



**POLITECNICO
DI TORINO**

POLITECNICO DI TORINO

Master Degree course in Communications and Computer Networks Engineering

Master Degree Thesis

The Evolution of Network Automation on Internet eXchange Point (IXP) by Software-Based Solutions

Supervisors

Prof. Paolo GIACHONE

Eng. Christian RACCA

Candidate

Hossein RASI

DECEMBER 2018

Acknowledgements

I would like to express my deep gratitude to Professor Paolo Giaccone, my research supervisor, for his patient, guidance, and useful evaluations of this master thesis.

I would like to express my very great appreciation to Christian Racca for his constructive professional advice during the planning and development of this thesis. I very much appreciate his enthusiasm to give his time so generously.

I am particularly grateful for the technical assistance given by Nicola Occelli and Andrea Beccaris from TOP-IX for providing me with facilities necessary to do my master thesis in TOP-IX.

I would like to offer my special thanks to Professor Marco Mellia who helped me to start this thesis and introduced me with TOP-IX consortium.

Many thanks to Politecnico di Torino for giving me the opportunity and also providing me the proper situation to do the master degree. I would like to thanks TOP-IX consortium that provides the requirements to create a testbed and networking equipment.

This journey would not have been possible without my loving parents and siblings, and their encouragement to follow my aspirations. I am thankful to them, for supporting me emotionally and financially. And to my lovely girlfriend, Nooshin, thank you for your warm support during this period.

Abstract

Network automation is a constructive solution that provides better consistency and efficiency in terms of IT-based operations which are repetitive and time-consuming. It also fulfills specific demands such as the interconnection with software tools or Web systems and supports network administrators in their daily jobs, that is getting more complex due to the diverseness of the devices and fast-evolving technology.

Internet eXchange Point (IXP) is a neutral point that connects multiple elements of today's Internet connectivity to one another. The thesis describes different applications of network automation in IXP and its advantages. A collaboration with TOP-IX (Torino Piedmont IXP, based in the north-west of Italy) provided a real testbed based on real networking devices and software tools.

Private Peering is one of the pioneer services which is offered by TOP-IX and has been automated by using a free open source automation tool named Netmiko (based on Python) and a web UI to obtain the required information of the switches. To date, the configuration steps were entirely manual which could take various configuration interval and might be influenced by the probable errors. While in the current work the automation process uses a multithreading approach that gives a much faster result and facilitates visualizing errors in the Web UI.

As an alternative approach, Private Peering has been implemented in a Software Defined Network (SDN) environment where the physical topology was created by Mininet and used ONOS as the SDN controller. An ONOS application which is called Virtual Private Lan Service (VPLS) has been used to VLAN provisioning into devices and supports the interconnection of multiple nodes in a single bridge Layer-2 broadcast domain over an OpenFlow network.

Finally, the last performing task was focused on automating the configuration procedure when a new client wants to join to an Internet eXchange Point through a Route server (BIRD is currently the open source routing daemon used in TOP-IX). The solution has been implemented using the framework Flask and JavaScript language to create a web UI and updates the TOP-IX BIRD configuration scripts to avoid IP address duplication.

Keywords: Network Automation, IXP, Private Peering, VLAN, Netmiko, BIRD, SDN, ONOS, Mininet, VPLS, Intent

List of Codes

3.1	Authentication information based on JSON file	28
3.2	CheckVtpServer Function	30
3.3	Python function to check interface condition	32
3.4	SubmitJson function written by ajax in JavaScript	35
3.5	Creation of OpenFlow router in Mininet	40

Contents

List of Figures	6
1 Introduction	9
1.1 Networking technologies and evolutions	9
1.1.1 Network automation	9
1.1.2 Machine learning and networking	10
1.1.3 SDN	11
1.2 Internet peering and IXP model and TOP-IX	12
1.2.1 Internet peering	12
1.2.2 Internet Exchange Point	13
1.2.3 TOP-IX	16
2 Problem architecture and modeling	19
2.1 Identifying and analyzing the needs in TOP-IX	19
2.1.1 Private Peering	19

2.1.2	Updating BIRD routing daemon	20
2.1.3	Private Peering by SDN	21
2.2	From problem setting to project definition	24
3	Implementation and analysis	25
3.1	Private Peering	25
3.1.1	BackEnd	25
3.1.2	FrontEnd	33
3.2	Private peering by SDN	38
3.2.1	Benchmark	44
3.3	BIRD automation	46
3.3.1	Backend	46
3.3.2	Frontend	47
4	Conclusions	53
4.1	The issues raised during the implementation	54
4.2	Importance of Network Automation for IXPs	55
4.3	Future directions	56
	Bibliography	57
	Appendices	59
.1	VTP server configuration step	61

.2	Custom topology creation script in Mininet	62
----	--	----

List of Figures

1.1	The typical workflow	11
1.2	Distribution of the IXPs (data from PCH)	14
1.3	TOP-IX infrastructure	18
2.1	VPLS components	22
2.2	Intent State Machine	23
3.1	Private Peering Topology	26
3.2	Initial style of the web form	34
3.3	Scheme of the Web template before starting the configuration	37
3.4	Private Peering Error list	37
3.5	Private Peering authentication failure schema in the CLI terminal	38
3.6	Private Peering authentication failure schema in the web interface	39
3.7	ONOS Web GUI	41
3.8	The instruction of installed intents from ONOS CLI	43
3.9	Default ONOS dump flows before activating VPLS	44
3.10	ONOS dump flows after activating VPLS	44

3.11 ONOS dump flows after sending a Unicast IP traffic (Ping)	45
3.12 Structure of TOP-IX client file	48
3.13 Scheme of the web UI for updating BIRD	49
3.14 Output in CLI for the BIRD configuration	51
3.15 Output in Web UI for the BIRD configuration	51
3.16 Updated version of client YAML file	52

Chapter 1

Introduction

1.1 Networking technologies and evolutions

1.1.1 Network automation

Network automation is the processes to automatize the management, configuration, monitoring, testing and deployment of networking devices. It not only saves the configuration time of the networking devices but also improves the capability of network maintenance by means of operations that are more understandable and implementable at large scales. Automation can be performed in both physical and virtual networking devices within a network and also on repetitive tasks.

The automation tools can be planned as a higher layer of abstraction that expands the functionality that can be offered by the networking devices. A high-level programming language can create a protected condition for developing programs. Simultaneously, it can also help the more experienced developers by implementing best practices that improve the reliability of the work. The idea of an autonomous network can rely on automation, programming and in many cases machine learning with an indicative plan.

Additionally, the network automation can retain the network in the state that appointed to follow with the organization's strategies. It can produce reports on the state of the network situation or modifications that might happen and dynamically remediates any unintended changes in the network. So as a result, IT automation can prevent spending most of the time to configure and monitor the devices by deploying the applications and so let the business to move ahead. For instance, based on the works done in [31] that have been deployed in Alibaba's global scale WAN using Netcraft [1] which shows 95%

reduction in the network incidents caused by configurations and 93% decline in average processing time for network updates.

Internet eXchange Point (IXP) is a term used to identify a network interconnected point that connects networks, content and application providers. They are built to minimize the ISP's network traffic transition that needs to go through the upstream providers. IXPs are located at strategic points throughout countries which let hundreds of Autonomous Systems (AS) to interconnect and agree on their traffic exchange. Latency reduction, enhancing routing efficiency and bandwidth, providing fault tolerance and fast data transition are the advantages of IXP. IXPs are one of the significant part of today's Internet connectivity also needs to use this automation. Firstly because, in IXPs or generally in service providers, automation is an essential strategy to focus more on network agility while controlling operational efficiency, limits, user satisfaction, and network security. So there is the possibility to automate routines, complex tasks that may be time-consuming, error-prone and repetitive in IXPs. There are plenty of open-source DevOps automation tools like Netmiko [2], Ansible [3], and Napalm [4] which are used for network automation purposes. The network automation features by means of software based products improve business integrity, and network agility as well as gives a better understanding about the network control.

1.1.2 Machine learning and networking

Machine learning has been developed from the struggles of a few computer engineers searching whether computers could learn how to play games, and a field of statistics that mostly overlooked computational problems. It is a set of algorithms that let the software applications to be more precise while predicting results without being programmed. Those algorithms get the input data and can predict the outcomes by statistically analyzing them.

Nowadays, Machine Learning techniques play a significant role in different areas such as speech recognition and computer vision. It is also a good solution for big data analytics. The idea of Big Data is described in [34] as high volume, numerous velocity and a high variety of data that need processing models to enable insight identification and enhanced decision making. In a machine learning method, the more data you provide to the system, the more it can learn from it and so it can return all of the evidence that you might need, and that is why it works so well with big data. Computer networks can also use the benefit of machine learning in different aspects. Building a sort of algorithms that can operate as a model to determine and implement decision making is the main feature of Machine learning that can be used in computer networking. In computer networks, the advantage of using machine learning can be felt as it can help to make the network scheduling much easier, aim the intrusion detection mechanisms and performance prediction by classification and prediction methods [24]. It can also create the analytic models to deal

with complex system behaviors such as throughput characteristics [24], load changing pattern in Content Delivery Networks (CDN) [32] and Internet traffic classification [33]. The typical workflow of machine learning for networking based on the study in [36] and can be demonstrated by Fig 1.1.

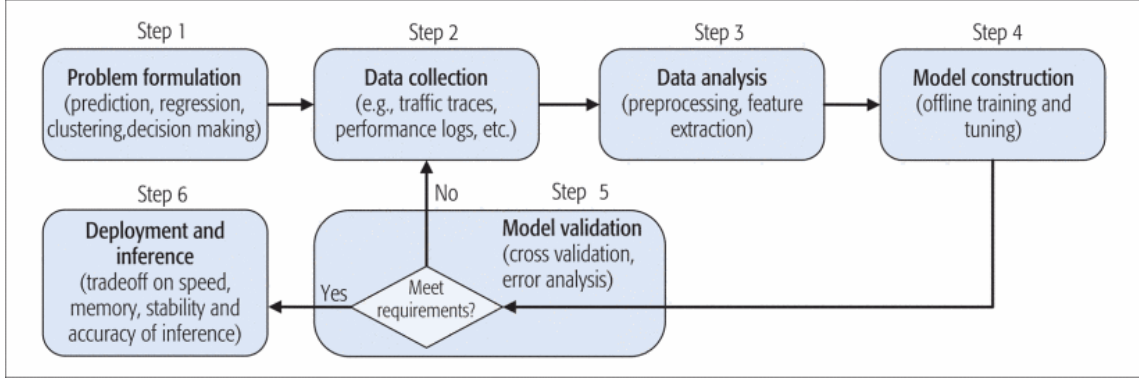


Figure 1.1. The typical workflow

Fig 1.1 shows the learning criterion in the networking field in different stages. These steps are not independent but have internal relationships.

Deep learning also, as a technique for implementing machine learning strategies, has investigated and implemented to provide solutions to improve the networking characteristics. Generally, machine learning algorithms used to learn from data and make proper future decisions and predictions. The output of the machine learning algorithms is usually a numerical value. However, Deep learning interprets data features and their relationships using "Neural Network" which can determine and make smart decisions by self-directed algorithms as a sub-field of Machine learning. The output of the deep learning can be a score, free text or an element. The latest work in [40] carries a broad review of preceding efforts that done by applying deep learning technology in network associated areas.

1.1.3 SDN

Internet traffic and users are growing incredibly, where almost everything is connected and there is accessibility from everywhere. Thereby, dealing with traditional IP networks becomes more complex and challenging to manipulate, from the configuration of devices to maintaining and monitoring or even expanding it. The current networking technology is also vertically integrated, meaning that the control plane (network controller) and data planes (traffic forwarders) are aggregated together so it reduces the network compliance [28] and that is one of the main reasons for creating SDN which splits this

vertical integration and separates the network's control logic from the forwarding devices (routers/switches).

In SDN, network devices act as a simple packet forwarder, while the brain or the control logic is performed in the controller (control plane). This feature of SDN proposes the advantages of a centralized system for network configuration, instead of the distributed method [30]. SDN also introduces the capability of programming network to adjust network flows. This behavior essentially focuses on protocols such as OpenFlow that helps controller to cooperate with networking devices (switches and routers). So it encourages the SDN network to be more dynamic to changing demands. This programmability also will help to automatize network.

Based on studies in [35], the history of the works that had been done over the years in the construction of programmable networking, divided into three stages. (1) active networks (1900-2000) which introduced the programmable functions in the network. (2) the separation of control and data plane (2001 - 2007) which allowed developing interfaces between control and data plane. (3) the OpenFlow API and network operating systems (2007-2010), described the first instances of the development of an open interface and the practical ways to separate control and data plane. Software-Defined Wide-Area Network (SDWAN) is also driven from Software Defined Network as an alternative technology. It is an application that applies to Wide Area Network (WAN) connections to connect enterprise networks. It might use a WAN connection between a central office and any other branch offices or between data centers beyond a large geographical area. SD-WAN technology separates traffic management from networking devices and applies them to individual applications to achieve enhanced performance, outstanding user experiences over geographically scattered positions, and simplify the deployment of wide-area networks.

1.2 Internet peering and IXP model and TOP-IX

1.2.1 Internet peering

The benefit of Internet services from the consumer point of view is directly related to the capability of connectivity between Internet Service Providers (ISP) and Autonomous Systems (AS) based on the agreement to exchange the routes and traffic between each other. There are two approaches to achieve this task. First one is transit, by which, the big backbone providers offer connectivity to ISPs to carry their traffic all over the Internet. The second strategy is peering agreement between two ISPs to deliver packets that dedicated for the users on peer's network using BGP (Border Gateway Protocol). Peering can be implemented by creating sessions of mutual accessibility between the autonomous systems connected to the Internet exchange infrastructure. There are two types

of Internet peering, private and public. Public peering is efficient if the two operators exchange a small amount of traffic and it can be implemented across a shared network that created by the IXPs. In private peering, as the name suggests, a direct and private connection is created between two providers for the purpose of interconnecting. However in Public Peering, by means of Internet exchange points, the providers freely decide there interconnection to other networks.

Furthermore, peering has several advantages, compared to transit since it consists of a payment by customers to a transit provider as an entity to carry the traffic through the global Internet. Firstly **Cost-effectiveness** compare to transit that needs payment to other company for traffic transportation. Secondly, the **Flexibility** that by connecting to an IXP, the consumers are no longer limited to one ISP so by broadening the network connectivity, it settles up with a more robust and adaptable distributed solution. Thirdly, the **Performance**, because peering increases efficiency and decreases redundancy and consequently, improves network performance. As a result, peering reduces the latency and increases the speed by letting the local traffic stays local, so the networks in the middle will wipe out which may act as a possible point of failure.

1.2.2 Internet Exchange Point

The idea of Internet eXchange Points (IXP) comes back to the creation of Network Access Points (NAPs) around 1994/95, which was an alternative plan to deliberately settle the decommissioning and transitioning of the National Science Foundation Network (NSFNET) backbone service to private industry [27]. IXPs are playing a vital role in Internet infrastructure where the data providers and ISPs meet and exchange the traffic between each other. It needs to state that researchers have understood that the Internet topology is changed to a flattering shape than before due to the presence of IXPs-transit paths which are neglecting the classic transit hierarchy [25]. As an approach, IXPs were useful and responsible for a smooth transition from a largely consistent network that started as an experiment to what characterized the modern Internet which comprised of the different set of elements that interconnect with each other to exchange information or services to end users or other customers [37]. The Fig 1.2 shows the worldwide distribution of the Internet Exchange Points based on PCH.

