Politecnico di Milano

School of Industrial and Information Engineering

Master of Science in Telecommunications Engineering

Electronics, Information and Bioengineering Department

Towards Smart Intents with Robust and Flexible Routing

Supervisor: Prof. Antonio CAPONE

Advisor: Eng. Davide SANVITO

Master Thesis of:

Mattia GULLÍ

Student ID. 841829

Accademic Year 2016–2017

Alla mia famiglia, per tutti i sacrifici fatti in questi anni...

Abstract

The main idea on which Software-Defined Networking (SDN) is based, is the separation of the control plane from the data plane. This new structure has inspired many programmers to find new solutions to improve network performance. SDN controller simplifies network configuration via high-level languages and abstraction. SDN-IP, an example of SDN application developed by ONF (Open Networking Foundation) inside ONOS (Open Network Operating System), allows a SDN network to connect to external networks on the Internet using the standard Border Gateway Protocol (BGP). ONOS builds these connections through the use of high-level policies, ONOS Intents, that, with the Intent Framework, are translated in OpenFlow rules. This allows the programmers not to think about how to write the low-level rules device by device. The NOS (Network Operating System) hides all the details of the process compilation and path computation, which without further constraints, just translate each Intent individually to one of the shortest paths.

In this work, the possibility of jointly re-optmizing the paths of the demands has been explored, through the monitoring of flow statistics into the network. The extension of the application SDN-IP would allow to monitor, periodically, the AS-to-AS traffic and re-route it to minimize the network congestion and so the Maximum Link Utilization. At the same time, we don't break the simplicity of the mapping between BGP announcements and the corresponding Intents. To avoid too frequent network reconfigurations, that would create network instability, we use a set of robust routing solution with a minimum time constraint in which a routing should be kept stable.

At the beginning, the traffic is forward according to the rules of the shortest path (default behaviour of Intents) and measured for an interval time (e.g. a day). After-

wards, through interfacing with an off-platform module, we compute a set of robust configurations to be applied for the next period. The module, also, via REST-API, gathers the measurements, defines two optimization models to compute the configurations and the activation times, and, finally, schedules the activation into the system. To cope with traffic deviations, with respect to the corresponding traffic profile of the training period, routings are computed to be robust over subsets of traffic matrix space.

Our goal is modify SDN-IP application to adapt the Intents, created according to BGP announcements, to PointToPoint Intents and periodically re-optimize the paths according to their statistics.

Sommario

L'idea principale su cui SDN è basato, è la separazione del piano di controllo da quello di inoltro. Questa nuova struttura ha inspirato molti programmatori a trovare nuove soluzioni per migliorare le performance della rete. Il controller SDN semplifica la configurazione della rete attraverso l'astrazione e l'utilizzo di linguaggi ad alto livello. SDN-IP, un esempio di applicazione SDN sviluppata da ONF (Open Networking Foundations) all'interno di ONOS (Open Network Operating System), permette ad una rete SDN di connettersi a reti esterne attraverso Internet usando lo standard BGP (Border Gateway Protocol). ONOS costruisce queste connessioni attraverso l'uso di policy ad alto livello, chiamati Intenti ONOS, che, con l'Intent Framework, vengono tradotte in regole OpenFlow. Questo aiuta i programmatori a non pensare più a come scrivere le regole a basso livello su ogni device. Il NOS (Sistema Operativo di Rete) nasconde tutti i dettagli del processo di compilazione e del calcolo dei path, e, senza ulteriori vincoli, traduce individualmente ogni Intento in un possibile shortest path.

In questo lavoro, viene esplorata la possibilità di ri-ottimizzare contemporaneamente i percorsi delle domande attraverso il monitoraggio delle statistiche dei flussi all'interno della rete. L'estensione dell'applicazione SDN-IP permetterebbe il monitoraggio periodico del traffico AS-to-AS, reinstradandolo con lo scopo di minimizzare la congestione della rete e quindi il massimo utilizzo dei link. Allo stesso tempo, non vogliamo rompere la semplicità con cui vengono mappati gli annunci BGP con i corrispondenti Intenti.

Per evitare frequenti cambi di configurazione di rete, che creerebbero instabilità, utilizziamo un set di soluzioni di routing robusto imponendo un tempo minimo in cui il routing deve essere mantenuto stabile.

All'inizio, il traffico viene instradato secondo le regole dello shortest path (il comportamento predefinito degli Intenti) e misurato per un intervallo di tempo (per esempio un giorno). Successivamente, attraverso l'interfacciamento con un modulo esterno, calcoliamo un set di configurazioni di routing robusto da applicare nell'intervallo successivo. Il modulo, inoltre, via REST API, raccoglie le misure, definisce due modelli di ottimizzazione per calcolare le configurazioni e i tempi di attivazione, e, infine, schedula le loro attivazioni all'interno del sistema. Per far fronte alle possibili deviazioni di traffico, rispetto al profilo studiato durante il periodo di monitoraggio, le configurazioni sono calcolate per essere robuste ad un sottoinsieme di matrici di traffico.

Il nostro obiettivo è quello di modificare l'applicazione SDN-IP per adattare gli Intenti, creati in accordo agli annunci BGP, in Intenti PointToPoint e periodicamente riottimizzare i percorsi secondo le loro statistiche.

Contents

List of Figures

Chapter 1

Introduction

In the last few years, Internet has become an important instrument for every person that work, travel or, simply, use latest generation devices. On average, every person use, during the day, from 2 to 5 devices which allows him to interface with Internet and/or with digital services offered by the network. Just think, how many function a smartphone or a tablet, device of common use, can offer us. We use them to exchange messages (Chat, Email, Social, Video conference), we update ourselves on the real time news or the situation of the traffic, we use digital services such as streaming content, gaming online or E-Commerce up to use enterprise services and resources installed in the "cloud". This development, and so this greater use of the "network", causes, however, a massive traffic that ISP should manage, guaranteeing a QoE (Quality of Experience) suitable for each offered service. During these years of research, many technologies have been presented and developed but the one that has received more attention is Software Defined Network together with the use of an open interface (e.g. OpenFlow).

Software Defined Network (SDN) has created enormous interest in academia and industry. It is a new network paradigm whose fundamental feature is the division of the control plane from the forwarding plane. This division allows to have a control plane more flexible and programmable, through the use of an external entity, called *SDN Controller*, which hosts a running instance of a *Network Operating System* (NOS) that is able to configure the forwarding plane according to the applications and network

services.

A **Network Operating System (NOS)** is a software program that controls other software and hardware running on a network. It also allows a network computers to communicate with one central hub and each other to share resources, run applications, and send messages. In this way, network operators have a better centralized view of the network allowing them to monitor and collect, more accurately, the statistics of the traffic behavior.

During the early days, the integration of SDN in the current network has created some problems since it was not possible to integrate a new system with ease in a few operations. One of these was the coexistence of the traditional IP networks with SDN deployment since each SDN deployment must be able to exchange reachability information, forward traffic, and express routing policies with existing IP networks.

Today on the Internet, without the use of SDN principle, peering between AS (Autonomous System), a network or group of networks under a common administration and with common routing policies, is done with BGPv4. [1]. **BGPv4** (Border Gateway Protocol version 4) is an inter-autonomous system routing protocol used to exchange routing information on the Internet and it is the protocol used between Internet service providers (ISP). Two protocols with a set of legacy IP network are used to exchange information: one is *external BGP* (eBGP) used to communicate to other BGP enabled systems in different AS while the other is *interior BGP* (iBGP) used between the routers in the same AS.

A possible solution to integrate SDN network, is given by an application integrated in ONOS, SDN-IP. **ONOS** (Open Network Operating System) [2] provides the control plane for a Software Defined Network (SDN), managing network components, such as switches and links, and running software programs or modules to provide communication services to end hosts and neighboring networks. In ONOS, the communication, so the connection with the devices, can be configured at different level of abstraction. One of these is made by *Intent Framework*. It allows an application to request a service from the network without having to know details of how the service will be performed. This permits network operators as well as application developers to program the net-

work at a high level in form of policy rather than mechanism. This directives are called *Intents*. These directives, afterwards, are translated in a set of rules to apply to the controller, so the operators no longer have to consider how to write the rules but just write the request. ONOS provides a global network view to applications, which, in turn, offers a view of the network (host, switches, link, and any other state associated with the network such as utilization). An application can program this network view through APIs. This helps the programmers to develop your own application or some integration to the system. In fact, ONOS applications and use cases often consist of customized communication routing, management, or monitoring services for Software Defined Network s.

SDN-IP [3], as we have already said, is an ONOS application that allows an SDN network to peer and exchange traffic with adjacent networks and to the rest of the Internet using BGP. A service provider can deploy a small SDN network as an AS and use the SDN-IP peering application to seamlessly connect the SDN network to the rest of the Internet using BGP. With the SDN-IP application, an SDN network appears as just another AS to the rest of the Internet. Moreover, a service provider can use the SDN-IP to inter-connect multiple SDN networks to create a large SDN AS that peers with the Internet the same way as other AS.

In Chapter 3, we will provide more information on ONOS, SDN-IP and how Intent Framework works.

1.1 Main Contribution

The main contribution that we want to implement in ONOS, is to extend the SDN-IP application with the aim of re-optimizing the paths according to events that can be defined based on flow level statistics. Without further constraints, Intents are individually compiled to one of the shortest path. In our work, we want to consider jointly multiple Intents in the compilation that can reactively take into account flow-level statistics events to optimize a global network objective as an example minimize the average Maximum Link Utilization (MLU). To do this, the application monitor,

periodically, the statistics of the AS-to-AS traffic and re-route it with the aim to minimize the MLU of the network. The part of re-creation of the new routing paths, is done by an *off-platform app*. Through the traffic measurement, previously retrieved for a training period via REST API, the application defines two optimization models to compute the routing configurations, activation times and finally schedules their activation at the proper time, also this via REST API. To avoid too frequent network reconfiguration, that would create network instability we put a limit to the minimum amount of time that a routing should be kept.

Finally, we compare the average MLU of the our solution with the integrated ONOS application routing (that is based on shortest path) using real traffic traces, demonstrating that our solution can decrease the MLU of the network up to 15%.

1.2 Thesis Structure

Chapter 2 gives an overview of SDN and how Traffic Engineering helps the network to improve the QoS, showing some state of art solutions. We will explain the main points on which SDN is based and the most famous standard used in SDN, OpenFlow.

Chapter 3 contains a more in-depth explanation of the main tools used in the thesis among which ONOS, SDN-IP, including a briefly description of its architecture, and the network emulator, Mininet.

Chapter 4 is a description of our work, based on the extension of the SDN-IP application, that can collect flow statistics of the network and, using an off-platform module, computes a set of robust routings to apply with the aim of minimizing network congestion.

Chapter 5 contains the results of the simulations. These results will then be compared with the shortest path routing of the legacy SDN-IP application.

Chapter 6 will present our conclusions and possible future works.

Chapter 2

Software-Defined Networking and Traffic Engineering

This chapter gives a small description about Software Defined Network, including one of the protocol used on SDN, Openflow, and how Traffic Engineering can help the service provider in the network configuration.

2.1 Software Defined Network

The SDN paradigm is aimed at supporting the dynamic and scalable computing and storage needs of modern telecommunication environments, by decoupling the control plane from the data plane. The first maintains a global view of the network and makes decision on how to forward packets, the second represents the part of the system that physically receives and forwards the packets. This migration of control, closely linked to individual devices on accessible computing devices, enables the underlying infrastructure to be abstract for applications and network services.

We can define SDN as a network architecture with four pillars main points:

1. The control and data planes are decoupled. Control functionality is removed from network devices that will become a simple (packet) forwarding elements;

2. Forwarding decisions are flow based: we can consider it as a set of packets field

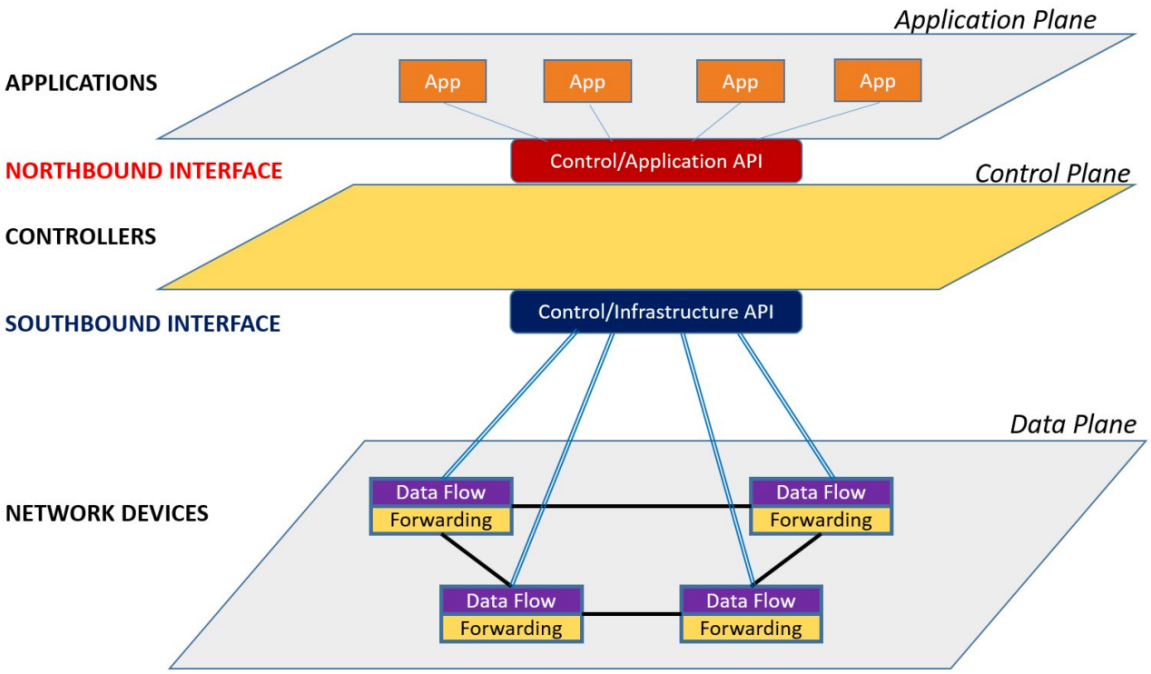

Figure 2.1: SDN Architecture

values, between a source and destination, acting as a match criterion and a relative set of actions (instructions). All these packets receive the same service policies at the forwarding devices. The flow abstraction allows unifying the behavior of different type of network devices, including routers, switches, firewalls and middleboxes. Flow programming allows for never-seen flexibility, limited only to the capabilities of the implemented flow tables.

3. Control logic is moved to an external entity, SDN controller. Provides the minimal resources and abstraction to facilitate the programming of forwarding devices based on a logically centralized, abstract network view.

4. The network is programmable through software application running on top of the controller that interacts with the underlying data plane devices.

