, pp. 52–61, Jun. 2014.

[25] V. Yazıcı, U. Kozat, and M. Oguz Sunay, "A new control plane for 5g network architecture with a case study on unified handoff, mobility, and routing management", *Communications Magazine, IEEE*, vol. 52, no. 11, pp. 76–85, Nov. 2014.

[26] J. Lee, M. Uddin, J. Tourrilhes, S. Sen, S. Banerjee, M. Arndt, K.-H. Kim, and T. Nadeem, "Mesdn: Mobile extension of sdn", in *Proceedings of the Fifth International Workshop on Mobile Cloud Computing & Services*, ser. MCS '14, Bretton Woods, New Hampshire, USA: ACM, 2014, pp. 7–14, ISBN: 978-1-4503-2824-1.

[27] C. Guimaraes, D. Corujo, and R. Aguiar, "Enhancing openflow with media independent management capabilities", in *Communications (ICC), 2014 IEEE International Conference on*, Jun. 2014, pp. 2995–3000.

[28] A. Rodriguez-Natal, V. Ermagan, F. Maino, and A. Cabellos, "An over-the-top sdn architecture for mobile nodes and home routers", 2015. [Online]. Available: https://www.ietf.org/proceedings/92/slides/slides-92-sdnrg-1.pdf.

[29] A. Oliva, J. Zúñiga, and P. Congdon, "Omniran sdn use cases", 2015. [Online]. Available: https://mentor.ieee.org/omniran/dcn/14/omniran-14-0029-00-0000-sdn-based-use-cases-for-bof.pptx.

[30] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: An intellectual history of programmable networks", *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, Apr. 2014, ISSN: 0146-4833. DOI: 10.1145/2602204.2602219. [Online]. Available: http://doi.acm.org/10.1145/2602204.2602219.

[31] R. Riggio, T. Rasheed, and F. Granelli, "Empower: A testbed for network function virtualization research and experimentation", in *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*, Nov. 2013, pp. 1–5. DOI: 10.1109/SDN4FNS.2013.6702538.

[32] T. Adame, A. Bel, B. Bellalta, J. Barcelo, and M. Oliver, "Ieee 802.11ah: The wifi approach for m2m communications", *Wireless Communications, IEEE*, vol. 21, no. 6, pp. 144–152, Dec. 2014, ISSN: 1536-1284. DOI: 10.1109/MWC.2014.7000982.

[33] E. Haleplidis, K. Pentikousis, S. Denazis, J. Hadi, D. Meyer, and O. Koufopavlou, "Software-defined networking (sdn): Layers and architecture terminology", Jan. 2015. DOI: 10.17487/RFC7426.

APPENDIX A

Network Nodes Configuration

In this appendix some of the configuration OvS scripts will be presented, showing the required bash commands to start OvS module and configure the OvS bridges of the framework network nodes.

Starting with the mobile node, section A.1 presents a script which can be extrapolated to any dual-interfaced node, in order to enable the traffic redirection from one interface to another. Then a section A.2 with a configuration example of the framework router, exemplifies the remaining wired network nodes, showing the used commands to create a VLAN and how to start the OvS module and configure the OvS bridge. Regarded to the access points, section A.3 shows the configuration of the hostapd software. Finally, the required command to run the POX controller with the necessary scripts is presented in section A.4.

A.1 MOBILE NODE

In this section the mobile node configuration script is presented in Listing 1, showing not only how to start the OvS module and attach ports to the OvS bridge, but also how a patch port can be created and a connection mode "out-of-band" with the controller can be established. To attach the mobile node to the remaining network, the command line *iwconfig ath1 sdn-ap1* was used, where *ath1* and *sdn-ap1* stands for the MN's wireless interface and AP's Extended Service Set Identification (ESSID)[1] to attached, respectively.