There are a set of database publishers that facilitate information exchanging related to peering. They also promote a better understanding of the current status of the Internet ecosystem. PeeringDB, European Internet Exchange Association (Euro-IX) are the most popular ones. PeeringDB, as a global database that intends to serve ISPs which wish to participate in the IXP peering ecosystem. Furthermore, PeeringDB provides full information about specific members and it is a default location for Internet peering database [5]. It is a database for peering and information associated to it. PeeringDB also allows the members to see information about networks that might want to peer with, where and

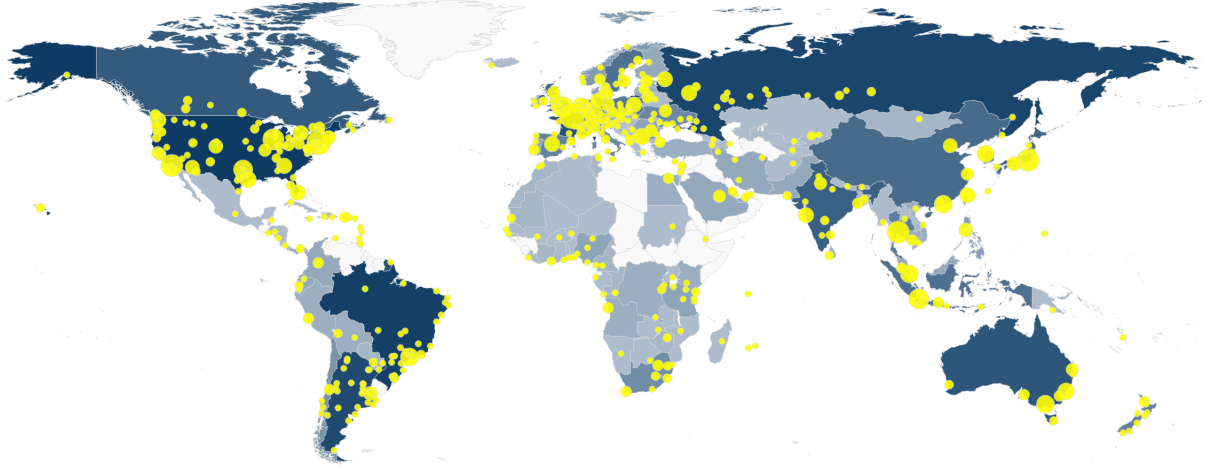


Figure 1.2. Distribution of the IXPs (data from PCH)

how can peer with them. The Euro-IX groups almost 80 IXPs to improve and extend the IXP community and aims the IXPs to recognize common ideas and it shares statics and useful information to all IXP members. Its membership comprises the European IXPs, which are operated as the common non-profit entities [6].

Moreover, some IXPs also considered new technologies such as Software Defined Exchanges (SDX) [26]. SDX makes the tasks, like time-of-day routing, dynamic traffic engineering for peering policy agreement and route decision based on external inputs, easier to plan and perform with excellent software control.

Based on works done in PCH [7], DE-CIX in Frankfurt is a world's leading interconnection platform that transmits almost 6 (Tb/s) as the peak traffic and 4 (Tb/s) as the average throughput. There are plenty of services that offered in DE-CIX such as SDX, Metro Vlan, Direct Clou, GlobePeer and Blackholing [8]. The second top IXP that located in the Netherlands is AMS-IX which established in the 1990s as a non-profit organization. The highest peak traffic is 5.664 (Tb/s) and is the first mobile peering points worldwide. The additional services that currently proposed by AMS-IX are Mobile Data Exchange (MDX), GPRS roaming exchange (GRX) and interconnection of IPX networks (Inter-IPX) [9].

Automation in IXP

As it has been discussed before (1.1.1), network automation is an aim that is stimulated by software defined networks and used programming languages for managing and monitoring

of networking components. However, the benefits of network automation are same also in IXPs as one of the main parts of today's Internet ecosystem. To specify the usage of automation features in IXPs, we can first discuss the main services and core roles of IXPs that have automatic behavior. To list them we can start with route collector which is crucial for managing routes and is a diagnostic and metric tool in IXPs. A route collector accepts most of the routes but advertises nothing. It is also an essential system in looking-glasses service which provides routing information in the backbone infrastructure level.

Border Gateway Protocol (BGP) as a unique and dynamic protocol, which is used by ASs (autonomous systems) around the world to communicate with each other. It is based on Exterior Gateway Protocol (EGP) which is working based on path vector routing protocol. BGP makes decisions based on the policies that are created by the network administrators and involvement agreement between ASs [23]. It is using standard configuration to establish the neighbor routes that are also called BGP peers. At first, BGP peering identified to minimize the management and operational overhead, so it connects a large global Internet Service Providers (ISP) to a great set of providers.

Nowadays, as an approach to BGP peering between ASs, BGP route servers became required for IXPs infrastructure from medium to large scale. They have all the features of the route collectors but also advertise routes to associating IXP members according to their routing policies. They are critical elements in the IXP ecosystem to offer an innovative peering solution to the members and form an open accessibility to a significant proportion of the Internet routes. Route servers do not route any traffic but only redistribute routes (originating from BGP protocol) between IXP members. So, an individual BGP session can make available all the announcements that created by all the ASs connected to the route server. The prefixes that are advertised by route servers in IXPs are covering 80-95 percent of today's traffic which also involved most of the popular destinations in today's Internet ecosystem [39]. The interaction between individual members of the route server and the others can occur through BGP communities. BGP community as a particular BGP property, allows a network operator to advertise a group of destinations in a single entity. They are basically labels that can be attached to BGP routes. There is also the possibility by the route reflectors to re-advertise iBGP-learned routes to other peers once the best path is selected or even block all paths for a particular prefix and not advertise it to other peers [29]. The most recognized route daemons that are currently used in IXPs are open-source networking applications amongst which BIRD, Quagga, and ExaBGP are more commonly used.

BIRD project relaunched in 2008 [10] (initial project published in 1999). It is an open-source daemon that was developed by CZ.NIC labs and nowadays most of the IXP communities are supporting it. It is competent to operate with multiple protocol instances and several routing tables. Furthermore, it uses simple, readable and structured-based configuration files. In case of modification or updating in the configuration file or in client list, it automatically reconfigures and applies necessary changes without any interruption

in other routing protocol sessions. It supports IP version 4 and version 6. It has a programmable route filter that can be implemented by scripting mechanisms which allow a highly effective expressive ability. It fundamentally uses the Linux operating system and kernel as a software router and data plane, so BIRD is operated as the control plane [38].

Furthermore, in today's Internet network, IXPs are acting as a fundamental element of traffic exchange, so it is of great importance to guarantee the reliable and effective operation in its network infrastructure. Currently, IXPs by software-based monitoring tools can monitor and analyze the traffic patterns. Monitoring and alerting software can analyze performance in real-time and by that, if any malfunction happens or any issue detected, it can be alerted immediately in many ways like email or SMS to the network administrators. There are other advantages of using monitoring tools such as optimizing network performance and availability, eliminating the requirements for manually controlling the availability of the resources.

1.2.3 TOP-IX

History

The idea of building an Internet eXchange Point in Piedmont region which takes place in the north-west of Italy came out in April 2002. It leads to the creation of the TOP-IX consortium which was established by a comprehensive innovation idea since the beginning. At first, TOP-IX consortium conducted just in Torino, but as the Internet consolidated itself as a powerful platform, then TOP-IX enlarged its connectivity to the Piedmont region. In the beginning, TOP-IX consists of 12 connected networks and now more than 100 networks connected to it. In September 2009, TOP-IX was the first Italian IXP that connected to Google as a hyper-scale network platform [11].

In addition to TOP-IX, there are other exchange points in Italy which are playing a significant role in today's Internet connectivity in Italy. For instance, Milan Internet exchange (MIX-IT) [12] is one of the biggest IXPs in Italy that handles a huge traffic exchange. It formed to improve the development of the Internet in Italy and to promote the interconnection between ISPs. Pooling@MIX is a service that offers in MIX to facilitate the network connection and to address to the groups of operators located outside of the Milan area. Internet exchange in Rome, Florence, Padova, and Palermo are also the other critical points in today's Italian Internet ecosystem.

Services

TOP-IX as one of the critical elements in today's Internet in Italy offers a set of interconnection services based on the distributed transmission infrastructure. The three main services that can implement in TOP-IX are Peering, Transit, and Marketplace. Peering is the function of the Internet exchange that allows creating bilateral or multilateral peering sessions. Transit, however, is an opportunity to establish a private point-to-point or multi-point link for backhauling requirements. The marketplace as a dedicated service in TOP-IX is an occasion to obtain services like IP transit, Cloud access, and Ethernet connectivity to/from different consortium members. The marketplace is available on Layer 2 transit, among demanding individuals and requires an agreement between the two or more than two members.

TOP-IX provides route server feature to facilitate the public peering service on its infrastructure. As discussed in 1.2.2, these systems can automate peering operations between members who use this service. There are two route servers activated in TOP-IX that defined to work as Autonomous System number 25309. The details about the BGP communities that can advertise to other members are available on RIPE website [13]. In TOP-IX however, the use of route server stands at the preference of the members of the consortium to freely choose to use this service.

An overview on platform

The Fig 1.3, shows the layer 2 infrastructure of the TOP-IX consortium. The infrastructure covers the centers in the Piedmont and Valle d'Aosta regions, as well as some interconnection nodes in Milan. As it can see in Fig 1.3, one of the main characteristics of the TOP-IX is to have a distributed platform. This architecture will help to guarantee a high level of reliability and accessibility. The accessibility of specific features of the TOP-IX consortium divided into four different nodes with various level of reliability and administration. Core, Backbone, EDGE and Remote access are the access nodes with different level of importance.

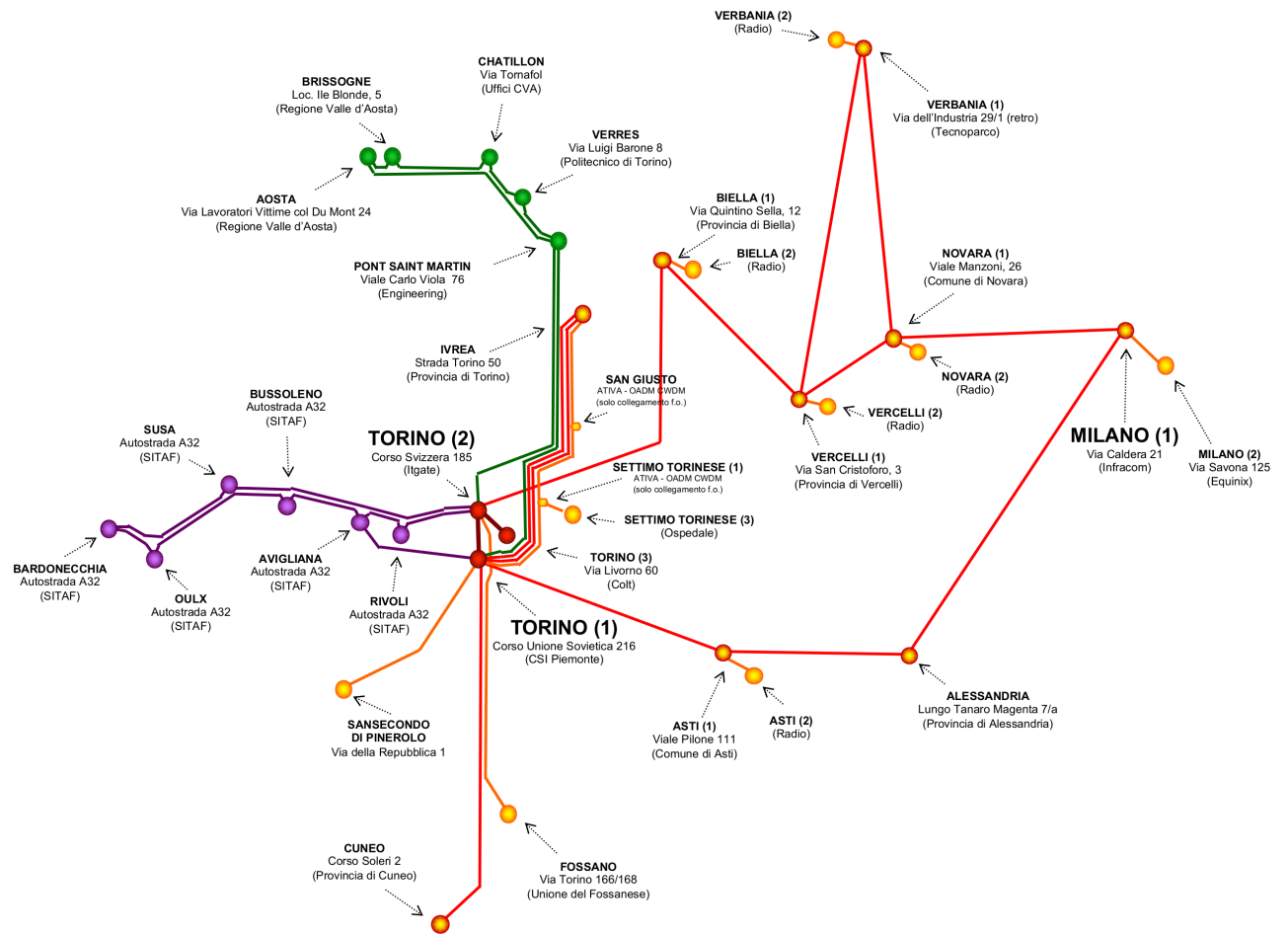


Figure 1.3. TOP-IX infrastructure

Chapter 2

Problem architecture and modeling

2.1 Identifying and analyzing the needs in TOP-IX

2.1.1 Private Peering

As it is mentioned earlier in [1.2.1](#), peering is defined when the two separated networks are interconnected and have the chance to exchange traffic between each other. However, it needs to know that, network partners do not need the charge each other for the creation of private peering.

Private peering, as one of the features of the peering strategies, can be implemented by creating a direct connection within two networks. It is mainly adopted when the two subnets require to send a big range of traffic between each other so, need to reduce the cost per megabits of traffic. Generally, peering is a cost-effective solution that lets to transmit the bandwidth at a lower price.

Additionally, a large number of IXPs throughout the world are offering the private peering as one of the main services to all of their new (not connected to IXP network yet) and the clients who want to produce a new private session between each other in the different part of the regions. If customers need a private connection that transmits their data reliably and much safer, private peering can be a good solution. It is also beneficial when the network administrators want to monitor the data transition, likewise, want to guaranty the network capability. Regularly, to create this service in IXPs requires a connection to the layer 2 IXP fabric. Network fabric is an enterprise term that

illustrates a network topology in which network elements pass data to each other through interconnecting devices. Each IXP member needs to connect to IXP peering fabric via a router which has the IP address that submitted by IXP.

However, the main idea to create an automated mechanism for setting up the private peering in IXPs is the time that needs to configure the devices as well as the time that required for maintaining the switches and routers during the month. It is hard to estimate the precise amount of time needed to create a private peering session but if considering the average number of switches to build a private session to 10, and for each need at least 5 minutes to connect and perform the essential configurations, so for performing one private peering, it would approximately require 50 minutes. Also, it needs to take into account that generally in big IXPs with a large number of peering requests, as the number of requests increases, the time that needs to set up the sessions will be significant. Although, in TOP-IX, the average number of private peering requests are 2 or 3 peers per month.

There are several network automation tools like open source libraries or paid support version software. Open source mechanisms provide much more elasticity and flexibility into various approaches. However, they need network automation knowledge and also programming skills, especially Python. Python is a programming language that is competent to write and work with a lot of tools. It developed in the early 1990s as the general confidence scripting language to increase readability and performance. It closely resembles the English language that let the users use the words like 'not' and 'in' inside the scripts and by that makes the code more readable.