By centralizing network state in the control layer, SDN gives the following benefits:

- **Directly Programmable**: Since the control functions are decoupled from the forwarding functions, the network can be configured by proprietary or open source automation tools;

- **Centralized Management**: Network intelligence is logically centralized in SDN controller software that maintains a global view of the network, which appears to applications and policy engines as a single and logical switch;
- **Reduce CapEx and OpEx**: SDN limits the need to purchase the purpose-built networking hardware and the cost for creating, running, and delivering services to costumers since the hardware or software switches/routers are more programmable, making easier to design, deploy and manage networks;
- **Simplicity**: Network design and operation are simplified because forwarding instructions are provided by SDN controllers instead of multiple, vendor-specific devices and protocols.

Also, SDN architecture supports a set of APIs that make it possible to implement common network services, including routing, access control, bandwidth management, Traffic Engineering and quality of service.

In most SDN implementations, the communication between the two planes is carried out by means of the OpenFlow protocol [4], which allows remote administration of packet forwarding tables in network devices, by adding, modifying and removing packet matching rules and associated actions.

SDN was first standardized in 2011 by the Open Networking Foundation (ONF), which is self-defined as "*a user-driven organization dedicated to the promotion and adoption of SDN, and implementing SDN through open standards, necessary to move the networking industry forward*". ONF is the entity behind the standardization of the OpenFlow protocol, which also inherently standardizes the interface between control and data planes which allows SDN deployment [5].

2.2 Openflow

OpenFlow (OF) is considered the de facto SDN standard [6]. The original concept for OpenFlow begun at Stanford University in 2008. By December 2009, Version 1.0 of the OpenFlow switch specification was released [7]. Since its inception, OpenFlow has

been managed by ONF. Originally, it was defined to create a communication protocol in SDN environments that permits SDN controller to interact, directly, with the forwarding plane of network devices such as routers or switches, both physical and virtual. This controller runs a software platform with the APIs enabling the direct control of data flow in a network. OpenFlow, also, defines a standard on which the concepts of control plane and forward plane can be abstracted, so these two functions no longer need to be on the same devices. This means that programmers can define your own rules on the control plane, without relying on some algorithms developed by others, and write your own control protocol to define package rules as needed. It's possible to manipulate flow tables and flow entries on network devices without directly connecting to the network devices using APIs to communicate with the controller which will take care of the details needed to update the network devices flow tables. In addition to this, the network programmers, can develop their own algorithms to control data flows and packets.

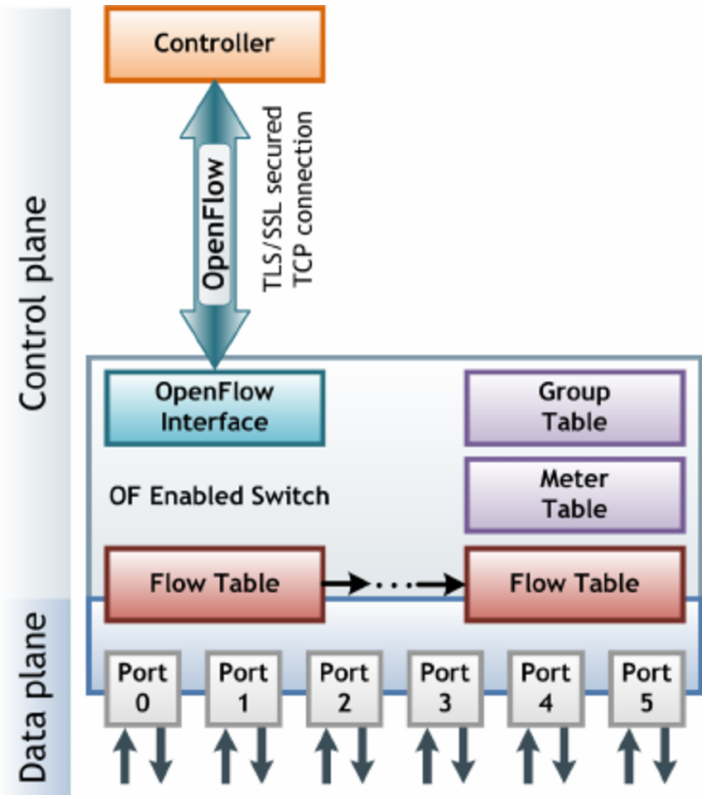

Figure 2.2: Main components of an OpenFlow switch

Focusing on its main features, we can recapitulate the benefits of OpenFlow:

- Brings network control functions out of switches and routers allowing direct access and manipulation of the forwarding plane of those devices;
- Specifies the basic primitives that can be used by an external application to program the forwarding plane of network devices, just like a set of instructions for a CPU;
- forwards flows according to pre-defined match rules statically or dynamically programmed by the SDN control software.

More details on the OpenFlow protocol and its version are available on ONF Site [8]

2.3 Traffic Engineering in SDN

Internet Traffic Engineering is defined as that aspect of Internet network engineering dealing with the issue of performance evaluation and performance optimization of operational IP networks. Traffic Engineering encompasses the application of technology and scientific principles to the measurement, characterization, modeling, and control of Internet traffic. [9].

As we can read from the RFC document, Traffic Engineering (TE) is an important network application, which studies measurement and management of network traffic, and designs reasonable routing mechanisms to improve network resources utilization and QoS (Quality of Service).

Thanks to the separation of the control plane from the forwarding plane, TE mechanism in SDN can be more efficiently and better to implement, since SDN controller provides a centralized visibility to the global network compared to older approaches such as MPLS-based or IP-based TEs. As presented in [10] [11], the above mentioned features of SDN help to solve current network Traffic Engineering issues. We can recap as follows:

- **Traffic monitoring and analysis**: Include all the techniques adopted to collect real-time information about network status, as an example resource usage, current topology and performance (on a per-link, per-flow, or per-packet basis);

- **Traffic scheduling and management**: Based of the information obtained from the analysis, we can predict the future traffic to avoid congestion and improve network efficiency with the algorithms that will adapt or reconfigure forwarding policies;

In the following subchapters, we will briefly describe the best known research solutions, at the moment, to measure and manage in SDN network.

2.3.1 Traffic Monitoring in SDN

Monitoring in SDN must be continuously in order to quickly adapt forwarding rules in response to changes in workload. The first solutions, integrated in IP networks, were NetFlow [12], developed by Cisco, sFlow [13], from InMon and JFlow [14], from Juniper Networks. These solutions are not suitable for SDN network due to excessive overhead in the statistic collection. So, the engineers developed other solutions. The following solutions try to reduce this problem and improve the accuracy of the statistics.

PayLess [15] is a query-based flow measurements framework with low costs based on polling that provides a flexible RESTful API for flow statistics collection. It maintaines an abstract view of network information for applications, and provides consolidate programming interfaces for a variety of network applications. Users can add their own framework and access to the collected data, stored at different aggregation levels by PayLess.

OpenTM [16] is a system based on traffic matrix (TM) estimation for OF networks. This is also query-based. It can detect and tracking all the active flows in the network according to the routing information from the OF controller and flow forwarding path information. Using the previous information, it builds the TM by adding the collected flow statistics between the same source-destination.

FlowSense [17] is a passive monitoring module for SDN controller that analyzes control messages between controller and switches. It uses these information to monitor network utilization such as bandwidth without an additional overhead.

OpenSketch [18] is a measurement framework that uses a software defined method to evaluate traffic. OpenSketch proposes that measurement control and data layers should be separated. Moreover, data planes are modeled as three-phase actions (hashing, filtering and counting) that can be configured dynamically at switches. Using this pipeline design, implemented on NetFPGA hardware, it abstracts a variety of measure algorithms into several general steps. At the same time, it provides various measurement modules in the control layer which allows users to adjust any phases of the data layer.

OpenNetMon [19] is a network monitoring module that can monitor throughput, packet loss rate and network delays continuously. It consists of three phases: the first, it measures the throughput, from only last switch using an adaptive frequency, through the ratio between the number of sampled packets for each flow and the sampling interval. The second, it calculates packet loss rates from the first and the last switch to gather counter statistics. Finally, the third, it calculates the delay using probe messages sent on the data layer.

2.3.2 Traffic Management in SDN

The purpose of traffic management is to improve network performance and maintain an high availability of the network. The *hash-based Equal-Cost Multi-Path* (ECMP) is a routing strategy where next-hop packet forwarding to a single destination can occur over multiple "best paths" which tie for top place in routing metric calculations. This is the "simple" effective load balancing solution but with some problems, including the inability to handle many large streams, called *elephant flows*, which are sent on the same path, causing load unbalanced links. To solve this problem, the researchers have developed many algorithms but in this part, we will describe only the best known load balancing models for data layer traffic.

Hedera [20] is a scalable and dynamic traffic management system that collects flow information from switches and schedules forwarding of packet to improve the utilization of network resources. The scheduling strategy of Hedera is based on three steps. First of all, it detects elephant flows at the edge switches and using ECMP by default. After that, Hedera collects, periodically, flow information and if it finds a flow rate which is beyond a specific threshold, then it is marked as an elephant flow. Finally, it calculates an appropriate path dynamically according to bandwidth request of the flow.

Mahout [21] is a system that reduces cost of traffic management using an additional backend server to detect elephant flows instead of forwarding equipment's. Mahout abstracts a special function layer of the OS of the backend server (the SHIM layer) that monitors local traffic and detects them using a socket buffer located on end hosts. With this buffer, the system is able to marked an elephant flow when the buffer exceeds a threshold. The Mahout controller, afterwards, computes the best path for this elephant flow and installs a flow-specific entry in the switch.

Another traffic load balancing mechanism is based on the wildcard scheduling technology [22] [23]. Wildcars can be utilized for aggregating client requests based on the range of IP prefixes. This permits the distribution and the redirection of many client requests without involving the controller for every new flow. Two principal solutions are DevoFlow and DIFANE.

DevoFlow [24] proposes that the controller should reroute elephant flows when there is certainty and not on the basis that all data flows are potential elephant flows, with the consequence of a reduction of interactions with the controller. DevoFlow is developed to allow aggressive use of wild-carded OpenFlow rules through a new mechanisms able to detect QoS-significant flows efficiently, by waiting until they actually become significant. DevoFlow also introduces a new mechanisms to allow switches to make local routing decisions for "microflows" without involving the controller.

The fundamental idea of **DIFANE** (DIstributed Flow Architecture for Networked Enterprises) [25] can be recapitulated in two parts. First, the controller distributes some rules to an authorized subset of switches. The controller runs a partitioning

algorithm that divides the rules equally and minimizes fragmentation of the rules across multiple authority switches. Second, switches can manage all packages in the data layer. When a flow cannot find a match in the local rules, it is encapsulated and redirect to an authorized subset of switches.

In some cases, the process to replace tradional networks for the SDN lasts a long time, so **IP/SDN hybrid network** emerged. This solution has only a part of the features of SDN, but it can take advantage of the partial and accurate global view of the network provided by SDN switches.

Compared to full SDN solutions, TE in IP/SDN network is more complicated. Some examples are OSHI (Open Source Hybrid IP/SDN) [26] that combines Quagga for OSPF (Open Shortest Path First) routing and SDN devices (e.g. Open vSwitch [27]) on Linux to provide backward compatibility for supporting incremental SDN deployments.

Agarwal et al. [28], in their solution, try to leverage the centralized controller to obtain a better network utilization such as reducing delays and packet losses.

Guo et al. [29] use an hybrid SDN/OSPF network. Starting from the idea that weights and flow splitting ratios of the SDN nodes can both be changed with the aim of minimizing the maximum link utilization. The controller can arbitrarily split the flows coming into the SDN nodes while the regular nodes still run OSPF.

Guo et al. [30] propose an heuristic algorithm to obtain a migration sequence of the legacy routers to SDN-enabled routers, to obtain the most of the benefits from the perspective of TE, so that we can decide where and how many routers to migrate firstly.

Chapter 3

SDN Tools

In this chapter we will define the instruments used in this work. We start to describe the general idea of ONOS, the operating system that we used in this thesis. We will continue with one of the ONOS applications which we worked on, SDN-IP, followed with a brief description of Mininet.

3.1 ONOS

ONOS (Open Network Operating System) [31] is an Open-Source Network OS (NetOS) developed and administrated by ONOS Project. It provides the control plane for SDN, managing network components, such as switches and links, and running software programs or modules to implement communication services to end hosts and neighboring networks [32].

ONOS, moreover, aims to:

- providing APIs and abstractions, resource allocation, and permissions, as well as user-facing software such as a CLI, a GUI, and system applications;
- managing the entire network rather than a single device, which can dramatically simplify management, configuration, and deployment of new software, hardware, and services;
- acting as an extensible, modular, distributed SDN controller.

ONOS can run as a distributed system across multiple servers, allowing it to use the CPU and memory resources of multiple servers while providing fault tolerance in the face of server failure and potentially supporting live upgrades of hardware and software without interrupting network traffic.

The ONOS kernel and core services, as well as ONOS applications, are written in Java as bundles that are loaded into the *Karaf OSGi container*. OSGi (Open Service Gateway initiative) is a component system for Java that allows modules to be installed and run dynamically in a single JVM (Java Virtual Machine). Since ONOS runs in the JVM, it can run on several underlying OS platforms.

Our contribution is based on SDN-IP application, implementing a new optimized and robust routing based on collection of Traffic Matrices, signed in the figure 3.1

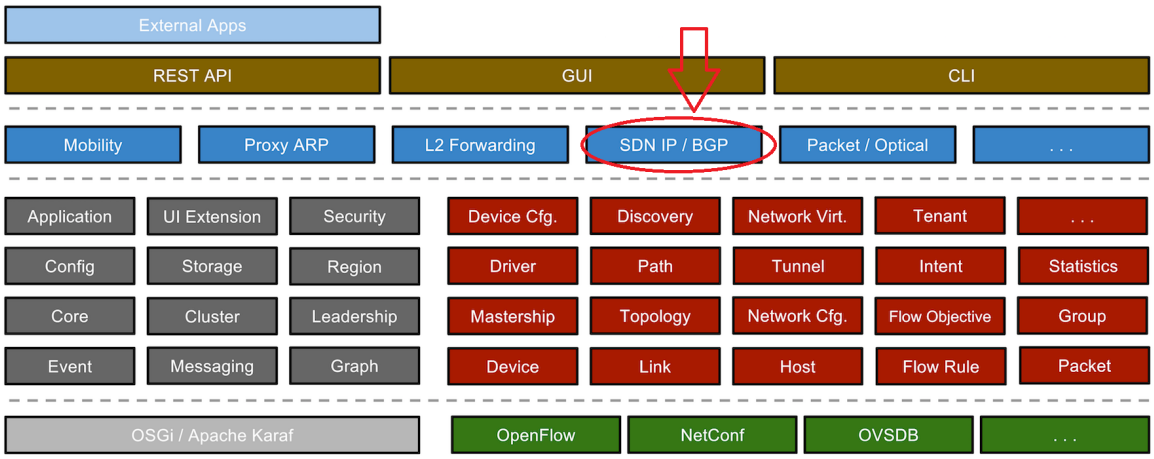

Figure 3.1: Our contribution in the ONOS subsystem

3.2 SDN-IP

3.2.1 Overview

SDN-IP [33] is an ONOS application that allows an SDN network to peer and exchange traffic with adjacent networks and to the rest of the Internet using the standard Border Gateway Protocol (BGP) [1]. Externally, from a BGP viewpoint, the SDN network appears and acts as a single Autonomous System (AS). Within the AS,

the SDN-IP application provides the integration mechanism between BGP and ONOS. From ONOS perspective, it's just an application that uses its services to install and update the appropriate forwarding state in the SDN data plane. At the protocol level, SDN-IP behaves as a regular *BGP speaker*.