[1]Commonly known as "network name"

```
# OvS MobileNode Configuration
## start OVS
echo "load ovs kernel module"
modprobe openvswitch

lsmod | grep openvswitch

ovsdb-server /usr/local/etc/openvswitch/conf.db \
--remote=punix:/usr/local/var/run/openvswitch/db.sock \
--remote=db:Open_vSwitch,Open_vSwitch,manager_options \
--private-key=db:Open_vSwitch,SSL,private_key \
--certificate=db:Open_vSwitch,SSL,certificate \
--bootstrap-ca-cert=db:Open_vSwitch,SSL,ca_cert --pidfile --detach --log-file

ovs-vsctl --no-wait init
ovs-vswitchd --pidfile --detach

# add bridge to ethernet interfaces
ovs-vsctl add-br br0
ovs-vsctl add-br br1

# attach interfaces to bridges
ovs-vsctl add-port br0 ath0
ovs-vsctl add-port br1 ath1

# create a patch port for each bridge
ovs-vsctl add-port br0 patch01 -- set interface patch01 type=patch
    options:peer=patch10
ovs-vsctl add-port br1 patch10 -- set interface patch10 type=patch
    options:peer=patch01

# connect OVS to a controller - "out-of-band"
ovs-vsctl set-controller br0 tcp:10.10.15.201:6633
ovs-vsctl set-controller br1 tcp:10.10.10.201:6633
ovs-vsctl set controller br0 connection-mode=out-of-band
ovs-vsctl set controller br1 connection-mode=out-of-band
ovs-vsctl set bridge br0 other-config:disable-in-band=true
ovs-vsctl set bridge br1 other-config:disable-in-band=true

# set attached interfaces in promiscous mode
ifconfig ath0 0 up
ifconfig ath1 0 up

# start ovs bridges
ifconfig br0 192.168.11.1 netmask 255.255.255.0 up
ifconfig br1 192.168.12.1 netmask 255.255.255.0 up
```

Listing 1: Mobile node's OvS configuration example

A.2 ROUTER

The network nodes connected via physical link have a similar configuration between them. In this matter, Listing 2 shows the router configuration, where the VLAN's and OvS configuration can be extrapolated for the remaining physical link attached nodes, with the required IP and VLAN tags modifications.

```
#!/bin/bash
### OVS

## configure VLAN's
modprobe 8021q
lsmod | grep 8021q

# control.6 for the R1 -> controller in-band (of protocol)
vconfig add control 6
ip addr add 10.10.10.254/24 dev control.6
vconfig add control 5
ip addr add 10.10.15.254/24 dev control.5

# control.2 for the R1 -> AP1
vconfig add control 2
ip addr add 10.10.11.254/24 dev control.2

# control.3 for the R1 -> AP2
vconfig add control 3
ip addr add 10.10.12.254/24 dev control.3

# control.4 for the R1 -> AP3
vconfig add control 4
ip addr add 10.10.13.254/24 dev control.4

# control.7 for the R1 -> Terminal Receiver
vconfig add control 7
ip addr add 192.168.10.254/24 dev control.7

## start OVS
echo "load ovs kernel module"
modprobe openvswitch

lsmod | grep openvswitch

ovsdb-server /usr/local/etc/openvswitch/conf.db \
--remote=punix:/usr/local/var/run/openvswitch/db.sock \
--remote=db:Open_vSwitch,Open_vSwitch,manager_options \
--private-key=db:Open_vSwitch,SSL,private_key \
--certificate=db:Open_vSwitch,SSL,certificate \
--bootstrap-ca-cert=db:Open_vSwitch,SSL,ca_cert --pidfile --detach --log-file

ovs-vsctl --no-wait init
ovs-vswitchd --pidfile --detach
# mask mac address, to resolve bridge conflite
ifconfig control.2 hw ether 00:33:33:33:33:22
```

```
ifconfig control.3 hw ether 00:33:33:33:33:33
ifconfig control.4 hw ether 00:33:33:33:33:44
ifconfig control.5 hw ether 00:33:33:33:33:55
ifconfig control.6 hw ether 00:33:33:33:33:66
ifconfig control.7 hw ether 00:33:33:33:33:77

ifconfig control.2 up
ifconfig control.3 up
ifconfig control.4 up
ifconfig control.5 up
ifconfig control.6 up
ifconfig control.7 0 up

# add bridge to ethernet interface (control.3)
ovs-vsctl add-br router
ovs-vsctl add-port router control.7

# connect R1 to a controller
ovs-vsctl set-controller router tcp:10.10.10.201:6633

# start R1 bridges
ifconfig router 192.168.10.254 netmask 255.255.255.0 up

# enable forwarding
echo 1 > /proc/sys/net/ipv4/ip_forward

### route table
# delete default gateway - eric
# ip route del default via 10.110.1.201
ip route add 192.168.11.0/24 via 10.10.11.101
ip route add 192.168.12.0/24 via 10.10.12.101
ip route add 192.168.13.0/24 via 10.10.13.101
```