One of the modules for network automation which work as a Python library is Netmiko. It simplified Secure Socket Shell (SSH) connection and network management of the devices based on the Paramiko SSH library. SSH is a network protocol that provides a secure environment and connection to a remote device. Netmiko successfully establishes an SSH connection to connect to the devices. Netmiko is a multi-vendor Python library that simplifies connections toward a broad range of network vendors and platforms. It also interpreted the execution of configuration commands for various vendors. In TOP-IX also, Netmiko can be a good choice because the majority of devices are based on Cisco solutions. Netmiko is supporting a set of Cisco platforms such as Cisco IOS, IOS-XE, ASA, NX-OS, IOS-XR, and WLC. It is worth mentioning that it is also compatible with other vendors like Arista, HP, Juniper, and Huawei [2].

2.1.2 Updating BIRD routing daemon

Internet eXchange Points (IXPs) provide IP data interconnection capabilities for their associates, by using distributed Layer 2 networking communication links like Ethernet or Fiber optics. The Border Gateway Protocol (BGP) [41], an inter-autonomous system

routing protocol, is adapted to aid the exchange of network reachability information across such media. However, the IXPs delivers new difficulties to interconnecting networks such as the expenses related to the interconnection system that induces serious operational and organizational scaling problems for IXP partners.

Route server or routing daemon is a process that settles routing data from border routers and spreads this information to dependent routers. Route servers can facilitate and automate peering management operations between consortium members that are wishing to use the services. They dynamically manage the routing table by interacting with daemons on other autonomous systems to exchange routing information. BGP sessions needs to be maintained to each of the peer's routers.

As discussed before (1.2.2), BIRD is Linux based Internet routing daemon that is responsible for controlling and managing forwarding tables. As an optional solution, TOP-IX is offering BIRD to perform public peering on its infrastructure to its members. By that, members that are using this service can automatically peer with each other.

The second task that requires to be automatized in TOP-IX is to create a method to gather new information about the new peers that want to be attached to the route server. Basically, in TOP-IX, there is a bash configuration script that runs every morning (1 am). This script builds the main BIRD configuration file for both IPv4 and IPv6 by a Python program which runs at the very first lines of the script. That program compiles and builds the configuration file of the BIRD by using the information about peers and all available announcements that made by connected autonomous systems (AS). There are also two YAML files based on the IP versions (IPv4 and IPv6) which consist of the critical information about the peers and clients that are using the route server. The task is to create a web template to arrange the information about new peer that wants to be added to the route server. As well as, running the main script and its subscripts after verification of the IP address for the new client. This verification procedure will aim to limit the mistakes that may happen by the network administrators.

2.1.3 Private Peering by SDN

The idea of "programmable network" has been introduced as a solution to aid network progression. As a new networking criterion, Software Defined Networking (SDN) is working in a way that forwarding device is decoupled from the control decision. Additionally, in SDN the network intelligence is centralized in software-based controllers. ONOS (open network operating system) is the controller that had been used for this scenario [14]. It provides an open platform that interprets the creation of network applications and services that can work over a broad range of hardware. It also gives high-level APIs for managing, monitoring, and programming networking devices. Moreover, ONOS provides a single-page web application (ONOS GUI) that proposes a visual interface to the ONOS

controller.

ONOS is a distributed system designed to work in a symmetric style where all the clusters are software-wise identical. Multiple machines can work collectively as a centralized, consistent distributed system, configured as a cluster. In this work, we used a single instance of the ONOS controller installed in a single machine. Moreover, ONOS includes an application management system for maintaining networking applications in a distributed manner. There are a set of default applications already installed in the controller which could be utilized once activated from ONOS CLI or web GUI.

The ONOS application that could satisfy the requirements and is used in this scenario is called Virtual Private LAN Service (VPLS) that provides multi-point broadcast layer-2 paths among devices in an OpenFlow network [15]. VPLS permits operators to build overlay networks on top of the OpenFlow infrastructure. As it discussed in [15], to verify connectivity between the hosts, after activating the VPLS application, there need at least two interfaces to be configured in the ONOS configuration interface property. Those interfaces must be associated with the equivalent VPLS list, so the individual hosts which linked in the same list can send/receive the packets labeled with the same or even different VLAN IDs.

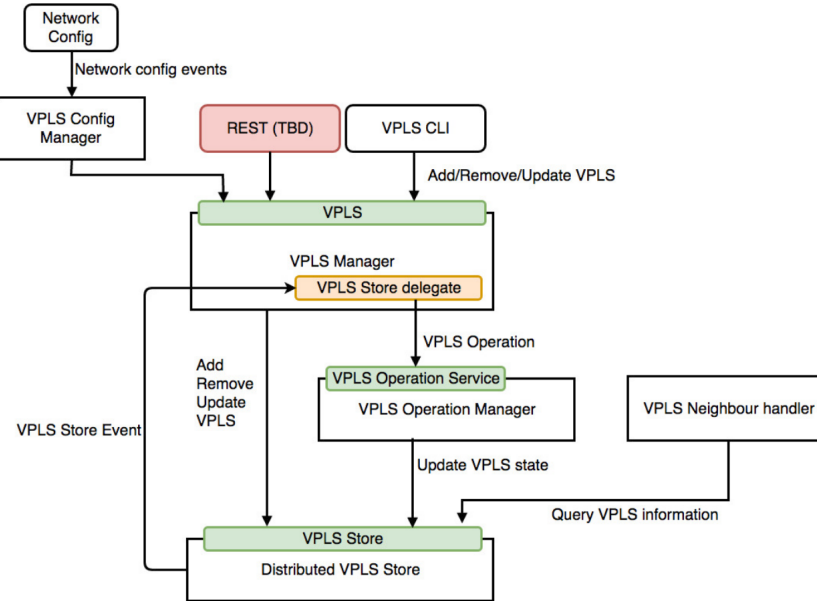


Figure 2.1. VPLS components

The Fig 2.1 shows the components that consisted in VPLS application. What is important to mention here is that VPLS proceeds by transforming the VPLS progress to the sets of intent operations in VPLS store delegate component. Generally, VPLS uses

the intent framework to provision unicast and broadcast connectivity between the edge node's ports of the OpenFlow network. There are two main intent forms per object in the VPLS list. First is for Broadcast traffic which is called single-point to the multi-point intent and the other is for unicast traffic, called multi-point to single-point intent. The features and the benefits of each intent will be discussed in 2.1.3.

The intent structure is a subsystem in ONOS that permits applications to define policies rather than mechanisms for the network control. ONOS Intent framework is one of the key Northbound abstractions. Northbound API permits the information to be exchanged between the SDN controller and the services and applications running over the OpenFlow network. The Fig 2.2 shows the intent state transition diagram for the compilation structure of the intents. The orange states are transitional which need to last only for a short period of time. The rest are parking states, where the states require to wait and so will take some times to go out of this states. When an intent submitted by an ONOS application, it will be sent asynchronously into the compiling phase and after that to the installing, and finally, to the installed state. Alternatively, an application may even withdraw an intent if it no longer wishes to hold it. The intents are compiled down into a set of "FlowRule" objects. It is good to know that the compilation process may include: (1) "IntentCompiler" which is the compilation of an Intent down into installable intent(s) or by a (2) "IntentInstaller" which is the reformation of installable Intents into FlowRuleBatchOperations including FlowRules.

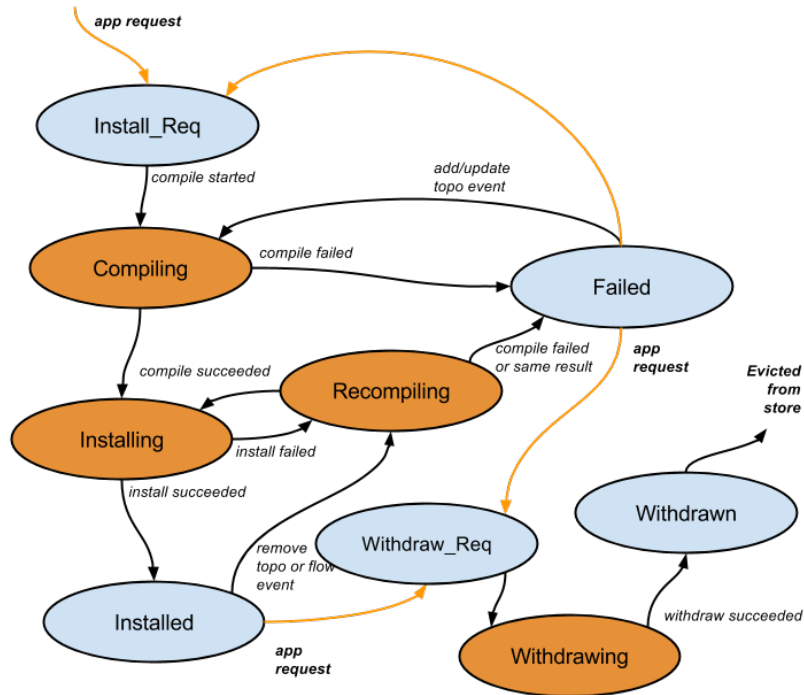


Figure 2.2. Intent State Machine

2.2 From problem setting to project definition

The main tasks that must be solved in TOP-IX are described in 2.1. However, it took some time to establish what is required at the first step to set up the project testbed. To properly recognize the projects, it needs to define the purposes of each task. As the outline recommends, having a web User template could be a great choice, firstly for the network administrators to not deal with complex scripts. Secondly, it is a reliable way to connect the errors that may occur while the backend script is running. The principal purpose of the creation of a web user interface (web UI) is to prepare the required information by the web forms and analyze and regenerate them in the way that can be understandable for the backend program. As soon as the web form filled correctly, the script will start its routines to automatize private peering creation or automatically configure the associated BIRD routing daemon files and runs the related privileges to the BIRD.

After outlining the objects of each task, a test environment that consists of all the components and features that might encounter is created. The first task needs two routers that take place in each peer and configured based on the particular VLAN. In the real scenario, TOP-IX submits two public IP addresses in the same subnet for both of the routers. However, as two routers will usually place out of TOP-IX network, the configuration of routers is beyond the scope of this thesis. In addition to routers, the topology also needs a set of switches to work as the layer 2 TOP-IX network fabric.

For the second task (Bird daemon automation), we set up an HP Portland DL360 G5 server with Ubuntu and Bird installed into it. To cancel the conflict probability with the main TOP-IX's Bird daemons, the test server does not have access to the Internet. We set the daemon to use the main TOP-IX's configuration files. However, because of simplification of the implementation of the main task, the BGP service configuration was neglected.

However, for the last implementation scenario to create the private peering by using the SDN, it was impossible to use the real networking devices. Cisco 3850 was the only switch that supports the OpenFlow and was available on that period for this purpose in TOP-IX. However, to be able to use this feature, it needs to install the Cisco plug-in for OpenFlow agent logical switch. But due to different technical and business reasons, installing the abovementioned plug-in was impossible. As a result, the physical topology formed by using Mininet [16] which lets to create a realistic virtual network and running OpenFlow switches and the SDN controller on a single machine. The SDN controller that used for this purpose is called ONOS [14] that produces the open source network operating system to build real software defined networks.

Chapter 3

Implementation and analysis

The main idea of this chapter is to focus on implementation part of this master thesis. This chapter is divided into three sections and inside each section the solutions, tools and approaches that had been used for each task is explained in detail.

3.1 Private Peering

This section is divided into two parts. The first part is going to focus mainly on the logic of the system and to solve the problem by Netmiko. In the second part, however, the key target is to create a web interface to collect the required information to implement Private Peering, as well as showing the errors that might face while the scripts are running.

3.1.1 BackEnd

As it has been discussed in the previous chapter ([2.1](#)), to implement Private Peering, it needs to construct a test environment that should be as close as possible to the real scenario. To build up this environment, the expectation is to build a solution using scripting techniques to configure the layer 2 Cisco switches. However, as it is mentioned before in [2.1.1](#), in the real private peering scenario, there need also two routers that should locate in the two sides of the connection and are responsible to transmit the traffic between the two networks. In this scenario also we specify two Cisco routers. [Fig 3.1](#) shows the network architecture designed to implement private peering in TOP-IX.

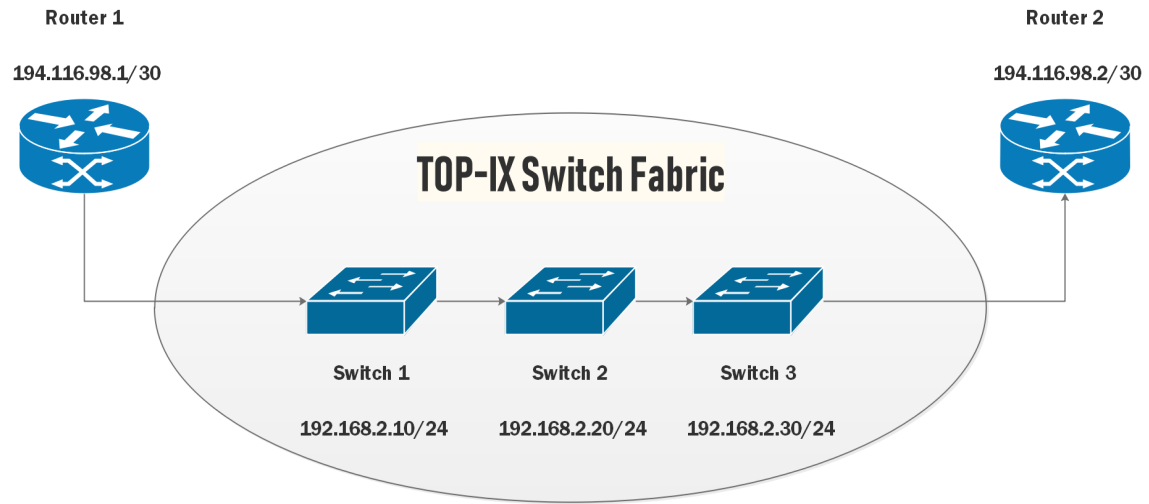


Figure 3.1. Private Peering Topology

As it can be seen in the Fig 3.1, there are three switches that are going to work as the TOP-IX layer 2 fabric in the real scenario. The switches are all Cisco platform (Cisco Catalyst 3750). The connection between switches at the physical layer was made by both twisted pair (cat 5e) and fiber optic (SFP) cables.

The first step is to check whether the Cisco IOS version of the switches supports the SSH functionality or not. We would need SSH connection because as it has been disused in 2.1.1, Netmiko uses SSH session to connect to networking devices.

It needs to consider that some old IOS versions do not support the SSH connectivity. So there might need to upgrade them into newer versions. After updating the Cisco IOS's, the next step would be to set up SSH client on CISCO IOS. The steps to configure SSH client for Cisco switches [17] are:

Step 1: Configure the host name command

```
hostname switch1
```

Step 2: Setting the DNS domain name

```
ip domain-name topix.com
```

Step 3: Generating SSH key

```
crypto key generate
```

```
ip ssh time-out 60
ip ssh authentication-retries 2
```

Step 4: Enable SSH transport support for the virtual type terminal

```
line vty 0 4
transport input ssh
```

Cisco CLI (command-line interface) has different command modes with different accessibility that a user can work. The initial mode that a user can log in is called *user EXEC* mode. This mode contains a limited set of commands with low accessibility. To access to the full list of commands, user must enter to another mode which is called *Privileged EXEC* mode. To enter into this mode user needs to use "*enable*" Exec command into the CLI. In privileged mode however, user does not have the access to configure commands, so to use those commands there is another mode which called *Global Configuration* mode and to enter into this mode, "*configure terminal*" command can be used. As the global configuration mode has almost full permission to configure a device and its interfaces, it is highly recommended to specify a secret password, if an user wants to enter in this mode. This will lead to add an additional level of security to Cisco devices and limit accessibility to global mode.

