IUL School of Technology and Architecture
Department of Information Science and Technology

Towards a Software Defined Network based Multi-Domain Architecture for the Internet of Things

Leonel Duque Piscalho Júnior

Dissertation submitted as partial fulfilment of the requirements for the degree of
Master in Telecommunications and Computer Engineering

Supervisor:
Prof. José André Rocha Sá Moura, Assistant Professor,
ISCTE – IUL
Co-Supervisor:
Prof. Rui Neto Marinheiro, Assistant Professor
ISCTE – IUL

December 2019

Acknowledgements

There is no true victory without effort and sacrifice. This master's dissertation could not come to an end without the precious support of many people.

Firstly, I thank God for always looking out for me and answering my prayers even though they were mostly done in times of despair and distress.

Special thanks to my advisors, Professors José Moura and Rui Marinheiro for all the patience, commitment, motivational and practical support with which they have always guided me in this work.

I also wish to thank my family, especially my parents Leonel Duque Piscalho and Maria Isabel de Carvalho for the unconditional support throughout my life, have been and always will be examples of work and honour for me. Still in my family, a special thanks to my girlfriend and partner Moira Margalha for the motivational support and the time she gave up her personal life to help me on this journey.

Special thanks to the Instituto de Telecomunicações for the conditions available for this dissertation.

Finally, thank my classmates Mickaël Cunha and Pedro Manso, despite they have already finished the course, have shown themselves available to help.

We know it is hard to go after a dream and fight for what you want, but all those sacrifices made today will be paid tomorrow when it all comes true.

Resumo

As redes atuais de comunicação são heterogéneas, com uma diversidade de dispositivos e serviços, que desafiam as redes tradicionais, dificultando a satisfação dos requisitos de qualidade de serviço (QoS). Com o advento das Redes Definidas por Software (SDN), novas ferramentas surgiram para projetar redes mais flexíveis. O SDN oferece uma gestão centralizada para os fluxos de dados em redes distribuídas de sensores.

Assim, o principal objetivo desta dissertação é de investigar uma solução que cumpra os requisitos de QoS do tráfego originado em dispositivos de Internet das coisas (IoT). Este tráfego é transmitido para a Internet, num sistema distribuído com múltiplos controladores SDN.

Para atingir o objetivo, projetamos uma topologia de rede com múltiplos domínios, cada um gerido pelo seu controlador. A comunicação entre os domínios, é feita através dum domínio de trânsito SDN com a aplicação SDN-IP do controlador Sistema Operativo de Rede Aberta (ONOS). Emulamos também uma rede para testar a QoS através de filas de espera do OpenvSwitch. O objetivo é criar prioridades de tráfego numa rede com dispositivos tradicionais e de IoT simulados.

De acordo com os testes realizados, conseguimos garantir a comunicação entre domínios SDN e comprovamos que a nossa proposta é reativa a uma falha na topologia. No cenário do QoS demostramos que, através da inserção de regras OpenFlow, conseguimos priorizar o tráfego e oferecer garantias de qualidade de serviço. Desta forma comprovamos que a nossa proposta é promissora para ser utilizada em cenários com múltiplos domínios administrativos.

Palavras-chave: SDN, IoT, QoS, ONOS, Múltiplos-Domínios

Abstract

The current communication networks are heterogeneous, with a diversity of devices and services that challenge traditional networks, making it difficult to meet quality of service (QoS) requirements. With the advent of software-defined networks (SDN), new tools have emerged to design more flexible networks. SDN offers centralized management for data streams in distributed sensor networks.

Thus, the main goal of this dissertation is to investigate a solution that meets the QoS requirements of traffic originating on Internet of Things (IoT) devices. This traffic is transmitted to the Internet in a distributed system with multiple SDN controllers.

To achieve the goal, we designed a multi-controller network topology, each managed by its controller. Communication between the domains is done via an SDN traffic domain with the Open Network Operating System (ONOS) controller SDN-IP application. We also emulated a network to test QoS through OpenvSwitch queues. The goal is to create traffic priorities in a network with traditional and simulated IoT devices.

According to our tests, we have been able to ensure the SDN inter-domain communication and have proven that our proposal is reactive to a topology failure. In the QoS scenario we have shown that through the insertion of OpenFlow rules, we are able to prioritize traffic and provide guarantees of quality of service. This proves that our proposal is promising for use in scenarios with multiple administrative domains.

Keywords: SDN, IoT, QoS, ONOS, Multi-Domain

Tables of Contents

List of Tables

List of Figures

List of Abbreviations

AMQP – Advanced Message Queuing Protocol

API – Application Programmable Interface

ARP – Address Resolution Protocol

AS – Autonomous System

BGP – Border Gateway Protocol

CLI – Command Line Interface

DISCO – Distributed multi-domain SDN controllers

GUI – Graphical User Interface

ICMP – Internet Control Message Protocol

ICONA – Inter Cluster ONOS Network application

IFWD – Intent Reactive Forwarding

IMR – Intent Monitor and Reroute

IoT – Internet of Things

IP – Internet Protocol

JSON – JavaScript Object Notation

MPLS – Multi Protocol Label Switching

NIB – Network Information Base

NOS – network operating system

ODL – OpendayLight

ONF – Open Networking Foundation

ONOS – Open Network Operating System

OSGi – Open Services Gateway Initiative

OVS	– OpenVSwitch
OvSDB	– Open vSwitch Database
QoS	– Quality of Service
REST	– Representational State Transfer
RFID	– Radio Frequency Identification
RTP	– Real-time Transport Protocol
SDI	– Software Defined Cross-Domain Routing
SDN	– Software Defined Networking
SNMP	– Simple Network Management Protocol
SSL	– Secure Socket Layer
TC	– Traffic Control
TCP	– Transmission Control Protocol
UDP	– User Datagram Protocol
VLAN	– Virtual Local Area Network
VLC	– Video Lan Client
VM	– Virtual Machine
WAN	– Wide Area Network
WSN	– Wireless sensor network
YANG	– Yet Another Next Generation

“If you have a positive attitude and constantly strive to give your best effort, eventually you will overcome your immediate problems and find you are ready for greater challenges”

Pat Riley

1. Introduction

1.1. Context

The exponential data traffic growth and the heterogeneity of communications networks are challenging the legacy networking management solutions, because these scenarios demand for a high-level of complexity to interconnect different types of services and smart devices like the internet of things networks (IoT). They exchange real-time information through the networking infrastructure which is processed by intelligent applications, which implies not only various types of traffic, but also the ability to offer quality of service (QoS) guarantees across the network [1].

Due to high complexity of the infrastructure configuration and the difficulty of legacy network innovation, many challenges arise to meet the requirements of today's networks, and the advent of software defined networks (SDN) offers to the network designers new methods for designing flexible and more efficient networks.

SDN stands out for its flexibility, programmability and centralized management, which makes the SDN an increasingly popular paradigm. It separates the data layer from the control layer to allow operational logic to pass to controller and data plane to handle only data routing. This mitigates some limitations of legacy solutions and accelerates innovation in several key network functions [2],[3].

Initially, most SDN contributions offered a single controller design to manage the entire network. Nevertheless, this faces some robustness and performance problems when it is deployed at larger networks. The robustness issue is due to the potential single point of failure of the centralized SDN controller; and, the performance issue is associated to the eventual bottleneck of having a single controller with scarce available computing resources to satisfy the entire service demand. An alternative to the single controller is the multiple controllers [3]. However, the design with multiple controllers increase the complexity of the network and put many challenges in managing efficiently the entire networking infrastructure.

Due to the size, heterogeneity, and complexity of current networks, approaches based on the hierarchical network division into multiple Autonomous System (AS) or SDN domains is a viable alternative. Each domain focuses on managing its own network subset and optimizing performance for providing QoS guarantees to end users. Some research like [4], study ways to improve the IP domain routing management and provide end-to-end QoS paths [5]. This research is based on a centralized controller approach that handles routing in only one administrative domain. However, the SDN configuration in inter-domain scenarios is more challenging, and the coherent interconnection of all these controllers is vital to ensure reliable end-to-end services, such as routing and QoS deployment.

The interaction between the different SDN domains depends on a inter domain routing protocol, and BGP is the base protocol for this interaction. ONOS [6] and ODL [7] SDN controllers are those that meet the distributed complement requirement and are most commonly used in large-scale WAN projects. Both are powerful SDN controllers with slight performance differences as shown in [8].

The authors of [9], suggest a solution designated by Inter Cluster ONOS Network application (ICONA). This solution manages a large networking scenario under the same administrative domain (i.e. GEANT network) with geographically distributed ONOS controllers. Their proposal aims to enhance fault tolerance and decrease the delay response to events originated in large-scale networks. Another contribution [10], proposes a gradual implementation of SDN-based solutions over different administrative domains which need to interoperate with other non-SDN based domains. They study a peering application among distinct Autonomous Systems (ASs) called SDN-IP which runs on the top of the SDN controller.

The SDN-IP application will be very important to achieve our goal, because unlike existing studies, our work aims to connect different SDN-based administrative domains to create consistent inter-domain routing. Due to the limited contributions and limited resources to ensure QoS on IoT networks, it gives for network administrators the freedom to implement their own QoS algorithms. However, we aim, as a novelty, to ensure QoS support in distributed systems with multiple SDN controllers.

1.2. Research Questions

The Quality of service gives resources to intelligently manage the bandwidth, minimizing packet loss and delays in different types of services. As the network grows and domains expand, the network load grows and the resources become exhausted. In this way, end users cannot have satisfied their initial QoS-level expectations for various applications .

So, the research question that serves as a base for this dissertation is "How to provide the necessary resources to meet QoS and robustness requirements for traffic originated in heterogeneous IoT devices, in a multi-domain SDN-based system?"

1.3. Research Goal

The main objective of this dissertation is to investigate a solution that provides inter-domain communication, failure robustness and meets the QoS requirements of traffic originating of IoT devices. So, to achieve that goal we will design, deploy a distributed SDN formed by multiple domains, a controller for each domain and and test QoS through differentiating traffic and installing OpenFlow rules according to each priority.

1.4. Investigation Method

To test the viability of our proposal, several studies were performed, as follows:

- Literature Review about SDN, its architecture, protocols, controller design and the different types of controllers. It will be also studied the inter-domain SDN communication and how QoS can be provided in relevant emerging scenarios like IoT;
- Design and Deployment of a network prototype to ensure communication between different SDN administrative domains and adopt methods to ensure QoS in the exchange of information between the end devices;
- Evaluate the results and prove that the proposed solution is valid to answer in a satisfactory way our initial research question.

1.5. Main contributions

The main contributions of this dissertation, is the implementation of a distributed network system, formed by multiple domains totally based on SDN able to meet QoS and robustness requirements for routing heterogeneous traffic inter-domain. The routed traffic is from heterogeneous devices, including IoT, located at the network edge.

1.6. Dissertation Outline

The remainder part of the current dissertation has the following organization. **Chapter 2** presents the literature review. All the theoretical background needed along the current work is discussed. Here, we analyze how a distributed SDN system works, and what are the more recent contributions in the current topic. **Chapter 3** presents our proposal including the system design. **Chapter 4** is about the deployment of the proposal, including the performed tests and a complete discussion about their results. Finally, **Chapter 5** presents some general conclusions about the current dissertation and some future work.

2. Literature Review

The literature review provided in this chapter, identifies the current related research and provides some background in the fundamental technologies and systems used along the current work. We will briefly discuss SDN and its architecture where we explain each layer and how one interacts with the other. The OpenFlow protocol will be presented, which is the most used to provision the communication between the physical devices and SDN controllers.

Different SDN designs for single and distributed controllers as well as existing solutions will be presented. A special attention is given to the distributed architecture approach of multiple controllers, and methods to ensure the inter-domain communication will also be studied. As SDN has been applied in different emerging areas, different ways of providing quality of service guarantees in IoT networks will be studied.

2.1. SDN Architecture

In legacy or traditional networks, the control and data forwarding functions are both embedded in the same network device (e.g. switch, router). In this way, it is difficult to deploy new network services because the manual configuration of this legacy network devices takes a long time and it is more prone to errors [12]. To overcome these limitations, the SDN is a viable candidate. The SDN, is a network paradigm, which introduces the programmability options to how the network infrastructures should operate. When the SDN concept is used, it implies, in each network device, the separation between the control layer and the data layer. In this way, the SDN controller provides the logic of network service operation, while physical devices only handle the data plane forwarding, as fast as possible, according to their switching fabric [14].

The SDN paradigm offers some advantages, in how the network management is performed, namely the simplicity for introducing new network services, the ability to innovate in the way the network is managed, reduced equipment costs and the high-level programming based on management policies, which is performed at the top of the SDN controllers. The reference [15], presents the following benefits for using the SDN approach:

- It is simpler and less error-prone to program, run applications, and modify network policies through high-level languages built into controllers compared to low-level configurations implemented directly on network devices;

- Offers a centralized network overview;
- Enables to automatically react to network topology changes, following high-level policies.

Since not everything just has advantages, the use of SDN has some limitations that have been worked to overcome them. Among these limitations are security issues, controllers become a priority target for attackers. Since all the "intelligence" of the network is centralized in a single point and if the controller attacked, can be compromising the entire network. To mitigate this limitation, there is the option to use distributed controllers, which can assist in the recovery of physical and logical failures. However, there is a difficulty in keeping the network state always up to date for all controllers in the network.

Another limitation is due to technical support for SDN networks. To be a new paradigm compared to other network architectures, there is a shortage of professionals in this area, so the support is limited.

The SDN architecture consists essentially of three layers: infrastructure or data, control and application. The Figure 1 shows the SDN architecture. We discuss below the several layers of this architecture.

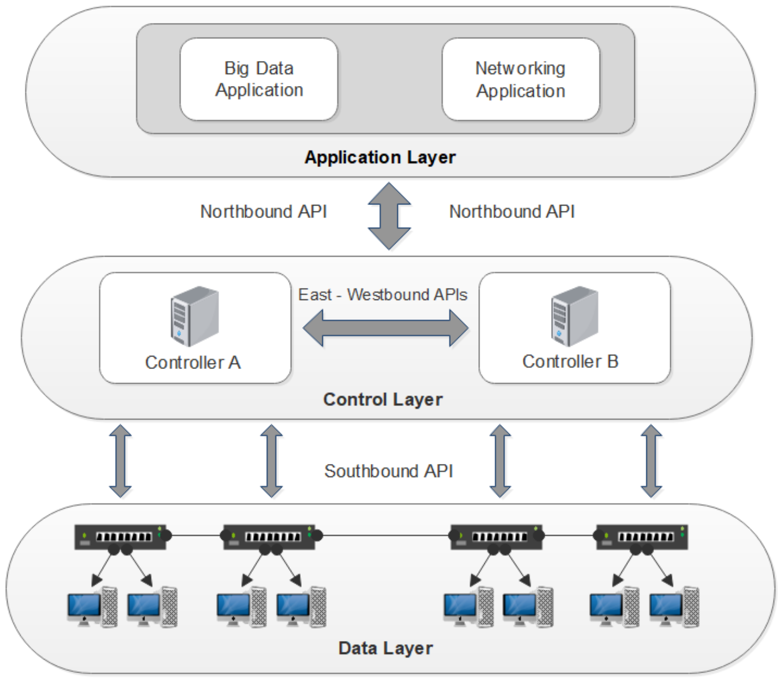


Figure 1. SDN Architecture

The **Data layer** is the bottom-most layer of the SDN architecture. It aggregates diverse SDN compatible network devices, such as routers and switches. This layer is responsible for forwarding packets, which divert packets through the network topology according to the decisions taken by the control layer. The communication between data layer network devices and SDN controllers is typically made by Southbound API [13][16].

The **Control layer** is an intermediate layer of the SDN architecture. It is also designated as the network operating system (NOS). It consists of one or more SDN controllers that support the network control logic. The communication between top-most applications and the SDN Controller is made via a northbound interface. This interface provides to the application layer a high-level abstraction of the network infrastructure, hiding specifics of the network infrastructure. There are several SDN controllers that will be discussed in the following sections. In spite of the existence of several SDN controllers, there is among them a common set of basic functions like topology manager, statistical manager, routing module, device manager, among others [13][16].

The **Application layer** is the top-most layer of the SDN architecture. This layer communicates with intermediate-level SDN controllers via Northbound APIs (e.g. RESTful). In this layer are running several programs such as monitoring, security, load balancing and flow control. Through the Northbound API, SDN services and applications have access to the network status and react to that by sending in the opposite direction some instructions to the SDN controllers. These instructions being executed by the SDN controllers imply the installation of local flow rules in the network devices forming the data plane [13][16].

Separating the three already mentioned layers of the SDN architecture, there are two vertical communication channels to connect each pair of them Northbound/Southbound APIs, as well as East/Westbound APIs to provide two horizontal communication channels between multiple controllers. We give below further details about all these communication channels.

The **Southbound API** is the communication channel between the data layer and the control layer. Consists of protocols that determine how the SDN controller should direct information to the data plan network configurations, flow entries installation, and insertion of forwarding rules into switches. There are many protocols such as OpenFlow, OvSDB [17], SNMP [18], NetConf

[19]. In this document, we will discuss only SDN solutions with OpenFlow (section 2.1.1) , that it is currently more used for this type of implementations [13] .

The **Northbound API** is used to connect the applications and network services from the application layer to the SDN controllers. This API provides a dynamic management of network traffic flows through programming feature. Different from Southbound API, the Northbound API is not supported by a standard protocol [13].

The **East/ Westbound API** provides the communication between the SDN controllers in a distributed network. They can be used to interconnect conventional IP domains and SDN networks, also connects different administrative domains with federated SDN controllers. Although there is no standardized protocol for this API, conventional border protocols such as BGP can be used to support the interconnection of remote SDN domains [13][20].

2.1.1. OpenFlow Protocol

The OpenFlow protocol emerged in 2008, from a project launched by Stanford University. In 2011, a group of service providers have created the organization called the Open Networking Foundation (ONF), to standardize and promote the use of OpenFlow network protocol using SDN-based solutions. The OpenFlow is currently the most used protocol by SDN systems. It provides the communication between the control layer and the data layer through the Southbound API. The OpenFlow protocol allows the control layer to centrally specify how data traffic is forwarded through the data layer. These traffic forwarding decisions are made in OpenFlow compatible devices, following packet forwarding rules stored into local flow tables of those devices. The Table 1 compares some relevant features among the different versions of OpenFlow Protocol [16][13].