Listing 2: Bash script with router's configuration example

A.3 ACCESS POINTS

In order to start the hostapd as a deamon, the configuration file presented in Listing 3 was created and linked to the file */etc/default/hostapd* through the line: *DAEMON_CONF="/etc/hostapd/hostapd.conf"*

```
interface=ath1
driver=nl80211
ssid=sdn-ap1
hw_mode=g
channel=6
```

Listing 3: Hostapd configuration file

With the purpose of setting up the respective wireless interface in the start up, the lines in Listing 4 were added to the file */etc/network/interfaces*.

```
auto ath1
iface ath1 inet static
address 192.168.11.101
netmask 255.255.255.0
```

Listing 4: Setting up the wireless interface

A.4 CONTROLLER

The controllability of framework was taken by the POX controller. To complement the already existent libriary, a new script, capable of performing the MN handover was created and made available as open-source on Github[2]. Listing 5 illustrates how POX can be started with the respective scripts.

```
The script ext/handover_mn.py handles the Mobile Node handover request.
Run it along with l3_learning
* You can run with the "py" component and use the CLI:
  ./pox.py forwarding.l3_learning handover_mn py

The script ext/handover_mn_rules.py handles the Mobile Node handover request
while implementes a certain number of dummy rules. As in the script before,
run it along with l3_learning

* You can run with the "py" component and use the CLI:
  ./pox.py forwarding.l3_learning handover_mn_rules py
```

Listing 5: Header of the POX's handover script

[2]http://atnog.github.io/of_mobilenode/

APPENDIX B

Traffic Monitor

In this appendix the traffic monitor of the access point will be explored. While in section B.1 a flow chart illustrates the execution flow, the bash code is presented in section B.2, Listing 6.

