



**University of Mohamed Khider Biskra**  
Faculty of Sciences and Technology  
Department of Electrical Engineering

## **MASTER'S DEGREE**

Sciences and Technology  
Field : Telecommunication  
Option : Network and Telecommunication

Ref. : .....

---

Presented and submitted by:  
**BOUMEZRAG Abdelhak**

On : 30 septembre 2020

### ***A Proposed of Novel Network Management Platform for Network Automation and Programmability with Implementation on GNS3***

---

#### **Jury :**

M.	DHIABI Fathi	MCB	University of Biskra	President
M.	BEKHOUCHE Khaled	MCA	University of Biskra	Examiner
M.	AMEID Sofiane	MAA	University of Biskra	Supervisor

Academic Year : 2019 - 2020



**University of Mohamed Khider Biskra**  
Faculty of Sciences and Technology  
Department of Electrical Engineering

## **MASTER'S DEGREE**

Sciences et Technologies  
Field : Telecommunication  
Option : Network and Telecommunication

### **Theme:**

***A Proposed of Novel Network Management  
Platform for Network Automation and  
Programmability with Implementation on  
GNS3***

**Presented by:**  
BOUMEZRAG Abdelhak

**Favorable opinion of the supervisor:**  
Mr. AMEID Sofiane

**Favorable opinion of the Jury President:**  
Mr. DHIABI Fathi

**Stamp and signature**



University of Mohamed Khider Biskra  
Faculty of Sciences and Technology  
Department of Electrical Engineering

## MASTER'S DEGREE

Sciences et Technologies  
Field : Telecommunication  
Option : Network and Telecommunication

### Theme:

# *A Proposed of Novel Network Management Platform for Network Automation and Programmability with Implementation on GNS3*

### Abstract (English and Arabic)

Network automation and programmability is a set of exciting technologies and approaches, that enables innovation in how we design and manage networks to avoid many problems as the operation time and effort. Hence, in this work, we proposed a modest workflow application named the Network Management Platform (NMP) based on Python packages as Netmiko, and SSH as the transport protocol, that is able to manage and operate the network without manual intervention in form of stages to meet the desired states in a given network, including the CLI based network devices. These methods are representing the future of networks, allowing the management of an increased number and variety of devices in a unitary and centrally way. these Platform stages were regularly tested in a campus network topology on GNS3.

**Keyword:** Network automation, Network programmability, SDN, Netmiko, Ansible, NETCONF, Python, GNS3.

أتمتة الشبكات وإمكانية البرمجة فيها، هي مجموعة من التقنيات والأساليب المثيرة، والتي تتيح الابتكار في كيفية تصميم وإدارة الشبكات لتجنب العديد من المشكلات مثل وقت التشغيل والجهد. وبالتالي، في هذا العمل، اقترحنا تطبيقاً متواضعاً لسير عمل يسمى منصة إدارة الشبكة (NMP) بالاعتماد على حزم Python مثل Netmiko و SSH كبروتوكول النقل، القادر على الإدارة والتشغيل دون تدخل يدوي، في شكل مراحل لتلبية الحالات المطلوبة في شبكة معينة، بما في ذلك أجهزة الشبكة القائمة على CLI. مثل هذه الأساليب تمثل مستقبل الشبكات، مما يسمح بإدارة عدد متزايد من الأجهزة بطريقة موحدة ومركزية. تم اختبار مراحل النظام الأساسي هذه بانتظام في شبكة طوبولوجيا مقترحة على الـ GNS3.

**كلمات مفتاحية:** أتمتة الشبكة، إمكانية البرمجة في الشبكة، SDN، Netmiko، Ansible، NETCONF، بايثون، GNS3.

**Dedication**

I dedicate this humble work to my great father and mother, my brothers, my whole family, my friends, and all who know and support me in this academic track from the closest person to the farthest.

The reader as well.

### Acknowledgments

First and foremost, praises and thanks to Allah, the Almighty, for His showers of blessings throughout my research work to complete this thesis successfully.

I would like to express my deep and sincere gratitude to my thesis supervisor Mr. AMEID Sofiane for giving me the opportunity to do this work and providing invaluable guidance throughout this research. His vision, sincerity, and motivation have deeply inspired me. It was a great privilege and honor to work and study under his guidance. I am extremely grateful for what he has offered me. I would also like to thank him for his friendship, empathy, and a great sense of humor.

Besides my supervisor, I would like to thank my committee members who were more than generous with their expertise and precious time. A special thanks to Mr. DHIABI Fethi, my committee president for his hours of reflecting, reading, patience throughout the entire process, as well as Mr. BAKHOUCHE Khalid for agreeing to serve on my committee as an examiner.

I am extremely grateful to my parents for their love, understanding, prayers, caring and sacrifices for educating and preparing me for my future, also I express my thanks to my brothers and sisters, for their support and empathy.

I would like to also thank my great friends, Imad Eddine and Moneim for their wonderful friendship and educational helps, Sofiane Bekkari also, despite the distance, the love of learning did not prevent us from studying new things together, my great friends also from the high school days till now, Abdelmomen, Islam, Zine Edine and Mosaab, I wish them all the best luck in their life.

## Table of Content

Abstract.....	ii
Dedication .....	iii
Acknowledgments.....	iv
List of Figures.....	viii
List of Tables .....	xi
List of Abbreviations .....	xii
General Introduction.....	1

---

### Chapter I: Network Automation And Programmability Overview

---

I.1 Introduction.....	5
I.2 Network Automation and Programmability Concept.....	5
I.3 Software Defining Networking.....	6
I.3.1 SDN Architecture.....	8
I.3.2 OpenFlow.....	10
I.4 Approaches and Use Cases.....	13
I.4.1 Controller-Based Network.....	14
I.4.1.1 Cisco DNA Center.....	14
I.4.2 Network Automation Frameworks.....	17
I.4.2.1 Ansible.....	18
I.4.3 Device APIs and Management Interfaces.....	22
I.4.3.1 REST API.....	23
I.4.3.2 NETCONF Protocol.....	27
I.4.3.3 CLI.....	29
I.5 Common Network Automation Tasks .....	30
I.6 Conclusion .....	32

---

## Chapter II: Management Interfaces Approach And The Proposed Workflow For Developing Automation Platform

---

<b>II.1</b>	<b>Introduction .....</b>	<b>34</b>
<b>II.2</b>	<b>PART A .....</b>	<b>34</b>
II.2.1	Python for Network Automation.....	34
II.2.1.1	Why python? .....	35
II.2.2	CLI Based Interaction Libraries .....	36
II.2.2.1	Netmiko .....	37
II.2.2.2	TextFSM Integration .....	42
II.2.3	Multithreading.....	43
<b>II.3</b>	<b>PART B .....</b>	<b>46</b>
II.3.1	The Proposed Network Management Platform.....	46
<b>II.4</b>	<b>Conclusion.....</b>	<b>50</b>

---

## Chapter III: Network Management Platform For Network Automation

---

<b>III.1</b>	<b>Introduction .....</b>	<b>52</b>
<b>III.2</b>	<b>Technical Requirements .....</b>	<b>52</b>
<b>III.3</b>	<b>Development Methodology .....</b>	<b>52</b>
<b>III.4</b>	<b>NMP Auxiliary Functions Development .....</b>	<b>53</b>
III.4.1	Performing SSH Connection .....	54
III.4.2	Getting Devices Version .....	55
III.4.3	Getting Devices Information as Inputs .....	56
<b>III.5</b>	<b>NMP Stages Development.....</b>	<b>57</b>
III.5.1	Network Configuration .....	57
III.5.2	Network Verification and Test Connection.....	59
III.5.3	Network Confirmation and Backups .....	62
III.5.4	Check Network and Report Generation .....	63

---

## Table of Content

---

III.5.5	Multithreading Integration .....	65
<b>III.6</b>	<b>Use Case Testing with NMP on GNS3.....</b>	<b>66</b>
III.6.1	Environment Setup.....	66
III.6.2	Network Topology Setup .....	67
III.6.3	Use Case Testing and Results.....	71
<b>III.7</b>	<b>Discussion .....</b>	<b>82</b>
III.7.1	Benefits, Drawbacks and limitations .....	83
III.7.2	Future works.....	83
<b>III.8</b>	<b>Conclusion .....</b>	<b>84</b>
	<b>General Conclusion .....</b>	<b>86</b>
	<b>References .....</b>	<b>88</b>



## List of Figures

<b>Figure I. 1:</b> Traditional network devices state and components [10].	7
<b>Figure I. 2:</b> Software defined network architecture from ONF [16].	9
<b>Figure I. 3:</b> Design of an OpenFlow switch and the communication with the controller [6].	11
<b>Figure I. 4:</b> Summarized plane for common approaches and use cases.	13
<b>Figure I. 5:</b> Cisco DNA Center with Northbound and Southbound Interfaces [10].	15
<b>Figure I. 6:</b> Cisco DNA Center dashboard [24].	16
<b>Figure I. 7:</b> Ansible workflow with network devices [28].	19
<b>Figure I. 8:</b> Example of an Ansible inventory file [30].	20
<b>Figure I. 9:</b> Example on an Ansible Playbook file [30].	21
<b>Figure I. 10:</b> Example of configuration file template using Jinja2 in the left with the associated values in a YAML file in the right [10].	22
<b>Figure I. 11:</b> Illustrated plan for devices APIs with their associated Python modules.	23
<b>Figure I. 12:</b> Client and server (web server) communication using REST API [35].	24
<b>Figure I. 13:</b> HTTP Verb and URI in an HTTP Request Header [10].	25
<b>Figure I. 14:</b> Format of the HTTP request and response between the client and the network device (server) [35].	25
<b>Figure I. 15:</b> HTTP Verbs in the context of network devices [35].	26
<b>Figure I. 16:</b> NETCONF protocol layer and content [40].	28
<b>Figure I. 17:</b> NETCONF communication with the initial capabilities exchange [41].	28
<b>Figure II. 1:</b> Check all the Netmiko sub-modules and classes on the Python interrupter.	40
<b>Figure II. 2:</b> Source code example for establishing SSH connection to Cisco IOS switch.	41
<b>Figure II. 3:</b> Applying TextFSM templates on show command inside netmiko script [53].	43
<b>Figure II. 4:</b> The process of scheduling a thread in the processor once we run a python file [46].	44
<b>Figure II. 5:</b> The result of the use case with serial query [48].	45
<b>Figure II. 6:</b> The output of the use case using parallel query [48].	46
<b>Figure II. 7:</b> The NMP prototype workflow.	47
<b>Figure II. 8:</b> Summarizing the management platform stages.	49
<b>Figure II. 9:</b> The Proposed Network Management Platform main menu.	50

## List of Figures

---

<b>Figure III. 1:</b> Example of developing CSV Reporting function using agile methodology. ....	53
<b>Figure III. 2:</b> The source code of the SSH connection function. ....	54
<b>Figure III. 3:</b> CSV file contains needed devices information. ....	55
<b>Figure III. 4:</b> The source code of the check version function. ....	56
<b>Figure III. 5:</b> The source code of device input function. ....	57
<b>Figure III. 6:</b> The source code of the configuration function. ....	58
<b>Figure III. 7:</b> Network configuration stage flowchart. ....	59
<b>Figure III. 8:</b> Network verification and testing connection stage flowchart. ....	60
<b>Figure III. 9:</b> The source code of test connection function. ....	61
<b>Figure III. 10:</b> The source code of confirmation and backups functions. ....	62
<b>Figure III. 11:</b> Network confirmation and backup Stage flowchart. ....	63
<b>Figure III. 12:</b> Checking network and report generation stage flowchart. ....	64
<b>Figure III. 13:</b> The configuration function integrated with threading functionality. ....	65
<b>Figure III. 14:</b> The network automation appliance in GNS3 marketplace. ....	66
<b>Figure III. 15:</b> Docker container connected to the NAT node on the workspace in GNS3 ....	67
<b>Figure III. 16:</b> Configuring the Docker network interface using Nano editor. ....	68
<b>Figure III. 17:</b> Docker obtained an IP address from DHCP. Confirm using Ifconfig. ....	68
<b>Figure III. 18:</b> The experimented campus network on GNS3. ....	69
<b>Figure III. 19:</b> Example of verifying remote access on S2, using SSH protocol. ....	71
<b>Figure III. 20:</b> Needed files to execute the NMP in local laptop ....	72
<b>Figure III. 21:</b> Sub-menus of the NMP stages shown in the terminal. ....	73
<b>Figure III. 22:</b> Configuration stage, execution progress, with and without multithreading. ....	74
<b>Figure III. 23:</b> Execution progress of Verify all devices option. ....	75
<b>Figure III. 24:</b> Manually entering Source and destination IPs. ....	76
<b>Figure III. 25:</b> Testing test connection process and the output result. ....	76
<b>Figure III. 26:</b> Entering device information manually for specific device interaction. ....	77
<b>Figure III. 27:</b> Testing the confirmation option and the terminal output. ....	78
<b>Figure III. 28:</b> Testing backups option and the output in the terminal and in local directory. ....	78
<b>Figure III. 29:</b> Check interfaces option testing and the output. ....	79
<b>Figure III. 30:</b> Check routing option testing for OSPF protocol and the output. ....	80
<b>Figure III. 31:</b> Check VLAN testing and output. ....	80
<b>Figure III. 32:</b> CSV report output in local laptop and in the terminal. ....	81

**List of Figures**

---

**Figure III. 33:** Global report spreadsheet opened with Excel . .....81

**Figure III. 34:** Syslog message indicates the connection between R1 and the admin. ....82

**Figure III. 35:** The output of show ip interface brief after executing the configuration stage. ....82

## List of Tables

**Table I. 1:** Main components of a flow entry in OpenFlow v1.5 [20]. .....11

**Table I. 2:** Four main network devices module [31]. .....20

**Table I. 3:** Comparing CRUD Actions to REST Verbs [10]. .....25

**Table I. 4:** Some generic headers that provides details about the meta-data [36]. .....26

**Table II. 1:** Open source python libraries for network automation [46]. .....36

**Table II. 2:** Netmiko supported devices under three categories [49]. .....38

**Table II. 3:** Netmiko commonly-used methods [49]. .....41

**Table III. 1:** Configuration files content on the test case. ....72

## List of Abbreviations

<p><b>SDN:</b> Software-Defined Networking.</p> <p><b>IoT:</b> Internet of Thing.</p> <p><b>AI:</b> Artificial Intelligence.</p> <p><b>RIP:</b> Routing Information Base.</p> <p><b>OSPF:</b> Open Shortest Path First.</p> <p><b>FIB:</b> Forwarding Information Base.</p> <p><b>SSH:</b> Secure Shell.</p> <p><b>CLI:</b> Command Line Interface.</p> <p><b>API:</b> Application Programming Interface.</p> <p><b>ONF:</b> Open Networking Foundation.</p> <p><b>REST:</b> REpresentational State Transfer.</p> <p><b>ODL:</b> OpenDaylight.</p> <p><b>YANG:</b> Yet Another Network Generation.</p> <p><b>NBI:</b> Northbound Interfaces.</p> <p><b>SBI:</b> Southbound Interfaces.</p> <p><b>NETCONF:</b> NETwork CONFiguration.</p> <p><b>DNA Center:</b> Digital Network Architecture Center.</p> <p><b>ACI:</b> Application Centric Infrastructure.</p> <p><b>QoS:</b> Quality of Service.</p> <p><b>IBN:</b> Intent Based Networking.</p> <p><b>SNMP:</b> Simple Network Management Protocol.</p> <p><b>RESTCONF:</b> REpresentational State Transfer CONFiguration.</p> <p><b>GUI:</b> Graphical User Interface.</p> <p><b>IT:</b> Information Technology.</p> <p><b>SGT:</b> Scalable Group Tag.</p> <p><b>ML:</b> Machine Learning.</p> <p><b>SDK:</b> Software Developer Kit.</p> <p><b>Syslog:</b> System Logging Protocol</p>	<p><b>CMS:</b> Configuration Management System.</p> <p><b>DSL:</b> Domain Specific Language.</p> <p><b>HTTP:</b> HyperText Transfer Protocol.</p> <p><b>IP:</b> Internet Protocol.</p> <p><b>IOS:</b> Internetwork Operating System.</p> <p><b>VLAN:</b> Virtual Local Area Network.</p> <p><b>YAML:</b> Yet Another Markup Language.</p> <p><b>JSON:</b> Java Script Object Notation.</p> <p><b>XML:</b> eXtensible Markup Language.</p> <p><b>UI:</b> User Interface.</p> <p><b>HTML:</b> HyperText Markup Language.</p> <p><b>CRUD:</b> Create, Read, Update, Delete.</p> <p><b>URI:</b> Uniform Resource Identifier.</p> <p><b>TCP:</b> Transmission Control Protocol.</p> <p><b>IETF:</b> Internet Engineering Task Force.</p> <p><b>RFC:</b> Request for Comments.</p> <p><b>RPC:</b> Remote procedure Call.</p> <p><b>SOAP:</b> Simple Object Access Protocol.</p> <p><b>VTY:</b> Virtual Terminal Line.</p> <p><b>MIB:</b> Management Information Base.</p> <p><b>CSV:</b> Comma Separated Value.</p> <p><b>BGP:</b> Border Gateway Protocol.</p> <p><b>EOF:</b> End of File.</p> <p><b>NMP:</b> Network Management Platform.</p> <p><b>VTP:</b> Virtual Trunking Protocol.</p> <p><b>GNS3:</b> Graphical Network Simulator-3.</p> <p><b>OSPF:</b> Open Shortest Path Protocol.</p> <p><b>VIRL:</b> Virtual Internet Routing Lab.</p> <p><b>DHCP:</b> Dynamic Host Configuration Protocol.</p> <p><b>NAT:</b> Network Address Translation.</p> <p><b>NTP:</b> Network Time Protocol.</p>
---	--

# **GENERAL INTRODUCTION**

# General Introduction

In recent years, networking has been the main influence and factor of development within large and small companies and organizations, as they guarantee high amounts of information and resources being transferred between users and servers for one reason or another, passing through many routers, switches, and middleboxes that are differ in their configuration, services, functionalities and heterogeneous nature. All the installed network appliances need to be regularly maintained and configured to make sure the network working properly and does not interrupt the company's business processes, however, the inevitable business augmentation attempts and the booming need for immediate network services necessarily lead to the networks scaling infrastructure, which means an increase in the number of network devices, branches, and services , and this what can pretty much give rise to the emergence of many network problems and challenges.