Netmiko library consists of a set of modules to work with. "**ConnectHandler**" is one of those modules that is responsible to create the SSH session. This factory function, choose the accurate Netmiko class based on the device type. Basically the information that are required to perform a SSH connection by ConnectHandler are: (1) IP address or DNS host name of each switch. (2) Type of device that is going to connect to it (here all of the switches are cisco-ios). (3) Username of the devices to log on with. (4) Password of each device to enter into enable mode. (5) Secret password of the global mode.

There are also other important modules in Netmiko that must be used after creating the SSH session. **Enable()**, to enter the device into the enable mode and **disconnect()** to close the SSH session in a secure way. The other module that is used very often is **send_command()**. It is responsible to execute a string of commands on the SSH channel by a pattern-based mechanism. Typically, it is used for show commands and will keep waiting to get the data until the network device prompt is detected. And the last Netmiko module that needs to be mentioned here is **send_config_set()**. The main functionality of this module is to send the configuration commands down to the open SSH channel. By means of this module, multiple commands can be sent to the networking devices. The commands can be separated simply by using comma between each other. The configuration commands will be executed one after the other. After finishing the execution of commands it will automatically exit and enter to the configuration mode. [18] shows all the possible classes and modules that can be used in Netmiko.

The most efficient way to provide the required information items to ConnectHandler

module is to create a ¹JavaScript Object Notation (JSON) database. JSON is used to create lists and also dictionaries in JavaScript. It also allows to import and export objects to text files. Reading and parsing data from JSON file is much easier for both human and also for the machine. Also, in this scenario, a JSON file is created that contains the information to create the SSH connection. Code 3.1 shows the structure of JSON file.

```
1  [  
2    {  
3      "ip": "192.168.2.10",  
4      "username": "topix",  
5      "password": "topix",  
6      "secret": "topix",  
7      "device_type": "cisco_ios",  
8      "host": "FS-TO-NEW"  
9    },  
10  
11   {  
12     "ip": "192.168.2.20",  
13     "username": "topix",  
14     "password": "topix",  
15     "secret": "topix",  
16     "device_type": "cisco_ios",  
17     "host": "Switch2"  
18   },  
19  
20   {  
21     "ip": "192.168.2.30",  
22     "username": "topix",  
23     "password": "topix",  
24     "secret": "topix",  
25     "device_type": "cisco_ios",  
26     "host": "Switch3"  
27   }  
28 ]
```

Code 3.1. Authentication information based on JSON file

After creating the JSON file, it is required to import the JSON library into the Python script and then to load the file into a Python list. However, this list is not readable for the ConnectHandler module yet. To connect separately each device, a global dictionary template has been created. This dictionary consists of the individual keys for

¹ A language-independent text format that applies conventions that are familiar to programmers and is easy for humans to read and write

ConnectHandler. The values of each key inside the dictionary are get from the Python list by a for loop.

Private Peering configuration requires the two network interfaces of each switch that are connected to other switches or to routers. This information is gathered from the network administrators and by means of a web interface which has been created by Javascript and ¹Flask module in Python. Web user interface (UI) creation will be discussed in 3.1.2.

Before starting to talk about the main configuration steps, it is of a great importance to know about Cisco Vlaning mechanism. A VLAN (virtual LAN) is introduced when a group of devices that located in the same or different LANs needs to be configured to communicate with each other. It enables a group of devices from different networks to work as a single logical network. However, it has the concept of logical grouping in the same broadcast domain. The main benefit of using VLANs in a network is that the number of broadcast domains can be incremented while its size remains fixed or even decreases. Since the traffic from different VLANs may need to travel over the same physical network, it is required to map the data to a particular network. This process is done by using a VLAN protocol, such as IEEE 801.1Q, 3Com's VLT, or Cisco ISL.

IEEE 802.1Q is an industry standard trunking encapsulation mechanism that is used to keep customer VLAN integrity across a service provider network [19]. Most of the modern VLANs use this protocol which adds an additional header or "tag" into each Ethernet frame. Data is sent between switches by a physical link called "trunk" that connects the network switches together. The trunking method is the switchport mode that must be configured on all switch interface modules. A trunk is a communication link planned to carry various signals simultaneously to provide a network connection between two points. The other VLAN trunking standard available in Cisco industry is ISL. ISL fully encapsulates the original Ethernet frame by appending a 26-byte header and a 4-byte FCS trailer. It was created to work with Ethernet, ATM, Token Ring, and FDDI. In this work we are using the IEEE 802.1Q, as the main trunking method to configure networking devices.

VLAN Trunk Protocol (VTP) is a solution in Cisco networking devices that can be introduced to reduce the administration time and difficulties in a switched network. By adding a new VLAN Id into the VTP server, it will be distributed to the other switches in the domain in a few seconds. This will significantly reduce the amount of time that is required to add the individual VLAN in the relevant switches. In this scenario, as all of the switches are in the Cisco platform so it is possible to use VTP server approach. However, there are some steps to configure a switch to work as a VTP server. First, setting a VTP domain name by using (*set VTP domain name*) and second, choosing its mode that must set to a server mode by (*set VTP mode [server | client |*

¹A microframework in Python based on Werkzeug, Jinja 2

transparent)). In this work, switch one with IP address (192.168.2.10) has been set to work as the VTP server and so other switches will work as the VTP clients.

In my solution, the VLAN ID and VLAN name that the new Private Peering session is going to work with, as well as the IP address of the VTP server, will be provided by the web user interface that is going to discuss in 3.1.2. Appendix .1 shows the script that has been written to add the new VLAN into VTP server by using Netmiko.

As it is illustrated by the script, after creating the SSH connection in the VTP server, the configuration commands are going to send to the devices by using `"send_config_set"` module. In many cases, it might happen that the chosen VTP server has been selected wrongly or it is not in the server mode. For this reason, a function has been created that can detect if the switch is currently in a VTP server mode or not. This function uses the output result of `"show vtp status"` which is a Cisco standard show command and then will parse into this output and can understand the situation of the selected switch. This function which is called `CheckVtpServer` also is useful if an error happens during VTP configuration and if so, it can show the error definition on the web platform and so will let the administrators to realize the reason of the error in a very clear way. The `CheckVtpServer` function is shown in Code 3.2.

First of all, it will create a Python list by using the `"splitlines()"` method with the output string lines from the `"show vtp status"` command. Then in a loop, it will search for the specific line which consists of the `"operation"` keyword which shows the mode of the VTP in the individual switch and then will append that line into a new Python list which is called `interfaces`. Then by setting a for loop into the new list and splitting each word by Python `"split()"` method, the function will search for the keywords `"Server"` and `"Client"` into the list and will get the current situation of the switch. It is good to know that each switch can work in just one of the abovementioned (Server or Client) modes. So, the main function will use this function to realize whether the specific switch is really in the server mode or not.

```
1 def CheckVtpServer(z):
2     lines = z.splitlines()
3     interfaces = []
4     for line in lines:
5         if "Operation" in line:
6             interfaces.append(line)
7         for line in interfaces:
8             words = line.split()
9             if "Server" in words:
10                 return "Server"
11             else:
12                 return "Client"
```

Code 3.2. CheckVtpServer Function

To improve the configuration time a multithreading approach based on Python programming language is used. Primarily, before using multithreading technique, the script took about four minutes to configure all of the devices. Regularly, the individual switch requires about one minute for configuration and other switches must wait until the configuration procedure of the current device finishes. As Kirk Byers, the producer of the Netmiko specified, the principal policy of the Netmiko is reliability and security. The two factors in Netmiko that inject delays during the configuration procedure, (1) `delay_factor` and (2) `global_delay_factor`. `Delay_factor` is used as a multiplication factor for timing delays. For instance, setting the `delay_factor` to two and using a "`send_command()`", the amount of time that requires to wait for the response will be doubled. Second is `global_delay_factor` that mainly is used to increase the delay for the whole configuration process and not just the `send_comand()`. Occasionally, `global_delay_factor` used when Netmiko wants to configure too slow devices. So by that, it can have adequate time for the device responses. So as a result, multithreading solution improves the configuration period from four minutes to less than two minutes, and this is because of the fact that the configuration commands are going to send to the devices in parallel.

The thread's arguments that need to be sent to the start routine function are the authentication information of each device, the two interface or port Ids associated to each switch, and the VLAN Id. For each device, it needs to create one thread which is responsible for configuring the device or run any function that is inside the body of the start routine function. The other related object that can cause issues in Cisco devices and needs to find a solution is the current state of the switch interfaces that are going to work in individual Private Peering configuration procedure. At first, It requires to understand the configuration commands that need to use when attempting to configure the interfaces based on proper configuration. The commands that applied respectively for configuring switch interfaces and provision the VLAN configuration are:

- 1: Specifying the interface to configure and enters the interface configuration mode

```
switch(config)# interface id
```

- 2: Configuring trunk encapsulation as dot1q

```
switch(config-if)# switchport trunk encapsulation dot1q
```

- 3: Enabling trunking on the interface

```
switch(config-if)# switchport mode trunk
```

- 4: Setting the proper VLAN to work in trunking mode in this interface

```
switch(config-if)# switchport trunk allowed vlan VlanID
```

or

```
switch(config-if)# switchport trunk allowed vlan add VlanID
```

- 5: Setting spanning tree mode

```
switch(config-if)# spanning-tree mode ra
```

6: Enabling the spanning tree for VLAN id

```
switch(config-if)# spanning-tree vlan VlanId
```

As it can be seen in above commands, in step 4, the configuration procedure needs initially to know whether any VLAN currently is set to the individual interface or not. The first option is used when the interface does not consist of any VLAN configuration setup or it is in the Cisco interface default configuration mode. But, the second command is used when there are VLAN/VLANs, already configured to work in a specific switch port and, it is also configured based on trunking mode. It needs to mention that a Cisco switch interface can't work in both trunk and access VLAN mode at the same time. So to solve this issue, based on the abovementioned consideration, it needs to find a way to configure interfaces dynamically. As a result, a function has been created to solve this issue.

The function uses the "**show runnig-config interface *interfaceID***" command, which is a standard Cisco show command to display the current running configuration in a specific switch interface. Then, the function will parse into the output of the show command and so, based on the current situation of the switch port, the program will make a dynamic decision and will choose the best approach. The function structure is shown in Code 3.3.

```
1 def intCond(x):
2     word = []
3     inf = []
4     lines = x.splitlines()
5     word = [t.split() for t in lines]
6     for i in word:
7         if 'switchport' in i and 'allowed' in i:
8             inf.append(i)
9     return inf
```

Code 3.3. Python function to check interface condition

First of all, the function creates a list of lines in the string and break them at the boundaries and splits them at the line breaks. Then strings will be cut into a list where each word is a list object by itself. The function will search for the keywords "switchport" and "allowed" into the list of lines. If it can find the keywords which means that there is at least one VLAN, provisioned to operate in this port, so the main function uses the second method of previously discussed steps(step 4) to add the new VLAN. Otherwise, it will use the first solution of step 4, as the interface configuration does not consists of any VLAN, to configure the interface. After understanding which configuration command must be used, Netmiko will start its main procedure to send the proper Cisco commands toward switches.

The last thing that needs to be mentioned here is the list of Python-based exceptions, connected to the Netmiko, to purposefully manage the errors that might happen during the code execution process. Generally, in Python, errors caused an exception or a syntax error. Netmiko API consists of some predefined exceptions that can help to manage devices properly. The Netmiko exceptions, used in this program are, (1) SSH connection timeout and (2) SSH authentication failures. The first is used when the script takes too long to send the tasks to the device while the SSH session is active. The second exception mainly happens when the API tries to connect with wrong authentication information to the networking devices.

3.1.2 FrontEnd

As discussed in 2.2, the procedure for the Private Peering implementation is divided into two parts. By letting the complexity of the process to stay in the backend, frontend development controls everything that is associated to what users can visually see first, in a web interface. The web interface is created because, from client-side, it involves everything that a user can encounter during the compilation process. It is also an efficient way to observe the errors that may happen during the process in a more readable way for both network administrators and the people with a low level of technical knowledge.

The tool that is used in the purpose of the web form creation is Flask [20]. Flask is a web framework that consists of tools and libraries to help the users to build a web application. It is in the category of micro-framework, which are the frameworks that have no dependencies to external resources. The Flask dependencies are Werkzeug which is WSGI and Jinja2. Werkzeug is a library in Python which uses for communicating with the web server. Jinja2 is Flask's template engine for Python which inspired from Django system template but gives more useful tools to the user than Django. The Flask is used because it is much more lightweight platform than Django and also it can be implemented in a single Python file which made its administration easier than Django.

The web form implementation procedure divided into two sub-programs. First is the core web controller which is based on Python and second is html file based on JavaScript. The core controller is responsible for specifying the initial requirements of the Flask to run the web form. The script in backend section which is proposed to configure devices will compile or call in this program. Additionally, by using Werkzeug methods, it can communicate with web template. The way to request the representation of the specific resources is named "GET" and it only retrieves web template data from the URL. The other method which is used to submit an object to particular resources is called "POST" and it always generates a modification in states or side impacts of the server. It also leads to classifying the errors that may happen during the script compilation by making a Python dictionary which consists of all the possible errors that may occur.

As discussed in 3.1.1, the private peering task needs some information to start its routine. That information will gather by a web form that will be explained in this section. Fig 3.2 shows the initial scheme of the web template with the empty fields.

The screenshot shows a web form titled "New Private Peering Configuration Data". At the top left is the logo for "top torino piemonte internet exchange". The form contains several input fields: "Vlan ID", "Vlan Name", "VTP Server", "IP", "Port1" (with a placeholder "Gix/x/x"), and "Port2" (with a placeholder "Gix/x/x"). Below these fields are two green buttons: "Next Switch" and "Submit Data". At the bottom of the form is a table with the following headers: "Device", "Port1", "Port2", and "Delete".

Figure 3.2. Initial style of the web form

It uses the WTForms which is a module for form validation. The form includes the field descriptions, take input, delegate, validation, aggregate errors and in general, keeping everything together. The core Flask controller as the central operation section will gather all the required information and puts them into a JSON file by *request* object in Flask.

This object is used by default in Flask and it automatically pushes a request context when checking a request. The module that parses and returns the data as a JSON file is "**get_json()**" which by default this function will return none, if the mimetype is not application/json. `Get_json()` can accept some parameters as the input functions but

in this scenario, performed in the default model. As a solution to correctly import the information into the backend script, a JSON template which consists of names (vlan, vlan_name, vtp, IP, port1, port2), which their values will be full fill by the output of the `request.get.json()` method.