B.1 Flow Chart

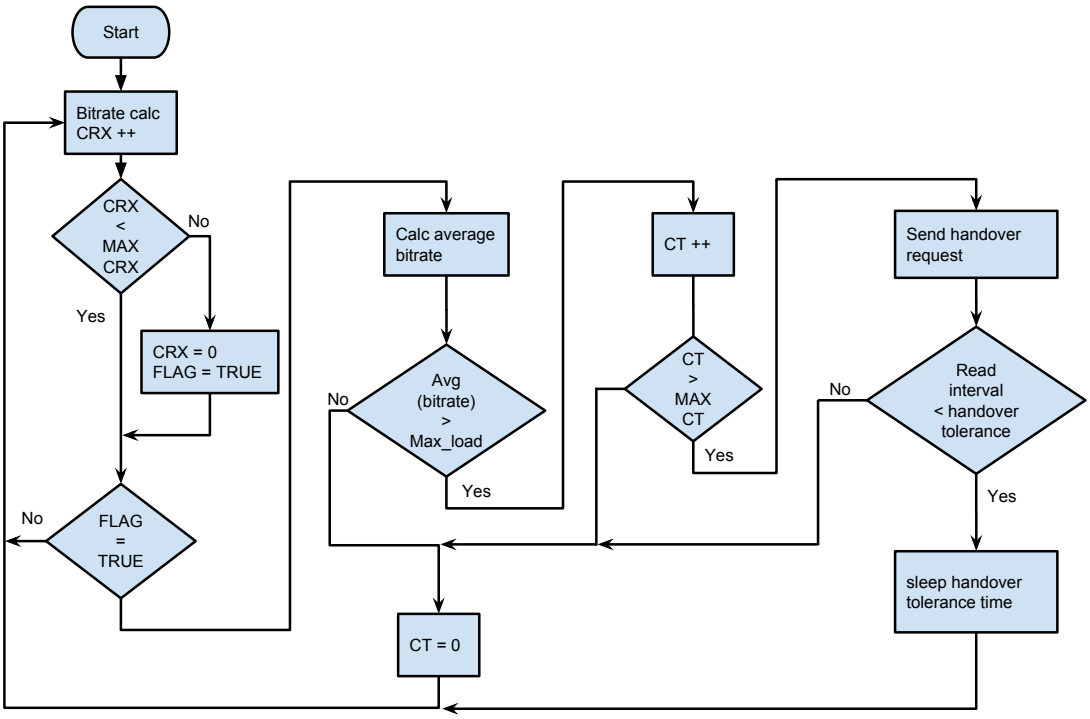


Figure B.1: Overload detecting flow chart

B.2 BASH SCRIPT

```
#!/bin/bash
AUTOR="Flavio Meneses @ University of Aveiro, 2015"
USAGE="Usage: `basename $0` -h | help"
# default values
IFACE="eth0"
T="10"

if [[ "$1" == "-h" ]]; then
  echo "$USAGE"
  echo "AUTOR: $AUTOR"
  echo ""
  echo "[-h ]                    help"
  echo "[-i | --interface]       select interface"
  echo "[-t | --timeinterval]    choose the time interval through the array
      index"
  echo "                         [0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 2 3 4
      5]"
  exit

elif [[ "$1" == "--default" ]]; then
  echo "DEFAULT inputs: -i $IFACE -t $T"

else
  if [[ $# < 2 ]]; then
    echo "Invalid argument [$1]"
    echo "$USAGE"
  exit
  fi
  while [[ $# > 1 ]]
  do
    key="$1"
    case $key in
      -i|--interface)
        IFACE="$2"
        shift
        ;;
      -t|--timeinterval)
        if [[ "$2" -lt 15 ]] && [[ "$2" -ge 1 ]]; then
          T="$2"
        else
          echo "Array index too high! (value between [1-14])"
          exit
        fi
        shift
        ;;
      *)
        echo "Invalid argument [$1 $2]"
        echo "$USAGE"
        exit
        ;;
    esac
```

```
    shift
  done
fi

TIME_INTERVAL=(0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 2 3 4 5)

# will be divided by 10
TIME_INTERVAL_BITRATE=(1 2 3 4 5 6 7 8 9 8 20 30 40 50)
HANDOVER_TOLERANCE=0.5
# variables
CRX=0
MAX_CRX=3
CT=0
MAX_CT=3
FLAG=false
# 54 Mbps = 6.75 MBps = 6912 KBps
#MAX_LOAD=6912
MAX_LOAD=$((20 * 1024 * 1024))
PERCENT=90
TRIGGER_LOAD=$((PERCENT * MAX_LOAD / 100))

echo Interface: "$IFACE"
echo Interval Time: "${TIME_INTERVAL[$T-1]}"
echo Trigger Load: "$TRIGGER_LOAD"

while true; do
  # read interface received bytes
  read RX_PCKTS_OLD <<< $(ifconfig $IFACE | grep 'RX bytes'| awk '{ print $2}'
      | grep -o [0-9] | tr -d '\n')
  sleep ${TIME_INTERVAL[$T-1]}
  # read interface received bytes afteer an interval time
  read RX_PCKTS <<< $(ifconfig $IFACE | grep 'RX bytes'| awk '{ print $2}' |
      grep -o [0-9] | tr -d '\n')

  # calc the interface bitrate and save it in the buffer
  # the multiplication and division by 10 is due to the fact the bash cannot
      calc float numbers
  RX_PCKTS_OLD=$(RX_PCKTS_OLD * 8)
  RX_PCKTS=$(RX_PCKTS * 8)
  RX_BITRATE_VALUES[$CRX]=$(((RX_PCKTS - RX_PCKTS_OLD) * 10 /
      TIME_INTERVAL_BITRATE[T-1]))
  #RX_BITRATE_VALUES[$CRX]=$(((RX_PCKTS - RX_PCKTS_OLD) * 10 /
      TIME_INTERVAL_BITRATE[T-1] / 1024 / 1024))
  #echo buffer: "${RX_BITRATE_VALUES[*]}"

  # create a circular buffer
  CRX=$((CRX+1))
  if [[ CRX -eq MAX_CRX ]]; then
    # buffer full
    CRX=0
    # active flag to allow the average calc
    if [[ "$FLAG" = false ]]; then
      #echo "flag"
      FLAG=true
```

```
    fi
  fi

  # if buffer is already full calc bitrate average
  if [[ "$FLAG" = true ]]; then
    # calc array average
    BITRATE_AVG=0
    for (( i = 0; i < $MAX_CRX; i++ )); do
      BITRATE_AVG=$((BITRATE_AVG + RX_BITRATE_VALUES[i]))
    done
    echo "$BITRATE_AVG"
    BITRATE_AVG=$((BITRATE_AVG / MAX_CRX))
      #BITRATE_AVG_M=$((BITRATE_AVG / 1024 / 1024))

      echo bitrate: "$BITRATE_AVG" bps
      #echo bitrate: "$BITRATE_AVG_M" Mbps

    if [[ BITRATE_AVG -ge TRIGGER_LOAD ]]; then
      #flag trigger count ++
      CT=$((CT+1))
      echo "bitrate: $BITRATE_AVG Mbps"
      echo "Trigger Counter: $CT"
      if [[ CT -eq MAX_CT ]]; then
        # send trigger to controller
        echo "help" > /dev/udp/33.33.33.33/58549
        echo "TRIGGER SENT!"
        CT=0
        if [[ ${TIME_INTERVAL[$T-1]} < $HANDOVER_TOLERANCE ]]; then
          #echo "WAIT FOR HANDOVER"
          sleep $HANDOVER_TOLERANCE
        fi

      fi
    else
      #flag trigger count=0
      CT=0
    fi
  fi

done
```