The classical network operating and equipping methods, will not be effective anymore, especially, when we are faced with the facts of network infrastructure scaling, vendor dependent command line syntax and structures, and the large number of devices CLI using only manual configurations, which is follow a standard operating procedure to achieve a set of administration tasks. This manual interaction represented in one command at a time entering are highly time consuming and prone to errors, in addition, it is typically daily repetitive chores and tedious, which may drain out technical ability, with the difficulty of understanding complex networking scenarios in other phase, that is vary across network elements types and vendors by less technical team members.

Over the last decade, the network industry and researchers take the challenge to solve these problems by introducing many solutions and paradigms, trying to exclude these difficulties. Software Defining Networking or SDN is one of the most popular concept, that tries to eliminate the network devices vendor dependency, also, aims to abstract and hide every low-level interaction with network elements by controlling and exercising automation directly on the data path via standard protocols and APIs, like OpenFlow, however, the traditional network that does not support the purest SDN model need to keep the pace and respond to the dynamic network changes, which led to the emergence of other trends and approaches under the term network automation and

programmability, which can be implemented on all kinds of network devices and topologies using different methods and effecting on different parts of the network devices itself, in most cases focusing on the configuration and management plane.

Python is the favored programming language in network engineering, for that we have created a simple campus network topology in GNS3, having as main component the Network Automation Docker Container Ubuntu image, with the character of a network controlling element. We have controlled the network devices in a programmatic way using Netmiko as the primary open source package, for designing and developing a network automation application.

### **A- Research Methodology and thesis outlines**

This work began with an in-depth investigation on the subject, based on a group of resources as professional and academic books, various scientific articles, several explanations and courses on different websites and platforms, and the official documentation of many technologies. After obtaining the general ideas for the research path, we were have divided the work as we can see in these outlines the coming thesis outlines.

**The first chapter** presents a theoretical reviews and informational background of what does exists in network industry as foundations and principals on network automation and programmability , trying to rich the topic by many terminology and definitions, including SDN and his affect, and giving many approaches with their associated use cases as example, which can be implemented as a solution for all kinds of network devices that is basically built on programming in mind (APIs ) or with the intended management interface (CLIs).

**The second chapter** focus on the third approach, where python programming language with its packages and modules are the primary tools for interacting with the management interfaces (devices CLIs). Netmiko and his functionalities are the most popular module for doing this, which gives as solutions for building our complete Network Management Platform (NMP) application. Other modules that we integrated as a resolution for side effects and problems like the execution time and unstructured data, which are threading and TextFSM modules that are explained as well.

**The third chapter** gives a complete demonstration for automating common network management tasks using the pre-explained and designed workflow of our Network Management Platform application (NMP), on a campus network topology emulated in GNS3 with Cisco VIRL



switches and routers IOS images.

### **B- Challenges and Thesis Objectives**

The key challenges are related to the variety of network devices configuration, and the large possible number of states in one element or a specific service configuration, it is inflexible when each time expect to get a specific output state, and we were previously incited the Python program to extract the useful information using a parsing process from a different state, which can break the whole code. In addition, some commands line was not intended to be in the running configuration files which can cause some errors in the verification methods. Also, another challenge is when we trying to make our platform reusable as much as possible in several placements.

The main objective of this effort is to reduce or avoid the mentioned problems in the traditional network, in addition, spurring network administrators and researchers to develop and innovate new applications that help in the daily work of configuring and operating on their network infrastructure, whatever the network topology and devices nature included using Python in network context, in this project we will demonstrate how we can do this, by creating an automation tool based on Netmiko and other helper libraries, which is a simple Network Management Platform user interface-based. The following other objectives are targeted in this research:

- Identify background information and approaches that already exist to apply and exercise network automation and programmability and recognize the differences in term of implementation.
- Identify a multivendor and open source solutions to automate the repetitive configuration and management tasks using Python to enhance the way we interact with network devices CLI.
- Identify and exercise an open source virtual environment to practice and iterate codes and scripts before the implementation in the real live network.
- Validate the automation process by creating and exercising the developed application and the workflow on a campus network use case with only Cisco devices in GNS3.
- Discuss the differences in terms of effort, time, and errors prone between the classical situation and the programmable culture, then declaring the benefits, drawbacks, limitation, and future works.

**CHAPTER I:**  
**NETWORK AUTOMATION AND**  
**PROGRAMMABILITY OVERVIEW**

### I.1 Introduction

Network programmability and automation have the main goal of simplifying the tasks involved in configuring, managing, and operating network equipment. Under these terms, there are many automation technologies and approaches that are aimed to effects on different parts of the network and comes as solutions to the mentioned problems or else, we would like to get a terminologies overview and place the very common from them as an informational background.

This chapter is divided into four major section, first section introduces the concept of the network automation and programmability while section two contains a general view of Software-Defined Networking paradigm; definitions and architecture, also his impact on network industry, the third section will dive more by given different mechanism and approaches to automate and manage the network, each with its specifications and utilization, in total, we mentioned three approaches with their common use cases tools, in additions, the last major section, gives common network administration tasks where automation make sense.

### I.2 Network Automation and Programmability Concept

Network automation, like most types of automation, is thought of as a means of doing things faster, which mean not manually, in the networking context, it is the process of automating the configuration, management, testing, deploying, and operating of physical and software-based devices within a network. The tasks that were normally done by the network or system administrator can be automated using a number of tools and technologies [1] [2]. Any type of network can use network automation including data centers, service providers, and campuses, to improve efficiency, avoiding repetitive tasks, saves time, reduce human error, and lower operating expenses.

Network programmability can have different meanings, depending on perspective. To a network engineer, programmability means interacting with a device or group of devices (driving configurations, troubleshooting, etc.) with a software that sits logically above the device [3].

Like the goal and the methods, network automation uses programming logic to manage network resources and services [4] [5], the network programmability toolset is the foundation for advanced next-generation network automation with adding prebuilt intelligence that can assist with

network deployments, operations, or troubleshooting, Network programmability makes automation simpler and more accessible through standard tools. Scripting languages are widely used by network and system administrators for automating tasks. Among the automation tools, Python and Ansible are the most popular, with Software Defined Networking (SDN) in picture [1].

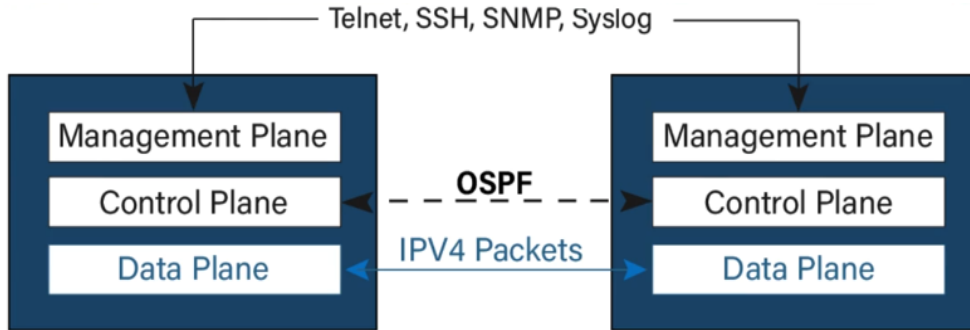
### I.3 Software Defining Networking

We cannot move forward in this thesis without talking about Software-Defined Networking or SDN, while the term programmable is used to generalize the concept of the simplified network management and reconfiguration [6], and since it has inspired and still inspire large companies and researchers to obtain innovations and technologies in the topic of automation and programmability. [7] [8].

The challenge behind the SDN technology is the attempts to find solutions to avoid a lot of problems that exist in the traditional networking architectures which we cited in the introduction, in addition, the significant limitations that must be overcome to meet modern IT requirements (big data, IoT, AI ...), the network must scale to accommodate increased workloads and traffic growth with greater agility, while also keeping costs at a minimum [9]. In a traditional network environment, each network device has a local control plane, a local data plane, and a local management plane, now the control plane is essentially the intelligence of the device, for example each network device has its own local routing table or a local RIP (routing information base) which is selected based on distributed and complex algorithms like OSPF. Each network device has also data plane, which mean how packet are forwarded through it, for example, it uses a FIB or forwarding information base that is derived from the RIB and from adjacency tables, so that the packet can be rewritten with the correct encapsulation to the destination.

The management plane includes protocols that allow network engineers to manage devices. Telnet and Secure Shell (SSH) are two of the most obvious management plane protocols, if we have multiple network devices from management point of view using the CLI or command line interface, we would have to connect with each of them individually and typically using manually configuration [10]. Advocates of SDN said that this is a complex way, because there is no single

hardware that has visibility of the entire network, each network device has to work out independently, Figure I.1, illustrate this state and these three components.



**Figure I. 1:** Traditional network devices state and components [10].

The intellectual history document [11], divided the programmable Networks over years into three stages:

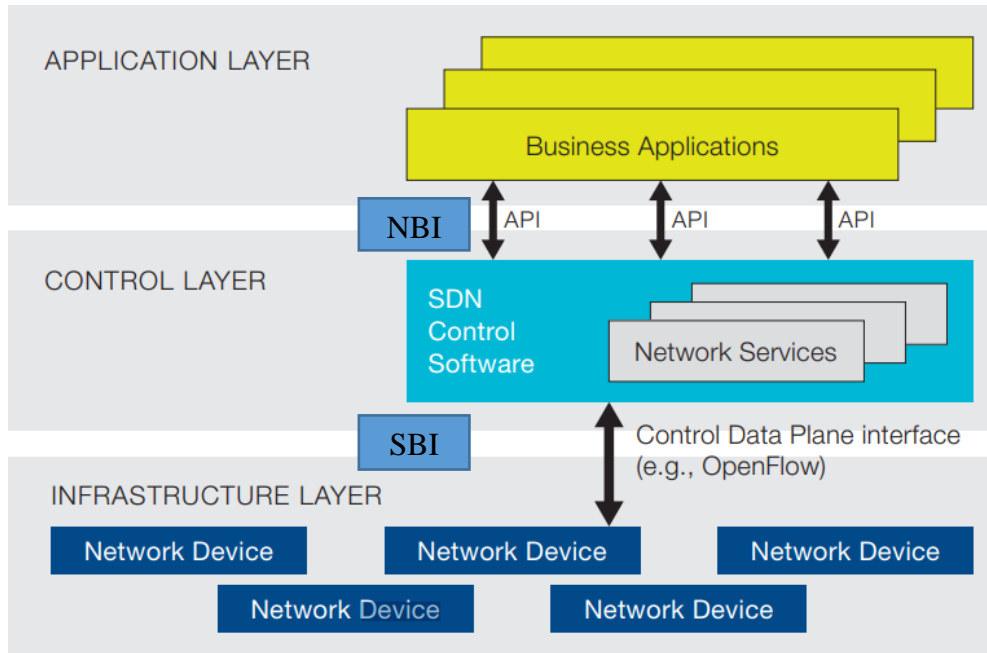
- a- Active networks (1900-2000):** which introduced the programmable functions in the network.
- b- The separation of control and data plane (2001 - 2007):** which allowed developing interfaces between control and data plane.
- c- The Openflow API and network operating systems (2007-2010):** described the first instances of the development of an open interface and the practical ways to separate control and data plane which introduced the stateful SDN model.

SDN is defined by the physical separation of the control and data plane of the networking devices [12] [13], it is changing the way we design and manage networks. SDN has two defining characteristics. First, an SDN separates the control plane (decides how to handle the traffic) from the data plane (forwards traffic according to decisions that the control plane makes). Second, an SDN consolidates the control plane, so that a single software control program controls multiple data-plane elements. The SDN control plane exercises direct control over the state in the network's data-plane elements (i.e., Routers, switches, and other middle boxes) via a well-defined Application Programming Interface (API) [11], that allows the entire network to be programmable and automated.

### I.3.1 SDN Architecture

The Open Networking Foundation (ONF) which is the first organization in charge of the SDN and Openflow protocol standards, proposed the architecture depicted on Figure I.2, It provides a logical view of the SDN architecture. Network intelligence is logically centralized in software-based SDN controllers, which maintain a global view of the network. The figure presents different components in the SDN architecture and they are as follows [14]:

- **Applications Layer:** Programs that communicate behaviors, policies, algorithms and needed resources with the SDN controller via APIs. These applications could include networking routing, network management, or business policies used to run large data centers.
- **Northbound interfaces NBI:** The connection between the controller and applications via applications programming interfaces, commonly uses REST API.
- **Controller layer:** The SDN Controller is a logical entity that receives instructions or requirements from the SDN Application layer and relays them to the networking devices. The controller also extracts information about the network from the hardware devices and communicates back to the SDN Applications with an abstracted view of the network, including statistics and events about what is happening, the Open Daylight controller as an example.
- **Southbound interfaces SBI:** The connection between the controller and infrastructure layer via application programming interfaces, Openflow protocol is the first API appeared in the SDN industry.
- **Infrastructure layer:** The SDN networking devices control the forwarding and data processing capabilities for the network. This includes forwarding and processing the data path.
- **East West interface:** In the case of a multi-controller-based architecture, the East West interface protocol manages interactions between the various controllers [15].



**Figure I. 2:** Software defined network architecture from ONF [16].

There are many types of controllers in the industry market, some of which are open source, and others are vendor-specific, each unit has specific environmental characteristics and required interfacing to operate on the live network, we cite as an example, NOX and POX controllers which are the first and the original OpenFlow based controllers from Stanford university, written in C++ and Python, it can be considered as general, open source for rapid development and prototyping of network applications [17]. Also, ODL or OpenDaylight which is a very common SDN controller project written in Java and it is YANG based data models, lead and maintained by the collaborative Linux Foundation, which is a community that has come together to fill the need for an open and reference framework for programmability and control solution through an open source SDN controller [17]. The ODL uses a large variety of options to communicate with the under laying network not just OpenFlow, but including many SBIs like NETCONF protocol. There are also vendor-specific products like Cisco DNA Center, ACI, VMware NSX, and Juniper Contrail ... etc.

As we said before, the first appearing of SDN was related to the OpenFlow protocol. However, it should be made clear that OpenFlow is just one (rather popular) out of many possible implementations of controller-switch interactions [6].