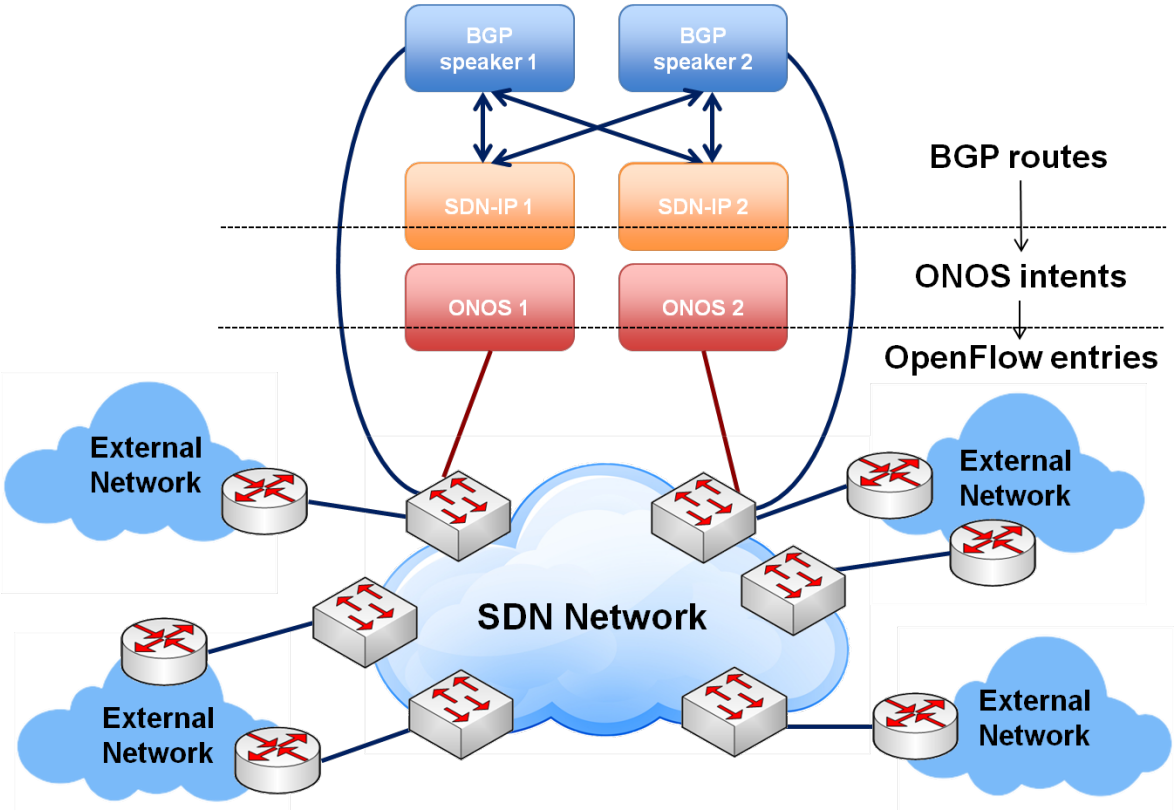

Figure 3.2: Architecture of SDN-IP

In a typical deployment scenario, we have SDN network composed by different OpenFlow switches, controlled by ONOS. ONOS runs as a cluster comprised of multiple instances cooperating, providing high performance, high availability and scalability.

Each external network is a different AS domain, which are connected at the edge of our SDN network through its *external BGP routers*. These routers will peer with the SDN network, more precisely to the data plane, through BGP. In order to communicate with the external BGP routers, the SDN network needs an IP address for each peering session. The IP addresses on the SDN network side are assigned to the BGP speakers. SDN-IP runs on a subset of ONOS instances and are fully connected to the BGP speakers. Only one instance of SDN-IP is active at the time (*SDN-IP Leader*) and is

responsible for making the appropriate ONOS API calls to install the necessary Intents at any given time, others are in stand-by mode and can take over in case the primary fails.

Into the SDN network, there are one or more BGP speakers. We can have existing BGP routers or any software that implements BGP (for example Quagga [34]). There are no specific requirements to implement a BGP speaker, it have to support both *eBGP* (external BGP, to exchange BGP routing information with the external routers of the adjacent networks) and *iBGP* (internal BGP, to propagate that information among SDN-IP application instances and themselves, that must be out-of band of the data plane).

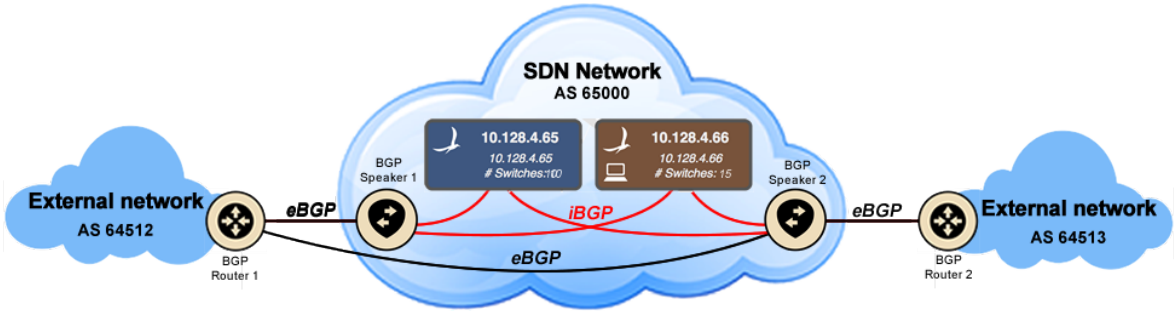

Figure 3.3: An example of a BGP configuration. The lines show BGP peering sessions.

The routes, advertised by the external border routers, belonging to external networks, are received by the BGP speakers within the SDN-IP network, processed, and eventually re-advertised to the other external networks. The routes are processed according to the normal BGP processing and routing policies. Similarly, those routes are also advertised to the SDN-IP application instances which act as a passive iBGP peer: it listens to BGP updates but it never advertises updates of its own. The best route for each destination is selected by the SDN-IP application according to the iBGP rules, and translated into an ONOS Application Intent Request. ONOS translates the Application Intent Request into forwarding rules in the data plane (*OpenFlow entries*). Those rules are used to forward the transit traffic between the interconnected IP networks.

In addition, SDN-IP creates the appropriate ONOS Intents that would allow the external BGP routers to peer with the BGP speakers by using the data plane. Thus,

ONOS itself can also detect failures in the data plane that affect the transiting traffic and act accordingly.

3.2.2 Intent Framework

Intent Framework [35] is an ONOS subsystem, that allows applications and operators to define policies using an high-level abstraction or language. We refer to these policy-based directives as **Intents**. An Intent is an immutable model object that describes an application's request to the ONOS core to change the network's behavior. At the lowest levels, Intents may be described in terms of:

- **Network Resource**: A set of object models, such as links, that tie back to the parts of the network affected by an Intent;
- **Constraints**: Weights applied to a set of network resources, such as bandwidth, optical frequency, and link type;
- **Criteria**: Packet header fields or patterns that describe a slice of traffic. An Intent's **TrafficSelector** carries criteria as a set of objects that implement the Criterion interface;
- **Instructions**: Actions to apply to a slice of traffic, such as header field modifications, or outputting through specific ports. An Intent's **TrafficTreatment** carries instructions as a set of objects that implement the Instruction interface.

It is responsibility of the ONOS controller to translate those policies into a network configuration changes, such as tunnel links being provisioned or flow rules being installed on a switch. ONOS does it with *Intent compiler*. Once the controller receives the Intent, it is assigned a unique *IntentID*, as well as being tagged with the *ApplicationID* of the application that sent it. Other parameters that Intent should contains are: Intent Key, Priority, Network Resource. Different kind of Intents might contains other attributes as an example Ingress Port(s), Egress Port(s) or Encapsulation Type.

Furthermore, ONOS compiles Intents into installable Intents and installs rule (a set of **FlowRule**) into specific network devices. From where, the Intent moves, asyn-

chronously through to the *compilation phase* to process the request. After the compilation phase, things move into the installing phase then end with an installed state. If the changes can't be made, they are moved into a failed state.

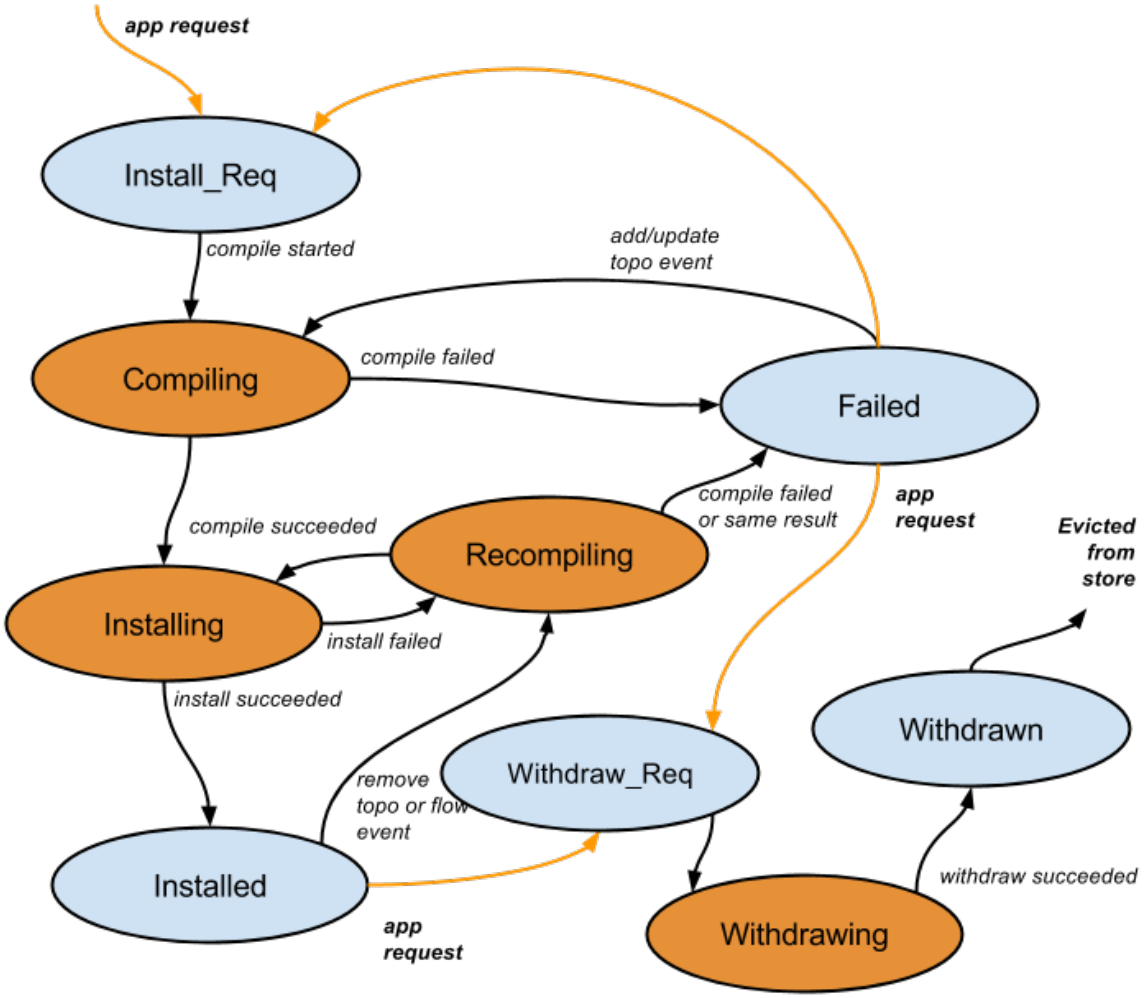

Figure 3.4: State transition diagram for the compilation process of each top-level Intent

3.2.3 SDN data plane connectivity

The data plane is used for:

- To carry the BGP control traffic among the internal BGP speakers and the external BGP routers (the eBGP peerings).

- To carry the transit data traffic among the external IP networks that traverses the SDN-IP network.

The eBGP control traffic (can be point-to-point or bidirectional) is associated with each eBGP peering. The path between end-points might change due to a data plane failures which are automatically detected by ONOS which, consequently, reroute the associated Intents.

The paths for the transit data traffic are defined by the BGP routes, advertised by the external BGP peers. If a BGP peer advertises a route for a specific IP prefix, and is chosen as the best next-hop for this route, SDN-IP is responsible for creating the corresponding data paths. All traffic destined to that IP prefix, entering from the remaining external IP networks, needs to be forwarded toward the best external next-hop BGP router. This procedure is done by ONOS Application Intents that creates a Multi-Point-to-Single-Point Application Intent for the IP prefix with the egress router (i.e. the best next-hop router) that is the single (egress) point for the Intent and the remaining external BGP routers are the (ingress) Multi-Points for the Intent.

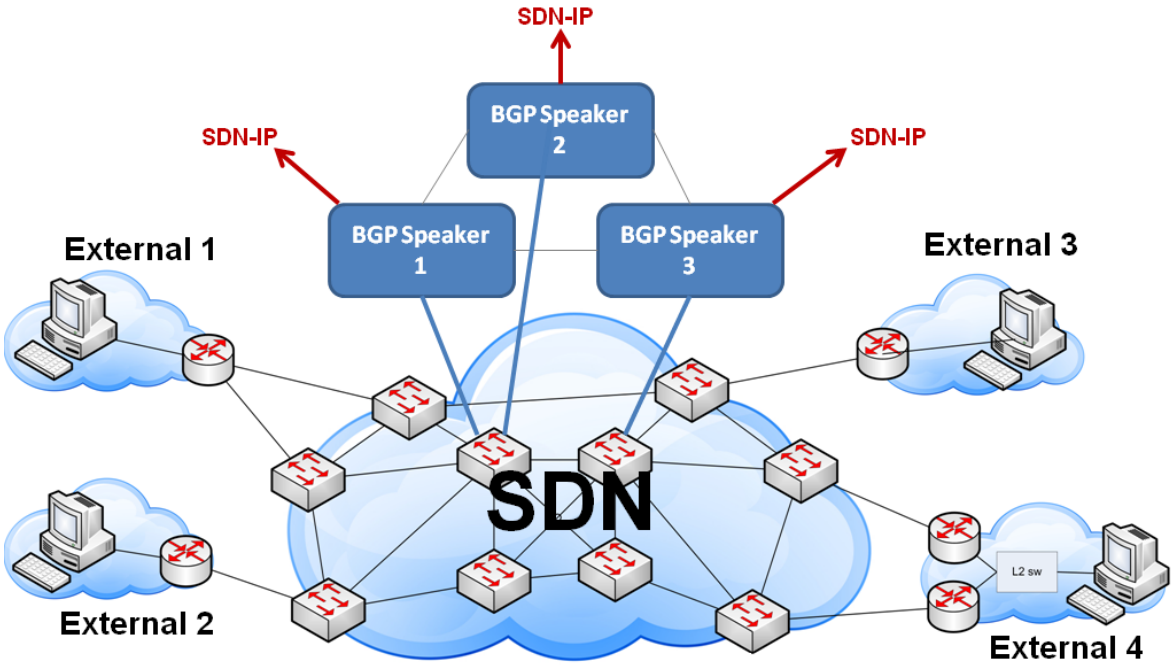

Figure 3.5: Structure of SDN Data Plane

The SDN-IP application is responsible for generating MultiPointToSinglePoint Application Intent requests and for updating those Intents in response to the BGP routing

dynamics. ONOS itself is responsible for compiling those requests, installing the corresponding forwarding flows in the data plane of the switches and for rerouting the Intents in case of failures within the SDN network itself.