Table 1. OpenFlow Versions

OpenFlow version	Date	Features
OpenFlow 1.0.0 [21]	Dec,2009	Single flow table, IPv4
OpenFlow 1.1.0 [22]	Feb,2011	Multiple flow table, group table, MPLS support and VLAN
OpenFlow 1.2.0 [23]	Dec,2011	IPv6, multiple controllers
OpenFlow 1.3.0 [23]	Jun,2012	Single flow measure, IPv6 extend header, Meters for QoS Capabilities
OpenFlow 1.4.0 [24]	Oct,2013	Flow table synchronization mechanism, bundling message
OpenFlow 1.5.0 [25]	Dec,2014	Data packet type identification process, egress table, scheduled bundle expansion

An OpenFlow switch manages several components, as shown in Figure 2. These components are several Flow Tables, a Secure Channel, a Meter Table and the client part of the OpenFlow Protocol.

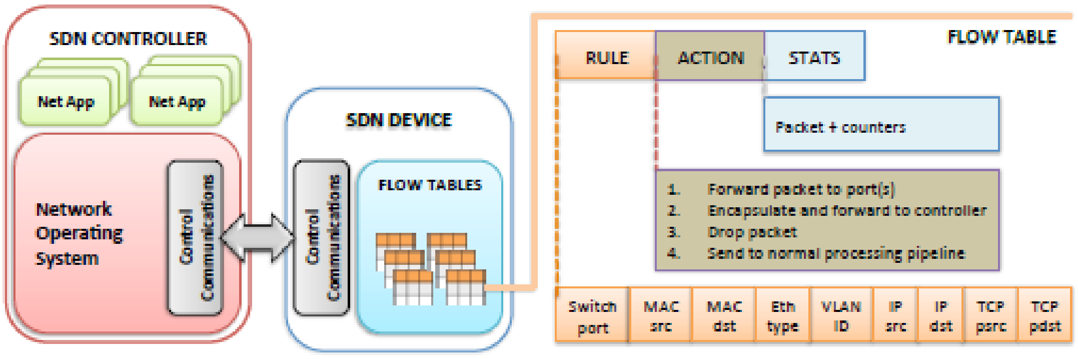


Figure 2. OpenFlow - enabled devices [15]

The OpenFlow secure channel connects each network device to the remote SDN controllers via secure or direct SSL channel over TCP. The secure channel supports three types of messages, controller-to-switch messages, asynchronous and symmetric messages. The external remote controller uses the OpenFlow protocol to manage OpenFlow enabled network devices. The list of SDN controllers are shown in the next sections. Flow tables and group tables are responsible for performing packet lookups, matches and message forwarding. The forwarding table consists of a list of flow entries in the matching format, actions, and counters. When a packet arrives, the header of that packet is compared against the diverse existing entries. In case there is a positive match, the counters associated to the matching rule are incremented and a particular action is performed. These actions can be as follows [16]:

- Forward the packet to a specific port if the received packet matches a flow entry in the flow table.
- If there is no rule with higher priority that matches the specifics of the received packet, the device tries to apply a default and more generic rule. This type of rule has normally an action that encapsulates the received packet and forwards it to the SDN controller for further analysis. After the analysis of the SDN controller has been made, the controller informs the device about the decision associated to the received packet. In addition, the SDN controller installs a new flow entry in the device to further local processing for the remaining packets of the same flow.
- Discard the package. This action can be taken to prevent malicious and denial of service attacks.

The meter table allows OpenFlow to implement simple QoS resources like traffic shaping, e.g. rate flow limitation.

2.1.2. SDN Controllers

The controller is the most important component of a SDN network. Located at the control layer, it has the main function to manage protocols and network resources. It also manages traffic on underlying network elements through a set of instructions called flow rules [26]. The communication with data layer network devices is done via Southbound API and with the top layer applications via Northbound API.

Nowadays, many SDN controllers are available, either open-source or commercial. Each controller features may differ from each other, but the main functionality of all controllers is similar, for example, topology information, statistics, notifications, and device management [13]. Therefore, the quantitative and qualitative comparative analysis of these controllers is very important to choose the more suitable SDN controller for a specific networking scenario. In this section, we will discuss about the SDN controller deployment design. The quantitative and qualitative comparative analysis of the various well-known SDN controllers will be presented.

2.1.2.1. SDN Controller Design

The initial SDN implementations used the single controller approach (see Figure 3) to manage the entire network [27]. In this approach, the network intelligence is centralized on a single decision point that keep an of network overview, including the traffic load on each device along the forwarding path. Some examples of single SDN controllers are as follows: Beacon [28], Floodlight [29], NOX [30], Ryu [31]. Despite the work in [31], which proves the good performance of this approach and some efforts like [32] to minimize the controller load, a single controller may not keep up with network growth when deployed on a large-scale system. An SDN design with a single controller can become unreliable due to the issue of a single point of failure. In addition, the single SDN controller can become overwhelmed when working with multiple simultaneous requests from the data plane, and thus cannot deliver the expected performance [27][16]. Issues such as scalability, reliability, and vulnerability in single controllers were mentioned in [33], and in [34].

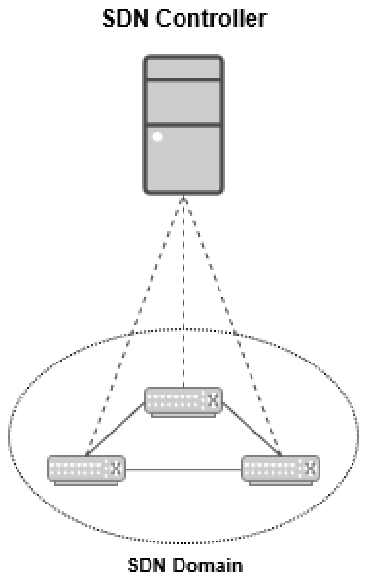


Figure 3. Single Controller Approach

Alternatively, the approach with multiple controllers mitigates the problems just discussed for the single controller approach [27]. Research such as [35], addresses the possibility to use a distributed controller plan in a WAN network, where multiple SDN controllers should be placed in the network topology to improve both control plane latency and fault tolerance. Despite, the multi-controller distributed approach has more advantages than the single-controller approach, challenges such as scalability, consistency, reliability and load balancing when implemented in large-scale networks were discussed in [16][20], solutions to overcome these challenges were also referenced in the literature.

The approach with multiple controllers can be classified in to two different types, such us, logically centralized but physically distributed and fully distributed [13]. In the first type (see figure 4), the controllers work in a coordinated way among them, i.e. they share information among them to keep a consistent and updated view of the entire network. Usually, the distributed system of SDN controllers has one master controller and others that are passive secondary controllers. The passive controllers, may be activated if the main controller fails. However, this method imposes many challenges, such as ensuring consistent synchronization among controllers in case of topology changes and if the master controller fails . Examples of this implementation are ONIX [36] and Hyperflow [37].

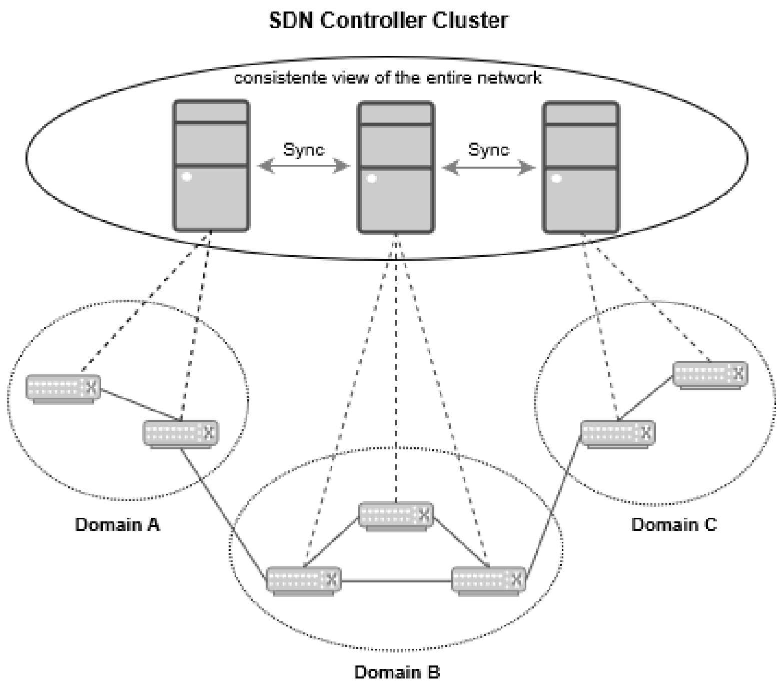


Figure 4. Logically centralized but physically distributed design

The second type of distributed controller approach is fully distributed controller. Here, the controllers are physically and logically distributed. Unlike the first approach, there is no state synchronization between controllers to keep a global view because to constant changes and inconsistencies in network state, it creates an overload that influences the bad performance of running SDN applications [13]. Each controller only manages its own domain, where they communicate with neighbouring controllers through specific routing protocols. Basically, fully distributed controllers can use two different designs, Flat Design and Hierarchical Design as shown in Figure 5.

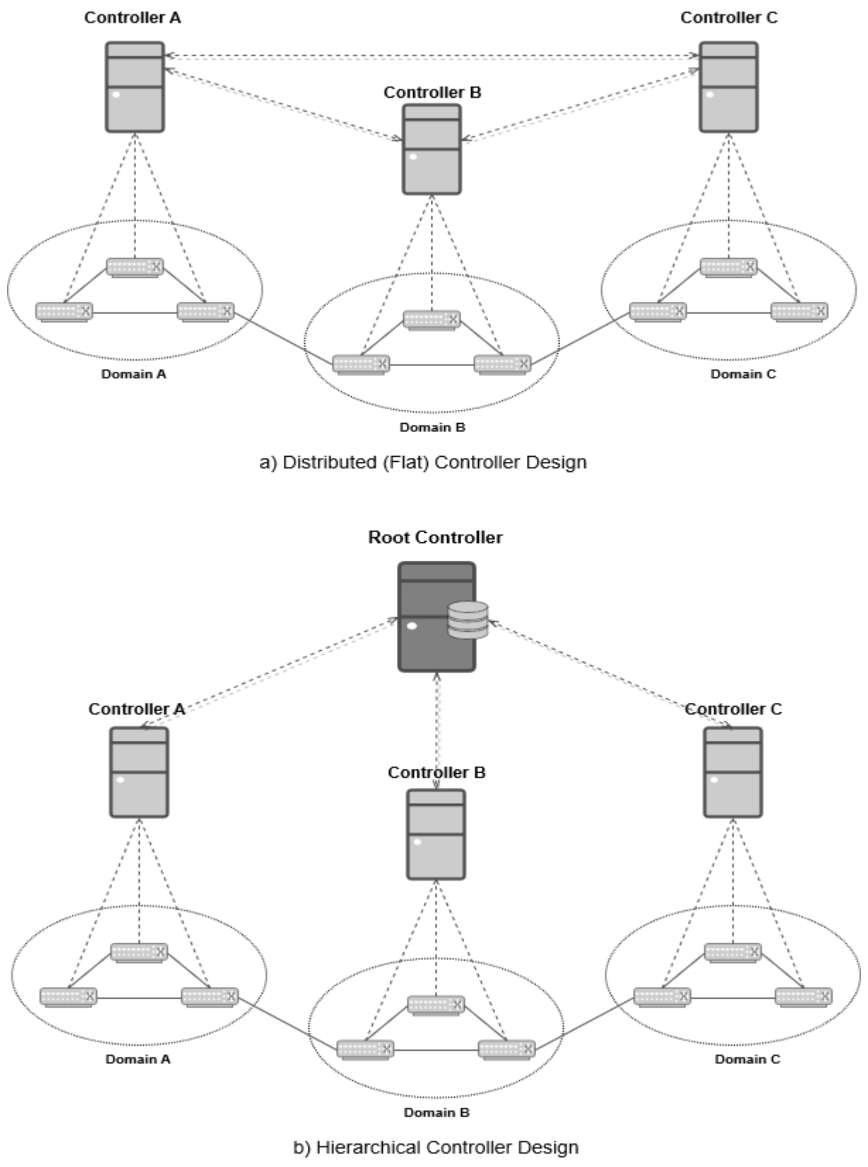


Figure 5. Distributed controller Approach

In the Flat Design (see Figure 5 (a)), controllers are distributed horizontally across different domains, where each controller manages a subset of the network. Different from the logically centralized but physically distributed, all controllers have equal rights and share information (e.g. topology, accessibility, device features, etc.) to each other, which communicates by East/Westbound APIs [27] [13].

The controllers can also be structured in a hierarchical or vertical architecture, with two layers of controllers. The Local Controllers are close to the data layer and only manage their own domain. Root Controllers are responsible for keeping all network information and ensure end-to-end communication among domain controllers [27] [13].

The fully distributed controller approach is the best solution for our research goal because, is the most realistic for deploying large-scale multi-domain networks while maintaining federation between controllers. Are typically examples of fully distributed controllers, OpenDaylight [7], ONOS [6] and Kandoo [38].

2.1.2.2. List of available controllers

Research such as [8], [39], presents the performance comparisons of the most popular open source controllers. They analyse the throughput and latency using benchmarking tools, in this case, Cbench. From the analysis made by [8], three controllers stood out for their good performance in different aspects. The ODL, contains better resources in terms of interface provider support. The ONOS provided the best performance results, with the ability to respond to requests faster in cases of traffic overload. However, RYU has the best latency results. The authors of [39], conclude from their comparison results that OpenDaylight and ONOS are the best choices, essentially for IoT scenarios.

The research paper [26], also makes quantitative analysis of nine different controllers using three benchmarking tools. In addition, it analyses thirty-four different controllers qualitatively their properties and capacities. It also presents different use cases of these controllers and the efforts made to improve their performance They have concluded that distributed controllers (OpenDaylight and ONOS) have slightly better performance in terms of latency and throughput compared to centralized multithreaded controllers (Floodlight, Beacon and Maestro), and significantly better performance than centralized and single-threaded controllers (NOX, POX and

RYU). However, despite the already mentioned winnings in terms of performance, the distributed controllers require more physical resources to run efficiently than other alternatives. The table 2, lists the most commonly available SDN controllers.

Table 2. SDN controller solutions

Controller	**Architecture**	**Description**
NOX [30]	Physically Centralized	The First OpenFlow controller. Used for high performance flow processing capabilities.
Beacon [28]	Physically Centralized	Java-based controller, supports high-performance stream processing capabilities using multithreaded pipeline and shared queues.
Floodlight [29]	Physically Centralized	Based on Beacon implementation, works with both physical and virtual OpenFlow switches to provide high-performance flow processing capabilities.
RYU [31]	Physically Centralized	Aims for logically centralized control and APIs, to create new management and network control applications.
ONIX [36]	Physically Distributed Logically Centralized	For large-scale network deployment, SDN's first distributed controller, it has a network information base (NIB) to manage its controllers.
Hyperflow [37]	Physically Distributed Logically Centralized	Designed over the NOX, it transmits and updates network events to the controllers in order to provide a consistent global view of the network.
ODL [7]	Fully Distributed	Java-based, supports a Cluster system for scalability and availability. Supports OSGi, Framework for programming through Northbound APIs.
ONOS [6]	Fully Distributed	Network operating system, ideal for multi-domain wide area network (WAN) and service provider networks. Provides scalability and fault tolerance
Kandoo [38]	Fully Distributed	Uses the hierarchical design with two controller level, local with own domain vision and root with global view and establishes communication between local controllers.

From the list, ONOS and ODL are the most popular open source controllers with fully distributed architecture design. ONOS is a distributed core controller designed for high availability, performance and scalability and support for next-generation devices. It keeps a global view of each controller instance running on different servers through the Multi-Cluster Peering provider, which helps to deploy services within a cluster formed by diverse domains. It utilizes a set of configurable packages (OSGi), which ensures interaction with top layer SDN architecture applications through high-level abstractions, e.g. Java and REST APIs. Different from ODL controllers, ONOS is designed to work toward service providers as well as carriers.

ODL is a controller that also supports the cluster system, which enables the network to be logically and / or physically divided into different network domains. The communication among the different ODL instances is made by the ODL-SDNi [40] application that works as an East-West protocol. It allows network developers to add new applications through a set of REST APIs (i.e. Yang-UI).

2.1.3. Inter Domains Communication

A multi-domain SDN architecture refers to a set of different administrative SDN domains or Autonomous Systems (ASs) that exchange information regarding the network status, QoS configuration, or other relevant network services such as packet routing to a destination prefix. From [41], controllers need to exchange information such as:

- **Reachability update:** helps to choose the best routing path between SDN domains, for a single flow that traverses the network infrastructure.
- **Flow setup, tear-down, and update requests:** has information such as path requirements, QoS that coordinate flow configuration requests.
- **Capability Update:** controllers exchange network-related resource information like bandwidth, QoS and others in order to aggregate the resources of the different controllers in the domain.

The East-Westbound APIs provides the SDN inter domain communication. However, the East-Westbound API has not been standardized, which brings to an interoperability challenge in deploying inter SDN domains projects. The Border Gateway Protocol (BGP) is the most commonly used Internet protocol for providing the end-to-end routing service over multiple administrative domains [41]. Then, each SDN controller needs to process an external learned BGP route to a destination prefix and translate it to local routing rules which are only valid within the network domain that controller is responsible for. It is expected that summing up the individual routing contributions from the diverse SDN controllers results in a final aggregated outcome that fulfils the end-to-end BGP route. In the real scenarios, an SDN domain contains different BGP routers that speak BGP and exchange updates to each other to achieve stability. The establishing BGP connection process, is shown in figure 6.