Additionally, in the HTML file that is created for the web template, it is mandatory to submit the input data into the JSON template. There is a JavaScript module for this specific purpose which is called AJAX which stands for Asynchronous JavaScript and XML with jQuery. JQuery is a very small JavaScript library which used to make working with JavaScript much simpler. It is a perfect tool for web applications and will let them be more powerful while transferring data in JSON between server and client. ¹JQuery AJAX is used to post the data form to the Python Flask. So the web application demands transmitting information to/from the server by synchronous request. It is possible by filling up the web form and hitting the submit button and at the end, directing to a new page to send further information toward the server or updating the current page. The function below shows the solution that is used for this purpose.

```
1 function submitJson()
2 {
3     $.ajax(
4     {
5         url: 'http://127.0.0.1:5000/result',
6         type: 'post',
7         contentType: 'application/json',
8         success: function (data)
9         {
10            if(data.error)
11            {
12                $.LoadingOverlay("hide");
13                alert(data.message);
14            }
15            else
16            {
17                $.LoadingOverlay('hide');
18                alert(data.message);
19            }
20        },
21        data: JSON.stringify(DataArray)
22    });
23 }
```

Code 3.4. SubmitJson function written by ajax in JavaScript

The function is using AJAX jQuery method that performs when the type of the request is in post method. The information data that is sent from the web page added to the web server which must be defined by the URL in a query string. The solution to convert

¹Perform an asynchronous HTTP (Ajax) request to load data from the server without a browser refresh.


the receiving data (JavaScript object or array) into a string is to use `JSON.stringify()` method. Basically, what `JSON.stringify()` do is to get JavaScript object and turns it into JSON text and then stores that JSON text in a string.

As it can be recognized from Fig 3.2, there are two buttons in the web template (New Switch and Submit data). They have both the JavaScript button type. New switch button should be used when we require to save the switch information and if we have more than one device to configure. When the New Switch button pushes, a new JSON template will be generated to get the new device data. And the Submit Data button is used to start the configuration routine firstly by submitting the information data into the JSON file and compiling the main configuration script which explained in the preceding section (3.1.1).

The Fig 3.3 demonstrates the scheme of the web template after saving all the information about the devices to start the Private Peering configuration procedure. There also specified a delete button that uses in the case that the switch information is proposed wrongly or consists of the false data. After finishing the data entry, by pushing the Submit Data button, the Private Peering routine will begin its operations.

As it has mentioned a lot of times, one of the reasons for web form creation was to show the errors that might probably face during the configuration procedure. As a solution, an error list created which has a Python dictionary type in the Flask controller script. The way that errors might happen are directly related to the information that sent from the created JSON file from the web template or while the individual information is correct but the specified device is not accessible. The error list established for the Private Peering configuration procedure with the unique error numbers shown in Fig 3.4.

Generally, the error checking procedure performed as the main configuration script start its compilation. If any error occurs, not only it is possible to see in the CLI terminal, but also it will show the error as an HTTP alarm in the web browser. The steps to show a specific error started when the Flask Python controller script calls the configuration program to start running. By default, the output zero means that the program finished successfully, so both the CLI and web template will demonstrate the message which refers to the successful configuration procedure. Any value except zero refers to the condition in which an error occurred during the configuration process. The Flask module that used to redirect the error definition to the web page is `jsonify()` which can import in Python from Flask library. `Jsonify` converts the JSON output in a response object with the application or JSON mimetype. So for the non zero outputs, `jsonify()` will define the status value as an error and then after obtaining the correct error, the individual error value displays on the screen. Fig 3.5 and Fig 3.6 show the output scheme of the both CLI terminal and web interface when the authentication information is wrongly defined to connect to the switches.



New Private Peering Configuration Data

Vlan ID

Vlan Name

VTP Server

IP

Port1

Port2

Device	Port1	Port2	Delete
192.168.2.10	Gi1/0/23	Gi1/0/27	<input style="width: 40px;" type="button" value="Delete"/>
192.168.2.20	Gi2/0/7	Gi2/0/8	<input style="width: 40px;" type="button" value="Delete"/>
192.168.2.30	Gi1/0/9	Gi1/0/25	<input style="width: 40px;" type="button" value="Delete"/>

Figure 3.3. Scheme of the Web template before starting the configuration

```
errordict={
  '1000': 'No Such Interface in Switch',
  '200': 'Wrong VTP Server Address...IT Is in Client Mode...Please Check Again',
  '151': 'Not Such IP Address',
  '166': 'VTP Server IP Address is not True',
  '1': 'Authentication Failed,Please Check Authentication information'
}
```

Figure 3.4. Private Peering Error list


```

127.0.0.1 - - [07/Aug/2018 11:52:02] "POST /result HTTP/1.1" 200 -
#####
Connecting to VTP server to add the new vlan
Traceback (most recent call last):
  File "/home/hoss/.local/lib/python3.5/site-packages/netmiko/base_connection.py", line 689, in establish_connection
    self.remote_conn_pre.connect(**ssh_connect_params)
  File "/home/hoss/.local/lib/python3.5/site-packages/paramiko/client.py", line 424, in connect
    passphrase,
  File "/home/hoss/.local/lib/python3.5/site-packages/paramiko/client.py", line 714, in _auth
    raise saved_exception
  File "/home/hoss/.local/lib/python3.5/site-packages/paramiko/client.py", line 701, in _auth
    self.transport.auth_password(username, password)
  File "/home/hoss/.local/lib/python3.5/site-packages/paramiko/transport.py", line 1381, in auth_password
    return self.auth_handler.wait_for_response(my_event)
  File "/home/hoss/.local/lib/python3.5/site-packages/paramiko/auth_handler.py", line 226, in wait_for_response
    raise e
paramiko.ssh_exception.AuthenticationException: Authentication failed.
During handling of the above exception, another exception occurred:
Traceback (most recent call last):
  File ".../tptpt.py", line 140, in <module>
    vtp_connection = ConnectHandler(**device)
  File "/home/hoss/.local/lib/python3.5/site-packages/netmiko/ssh_dispatcher.py", line 178, in ConnectHandler
    return ConnectionClass(*args, **kwargs)
  File "/home/hoss/.local/lib/python3.5/site-packages/netmiko/base_connection.py", line 207, in __init__
    self.establish_connection()
  File "/home/hoss/.local/lib/python3.5/site-packages/netmiko/base_connection.py", line 698, in establish_connection
    raise NetMikoAuthenticationException(msg)
netmiko.ssh_exception.NetMikoAuthenticationException: Authentication failure: unable to connect cisco_ios 192.168.2.10:22
Authentication failed.
127.0.0.1 - - [07/Aug/2018 11:54:46] "POST /result HTTP/1.1" 200 -

```

Figure 3.5. Private Peering authentication failure schema in the CLI terminal

3.2 Private peering by SDN

As it is discussed in 2.1.3, the physical topology of the Private Peering has been created based on OpenFlow Network and by using the Mininet. It is mainly due to the fact that the current available networking devices in TOP-IX did not provide us the possibility to perform an OpenFlow approach. So the last part of this master thesis was created using the Mininet and ONOS as the control plane.

Mininet is used to solve the problem by creating prototypes and simulation in a virtual mode. Mininet provides rapid prototyping for large or small networks on a single system. The Mininet installation methods are explained in [16]. However, as a solution, the SDN environment is created by using an official virtual machine installed in VirtualBox. This machine is an Ubuntu 16.04.3 which is downloaded as a VM ova file from ONOS tutorial web site and contains ONOS and Mininet in a single machine to implement the proper environment.

The first step to create the proper environment is to run the Openflow controller (ONOS) as a single individual machine. As the machine already consists an installed ONOS, it just would needed to run the ONOS locally in it. So the ONOS template has been built by buck method from ONOS directory as a local buck daemon. It is of great importance to mention that ONOS controller comprises of ONOS Web GUI and ONOS CLI. ONOS web GUI is a single web application page that provides a visual interface to the controller. It consists of different modules that allowed to monitor and control the network from the web browser. By using "http://localhost:8181/onos/ui" web URL

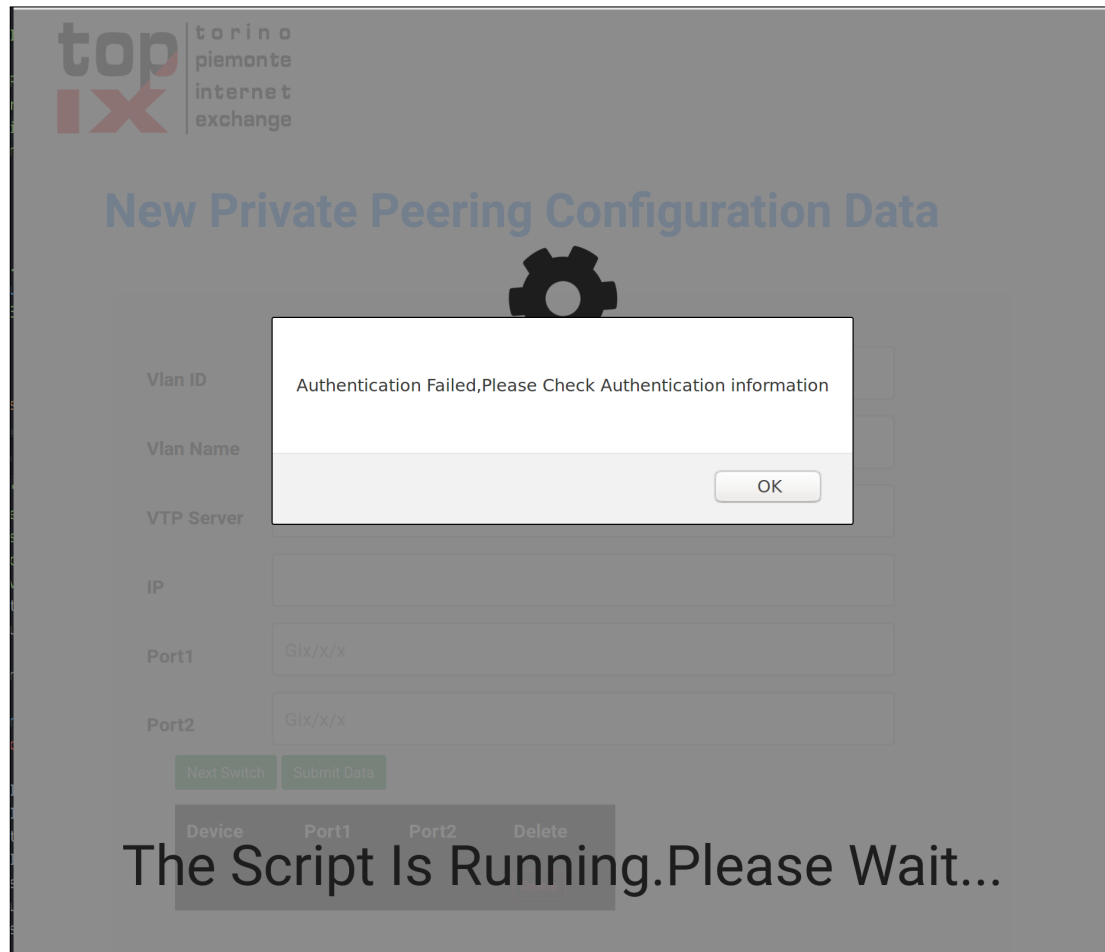


Figure 3.6. Private Peering authentication failure schema in the web interface

address, it is possible to access GUI on a localhost machine. ONOS CLI works as an extension of Karaf's CLI [21]. It gives the capability of leveraging features such as the ability to load and unload the ONOS applications and programmatic extensibility to manage the network. By using "onos localhost" on a machine terminal, it is possible to access to CLI of an ONOS which is running on a local machine.

Now it is time to create the physical topology by using Mininet. Mininet provides the possibility to create and interact with a software defined network prototype. In this work the custom network topology is created by using a Python script. The principal goal of this section is to create a network topology that consists of three OpenFlow switches which act as the TOP-IX network fabric and two routers as the customers router for Private Peering scenario. However, there is no individual module that performs like a layer-3 router in Mininet. So, to rectify this issue, the Mininet node (host), converts into the router using IP forwarding feature that already built into the Linux operating

systems. As it is discussed in [22], the networking routes can be inserted into the routing table by using "ip route" or "route" commands on the associated virtual nodes in Mininet. Additionally, based on the principal instruction of making Private Peering scenario, the routers must transfer the traffics in a completely private connection that labeled based on specific VLAN that is configured in the proper interfaces. So this is done by using the definition of VLANing feature in the Linux kernel. The VLAN installation requires first to get and install the Linux VLAN package by using "*sudo apt-get install vlan*" in the Linux terminal.

The Python class that is responsible for creating Openflow routers which also enables the VLAN features in the interfaces, is shown in Code 3.5.

```

1  class LinuxRouter( Node ):
2      "A Node with IP forwarding enabled."
3
4      def config( self, vlan=100, **params ):
5          x = super( LinuxRouter, self ).config( **params )
6          intf = self.defaultIntf()
7          self.cmd( 'ifconfig %s inet 0' % intf )
8          self.cmd( 'vconfig add %s %d' % ( intf, vlan ) )
9          self.cmd( 'ifconfig %s.%d inet %s' % ( intf, vlan, params['ip']
10                  ) )
11          newName = '%s.%d' % ( intf, vlan )
12          intf.name = newName
13          self.nameToIntf[ newName ] = intf
14          return x
15          # Enable forwarding on the router
16          self.cmd( 'sysctl net.ipv4.ip_forward=1' )
17 routers = { 'vlan': LinuxRouter }
```

Code 3.5. Creation of OpenFlow router in Mininet

This class permits to create nodes with enabled IP forwarding. As well as, attempts to create VLAN interfaces and assign the IP address to the networking interfaces. Additionally, it updates the router interfaces based on the proper name and VLAN ID. Then the routers are called in topology creation procedure in Mininet. The IP address of each router, as well as the VLAN Id, set like the implementation of Private Peering into the real devices. The IP subnet for the routers sets to 194.116.98.0/30 that can cover only two host IP addresses and the VLAN ID as the previous section 3.1.1 set to 77. So, the custom topology which is consists of three OpenFlow switches, two routers, and a single controller which runs in a local machine and will manage all the networking devices has been created and it can be seen in appendix.2. Then, the Mininet configuration Python script must run by root privilege. It creates the physical scheme of the OpenFlow network. The solution uses the REST API to upload the JSON configuration file which consists of the interfaces information and VPLS structure that interfaces must use. In order to allow the VPLS to verify the network connectivity between two routers, there need to configure the interfaces of each router in the ONOS interfaces configuration.

The Fig 3.7 illustrates the scheme of the OpenFlow topology from ONOS web GUI for Private Peering implementation scenario.

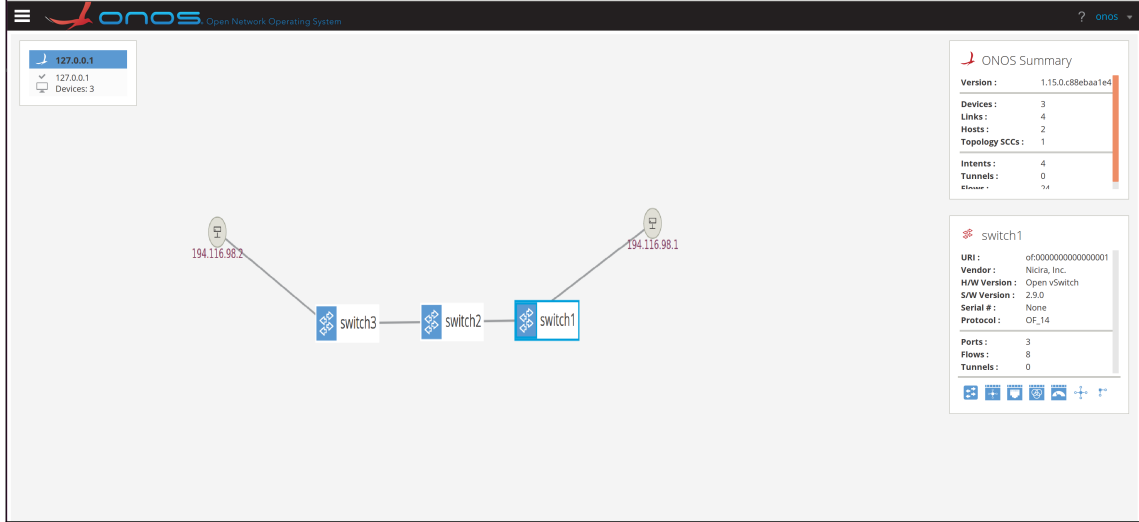


Figure 3.7. ONOS Web GUI

After setting up the physical network topology, now it is time to discuss about the VPLS and the way that it encapsulates the OpenFlow traffics. As it is discussed earlier 2.1.3, VPLS is an ONOS application that permits to provision and creates L2 circuits between various end-point OpenFlow devices. It generally manages the virtual private LANs by providing the public API. The CURD functionality to create/update/read/delete has been provided by this API which benefits to other applications to work with VPLS components. VPLS manager also can handle the events that may occur to the hosts while want to attach or detach to VPLS list.