3.2.4 SDN control plane connectivity

The BGP speakers within the SDN network and the SDN-IP application instances communicate using iBGP. The peering sessions are created in the control plane, hence each BGP speaker needs to be connected to it.

SDN-IP implements a subset of the iBGP protocol: it only receives and processes BGP routing information from the BGP speakers, but it never originates or retransmits BGP routes.

Once SDN-IP receives the advertisements from the external routers (received via the BGP speakers), it transforms the routing information into Intent requests to ONOS. ONOS itself then translates them into OpenFlow entries on the switches.

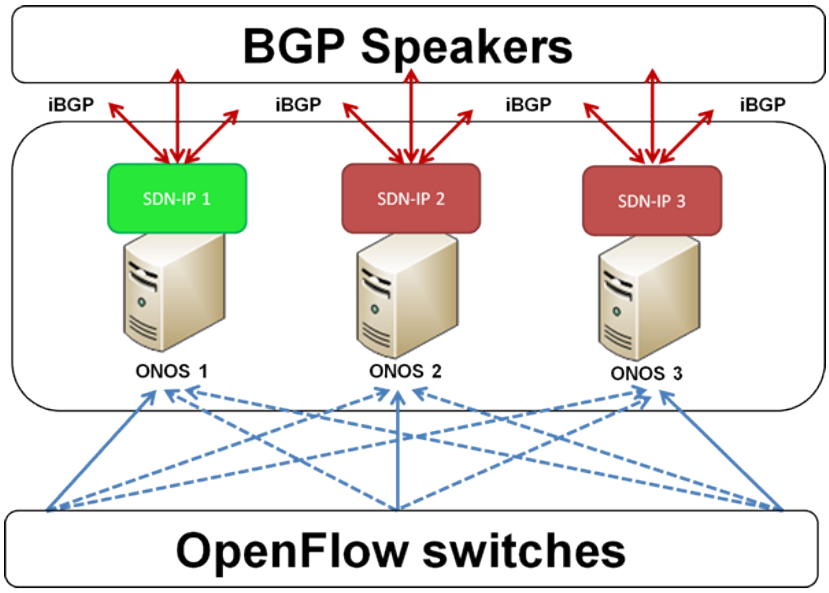

Figure 3.6: Structure of SDN Control Plane

Since the BGP speakers use iBGP to propagate the routing information to the SDN-IP application instances, it is important to inter-connect them as in a regular BGP deployment: through a full mesh or using a route reflector server. Thus, we can utilize the failure resistance provided by BGP itself. Note that no iBGP peering is

needed between the SDN-IP application instances: the peering is between the BGP speakers and the SDN-IP application instances.

3.2.5 Application Intents used by SDN-IP

In this paragraph we will examine the two type of Application Intents used by SDN-IP: **PointToPoint** and **MultiPointtoSinglePoint**.

The first are relative about the uni-directional Intents used to establish the BGP peering session between the external routers and the SDN BGP speaker (two single attachment points in the SDN network). Each attachment point contains the SDN switch DPID (*Datapath ID*, a unique identifier for the switch), the switch port, and the MAC address of the relative BGP speaker/router.

The second one, are uni-directional Intents used to connect the subnets of the external networks together. Each Intent is associated with the IP prefix of the destination, and connects, with a single egress attachment point, the best next-hop router towards the destination IP prefix. At the ingress edge of the SDN network, an IP packet is matched on the destination IP prefix, using the longest network prefix. When the selected entry is chosen, the MAC address of the packet is changed with the MAC address of the egress IP router of the destination and afterwards forwarded to the egress port.

3.3 Mininet

Mininet [36] is a lightweight container orchestration system for network emulation. It supports a rapid development model for SDN, with flexible and easily reconfigurable topologies and configurations that starts a network in few seconds. It creates a realistic virtual network, running real kernel, switch and application code, on a single machine (Virtual Machine, cloud or native) with a single command. Mininet hosts run standard Linux network software, and its switches support OpenFlow for highly flexible custom routing and Software-Defined Networking .

Mininet supports research, development, learning, prototyping, testing, debugging, and any other tasks that could benefit from having a complete experimental network

on a laptop or other PC. In fact:

- Provides a simple and inexpensive network testbed for developing OpenFlow applications;
- Enables multiple concurrent developers to work independently on the same topology;
- Supports system-level regression tests, which are repeatable and easily packaged;
- Enables complex topology testing, without the need to wire up a physical network;
- Includes a CLI that is topology-aware and OpenFlow-aware, for debugging or running network-wide tests;
- Supports arbitrary custom topologies, and includes a basic set of parametrized topologies is usable out of the box without programming;
- Provides a straightforward and extensible Python API for network creation and experimentation;

Mininet provides an easy way to get correct system behavior (to the extent supported by your hardware) and to experiment with topologies. Mininet networks run "real code", including standard Unix/Linux network applications, as well as the real Linux kernel and network stack. Thanks to this, the code developed and tested on Mininet (for an OpenFlow controller, modified switch, or host) can move to a real system with minimal changes, for real-world testing, performance evaluation, and deployment. Most importantly, this means that a design that works in Mininet can usually move directly to hardware switches for line-rate packet forwarding.

Chapter 4

Smart Intents for Onos

This chapter is the core of this work: here we present how we extend the SDN-IP application and the interactions with the external rerouting logic. We divide the explanation in three parts. In the first part, we describe the legacy SDN-IP and the step to build the network and the communication among the routers and switches. In the second part, we illustrate our extension to the SDN-IP application. Finally, we explain the connection with the off-platform optimization module that will handle the collection of traffic matrices and the computation of the optimal set of robust routings to be applied to the considered network.

4.1 Legacy SDN-IP

As we have already said, SDN-IP helps network operators to use SDN to connect network to the rest of the Internet using BGP protocol. In the figure 3.2, we can see the global view of the architecture of the SDN-IP. To create the network to test the application, we will use Mininet that allows to create a virtual network with switches, routers and the links.

In figure 4.1, we have an example of SDN network which contains 6 OpenFlow switches (blue square) that build up the network. The lateral circles represent the *external BGP routers*, while the above symbol represents the *internal BGP speaker* located in our SDN network. This last node has the peering function with all the external

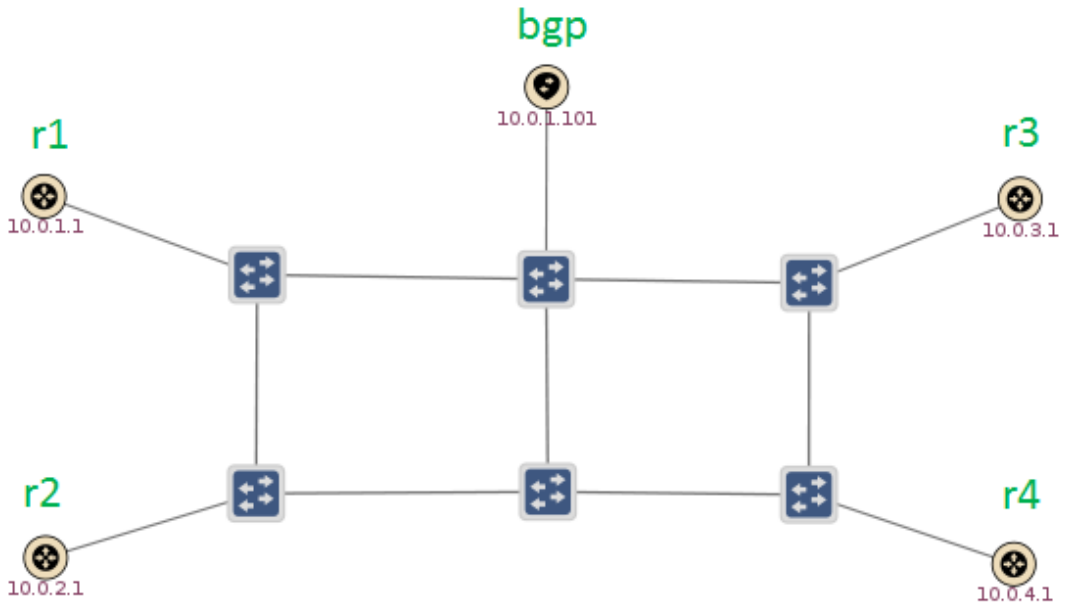

Figure 4.1: An example of SDN Network

BGP routers, learns BGP routes from them, and relays those routes to the SDN-IP application running on ONOS. The external BGP routers are the border routers that reside in other networks, that have some subnets, that want to exchange traffic with them. Subnets are not visible by SDN-IP, because are under an external administrative domain. The behaviour of external routers and the BGP speaker are emulated by a software called *Quagga*. Quagga [34] is a routing software package that provides TCP/IP based routing services with routing protocols support such as BGPv4. Quagga permits us to simulate external BGP routers belonging to other administrative domains. The goal of SDN-IP is to be able to talk BGP with these routers in order to configure the network to permit traffic exchange between the different external ASes.

4.1.1 Into the code

Running an emulated SDN network controlled by ONOS requires two main components: the first one is the fundamental one that allows you to start the system, then the ONOS core and applications. This part is written in Java. The second component

is in charge of creating the network and its components. This part, instead, is written in Python.

In the Python part, we can find the Mininet script, which reads network specifications from a few JSON file, and creates all the objects of the network (routers, hosts, links and the assignment of IP addresses). Currently, SDN-IP only supports reading static configuration, placed in a file called *network-cfg.json*, read at startup and pushed into ONOS.

In this extract of the configuration file, there are two main blocks:

```
------------------------------------------------------------

    "ports" : {
        "of:0000000000000005/4" : {
            "interfaces" : [
                {
                    "name" : "sw5-4",
                    "ips"  : [ "10.0.1.2/24" ],
                    "mac"  : "00:00:00:00:00:01"
                }
            ]
        },
    }, ... ...,

    "apps" : {
        "org.onosproject.router" : {
            "bgp" : {
                "bgpSpeakers" : [
                    {
                        "name" : "speaker1",
                        "connectPoint" : "of:0000000000000001/7",
                        "peers" : [
```

```
                        "10.0.1.1",
                        "10.0.3.1",
                        "10.0.7.1"
                    ]
                },
            ]
        }
    }
}
```

- The top part contains the *ports section* that specifies an *interface* on each switch port that connects to an external BGP router. An interface is a set of addresses that are logically mapped to the switch port. The minimum information required are an IP address and a MAC address. This section allows SDN-IP application to define which ports are designed as peering interfaces.

- The bottom part contains the *bgp section* that specifies the configuration about the internal BGP speaker. Each BGP speaker has a connnectPoint (switch port where BGP speaker is physically plugged in) and a list of peer IP addresses with which the BGP speaker is peering with. This list of addresses should have a corresponding address into the ports section because SDN-IP will determine which port the peer is connected to find an interface with an IP address in the same subnet. When SDN-IP determines the association between the two components, it will install Intents to allow BGP traffic to flow through the network between these two points.

The other files that the script needs are the BGP configuration, one for each router. These files are read from Quagga software and contains the subnet(s) attached to the relative router and other parameters such as the advertisement time and the assigned IP.

Below we can see an example of BGP configuration file for a router with two attached subnets.

```
-------------------------------------------------------------


BGP configuration for r1
hostname r1
password xyz
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
  network 192.168.10.0/24
log stdout


-------------------------------------------------------------
```

The Java part contains the engine of the system. When ONOS starts, all the applications within the ONOS_APPS environment variable are activated, including SDN-IP. At this point, the application reads the configuration files, responds to ARP requests between the external BGP routers and internal BGP speaker and will install Intents in the network to establish connectivity for the BGP peering sessions. Then it will begin to receive routes and translate them to MultiPointToSinglePoint Intents that will be finally compiled to low-level OF messages.

The generation of a MultiPointToSinglePoint Intent into SDN-IP occurs by *generateRouteIntent()* function that it is triggered for each BGP announcement received. From the announcement, SDN-IP application obtains the announced destination IP

prefix, the IP and the MAC address of the external BGP router from where the message come from. Based on these information, the application will generate an Intent matching the destination IP prefix and rewriting the destination MAC address for packets coming from any other switch connected to an external BGP router. Those packets will be routed towards the external BGP router from where the announcement has been received. The output of this function will be a MultiPointToSinglePoint object that will be submited to the *IntentSynchronizer* to get finally installed into the system.

4.2 New Version: Extended SDN-IP

MultiPointToSinglePoint aggregates multiple traffic as sources matches on destination IP prefix. This doesn't allow to study the statistics of the flows correctly, from the Intents, since we can't determine where the traffic comes from. To overcome the problem, instead of installing the MultiPointToSinglePoint Intents, as does the legacy version, we create the connection between the subnet as PointToPoint routing by SRC/DST IP prefix.

In the following subchapter we will go into details about the creation of PointToPoint Intent, how we create and handle the objects to be sent to the off-platform module and how ONOS manages our routing changes within the network.

4.2.1 PointToPoint Intent

When the application starts, we have to create three listeners to core system services to get information about the topology and flows. The first allows us to take notice about components within the network (e.g. IP prefix of the subnet, information of the links). The second will be used to collect information relating to flows already in the environment. The core triggers the statistics manager every 5 seconds and the listener, while listening to the core, is able to get the event with its information. The third allows us to notice announcements that come to the network (BGP announcements) and create PointToPoint Intents routing with SRC/DST IP prefix. To simplify the

correct explanation of information, we will call the first listener as *interfaceService*, the second as *flowRuleService* and the last as *routeService*.

The creation of a source-destination (SRC-DST) Intent occurs every time *routeService* receives an event, so a new BGP announcement. The creation happens in some steps after we have received at least two announcements from two different peering interfaces. A couple of announcements received from the same external BGP router doesn't imply the creation of an Intent because that traffic will be handled outside the SDN network. As in the MultiPointToSinglePoint, the new function (called *generateSrcDstRouteIntents()*) gets in input the same parameters. Using the interfaceService, we find the connection between the switch and the external router of the input subnet, using the IP prefix of the external router found in the event. Afterwards, it updates the list of announced IP prefix and the MAC address detected from the ConnectPoint (CP) of the received announcement. Finally, through the interfaceService and the announced IP prefix in the event, we find the interface between the switch and the external router connecting the announced subnet (i.e. the destination).