### I.3.2 OpenFlow

According to [15], OpenFlow is the protocol used for managing the southbound interface of the generalized SDN architecture. It is the first standard interface defined to facilitate interaction between the control and data planes of the SDN architecture. OpenFlow provides software-based access to the flow tables that instruct switches and routers how to direct network traffic. Using these flow tables, administrators can quickly change network layout and traffic flow. In addition, the OpenFlow protocol provides a basic set of management tools which can be used to control features such as topology changes and packet filtering.

The supervisor of this project Nick McKeown in his whitepaper [18], gave great details about the OpenFlow Switch specification which is a feature that can help to extend programmability and independent from network device vendor propriety specification. An OpenFlow Switch consists of at least three parts:

- **One or many Flow Tables:** includes actions associated with each flow entry, to tell the switch how to process the flow,
- **A Secure Channel:** that connects the switch to a remote control process (called the controller), allowing commands and packets to be sent between a controller and the switch
- **The OpenFlow Protocol:** which provides an open and standard way for a controller to communicate with a switch.

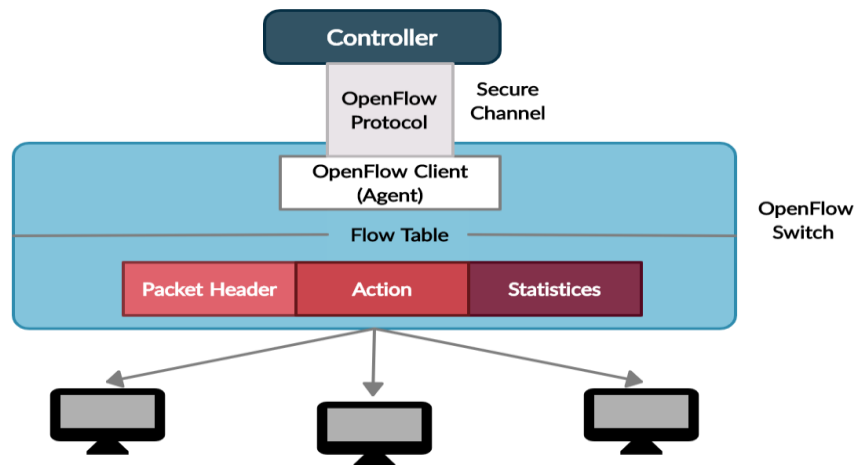
He mentions also [18], that by specifying a standard interface (the OpenFlow Protocol) through which entries in the Flow Table can be defined externally, the OpenFlow Switch avoids the need for researchers to program the switch. It is useful to categorize switches into dedicated OpenFlow switches that do not support normal Layer 2 and Layer 3 processing, and OpenFlow-enabled general purpose commercial Ethernet switches and routers, to which the OpenFlow Protocol and interfaces have been added as a new feature.

These OpenFlow switches placed in the infrastructure layer controlled by a software program on top of it installed on the hardware controller, specifically, it is responsible for populating and manipulating the flow tables. By insertion, modification and removal of flow entries, the controller can modify the behavior of the switches with regard to forwarding. The OpenFlow specification defines the protocol that enables the controller to instruct the switches. To that end, the controller uses the secure control channel [19].



The flow table itself is responsible for maintaining the information required by the switch in order to forward packets [6]. As Figure I.3 depicted, an entry in the Flow table has three fields:

- A packet header that defines the flow.
- The action, which defines how the packets should be processed.
- Statistics, which keep track of the number of packets and bytes for each flow, and the time since the last packet matched the flow (to help with the removal of inactive flows).



**Figure I. 3:** Design of an OpenFlow switch and the communication with the controller [6].

There are many versions for the OpenFlow protocol that is aim to extend capabilities and fix issues evolved during ONF’s standardization process [20], from version 1.0 where there are only 12 fixed match fields and a single flow table to the latest version (1.5) that features multiple tables, over 41 matching fields and a bunch of new functions, the structure of the flow entry in OpenFlow 1.5 (last version) is shown in Table I.1.

**Table I. 1:** Main components of a flow entry in OpenFlow v1.5 [20].

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flag
--------------	----------	----------	--------------	----------	--------	------

Each flow entry contains:

- **Match fields:** match against packet.
- **Priority:** matching precedence of the flow entries.
- **Instructions:** set of instructions that are executed when a packet matches the entry.

- **Timeouts:** maximum amount of idle time.
- **Cookie:** opaque data value chosen by the controller.
- **Flags:** alter the way flow entries are managed.

When a packet arrives from a switch port, it is compared with the match fields in the flow entries. If the packet is matched, it will be processed as indicated in the instructions [20].

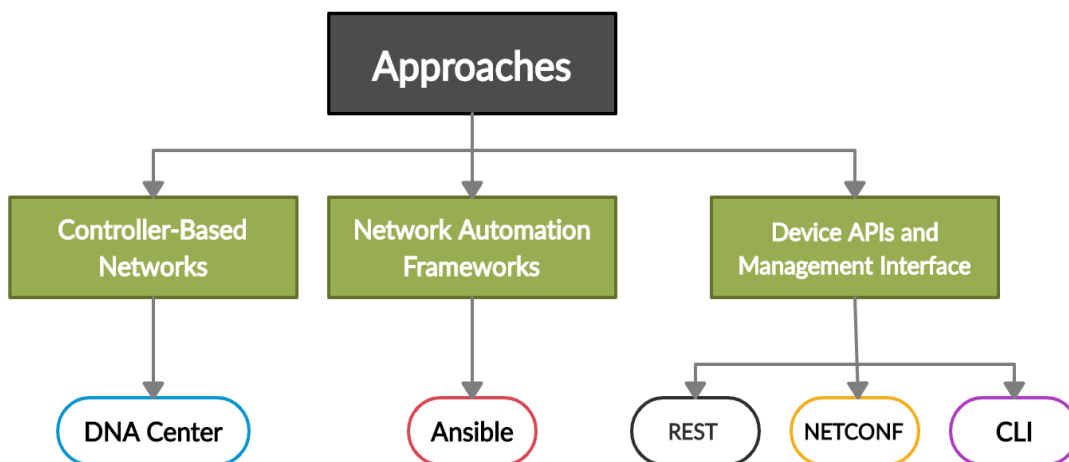
As a review, the SDN promises to dramatically simplify network control, management, and enable innovation through network programmability, the migration of control plane, which used to be tightly integrated in the networking devices (for example, Ethernet switches) into accessible and logically centralized controllers, enables the underlying networking infrastructure to be abstracted from the application's point of view. This separation provides a more flexible, programmable, cost efficient, and innovative network architecture. Besides the network abstraction, the SDN architecture will provide a set of Application Programming Interfaces (APIs) that simplifies the implementation of common network services (for example, routing, multicast, security, QoS, and various forms of policy management) [17]. One common southbound interface implementations of the SDN is called OpenFlow. The separation of the forwarding hardware from the control logic allows easier deployment of new protocols and applications, straightforward network visualization and management, and consolidation of various middle boxes into software control. With SDN, enterprises and carriers gain vendor-independent control over the entire network from a single logical point, which greatly simplifies the network design and operation. SDN also greatly simplifies the network devices themselves, since they no longer need to understand and process thousands of protocol standards but merely accept instructions from the SDN controllers [16].

These benefits and advantages suggested by the SDN architecture, make leaders in networking industry rethink about the current state of the network, however, the purest definition of SDN model with the OpenFlow protocol might have limitations and drawbacks that must be taken into account, including, the scalability performance issue, and the central point of failure which make the ability of hacking the whole network system from a central point, also technology change which require a lot of effort like training and buying new software and clean out all the setting and policies because of the presence of the traditional devices and systems in the network that does not support this paradigm [21].

### I.4 Approaches and Use Cases.

When we are coming to implement and research about network automation and programmability, it may encounter many network industry trends and technologies that emerged over the past 10 years, many of them are considered as SDN and can “get put under the *SDN umbrella*”, and we note that is not necessarily decoupling the control from the data plane, as well as, controller-based networks, APIs on network devices, network automation tools (refers to the automation frameworks), and the list goes on [22], since they shared the same global idea of running software on top of the underlying infrastructure network to automate boring and repetitive tasks, centralize and simplify the network management, abstract and hide the low level interactions with network devices. The concept behind these trends aims to create hybrid or newer solutions and approaches, depending on the environment and the network devices themselves, for automating the configuration management and daily tasks (will be explained in the fourth section), on both traditional and more recent network, and this led the network automation and programmability implementation strategies differ from perspective to another.

In this section, we are going to explain briefly three common approaches and given use cases examples to automate the management plane and operations on the network in separated subsections, which are cited above, and summarized on the Figure I.4 as well.



**Figure I. 4:** Summarized plane for common approaches and use cases.

In the first approach, we will introduce the paradigm of network controlling, and given a use case example with the DNA Center, in the second approach, network automation will be based on

the direct interaction with the network devices, using pre-built frameworks and tools like Ansible. Now, these two previous approaches are typically using device APIs, CLI, and Python under the hood, but this does not preclude the third approach that provide more clarification of these APIs including, REST, NETCONF, and the CLI (management interface), with the ability of creating special form for direct interfacing using Python programing language which enable innovation.

### I.4.1 Controller-Based Network

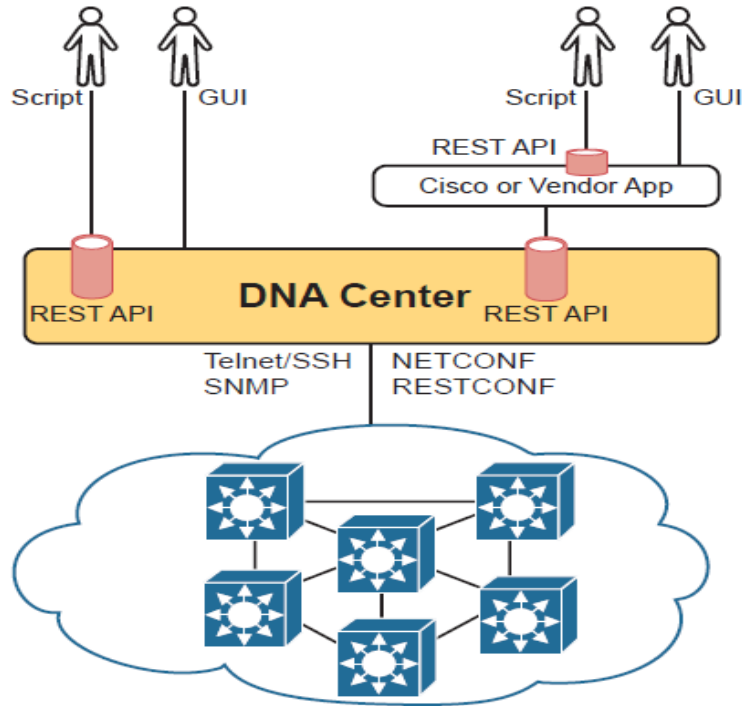
Controllers enable programs to automatically configure and operate networks through powerful application programming interfaces (APIs), allowing three main principles, which are, programmability, abstraction, and centralization from the underlying physical network. In this approach, network automation is based on a centralized controller to configure and manage networking devices, pretty much like the SDN paradigm. However, the controller may be a stateless, which mean each network device has its own control plane and this led to extend the capability of automation over the traditional networking devices by automating the management plane. There is a lot of property and vendor specific automation controllers, to explain more, and as use case, in this approach we chose to explore briefly the last controller released by Cisco which is DNA Center.

#### I.4.1.1 Cisco DNA Center

According to [23] [10], Cisco DNA Center is a network management controller for the enterprise released on 2018, and is based on the fundamentals of Cisco Digital Network Architecture (DNA) and Intent Based Networking (IBN), it delivers pre-installed on a cisco appliance, the software follows the same general SDN controller architecture concepts, Figure I.5, shows the general idea.

As we can see in the figure, Cisco DNA Center supports several southbound APIs to communicate with the devices under two categories:

- Protocols to support traditional networking devices/software versions: Telnet, SSH, SNMP.
- Protocols to support more recent networking devices/software versions: NETCONF, RESTCONF.



**Figure I. 5:** Cisco DNA Center with Northbound and Southbound Interfaces [10].

In the northbound APIs it supports the controlling using scripts that will communicating with the offered REST API platform, or using the pre-built GUI interface which will provides end-to-end network visibility and uses network insights to optimize network performance and deliver the best user and application experience [10].

Cisco DNA Center offers a single dashboard for every core function in the network. With this platform, IT can become nimbler and respond to changes and challenges faster and more intelligently, and it is a set of software solutions that provide [24]:

- A management platform for all of the network.
- An intent-based networking controller for automation of the policies, segmentation, and services configurations.
- An assurance engine to guarantee the best network experience for all the users.

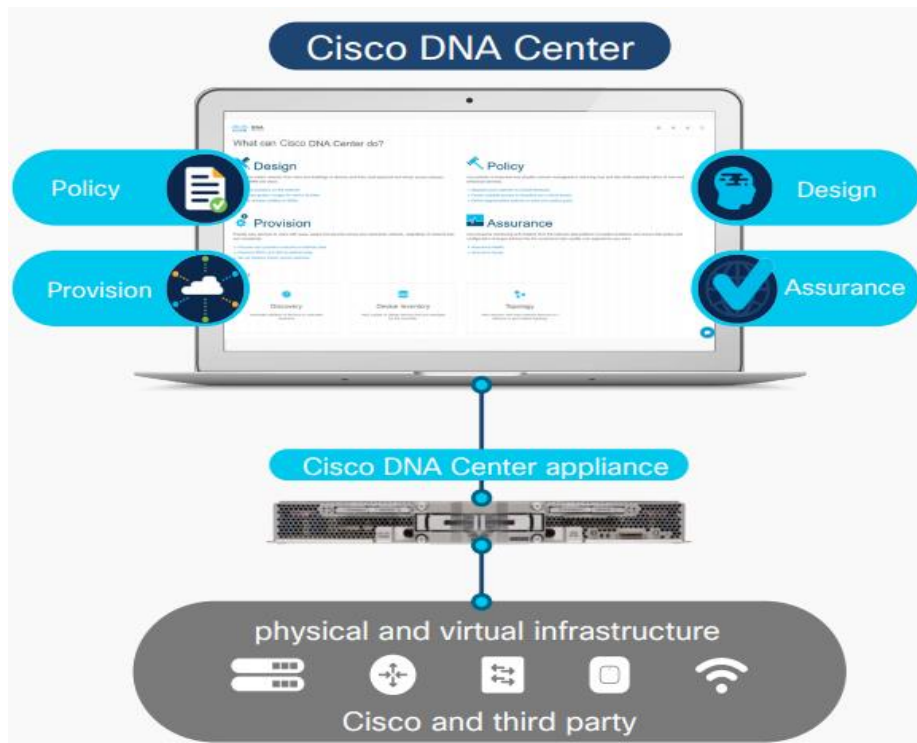
### a- Services

Cisco DNA Center software resides on the Cisco DNA Center appliance and controls all of the Cisco devices both physical and virtual and supports fabric and nonfabric deployments. From

## Chapter I: Network Automation and Programmability Overview

---

the main menu in the dashboard illustrated in Figure I.6, Cisco DNA Center has four general sections aligned to IT workflows [24]:



**Figure I. 6:** Cisco DNA Center dashboard [24].

- **Design:** the ability to design the network for consistent configurations by device and by site, also offer a physical maps and logical topologies to help provide quick visual reference.
- **Policy:** Translate business intent into network policies and apply those policies, such as access control, traffic routing, and quality of service, consistently over the entire wired and wireless infrastructure.
- **Provision:** Device provisioning is a simple drag-and-drop task. The profiles (called scalable group tags or “SGTs”) in the Cisco DNA Center inventory list are assigned a policy, and this policy will always follow the identity. The process is completely automated and zero-touch. New devices added to the network are assigned to an SGT based on identity—greatly facilitating remote office setups.
- **Assurance:** Cisco DNA Assurance, using AI/ML, enables every point on the network to become a sensor, sending continuous streaming telemetry on application performance and user connectivity in real time.