VPLS uses the Intent framework to provision the broadcast/unicast connection between the interfaces of the hosts (here routers). The usage of intent definition in VPLS lets to minimize the complexity of provisioning of the flows on the OpenFlow switches and performs the error recovery in case of failures occur. The ONOS concept allows the intent and interprets them by intent compilation procedure into installable intents. The different components of the VPLS is shown in Fig 2.1. It needs to know that a new VPLS operation will be queued when it is generated and submitted to VPLS operation service. There is an operation scheduler that performs the optimization of multiple operations if there exists. The VPLS operation executor is the component that generates different intents based on different operational task like add, remove or update of the VPLS lists. It is important to know that the VPLS application requires at least two interfaces to be configured in the ONOS interface configuration context and be associated to the same VPLS. In this scenario, the interfaces are related to two OpenFlow routers that mentioned earlier.

The operation executor will produce two types of intents when the new interface of the OpenFlow edge-node wants to add into the VPLS list. The first one is Single-to-Multi point intent which is for the broadcast traffic and other is Multi-to-Single point intent for unicast traffics. Single-to-Multi is used to provide broadcast provisioning intents for the objects inside the same VPLS list. The intent ingress (source host/router) selector is determined using the edge in-port, the broadcast destination MAC address (FF:FF:FF:FF:FF:FF), and the individual VLAN Id of the source node. The egress points are related to all of the other edge ports that the destination hosts are associated and connected to in the same VPLS. However, the Multi-to-Single intents provide the unicast definition in the same VPLS list and have the opposite definition in term of ingress and egress points comparing to the broadcast intents. The Fig 3.8 shows the intents installed after establishing a new VPLS list called "VLAN77" and after adding the proper interfaces into the VPLS list by ONOS CLI. So regarding the scenario, as there are two interfaces related to the routers, the VPLS operation executor creates two intents for each object. As well as, VPLS accepts three type of encapsulation methods. (1)None which means that no encapsulation mode is applied for the provided list, (2)VLAN which is based IEEE802.1Q encapsulation method, and (3) ¹MPLS (Multiprotocol Label Switching). Fig 3.8 shows that the encapsulation type is based on VLAN method.

Additionally, we can also check the behavior of the flow rule subsystem in the ONOS. The flow rule subsystem is responsible for installing and managing the flow rules in the networking devices. This subsystem uses the definition of the distributed flow table that a master copy of the rules that prevails with the controller and will then be pushed down into the devices. It needs to mention that the flows are installed into the subsystem through the FlowRuleServe API. The figures 3.9, 3.10, and 3.11 shows the dump flows that the ONOS applications pushed into the OpenFlow switches. There are different flows for different traffics that are found by the controller. They are the outputs of the "**flows -s**" from the ONOS CLI.

Fig 3.9 shows the dump flows that pushed by ONOS into the devices based on different different flow ids and traffic selectors that identifies the requirement rules to apply into the OpenFlow traffics. So based on the default applications that installed in the ONOS, we can see some specific rules like ²BDDP and LLDP (links discovery methods), ³ARP, and IPv4 traffics.

Fig 3.10 shows that the when activating the VPLS application in controller, it will

¹A Layer-3 VPN label for each route - RFC4364

²BDDP and LLDP are LinkDiscovery mechanisms that ONOS uses based on probes to discover paths in Openflows network. A LinkDiscovery instance sends out probe messages that is containing LLDPs and BDDPs. LLDP probes are dropped after one network hop on both SDN controlled and legacy network.

³Address Resolution Protocol (ARP) to map the IP network address to the hardware address.

```

onos> intents
Id: 0x0
State: INSTALLED
Key: Vlan77-brc-of:0000000000000001-2-FF:FF:FF:FF:FF:FF
Intent type: SinglePointToMultiPointIntent
Application Id: org.onosproject.vpls
Leader Id: 127.0.0.1
Common ingress selector: ETH_DST:FF:FF:FF:FF:FF:FF
Treatment: [NOACTION]
Constraints: [PartialFailureConstraint, EncapsulationConstraint{encapType=VLAN}]
Ingress connect points and individual selectors
-> Connect Point: of:0000000000000001/2 Selector: VLAN_VID:77
Egress connect points and individual selectors
-> Connect Point: of:0000000000000003/2 Selector: VLAN_VID:77

Id: 0x9
State: INSTALLED
Key: Vlan77-uni-of:0000000000000001-2-7A:CA:40:AA:B4:15
Intent type: MultiPointToSinglePointIntent
Application Id: org.onosproject.vpls
Leader Id: 127.0.0.1
Common ingress selector: ETH_DST:7A:CA:40:AA:B4:15
Treatment: [NOACTION]
Constraints: [PartialFailureConstraint, EncapsulationConstraint{encapType=VLAN}]
Ingress connect points and individual selectors
-> Connect Point: of:0000000000000003/2 Selector: VLAN_VID:77
Egress connect points and individual selectors
-> Connect Point: of:0000000000000001/2 Selector: VLAN_VID:77

Id: 0x8
State: INSTALLED
Key: Vlan77-uni-of:0000000000000003-2-4A:99:65:E0:66:9D
Intent type: MultiPointToSinglePointIntent
Application Id: org.onosproject.vpls
Leader Id: 127.0.0.1
Common ingress selector: ETH_DST:4A:99:65:E0:66:9D
Treatment: [NOACTION]
Constraints: [PartialFailureConstraint, EncapsulationConstraint{encapType=VLAN}]
Ingress connect points and individual selectors
-> Connect Point: of:0000000000000001/2 Selector: VLAN_VID:77
Egress connect points and individual selectors
-> Connect Point: of:0000000000000003/2 Selector: VLAN_VID:77

Id: 0x1
State: INSTALLED
Key: Vlan77-brc-of:0000000000000003-2-FF:FF:FF:FF:FF:FF
Intent type: SinglePointToMultiPointIntent
Application Id: org.onosproject.vpls
Leader Id: 127.0.0.1
Common ingress selector: ETH_DST:FF:FF:FF:FF:FF:FF
Treatment: [NOACTION]
Constraints: [PartialFailureConstraint, EncapsulationConstraint{encapType=VLAN}]
Ingress connect points and individual selectors
-> Connect Point: of:0000000000000003/2 Selector: VLAN_VID:77
Egress connect points and individual selectors
-> Connect Point: of:0000000000000001/2 Selector: VLAN_VID:77

```

Figure 3.8. The instruction of installed intents from ONOS CLI

create two new flows in all of the switches that are based on incoming and outgoing traffics for different switch ports and characterized by different VLAN Ids. Before sending any traffic between edge-nodes, the destination hardware address is set to the Broadcast address for the edge switches (Switch 1 and 2). Fig 3.11 shows when sending out a unicast IP traffic (Ping), the switches could define the MAC address of the edge nodes by using ARP protocol. The switches then add the two unicast flows for different ports which are based on the incoming and outgoing traffics into their flow lists.

```

onos> flows -s
deviceId=of:0000000000000001, flowRuleCount=4
  ADDED, bytes=1251, packets=9, table=0, priority=40000, selector=[ETH_TYPE:bddp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
  ADDED, bytes=1251, packets=9, table=0, priority=40000, selector=[ETH_TYPE:lldp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
  ADDED, bytes=0, packets=0, table=0, priority=40000, selector=[ETH_TYPE:arp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
  ADDED, bytes=0, packets=0, table=0, priority=5, selector=[ETH_TYPE:ipv4], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
deviceId=of:0000000000000002, flowRuleCount=4
  ADDED, bytes=2502, packets=18, table=0, priority=40000, selector=[ETH_TYPE:lldp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
  ADDED, bytes=2502, packets=18, table=0, priority=40000, selector=[ETH_TYPE:arp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
  ADDED, bytes=2502, packets=18, table=0, priority=40000, selector=[ETH_TYPE:bddp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
  ADDED, bytes=0, packets=0, table=0, priority=5, selector=[ETH_TYPE:ipv4], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
deviceId=of:0000000000000003, flowRuleCount=4
  ADDED, bytes=1251, packets=9, table=0, priority=40000, selector=[ETH_TYPE:bddp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
  ADDED, bytes=1251, packets=9, table=0, priority=40000, selector=[ETH_TYPE:lldp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
  ADDED, bytes=0, packets=0, table=0, priority=40000, selector=[ETH_TYPE:arp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
  ADDED, bytes=0, packets=0, table=0, priority=5, selector=[ETH_TYPE:ipv4], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]

```

Figure 3.9. Default ONOS dump flows before activating VPLS

```

onos> flows -s
deviceId=of:0000000000000001, flowRuleCount=6
  ADDED, bytes=13900, packets=100, table=0, priority=40000, selector=[ETH_TYPE:bddp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
  ADDED, bytes=13900, packets=100, table=0, priority=40000, selector=[ETH_TYPE:lldp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
  ADDED, bytes=0, packets=0, table=0, priority=1100, selector=[IN_PORT:1, VLAN_VID:2195], treatment=[immediate=[VLAN_ID:77, OUTPUT:2]]
  ADDED, bytes=0, packets=0, table=0, priority=1100, selector=[IN_PORT:2, ETH_DST:FF:FF:FF:FF:FF:FF, VLAN_VID:77], treatment=[immediate=[VLAN_ID:2069, OUTPUT:1]]
  ADDED, bytes=0, packets=0, table=0, priority=5, selector=[ETH_TYPE:ipv4], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
deviceId=of:0000000000000002, flowRuleCount=6
  ADDED, bytes=27800, packets=200, table=0, priority=40000, selector=[ETH_TYPE:arp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
  ADDED, bytes=27800, packets=200, table=0, priority=40000, selector=[ETH_TYPE:bddp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
  ADDED, bytes=0, packets=0, table=0, priority=1100, selector=[IN_PORT:2, VLAN_VID:1570], treatment=[immediate=[VLAN_ID:2195, OUTPUT:1]]
  ADDED, bytes=0, packets=0, table=0, priority=1100, selector=[IN_PORT:1, VLAN_VID:2069], treatment=[immediate=[VLAN_ID:3994, OUTPUT:2]]
  ADDED, bytes=0, packets=0, table=0, priority=5, selector=[ETH_TYPE:ipv4], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
deviceId=of:0000000000000003, flowRuleCount=6
  ADDED, bytes=13900, packets=100, table=0, priority=40000, selector=[ETH_TYPE:bddp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
  ADDED, bytes=13900, packets=100, table=0, priority=40000, selector=[ETH_TYPE:lldp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
  ADDED, bytes=0, packets=0, table=0, priority=1100, selector=[IN_PORT:2, ETH_DST:FF:FF:FF:FF:FF:FF, VLAN_VID:77], treatment=[immediate=[VLAN_ID:1570, OUTPUT:1]]
  ADDED, bytes=0, packets=0, table=0, priority=1100, selector=[IN_PORT:1, VLAN_VID:3994], treatment=[immediate=[VLAN_ID:77, OUTPUT:2]]
  ADDED, bytes=0, packets=0, table=0, priority=5, selector=[ETH_TYPE:ipv4], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]

```

Figure 3.10. ONOS dump flows after activating VPLS

3.2.1 Benchmark

There are many aspects to be compared between the enterprise network (TCP/IP based network) with respect to SDN based on the OpenFlow protocol. The fundamental rule of using the SDN network based on OpenFlow forwarding protocol is that the data plane separated with forwarding plane and is directly programmable by an SDN controller. SDN provides a centralized view of the entire network and can accelerate service delivery in provisioning new configuration into the networking devices.

Before preceding to the next steps, to compare the two approaches, it is essential to represent some information about the working environment of the two techniques. As it has been discussed before, the SDN testbed has been created in a completely virtual environment and in a personal computer with limited resources. Those resources need to be normalized and shared among virtual routers and switches. Mininet provides the opportunity to create a virtual environment and consists of the software switching definition rather than hardware switching. So that would probably cause a significant drawback in the long run for the performance of the controller since all of the OpenFlow devices and the ONOS controller had been created in the same system. In a real network, switches and hosts could be operated in parallel and independently. In contrast, in an emulated environment, parallel operations are executed sequentially due to the limited number of CPU cores. However, It should be mentioned that the main purpose to automate the Private Peering was not to the performance of the network connectivity since most of the


```

onos> flows -s
deviceId:0000000000000001, flowRuleCount=8
ADDED, bytes=84095, packets=605, table=0, priority=40000, selector=[ETH_TYPE:bdp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
ADDED, bytes=84095, packets=605, table=0, priority=40000, selector=[ETH_TYPE:lldp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
ADDED, bytes=3266, packets=71, table=0, priority=40000, selector=[ETH_TYPE:arp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
ADDED, bytes=120666, packets=1183, table=0, priority=1200, selector=[IN_PORT:2, ETH_DST:EE:E7:6F:8B:53:F2, VLAN_ID:77], treatment=[immediate=[VLAN_ID:3920, OUTPUT:1]]
ADDED, bytes=120768, packets=1184, table=0, priority=1200, selector=[IN_PORT:1, VLAN_ID:468], treatment=[immediate=[VLAN_ID:77, OUTPUT:2]]
ADDED, bytes=0, packets=0, table=0, priority=1100, selector=[IN_PORT:1, VLAN_ID:2195], treatment=[immediate=[VLAN_ID:77, OUTPUT:2]]
ADDED, bytes=0, packets=0, table=0, priority=1100, selector=[IN_PORT:2, ETH_DST:FF:FF:FF:FF:FF:FF, VLAN_ID:77], treatment=[immediate=[VLAN_ID:2069, OUTPUT:1]]
ADDED, bytes=510, packets=5, table=0, priority=5, selector=[ETH_TYPE:ipv4], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
deviceId:0000000000000002, flowRuleCount=8
ADDED, bytes=168190, packets=1210, table=0, priority=40000, selector=[ETH_TYPE:lldp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
ADDED, bytes=0, packets=0, table=0, priority=40000, selector=[ETH_TYPE:arp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
ADDED, bytes=168190, packets=1210, table=0, priority=40000, selector=[ETH_TYPE:bdp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
ADDED, bytes=120768, packets=1184, table=0, priority=1200, selector=[IN_PORT:2, VLAN_ID:3469], treatment=[immediate=[VLAN_ID:468, OUTPUT:1]]
ADDED, bytes=120666, packets=1183, table=0, priority=1200, selector=[IN_PORT:1, VLAN_ID:3920], treatment=[immediate=[VLAN_ID:1222, OUTPUT:2]]
ADDED, bytes=0, packets=0, table=0, priority=1100, selector=[IN_PORT:2, VLAN_ID:1570], treatment=[immediate=[VLAN_ID:2195, OUTPUT:1]]
ADDED, bytes=0, packets=0, table=0, priority=1100, selector=[IN_PORT:1, VLAN_ID:2069], treatment=[immediate=[VLAN_ID:3994, OUTPUT:2]]
ADDED, bytes=510, packets=5, table=0, priority=5, selector=[ETH_TYPE:ipv4], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
deviceId:0000000000000003, flowRuleCount=8
ADDED, bytes=84095, packets=605, table=0, priority=40000, selector=[ETH_TYPE:bdp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
ADDED, bytes=84095, packets=605, table=0, priority=40000, selector=[ETH_TYPE:lldp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
ADDED, bytes=3312, packets=72, table=0, priority=40000, selector=[ETH_TYPE:arp], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]
ADDED, bytes=120666, packets=1183, table=0, priority=1200, selector=[IN_PORT:1, VLAN_ID:1222], treatment=[immediate=[VLAN_ID:77, OUTPUT:2]]
ADDED, bytes=120768, packets=1184, table=0, priority=1200, selector=[IN_PORT:2, ETH_DST:86:1F:95:CE:97:FF, VLAN_ID:77], treatment=[immediate=[VLAN_ID:3469, OUTPUT:1]]
ADDED, bytes=0, packets=0, table=0, priority=1100, selector=[IN_PORT:2, ETH_DST:FF:FF:FF:FF:FF:FF, VLAN_ID:77], treatment=[immediate=[VLAN_ID:1570, OUTPUT:1]]
ADDED, bytes=0, packets=0, table=0, priority=1100, selector=[IN_PORT:1, VLAN_ID:3994], treatment=[immediate=[VLAN_ID:1577, OUTPUT:2]]
ADDED, bytes=510, packets=5, table=0, priority=5, selector=[ETH_TYPE:ipv4], treatment=[immediate=[OUTPUT:CONTROLLER], clearDeferred]

```

Figure 3.11. ONOS dump flows after sending a Unicast IP traffic (Ping)

devices in the enterprise environment (real test) were in Cisco family and had the good connectivity. Working in a real environment and dealing with real network traffic with a lot of constraints is completely different with a virtual situation. The principal benefits of SDN over common approaches are based on the fact that SDN permits to simultaneously test and use new applications, minimizes operating risks and allows centralized management of each switch.