When all necessary parameters have been found, it can create two Intents, forwarding on the shortest path (ONOS can't create bidirectional Intent independently, so we have to call two times the generation function of the Intent).

All these steps are redone for every BGP announcements received by the internal BGP speaker using as a destination all the possible peering interfaces, different from the one where it receives the announcement, discovered and saved in the list of announced IP prefix.

4.2.2 Setup of AS-to-AS Traffic Matrices (TMs)

Whenever a new announcement arrives, in addition to the generation of the Intent, SDN-IP will initialize the TMs of the demand. Ideally, a Traffic Matrix (TM) describes the instantaneous data rate for each couple of SRC-DST nodes at a given time. In fact, into the object, we will save the key, formed as SRC-DST IP prefix, the amount of bytes collected until that moment and its time instant.

Every 5 seconds, flowRuleService is triggered with the RuleUpdate event. From the

object event, it extracts the information about the flow to calculate and update TMs. These information will be used to create a set of sampled TMs that can be requested from the off-platform optimization module. Every time a sample is sent to the module, it will be deleted (i.e. consumed) from ONOS. Each sample is timestamped by the controller to make the external module able to align correctly the samples received.

4.2.3 Information exchange between ONOS and the external module: REST API

Since ONOS and the module are not directly connect in the same environment, because running an optimization tool in the same machine running ONOS, would kill the high performance key requirement of ONOS, so we need to find a way to exchange information easily and quickly. **REST** (REpresentational State Transfer) is the underlying architectural principle of the web. The amazing thing about the web is the fact that clients (browsers) and servers can interact in complex ways without the client knowing anything beforehand about the server and the resources it hosts. The key constraint is that the server and client must both agree on the media used, which in the case of the web is HTML. An *API*, that adheres to the principles of REST, does not require the client to know anything about the structure of the API. Rather, the server needs to provide whatever information the client needs to interact with the service. An HTML form is an example of this: the server specifies the location of the resource and the required fields. The browser doesn't know in advance where to submit the information, and it doesn't know in advance what information to submit. Both forms of information are entirely supplied by the server.

So, how does this apply to HTTP, and how can it be implemented in practice? HTTP is oriented around verbs and resources. HTTP methods like Create, Retrieve, Update, Delete becomes POST, GET, PUT, and DELETE.

Exploiting the already currently implemented REST API in ONOS and through 4 specific functions, we are able to retrieve statistic information and, after the optimization phase, forward and apply new routing configurations. The format used to exchange information is JSON (JavaScript Object Notation). In figure 4.3, we can see

Figure 4.2: Connection between ONOS and the off-platform optimization module with SSH over Internet

the iterations between ONOS, the external module and the functions used with REST API.

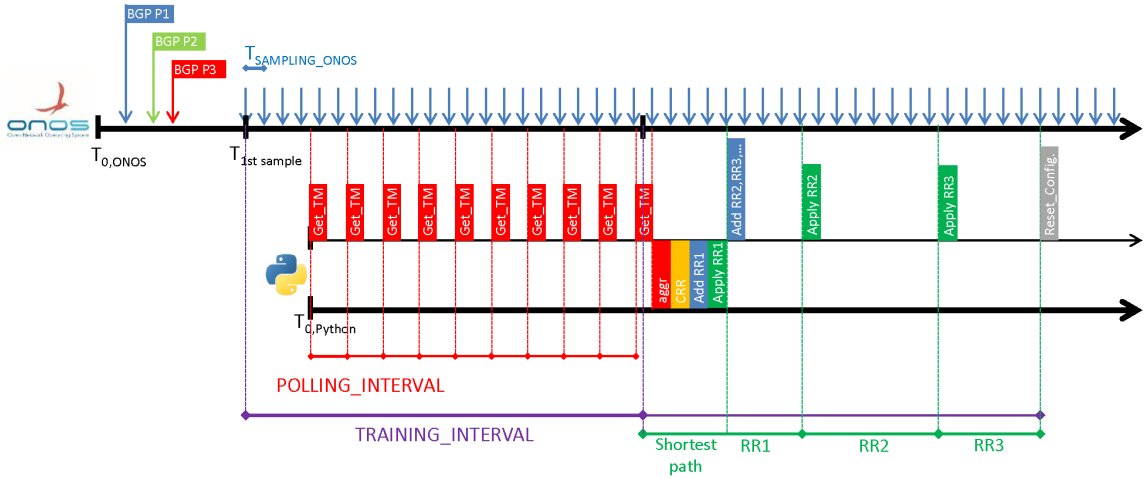

Figure 4.3: Iterations between ONOS and the external module for a day

So, let's describe the various interactions between the two systems.

- **First step - Get_TM**: After creating the Intents, ONOS updates all flow information of all devices every TSampling_ONOS seconds. During the update, the application creates the samples (which contain the timestamp of the controller, the key of the demand and the amount of bytes exchanged) of the demands cre-

ated with PointToPoint Intent. Then, the off-platform module, through REST, retrieves the samples for a *Training_Interval* to estimate the set of TMs that will be used as input to the optimization model to minimize the Maximum Link Utilization (MLU). Collected samples "are deleted" by ONOS. The instant when it receives the first sample, determines the beginning of Training_Interval. This allows the module to sync, even if it is launched late respect to ONOS. The module collects samples from ONOS every *Polling_Interval* and extracts bytes and timestamp to calculate the average bitrate with the required granularity. The time difference is calculated as the difference in time between the sample timestamp (it takes at least two samples for *Aggregation_Interval*, which must have a value at least twice respect of TSampling_Onos) until the end of the Training_Interval.

But what happen if new announcement arrives or if an Intent is removed (because a subnet is removed) during the Training_Interval? Simply, the algorithm polls until removal (or from when it is added to the end of Training_Interval, in case of a new announcement arrived) and will make a contribution of 0 bit/s for those demands.

Below an example of JSON response of Get_TM:

```
{
    "response": [
        {
            "timestamp": 1502806046,
            "demand": "192.168.1.0/24=192.168.2.0/24",
            "bytes": 0
        },
        {
            "timestamp": 1502806046,
            "demand": "192.168.10.0/24=192.168.2.0/24",
```

```
            "bytes": 30
        }, ... ...
    ]
}
```

- **Second step - Add_Routing & Apply_Routing**: Finished the Training_Interval, the external module computes CRR (see the next subchapter for more information), loads and activates the first Robust Routing and, finally, schedules the activations of subsequent RRs. All network operations are made in a separate thread to avoid losing synchronization, still based on the instant of the first sample. If the algorithm fails to compute a new CRR with the new data or the new CRR is not feasible, the last RR applied in the system is maintained.

  Below an example of JSON response of Add_Routing:

```
{
    "routing_list": [
        {
            "r_config": [
                {
                    "demand":
                        ["192.168.1.0/24",
                        "192.168.2.0/24"],
                    "paths": [
                        {
                            "path":
                                ["of:00000000000000a1",
                                "of:00000000000000a2"],
```

```
                        "weight": 1.0
                    }
                ]
            }
        ],
        "r_ID": 0
    }
]
}
```

Within JSON, CRR provides K set of routing configurations to apply with an identification index (r_ID). Each configuration has, for every demand, a set of OF switches (of:00000000000000a1) on which the demand must be routed. The creation of the Intent is left to ONOS that creates an Intent by imposing that the connection is made using the OF switches passed in the JSON. This can be done using a WaypointConstraint. ONOS, however, can't handle the Intent as PointToPoint with a WaypointConstraint because the set of paths filtered by the compiler, according to the set of nodes (Waypoints), include only the shortest paths. In some cases, the off-platform optimization module selected a longer path to minimize the MLU so, in this cases, the Intent compilation would fail. The solution that we adopted is to create a LinkCollection Intent that permits us to force a new path even different from the shortest path. The drawback we found is that it doesn't take into account protection so ONOS is not able to automatically recover from failures because the set of links to be traversed become part of the objective.

- **Third step - Reset_Routing and Repeat**: The last operation is performed only in some cases after applying all RRs. The reset operation of the loaded RR configurations within the module, is performed only if we are interested in reiterating collect&apply operations for several days. This allows the system to

be studied, with the applied RRs, to try to further improve the MLU with the traffic in the various days until the end of the simulation time.

4.2.4 CRR computation

In this subchapter, we briefly describe how the module computes the **CRR** (Clustered Robust Routing). The complete description and the experimental results can be found in another Master Thesis [37]. The reference model is **Time&Routing-aware clustering**.

While that thesis defines the optimization model, in this thesis, we implemented it in a complete SDN framework. The CRR algorithm is implemented as an off-platform module of the network controller. It takes as input a set of TMs representative of the period in which robust routing configurations should be designed and groups into clusters to compute RR configurations over them. It splits the TM domain into N clusters and computes, for each subset of TMs, a routing configuration, which is robust against any possible traffic variation within the cluster. To avoid oscillation between routing configurations, a cluster is built with a minimum time length that results in a minimum utilization of the same routing configuration. These TMs can be obtained in several ways: they can be measurements from past network conditions, or the outcome of a TM prediction module, or even synthetically generated.

CRR is an iterative algorithm that achieves two objectives:

1. Covering the entire TM space so that a feasible routing configuration is available for any traffic condition.

2. Reducing the number of routing changes by creating a small set of robust routing (RR) configurations that can be used for a minimum duration whenever one of them is applied.

Chapter 5

Experimental Evaluation

In this chapter, we will examine the system performance by studying the application of the algorithm in different configurations. We will study 3 different CRR application versions that differ in number of days and the way we apply it, showing how our algorithm can reduce the Maximum Link Utilization (MLU) in a real case scenario.

5.1 Simulation Environment

To study the performance of the our model, we used the American network Abilene. This is an high-performance backbone network created by the Internet2 community [38] in the late 1990s that connects universities with high throughput links. This network is often cited in the publications since topology and traffic data are publicly available and find information for private network in Internet is very much difficult. Abilene was updated several times during the years but in our work, we want to use the configuration in 2004 with 11 nodes and 14 bi-directional links shown in figure 5.1.

All links have a capacity of about 10 Gb/s except one, between Atlanta and Indianapolis, of 2.5 Gb/s. Other information can be found in the website [39] [40] where there is also available almost 6 months of Traffic Matrices data from March to September 2004 with several breaks in between, so the set is slightly smaller. Each TM is made of 11 x 11 demands, where each value is an estimated average data rate in 5 minutes of measurements for each SRC-DST pair. Then, for each day, we have 288

Figure 5.1: Abilene backbone topology

TMs with granularity of 5 minutes (288 x 5 minutes = 1440 minutes = 24 hours). The configuration of the adopted network was introduced in April 2003 [41] [42] and worked until the following update in 2007 [43].

The reference version of software that we are used to develop our work are: Ubuntu 14.04.5 LTS [44], Python 2.7 [45] designed for working in Linux Environment, Mininet 2.2.1 [36], OpenVSwitch 2.5.2 [27], ONOS 1.8.0-rc4 [46], Apache Karaf 3.0.5 [47], Apache Maven 3.3.9 [48] and Gurobi 7.5 [49] as mathematical solver which include a library for Python.

5.2 About the version used

In the development and test part, we have implemented three versions to study the application of the algorithm. These three versions differ in the number of days we study the network.

1. The first version, the most ideal one, is the one that allows us to directly study the daily difference between the shortest path applied by ONOS and one provided by

CRR. This version allows us to validate how the communication between ONOS and the external module is correct and that allows us to show that we can force a routing into the application from the off-platform module. We collect TMs for a Training_Interval. Afterwards, the module computes CRR over this data and apply the new robust routings to the traffic data of the same period. We can think of it as if in the next day we had the same traffic of the previous day where we collected TMs. Otherwise, we can interpret it as if we had perfect knowledge of traffic at every moment of the training period. Quite unrealistic situation.

2. In the second version, we consider a more realistic scenario: we want to compare 2-days traffic in the legacy SDN-IP (both the days are forwarded on the shortest path) with the extended SDN-IP (the first day is forwarded on the shortest path, while the second day with the CRR computed over the first one). This version allows us to show how in a real case, the study of data for a given period allows to improve the MLU of the network. We may think it as if we had studied the network for an entire day and, the following day, the algorithm (and therefore the various RRs) is applied.

3. In the last version that we consider, we will apply the algorithm during more days. The idea is always the same. It collects TMs for a Training_Interval, it computes CRR and applies the set of RRs during the next day but with a small difference. In this version, we want the external module is able to collect TMs and apply the CRR in the same Training_Interval. Let's explain the procedure better. During the first day, it collects and computes the various RRs and activation times over a network that forwards demands with shortest path rules. In the following day, it applies CRR based on the TMs collected during the previous day while the module continues to collect TMs on the current day. At the end of the Training_Interval, computes the new RRs to be applied to the next day. It repeats the following procedure for the next Training_Interval days.

5.3 System test

The simulations we will show are carried out on a single week of Abilene data. To generate traffic within the network, we use IPERF3. Due to some bugs in the program that can not properly generate some values of the TM (too low values are generated incorrectly causing inconsistent statistics), we had to impose TM values within a certain range.

The generation of traffic occurs with a function that creates, for each pair of subnets, the list of rates to generate. The Mininet script creates, for each pair, a list of traffic generations to be executed sequentially, each one long an Aggregation_Interval. Each sequence is launched in background/foreground with xterm, in parallel to the other. In order to coordinate ONOS, traffic generation and CRR, Mininet can automatically estimate how many Intents it will need to be installed (it does it by parsing the network configuration and Quagga files) so that it knows when all the announcements have arrived. Also, traffic does not start until a special UDP packet is received on port 12345. In this way, it can synchronize the start of the measurements between the external module and the beginning of the traffic transmission.

Example of traffic generation list

```
------------------------------------------------------------

TM_per_demand = {('192.168.1.1', '192.168.5.1'): [15000, 15000000, 15],
('192.168.2.1', '192.168.4.1'): [20000, 10000000, 150]}, ... ...

------------------------------------------------------------
```

In table 5.1, we list the main parameters used by the two systems (ONOS and off-platform module) in our simulations. For each parameter, we also include a brief description.

Because of the long CRR computing times for the choice of unsplittable paths, we had to force a smaller number of active nodes than the full network (exactly 5/11). This also allows us to have a first view of the gain on network congestion, allowing short

study times even in the third version. The active nodes we have considered are those of Washington, Chicago, Denver, Kansas City, Indianapolis. It should be noted that 5 minutes of data have been replied for 5 see, thus in a real world implementation, the application would have 60 x Time for solving the CRR. In addition, if we implemented the splittable routing in ONOS, solving time could be further decreased.