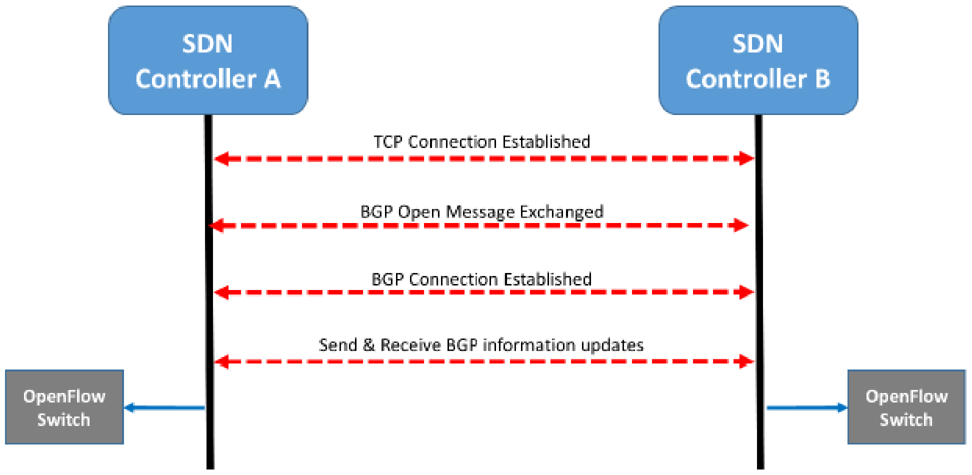

SDN Controller A
SDN Controller B
TCP Connection Established
BGP Open Message Exchanged
BGP Connection Established
Send & Receive BGP information updates
OpenFlow Switch
OpenFlow Switch

Figure 6. BGP Session [41]

The SDN controllers must be started as BGP speakers and a BGP_START connection happen. The BGP session is established through the Transmission Control Protocol (TCP) port to exchange messages and BGP configuration information for each neighbour (another BGP speaker) is configured manually. Once the TCP connection is established, it will be moved to OPEN state and during OPEN state, BGP speakers can negotiate session resources using OPEN messages (per RFC 5492). BGP peer are in a session, the controllers pass to the ESTABLISHED state and exchanges BGP UPDATE messages. These messages contain information such as accessibility data, bandwidth information, and other information that facilitate routing between SDN domains. The routing choice, happens when there is more than single path available based on the BGP process. When the path is established, the packets are successfully forwarded between SDN domains, through the BGP OPEN and UPDATE messages [41] [42].

Therefore, based on the potential and research found in the literature on the development of communication between SDN domains, BGP will be used as the Inter-domain SDN communication protocol throughout this thesis. Research like WE Bridge [43] and in fully distributed controllers such as OpenDaylight and ONOS use BGP to ensure communication between SDN domains. Table 3, presents methods to implement a project with BGP in some SDN controllers.

Table 3. SDN Inter-Domain Communication methods

Inter-Domain SDN Communications	**Proposed Base Protocol (s)**	**Description**
IETF SDNi [44]	BGP or SIP	Using the extensions of BGP or SIP
DISCO [45]	AMQP	Limit the interoperability with legacy IP network, proposed an additional BGP agent
EW Bridge [43]	BGP	Modify BGP update into JSON form
ODL SDNi [40]	BGP	Exploit BGP applications of the controller
ONOS SDN-IP	BGP	Exploit BGP applications of the controller

SDN Interconnection (SDNi) [44] was the first work to provide communication between SDN domains. SDNi proposed BGP and SIP as implementation protocols. Since 2012 there are no advances related to SDNi and thus appears another framework called DISCO [45] for communication between domains. This framework works on the Floodlight controller and initially used the Advanced Messaging Queuing Protocol (AMQP). However, due to limitations of interoperability with legacy networks they are using the BGP protocol.

East-West Bridge (EW Bridge) [43], emerges as an advanced method of interoperability between SDN and legacy IP networks, allowing it to exchange elementary network information between distinct administrative domains. It uses BGP with the modification of the BGP UPDATE message to use JavaScript Object Notation (JSON).

ODL-SDNi [40] is a BGP application that supports the exchange of information between ODL controllers in multiple domains. In ONOS controller, the SDN-IP application [46] was developed to ensure interconnection between SDN domains with legacy IP network domains and SDN domains.

2.2. ONOS Controller Overview

To achieve the goal of this dissertation, the controller chosen was the ONOS. To work with ONOS, it is essential to understand its operation and internal architecture. In this section, we will take a brief look about this controller and how the different ONOS modules are organized. In addition, we analyse the main inter-domain proposals and projects already proposed, especially, the SDN-IP, which is an application running in ONOS that connects distinct SDN domains using the BGP protocol.

2.2.1. System Components

ONOS is one of the most widely used controllers for fully distributed scenario deployments. Developed by ON.LAB, was designed to meet carrier-level requirements for scalability, high availability, and performance. ONOS, can work as a cluster system (multiple controller instances), providing fault tolerance if any instance fails, and real time updates of the network, without interfering with the network traffic. Based on java, it has many modules managed as Open Service Gateway Initiative (OSGi) bundles in the Karaf environment. So, it provides a high-level abstraction to application programmers.

So, it is possible to develop new applications to run on ONOS. Distributed core, full coherence, north and south abstraction, software modularity, easy addition and maintenance of servers, are the most important benefits of ONOS architecture [47]. The system components of the ONOS architecture are shown in Figure 7.

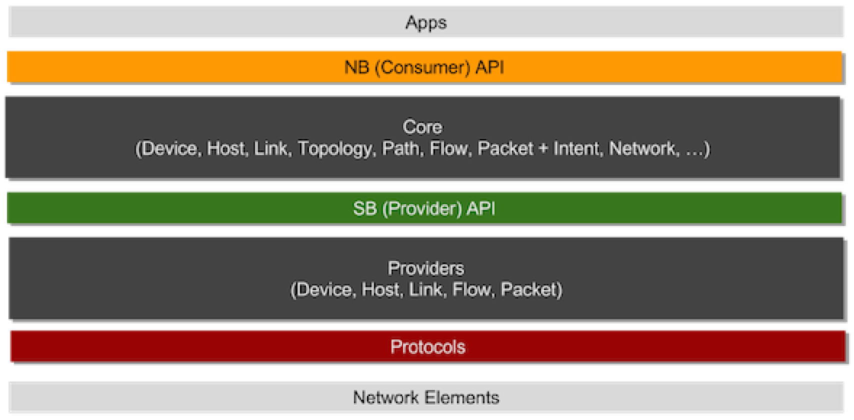

Apps
NB (Consumer) API
Core
(Device, Host, Link, Topology, Path, Flow, Packet + Intent, Network, ...)
SB (Provider) API
Providers
(Device, Host, Link, Flow, Packet)
Protocols
Network Elements

Figure 7. ONOS Architecture [47]

The **Distributed Core** is responsible for the management of resources, provides scalability, high availability and performance. In addition, it offers resources to the level of the operator control plan of the SDN. The ability of ONOS on being run as a cluster, allows on quickly meet the needs of the control plan SDN and networks of service providers.

The **Northbound Abstraction/APIs** provide configuration and management services for the development SDN applications. Includes representation by means of network graphs and intentions of applications that facilitate the development of service control, configuration and management.

The **Southbound Abstraction/APIs** provide southbound protocol plugins to communicate with network devices through OpenFlow. The southbound abstraction enables support for protocols to communicate with legacy devices, that isolate the Core of the many communication protocols.

The **Software Modularity** allows the system to be easily customized. Makes it easy to develop, debug, maintain, and upgrade ONOS as a software system by a community of developers and by the providers.

2.2.2. ONOS Intent Framework

The SDN, essentially depends on its ability to support many types of applications through Northbound Interface (NBI). The most recent SDN controllers such as ODL and ONOS mentioned previously, offer an NBI capable of sending intents and converting them, through a compiler into low-level flow rules to be installed on network devices. In this session, we will focus on the ONOS controller that has an intent-based NBI called the Intent Framework [13]. The Intents represents the highest level of abstraction. They are like virtual tunnels. The application developers can express their "intentions", through high-level policies without worrying about the specifics how each "intent" is deployed at the data plane layer.

The Intent Framework [48], is an ONOS subsystem that allows applications and operators to specify policies using a high-level abstraction or language. These policy-based policies are called Intents. The ONOS controller core accepts the intents request and converts these policies to routing rules installed on the network devices. The process of requesting and installing an intent is represented in figure 8.

When the controller receives an Intent, it is identified using two parameters, the unique IntendID and ApplicationID of the sending application. As soon as an intent request is sent by an application, it is directly forwarded to the compilation stage, where the request will be processed. At this stage, the request for intent is converted to installable intents. If an application requests an unavailable goal (for example, connectivity between unconnected devices), it can be recompiled again. After the compilation phase, it is sent to the installation phase, where an installable intent will be converted to flow rules. If successful, the process ends with the installed state, otherwise they will go to the failed state.

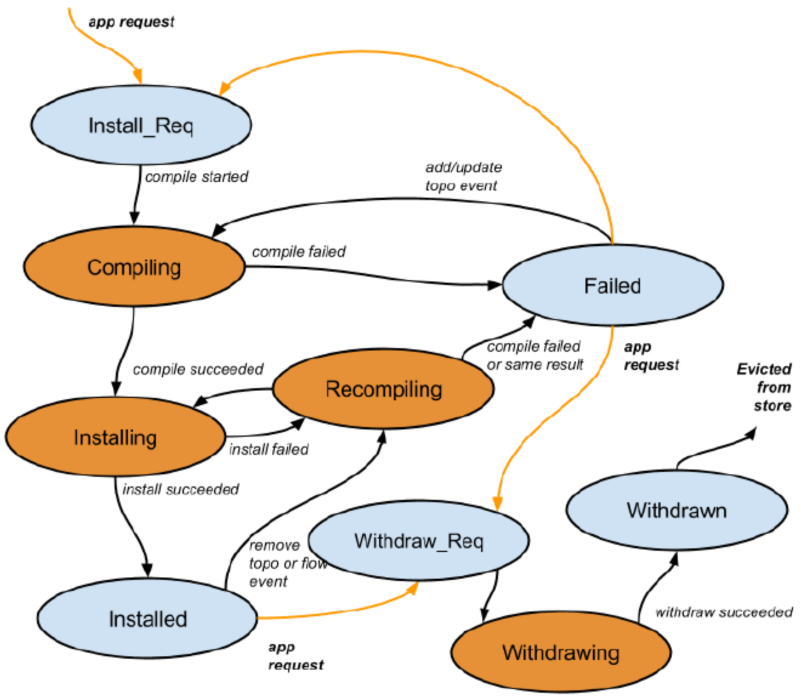


Figure 8. ONOS Intent Framework Compilation and Flow Installation [46]

There are many types of Intents, but only the ones we will use in our research work will be referenced in the next session, which will work on the SDN-IP application. Each type of Intent allows the ONOS core to translate high level policies into low level rules installed on network devices. In addition to intents for connecting hosts (for example, host-to-host intent), some intents make it possible to specify a set of constraints to limit compilation results (for example, to determine the resulting paths go through a set of nodes or to reserve a certain amount of bandwidth for each path [13].

2.2.3. SDN-IP ONOS Application

The SDN-IP application is used to exchange data among different SDN administrative domains with ONOS controllers. SDN-IP allows a software-defined network to connect to other external Internet networks using the BGP protocol. The BGP sees a SDN domain as any traditional autonomous system (AS), where within the AS, the SDN-IP works as a BGP speaker and provides means for integration with the ONOS controller. In addition, the BGP speaker uses the ONOS services to install and update the right forwarding state at the SDN data layer. Figure 9 shows the SDN-IP architecture.

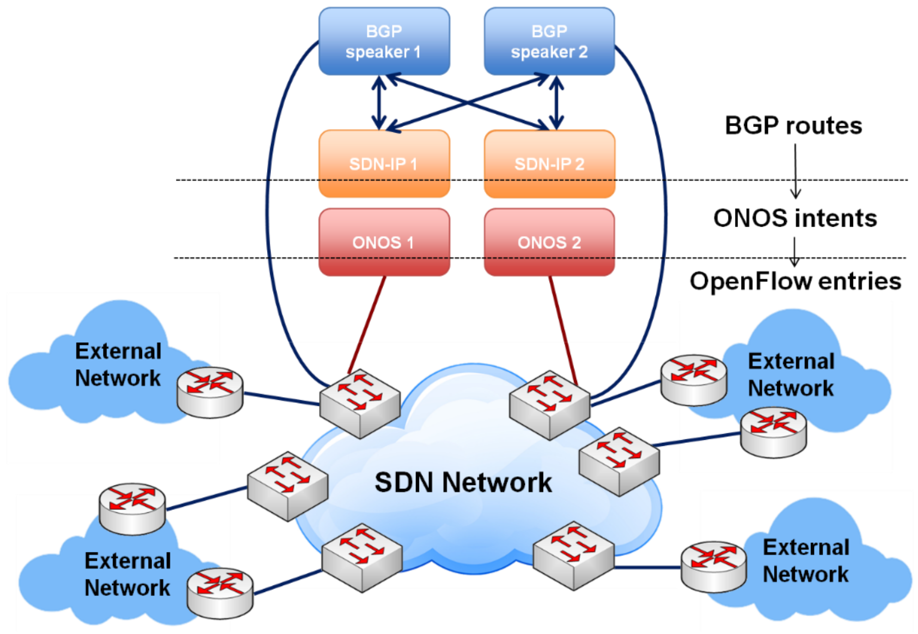


Figure 9. SDN-IP Architecture [46]

Basically, an SDN-IP network works as an autonomous transit system responsible for interconnecting distinct domains of IP networks. Each domain interfaces with the transit SDN network through its BGP speaker border routers. In transit SDN network, multiple OpenFlow switches are managed by one or more ONOS controllers for high availability and scalability running internal BGP speakers. The SDN-IP application supports one or more internal BGP speakers. The other instances will be activated if the main instance fails. However, only one instance of SDN-IP is currently active and is responsible for making the appropriate ONOS API calls to install Intents.

To exchange BGP routing information with the border routers of other domains, the system uses eBGP and iBGP to disseminate that information inside each domain. Border routers from external domains announce routes to BGP speakers that are processed according to BGP routing policies and announced to other external domains. The routes are also advertised to SDN-IP application instances that act as iBGP peers. After that, SDN-IP will choose the best routing forward according to iBGP announcements and finally these are translated in intents at the ONOS application. The intents are translated by ONOS into flow routing rules to exchange traffic between interconnected IP domains in the network. ONOS has an internal Intent Framework, as referenced in a previously session, for installing low-level flow rules on data layer network devices through high-level abstract intentions.

Basically, SDN-IP uses two types of intents, namely **Point-To-Point** and **Multi-Point-to-Single-Point** Intents [46].

The first intent type consists of one-way intent used to connect via external BGP protocol external routers to SDN BGP speakers. Each Intent connects two unique connection points in the SDN network, each containing information such as SDN DPID switch, a switch port identifier, and the BGP router / speaker MAC address.

The second intent type consists of intents used to connect hosts from external networks, i.e., it creates communication between network devices from different domains. The intent corresponds to packets destined for the IP prefix and modifies the MAC destination address to the physical address of the next hop router. In SDN-IP, one of the main advantages of relying on the Intent Framework is that the application automatically restores BGP session connectivity and transit traffic between domains, without changing any settings in the application code.

In [11], was created a SDN testbed, aimed to connect different SDN-based domains to form an SDN Internet, with a more refined method through a mechanism called SDI (Software Defined Cross-Domain Routing). According to the author, it can improve the ability to express multiple paths and inter-domain routing policies based on flow-level traffic control by combining multiple fields in the IP header. The communication is ensured by BGP protocol with floodlight controller based in WE-Bridge [43] technology, SDI uses the path vector routing algorithm and the hop-to-hop propagation mechanism, like BGP between SDN domains.

In research [10], a scenario has been implemented for the exchange of routing information between legacy autonomous systems via an SDN-based autonomous transit system. The transit

AS, is running on top of the controller the SDN-IP application developed by the ONOS group as described in the previous chapter, which acts as a BGP speaker and provides the exchange routing information between the different legacy ASs. The network is emulated using Mininet and shows a topology with an SDN-based AS, interconnected three legacies ASs.

In [49], it is discussed KREONET, which consists in a national research network in Korea. They have proposed a SDN system to evolve KREONET to a virtualized, dynamic and flexible environment. KREONET-S adopts distributed control architecture, with SDN-IP application running over the ONOS controller to provide a federated SDN service.

The authors of [9], suggest a solution called the Inter Cluster ONOS Network (ICONA) application. ICONA divides the service provider's network into multiple domains, each managed by a different cluster of ONOS instances. This application provides a network status orchestration and synchronization mechanism on all instances of the ONOS cluster. This proposal aims to improve fault tolerance and decrease the delay response to events originating in large scale networks. The biggest success story of an ICONA implementation was in the GEANT project [50]. This project is a pan-European network linking Europe's national research and education networks.

2.3. SDN for Emerging Technologies

SDN has been applied in new technology areas. The Internet of things (IoT), has attracted a great deal of attention and SDN presents methods to improve these emerging networking scenarios. The IoT is a network of physical devices and sensors with embedded technology that contains the ability to interact with the local environment. Usually, are physical devices equipped with RFID tags, actuators, wireless sensors, and / or wireless communication devices when connected to the Internet from an IoT network. The IoT network not only collects data but it also exchanges it to some servers located at remote clouds or even to some fog servers located at the network periphery. There are many areas of IoT application such as health automation, smart homes, smart transportation, environmental monitoring system, or smart grid, among others. The authors in [46], [47] present research about IoT, architecture, technologies, applications, and IoT-related issues.

Recent work has highlighted the high relevance of SDN-based systems for controlling network domains formed by IoT devices and surveyed some solution already available [9]. However, SDN solutions for wireless infrastructureless networks and, more specifically, in wireless sensor (and actuators) networks (WSNs), do not abound [10]. Unfortunately, delivering end-to-end service orchestration chains, across multiple SDN domains, for an IoT infrastructure deployment, including data collection at the cloud, edge processing, and publishing services with quality differentiation it is still at its infancy [11].