As it has been discussed, the main idea of automating the Private Peering was focusing on the agile development of the process and to limit the human errors that might happen during configuration. Comparing the configuration period between two approaches operationally is not efficient. Initially, due to the fact that preparing the SDN environment consists of activating the controller and individual applications, creating the physical topology by Mininet and pushing the proper interfaces information would take a significant amount of time (almost 6 minutes). This procedure varies based on available resources and the performance of the system. However, by neglecting the required time to activate the SDN controller and considering that the physical network is already up and is connected to the controller, the amount of time required to configure VPLS application to create a Layer-2 broadcast domain is negligible. As the controller is integrated with all of the networking elements, pushing the new configurations will be done very quick. The controller would adopt the OpenFlow protocol to communicate with OpenFlow switches.

3.3 BIRD automation

Like the previous task, this task also will be divided into two parts. First, the definition of the task and solution demonstration. The second part will be concentrated on the visual appearances that solved by designing a web UI to gather the essential information and run the Bird daemon.

3.3.1 Backend

As it has been discussed in 2.1.2, the BIRD configuration procedure in TOP-IX comprises of different steps. These steps modify the route server features to work automatically. The main script which is responsible to create the BIRD configuration file is called "configure.sh". This bash script is running every day to update the BGP client prefix lists and to let the BIRD to advertise the peering sessions to the route server's members. The script manages both IP versions (V4 and V6) routing configuration. Additionally, the bash script runs a Python program that is called "create_config.py" which is responsible to directly create the configuration files by predefined sources and templates.

To create the configuration files, it is necessary to store the client or route server's member information. The objects that are used to this purpose are a set of YAML files and are different based on IP address versions. The Python script uses those YAML files that are comprises of the information to create the BIRD configuration files. The process in TOP-IX to add a new client information to the main client list was fully hand-operated. As the principal demand for this part of my thesis, there need to find a way to automatically update the client list after gathering the information from the web UI. Alternatively, there need to find a mechanism to verify whether the new client that is going to add to the YAML file is new or is a duplicated one.

Before addressing the web user interface (UI) creation structure and the process that consolidates the information, it is of the great importance to illustrate the required adjustments that would need to perform in the TOP-IX Python script. The function that is responsible for loading the client list is called "load_clients" that is characterized in terms of version of the IP address and also the appropriate route server.

For the principal step, a Python list has been created to load the TOP-IX's YAML client file into it. After loading the YAML file, inside a loop, all of the IP addresses will be appended into the list. The only object which is unique amongst clients is the IP address of the members, so it is likely to check if the new client requires to add in the TOP-IX client list or not. If the new client is not a duplicated one, it must be added to the YAML file with all of the required parameters in a way that can readable to the

route server. The procedure and the solution to get the information from web UI and the way to show the error when the new entry is duplicated will be discussed in the frontend implementation section. What needs to be mentioned here is that, as a solution, for the new client information, the web UI will make first a JSON template and then will dump it into a temporary YAML file. The temporary YAML file structured in a way that the BIRD daemon can use it after adding to the end of the main BIRD's client list.

The Fig 3.12 shows the lines of the client file (YAML file). It provides the general policies and clients configuration options and the required information that needs for the members who want to use the route server features.

As we can see from Fig 3.12, each client specified by an Autonomous System (AS) number, as well as the other related information. The reason why and how they are using is out of the scope of this thesis. It can be seen that the clients separated by an empty line and started by a dash at the beginning and before the "as" keyword.

So as the final step for the Backend programming section, we need to find a solution to add the temporary YAML file which contains of the new information of the new member to the end of the TOP-IX client file. It can be done by using read/write file module in Python which is a native and unique feature in the language. Firstly, it needs to open the file by `open()` function which must return to a file object (file descriptor). File object consists of the attributes and the methods that can be used to gather the information about the opened file. To open a file in Python by builtin `open()` function, we can specify various modes as the syntax arguments. Briefly, the modes can be "r" to only read the file, "w" used for editing and writing the information into the file and "a" which is appending mode to add new data to the end of the file. After opening the files, it is time to read the file, as the specific characters by `file.read()` method. Both of the YAML files must be opened but the client files in the appending mode and the temporary file in the reading mode. The next step is to write the exact characters from the temporary file to the client file by the `file.write()` function. Alternatively, we need to put a separation between the clients by the empty space line and it can be simply done by adding a `"\n"`, as a syntax argument in the write function. In the end, both files must be closed to terminate resources that are in use to make them available for the system by `close()` function.

3.3.2 Frontend

For the BIRD configuration automation task, a web UI has been created to get the information and produce the template YAML file. The tools that has been used for web form creation is Flask and JavaScript that introduced in 3.1.2. The method of web UI creation and its performance is almost the same with 3.1.2. However, in this scenario, as the final step, we must transfer the data from the JSON to the YAML file. It is because of the reason that the Flask form can export the data into a JSON file. It has

```
- as: 30919
# description: AFETI
  name: AFETI
  import: AS30919
  ip: 194.116.96.240
  ixtype: franceix
  asexport: 30919

- as: 34019
# description: HIVANE
  name: HIVANE
  import: AS-HIVANE
  ip: 194.116.96.241
  ixtype: franceix
  asexport: 34019

- as: 41157
# description: OXYMIUM
  name: OXYMIUM
  import: AS41157
  ip: 194.116.96.242
  ixtype: franceix
  asexport: 41157

- as: 35625
# description: ATE
  name: ATE
  import: AS-ATE
  ip: 194.116.96.243
  ixtype: franceix
  asexport: 35625

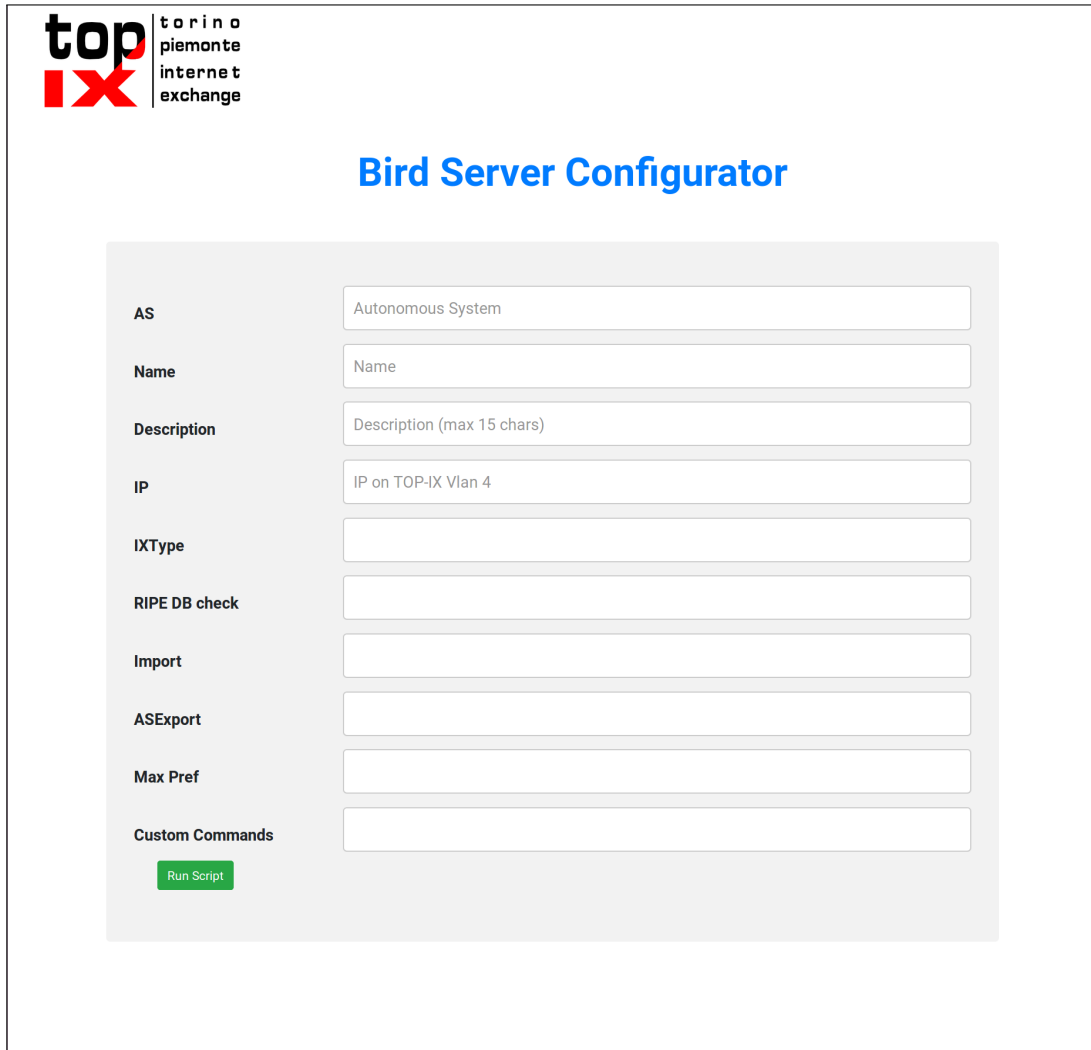
- as: 34422
# description: LPGH
  name: LPGH
  import: AS-LPGHC
  ip: 194.116.96.244
  ixtype: franceix
  asexport: 34422

- as: 65535
# description: GOOGLE_CACHE
  name: GOOGLE_CACHE
  import: AS65535
  ip: 194.116.101.16
  ixtype: cdn
  asexport: 65535
  custom_commands: multihop 255
  limit: 0
```

Figure 3.12. Structure of TOP-IX client file

been discussed in 3.1.2 that, AJAX would be a decent tool to produce the JSON file and exclude the errors. The selected error that must be observed in case of happening is when the new entry is already inside the list with the same IP address.

Fig 3.13 shows the empty web template that has been created to get the new information of the new peer for the Bird daemon configuration. Fig 3.13 shows the required



topix torino piemonte internet exchange

Bird Server Configurator

AS	Autonomous System
Name	Name
Description	Description (max 15 chars)
IP	IP on TOP-IX Vlan 4
IXType	
RIPE DB check	
Import	
ASEExport	
Max Pref	
Custom Commands	

[Run Script](#)

Figure 3.13. Scheme of the web UI for updating BIRD

information that would be needed, if a new client wants to use Bird’s features such as advertising the BGP prefixes to other members. As it can be illustrated from Fig 3.12, the fill out policy of the web UI is different per member. Means that there might be cases like the last insertion in Fig 3.12 that consists of two more lines (custom_command and limit) than the others. The Bird configurator, by using the proper ¹Jinja2 templates, will select the proper data based on the peer’s condition and will create the BIRD configuration file.

¹A full-featured template engine for Python that supports the Unicode, as an arbitrary integrated sandboxed execution context. It is inspired by Django’s templating system but extends it with a strong language by providing a very powerful set of tools to template authors.

The Flask controller sets the output JSON file from the web UI by using `"request.get_json()"` into a Python list object. Here, a template YAML file which consists of the BIRD required elements (as, name, import, ...) as the YAML object keys and without any YAML object values. Those elements (YAML keys) are precisely the same as the objective elements of the main TOP-IX client file. Then, in a loop and by using the verification methods, the associated object keys pushed from the web UI into the JSON object will be copied into the temporary YAML file. It is worth mentioning that the stipulated names for the key elements in both templates (YAML and JSON) files are defined similarly to limit the potential obstacles that might happen in the script and gain the code more straightforward. So, after preparing the temporary YAML file, it is time to verify the new member's IP address and define whether it is a new entry or is the duplicated one. In case of duplication, the Bird configuration procedure will not start its cycle and will expose an error message into the web browser to notify the reason for the error. The solution to rectify this issue is like the previous task 3.1.2 that used the Flask's `jsonify` module. `Jsonify` will redirect the error message from the error list to the Javascript template and then to show to the user.

If no duplication happens, the script will run the Bird configurator bash command that has been discussed in 3.3.1. However, as the file directories for the web template and the bash file are different, there need to change the directory environment before compiling the bash file. Additionally, the bash script should get as the syntax argument, the name of the route server because there are two route servers currently working in the TOP-IX consortium. The bash script *configure.sh* will create the bash configuration files automatically based on the information that prepared to it. Fig 3.14 and Fig 3.15 show the final output in backend (CLI) and frontend (Web browser), after filling out the web form with corresponding data and pushing the "Run Script" button at the left bottom of the web UI. As it is highlighted in Fig 3.14, the route server could attain new object that appended at the end of the client YAML file and could run or reconfigure the route server accurately.

```
Use a production WSGI server instead.
* Debug mode: on
* Running on http://192.168.2.254:5002/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 123-864-787
Start configuration for rs1
-----
--- IPv4 section ---
-----
Script PID 3547 called with following params:
-ip_ver: 4
-rs: rs1
Found 119 peer
Script takes 0.266172 CPU seconds
Script takes 0.266241073608 global seconds
Configuration IPv4 correctly created
[OK] Configuration changed. I'm going to configure bird
[OK] BIRD successfully reconfigured.
BIRD 1.6.4 ready.
kernel1: reloading
device1: reload failed
static1: reload failed
[OK] BIRD successfully reloaded.
192.168.2.2 - - [07/Aug/2018 14:55:55] "POST /post_json HTTP/1.1" 200 -
```

Figure 3.14. Output in CLI for the BIRD configuration

The screenshot shows a web interface for configuring BIRD. At the top left is the logo for 'topix torino piemonte internet exchange'. A modal window titled 'Task Finished' is displayed, containing the message: 'THE CLIENT LIST HAS UPDATED SUCCESSFULLY AND THE SCRIPT FINISHED ITS INSTRUCTIONS'. Below the modal is a form with the following fields:

Field	Value
AS	Autonomous System
Name	Name
Description	Description (max 15 chars)
IP	IP on TOP-IX Vlan 4
IXType	
RIPE DB check	
Import	
ASExport	
Max Pref	
Custom Commands	

At the bottom of the form is a green button labeled 'Run Script'.