In the next chapters, we will show the application of our algorithm in all versions for a few selected days in three different graphics. The first graph is related to the variation of the bitrate of each active demand (we will only show traffic between two active network nodes for a total of 20 active demands. Each line represents a demand). On the X axis, we will show the variation of time, divided over the various days (In the first version, we have a single traffic day. In the second version, we have 2-days traffic and, finally, in the third version, we have considered the whole week of Abilene Data). On the y axis, we will show the bitrate values, sampled at each Polling Interval, measured in bit/s. Demands with higher bitrate values will be the most decisive in the calculation of the MLU. The second and third graph, we will show the variation of the MLU depending on time. As in the first graph, the time is divided into days according to the studied version. In the graphs, we will compare the variation of MLU in the shortest path case (blue line) and in case of CRR application (red line). We have also included a reference to the average MLU for each day and for each version (dotted line) so as to have a simpler comparison value between the two versions. The data used in the two graphs are different: in the second graph we will show an ideal version where TM data is statically read from a file while the third will show a measured version where the data are generated within our SDN-IP application and then read according to the procedure outlined in Chapter 4. In the measurements made in ONOS, we have to consider a part of the overhead created by the system on the links (Keep Alive Packets) and by the IPERF3 that we can not neglect or avoid.

Table 5.1: Main parameters values used in simulations

Name of variable	Value	Description
TSampling_ Onos	5	Update time for ONOS device statistics, in seconds
Polling Interval	5	TM samples polling interval, in seconds
Aggregation Interval	10	TM granularity, in seconds
Training_ Interval	600	TM gathering period before computing CRR, in seconds
Link Capacity	2e8	Capacity link of the virtual network (Mininet), in bit/s
Max_ Num_ of_ TM	288	Maximum number of TMs collected each day
Sampled_ Num_ of_ TM	60	Number of sampled TMs collected in a training interval
Sampled_ Num_ of_ MLU	120	Number of MLU samples computed in a Training_Interval
Max_ TM_ Value	1e8	Upper limit of TM values
Min_ TM_ Value	2e6	Lower limit of TM values
Min_ Lenght	1	Minimum number of consecutive matrices for which the same routing needs to be maintained
K	3	Number of blocks in which we divide active matrices