Listing 6: Traffic monitor bash script

APPENDIX C

Demos

In this appendix the demo files will be explored, where in the section C.1 the mobile node demo is presented, showing the required commands to start the video streaming as well as the power and bitrate transmission. Section C.2 explores both mobile node and access point demos, illustrating the data transmission using the *iperf* application in the MN's case. The AP demo shows how the traffic monitor was started.

C.1 SCENARIO 1

```
#!/bin/bash
# DEMO: handover with openflow

## start capture
sudo tcpdump -ni ath1 -w file_ath1_dump.pcap &
sudo tcpdump -ni ath0 -w file_ath0_dump.pcap &

## implement basic rules in MobileNode's ovs-bridges
sudo ovs-ofctl add-flow br0 priority=0,actions=normal
sudo ovs-ofctl add-flow br0 priority=0,actions=normal
sleep 2

## T=0
# start video streaming
cvlc -dvvv BigBuckBunny1080p.mp4 --sout
    '#duplicate{dst=rtp{{mux=ts,dst=192.168.10.1}}}'
sleep 20
## T=20
# reduce the troughput - decrease the TxPower and bit rate
sudo iwconfig ath0 txpower 1
sudo iwconfig ath0 rate 5.5M
sleep 10
## T=30
```

```
# send "handover" solicitation packet to the controller
echo "help" > /dev/udp/11.11.11.11/58549
sleep 30

## T=60
# kill the processes
sudo pkill tcpdump
pkill vlc
```

Listing 7: Bash script demo for the SDN extension scenario

C.2 SCENARIO 2

```
#!/bin/bash
# DEMO: handover with openflow

## start capture
sudo tcpdump -ni ath1 -w file_ath1_dump.pcap &
sudo tcpdump -ni ath0 -w file_ath0_dump.pcap &

## implement basic rules in MobileNode's ovs-bridges
sudo ovs-ofctl add-flow br0 priority=0,actions=normal
sudo ovs-ofctl add-flow br0 priority=0,actions=normal
sleep 2

## T=0
# start data streaming
iperf -c 192.168.10.1 -u -b 8m -t 60

## T=60
# kill the processes
sudo pkill tcpdump
```

Listing 8: Mobile node bash script demo for the multiple flow scenario

```
#!/bin/bash
# DEMO: handover with openflow

## start traffic monitor with a 1 second of reading interval as a deamon
./parseRXpckts.sh -i ath1 -t 10 &
sleep 2

## start capture
sudo tcpdump -ni ath1 -w file_ath1_dump.pcap &
sudo tcpdump -ni control.3 -w file_c3_dump.pcap &

## implement basic rules in MobileNode's ovs-bridges
sudo ovs-ofctl add-flow ap2 priority=0,actions=normal
sleep 60

## T=60
# kill the processes
sudo pkill tcpdump
sudo pkill parseRXpckts
```

Listing 9: AP2 bash script demo for the multiple flow scenario