It offers also a broad set of APIs, SDKs, and adapters that extend the capabilities of Cisco DNA Center to external applications, cross-architectural domains, systems and processes, and third-party devices, this controller extensible software platform includes integrated tools for network management, automation, virtualization, analytics and assurance, security, and Internet of Things (IoT) connectivity and can also interface with business-critical tools, it has a lot of services and robust features that aims to bring together the benefits of efficiency and lower risk by combining automation and abstraction in the form of a GUI and intent-based networking concepts to allow users to easily configure and deploy networks [24].

### I.4.2 Network Automation Frameworks

One of the foundation mechanisms that we discuss in this approach to enrich out the research, is the network automation frameworks, there are a lot of them, and often called tools, each framework performs a set of software packages and pre-defined rules that address the Configuration Management System on the network infrastructure. Ansible, SaltStack, Puppet, Chef, and many of tools out there have the capabilities to quickly provision infrastructure by avoiding sequential and manual interactions. With CMS, by combining the automation tools with the characteristics in the desired state we end up with system consistency across the entire system, including appliances from router to switches, middle boxes and even server automation use cases [25], all these tools have commonalities and differences architectural points between them, and they are listed in two parts bellow.

**a- Commonalities points [25]:**

- All about automating configuration management.
- Checking the current state before changing it (*Idempotency*).
- Facts and gathering information.
- Work with Modules and libraries behind the scenes.
- Open source foundation.

**b- Differences points [26]:**

- Agent-based versus agentless.
- Centralized versus decentralized.
- Custom protocol versus standards-based protocol.

- Language for extensibility.
- Domain-specific language (DSL) versus standards-based data formats and general purpose languages.
- Push versus pull versus event-driven.

In the next sub-section, we are going to focus more on Ansible as an example from those frameworks, which is the common use case tool, because, it is agentless, easy to use, and popular in networking community [25].

### I.4.2.1 Ansible

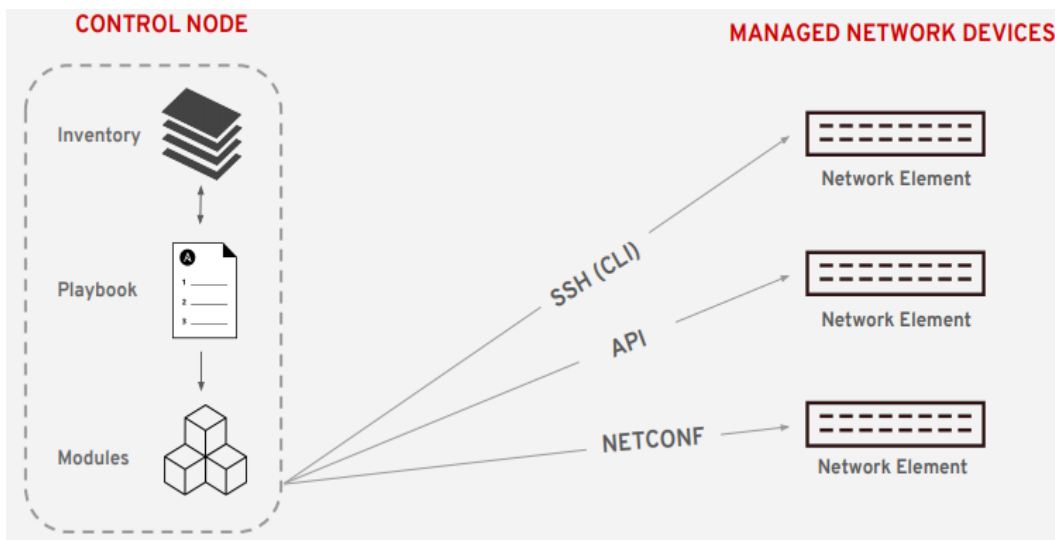
Ansible is a popular network automation framework written in Python that has been used to automate IT operations and configuration management. It simplifies the management of different infrastructure nodes and translates the business logic into well-defined procedures [27], it is a free and open-source software created by Michael DeHaan in 2012, acquired by Red Hat corporation in 2015 [22]. It started support for networking devices beginning with Ansible 1.9, and with Ansible 2.9, its current support for network devices have grown extensively. It can interact with network devices using either SSH or via API if the network vendors support APIs on their equipment [27]. Ansible characterized by [22]:

- **Simplicity:** No need to special coding to get started, all instructions, or tasks to be automated, are documented in a standard, human-readable data format that anyone can understand.
- **Agentless:** No need to install agent or additional software in the targeted network devices.
- **Extensibility:** Since it is open source and written in Python, it can be extended, by adding modules and integrate other to enable a given set of functionality.

Ansible and other tools, was intended to automate server IT infrastructure, it executes in a distributed fashion, where the Ansible control host connects to each server being automated via SSH and subsequent copies Python code to each server, this code is what performs the automation task at hand. After it extends the capability for automating network devices, the idea was changed a little bit to operates in a centralized fashion, communication via SSH (CLI), HTTP based API, NETCONF ... etc. and Python codes are run locally in the control node, instead of copying in the



targeted network devices [26]. Figure I.7, illustrate the general idea of the Ansible workflow with network element.



**Figure I. 7:** Ansible workflow with network devices [28].

As the figure shows, the Playbook instructions and execution are based on modules which are simply a set of Python files, in order to know how to use Ansible for network automation we need to get familiar with some terminologies and concepts, those terms are extracted from the Ansible documentation [29], in combination with the Edelman Ansible Report [22], to add more explanation, we given also examples from others other references.

### **a- Control node**

Any machine (except Windows) with Ansible installed. can run commands and playbooks, invoking `/usr/bin/ansible` or `/usr/bin/ansible-playbook`. We can use any computer that has Python installed on it as a control node - laptops, shared desktops, and servers for running Ansible.

### **b- Managed nodes**

The network devices (and/or servers) that are managed with Ansible, are also called hosts. Ansible is not installed on managed nodes.

### **c- Inventory**

The managed nodes are listed in an inventory file or sometimes called a *hostfile*. the inventory can specify information like IP address for each managed node. An inventory can also organize managed nodes by creating and nesting groups for easier scaling. Using an inventory file,

such as the coming example, enables us to automate tasks for specific hosts and groups of hosts by referencing the proper host/group using the hosts parameter that exists at the top section of each play. It is also possible to store variables like the username and password within an inventory file. Figure I.8, shows an example of an inventory file include two groups of cisco devices, IOS, and NXOS, with specifying common variables.

```
[all:vars]
ansible_user=admin
ansible_password=ansible123

[ios]
rtr1 ansible_host=52.14.208.176
rtr2 ansible_host=18.221.195.152

[nxos]
switch01 ansible_host=3.15.11.56

[ios:vars]
ansible_network_os=ios
ansible_connection=network_cli

[nxos:vars]
ansible_network_os=nxos
ansible_connection=nxapi
```

**Figure I. 8:** Example of an Ansible inventory file [30].

### d- Modules

The units of code Ansible executes. Each module represents a particular use of task, from administering users on a specific type of database to managing VLAN interfaces on a specific type of network device. we can invoke a single module with a task, or invoke several different modules in a playbook. There are four common network module and they are listed in following Table I.2 [31].

**Table I. 2:** Four main network devices module [31].

Module	Description
<b>command</b>	Command modules run arbitrary commands on a network device.
<b>config</b>	Config modules allow configuration on the network device in a stateful way (idempotent)
<b>facts</b>	Fact modules return structured data about the network device
<b>resource</b>	Resource module can read and configure a specific resource on a network device.(e.g. VLAN).

### e- Playbooks

The playbook is the top-level object that is executed to automate, it is an ordered list of Plays and tasks saved, so it can run those tasks in that order repeatedly. Playbooks can include variables as well as tasks, written in YAML (Yet Another Markup Language) and easy to read, write, share and understand. Figure I.9, shows an example of a Playbook file that include one play and one task for adding VLANs on a Cisco host.

```
---
- name: add VLANs
  hosts: cisco
  tasks:
    - name: add VLAN configuration
      ios_vlans:
        config:
          - name: desktops
            vlan_id: 20
          - name: servers
            vlan_id: 30
          - name: printers
            vlan_id: 40
```

**Figure I. 9:** Example on an Ansible Playbook file [30].

**Note:** YAML is just another standard type of structured data format, as JSON and XML, it starts with “---” in the top of the file, and it is based on indentation for separating and describe lists and object information.

### f- Play

One or more plays can exist within an Ansible playbook. In the previous Playbook example, there are a single play within the playbook starts with a header section where play-specific parameters are defined (name, hosts, and it may include the connection type). Each play is comprised of one or more tasks.

### g- Tasks

The units of action in Ansible. We can execute a single task once with an ad-hoc command. Tasks can also use the name parameter just as plays can. The next line after declaring the task name in the example shown in Figure I.9, task starts with *ios\_vlan* and it will execute the Ansible module called *ios\_vlan*.

### h- Templating

Ansible supports Jinja2 language which will be able to create templates that represent a device's configuration but with variables, as the example in Figure I.10.

<pre>hostname {{hostname}} ! interface GigabitEthernet0/0 ip address {{address1}} {{mask1}} ip ospf {{OSPF_PID}} area {{area}} ! interface GigabitEthernet0/1/0 ip address {{address2}} {{mask2}} ip ospf {{OSPF_PID}} area {{area}}</pre>	<pre>--- hostname: BR1 address1: 10.1.1.1 mask1: 255.255.255.0 address2: 10.1.12.1 mask2: 255.255.255.0 RID: 1.1.1.1 area: '11' OSPF_PID: '1'</pre>
--	---

**Figure I. 10:** Example of configuration file template using Jinja2 in the left with the associated values in a YAML file in the right [10].

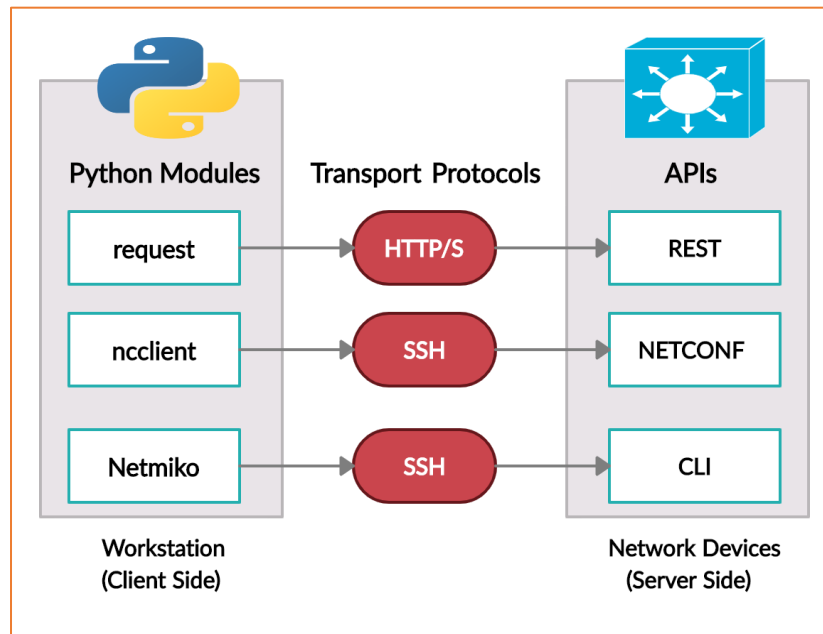
**Note:** The goal of using templates is to decouple the inputs from the underlying vendor proprietary syntax (CLI) of the configuration file in separated files, in order to avoid the writing of repetitive lines, then Ansible helps to bridge the gap between rendering the inputs and values placed in a YAML file with the configuration templates [22].

As a review, Ansible offers a great and a robust way to automate the configuration of networking devices in form of simple tasks in single or multiple playbooks, it does not need any agents or additional software to initialize it, also, it leverages Python for extensibility, it supports also templating using the *Jinja* templating language. Ansible uses many programming interfaces, including the ability to interface with the CLI via SSH in automatic fashion. In the next sub-section, we are going to add more clarification on those application programming interfaces (APIs), and the intended management interface (CLI), for each network device, and how we can deal with them using Python programming language as a separated approach.

### I.4.3 Device APIs and Management Interfaces

A major aspect about what actually makes network automation and programmability possible and continue in evolving is the application programming interfaces or APIs on network devices, before we dive in this approach, we might be wondering about what is an API. An Application

Programming interface is just a way for one program to communicate or learn the variables and data structures (often JSON/XML based), used by another program, making logic choices based on those values, changing the values of those variables, creating new variables, and deleting variables. APIs allow programs running on different computers to work cooperatively, exchanging data to achieve some goal [10]. If we still be wondering about what it does in networking, well several types of network device APIs already exist, each with a different set of conventions to meet a different set of needs, for example the Openflow protocol that we mentioned earlier, REST, NETCONF protocol, and even the Command Line Interfaces (CLI) are considered as an APIs in automatic network management system (but it is UI, designed more for human to use) [32], Since we can deal with them directly using python, this sub-section will provide a brief specification for those common APIs types found on network devices, with given associated Python modules. Figure I.11, illustrate the general idea between Python libraries choices in one side through transport protocols, that can be interfacing with Device APIs in the other side.



**Figure I. 11:** Illustrated plan for devices APIs with their associated Python modules.

### I.4.3.1 REST API

An API that uses REST methods which stands for REpresentational State Transfer, is often referred to a RESTful API, are becoming more popular and more commonly used in the networking industry, although they've been around since the early 2000s [26]. Roy Fielding, the creator of the

## Chapter I: Network Automation and Programmability Overview

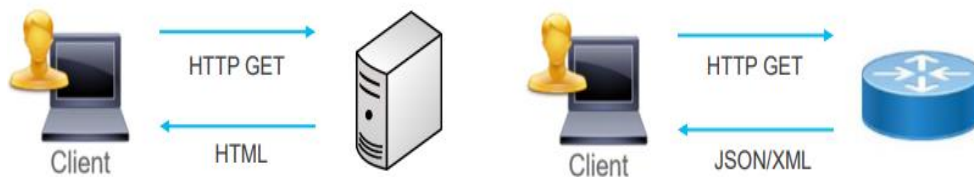
---

REST API defines six foundational rules about what makes a REST API and what does not, and they are as flow [33]:

- It uses a client/server architecture.
- It follows a stateless operation.
- It defines a clear statement of cacheable/uncacheable.
- It has a uniform interface.
- It has a hierarchical layered system.
- It allows the code-on-demand.

**Note:** when we are honoring the six guiding principles of REST, we can call this application interface RESTful [33].

Most of the APIs that exist today within network infrastructure are HTTP-based RESTful APIs, this means that when we hear about a RESTful API on a network device or SDN controller, it is an API that will be communicating between a client and a server [34]. The client would be an application such as a Python script or web UI application, which will be requesting the server for needed information, and the server would be the network device or controller that will be responding by sending structured data like JSON or XML, much like we using a web browser to request a web site for HTML pages. Figure I.12, illustrate the general idea.



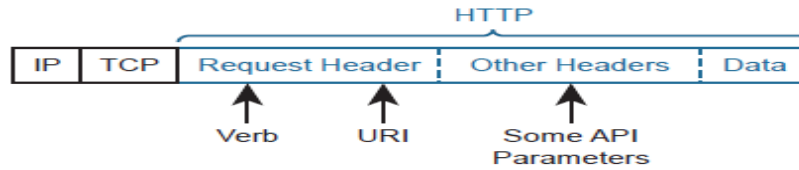
**Figure I. 12:** Client and server (web server) communication using REST API [35].

The software industry uses a memorable acronym CRUD for the four primary actions performed by an application. Those actions are: Create, Retrieve/Read, Update, Delete [10]. RESTful APIs use HTTP methods or verbs to gather and manipulate data and variables that mirror CRUD actions. Each action has the associated HTTP verbs that works with it, and they are listed in the Table I.3 below.