5.4 Use case: Version 1

5.4.1 Day 1

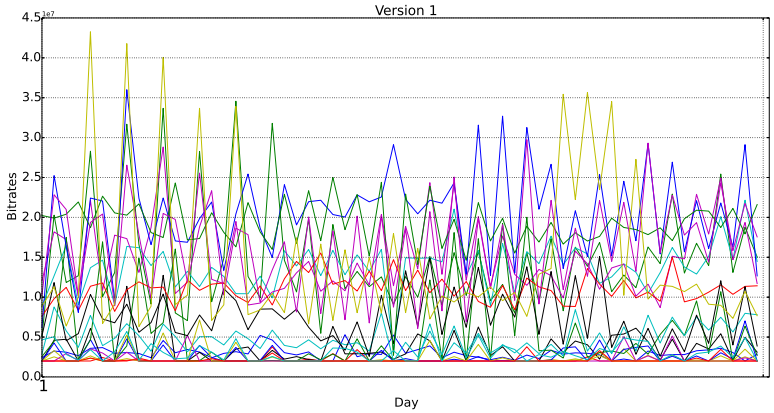

Figure 5.2: Bitrate measurement: Day 1 in Version 1

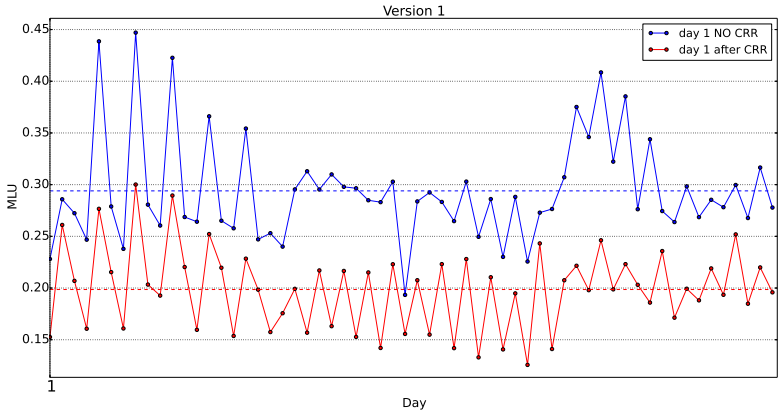

Figure 5.3: Ideal Version: Day 1 in Version 1

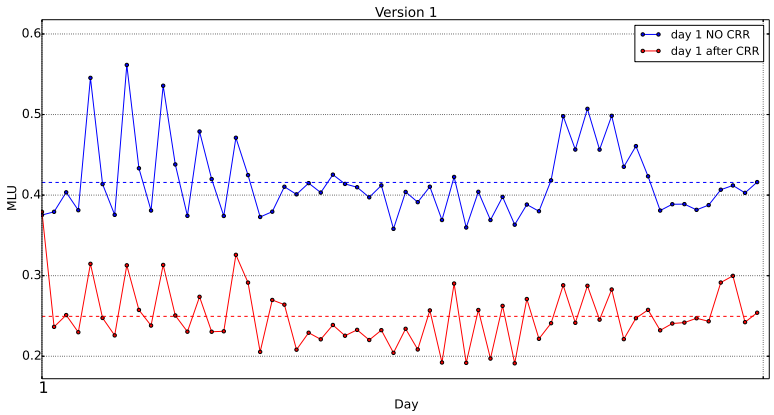

Figure 5.4: Measurement Version: Day 1 in Version 1

5.4.2 Day 3

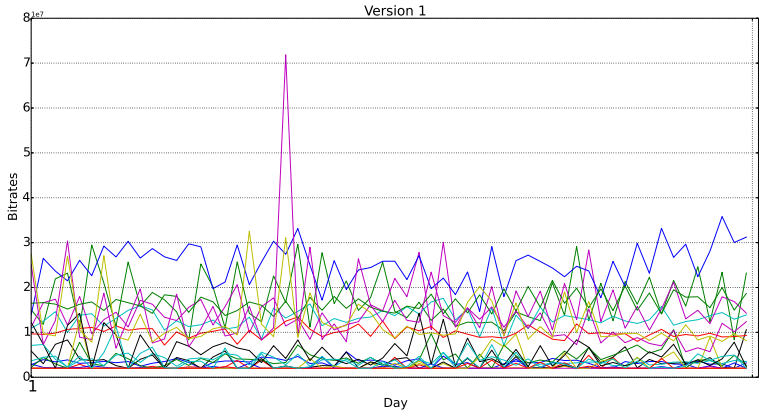

Figure 5.5: Bitrate measurement: Day 3 in Version 1

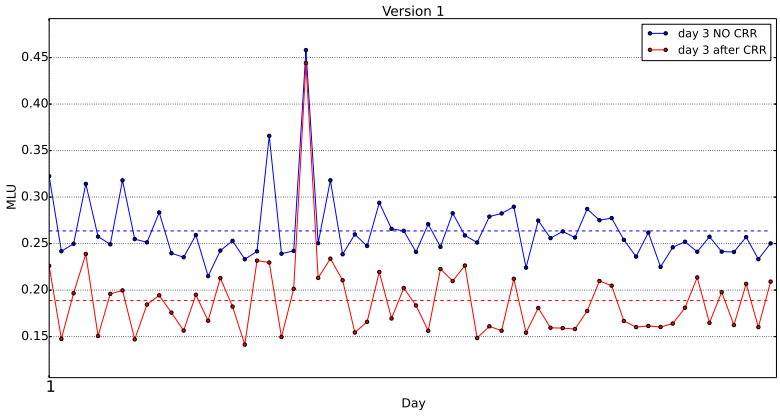

Figure 5.6: Ideal Version: Day 3 in Version 1

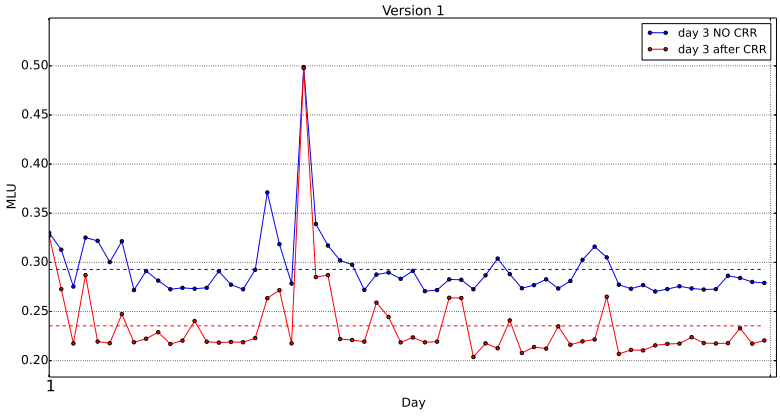

Figure 5.7: Measurement Version: Day 3 in Version 1

5.4.3 Day 5

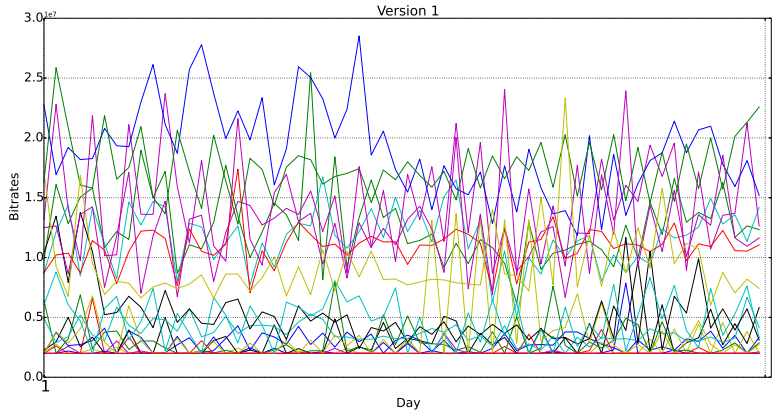

Figure 5.8: Bitrate measurement: Day 5 in Version 1

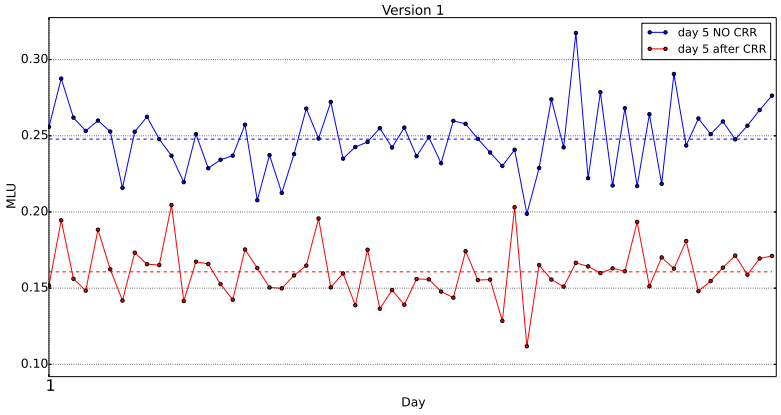

Figure 5.9: Ideal Version: Day 5 in Version 1

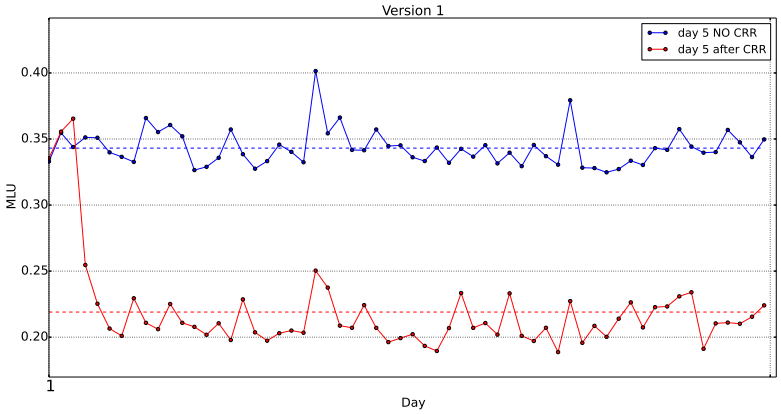

Figure 5.10: Measurement Version: Day 5 in Version 1

5.4.4 Day 7

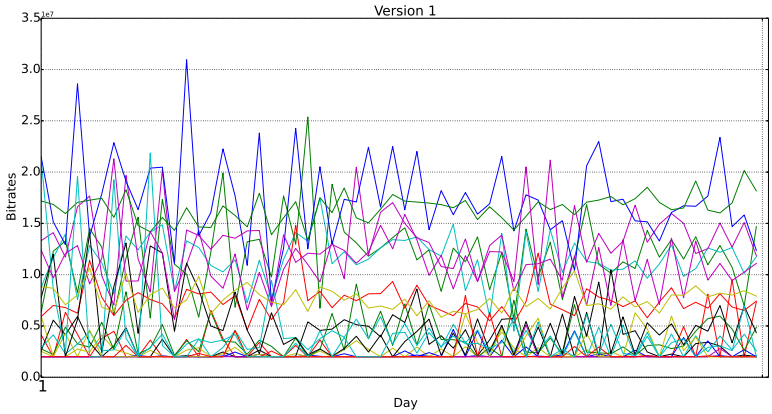

Figure 5.11: Bitrate measurement: Day 7 in Version 1

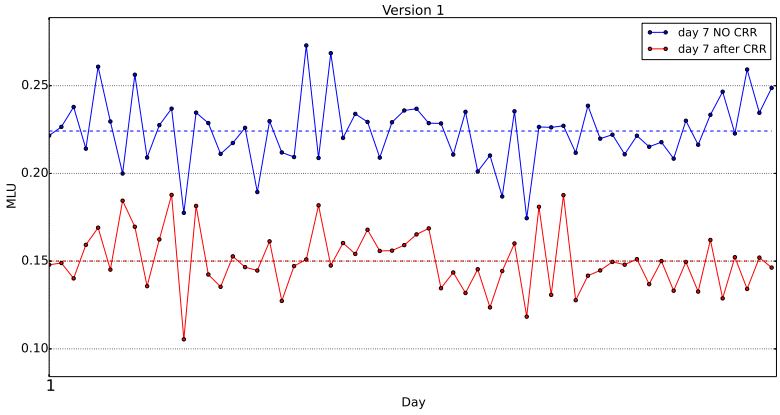

Figure 5.12: Ideal Version: Day 7 in Version 1

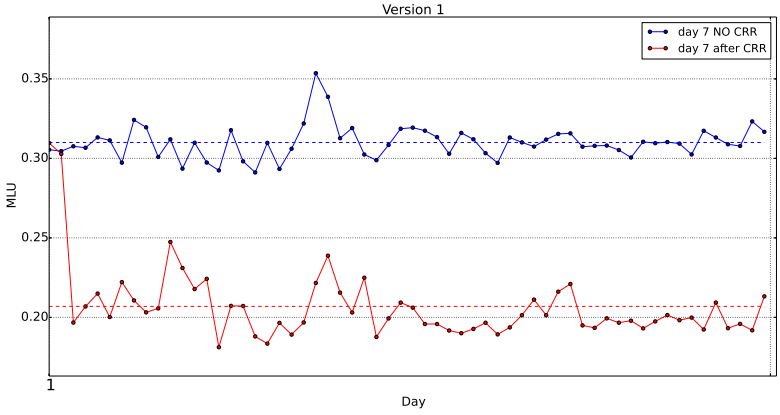

Figure 5.13: Measurement Version: Day 7 in Version 1

5.5 Use case: Version 2

5.5.1 Day 2-3

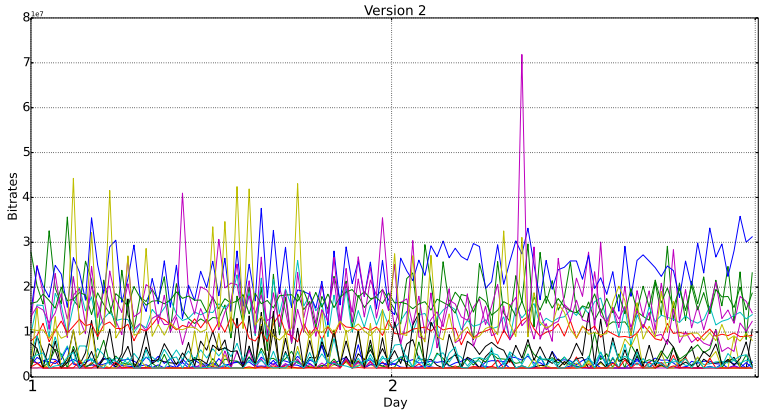

Figure 5.14: Bitrate measurement: Day 2-3 in Version 2

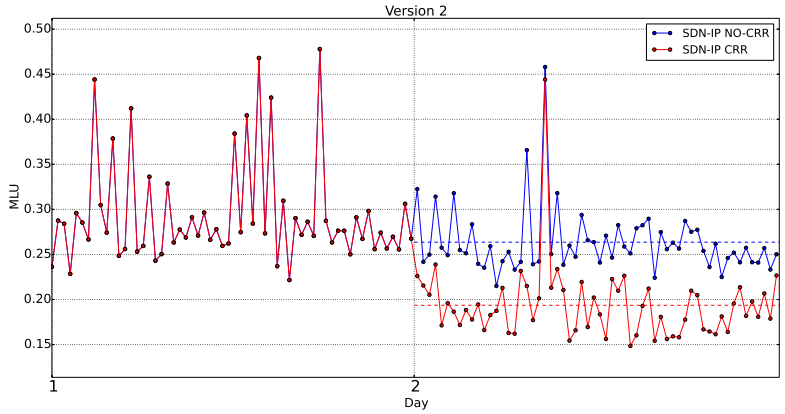

Figure 5.15: Ideal Version: Day 2-3 in Version 2

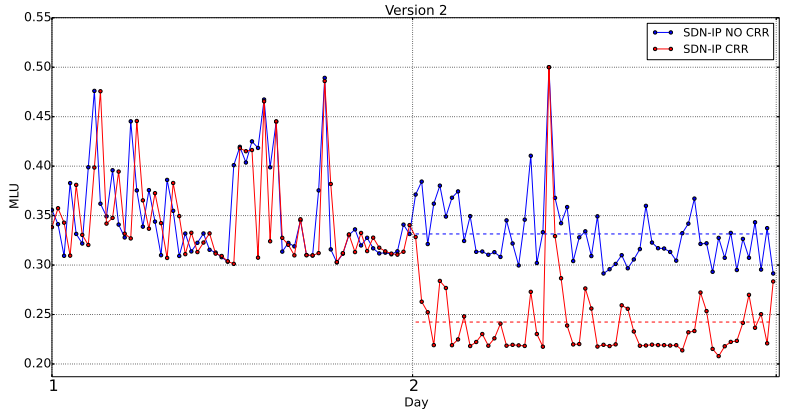

Figure 5.16: Measurement Version: Day 2-3 in Version 2

5.5.2 Day 4-5

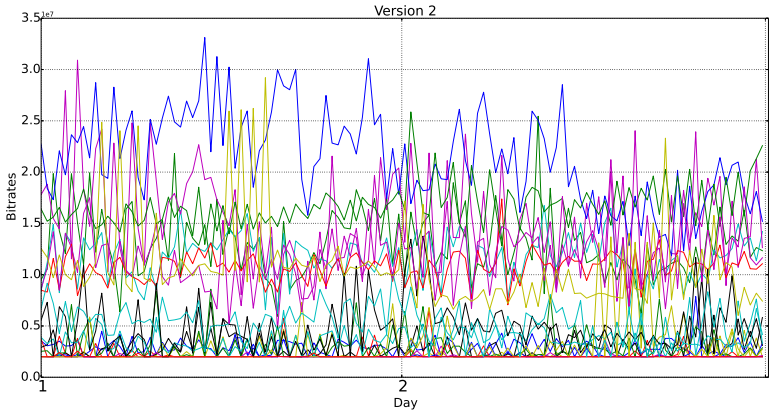

Figure 5.17: Bitrate measurement: Day 4-5 in Version 2

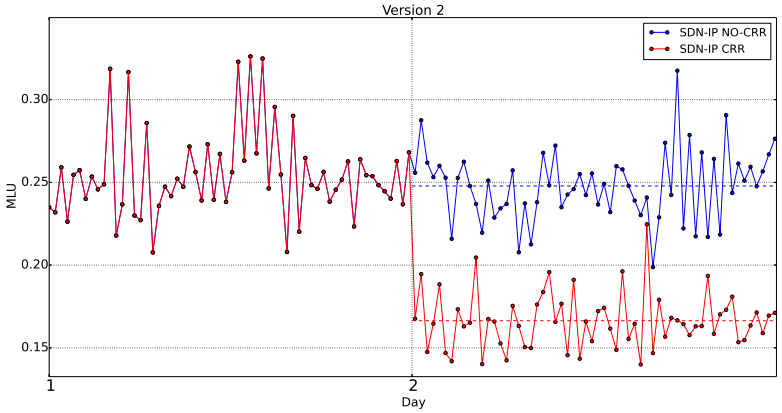

Figure 5.18: Ideal Version: Day 4-5 in Version 2

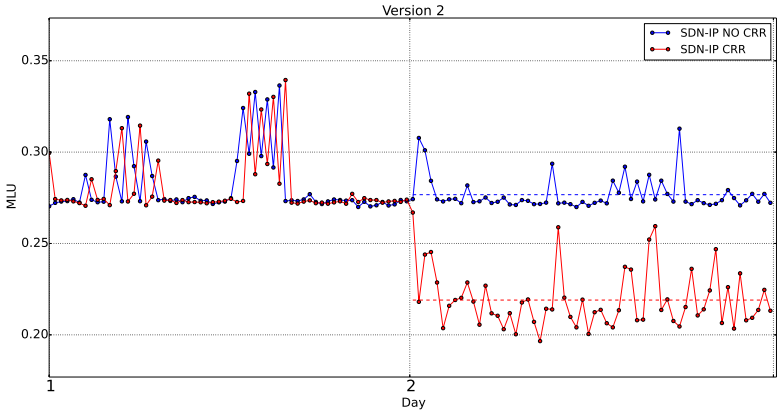

Figure 5.19: Measurement Version: Day 4-5 in Version 2

5.5.3 Day 5-6

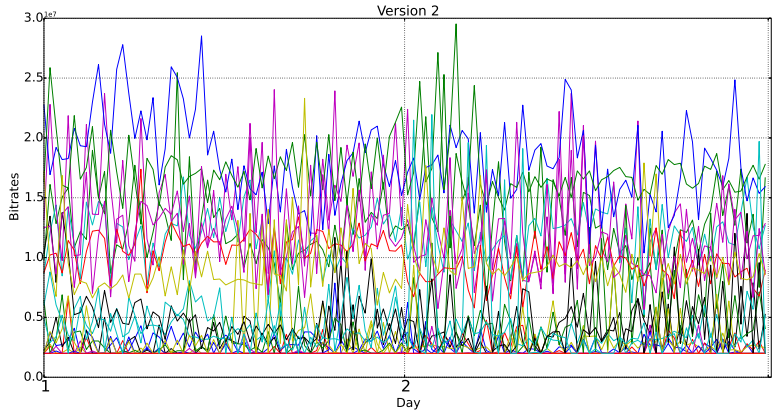

Figure 5.20: Bitrate measurement: Day 5-6 in Version 2

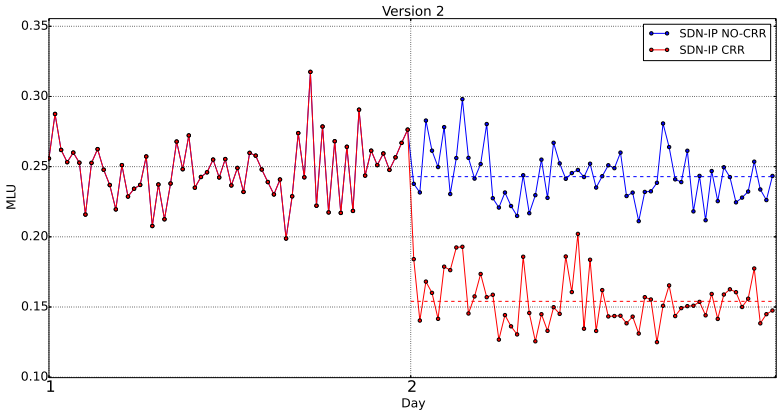

Figure 5.21: Ideal Version: Day 5-6 in Version 2

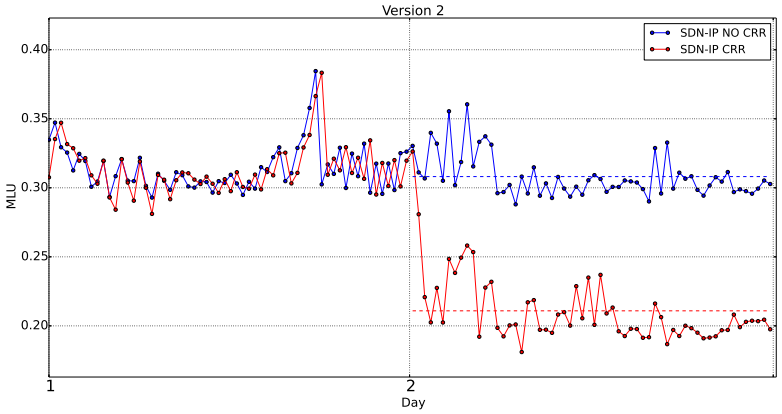

Figure 5.22: Measurement Version: Day 5-6 in Version 2

5.5.4 Day 6-7

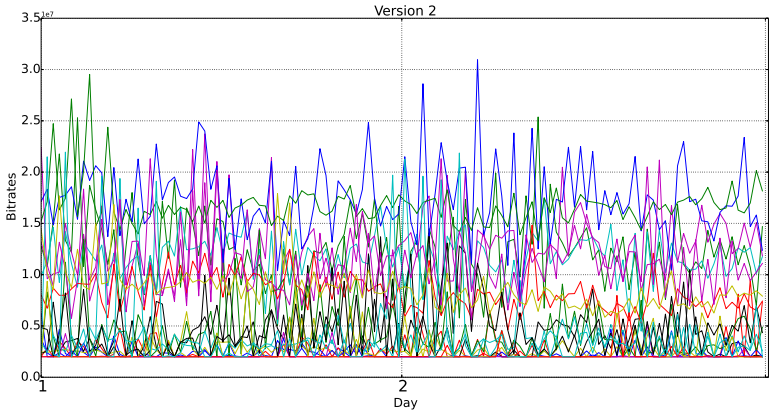

Figure 5.23: Bitrate measurement: Day 6-7 in Version 2

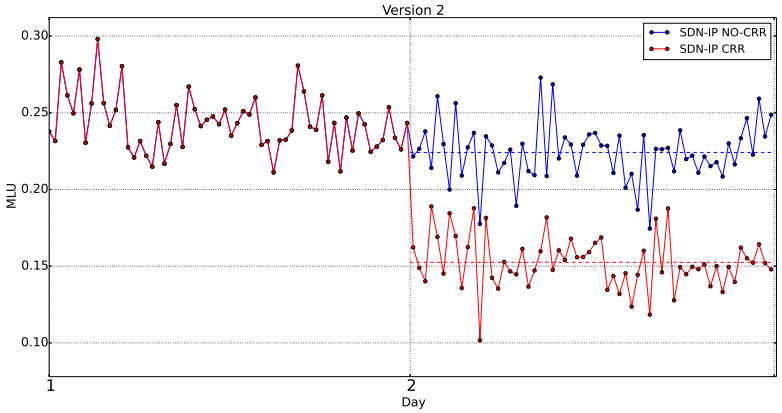

Figure 5.24: Ideal Version: Day 6-7 in Version 2

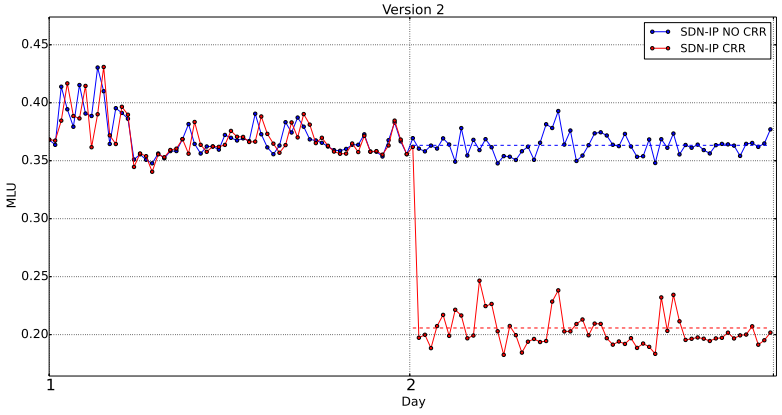

Figure 5.25: Measurement Version: Day 6-7 in Version 2

5.6 Use case: Version 3

5.6.1 Day 1 -> 7

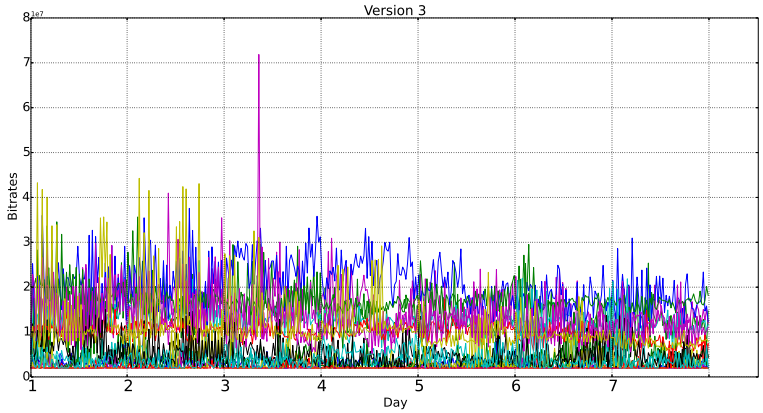

Figure 5.26: Bitrate measurement in Version 3

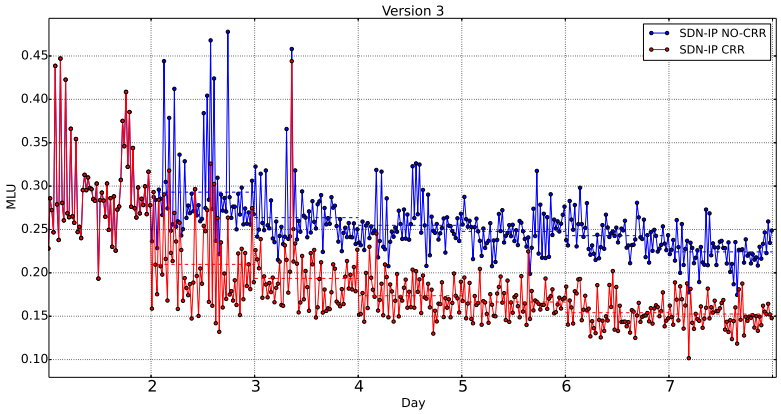

Figure 5.27: Ideal Version in Version 3

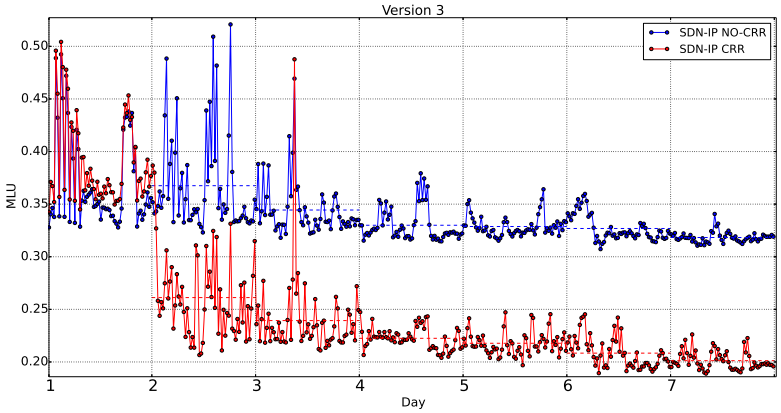

Figure 5.28: Measurement Version in Version 3

5.7 Results Considerations

From the study of the results, we can make some considerations by dividing them into some points:

- The first consideration we can make is on the measurement system given by our algorithm. Looking at the daily version, version 1, we can see how the measures taken, follow those shown in the ideal version (printed by reading the rate values directly from the files).