2.3.1. Related Research in QoS

The IoT keeps expanding in terms of domains, interconnected devices, data, and applications. About QoS, IoT has several issues such as availability, reliability, mobility, performance, scalability, and interoperability. Several surveys have attempted to define QoS strategies and QoS architectures, as it is necessary to ensure adequate mechanisms at each IoT layer. The authors in [1], provides an in-depth analysis of QoS issues across various types of IoT network architecture, and implement their proposal in the application of a smart city.

In [51], investigate the different types of traffic for IoT with many QoS requirements and different priority levels. An analytical model is presented for scheduling priority-based traffic on an established capacity queue system and evaluates model performance for delay-sensitive traffic against low priority traffic. The two proposals to ensure QoS are applicable in a standard IoT context. About the quality of service in SDN and OpenFlow the contributions are more limited. There are no defaults defined as IntServ and DiServ in SDN [48]. Nevertheless, it increases the flexibility of the architecture and gives network administrators and developers the freedom to implement their own QoS algorithms. Therefore, more complex QoS functions, such as Traffic Engineering (TE), Load Balancing (LB), need to be deployed as an SDN program or application, which then generates actions to be applied to the network devices through the controller.

In [12], is discussed a proposal that aims to satisfy the QoS requirements through dynamic resource allocation in SDN. The authors present a leaf-based structure, classifying flows into different priority classes. According to them, the controller must know the state of the network, including load, delay and jitter. So, they create a separate thread to periodically monitor network usage. In addition, there is a proxy to reduce frequent communication between the switches and the controller, which can generate extra traffic. The proposal was tested on Mininet and on a physical network with real hardware switches and controllers. Experimental results show that the algorithm meets the requirement for QoS streams. In addition, it is applicable for streaming video, applications, multimedia that evaluate with different QoS metrics.

The authors in [52] present an experimental assessment of bandwidth utilization of traffic between ONOS controllers. The research explores the use of a physically distributed but logically centralized controller. Scenarios with two and three controllers were analysed. In both scenarios, shows a similar behaviour, that is, an increase in linear traffic between nodes. For the three-controller scenario, it was found that bandwidth utilization was lower than the first scenario, due to the smoother consistency of Anti-Entropy, which used a random controller selection. So, according to the authors, traffic distribution between controllers contributes to QoS issues on SDN networks when deployed at a large scale in the real world.

In [53], they present two possible use cases to ensure quality of service (QoS) through ONOS Intent Monitor and Reroute (IMR) [54]. The first use case shows how a user can request path monitoring and optimization through intents created by the ONOS Intent Reactive Forwarding (IFWD) application [55]. In this case, the application code is not changed and IMR service is enabled through the ONOS CLI. So, an algorithm was deployed to maximize the throughput of flows carried by the intent created by the application. In the second use case, they used the IMR to improve the performance of the SDN-IP application, already explained in the previous chapter. Here, they used a much more advanced external routing logic based on optimization tools to minimize Maximum Network Link Utilization.

2.4. Chapter Conclusion

The literature review presented in this chapter allowed us to gain knowledge to understand SDN concepts and the current status of cross-domain communications, including a description of BGP. We study SDN architecture, its layers, interfaces, and OpenFlow protocol as the main SDN enabler. This review discussed the different architectures and type of SDN controllers, in which we concluded that approaches based on distributed multiple controllers have resolved the limitations of controller scalability, fault tolerance, and overloading.

We study the application of SDN in emergent areas such as IoT, and how we can use SDN to ensure interoperability between heterogeneous networks, providing QoS and allowing the management of the large volume of data generated by these networks.

3. System Design

As previously mentioned, this dissertation aims to study how to deploy a distributed system with multiple SDN controllers to support the end-to-end communication among the distinct networking domains. In addition, we intend to deploy a system capable of providing enough resources to meet the QoS requirements of traffic originating on heterogeneous IoT devices. In this chapter, we will present our system design and our ideas so that the goal of the proposal is achieved.

The figure 10 presents the design of the proposed system formed by three physically distributed SDN domains. Despite each domain has its own SDN controller, the network logic is centralized on the central domain controller that operates as a transit autonomous system (AS), which interconnects the different external SDN domains via border routers BGP. Therefore, the external domain controllers deal only with local events belonging to their own domain. For better insight of our prototype, we will be based on the SDN architecture already mentioned in the literature review in section 2.1.

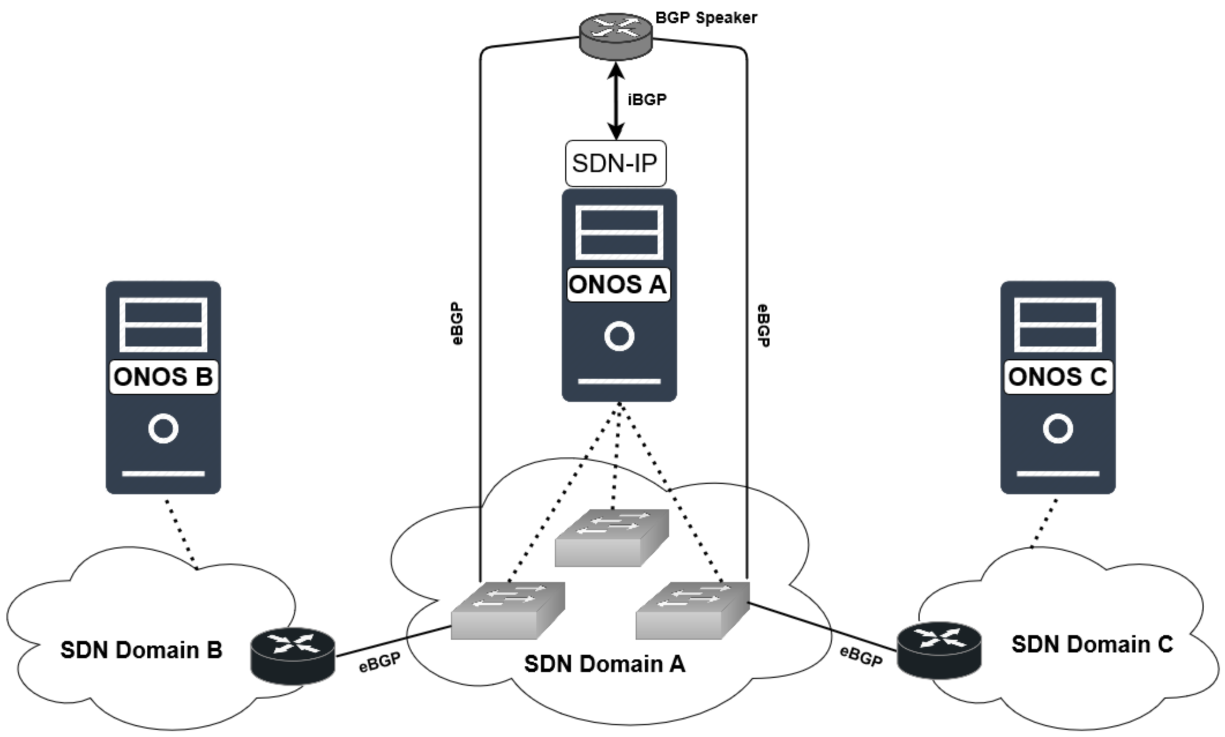


Figure 10. System Design

In the **Data Layer** will be the various network devices compatible with SDN**.** These include BGP speakers, software-based BGPs border routers, in this case Quagga, which is used to collect network information and convert it to routing updates. In addition to Hosts representing the end devices, OpenFlow Switches (OVS) are the most important device of this layer that are responsible for forwarding traffic according to policies sent from the controller. To emulate the network, Mininet emulator was the option. A custom python script called "interdomain.py" was developed to build the network topology of our project.

The logic of this implementation in the **Control Layer** will be based on the ONOS controller explained in section 2.2. The reason for choosing ONOS are:

- APIs and abstractions provided by controller that allows to add features and permissions.
- Simplicity of use due to its user-oriented software such as CLI, GUI and standard system applications.
- It is an extensible, modular and distributed SDN controller.
- Solid documentation and information sharing through the SDN application developer community.

The communication between the control layer and the data layer is done by OpenFlow protocol.

In the **Application Layer** is where the various applications will be running, and the network administrators can define mechanisms that will be activated by the controller so that network behavior is the expected. In this design, the expected network operation is the communication between distinct SDN domains. Therefore, the SDN-IP application will be running to enable the communication between SDN domains using BGP as explained in section 2.1.3. Some auxiliary ONOS applications will need to be installed (i.e. Configs and ProxyARP). These applications are required and for SDN-IP operation, allow the controller to read multiple configuration files and respond to ARP requests between the external border routers and BGP speakers. About the QoS, a script called "set_priority" flow rules will be installed through a flow POST request to the ONOS REST APIs, and these rules will be installed on switches that will allow traffic to be forwarded to different queues according to each traffic priority.

3.1. Inter-domain communication strategy

Each domain is controlled by an SDN controller located at the intermediate level of the proposed architecture (see Figure 10). The topology routing operation logic will be completely centralized in the central domain A, which will have running the applications capable of ensuring inter-domain communication. In this way, the SDN domain A, works as a transit autonomous system (AS), which interconnects different externals SDN domains that interface with the domain A, through Border Gateway Protocol (BGP) border routers. Each administrative domain contains at its data path layer several SDN-based switches controlled by the SDN controller responsible for that domain. The SDN controller should be designed for high scalability and availability. It should also support some relevant network services (e.g. routing) among the different administrative domains, enabling what is normally designated as the WAN.

As it was already mentioned, there is the SDN-IP application running above the Domain A SDN controller, which allows communication among software-defined networks, using the path vector routing protocol, BGP. Within each SDN network, there are one or more internal BGP speakers. BGP speakers can be BGP routers or software that implements BGP functionality. The operation of this system is quite simple, as it is following explained. The announced routes by Domains B and C border BGP routers are received by the BGP speakers in domain A, which are processed according to BGP processing and routing rules. The best route for each destination prefix is chosen and translated into intents by the SDN controller. An intent is like a high-level tunnel directly connecting two network devices (not necessarily direct neighbours at the data path layer). Then, each intent is converted into forwarding rules. Afterwards, these rules are transferred from the SDN controller to each network device (e.g. switch) involved in the initial BGP routing path.

As already mentioned in section 2.2.1, there are several types of intent. SDN-IP installs two types of intent as shown in Chapter 4. Point to Point Intents ensure the connection between BGP external speakers and BGP internal nodes and creates a Multi-Point to Single-Point intent, allowing communication between devices from different external domains. The SDN-IP application has the great advantage that in case of topology change, it will automatically restore BGP session connectivity and transit traffic between network domains without having to change application code.

3.2. QoS Strategy

With QoS we aim to guarantee end users with enough bandwidth for the best possible network performance, according to applications requirements. There are many ways in which such guarantees can be obtained. One of its precepts is that, when requested, traffic should not be treated equally, i.e., prioritizing bandwidth is a viable alternative to ensure quality of service through priority access.

Our goal is to provide enough resources to meet traffic QoS requirements from heterogeneous IoT devices. Therefore, our proposal is to create a system that allows differentiating different types of traffic and installing OpenFlow rules on switches according to different priority levels. The priorities are managed through of different virtual output queues and meter tables will be defined in the OpenFlow specification in version 1.3 as shown in figure 11.

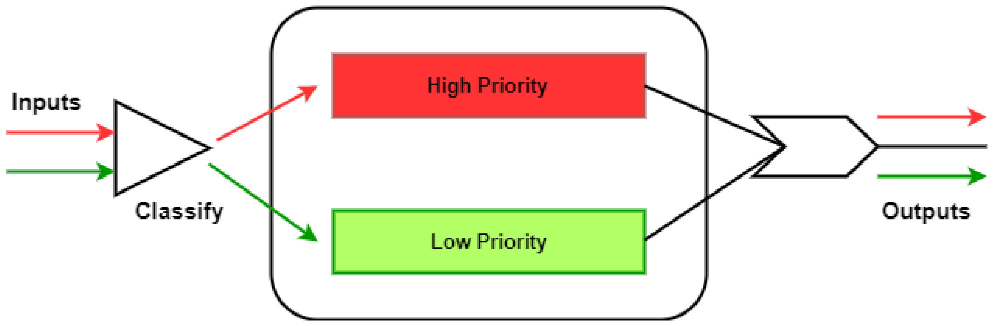


Figure 11. Queues Management

In this OpenFlow version, the concept of meter tables was introduced to achieve more granular QoS in OpenFlow networks. The queues manage the traffic exit rate and meter tables are used to monitor the traffic rate before the exit. The traffic will be prioritized through a script called "set_priority". When the script is executed, it should be possible to put the priority in high or low, and in this way, we can protect the traffic and offer guarantees on the quality of service.

Figure 12 shows the flowchart of the possible actions that when the script is executed will be able to do.

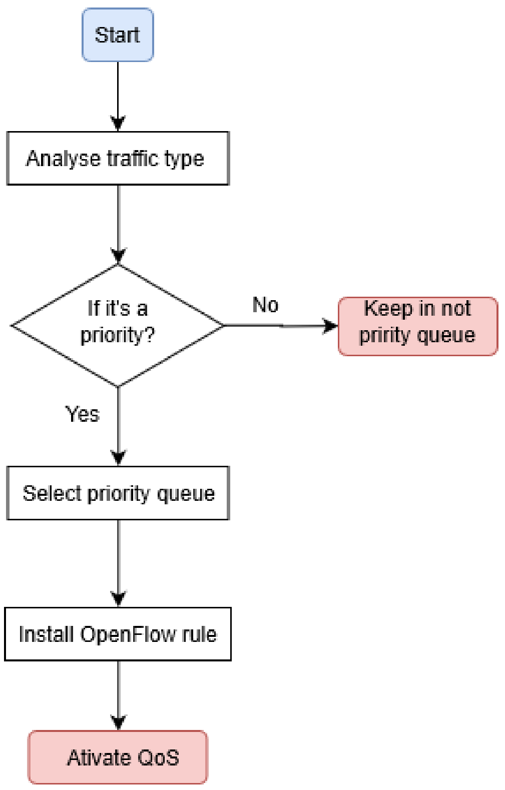


Figure 12. Traffic priority Flowchart

The first task is to analyse the incoming traffic, and this analysis is done through the source and destination IPs. If these IPs match the traffic we define as a priority, let's move this traffic to the high priority queue and the OpenFlow rule will be installed on the switch.

About an IoT context, we propose the deployment of a video vigilance system in public environments for security reasons, and this system is being monitored by a remote entity (e.g. Police). Basically, it will consist of a mixed network with vigilance cameras equipped with motion sensors and traditional network devices such as users' computers that generate another type of traffic. The cameras with motion sensors will be emulated devices, but that can be applied with real sensors.

In this case, if the motion sensor detects any motion, the system must be able to recognize incoming traffic and install OpenFlow rules in the high priority queue to stream uninterrupted quality video and other competing traffic remained in the lower priority queue for not disturb the transmission of the image. If there is no more movement, the system should move traffic to the lowest priority queue to leave the queue free for new priority situations.

3.3. Chapter Conclusions

In this chapter we present our proposal and all we idealized for this work. We present the general system design applicable an IoT context of a network formed by multiple distributed domains and the strategy used to ensure the inter-domain communication. Basically the strategy was the use of SDN-IP application which allows the externals SDN domains exchange network information via BGP protocol. This application will be installed on ONOS controller of the central transit domain that will make all the routing management.

About QoS, we think of a scenario based on different traffic priority queues. The system should analyse incoming traffic and install OpenFlow Rules according to each priority level. The scenario that will be emulated, consists on a video vigilance system to monitoring the public road through of cameras with move sensors. We will prioritize the traffic from the cameras when they detect some movement in order to transmit without interruption the image. The proposal presented in this chapter will be implemented in practice in the next chapter 4.

4. System Deployment

In this chapter, we present how our project was deployed. A detailed description of the deployment strategy of our testbed topology and its configurations will be given. A description of QoS implementation strategies will be described simulating a scenario that may be applicable in an IoT context. Finally, a description of the test scenarios and the results obtained on the fulfilment or not of the objectives of this dissertation will be performed.

4.1. Technologies and Tools

Table 4 lists the main technologies and tools that we have used in the proposed system.

Table 4. Technologies and Tools

Category	Software/ technology
Northbound Application	SDN-IP
SDN Controller	ONOS 1.15.0
Software Switch	OpenvSwitch 2.9.2
Southbound Communication	OpenFlow 1.3
Inter-domain Protocol	BGP
Network Emulator	Mininet
BGP Software	Quagga
Traffic Analyser	Wireshark, Tcpdump
Virtual Hypervisor	Oracle Virtual Box
VM Operating System	Ubuntu 18.04
Traffic Generator and Measurement	Iperf
Video transmitter Application	VLC

To simulate the system environment, we use a virtual machine, Oracle Virtual Box platform with Ubuntu Operating System in version 18.04. To emulate network topology an their devices like OVS, hosts and create their links in the data layer, we use the Mininet emulator. This emulator provides a virtual network on a computer, similar to a complex real-world network. The network topology was created through Python language code. The Mininet will also be used with changes in the switch's parings to make our QoS experiences. Still at the data layer, routers with the Quagga software will be configured to enable BGP on the controllers. We use the ONOS controller in version 1.15.0 to manage each SDN domain. The OpenFlow protocol will be used for communication between the controller and the data. In the top layer will be running the SDN-IP application, the previously explained, to ensure communication between SDN domains via BGP. Other software implementations were used, Wireshark and Tcpdump to analyse network traffic and Iperf to test and measure bandwidth, perform packet injection to measure the performance of our network topology. It will also be used to test the effects of QoS settings.

4.2. Test Topology Setup

The general idea is to deploy a scenario where is provided end-to-end communication between different SDN domains across multiple paths to meet QoS requirements. A virtual network topology was built to meet these conditions and is presented in figure 13.