**Table I. 3:** Comparing CRUD Actions to REST Verbs [10].

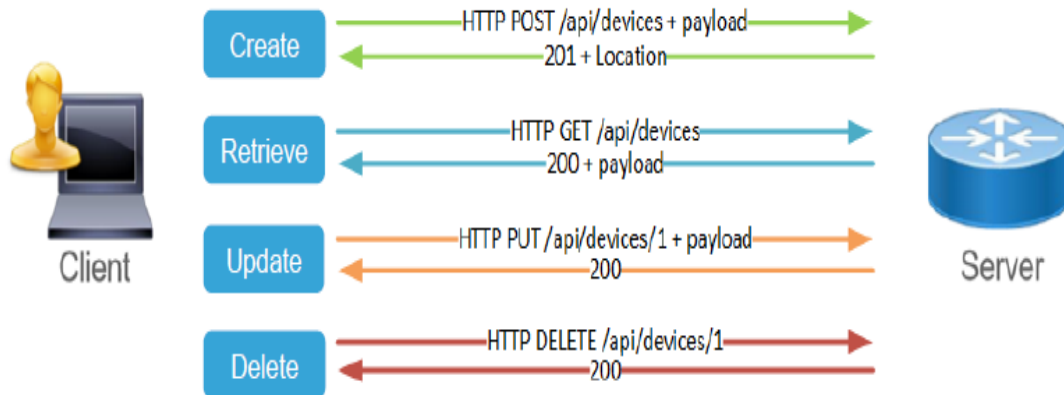
Action	CRUD Term	REST (HTTP) Verb
Create new data structures and variables	Create	POST
Read (retrieve) variable names, structures, and values	Read	GET
Update or replace values of some variable	Update	PATCH, PUT
Delete some variables and data structures	Delete	DELETE

HTTP has the concept of an HTTP request and reply, with the client sending a request and with the server answering back with a reply. Each request/reply lists an action verb in the HTTP request header, which defines the HTTP action. The HTTP request messages also include a URI which identifies the resource being manipulated for this request, as always, the HTTP message is carried in IP and TCP, with headers and data, as represented in Figure I.13.



**Figure I. 13:** HTTP Verb and URI in an HTTP Request Header [10].

The response from the sever also include a code message that expresses the status of the actions, like 201; that refer to a new resource created, or 200; which mean that the request is succeeded, and other status codes, each one has a specific response description. In Figure I.14, we can see the process clearly between a client and a network device.



**Figure I. 14:** Format of the HTTP request and response between the client and the network device (server) [35].

There are some other generic headers that we should be comfortable with and expect to see, they provide details about the meta data and how the REST based HTTP API work and they are a part from the request and the response object, including the Content-Type and Accept and other parameters listed in Table I.4, with example value and the purpose [36].

**Table I. 4:** Some generic headers that provides details about the meta-data [36].

Headers	Example Value	Purpose
Content-Type	application/json	Specify the format of the data in the body
Accept	application/json	Specify the requested format for returned data
Authorization	Basic dmFncmFudDp2YWdyYW50	Provide credentials to authorize a request
Date	Tue, 25 Jul 2017 19:26:00 GMT	Date and time of the message

As we mentioned, there is an associate Python module available in the PyPI online repository which is *request*, that gives us the ability to build a full REST based HTTP client API with simple authentication, headers and response tracking, it includes all the REST API calls and HTTP functions in form of a pre-built python methods like *get ()*, and *put ()*, in order to manipulate on the networking devices resources in a programmable style. In Figure I.15, we can see some use cases using the REST API based on the HTTP protocol in the context of network device.



**Figure I. 15:** HTTP Verbs in the context of network devices [35].



### I.4.3.2 NETCONF Protocol

As we know that the Simple Network Management Protocol (SNMP) had been widely used for fault handling and monitoring for many years. However, his capabilities in configuration especially in complex and large scale networks are very limited [37], and it was clear that NETwork CONFiguration has the ability to replace SNMP as the incumbent configuration management protocol [38].

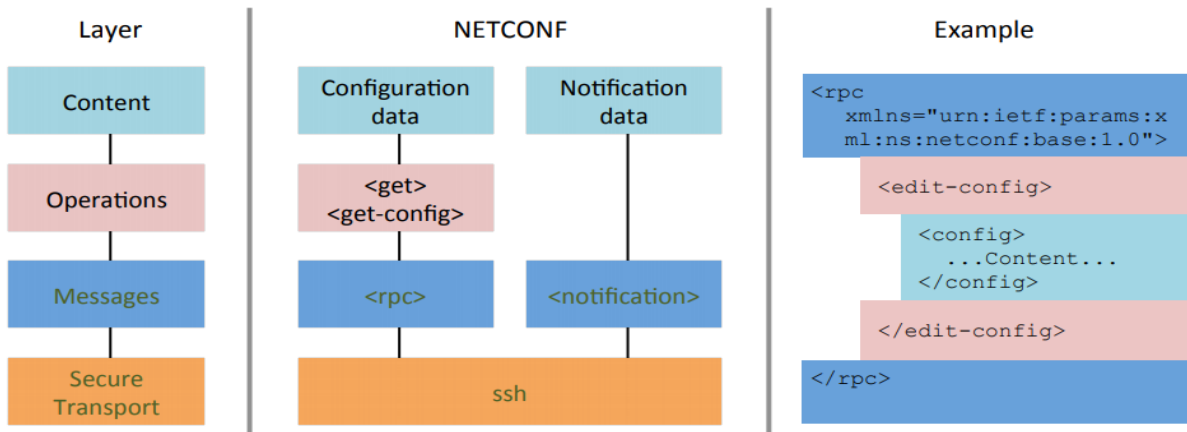
NETCONF protocol is an IETF network management protocol initially defined in RFC 4741 (2006), and revised in RFC 6241 (2011) [39]. The NETCONF protocol defines a simple mechanism through which a network device can be managed, configuration data information can be retrieved, and new configuration data can be uploaded and manipulated. The protocol allows the device to expose a full, formal API. Applications can use this straightforward API to send and receive full and partial configuration data sets [39].

NETCONF uses a simple remote procedure call (RPC) that is realized by exchanging *<rpc>* and *<rpc-reply>* messages encoded in XML format to facilitate communication between a client and a server. The client can be a script or application typically running as part of a network manager. The server is typically a network device. NETCONF architecture is designed to distinguish between writable configuration data used to control the operation of a device and state data containing device statistics and status description. Configuration data can be retrieved by *<get-config>*, modified by *<edit-config>*, copied by *<copy-config>*, and delet by *<delete-config>*, whereas *<get>* is used to retrieve available state and configuration data. In addition, NETCONF distinguishes between three configurations datastores on a managed device [37]:

- a- Running:** configuration currently active on the device
- b- Candidate:** a standby configuration, which can be manipulated without affecting the current device's running configuration.
- c- Startup:** the initial configuration of a device.

This protocol has a conceptual of four-layered process illustrated in Figure I.16, with an example.

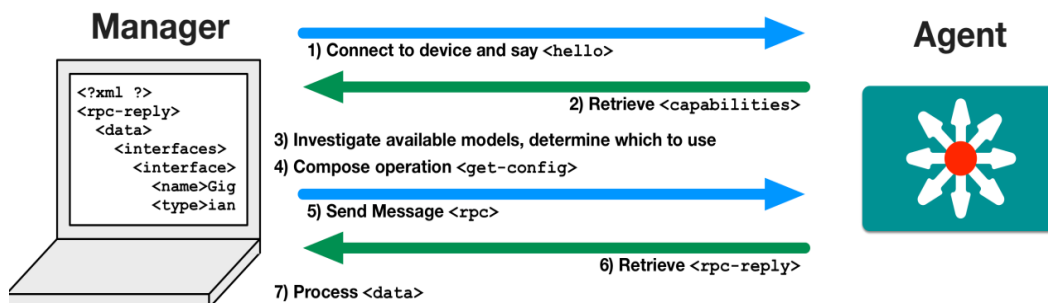
Each layer has a specific role and characteristic and they are as follow [39]:



**Figure I. 16:** NETCONF protocol layer and content [40].

- The Secure Transport layer provides a communication path between the client and server. NETCONF can be layered over any transport protocol that provides a set of basic requirements including SSH protocol (port 830 by default) and SOAP.
- The Messages layer provides a simple, transport-independent framing mechanism for encoding RPCs and notifications.
- The Operations layer defines a set of base protocol operations invoked as RPC methods with XML-encoded parameters, like we saw in the example (*<edit-config>*).
- The Content layer uses YANG data modeling language which has been developed (standardized in RFC 6020) for specifying NETCONF data models and protocol operations sent to or from a network device formatted in XML as well.

The IETF group defines a set of capabilities that a client or a server may implement. Each peer advertises its capabilities by sending them during an initial capabilities exchange using *<hello>* element. Each peer needs to understand only those capabilities that it might use and must ignore any capability received from the other peer that it does not require or does not understand.



**Figure I. 17:** NETCONF communication with the initial capabilities exchange [41].

The example in Figure I.17, depicts the NETCONF communication started by exchanging capabilities in the first and second steps between the manager and the agent installed on the network element, based on these capabilities, the manager will send and receives needed operations encapsulated by RPCs (steps from three to seven).

NETCONF protocol is a major step towards an automated XML-Based network management system. The major issue of NETCONF is a lack of support from the industry, and few publications on the NETCONF implementation [37], Cisco as an example Started to support NETCONF/YANG recently on his devices depending on the releases.

There is a dedicated Python module to handle NETCONF programmatically, which is *ncclient* library that facilitates client-side interfacing and application development using pre-defined methods, like *connect()* that accept device information from the *host*, *port*, and *username* as parameters to handle the connection on the targeted network devices, and then passing the operational methods like *get\_config()* and *edit\_config()* to manipulate data [42].

### I.4.3.3 CLI

The command Line Interface consider as the primary user interface when we configuring, monitoring and maintaining networking devices, this interface is accessible either directly using console or terminal, or using remote access methods which are SSH and Telnet protocols. Telnet protocol offers support for connecting to a device which has the support for it and managing it from a remote location. The main problem of the Telnet protocol is the lack of security within its session. All the data sent across the network are being sent as plain text, neither one of these sent packets aren't being encrypted.

SSH protocol offers a secure alternative for remote management. The SSH creates encrypted sessions using a public and private key and authenticates the user that tries to by comparing the credentials configured to those entered and then granting access to the legitimate users [43]. The reality is that the migration from Telnet to SSH is arguably the biggest shift we've had in network operations over the past decade. There are some requirements that are expected to be configured before starting a Telnet or SSH session with a device. These requirements are: configuring a hostname, a logical address, an enable password (for data network devices), a user and a password

and at least one VTY line that permits telnet traffic, also domain address and generating encryption key specifically for SSH.

The most important thing to realize as it pertains to managing devices via the CLI is that the CLI was built for humans. It was put on devices to improve usability for human operators. The CLI was not meant to be used for machine-to-machine communication. However, when all we have is the CLI, CLI is what gets used. This is why there are plenty of network management platforms and custom scripts that have been built over the past two decades that perform management and automated operations using the CLI over SSH [26].

We are going to explain more this approach, since that we could use the CLI from automation perspective to build a system to ease the configuration management of networking devices, using python scripting languages, taking advantage of libraries like Paramiko and Netmiko, which we will dive into more in the second and the third chapter. Telnet protocol also has his own module in python which is Telnetlib and it is a standard Python Library, it used to handle the connections to network devices in order to manage them.

### I.5 Common Network Automation Tasks

All the aforementioned approaches in network automation effect on the administrative tasks in a positive way as they save effort, time and cost, in this section, we will shed the light on the most common tasks for which who deals with the networking devices can deploys an automated workflow using the preceding approaches and tools, Edelman, Scott and Matt in their book Summarize all these tasks in which network automation make sense and they are as follow [26]:

#### **a- Device Provisioning:**

Involves the process of preparing and equipping a network device to allow it to provide new services to its users. If we take this process and break it down into two steps, the first step is creating the configuration file, and the second is pushing the configuration onto the device.

#### **b- Data Collection**

This task includes all of the methods of monitoring and collecting important data from the network, typically using protocols like SNMP, these tools poll certain management information

bases (MIBs) and return data to the monitoring tool. Based on the data being returned, it may be more or less than actually need.

### **c- Migrations**

Network migrations, therefore, involves transferring the data and programs from an old network to a new network. This migration can be in an effort to start using a totally new network, but it can also include extending the current network with an add-on system.

### **d- Configuration Management**

Deploying, pushing, and managing the configuration state of the device. This includes anything as basic as VLAN provisioning to more complex workflows that configure a top-of-rack switches, firewalls, load balancers, and advanced security infrastructure, to deploy three-tier applications.

### **e- Compliance**

It refers to the process of configuration compliance checks and configuration validation; it uses the data gathered and it is more than possible to verify if something is True or False. It's easy enough to start small with one compliance check and then gradually add more as needed.

### **f- Reporting**

Reporting is another administrative task which needs to collect important information to express condition and state within in network, the report can be in many formats including HTML reports that are deployed to a web server for easy viewing, or it could be a CSV file, even in a simple text file.

### **g- Troubleshooting**

Network troubleshooting is the combined measures and processes used to identify, diagnose and solve problems within a computer network. It's a logical process that network engineers use to resolve network problems and improve network operations. Troubleshooting is an iterative process used also the collected data to analyze [44].

If we can notice, all of these works meet in two processes, namely pulling and push data, any network administrator can easily create automated system workflows to accomplish these tasks

using one or more of the previously mentioned approaches, depending on the capabilities of the device and what it can support, instead of logging into each device and do this works. In this thesis, we created a management platform that demonstrate how we can do those, which we will explain more in the second and the third chapter.

### **I.6 Conclusion**

In this informational background chapter, we presented a brief explication of the most common approaches and methods, that started to emerge in the network industry over the last decade, which are fall under the topic of network automation and programmability, including, controller-based networking, network automation framework, devices APIs and the management interfaces, we gave also in a separated section, most of the common tasks that can be addressed using these technologies, we introduced the paradigm of software-defined networking to solve the mentioned problems in the general introduction by its specification, also, it inspired network automation and programmability to extend and cover many contexts and domains, it forced the evolving in network devices to be more programmable in mind.

**CHAPTER II:**  
**MANAGEMENT INTERFACES**  
**APPROACH AND THE PROPOSED**  
**WORKFLOW FOR DEVELOPING**  
**AUTOMATION PLATFORM**

## **Chapter II: Management interfaces Approach and the Proposed Workflow for Developing Automation Platform.**

---

### **II.1 Introduction**

The process of automating tasks in network infrastructure actually is differ based on the nature of the network device and what it can support, which is not something desirable in a multi-vendor and multi-nature network environment. Luckily, Python is broadly used to perform network automation with just the CLI that exists in all devices, with his wide set of open source libraries (such as Netmiko and Paramiko), also there are endless possibilities for network devices interactions for different vendors.

In the research trip for solutions or reduce the mentioned problematics, we adopted to use the third automation approach described in the first chapter, exactly Python with CLI, this approach enable innovation, development and extensibility with Python.

In this chapter and in the **PART A**, we will explore the reason why exactly Python for networking contexts, and then we are going to focus on Netmiko library which is one of the most widely used libraries for network interactions. In addition, we will perform some best practices and integrated functionalities such as multithreading and TextFSM (which is Python module created by Google).

In **PART B**, we will explore our proposed framework and workflow to develop a Network Management Platform (NMP) intended for the network automation.

### **II.2 PART A**

#### **II.2.1 Python for Network Automation**

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse [45]. It addresses many areas in technology, web and internet development, data mining and visualization, desktop



## Chapter II: Management interfaces Approach and the Proposed Workflow for Developing Automation Platform.

---

GUI, analysis, game building, and automation testing; that's why it's called a general-purpose language [46].

### II.2.1.1 Why python?

The functionality and flexibility of the programming language are provided by its ability to work with certain libraries, also called modules. These libraries contain functions and protocol implementations that allow developers to create relatively small size but very powerful scripts, that can be reused as functions in other scripts. In order to increase the effectiveness of the programming language, its developers have allowed the addition of non-standard libraries, with added features being part of the extended Python library [43].

From [26] , it's worth pointing out that we do not hold any technology religion to Python. However, we feel when it comes to network automation it is a great first choice for several reasons. First, Python is a dynamically typed language, meaning that we don't need to define variables and objects before we start using them. This simplifies the getting started process. Second, Python is also super readable. It's common to see conditional statements like `if device in device_list:` and in that statement, we can easily decipher that we are simply checking to see if a device is *in* a particular list of devices. Another reason is that network vendors and open source projects are building a great set of libraries and tools using Python. This just adds to the benefit of learning to program with Python. According [46], also, there are three reasons of choosing Python for network automation:

- **Readability and ease of use:** Python is similar to the English language, structured to have readable statements, and does not require ‘;’ or curly braces “{”.
- **Libraries:** Python has a wide range of libraries and packages located on a website called *PyPI* (<https://pypi.python.org/pypi>), and linked to a GitHub repository. When we want to download the library in our PC, we use a tool called *pip* to connect to *PyPI* and download it locally.