- The second consideration we can make, is the short reaction time of the system (ONOS) during the application of the first RR (and subsequent RR scheduling). Our system (CRR) works with polling and update statistics of 5 seconds (in reality, these times are much higher. Training_Interval times can also reach 24 hours, times that in our simulations were really too high and required too much computing resources).

- Another consideration we can make, the most important one, is the gain that the algorithm offers on the average MLU (we can see it by studying the dotted line in the graphics). We can see that the gap between the NO CRR version (ONOS shortest path, blue line) and CRR version (red line) is about 10-12% with peaks of 15%. [e.g. Figure 5.10, Figure 5.25].

- Fast computing times: Computation of the CRR, then of the various RR configurations and activation times, took place within a short time with these parameters [5.1]. In some tests done, and not published, we noticed that CRR computing times tended to grow exponentially as active nodes increased and as the number of consecutive matrices and the number of blocks varies (K, Min_Lenght). This also caused CRR failure cases due to the excessive computation time of RR configurations which exceeded a set threshold (in our case, the threshold is given by a Training_Interval for all versions). These long times are caused by the calculation of unsplittable routing.

The clustering computing part was made on a virtual machine with 2 Core (Intel i5 7200U) and 4 GByte of Ram.

- The last consideration we can make from these results is found in the latest version, the 3rd, the most complete and most realistic one to apply. We note that the algorithm is also correctly applied recursively over several days (in this case 7, but it can be extended for more than a week). The algorithm is able to continuously optimize the average daily link utilization, even if an optimal RR configuration is already applied.

  Also, in V3, we can notice an actual gain between the shortest path and CRR case up to 15% which could improve day by day on traffic variation according to TMs. This allows us to show how our algorithm can reduce network congestion even with "high" traffic variations between two periods of Training_Interval.

- The values obtained in this configuration do not show excessive MLU burst because of the limits we had to put in the TM values for the previously mentioned bugs on IPERF3. Moreover, we expect low congestion because the capacity is over provisioned in order to guarantee good network conditions for several years.

Chapter 6

Conclusions & Future Works

The basic concept behind SDN, the division of the control plan from the forwarding, has created many innovative solutions, including the opportunity to integrate Traffic Engineering modules that can study the network and optimize network configuration based on traffic. Centralized management by an operating system in the controller, allows a better view of the network and therefore the development of new solutions to improve the Quality of the Service. In fact, the flexibility of SDN allowed Traffic Engineering to benefit it, especially for the ability to collect statistics on the devices, and allows the application of new dynamic routing reconfiguration and no more static reconfiguration than in old approaches except for link failure cases.

In this work, we wanted to show how robust routing approach can be made adaptive in an SDN context. However, to show its functionality and its benefits, we needed a valid SDN network and a system able to interact with an off-platform application, such as ONOS. ONOS, NOS developed by ONF, allowed us, with the help of the SDN-IP application, to have a ready-to-use and editable network.

One of the best benefits offered by ONOS are the Intents, a way to create high-level device connections, without the user or the programmer knowing how to write OpenFlow rules. One of the disadvantages that ONOS has during this work on the SDN-IP application, is not to have an efficient statistics management system and not to be able to react by itself with new routing to network performance drops due to congestion, but only to link failures. The main objective of the thesis was to develop

and extend a system for collecting statistics in the network, based on the creation and collection of traffic matrices exploiting the information in the Intents. For this reason, we modified the Intent, to be based on the SRC/DST IP Prefix routing of the external subnets and not just for DST IP Prefix. In addition, the creation of a communication way between the off-platform application and SDN-IP via REST, has allowed us to force new routing within the network, different from those applied by ONOS (based on shortest path rules), based on the TM study collected by the module.

We have presented 3 case studies of the CRR application that have shown how the creation of an efficient communication system between ONOS, which install the new Intent version, and the external module, which computes RR configurations, allow to reduce the congestion of the network, so also the Maximum Link Utilization, up to 15% in a short time and continuously over time, as shown in the last case study (V3). The performance of the algorithm can be even better with respect to a more challenging scenario, since we used public available measurements of a real network with no big issues in terms of congestion and sensitive to some measurement errors as quoted by the author in the reference site. Also, due to the long computation times of the unsplittable version of the CRR, we had to test on a network with less active nodes compared to the full version.

6.1 Future Work

The work we have presented in this thesis has created a solid structure for future works and proposals. We will list some points we have studied as possible future work in the short term:

- **Splittable Routing**: A first future work that we considered as necessary is to implement a splittable routing in the ONOS application. The model is already configured for this possibility and would accelerate the computation of RRs within large networks with many more active nodes than those considered. In tests done with multiple active nodes, the computation of RR unsplittable tended to grow exponentially, causing CRR failure due to excessive computation time.

- **Dynamic Reconfiguration**: The traffic configurations that the module creates are based on the pseudo-periodicity traffic, so assuming that the RR cluster is valid throughout the next Training_Interval. If the traffic is substantially deviating from the expected scenario, a possible solution would be to looking at the current traffic situation on the network and might decide that there is a different routing solution, among the precomputed set, which is better suited for the current scenario. The controller might even try to adapt and refine the precomputed set of robust routing configurations according to real time measurements. Moreover, we could exploit the high memory availability of the controller to keep a history of robust routing configurations previously applied to have a larger choice of precomputed routings to be applied in case of unexpected scenarios.

- **GRPC**: At this time, our systems communicate and exchange information (JSON) through the REST API service. In the future, it is possible to create a common interface that implement both REST APIs or a new definition of gRPC. gRPC allows to exchange informations faster since it would retrieve statistics as soon as FlowRule is triggered. In the case of REST, the data is retrieved after a Training_Interval by the off-platform module (This implies that the application must cache the statistics).

- **Distributed System**: We could extend ONOS running on multiple instances (so it can run a SDN-IP application on every instance). In our version, the system runs on one instance. Every SDN-IP application listen to route updates coming from BGP, but only one SDN-IP instance is designated to be the responsible for pushing Intents to ONOS. This would allow to retrieve statistics from multiple instances since all instances have a complete view of the network and therefore also have access to device statistics.

- **Connection disruption during re-routing operation**: Re-submit of an Intent (that is created as LinkCollection or as a PointToPoint (Shortest Path)) causes the old intent to be withdrawn and the new one to be submitted, generating a small connection disruption. We need to implement some consistent

updates mechanism. There is a patch on the ONOS Wiki in the approval phase which would allow "Non-disruptive Intent reallocation" that can be applied when it will be available. [50].

Bibliography

[1] Bgp [rfc 4271]. http://www.rfc-editor.org/rfc/rfc4271.txt.

[2] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[3] Pingping Lin, Jonathan Hart, Umesh Krishnaswamy, Tetsuya Murakami, Masayoshi Kobayashi, Ali Al-Shabibi, Kuang-Ching Wang, and Jun Bi. Seamless interworking of sdn and ip. In *ACM SIGCOMM computer communication review*, volume 43, pages 475–476. ACM, 2013.

[4] Openflow - Open Networking Foundation. https://www.opennetworking.org.

[5] Open Networking Foundation. "Software-Defined Networking: The New Norm for Networks". https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf.

[6] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[7] OpenFlow Switch Specification. Version 1.0.0 (wire protocol 0x01). *Open Networking Foundation*, 2009. https://www.opennetworking.org/images/

stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf.

[8] Openflow Specification - Open Networking Foundation. https://www.opennetworking.org/projects/open-datapath/.

[9] Daniel Awduche, Angela Chiu, Anwar Elwalid, Indra Widjaja, and XiPeng Xiao. Overview and principles of internet traffic engineering. Technical report, 2002.

[10] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 44(2):87–98, 2014.

[11] Zhaogang Shu, Jiafu Wan, Jiaxiang Lin, Shiyong Wang, Di Li, Seungmin Rho, and Changcai Yang. Traffic engineering in software-defined networking: Measurement and management. *IEEE Access*, 4:3246–3256, 2016.

[12] Cisco IOS NetFlow. Introduction to cisco ios netflow-a technical overview. *White Paper, Last updated: February*, 2006.

[13] sflow. http://www.sflow.org/sFlowOverview.pdf.

[14] Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241. ACM, 1999.

[15] Shihabur Rahman Chowdhury, Md Faizul Bari, Reaz Ahmed, and Raouf Boutaba. Payless: A low cost network monitoring framework for software defined networks. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–9. IEEE, 2014.

[16] Amin Tootoonchian, Monia Ghobadi, and Yashar Ganjali. Opentm: traffic matrix estimator for openflow networks. In *International Conference on Passive and Active Network Measurement*, pages 201–210. Springer, 2010.

[17] Curtis Yu, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Harsha V Madhyastha. Flowsense: Monitoring network utilization with zero measurement cost. In *International Conference on Passive and Active Network Measurement*, pages 31–41. Springer, 2013.

[18] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *NSDI*, volume 13, pages 29–42, 2013.

[19] Niels LM Van Adrichem, Christian Doerr, and Fernando A Kuipers. Opennetmon: Network monitoring in openflow software-defined networks. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–8. IEEE, 2014.

[20] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010.

[21] Andrew R Curtis, Wonho Kim, and Praveen Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *INFOCOM, 2011 Proceedings IEEE*, pages 1629–1637. IEEE, 2011.

[22] Philip Wette and Holger Karl. Which flows are hiding behind my wildcard rule?: adding packet sampling to openflow. *ACM SIGCOMM Computer Communication Review*, 43(4):541–542, 2013.

[23] Richard Wang, Dana Butnariu, Jennifer Rexford, et al. Openflow-based server load balancing gone wild. *Hot-ICE*, 11:12–12, 2011.

[24] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: Scaling flow management for high-performance networks. *ACM SIGCOMM Computer Communication Review*, 41(4):254–265, 2011.

[25] Minlan Yu, Jennifer Rexford, Michael J Freedman, and Jia Wang. Scalable flow-based networking with difane. *ACM SIGCOMM Computer Communication Review*, 40(4):351–362, 2010.

[26] Stefano Salsano, Pier Luigi Ventre, Luca Prete, Giuseppe Siracusano, Matteo Gerola, and Elio Salvadori. Oshi-open source hybrid ip/sdn networking (and its emulation on mininet and on distributed sdn testbeds). In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 13–18. IEEE, 2014.

[27] Open vSwitch. `http://openvswitch.org/`.

[28] Sugam Agarwal, Murali Kodialam, and TV Lakshman. Traffic engineering in software defined networks. In *INFOCOM, 2013 Proceedings IEEE*, pages 2211–2219. IEEE, 2013.

[29] Yingya Guo, Zhiliang Wang, Xia Yin, Xingang Shi, and Jianping Wu. Traffic engineering in sdn/ospf hybrid network. In *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*, pages 563–568. IEEE, 2014.

[30] Yingya Guo, Zhiliang Wang, Xia Yin, Xingang Shi, Jianping Wu, and Han Zhang. Incremental deployment for traffic engineering in hybrid sdn network. In *Computing and Communications Conference (IPCCC), 2015 IEEE 34th International Performance*, pages 1–8. IEEE, 2015.

[31] Onos: Open Network Operating System. `http://onosproject.org`.

[32] Onos Wiki. `https://wiki.onosproject.org/`.

[33] Sdn-ip Architecture. `https://wiki.onosproject.org/display/ONOS/SDN-IP+Architecture`.

[34] Quagga Software Routing Suite. `https://wiki.onosproject.org/`.

[35] Intent Framework architecture. `https://wiki.onosproject.org/display/ONOS/Intent+Framework`.

[36] Mininet: An Instant Virtual Network on your Laptop. `http://mininet.org`.

[37] Nicola Rosada. Master Thesis: Clustered robust routing solutions in software defined networks, 2016.

[38] Internet2. https://www.internet2.edu/.

[39] S. Orlowski, M. Pióro, A. Tomaszewski, and R. Wessäly. SNDlib 1.0–Survivable Network Design Library. In *Proceedings of the 3rd International Network Optimization Conference (INOC 2007), Spa, Belgium*, April 2007. http://sndlib.zib.de, extended version accepted in Networks, 2009.

[40] S. Orlowski, M. Pióro, A. Tomaszewski, and R. Wessäly. SNDlib 1.0–Survivable Network Design Library. *Networks*, 55(3):276–286, 2010.

[41] Abilene traffic matrices. http://www.cs.utexas.edu/~yzhang/research/AbileneTM/.

[42] Abilene update session. http://www.internet2.edu/presentations/spring03/20030410-Abilene-Corbato.pdf.

[43] Internet2/incommon update. https://www.terena.org/activities/tf-emc2/meetings/8/slides/i2-inc-update.emc2.2007-03.pdf.

[44] Ubuntu LTS. https://www.ubuntu-it.org/.

[45] Python. https://www.python.org/.

[46] Onos code from GitHub. https://github.com/opennetworkinglab/onos.

[47] Apache Karaf. https://karaf.apache.org/.

[48] Apache Maven. https://maven.apache.org/.

[49] Inc. Gurobi Optimization. http://www.gurobi.com/documentation/7.5/refman/index.html.

[50] Non-disruptive intent reallocation github. https://github.com/opennetworkinglab/onos/commit/4f68ec98fd332870632ac81f97deae1079badfad.