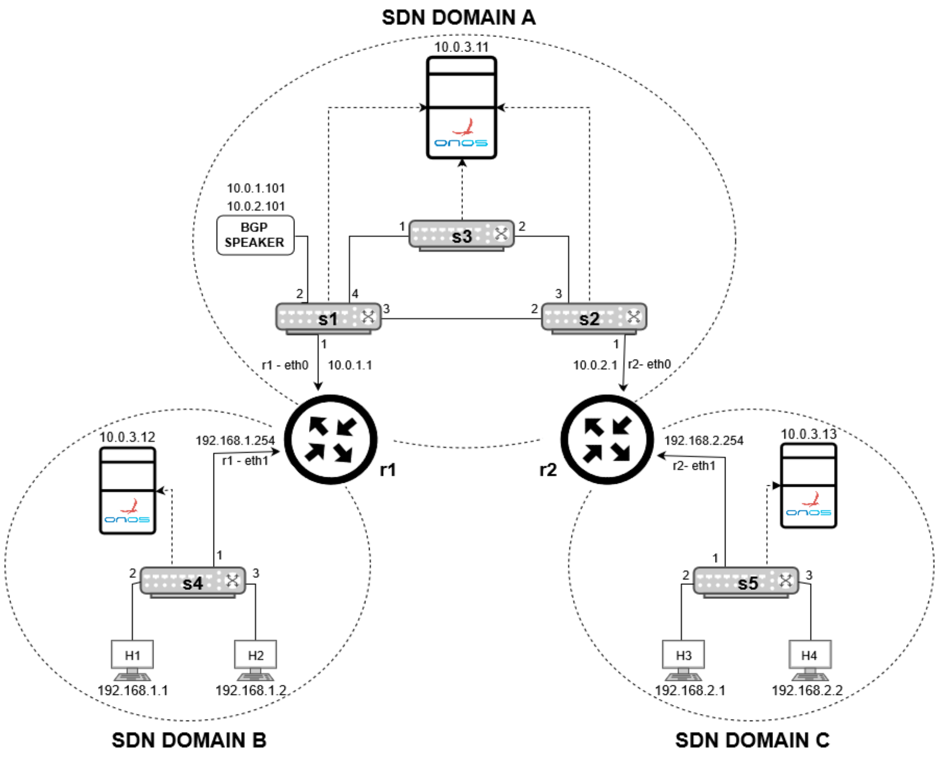


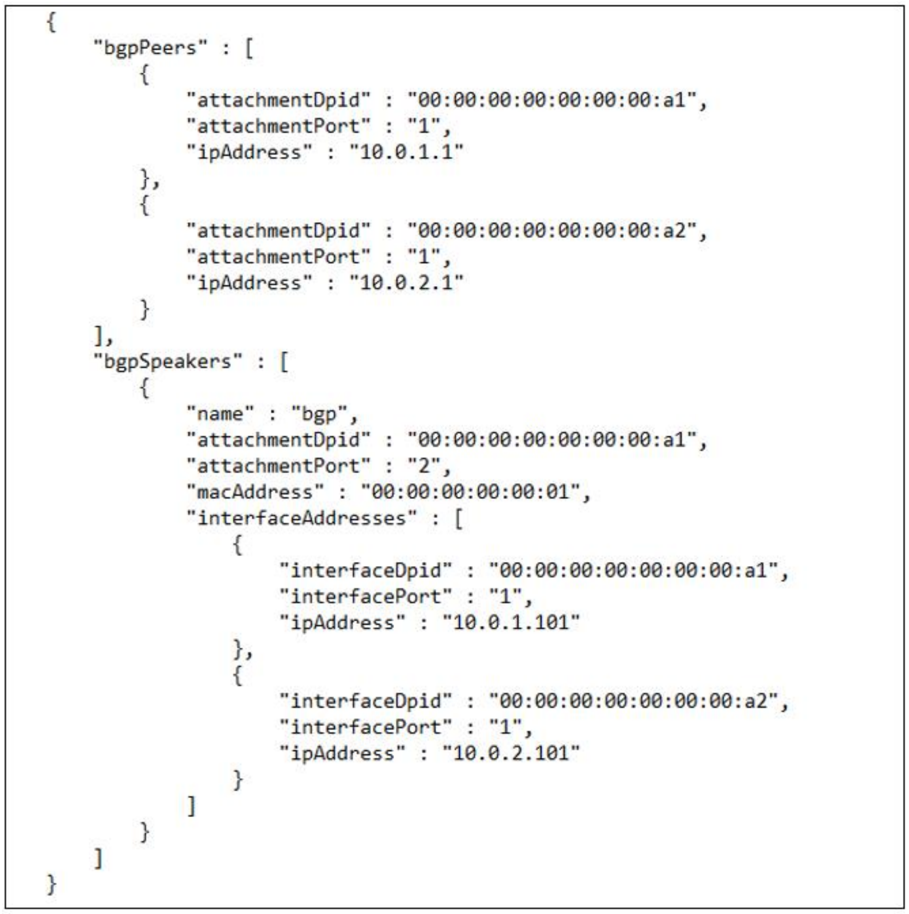
Figure 13. Inter-domain Testbed scenario

The topology has three SDN domains, each one controlled by its own ONOS controller. The central domain (A) acts as the transit domain or root controller where has all the centralized logic, responsible for interconnecting the remaining external networks. Each external network, in this case B, C are considered different domains that interface with the central domain (A) through border routers (r1 and r2), which runs Quagga. BGP configuration were made for each router. In the central domain (A), there is an SDN controller with an SDN-IP application running on its top that learns BGP routes to destination prefixes previously announced by the BGP routers of the network topology. After, the learning phase, the SDN controller of domain A translates each learned BGP route to SDN intents. Then, the same SDN controller converts each intent in to several flow rules which are then transferred from the SDN controller to the data plane switches, using the OpenFlow protocol. These switches are the ones previously selected by the SDN controller to support a specific BGP route path across the central domain, i.e. domain A.

In terms of physical equipment present in each external domain, there are one Open flow switch and two hosts connected to it, as shown in the figure 13, with their IP addresses and linked interfaces. In the central domain A, are three OpenFlow switches connected to the ONOS controller. S1 and S2 are connected to the border routers through interface 1. In addition, the internal BGP Speaker is connected to Switch 1 via interface 2, which will forward routing information listened of BGP border routers to the controller with the SDN-IP application installed. The Switch 3 (s3) has no direct influence on inter domain communication, it will serve as an alternate path in case of primary link failure and for QoS implementation.

4.2.1. Routers and BGP Speaker Configurations

For SDN-IP to know where BGP internal and external BGP speakers are, so that it can respond to ARP correctly and program connectivity for BGP traffic, the BGP routers and speakers must be configured accordingly. For each pairing session configured, there will be a pair of IP addresses. The first is the address of the external pair and the other address is used by the BGP speaker, usually these addresses must be on the same subnet (e.g. IP 10.0.1.101 of the speaker is associated with domain B 10.0.1.1). The configurations must be placed directly in a file with extension ". JSON" to be recognized by the SDN-IP application. In our scenario, the configurations were made manually and saved in the "network-cfg.json" file (see figure 14), because the SDN-IP currently only supports reading the static configuration.



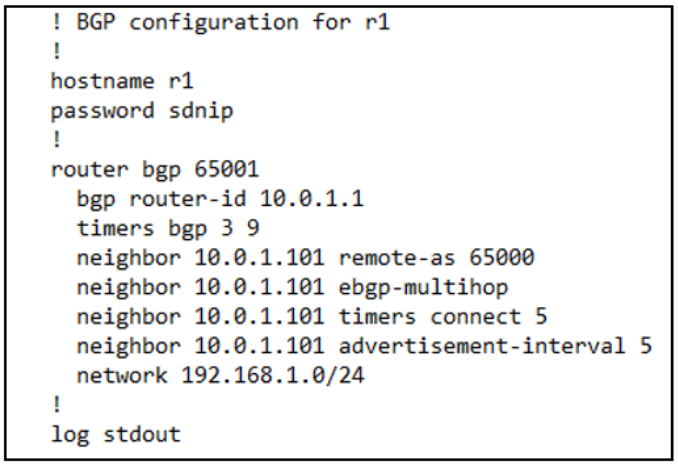
```
{
    "bgpPeers" : [
        {
            "attachmentDpid" : "00:00:00:00:00:00:00:a1",
            "attachmentPort" : "1",
            "ipAddress" : "10.0.1.1"
        },
        {
            "attachmentDpid" : "00:00:00:00:00:00:00:a2",
            "attachmentPort" : "1",
            "ipAddress" : "10.0.2.1"
        }
    ],
    "bgpSpeakers" : [
        {
            "name" : "bgp",
            "attachmentDpid" : "00:00:00:00:00:00:00:a1",
            "attachmentPort" : "2",
            "macAddress" : "00:00:00:00:00:01",
            "interfaceAddresses" : [
                {
                    "interfaceDpid" : "00:00:00:00:00:00:00:a1",
                    "interfacePort" : "1",
                    "ipAddress" : "10.0.1.101"
                },
                {
                    "interfaceDpid" : "00:00:00:00:00:00:00:a2",
                    "interfacePort" : "1",
                    "ipAddress" : "10.0.2.101"
                }
            ]
        }
    ]
}
```

Figure 14. Netwok-cfg.json File

In this type of files, the configurations are divided into two sections as shown in Figure 14, from our configuration file:

- **Port configuration:** we configure the ports on the switch interfaces that connect to the external BGP border routers. In our topology, switches 1 and 2 were configured, connected to the r1 and r2 routers. In addition to the port number, the IP address and MAC address has been specified.
- **BGP configuration:** we configure the internal BGP speakers of our SDN network topology. In this section, we add a connection point to our BGP speaker, that is, we specify the switch in which the BGP speaker is connected. The list of pairs with the addresses of the BGP speaker is listening was also specified.

To announce the routes, we have to make the BGP configurations separately. As the BGP router is a Quagga process, then we configure each router in a file (quagga.config). It will be instantiated in the python script of our network topology the file "quagga-sdn.config" (domain A quagga configurations), which have all configurations of routers and speakers. Figure 15 shows the router one (r1) configurations. The router two (r2) configuration is similar like r1.

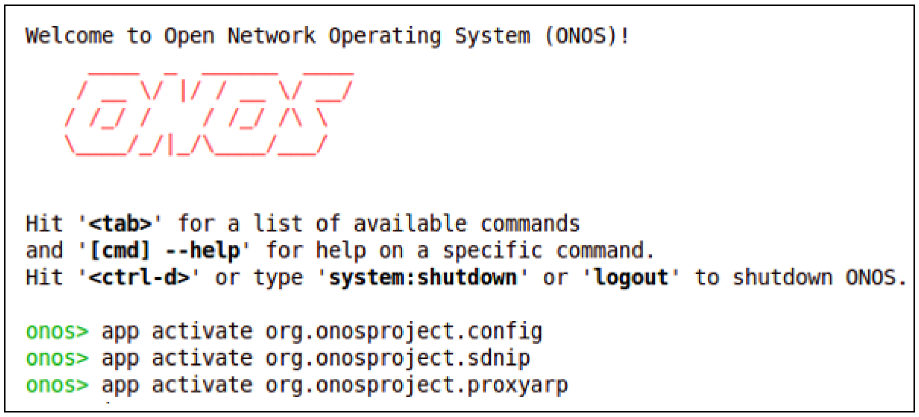


```
! BGP configuration for r1
!
hostname r1
password sdnip
!
router bgp 65001
  bgp router-id 10.0.1.1
  timers bgp 3 9
  neighbor 10.0.1.101 remote-as 65000
  neighbor 10.0.1.101 ebgp-multihop
  neighbor 10.0.1.101 timers connect 5
  neighbor 10.0.1.101 advertisement-interval 5
  network 192.168.1.0/24
!
log stdout
```

Figure 15. Quagga configuration

4.2.2. Start-up SDN-IP on ONOS

Firstly, we must start the ONOS controller system and for that we start using Docker Images. This, allows in safely way, run isolated applications in a container, packed with all its dependencies and libraries. After ONOS is running, we must install additional applications on which SDN-IP depends. These applications allow ONOS to read BGP configuration files and respond to ARP requests between external border routers and BGP internal speakers. The commands to install the applications are in Figure 16.



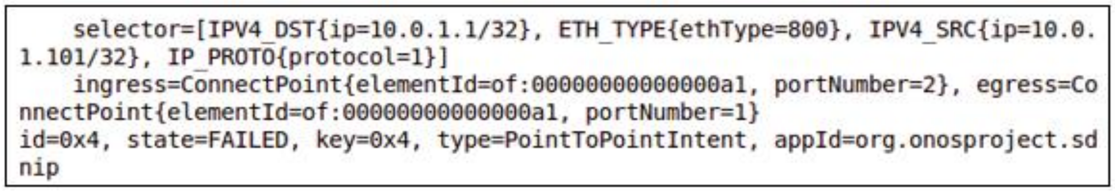
```
Welcome to Open Network Operating System (ONOS)!

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown ONOS.

onos> app activate org.onosproject.config
onos> app activate org.onosproject.sdnip
onos> app activate org.onosproject.proxyarp
```

Figure 16. ONOS CLI with SDN-IP application Installed

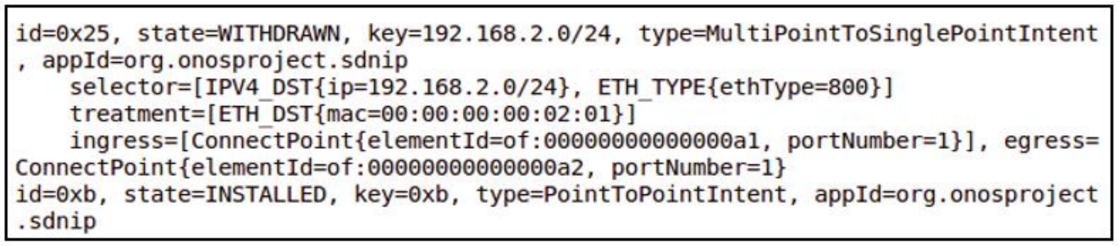
Next, let's start our testbed topology saved in a python file called interdomain.py and connect it to our SDN controller. Many events happen as soon as the SDN-IP application is installed on the top of ONOS and paired with the network like installing Intents. Figure 17 and 18 shows examples of the intents which the application has installed to operate on our test topology.



```
    selector=[IPV4_DST{ip=10.0.1.1/32}, ETH_TYPE{ethType=800}, IPV4_SRC{ip=10.0.
1.101/32}, IP_PROTO{protocol=1}]
    ingress=ConnectPoint{elementId=of:00000000000000a1, portNumber=2}, egress=Co
nnectPoint{elementId=of:00000000000000a1, portNumber=1}
id=0x4, state=FAILED, key=0x4, type=PointToPointIntent, appId=org.onosproject.sd
nip
```

Figure 17. Point to Point Intent

This Figure 17 shows the **Point to Point Intents** installed, that allows the border BGP routers to communicate with our internal BGP speaker.

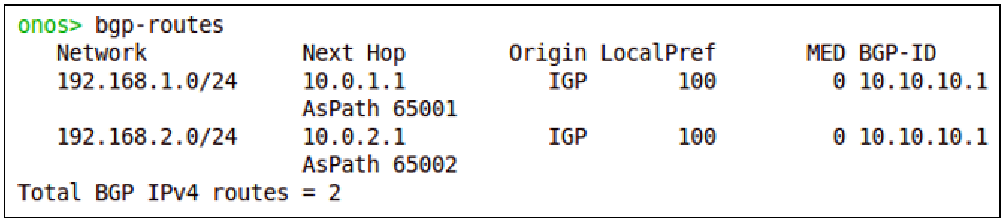


```
id=0x25, state=WITHDRAWN, key=192.168.2.0/24, type=MultiPointToSinglePointIntent
, appId=org.onosproject.sdnip
    selector=[IPV4_DST{ip=192.168.2.0/24}, ETH_TYPE{ethType=800}]
    treatment=[ETH_DST{mac=00:00:00:00:02:01}]
    ingress=[ConnectPoint{elementId=of:00000000000000a1, portNumber=1}], egress=
ConnectPoint{elementId=of:00000000000000a2, portNumber=1}
id=0xb, state=INSTALLED, key=0xb, type=PointToPointIntent, appId=org.onosproject
.sdnip
```

Figure 18. MultiPoint to SinglePoint Intent

Based on the exchange of information obtained from the first intent, it will allow external BGP routers to relay routes capable of forwarding to SDN-IP using **Multi Point to Single Point Intents** and thus communicate between external domains (B and C).

ONOS received the possible routes and converted them to rules on the switches using the intent API. Figure 19 shows the best routes received from BGP peers, including BGP specific information.



```
onos> bgp-routes
   Network            Next Hop         Origin LocalPref       MED BGP-ID
   192.168.1.0/24     10.0.1.1            IGP       100         0 10.10.10.1
                      AsPath 65001
   192.168.2.0/24     10.0.2.1            IGP       100         0 10.10.10.1
                      AsPath 65002
Total BGP IPv4 routes = 2
```

Figure 19. BGP Routes

Note that the intents in our topology are for routing issues only as we will show in the testing section, it has no impact on QoS issues whose implementation will be demonstrated in the system evaluation section 4.4.

4.3. QoS Deployment

Our goal, as described in the previous chapter in section 3.2, is to provide a mechanism that makes traffic prioritization decisions to test programmable network QoS through the OpenFlow 1.3 queues.

We consider a scenario, consisting of vigilance cameras equipped with motion sensors transmitting RTP video flow and generic user computers generating UDP traffic. Initially the traffic is all going in the same row and if the motion sensor detects any movement, the surveillance cameras should have higher priority passed to another queue and consequently transmit the highest quality image without interference generated by traffic from other devices. Similarly, if there is no movement the cameras transmit normal quality image and continue to divide the bandwidth with the generic traffic to leave the bandwidth free for the priority traffic.

Figure 20 is a testbed called "QoS_topology.py" with dummy emulated IoT devices (sensor cameras), end hosts, OpenFlow Switches in version 2.9.2, and ONOS controller that applies traffic prioritization rules to active flows. We limit links through TCLinks (Mininet fixed bandwidth emulated links) between S1 to S2, between hosts and S2 with 10 Mbits / s bandwidth.