Network vendors, such as Cisco, Juniper, and Arista developed libraries to facilitate access to their platforms. Most vendors are pushing to make their libraries easy to use and require minimum installation and configuration steps to retrieve useful information from devices.

## Chapter II: Management interfaces Approach and the Proposed Workflow for Developing Automation Platform.

---

- **Powerful:** Python tries to minimize the number of steps required to reach the end-result. As we know, to print *hello world* using Java, we will need a block of code, instead in Python, we need just one line.

### II.2.2 CLI Based Interaction Libraries

Python has many open source and proprietary libraries and modules that interface with the device CLI as an API, since it does not return structured data, the programmer must parse the string output to pick-up and load needed information on the program. The management relied on the engineer's interpretation of the data returned from the device for appropriate action [47].

Network environments contain multiple devices from different vendors, and each device plays a different role. Design an automation framework for network devices are essential. Large enterprises and service providers usually tend to design a workflow that can automate different network tasks and improve network resiliency and agility [46]. The workflow contains a series of related tasks that together form a process or a workflow that will be executed when there's a change needed on the network.

In other sides, there are also attempts to build and develop open source libraries that look for to support all kinds of networking devices. In Table II.1, are listed the most popular open source Python libraries, that are used to automate network devices. We refer to these three common libraries from the referenced book with definition and the project link source [46].

**Table II. 1:** Open source python libraries for network automation [46].

<b>Network Library</b>	<b>Description</b>	<b>Link</b>
Netmiko	A multi-vendor library that supports SSHing and Telnet for network devices and executes commands on it. Support includes Cisco, Arista, Juniper, HP, Ciena, and many other vendors.	<a href="https://github.com/ktybers/netmiko">https://github.com/ktybers/netmiko</a>
	A Python library that works as a wrapper for the official Vendor API. It provides abstraction methods that connect to devices from multiple vendors and extract information	<a href="https://github.com/napalm-automation/napalm">https://github.com/napalm-automation/napalm</a>

## Chapter II: Management interfaces Approach and the Proposed Workflow for Developing Automation Platform.

---

Napalm	from it while returning the output in a structured format. This can be easily processed by software.	
Nornir	A new automation framework based on Python and consumed directly from Python code without a need to have custom DSL (Domain Specific Language). The Python code is called runbook and contains a set of tasks that can run against the devices stored in the inventory (supports also Ansible inventory format). The tasks can utilize other libraries (such as NAPALM) to get information or configure the devices.	<a href="https://github.com/nornir-automation/nornir">https://github.com/nornir-automation/nornir</a>

As long as we will use devices in which they have only Command Line Interface (CLI) as the intended method of management through terminal programs, in our experimental part, those libraries above help in automating network devices with only SSH session, in the next section we will dive more in the Netmiko library since it was one of the first attempts, one of the most widely used libraries for network interactions [48], and also very easy to use, we will give some getting started examples to hide the details of using it, as long as we will use it to develop our specific model in the practical part.

### II.2.2.1 Netmiko

According to [49], Netmiko is a multivendor library that simplifies the process of creating SSH connections to different network devices. Since late 2014 Kirk Byers has been working on this open source project available on GitHub. Essentially, Netmiko is based on the standard Paramiko SSH library, that is more general in which we would have to handle all the details to manage each step of the process, enabling access, entering config mode, sending the changes and saving the changes. Fortunately, other frameworks, like Netmiko, do a lot of this work for us by adding some levels of abstraction for networking devices. Netmiko extends the Paramiko ability of SSH to add enhancements, such as going into configuration mode in network routers, sending commands, receiving output based upon the commands, adding enhancements to wait for certain

## Chapter II: Management interfaces Approach and the Proposed Workflow for Developing Automation Platform.

---

commands to finish executing, and also taking care of yes/no interactive prompts during command execution [48].

### a- Why was Netmiko created?

The owner of this library answered this question and he mentions the problems and the challenges behind it, he had observed with many individuals encountered similar issues with Python-SSH and network devices. For example, HP ProCurve switches have ANSI escape codes in the output or the Cisco WLC has an extra 'login as:' message. These types of issues can soak up a lot of development and troubleshooting time and, what is worse, people keep solving the same issues over and over again (including sometimes not solving them and giving up).

So Netmiko was created to simplify this lower-level SSH management across a wide set of networking vendors and platforms [49]. It helps users to hide many details of common device communications functions which gives a greater abstraction with a variety of network device models [50]. Netmiko is much simpler than Paramiko, supports a wide range of devices under three categories on the last update (January 2019) and they are listed in the Table II.2 below:

**Table II. 2:** Netmiko supported devices under three categories [49].

Regularly tested	Limited testing	Experimental
Arista vEOS	Alcatel AOS6/AOS8	A10
Cisco ASA	Avaya ERS	Accedian
Cisco IOS	Avaya VSP	Aruba
Cisco IOS-XE	Brocade VDX	Ciena SAOS
Cisco IOS-XR	Brocade MLX/NetIron	Cisco Telepresence
Cisco NX-OS	Calix B6	CheckPoint GAiA
Cisco SG300	Cisco WLC	Coriant
HP Comware7	Dell-Force10	Eltex
HP ProCurve	Dell PowerConnect	Enterasys
Juniper Junos	Huawei	Extreme EXOS
Linux	Mellanox	Extreme Wing
	NetApp cDOT	F5 LTM
	Palo Alto PAN-OS	Fortinet
	Pluribus	MRV OptiSwitch
	Ruckus ICX/FastIron	Nokia SR-OS
	Ubiquity EdgeSwitch	QuantaMesh
	Vyatta VyOS	

## Chapter II: Management interfaces Approach and the Proposed Workflow for Developing Automation Platform.

---

### b- Netmiko purposes

Kirk Byers also summarizes the purposes of this library on his web site articles [49], in four sentences and they are as follow:

- Successfully establish an SSH connection to the device.
- Simplify the execution of show commands and the retrieval of output data.
- Simplify execution of configuration commands, including possibly commit actions.
- Do the above across a broad set of networking vendors and platforms.

### c- The usage of Netmiko

Like any library in python, Netmiko has many classes and sub-modules to perform certain tasks. These components are well explained in the API documentation available in GitHub [51], before initialize this library and before getting in the manipulation, we must install the package available in the *PyPI* python repository first, using *pip* package manager as follow, in the terminal we type:

```
pip3 install netmiko
```

**Note:** while we were doing this thesis, we found Netmiko version **3.0.0** and it was only supported by python version **3.6** and **3.7**.

Once Netmiko package install in the local computer than we can now check the different components and options available on it. By calling this library on the interrupter and pass *dir(netmiko)*, to extract from the directory all the sub-modules that we can use it as a list. From Figure II.1, we can see a list of some classes and sub-modules work under Netmiko.

## Chapter II: Management interfaces Approach and the Proposed Workflow for Developing Automation Platform.

---

```
In [1]: import netmiko

In [2]: dir(netmiko)
Out[2]:
['BaseConnection',
 'CNTL_SHIFT_6',
 'ConnectHandler',
 'FileTransfer',
 'InLineTransfer',
 'NetMikoAuthenticationException',
 'NetMikoTimeoutException',
 'Netmiko',
 'NetmikoAuthenticationException',
 'NetmikoTimeoutException',
 'SCPConn',
 'SSHDetect',
```

**Figure II. 1:** Check all the Netmiko sub-modules and classes on the Python interrupter.

Now, each one on this list has a specific role while we dealing with networking devices. Hence, in order to establish an SSH session between the interrupter or any code editors we must use one from these highlighted classes in the Figure II.1, *ConnectHandler* or *Netmiko* are factory functions (they perform the same functionality) [49], that selects the correct Netmiko class based upon the *device\_type* parameter. In addition, other parameters are required to directly effect on the specific device. Establishing a connection to a device consists of the following workflow:

1. Create an object representing the device we are going to connect to. This network device object contains attributes such as IP address, SSH port (optional), username, password, and hostname or the IP address.

We can read the data from a JSON file containing network device information, create a dictionary with key-value pairs, read it from a database, etc. The end result being that we create an object with the attributes required to connect to it.

2. Establish the SSH session.
3. Take actions on the device (send commands, parse output, etc.)
4. Disconnect from the device.

Figure II.2, shows a complete example, that contains all of these steps:

## Chapter II: Management interfaces Approach and the Proposed Workflow for Developing Automation Platform.

---

```
1 from netmiko import ConnectHandler #importing required library
2
3 iosv_l2 = {                                #setting a dectionry of devices information
4     'device_type': 'cisco_ios',
5     'ip': '192.168.122.72',
6     'username': 'admin',
7     'password': '2020'
8 }
9
10                                     #passing the the dict in ConnectHandler class
11                                     #and establish an SSH cannel
12 net_connect = ConnectHandler(**iosv_l2)
13 output = net_connect.send_command('show ip int brief') #sending command in the cannnel
14 print (output) #print the apeared output
15 net_connect.disconnect() # closing the channel
```

**Figure II. 2:** Source code example for establishing SSH connection to Cisco IOS switch.

The output here should be similar to what we usually facing in the CLI when we type *show ip interface brief*, in raw text format (string). Now, sending command in the SSH channel is one from many methods that can be used, the following table contains Netmiko commonly used methods.

**Table II. 3:** Netmiko commonly-used methods [49].

Methods	Description
<code>net_connect.send_command()</code>	Send command down the channel, return output back (pattern based).
<code>net_connect.send_command_timing()</code>	Send command down the channel, return output back (timing based).
<code>net_connect.send_config_set()</code>	Send configuration commands to remote device.
<code>net_connect.send_config_from_file()</code>	Send configuration commands loaded from a file.
<code>net_connect.save_config()</code>	Save the running-config to the startup-config.
<code>net_connect.enable()</code>	Enter enable mode.
<code>net_connect.find_prompt()</code>	Return the current router prompt.
<code>net_connect.commit()</code>	Execute a commit action on Juniper and IOS-XR.
<code>net_connect.disconnect()</code>	Close the connection.
<code>net_connect.write_channel()</code>	Low-level write of the channel.
<code>net_connect.read_channel()</code>	Low-level write of the channel.

## Chapter II: Management interfaces Approach and the Proposed Workflow for Developing Automation Platform.

---

### II.2.2.2 TextFSM Integration

As we explained, netmiko can get in the devices CLI and retrieve only strings data which are unstructured, this process known as the *screen scraping*, it does not help for quickly pull out needed information to do something else based on it. TextFSM and *ntc\_template* exist to avoid relatively this problem.

TextFSM is a Python module created by Google that implements a template based state machine for parsing semi-formatted text. Originally developed to allow programmatic access to information given by the output of CLI driven devices, such as network routers and switches, it can however be used for any such textual output [52]. it requires that we define a template consisting of variables and rules. Then process strings against this template and from this, we can obtain structured data [53].

Kirk Byers implements an example based on this module on *show\_ip\_bgp.txt* file, which represent the string output of show IP BGP command in Cisco router, he has manually stripped certain header and he was able to convert it into structured data, then he mentions these steps for the TextFSM process [53]:

- Network device output;
- Processed by a TextFSM template;
- Produces structured data.

In the same context of kirk, various people have already done the parsing for us (for some set of platforms and show commands). Network to Code group created a collection of templates available in GitHub, that can be integrated very easily with Netmiko in order to replace the CLI strings output into JSON format, by rendering these templates. The following steps show us how this process can be done:

- Install TextFSM templates from GitHub *ntc\_templates* into the home directory using *git clone*.

```
$ git clone https://github.com/networktocode/ntc-templates?__s=XXXXXXXX
```

- Setting the index file in an environment variable.



## Chapter II: Management interfaces Approach and the Proposed Workflow for Developing Automation Platform.

---

```
$ export NET_TEXTFSM=/path/to/ntc-templates/templates/
```

The index file is just a way of mapping between platform, command, and the corresponding TextFSM template.

- Using TextFSM in Netmiko scripts.

```
In [6]: net_connect.send_command("show ip int brief", use_textfsm=True)
```

Just by setting `use_textfsm=True` argument in `show_command()` methods, the script will look for the environment variable to find the index file and map the command to her own template platform. We can now access to the key and value pairs in the script and adjust based on it, see the structured output format using pre-built TextFSM template (`ntc_template`) in Figure II.3.

```
In [6]: net_connect.send_command("show ip int brief", use_textfsm=True)
Out[6]:
[{'intf': 'FastEthernet0',
  'ipaddr': 'unassigned',
  'status': 'down',
  'proto': 'down'},
 {'intf': 'FastEthernet1',
  'ipaddr': 'unassigned',
  'status': 'down',
  'proto': 'down'},
 {'intf': 'FastEthernet2',
  'ipaddr': 'unassigned',
  'status': 'down',
  'proto': 'down'},
 {'intf': 'FastEthernet3',
  'ipaddr': 'unassigned',
  'status': 'down',
  'proto': 'down'},
 {'intf': 'FastEthernet4',
  'ipaddr': '10.220.88.20',
  'status': 'up',
  'proto': 'up'},
 {'intf': 'Vlan1', 'ipaddr': 'unassigned', 'status': 'down', 'proto': 'down'}]
```

**Figure II. 3:** Applying TextFSM templates on show command inside netmiko script [53].

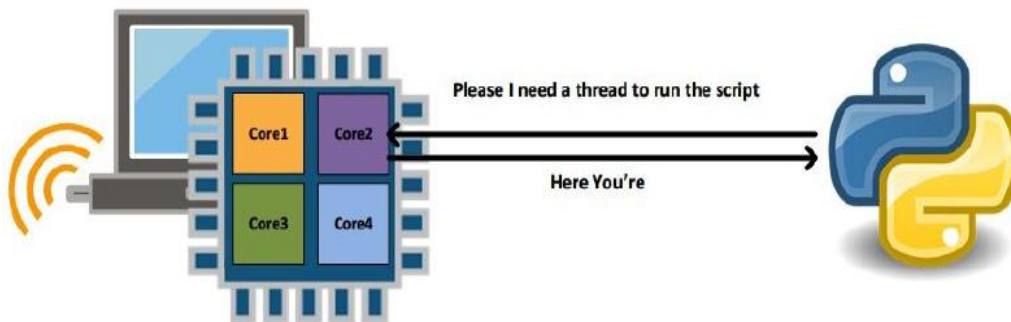
### II.2.3 Multithreading

One of the best practices to do in the automation process using python with libraries like Netmiko, is that we can speed up the execution more by integrating Threading library, it is an efficient way to quickly perform actions when the Python files gets in execution, before we dive in this, the following steps are happening when we do the normal execution of python scripts [46]:

## Chapter II: Management interfaces Approach and the Proposed Workflow for Developing Automation Platform.

---

- Instructs computer processor to schedule a thread (which is the smallest unit of processing).
- The allocated thread will start to execute your script line by line. A thread can do anything, including interacting with I/O devices, connecting to routers, printing output, performing mathematical equations, and more.
- Once the script hits the End of File (EOF), the thread will be terminated and returned to the free pool, to be used by other processes. Then, the script is terminated



**Figure II. 4:** The process of scheduling a thread in the processor once we run a python file [46].

Hence, in the normal case with the Netmiko script, if we need to configure multiple network devices, and if each one takes around 10 seconds to log in, send commands, and log out, and we have around 30 network devices that we need to send commands to it, we would need 300 ( $10 \times 30 = 300$ ) seconds for the program to complete the execution. If we are looking for more advanced or complex calculations, on each item, which might take up to a minute, then it will take 30 minutes for just 30 devices. This starts to become very inefficient when our complexity and scalability grows [54].