Figure 3.15. Output in Web UI for the BIRD configuration

And at the end, the Fig 3.16 represents that the client YAML file updated correctly with new information that produced by the web UI. It can see that the structure of the new entry data is almost equal with the previously added configuration data.

```
- as: 34019
# description: HIVANE
  name: HIVANE
  import: AS-HIVANE
  ip: 194.116.96.241
  ixtype: franceix
  asexport: 34019

- as: 41157
# description: OXYMIUM
  name: OXYMIUM
  import: AS41157
  ip: 194.116.96.242
  ixtype: franceix
  asexport: 41157

- as: 35625
# description: ATE
  name: ATE
  import: AS-ATE
  ip: 194.116.96.243
  ixtype: franceix
  asexport: 35625

- as: 34422
# description: LPGH
  name: LPGH
  import: AS-LPGHC
  ip: 194.116.96.244
  ixtype: franceix
  asexport: 34422

- as: 65535
# description: GOOGLE_CACHE
  name: GOOGLE_CACHE
  import: AS65535
  ip: 194.116.101.16
  ixtype: cdn
  asexport: 65535
  custom_commands: multihop 255
  limit: 0

- as: '1234'
  asexport: '1234'
  custom: ''
  description: this is test
  import: AS1234
  ip: 192.168.2.10
  ixtype: topix
  max: ''
  name: TEST2
  ripe: ''
```

Figure 3.16. Updated version of client YAML file

Chapter 4

Conclusions

This thesis is structured by different components and programming languages to design, implement, and help to improve the networking quality. The principal goal of this thesis was to find the best solution to connect the software developing features into the network operation and engineering. In fact, we tried to use the best practices, tools, and ideas from software development to the networking area.

Network monitoring and moving from the old models with SNMP, Syslog, and NetFlow and adopting new strategic models around machine learning approaches is part of the network automation. The goal was to understand how those tools are fundamentally working and enable them to the operational scenarios. In network automation, we can take the concept of DevOps like configuration management. Basically, DevOps indicates to the enterprise software development that uses an agile relationship between developers and IT operational team. It tries to improve this relationship by promoting better interaction between these two business units.

The objective of this master thesis also tried to highlight the importance of using Software Defined Network in today's networking paradigm. As it has been discussed, the fundamental aspects of the SDN were to define a distinct separation between network control plane and network forwarding and the transition of network logic from hardware implementation to software.

As it has been discussed, this thesis has been divided into three totally practical works. (1) The automation of Private Peering, as an extensive service which is offered by TOP-IX both in the real environment and software based environment using different tools. (2) Implementing an automated process to update and maintain the BIRD routing daemon. The automation of Private Peering firstly done using the real operational Cisco switches in TOP-IX. The solution used Netmiko, an open-source Python library

to send the configuration management instructions dynamically and a Web UI based on Flask and JavaScript to effectively get the required information from the Network administrators and try to create a visual representation if any error happens during the configuration procedure. The solution used a multi-threading method based on Python to send the configuration commands to multiple devices in parallel and at the same time. The proposed solution can be readily used in practice. The approach is applicable even for much bigger areas which consist of a higher number of devices to configure. As an alternative approach, a virtualized OpenFlow environment created by using Mininet and ONOS to design a proper situation based on Private Peering implementation needs.

4.1 The issues raised during the implementation

During Private Peering automation in real environment [3.1.1](#), the issues were mainly related to the structure of the API (Netmiko) and the way to connect it to the project definition. As it is mentioned before in [3.1.1](#), to configure multiple devices by Netmiko, by default it will send commands first to one device and then to the next one, separately and does not provide any parallelization definition. So, to minimize the configuration period, we must define a solution to prevent waiting time for the individual devices. To do so, I used the multithreading definition in Python. However, multithreading demands more precise programming to avoid non-intuitive behaviors such as deadlocks and racing conditions.

As well as, the other issue was related to the time that needs to check the current setting of the switch interfaces and finding the best solution to dynamically configure the devices based on the proper VLAN. Before finding the best solution which was discussed in [3.1.1](#), three other Cisco show commands had experimented but none of them has the adaptability of "show running config interface" command to different interface setups. This command shows the current configuration for the individual interface which involves the VLAN configuration.

The other time-consuming issue was related to the connection between the Frontend and Backend and to create a readable JSON file from the Frontend. It must consider that the problems were mainly associated with JavaScript modules and error establishment procedure. For instance, the solution for Private Peering automation used three operational scripts. The API configuration script based on Python, the backend Flask controller based on Python and the Frontend web UI script based on JavaScript. The final solution is directly related to connectivity and proper synchronization between related scripts.

4.2 Importance of Network Automation for IXPs

As it has been discussed many times, network automation can make the work easier. However, the knowledge of programming fundamentals is ultimately crucial for network automation. There are multiple possible operational approaches in IXP's throughout the world that are related to automate tasks. Automation is of a great importance in IXP's firstly to get rid of the error-prone manual configuration jobs which in many cases are related to human mistakes. In most instances, the customers must wait for an individual person in IXP who is directly associated to maintain the manual changes. It might happen that the networking equipment in customer side fails and they need to replace with the new device and so the MAC address will change. The new device port will not work until in IXP, they change the configuration based on new information. So, it is essential that in the customer side, they have the possibility to inform the IXP about any changes by using the user interfaces and in the IXP side, the device maintenance performed in a automatic and dynamic way. By using automation approaches, the configuration does not need a human decision in IXP. They can be done any time, in different situations and for various purposes.

Additionally, automation can operationally perform to configure the IXP participant edge ports such as ¹Dot1q framing, layer 2 filters, speed controlling and so on. It is also possible to go one step ahead and configure the IXP core features. There are a significant amount of configuration items for the network in IXP which are massively repetitive and almost impossible to visually inspect for the correctness. Each of those configuration items might have multiple steps associated with it like having different AS numbers or IP addresses or interface numbers. So there are a lot of types of data that all might look the same and so manual configuration might face with critical problems.

The other very important aspect is that the automated data models must be portable. The automation tasks should not be limited just for one IXP or just for a particular network device. It is a critical matter that should be taken into consideration before starting to automate complex tasks and to limit the additional time-consuming efforts to debug or to update the code for the new environments. However, this compatibility needs a very well-defined plan with the knowledge of the numerous conditions which is rather very difficult to achieve.

Automation is also a good solution for the security of the IXP network. There are plenty of threats to IXP's which can cause a very harmful impact on the performance of the network. Additionally, automation can help network security professionals with their leading jobs like policy updating, finding hidden threats, extend network protections

¹Dot1q framing or VLAN tagging is a technique developed by Cisco to help classify packets moving through trunk links

against defining attacks and so on. Alternatively, automated machine learning algorithms can be a constructive solution for the DDoS attack detection and protection in IXP's. It can lead to automate the ¹Blackholing techniques. However, there are some concerns about network security process automation, particularly when it comes to maintaining security deployments. Lack of trust in the technology, fear to change of the way that people doing the works and loss the control on the critical decision making are some example of the concerns around incorporating automation into the cyber security.

Moreover, SDN planning provides a virtual network that tends to transform the current network paradigm into a more flexible and programmable design. I am of the belief that the future networking will stick more on software-based solutions and SDN. On the other hand, SDN gives a complete view of networking elements and services. Additionally, the SDN controller gives the opportunity to reconfigure the SDN and provide seamless migration of virtual machines around the network. Furthermore, SDN gives the chance to dynamically insert a virtual load-balancer or a firewall.

4.3 Future directions

The importance of network automation has been discussed in conclusion sections. Private Peering automation, as one of the pioneer features in TOP-IX was considered as the starting point to automate most of the demanding and time-consuming tasks. The key aspect that limits to test the OpenFlow environment was directly related to the restriction to implement the testbed in a real context. The real context means to have the possibility to work with real OpenFlow supported switches and having hands-on experience into the IXP network. Unfortunately, during my thesis, as TOP-IX could not provide the OpenFlow switches, it was impossible to have a real performance benchmark. So, as a result, the future task can be dedicated to have a real testbed with some OpenFlow switches. It would give the chance to test the OpenFlow traffics and also test the VPLS application in an operational scenario.

¹A network-based reactive defense mechanism to counter with DDoS attack used in IXPs.

Bibliography

- [1] <https://www.netcraft.com/>.
- [2] <https://pynet.twb-tech.com/blog/automation/netmiko.html>.
- [3] <https://www.ansible.com/>.
- [4] <https://napalm-automation.net/>.
- [5] <https://www.peeringdb.com/>.
- [6] <https://www.euro-ix.net>.
- [7] <https://www.pch.net/ixp/dir>.
- [8] <https://www.de-cix.net/>.
- [9] <https://ams-ix.net/>.
- [10] <http://bird.network.cz>.
- [11] <https://www.top-ix.org>.
- [12] <https://www.mix-it.net/index.php?lang=en>.
- [13] <https://apps.db.ripe.net/db-web-ui/#/query?bflag&searchtext=as25309&source=RIPE#resultsSection>.
- [14] <https://onosproject.org/>.
- [15] <https://wiki.onosproject.org/display/ONOS/Virtual+Private+LAN+Service+-+VPLS>.
- [16] <http://mininet.org/>.
- [17] <https://www.cisco.com/c/en/us/support/docs/security-vpn/secure-shell-ssh/4145-ssh.html>.
- [18] <https://media.readthedocs.org/pdf/netmiko/stable/netmiko.pdf>.
- [19] https://www.cisco.com/c/en/us/td/docs/routers/connectedgrid/switch_module_swcg/cgr-esm-configuration/config_vlans.html#99222.
- [20] <http://flask.pocoo.org/>.
- [21] <https://karaf.apache.org/>.
- [22] <https://github.com/mininet/mininet/blob/master/examples/linuxrouter.py/>.
- [23] I. Dolnak. The purpose of creating bgp route servers at internet exchange points. IEEE, 2017.
- [24] Y. Sun et. Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction. page 272. SIGCOMM, 2016.

- [25] A. Dhamdhere et al. The internet is flat: modeling the transition from a transit hierarchy to a peering mesh. Co-NEXT '10 Proceedings of the 6th International Conference Article No. 21, 2010.
- [26] A. Gupta et al. Sdx: a software defined internet exchange. SIGCOMM, 2014.
- [27] B. Ager et al. Anatomy of a large european ixp. pages 163–174. SIGCOMM, 2012.
- [28] D. Kreutz et al. Software-defined networking: A comprehensive survey. pages 14–76. IEEE, 2014.
- [29] D. McPherson et al. Border gateway protocol (bgp) persistent route oscillation condition, rfc3345. IETF, 2002.
- [30] H. Kim et al. Improving network management with software defined networking. pages 114–119. IEEE, 2013.
- [31] HH Liu et al. Automatic life cycle management of network configurations. ACM, 2018.
- [32] J. Jiang et al. Cfa: A practical prediction system for video qoe optimization. page 137. NSDI, 2016.
- [33] L. Jun et al. Internet traffic classification using machine learning. pages 239–243. IEEE Conference, 2007.
- [34] M. A. Beyer et al. The importance of big data: A definition. Gartner, 2012.
- [35] M. Feamster et al. The road to sdn: and intellectual history of programmable networks. ACM SIGCOMM Computer Communication Review, 2014.
- [36] M. Wang et al. Machine learning for networking: Workflow, advances and opportunities. pages 92–99. IEEE, 2017.
- [37] N. Chatsiz et al. On the importance of internet exchange points for today’s internet ecosystem. 2013.
- [38] O. Zajicek et al. Bird internet routing daemon. Proceedings of netdev 0.1, Feb 14-17, 2015, Ottawa, On, Canada, 2015.
- [39] P. Richter et al. Peering at peerings: On the role of ixp route servers. pages 31–44. IMC '14 Proceedings of the 2014 Conference on Internet Measurement Conference, 2014.
- [40] Z. Fadlullah et al. State-of-the-art deep learning: evolving machine intelligence towards tomorrow’s intelligent network traffic control systems. pages 2432–2455. IEEE, 2017.
- [41] Ed et al. Y. Rekhter. A border gateway protocol 4. RFC BGP - RFC 4721 | The most updated BGP Looking Glass database, 2006.

Appendices

.1 VTP server configuration step

Cisco VTP server Configuration function which also will check the probable errors during the VTP server configuration procedure.

```
1     vlan_name = peers[0]["vlan_name"]
2     vtp = peers[0]["vtp"]
3     print('#'*60)
4     print('Connecting to VTP server to add the new vlan')
5     device ["ip"] = vtp
6     iplist = []
7     for l in devices:
8         iplist.append(l["ip"])
9     if vtp in iplist:
10         for i in devices:
11             if i ["ip"] == vtp:
12                 device ['username'] = i ["username"]
13                 device ['password'] = i ["password"]
14                 device ['secret'] = i ["secret"]
15                 device ['device_type'] = "cisco_ios"
16
17         vtp_connection = ConnectHandler(**device)
18         vtp_connection.enable()
19         out2 = vtp_connection.send_command('show vtp status')
20         VTPStatus = CheckVtpServer(out2)
21         if VTPStatus == "Server":
22             vtp_config = ['vlan '+vlan_id,'name '+vlan_name,'no shutdown
23                             ']
24             vtp_output = vtp_connection.send_config_set(vtp_config)
25             print('Vtp server updating... ')
26             vtp_connection.disconnect()
27         else:
28             print("The Selected VTP server is not actually a VTP Server
29                     ,\
30                     It is in VTP Clinet mode...Leaving")
31             exit(200)
32     else:
33         print("There is No Such IP Adress in DATABASE As the VTP Server
34                 ...leaving")
35         exit(166)
```

.2 Custom topology creation script in Mininet

The Python class that is responsible for custom OpenFlow topology creation in Mininet.

```
1
2 class NetworkTopo( Topo ):
3     "Private Peering Topology"
4     # def addSwitch(self, name, **opts ):
5     #     kwargs = { 'protocols' : 'OpenFlow13' }
6     #     kwargs.update( opts )
7     #     return super(NetworkTopo, self).addSwitch( name, **kwargs
8         )
9
10    def __init__( self ):
11        "Create an empty network and add nodes to it."
12        Topo.__init__( self )
13        #
14        info( '*** Adding switch\n' )
15        switch1 = self.addSwitch( 's1' )
16        switch2 = self.addSwitch( 's2' )
17        switch3 = self.addSwitch( 's3' )
18
19        info( '*** Adding Routers\n' )
20        router1 = self.addNode( 'r1', cls=LinuxRouter, ip
21            ='194.116.98.1/30',vlan=77 )
22        router2 = self.addNode( 'r2', cls=LinuxRouter, ip
23            ='194.116.98.2/30',vlan=77 )
24
25
26        info( '*** Creating links\n' )
27        self.addLink( switch1, switch2 )
28        self.addLink( switch2, switch3 )
29        self.addLink( switch1, router1,intfName2='r1-eth1' )
30        self.addLink( switch3, router2,intfName2='r2-eth1' )
31
32    topos = { 'test': ( lambda: NetworkTopo() ) }
33    def run():
34        topo = NetworkTopo()
35        net = Mininet(topo=topo,controller=RemoteController)
36        # c1 = net.addController('c1', controller=RemoteController,
37            ip="172.17.0.1")
38        net.start()
39        CLI( net )
40        net.stop()
41
42    if __name__ == '__main__':
43        setLogLevel( 'info' )
44        run()
```