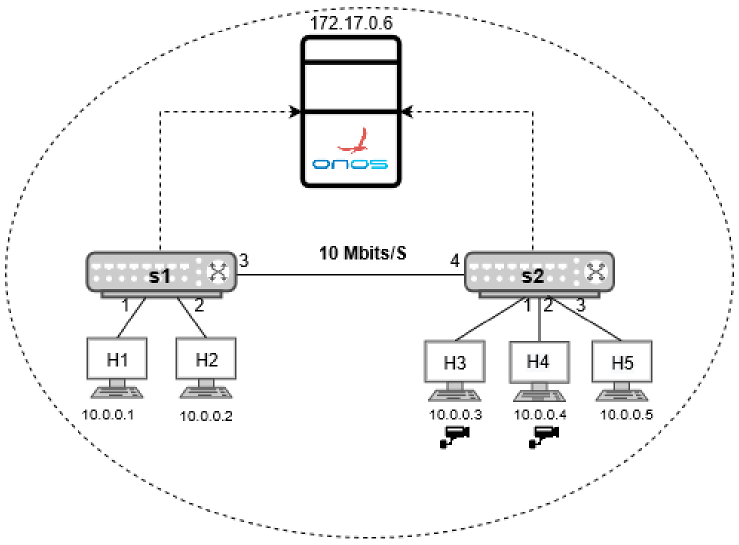


Figure 20. QoS Testbed

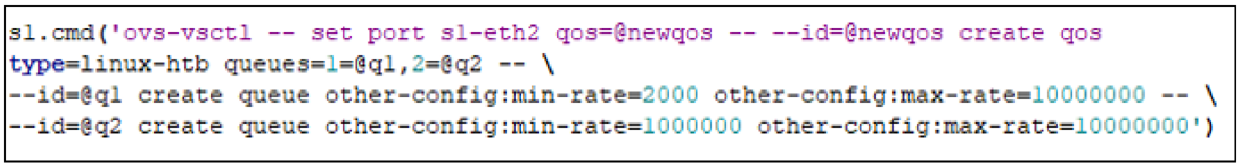
4.3.1. Queues and flows Configurations

For our system, we created two queues with different priority levels, as shown in table 5 with the appropriate rate for each one.

Table 5. Priority Level Queues

Queue	Priority	Rate (Min; Max)
Q1	Low	2 Kbits/s; 10 Mbits/s
Q2	High	1 Mbits/s; 10 Mbits/s

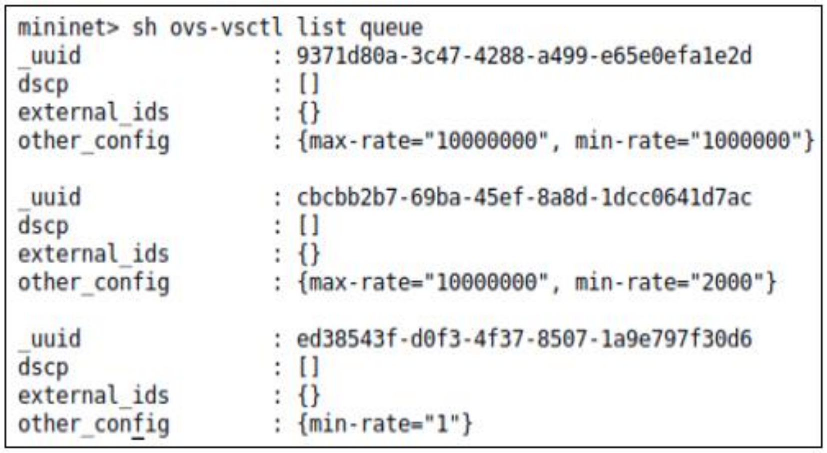
The queues were created using the ovs-vsctl command in OVS. This command creates an entry in OVSDB and implements it on the switch using Linux TC. In our case the queues were directly created in our Mininet script as shown in figure 21.



```
s1.cmd('ovs-vsctl -- set port s1-eth2 qos=@newqos -- --id=@newqos create qos
type=linux-htb queues=1=@q1,2=@q2 -- \
--id=@q1 create queue other-config:min-rate=2000 other-config:max-rate=10000000 -- \
--id=@q2 create queue other-config:min-rate=1000000 other-config:max-rate=10000000')
```

Figure 21. Queues configuration

The figure above creates a QoS and queues on switch S1 on port eth2. The queue 1 is for low and 2 for high priority, activated as soon as there is any motion detected by the sensor and the vigilance camera in question will transmit images with full bandwidth. Figure 22 shows the queues created in Mininet as soon as the topology is started.



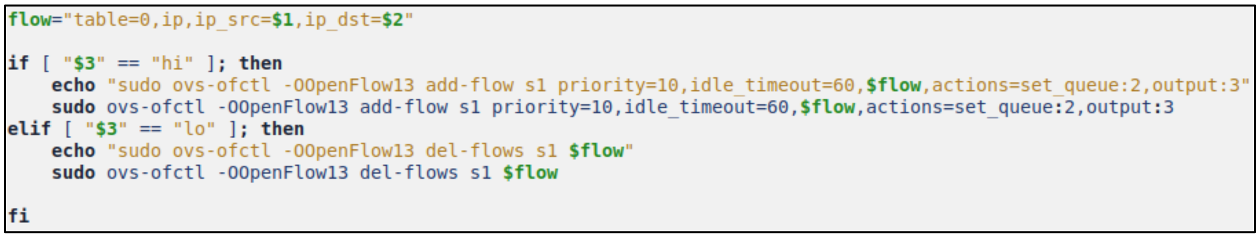
```
mininet> sh ovs-vsctl list queue
_uuid               : 9371d80a-3c47-4288-a499-e65e0efa1e2d
dscp                : []
external_ids        : {}
other_config        : {max-rate="10000000", min-rate="1000000"}

_uuid               : cbcbb2b7-69ba-45ef-8a8d-1dcc0641d7ac
dscp                : []
external_ids        : {}
other_config        : {max-rate="10000000", min-rate="2000"}

_uuid               : ed38543f-d0f3-4f37-8507-1a9e797f30d6
dscp                : []
external_ids        : {}
other_config        : {min-rate="1"}
```

Figure 22. Queues validation at Mininet CLI

To simulate a motion detection, we implemented a script called "set_priority" shown in figure 23. This script differentiates the data traffic through source and destination IP and installs OpenFlow rules on switches according to each priority queue. It can set traffic to (hi) for high priority and (lo) return to low priority as soon as there is no movement.



```
flow="table=0,ip,ip_src=$1,ip_dst=$2"

if [ "$3" == "hi" ]; then
    echo "sudo ovs-ofctl -OOpenFlow13 add-flow s1 priority=10,idle_timeout=60,$flow,actions=set_queue:2,output:3"
    sudo ovs-ofctl -OOpenFlow13 add-flow s1 priority=10,idle_timeout=60,$flow,actions=set_queue:2,output:3
elif [ "$3" == "lo" ]; then
    echo "sudo ovs-ofctl -OOpenFlow13 del-flows s1 $flow"
    sudo ovs-ofctl -OOpenFlow13 del-flows s1 $flow

fi
```

Figure 23. OpenFlow rules in set_priority script

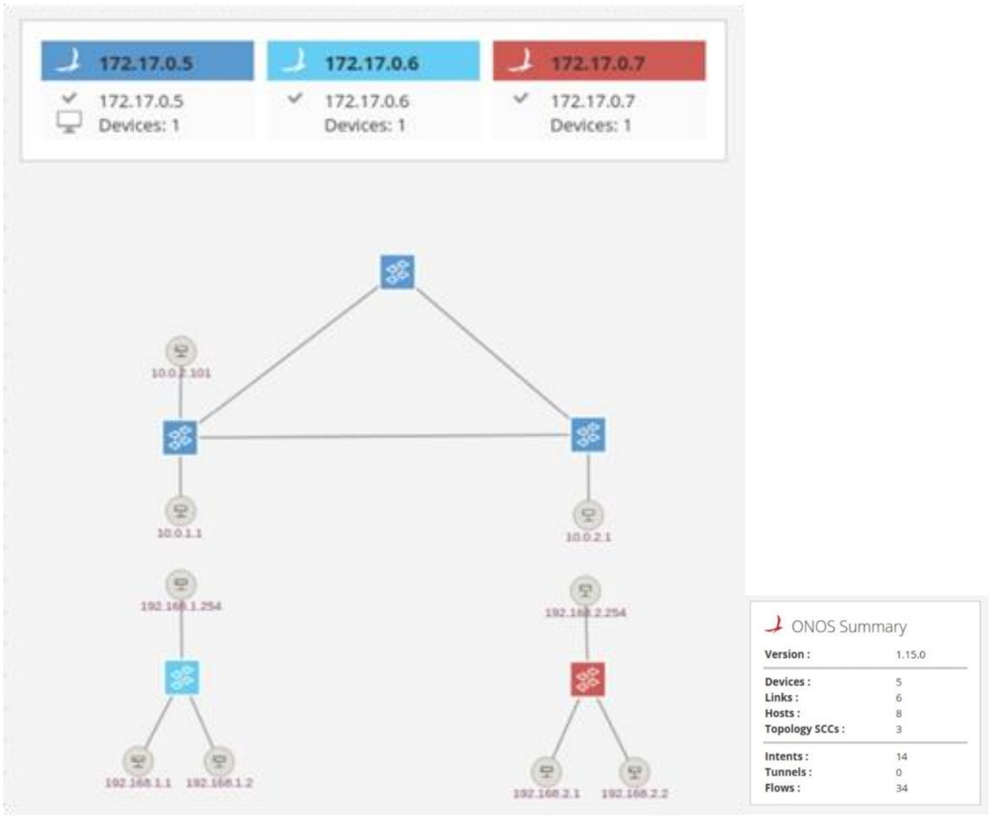
4.4. System Evaluation

In this section, we present the system evaluation and demonstration of the results of the obtained tests. In this case, we will perform the following tests:

- SDN Inter-Domain Communication Test;
- Controller Failure Test;
- Link Failure Test;
- QoS Data Rate Test.

4.4.1. SDN Inter-Domain Communication Test

In Figure 24 shows the web ONOS GUI. The figure shows that our topology being controlled by SDN ONOS controllers and the summary information.

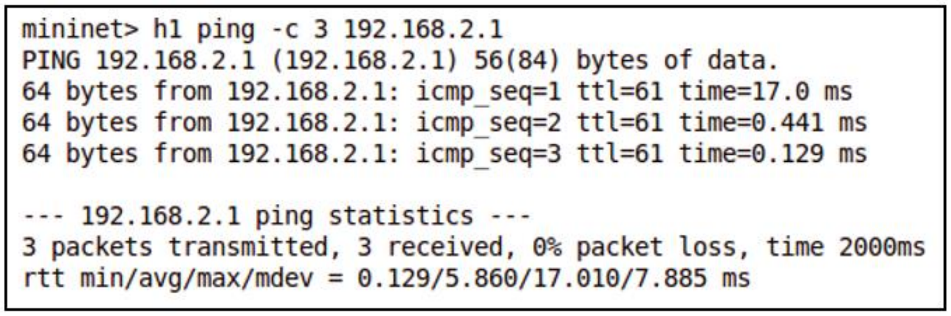


ONOS Summary

Version :	1.15.0
Devices :	5
Links :	6
Hosts :	8
Topology SCCs :	3
Intents :	14
Tunnels :	0
Flows :	34

Figure 24. The topology at ONOS GUI

There are three SDN controllers, each represented by a colour to differentiate the domain in which it manages. The first SDN controller (172.17.0.5) controls the transit domain which contain three central switches . The second SDN controller (172.17.0.6), represented by the light blue colour, manages the left domain, which contains a single switch, interconnecting two terminal hosts (for example, h1 with IP address 192.168.1.1/24). The same happen with the SDN domain (172.17.0.7) represented by red colour on the right, which contains a switch with two hosts (h3 and h4). So, we have a physically distributed system with multiple controllers, each managing their own domain autonomously, but the central domain managing the inter-domain logic. We have validated our system using ICMP traffic originated at host h1 (192.168.1.1) and with the destination host h3 (192.168.2.1). Figure 25 shows the successful PING between domain B host 1 and domain C host 3.



```
mininet> h1 ping -c 3 192.168.2.1
PING 192.168.2.1 (192.168.2.1) 56(84) bytes of data.
64 bytes from 192.168.2.1: icmp_seq=1 ttl=61 time=17.0 ms
64 bytes from 192.168.2.1: icmp_seq=2 ttl=61 time=0.441 ms
64 bytes from 192.168.2.1: icmp_seq=3 ttl=61 time=0.129 ms

--- 192.168.2.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.129/5.860/17.010/7.885 ms
```

Figure 25. Connectivity test

4.4.2. Controller Failure Test

As in our system the controllers are presented in a cluster, let's see how ONOS reacts to the failure of one of the controllers. In the ONOS CLI we will shut down one of the instance controllers (172.17.0.7) in order to inactivate the controller. Then, we verify that the powered down controller will turn grey to indicate that the node is not accessible, as shown in figure 26.

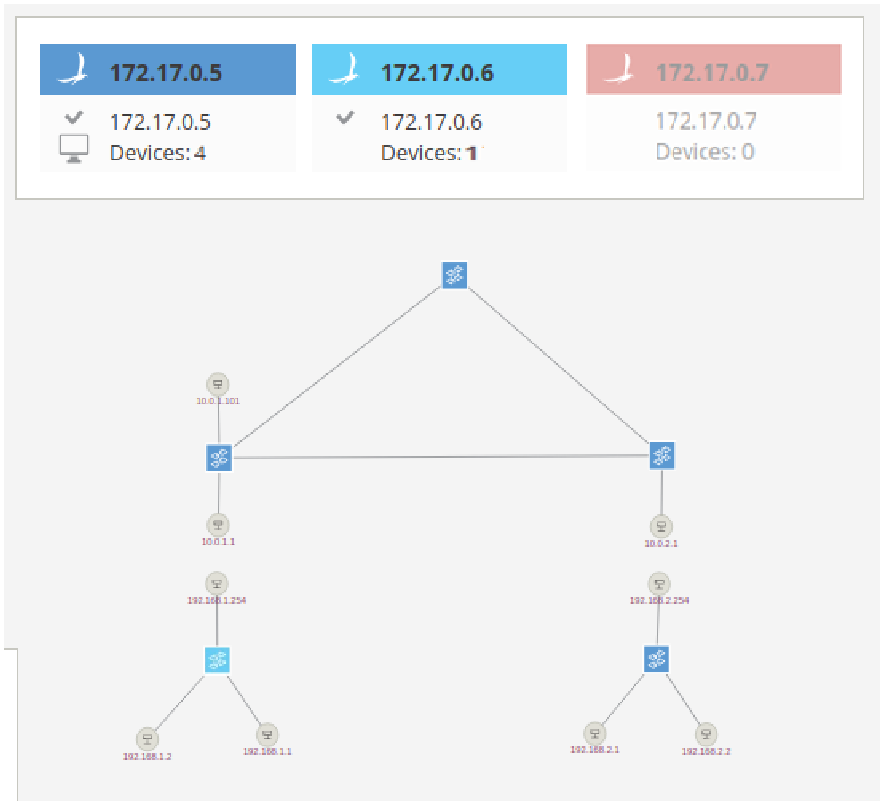


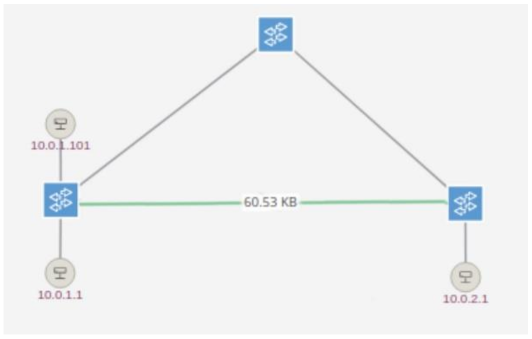
Figure 26. Failure of controller in a cluster

4.4.3. Link Failure Test

The system failure detection is a very important aspect of ensuring fault tolerance in large scale distributed systems. Our solution should be able to detect link failures. In order to evaluate and simulate a failure, we first must analyse which path the traffic goes. In our case, if the SDN controller detects a link failure, it can quickly and effectively divert traffic to an alternate path to ensure the service until the primary link is operational again. Our main goal is to reduce the time required to detect a failure.

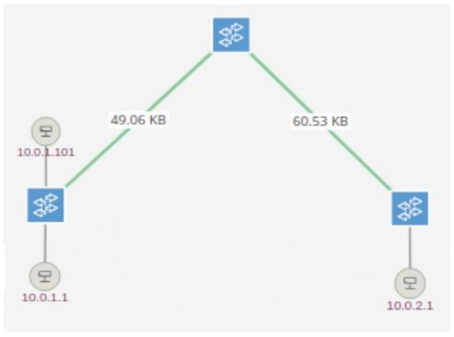
Table 6 shows the results of catches made by Tcpdump. At this time, the topology was operating without any failure and the used routing path between h1 and h3 was that involving the switches s1 and s2 of the transit Domain A (s1-eth3, s2-eth2). One can also note that the initial TTL of the ICMP Request is 64 (h1-eth0) is decremented down to 61 (h2-eth0), meaning that message has traversed three routers (i.e. r1, BGP speaker, r2) on its way from the source node to the destination node. Through the shortcut “A” in the ONOS GUI it is also possible to see the traffic path and its speed.

Table 6. Captured ICMP Request message from h1 to h3, s1-eth3 up

Topology	
H1 - eth0	10:58:36.709985 IP (tos 0x0, ttl 64, id 13117, offset 0, flags [DF], proto ICMP (1), length 84) 192.168.2.1 > 192.168.2.1: ICMP echo request, id 12153, seq 1, length 64
S2 - eth2	10:58:36.741345 IP (tos 0x0, ttl 62, id 13117, offset 0, flags [DF], proto ICMP (1), length 84) 192.168.2.1 > 192.168.2.1: ICMP echo replay, id 13117, seq 1, length 64
S3 - eth1	
H3 - eth0	10:58:36.756439 IP (tos 0x0, ttl 61, id 3307, offset 0, flag [DF], proto ICMP (1), length 84) 192.168.2.1 > 192.168.2.1: ICMP echo replay, id 12153, seq 1, length 64

Then, we turned off the link between s1 and s2, forcing the link to fail. The traffic captured from this second test is shown in Table 7. Analysing these results, one can conclude that the SDN-IP/BGP proposal has detected the topology failure and automatically has successfully selected an alternative path through the transit Domain A (s1-eth4, s3-eth1, s3-eth2, s2-eth3).