To help with this, we need to add parallelism to our program. What this simply means is, we log in simultaneously on all 30 network devices, and perform the same tasks to fetch the required at the same time and this saves a lot of time, depending on the capabilities of the computer processor, a thread is nothing but another instance of the same function being called, and calling it 30 times means we are invoking 30 threads at the same time to perform the same tasks. here a use case example that will demonstrate the difference between these two situations, serial query and parallel query using the threading library [48]:

## Chapter II: Management interfaces Approach and the Proposed Workflow for Developing Automation Platform.

---

```
1. #serial_query.py
2. from netmiko import ConnectHandler
3. from datetime import datetime
4. startTime = datetime.now()
5. for n in range(1, 5):
6.     ip="192.168.20.{0}".format(n)
7.     device = ConnectHandler(device_type='cisco_ios', ip=ip,
8.     username='test',password='test')
9.     output = device.send_command("show run | in hostname")
10.    output=output.split(" ")
11.    hostname=output[1]
12.    print ("Hostname for IP %s is %s" % (ip,hostname))
13. print ("\nTotal execution time:")
14. print(datetime.now() - startTime)
```

The output of running the preceding command is as follows:

```
Hostname for IP 192.168.20.1 is rtr1
Hostname for IP 192.168.20.2 is rtr2
Hostname for IP 192.168.20.3 is rtr3
Hostname for IP 192.168.20.4 is rtr4

Total exection time:
0:00:26.844389
>>>
```

**Figure II. 5:** The result of the use case with serial query [48].

```
1. #parallel_query.py
2. from netmiko import ConnectHandler
3. from datetime import datetime
4. from threading import Thread
5. startTime = datetime.now()
6. threads = []
7. def checkparallel(ip):
8.     device = ConnectHandler(device_type='cisco_ios', ip=ip,
9.     username='test', password='test')
10.    output = device.send_command("show run | in hostname")
11.    output=output.split(" ")
12.    hostname=output[1]
13.    print ("\nHostname for IP %s is %s" % (ip,hostname))
14. for n in range(1, 5):
15.    ip="192.168.20.{0}".format(n)
16.    t = Thread(target=checkparallel, args= (ip,))
17.    t.start()
18.    threads.append(t)
19. #wait for all threads to completed
20. for t in threads:
21.    t.join()
22. print ("\nTotal execution time:")
23. print(datetime.now() - startTime)
```

The output of running the preceding command is as follows:

## Chapter II: Management interfaces Approach and the Proposed Workflow for Developing Automation Platform.

---

```
Hostname for IP 192.168.20.4 is rtr4
Hostname for IP 192.168.20.2 is rtr2
Hostname for IP 192.168.20.3 is rtr3
Hostname for IP 192.168.20.1 is rtr1
Total execution time:
0:00:07.969298
>>>
```

**Figure II. 6:** The output of the use case using parallel query [48].

The calling to the same set of routers being done in parallel takes approximately 8 seconds to fetch the results. As compared to the previous example, 26 seconds is down to 8 seconds for the response. Here are some key points to consider in the previous example:

- The *start ()* method is used to get the thread to invoke the function called in the thread.
- The *join ()* method specifies that until all the threads are complete, the program will not proceed to the next step.
- The output in the program is not in order for parallel threads because, the moment any thread is completed, the output is printed, irrespective of the order. This is different to sequential execution, since parallel threads do not wait for any previous thread to complete before executing another. So, any thread that completes will print its value and end.
- The more threads that assign to the script (and that are permitted by the processor or OS), the faster the script will run [46].

### II.3 PART B

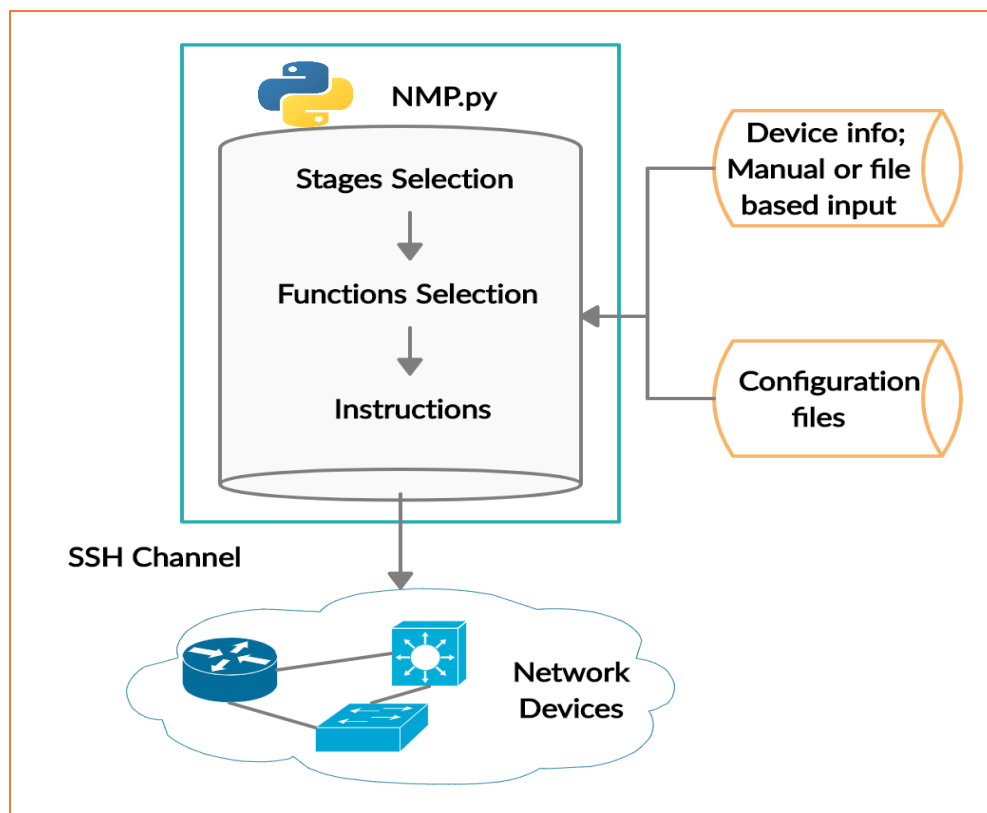
#### II.3.1 The Proposed Network Management Platform

We propose a prototype framework and a workflow as illustrated in Figure II.7, for an efficient application and we have named it Network Management Platform (NMP), it is designed to address and facilitate the network management tasks, where we had to put the strategy and make the example for automated process.

It is necessary to know exactly the nature of the tasks that a network administrator wants to directly affect, and relate them by the program logic, taking into account the efficiency, integrity, reusability, and ease of using the platform.

## Chapter II: Management interfaces Approach and the Proposed Workflow for Developing Automation Platform.

For this project, we cover four common main aspects where automation makes sense, which are: configuration management and provisioning, verification and testing, confirmation and backups, continuous checking and reports generation, that's all cover most of an administrator daily work, and they all about pushing and gathering specific information from devices CLI using SSH protocol. Each aspect has a different strategy to do when we coming to coding, In the following parts, we will introduce each of these tasks, what they supposed to do, and how it uses them in NMP to achieve a network automation process, before we get in the platform building explanation from programming side using python.



**Figure II. 7:** The NMP prototype workflow.

The NMP use pure python language and it is a Netmiko-based modeling application, integrated with other libraries and functionalities, it provides four main parties in form of stages, that match the four administration aspects mentioned above, each one from the stages has functions dedicated to a specific purpose.

## Chapter II: Management interfaces Approach and the Proposed Workflow for Developing Automation Platform.

---

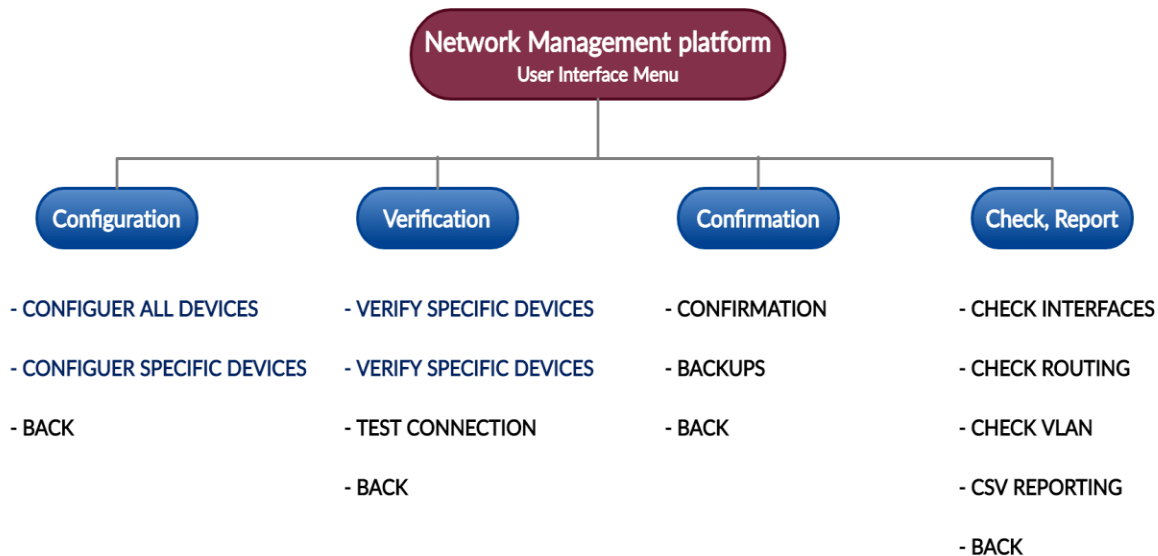
- **The first stage** of the platform is concerned about the task of the configuration management and provisioning across all network regardless of the number of devices and its type, changing, updating, assign configuration about routing protocols, VLAN, QoS and other services to ensure consistency, in some cases, we need to interact with some specific devices or layers in the network, depending on their implementation policies, for this purpose we decided to separate this stage into two functions, which is the configuration of all devices and the configuration of specific devices.
- **The second stage** has three functions and it is about the verification processes. The first and the second functions relevant to the configuration process and what we were done before to ensure if it is done successfully or some input errors are happen, notice that if we choose the first function in the configuration part we have to choose the first of the verification part as well, and so on, it will show us exactly on which device the error occurred and the exact command line type that does not passed correctly. The third function in this stage, is about network testing using test connection function, so every time when we implement new services, we have to check the connectivity between the devices to ensure the correctness of what we did before, using *Ping* command. One of the advantages of separating tasks from each other is the ability to go back and correct previous mistakes before going to the confirmation stage, so if the verification stage does not pass as we want, we will have the ability to return and modify in the first part until we got the purpose.
- **The third stage** is when any administrator finishes setting up tasks, where must save the changes in the network device flash memory for the safety and security, in the case of turning off or restarting the devices. For this, the confirmation unit in the third stage of NMP was created; in addition, it can ensure backups are happening regularly, so he never loses critical network data. This platform improves network security by automating backups and save it in local or another location on our main server, and this what is done on the second function.
- **The fourth stage** in NMP it is concerned in monitoring of the most important network policies implemented in network devices, which helps in the monitoring and troubleshooting,

## Chapter II: Management interfaces Approach and the Proposed Workflow for Developing Automation Platform.

---

so it ease the process of finding issues in somewhere, we built four monitoring main functions, which are firstly, interfaces information and it is about all the ports used or not used, like if this port is up or down, the up time, input packet and more, secondly, it performs the check routing protocol implemented on the devices, like routing tables and the database, thirdly, we have another important function which is checking VLAN and VTP status, this function is specified for the switches because VLAN are running only on it. We can extend to add other functionalities based on what we need to check. Also, one of the necessary tasks performed by the network administrators is to write long and boring reports continuously that express the network situation and send it to the network manager or to higher destinations. Instead of logging into an application and running reports every time, we created a workflow that will auto generate the work for us and this is what the fourth function in the last stage does.

A summary of all the stages our NMP are resumed as an organigram in Figure II.8.



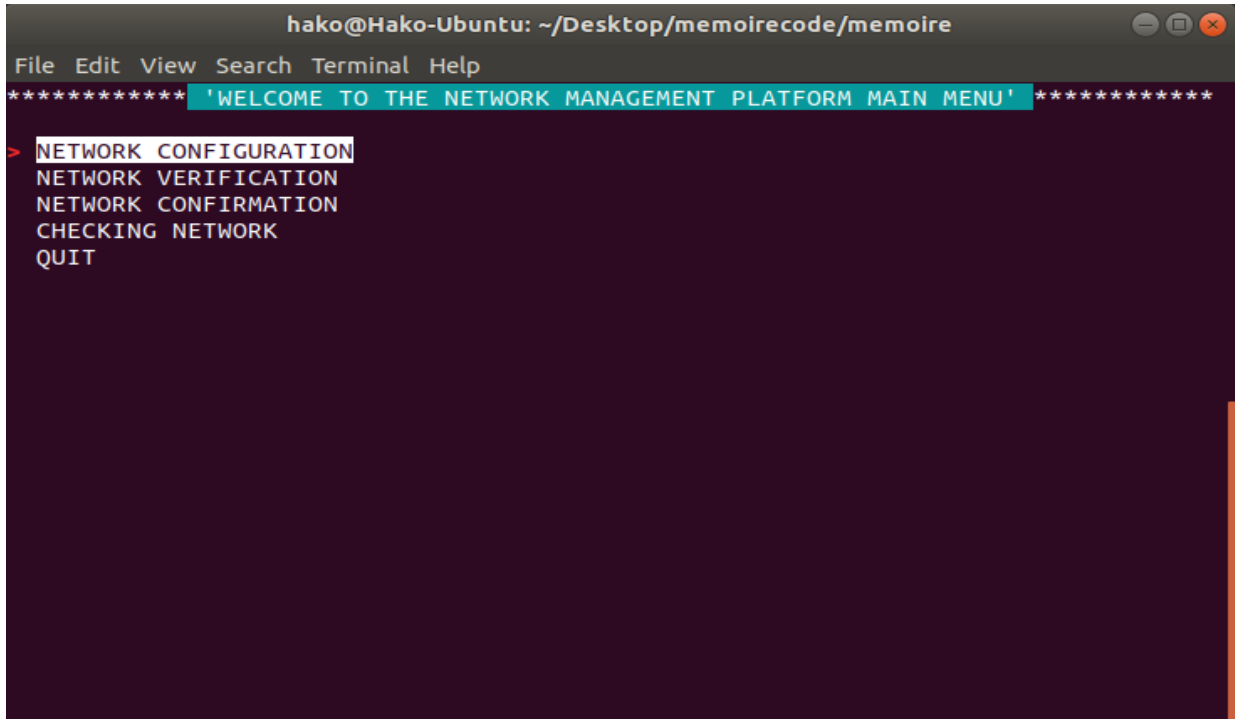
**Figure II. 8:** Summarizing the management platform stages.

This platform is a user interface menu based on selection between the four previous modes, performed by the integration with *simple\_term\_menu* python library to works directly in the terminal as shown in Figure II.9, in order to ease the use. Currently, Linux and MacOS are supported for this module [55]. When we choose the appropriate mode, we hit return to enter in the sub-modes where we find our specific functions with the ability of back again to the main menu

## Chapter II: Management interfaces Approach and the Proposed Workflow for Developing Automation Platform.

---

for each mode. After explaining the code source providing these operations in the third chapter, we are going to do a test case of all these parts separately, with giving results and performance analyzed.



```
hako@Hako-Ubuntu: ~/Desktop/memoirecode/memoire
File Edit View Search Terminal Help
***** 'WELCOME TO THE NETWORK MANAGEMENT PLATFORM MAIN MENU' *****
> NETWORK CONFIGURATION
  NETWORK VERIFICATION
  NETWORK CONFIRMATION
  CHECKING NETWORK
  QUIT
```

**Figure II. 9:** The Proposed Network Management Platform main menu.

### II.4 Conclusion

Python is a great choice to get started and building powerful network automation applications, since it provides pre-built modules to perform programmable administration tasks, like configuration and state verification. As we mention, one of the best modules included in Python is Netmiko, because it supports a wide range of networking devices and easy to use, it simplifies the process of establishing SSH connection between the program script and the remote CLI using *ConnectHandler* class, it has also some pre-defined methods like *send\_command* and *save\_config* to minimize hard coding. There are many functionalities that we can integrate into our code as multithreading and TextFSM, and that is what we will be used to develop our proposed automation platform NMP.



**CHAPTER III:  
NETWORK MANAGEMENT  
PLATFORM FOR NETWORK  
AUTOMATION**

### III.1 Introduction

In this experimental chapter, we will dive more into our NMP toward an automatic network management system. After providing the technical requirements and the development methodology, the following sections give a detailed explanation regarding the application development of NMP, because it is addressed to perform the most of the network administration tasks easily, as detailed in the PART B of the previous chapter. Also to know how it easy and flexible can get an automation process based on Netmiko library.