Table 7. Captured ICMP Request message from h1 to h2, s1-eth2 down

Topology	
H1 - eth0	11:05:34.305256 IP (tos 0x0, ttl 64, id 61729, offset 0, flags [DF], proto ICMP (1), length 84) 192.168.2.1 > 192.168.2.1: ICMP echo request, id 11341, seq 1, length 64
S2 - eth2	
S3 - eth1	11:05:34.327286 IP (tos 0x0, ttl 62, id 61729, offset 0, flags [DF], proto ICMP (1), length 84) 192.168.2.1 > 192.168.2.1: ICMP echo replay, id 12361, seq 1, length 64
H3 - eth0	11:05:34.329246 IP (tos 0x0, ttl 61, id 61729, offset 0, flags [DF], proto ICMP (1), length 84) 192.168.2.1 > 192.168.2.1: ICMP echo request, id 11341, seq 1, length 64

We have validated the SDN-IP/BGP integration proposal, using a scenario where after we have failed a network interface in use by a specific routing path, that failure was detected and corrected in an adequate way by choosing an alternative path, avoiding the disruption of the network operation. Figure 27 shows the graph referring to the system reaction in case of a link failure. The system has been operating without fail for up to 28 seconds. When the link failure occurs there is a slight drop in debt that quickly settles after three seconds but at no time is the system unavailable due to this link failure.

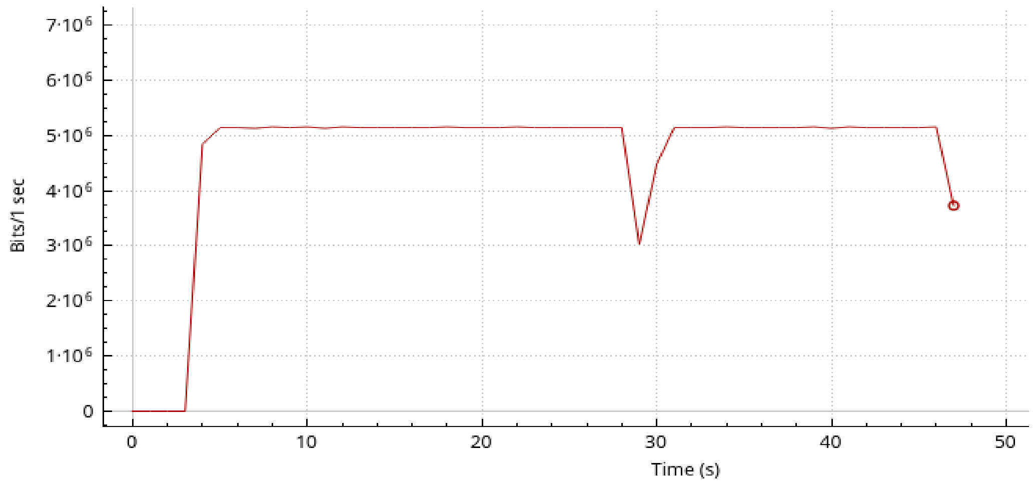


Figure 27. Link failure results

4.5. QoS Data Rate Test

In this section, we will validate our QoS deployment tests. Essentially, we will test the reaction of our system at three interval moments as shown in the figure 28.

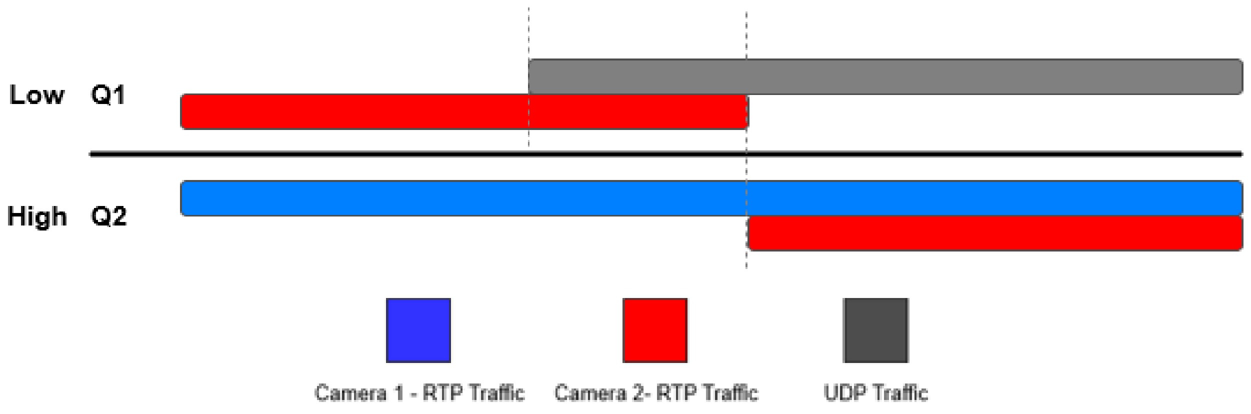


Figure 28. QoS moments tests

The first moment is when the traffic generated by camera 1 and camera 2 transmit video without interference from any competing traffic. The second moment is when we inject concurrent traffic to disrupt camera transmission and the third moment is when we apply the QoS mechanism to protect the video traffic transmitted by the cameras.

From the beginning, traffic from camera 1 will be in the priority queue, so that we can analyze the reaction of video from camera 2 at all times of our test.

When the topology “QoS_topology.py” is started, five terminals will be opened, two of them are VLC terminals that will transmit a video simulating a vigilance camera and we will use the video, “video_test.mp4” as proof of concept. In terminal 1 which we named "RTP Video Emitter" we started broadcasting Video Stream with the following command: ./send_video.sh video_test.mp4. Figure 29 shows the video be transmitted on both servers and simulates the vigilance camera transmission.

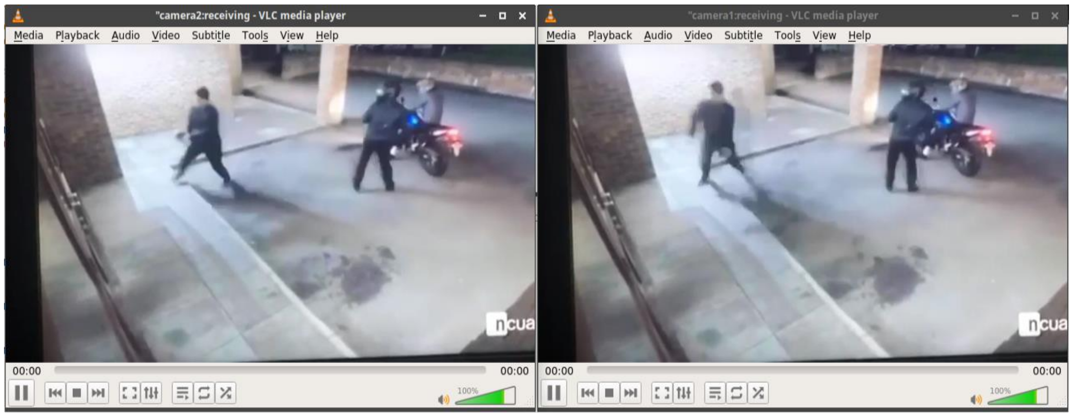
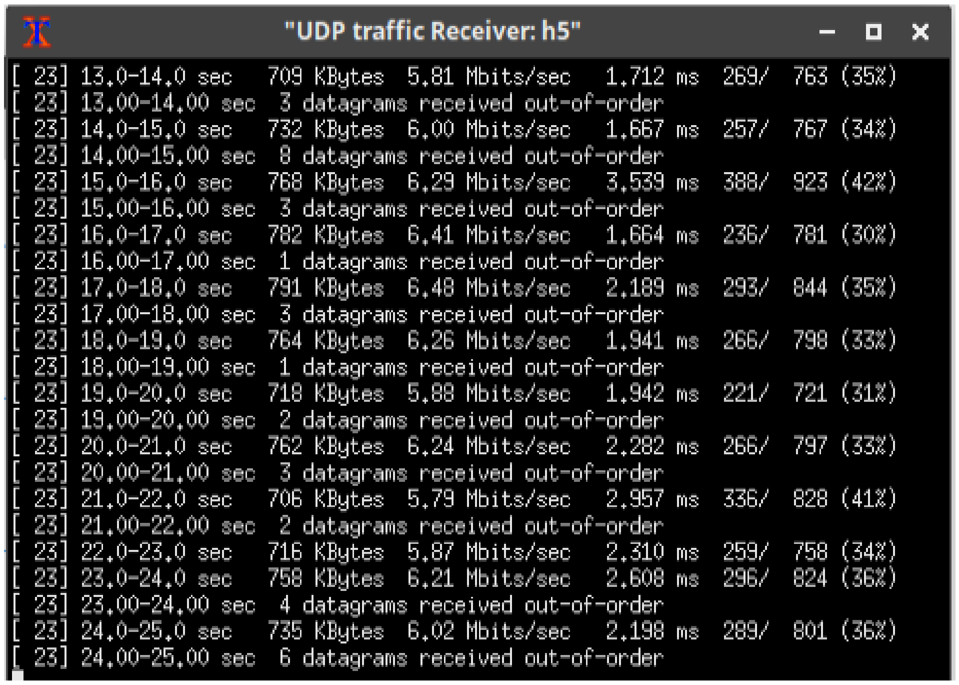

"camera2:receiving - VLC media player
Media Playback Audio Video Subtitle Tools View Help
00:00
00:00
"camera1:receiving - VLC media player
Media Playback Audio Video Subtitle Tools View Help
00:00
00:00

Figure 29. Vigilance camera without traffic competition

At this time the images are being transmitted with no problem as no other traffic interfering with this transmission. As mentioned before one of the video servers was already in the priority queue, in this case the video on the left side and the other server is in the non-priority queue sharing traffic with other devices.

4.5.1. Test without QoS

In this test, we generate a UDP data stream through Iperf that will pass through the low priority queue for the purpose of competing for camera bandwidth also in the same queue (for 200 seconds) as shown in figure 30. In terminal 2 “UDP Traffic Sender” we enter command: iperf -u -c 10.0.0.5 -b 8M -t 200.



```
[ 23] 13.0-14.0 sec   709 KBytes  5.81 Mbits/sec   1.712 ms  269/  763 (35%)
[ 23] 13.00-14.00 sec  3 datagrams received out-of-order
[ 23] 14.0-15.0 sec   732 KBytes  6.00 Mbits/sec   1.667 ms  257/  767 (34%)
[ 23] 14.00-15.00 sec  8 datagrams received out-of-order
[ 23] 15.0-16.0 sec   768 KBytes  6.29 Mbits/sec   3.539 ms  388/  923 (42%)
[ 23] 15.00-16.00 sec  3 datagrams received out-of-order
[ 23] 16.0-17.0 sec   782 KBytes  6.41 Mbits/sec   1.664 ms  236/  781 (30%)
[ 23] 16.00-17.00 sec  1 datagrams received out-of-order
[ 23] 17.0-18.0 sec   791 KBytes  6.48 Mbits/sec   2.189 ms  293/  844 (35%)
[ 23] 17.00-18.00 sec  3 datagrams received out-of-order
[ 23] 18.0-19.0 sec   764 KBytes  6.26 Mbits/sec   1.941 ms  266/  798 (33%)
[ 23] 18.00-19.00 sec  1 datagrams received out-of-order
[ 23] 19.0-20.0 sec   718 KBytes  5.88 Mbits/sec   1.942 ms  221/  721 (31%)
[ 23] 19.00-20.00 sec  2 datagrams received out-of-order
[ 23] 20.0-21.0 sec   762 KBytes  6.24 Mbits/sec   2.282 ms  266/  797 (33%)
[ 23] 20.00-21.00 sec  3 datagrams received out-of-order
[ 23] 21.0-22.0 sec   706 KBytes  5.79 Mbits/sec   2.957 ms  336/  828 (41%)
[ 23] 21.00-22.00 sec  2 datagrams received out-of-order
[ 23] 22.0-23.0 sec   716 KBytes  5.87 Mbits/sec   2.310 ms  259/  758 (34%)
[ 23] 23.0-24.0 sec   758 KBytes  6.21 Mbits/sec   2.608 ms  296/  824 (36%)
[ 23] 23.00-24.00 sec  4 datagrams received out-of-order
[ 23] 24.0-25.0 sec   735 KBytes  6.02 Mbits/sec   2.198 ms  289/  801 (36%)
[ 23] 24.00-25.00 sec  6 datagrams received out-of-order
```

Figure 30. UDP traffic receiver

Figure 31 shows that host2's injection of UDP traffic is negatively influencing quality real-time video transmission because its video stream shares the s1 switch output queue with the UDP traffic data stream. We can see that the left side camera initially in the high priority queue continues to stream the video smoothly, while the right-side camera is experiencing quality issues and the video jam several times.

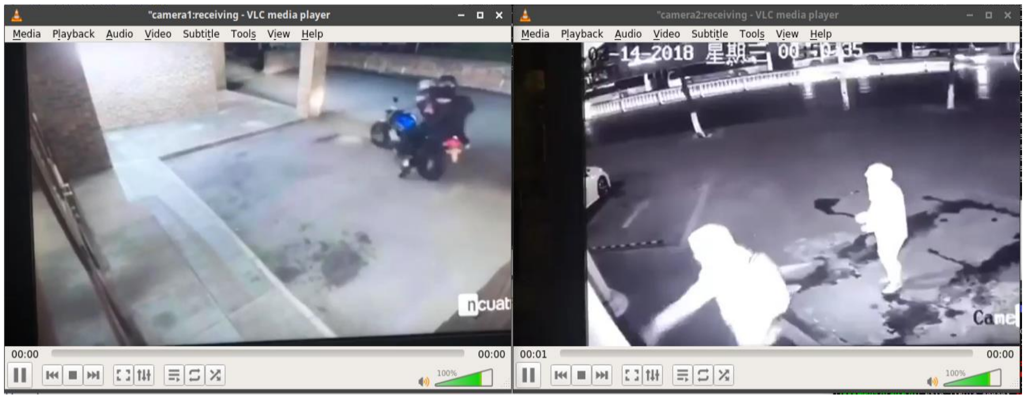
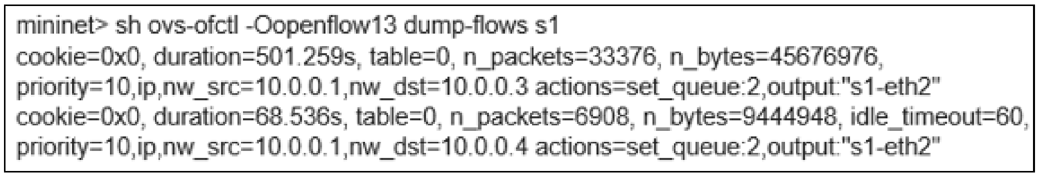

"camera1:receiving - VLC media player
Media Playback Audio Video Subtitle Tools View Help
00:00
00:00
"camera2:receiving - VLC media player
Media Playback Audio Video Subtitle Tools View Help
-14-2018 星期三 00:50:35
00:01
00:00

Figure 31. Vigilance camera with traffic competition

4.5.2. Test with QoS

In this test we intend to dynamically reassign the video host, that camera 2 will switch to high priority queue to improve its transmission quality. Therefore, as we do not have a real sensor that can detect motion and automatically make this priority queuing, we send an OpenFlow rule directly to switch s1 to do this update. So, we open a terminal and execute the script "set_priority.sh" with the command: ./set_priority.sh 10.0.0.1 10.0.0.4 hi. We can see the rule added in figure 32.



```
mininet> sh ovs-ofctl -Oopenflow13 dump-flows s1
cookie=0x0, duration=501.259s, table=0, n_packets=33376, n_bytes=45676976,
priority=10,ip,nw_src=10.0.0.1,nw_dst=10.0.0.3 actions=set_queue:2,output:"s1-eth2"
cookie=0x0, duration=68.536s, table=0, n_packets=6908, n_bytes=9444948, idle_timeout=60,
priority=10,ip,nw_src=10.0.0.1,nw_dst=10.0.0.4 actions=set_queue:2,output:"s1-eth2"
```

Figure 32. Flow rules at S1

From this moment on, host 4 referring to camera 2 is now in the priority queue and transmission starts to be transmitted with good quality like camera 1 transmission, as shown in figure 33.

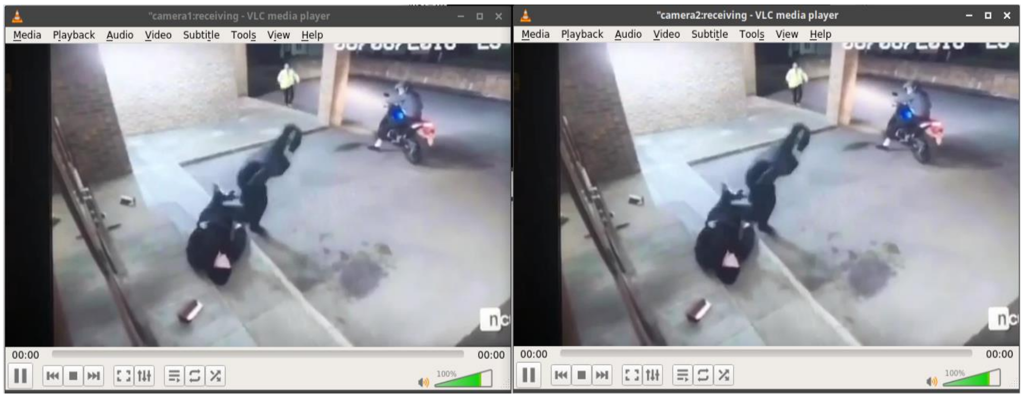

"camera1:receiving - VLC media player
Media Playback Audio Video Subtitle Tools View Help
00:00
00:00
"camera2:receiving - VLC media player
Media Playback Audio Video Subtitle Tools View Help
00:00
00:00

Figure 33. Transmission with QoS

If there is no more movement, we can simulate this situation with the same script through the command: ./set_priority.sh 10.0.0.1 10.0.0.3 lo. This command will cause the video user to start see bad transmission because we have dynamically reassigned the video flow to the non-priority queue.

4.5.3. Results

Figure 29 is referred to a graph showing the events in a temporal order of the moment that there is interference in traffic and QoS in action.

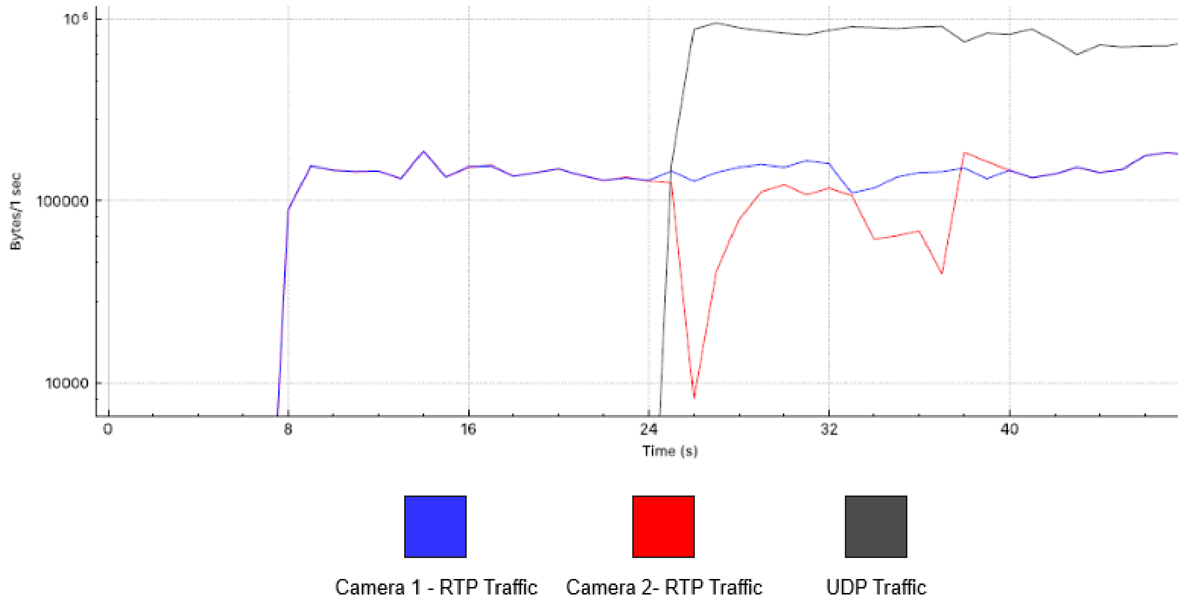


Figure 34. QoS results

We can see the reaction in the three interval moments. The first one is when there is no interference in the video traffic transmission. We can see that when the video transmission starts, the blue line (camera 1) is transmitting the video simultaneously to the red line (camera 2).In this moment, the camera 1 is in the high priority queues and camera 2 in the low priority queue.

The second interval begins around the second 24, when UDP traffic is injected for the purpose to causing interference with the camera 2 video transmission. Therefore, we can see that UDP (black line) traffic assumes practically all bandwidth and the red line traffic decreases, and camera 2 video faces transmission problems. This is because UDP traffic is competing bandwidth with camera 2 at the same queue. Currently, we do not feel the influence of QoS.

The last interval is applied QoS to improve the transmission quality of camera 2, and this occurs around the second 26. A flow rule has been dynamically applied to change the problematic video to the high priority queue. However, we can see that the transmission will begin to improve, and the red line will return to normal as it was in the first interval. We note that UDP is no longer interfering with the quality of the transmission.

4.6. Chapter Conclusions

In this chapter, we demonstrate how our proposal presented in chapter 3 was deployed in practice. First, we present a list of the main technologies and tools used. We present the detailed deployment testbed with their physical, logical components and IP identifiers. The topologies were about the inter-domain communication scenario and QoS scenario. We also demonstrated how each one was configured.

After the practical deployment, we present the system evaluation and demonstration of the test results obtained. These tests are related to inter-domain SDN communication, failures robustness, and QoS data rate testing. All deployment processes and tests was performed to be applied in an IoT context.

5. Conclusions and Future Works

In this chapter, some conclusions of this dissertation are discussed and a brief critical reflection in order to analyse if we can answer the initial research question in order to achieve the final goal. In this dissertation we face limitations that are also discussed and leave some suggestions of what could be improved to possible future works.

5.1. Conclusions

With the exponential growth of data traffic and the heterogeneity of today's communications networks, it challenges legacy network management solutions such that it is almost impossible to keep up with demand and the required QoS and performance requirements. As an ally for overcoming the problems faced by legacy networks, SDN due to their programmability and centralized management, offers the network designer’s tools to design flexible networks in distributed IoT networks.

While multi-controller scenarios in distributed systems increase network complexity and pose many challenges in efficient network infrastructure management, it is the best way to achieve good network performance and offer QoS guarantees because dividing the network into different Domains allows each controller to manage their domain independently and can cooperate with other domains consistently.

Therefore, the main goal of this dissertation was to understand how we can deploy and manage a network infrastructure consisting of several distinct administrative domains, in order to meet the QoS requirements in heterogeneous IoT networks. So, to achieve the proposed goal, a literature review was made with the purpose of understanding the functioning of the SDN, protocols and how we could ensure communication between distinct administrative domains.

With this, we designed and deployed a distributed network system, with two SDN edge domains, interconnected by an SDN traffic domain where all topology routing logic is centralized. We use knowledge related to BGP and the SDN-IP application running on the ONOS controller to act as a BGP speaker and translate intent into routing rules installed on OpenFlow switches.

After that, we pretended to deploy a way to ensure QoS in such scenarios. We opted to provide a mechanism that makes a traffic prioritization decision to test the QoS through OpenFlow

1.3 queues. The proposed scenario consists of a mixed network with traditional network devices and fictitious smart devices simulating IoT devices. These are vigilance cameras equipped with motion sensors, installed in public environments to meet security concerns. Therefore, we create priority queues that will be dynamically activated as soon as the cameras detect any movement and can transmit the highest quality image possible without interference from other network traffic.

Then we reach the stage of trying to answer the research question initially posed, "How to provide the necessary resources to meet QoS and robustness requirements for traffic originated in heterogeneous IoT devices, in a multi-domain SDN-based system?" According to the experimental result made in chapter 4 in section 4.4.1, we have shown that we are able to ensure communication between physically distributed SDN domains via the BGP protocol through a transit SDN system with the SDN-IP application running on the ONOS controller. We also demonstrate that our scenario is sensitive to link failures by redirecting traffic directly to another available path without causing service downtime.

Referring to the quality of service we demonstrate through the results obtained in chapter 4 in section 4.5.3, our system differs traffic and installs OpenFlow rules according to each priority in the corresponding queue. We have shown that vigilance cameras move to the priority queue as soon as they detect any movement and transmit the quality image without interference from other types of traffic, thus meeting safety concerns in public environments.

5.2. Future Work

Although our desire is always to do our best to achieve a goal, because of the limitations that have arisen during this work and the time that eludes us when we need it most, we are not always able to implement all our ideas. Therefore, it follows the ideas that we were unable to implement and suggestions for future work. Keeping this in mind, there are several possible ways to continue with the current work.

Improve the communication between SDN domains by creating a federation between them, that is, federate programmable resources distributed across different administrative domains through a cross-domain routing system with flexible policies for all members.

The use of real sensors to analyse system behaviour in terms of quality of service. So our application would be even more automated in making traffic prioritization decisions rather than manually, so this is a suggestion for future work.

Bibliography

[1] J. Jin, M. Palaniswami, J. Gubbi, and T. Luo, "Network architecture and QoS issues in the internet of things for a smart city" *International Symposium on Communications and Information Technologies (ISCIT)*, pp. 974–979, 2012.

[2] W. Xia *et al.*, "A Survey on Software-Defined Networking", *IEEE Communication Surveys & Tutorials*, vol. 17, no. 1, pp. 27–51, 2015.

[3] Y. Zhang, L. Cui, W. Wang, and Y. Zhang, "A survey on software defined networking with multiple controllers," *J. Netw. Comput. Appl.*, vol. 103. December 2017, pp. 101–118, 2018.

[4] A. Gupta *et al.*, "SDX: A software defined internet exchange," *Comput. Commun. Rev.*, vol. 44, no. 4, pp. 551–562, 2015.

[5] V. Kotronis, X. Dimitropoulos, R. Kloti, B. Ager, P. Georgopoulos, and S. Schmid, "Control Exchange Points: Providing QoS-enabled End-to-End Services via SDN-based Inter-domain Routing Orchestration," *T-Labs & TU Berlin, Germany Introduction*.pp. 3–4, 2016.

[6] P. Berde *et al.*, "ONOS : Towards an Open , Distributed SDN OS," pp. 1–6.

[7] S. Badotra, "Open Daylight as a Controller for Software Defined Networking," *International Journal of Advanced Computer Research*. 2018.

[8] L. Mamushiane, A. Lysko, and S. Dlamini, "A comparative evaluation of the performance of popular SDN controllers," *IFIP Wirel. Days*, vol. 2018-April, pp. 54–59, 2018.

[9] M. Gerola *et al.*, "ICONA: Inter Cluster ONOS Network Application," 2015.

[10] P. Lin *et al.*, "Seamless interworking of SDN and IP," *Comput. Commun. Rev.*, vol. 43, no. 4, pp. 475–476, 2013.

[11] D. Gupta and R. Jahan, "Inter-SDN Controller Communication: Using Border Gateway Protocol," no 1. April, pp. 1–16, 2014.

[12] C. Xu, B. Chen, and H. Qian, "Quality of service guaranteed resource management dynamically in software defined network," *J. Commun.*, vol. 10, no. 11, pp. 843–850, 2015.

[13] Z. Latif, K. Sharif, F. Li, M. M. Karim, and Y. Wang, "A Comprehensive Survey of Interface Protocols for Software Defined Networks," *IEEE Globecom Workshops (GC Wkshps)*, Dec pp. 1–30, 2019.

[14] R. Amin, M. Reisslein, and N. Shah, "Hybrid SDN networks: A survey of existing approaches," *IEEE Commun. Surv. Tutorials*, vol. 20, no. 4, pp. 3259–3306, 2018.

[15] D. Kreutz *et al.*, "Software-Defined Networking: A Comprehensive Survey," *IEEE Communications Surveys & Tutorials*, pp. 1–61, 2014.

[16] I. C. Surveys *et al.*, "Distributed SDN Control : Survey, Taxonomy and Challenges," *IEEE Communications Surveys & Tutorials*, no 1. December, 2017.

[17] B. Davie *et al.*, "A database approach to SDN control plane design," *Comput. Commun. Rev.*, vol. 47, no. 1, pp. 15–26, 2017.

[18] Murray. P, Stalvig. P, "SNMP : Simplified", *F5 White Paper*, 2015.

[19] N. Sambo, A. Giorgetti, F. Cugini, M. Dallaglio, and P. Castoldi, "Control and Management of Sliceable Transponders," *Eur. Conf. Opt. Commun. ECOC*, vol. 2017-Septe, no. 3, pp.

1–3, 2017.

[20] F. X. A. Wibowo, M. A. Gregory, K. Ahmed, and K. M. Gomez, "Multi-domain Software Defined Networking: Research status and challenges," *26th International Telecommunication Networks and Applications Conference (ITNAC),* vol. 87, no. March, pp. 32–45, 2017.

[21] V. W. Protocol, "OpenFlow Switch Specification," vol. 0, 2009.

[22] V. Implemented and W. Protocol, "OpenFlow Switch Specification," 2011.

[23] V. W. Protocol, "OpenFlow Switch Specification," vol. 0, pp. 0–105, 2012.

[24] V. W. Protocol, "OpenFlow Switch Specification," vol. 0, pp. 1–206, 2013.

[25] V. Protocol, "OpenFlow Switch Specification" vol. 0, 2014.

[26] L. Zhu, M. M. Karim, K. Sharif, F. Li, X. Du, and M. Guizani, "SDN Controllers: Benchmarking & Performance Evaluation," *Under Review at IEEE JSAC* pp. 1–14, 2019.

[27] T. Hu, Z. Guo, P. Yi, T. Baker, and J. Lan, "Multi-controller Based Software-Defined Networking: A Survey," *IEEE Access*, vol. 6, pp. 15980–15996, 2018.

[28] D. Erickson, "The Beacon OpenFlow Controller," 2013.

[29] V. B. Harkal, "Software Defined Networking with Floodlight Controller," *Int. J. Comput. Appl.*, pp. 975–8887, 2016.

[30] N. Gude, J. Pettit, and S. Shenker, "Nox: Os for Networks", *In ACM SIGCOMM Computer Communication Review,* 2009.

[31] S. Asadollahi, B. Goswami, and M. Sameer, "Ryu controller's scalability experiment on software defined networks," *2018 IEEE Int. Conf. Curr. Trends Adv. Comput. ICCTAC 2018*, pp. 1–5, 2018.

[32] O. Michel and E. Keller, "SDN in wide-area networks: A survey," *Proc. IEEE Fourth Int. Conf. Softw. Defin. Syst.*, pp. 37–42, 2017.

[33] M. Karakus and A. Durresi, "A survey: Control plane scalability issues and approaches in Software-Defined Networking (SDN)," *Comput. Networks*, vol. 112, pp. 279–293, 2017.

[34] O. Michel and E. Keller, "SDN in wide-area networks: A survey," *2017 4th Int. Conf. Softw. Defin. Syst. SDS 2017*, pp. 37–42, 2017.

[35] M. T. I. Ul Huque, W. Si, G. Jourjon, and V. Gramoli, "Large-Scale Dynamic Controller Placement," *IEEE Trans. Netw. Serv. Manag.*, vol. 14, no. 1, pp. 63–76, 2017.

[36] T. Koponen *et al.*, "Onix: A Distributed Control Platform for Large-scale Production Networks."

[37] A. Tootoonchian, "HyperFlow: A Distributed Control Plane for OpenFlow", *SIGCOMM Computer Communication Review 38,* 2010.

[38] S. H. Yeganeh, "Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications," pp. 19–24, 2012.

[39] O. Salman, I. H. Elhajj, A. Kayssi, and A. Chehab, "SDN controllers: A comparative study," *Proc. 18th Mediterr. Electrotech. Conf. Intell. Effic. Technol. Serv. Citizen, MELECON 2016,* no. October 2017, 2016.

[40] and D. C. K. R. Jahan, S. Shaik, K. Kotaru, "ODL-SDNi," *ODL Wiki*. 2014.

[41] D. Gupta, T. Consultancy, and S. Limited, “Inter-SDN Controller Communication : Using Border Gateway Protocol,” no. April, 2018.

[42] F. X. A. Wibowo and M. A. Gregory, “Software Defined Networking properties in multi-domain networks,” *26th Int. Telecommun. Networks Appl. Conf. ITNAC 2016*, pp. 95–100, 2017.

[43] P. Lin, J. Bi, Z. Chen, Y. Wang, H. Hu, and A. Xu, “WE-bridge: West-east bridge for SDN inter-domain network peering,” *Proc. - IEEE INFOCOM*, no. February 2015, pp. 111–112, 2014.

[44] H. Yin, H. Xie, T. Tsou, P. Aranda, D. Lopez, and R.Sidi, “SDNi: A Message Exchange Protocol for Software Defined Networks (SDNS) across,” *Internet Research Task Force*. pp. 1–14, 2012.

[45] K. Phemius *et al.*, “DISCO : Distributed Multi-domain SDN Controllers To cite this version : HAL Id : hal-00854899 DISCO : Distributed Multi-domain SDN Controllers,” 2013.

[46] J. H. and L. Prete., “SDN-IP - ONOS,” *ONOS wiki*. 2016.

[47] ONOS, “ONOS : An Overview,” *ONOS wiki*. .

[48] ONOS, “Intent Framework architecture.,” *ONOS wiki*. .

[49] D. Kim, Y. H. Kim, C. Park, and K. Il Kim, “KREONET-S: Software-defined wide area network design and deployment on KREONET,” *IAENG Int. J. Comput. Sci.*, vol. 45, no. 1, pp. 27–33, 2018.

[50] G. T. Service and O. Icona, “GEANT′s Pan-European ONOS ICONA Deployment Delivers Benefits of Open Source SDN to Europe′s Research and Education Networks”, 2015.

[51] I. Awan, M. Younas, and W. Naveed, “Modelling QoS in IoT applications,” *Proc. - 2014 Int. Conf. Network-Based Inf. Syst. NBiS 2014*, pp. 99–105, 2014.

[52] A. S. Muqaddas, A. Bianco, P. Giaccone, and G. Maier, “Inter-controller traffic in ONOS clusters for SDN networks,” *2016 IEEE Int. Conf. Commun. ICC 2016*, pp. 1–17, 2016.

[53] D. Sanvito, D. Moro, M. Gulli, I. Filippini, A. Capone, and A. Campanella, “Enabling external routing logic in ONOS with Intent Monitor and Reroute service,” *2018 4th IEEE Conf. Netw. Softwarization Work. NetSoft 2018*, no 1. NetSoft, pp. 37–45, 2018.

[54] D. Sanvito, D. Moro, M. Gulli, I. Filippini, A. Capone, and A. Campanella, “ONOS Intent Monitor and Reroute service: Enabling plugplay routing logic,” *2018 4th IEEE Conf. Netw. Softwarization Work. NetSoft 2018*, no 2. NetSoft, pp. 456–461, 2018.

[55] B. K. Thapa and B. Dikici, “Reactive Forwarding Applications in ONOS,” no 2, *2016 IEEE Int. Conf. Commun. ICC*, January, 2018.

[56] K. Ishiguro, “Quagga A routing software package for TCP/IP networks Quagga 1.2.0,” no. March, *Int. J. Comput. Appl*, 2017.