The auxiliary and stages functions development of the NMP are well explained in two separated section, including the workflow and Python program step by step followed by a proof discussion.

In the testing section, we will use GNS3 that is one of the most popular choices in network emulation, for building a real case of campus network topology to test our automation platform on. We will follow this experiment by discussing the benefits, drawbacks and limitations of our platform, and finally we propose a future enhancements and perspectives.

### III.2 Technical Requirements

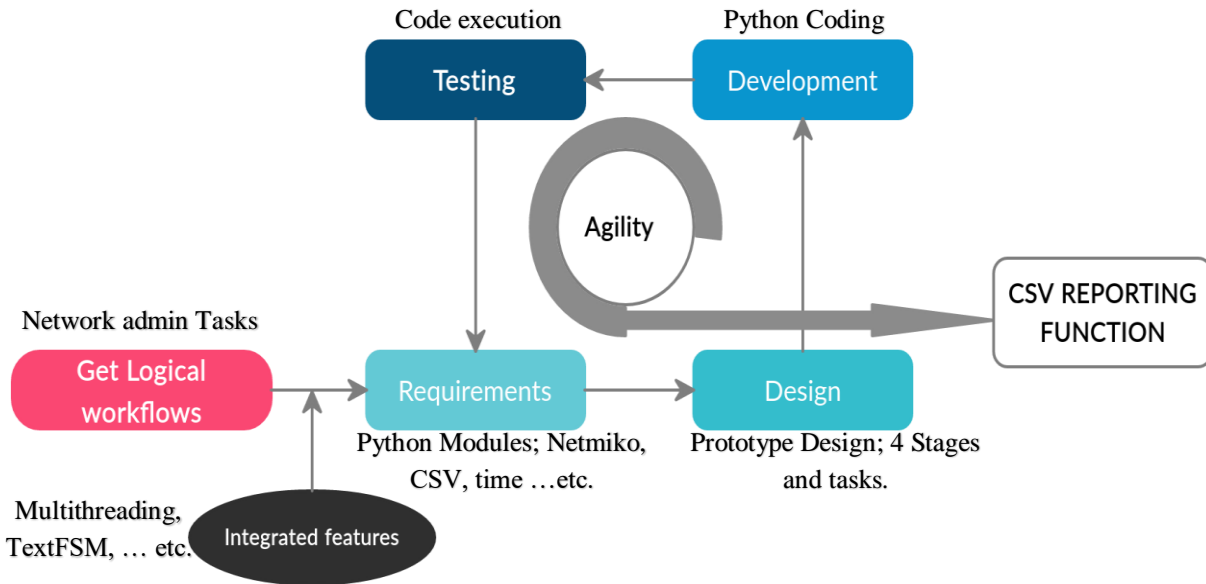
The following technical requirement has been used in this chapter:

- **A laptop with Ubuntu distribution v18. 04:** The best thing with it, is we don't need to use nested virtualization when we install GNS3 which means get a higher performance, also, it is very flexible when it comes into programming and application development.
- **GNS3 v2.2.5:** Open source computer software for network emulation.
- **Sublime Text3 v3.2.2:** Sophisticated text editor for code, markup and prose, which has a slick user interface, extraordinary features, and good performance [56].

### III.3 Development Methodology

The management platform development method was based on an agile methodology as we can see in Figure III.1, in other words starting with some code as we saw it in the second chapter, and then improve it again to accomplish more, that's helpful in the continuous iteration and the ongoing testing directly in the network built it previously in GNS3, to get the logic behind the

manual methods, and poured it into the program script, at the same time we improved it as much as possible to get a comprehensive work regarding the administration's tasks mentioned in the second chapter.



**Figure III. 1:** Example of developing CSV Reporting function using agile methodology.

These methods provide agility between different stages, starting from getting logical workflows and integrated features, passing by setting up software requirements, designing ideas, developing, and ending up by testing and getting feedback about the program which gives the chance again for repeating the same life cycle, this method is suitable for those kinds of experimentation mode, as long as the network administrator is the same who iterate and develop this software.

### III.4 NMP Auxiliary Functions Development

The platform development passed through several steps before getting its last shape. To organize the work, we determined the development methodology shown above, and we were in each time approached the network manual interaction with the automated processes.

Netmiko library provides and facilitates access to the devices, since we divided the platform into several stages, the connection using SSH should be provided every time, so we separated this function into a single unit to facilitate the development organization, there are also two other units

that we called auxiliary functions which we will explain their development below, each one has a specific role in the main program and the platform stages.

### III.4.1 Performing SSH Connection

In the first attempts, we noticed the need of an SSH connection in each stage depending on the input devices information, which are changeable in each network element, so this option was separated as a standard function to be called up when it needed. In the first script shown in Figure III.2, we defined a function called `ssh_connection ()`, that take as parameter the `device` information's in order to fill in the `info_device` variable dictionary, the IP addresses and the hostnames of the network devices as values. The type of this parameter is a list that is taken from each line in a CSV file shown in Figure III.3, which will be opened on the script once we run the program, we imported the CSV library to get this functionality.

```
16 ▼ def ssh_connection(device):
17     info_device = {
18         'device_type': 'cisco_ios',
19         'ip': device[0],
20         'host': device[1],
21         'username': 'admin',
22         'password': '2020'
23     }
24     try:
25         connection = ConnectHandler(**info_device)
26     except (AuthenticationException):
27         print('Authentication failure: '+ ip)
28     except (NetMikoTimeoutException):
29         print('Timeout to device: ' + ip)
30     except (EOFError):
31         print('End of file while attempting device: '+ ip)
32     except (SSHException):
33         print('Be sure that SSH is enabled in: '+ ip +'?')
34     except Exception as unknown_error:
35         print('Some other error: '+unknown_error)
36     return connection
```

**Figure III. 2:** The source code of the SSH connection function.

As we referred before, this key/value pairs placed in the dictionary are necessary for the `ConnectHandler` module, which is responsible for creating the SSH session in the platform. To perform this, we must import this module from Netmiko library. Based on the device type value, it will choose the accurate Netmiko class specified for Cisco IOS to work with, to fits all the requirements it must take also the IP address, the hostname, the username, and the password values

to specify exactly the targeted devices. In our case, we used a unified username and password that will be prompted in the first time when we run the script for taking advantage also of the platform security using `getpass` library, if it is not necessary to unify them, we can just add the specific username and password of each device in the previous CSV file.

	A	B
1	192.168.122.211	R1
2	192.168.122.202	S2
3	192.168.122.203	S3
4	192.168.122.204	S4
5	192.168.122.205	S5
6	192.168.122.206	S6

**Figure III. 3:** CSV file contains needed devices information.

After we passed the dictionary in the `ConnectHandler ()` with two stars, we implemented some errors handling using a `try/except` block, which will allow us to catch errors quickly, so we will try to connect into the devices and based on some giving exception, we will print a simple statement that will describe the problem rather than breaking the whole program. In the end, and if no errors exist, this script will return the connection that will be necessary on other sections on this platform.

### III.4.2 Getting Devices Version

We knew that we have to interact with different devices types to get an integrated network topology, whereas each type has a specifications that works based on it in term of services and functionalities, so in some cases, switches commands line must be different form routers, this is why we must build the second auxiliary independent function to help in defining these network devices types and set it to the platform at any place where it called, in order to know which direction it should go to and send instructions.

`check_version ()` shown in Figure III.4, is a function that takes connections as a parameter from the previous SSH function, then gives it a standard list of versions that are specific to each type of device, in our experimented GNS3 topology, we used two versions, one for the switch and the other for the router, then from the connection providing as a parameter we send `show version`

command applying the `send_command()` netmiko method, and stored the output in `output_version` variable.

```
40 def check_version(connection):
41     list_versions = ['vios_12-ADVENTERPRISEK9-M', 'VIOS-ADVENTERPRISEK9-M']
42     output_version = connection.send_command('show version')
43     for software_ver in list_versions:
44         int_version = 0
45         int_version = output_version.find(software_ver)
46         if int_version > 0:
47             break
48         else:
49             pass
50     return software_ver
```

**Figure III. 4:** The source code of the check version function.

For the python program to get the version we must loop over each item in the previous list and try to catch it using the pre-built in python `find()` method, with keeping track of the declared integer if it is great than one or not, at the end, this function will return the software version of each network device based on a given list.

### III.4.3 Getting Devices Information as Inputs

The third auxiliary function as we can see it in Figure III.5, is about manually store devices information in the program, rather than getting them from the previous CSV file, as we referred before, this platform must interact with some network devices in some cases because of the different configuration nature from a set to another. It could be a single network device.

`device_input()` is a function with no parameter that returns an array containing network devices information, the hostname and the IP address previously specified for it, each list from this array present a single network device, the trick with this is to prompting the user for it and store them in the `input_list` while we do not enter `n` to break the truth of the while loop, in each loop the script will append the `ip` and the `name` in `input_list` and then will append this last to the `devices_list` variable (lists within a list).

```
54 def device_input():
55     input_list = []
56     devices_list = []
57     while True:
58         ip = input(Back.GREEN + '\nEnter the IP address of the device: '+Style.RESET_ALL)
59         name = input(Back.GREEN + 'Enter the hostname : '+Style.RESET_ALL)
60         input_list.append(ip)
61         input_list.append(name)
62         ask = input("\n Do you want more devices? answer by 'y' or 'n'! : " )
63         devices_list.append(input_list)
64         input_list = []
65         if ask == 'y':
66             continue
67         elif ask == 'n':
68             break
69         else:
70             input("\n Do you want more devices? answer by 'y' or 'n'! : " )
71     return devices_list
```

**Figure III. 5:** The source code of device input function.

These three auxiliary functions will provide the SSH connection between the platform stages function and network elements, also, it helps to get devices type, and specific devices to interact with.

### III.5 NMP Stages Development

In this section, we will explain the four main stages in the proposed NMP, and how those previous auxiliary functions can affect them in order to get administrative tasks done automatically.

#### III.5.1 Network Configuration

When it comes to the configuration tasks in the network infrastructure, we have to select the first option from the network management platform main menu which is simply NETWORK CONFIGURATION, this stage offers an automated operation of pushing command lines via an SSH channel to the network appliances, also it has a unique main function which is *configuration* () show in Figure III.6, that is taken the devices information as a parameter, this function shared between the two options under this stage. The main program will call this single function with just different located input and employing, the first option will take input from the previous CSV file

to address all existed devices, and the second from the data stored manually using *device\_input ()* auxiliary function to address specific devices.

```
75 def configuration(device):
76     ip = device [0]
77     name = device [1]
78     connection = ssh_connection(device)
79     print (Back.GREEN+'\nconnection to '+ name + Style.RESET_ALL)
80     software_ver = check_version(connection)
81     if software_ver == 'vios_l2-ADVENTERPRISEK9-M':
82         print ('Running Switch config file \n')
83         output = connection.send_config_set(switch_config_file)
84         #print(output)
85     elif software_ver == 'VIOS-ADVENTERPRISEK9-M':
86         print ('Running Router config_file \n')
87         output = connection.send_config_set(router_config_file)
88     connection.disconnect()
89
90     return output
```

**Figure III. 6:** The source code of the configuration function.

As we can see the flowchart in the Figure III.7, once we select the first option, the system will give us three choices based on the intended devices for the automation process, whatever the first or the second choice, the program will loop over each device given as input (from the CSV file or prompted insertion) and pass the *configuration ()* main function to perform the desired purpose on each network device. This function shown in Figure III.6, will call the preceding *ssh\_connection ()* function to perform the communication to the network devices and store this on the inserted connection variable, based on it, we will call also the *check\_version ()* to define the version of each one on the platform, if the software version equal to *vios\_l2-ADVENTERPRISEK9 M* which indicate that it is a switch, we will directly pass the switch config file on the *set\_config\_set ()* Netmiko methods which will automatically send the configuration commands down to the previous open SSH channel. If it's equal to *VIOS-ADVENTERPRISEK9-M* which indicate that it is a router, we will directly pass the router config file in the same preceding Netmiko methods.

As we mention in the second chapter, the *send\_config\_set ()* methods, can simply send multiple commands to the networking devices from a text file located along as with the python file in the same repository. Finally, and after we implement this, we must close the connection in a secure manner, using *disconnect ()* Netmiko methods.



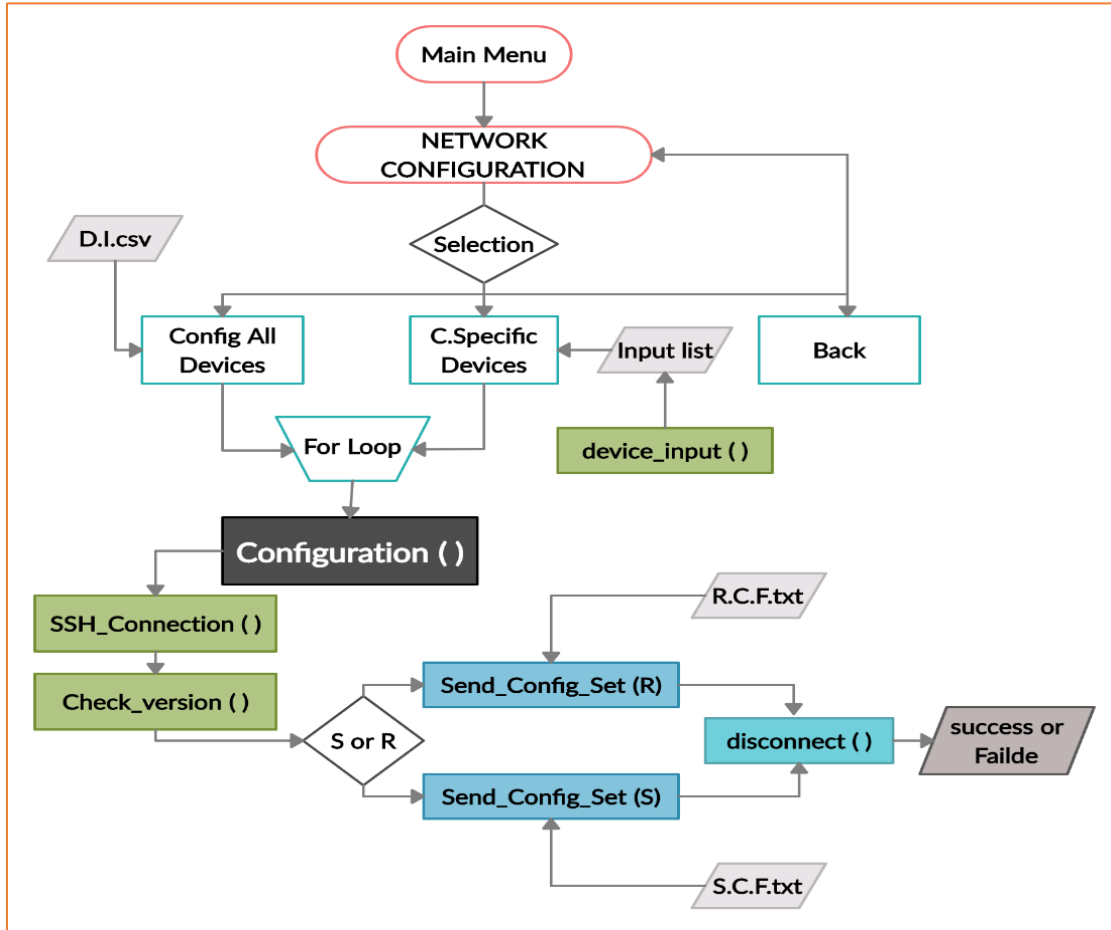


Figure III. 7: Network configuration stage flowchart.

### III.5.2 Network Verification and Test Connection

Usually, when we have finished the process of changing and updating the configuration, we must verify if what we did was done successfully or not, hence, sometimes we send command lines from the file down to the devices but in one way or another an error may occur, devices may not recognize them, or there is a written error in the command itself ... etc. Also, what we need to do is to test the connection and the reachability between the network appliances, a specific group or all networking devices, and this depends on the implementation policies, for that we created the NETWORK VERIFICATION stage. If in the previous stage, we dealt with some or all of the devices, at this stage the same order should be done, just like we see in the earlier and these two options of verification, the same input methods, a for loop over each device, and shared the same *verification ()* function as we can see in the flowchart in the Figure III.8.

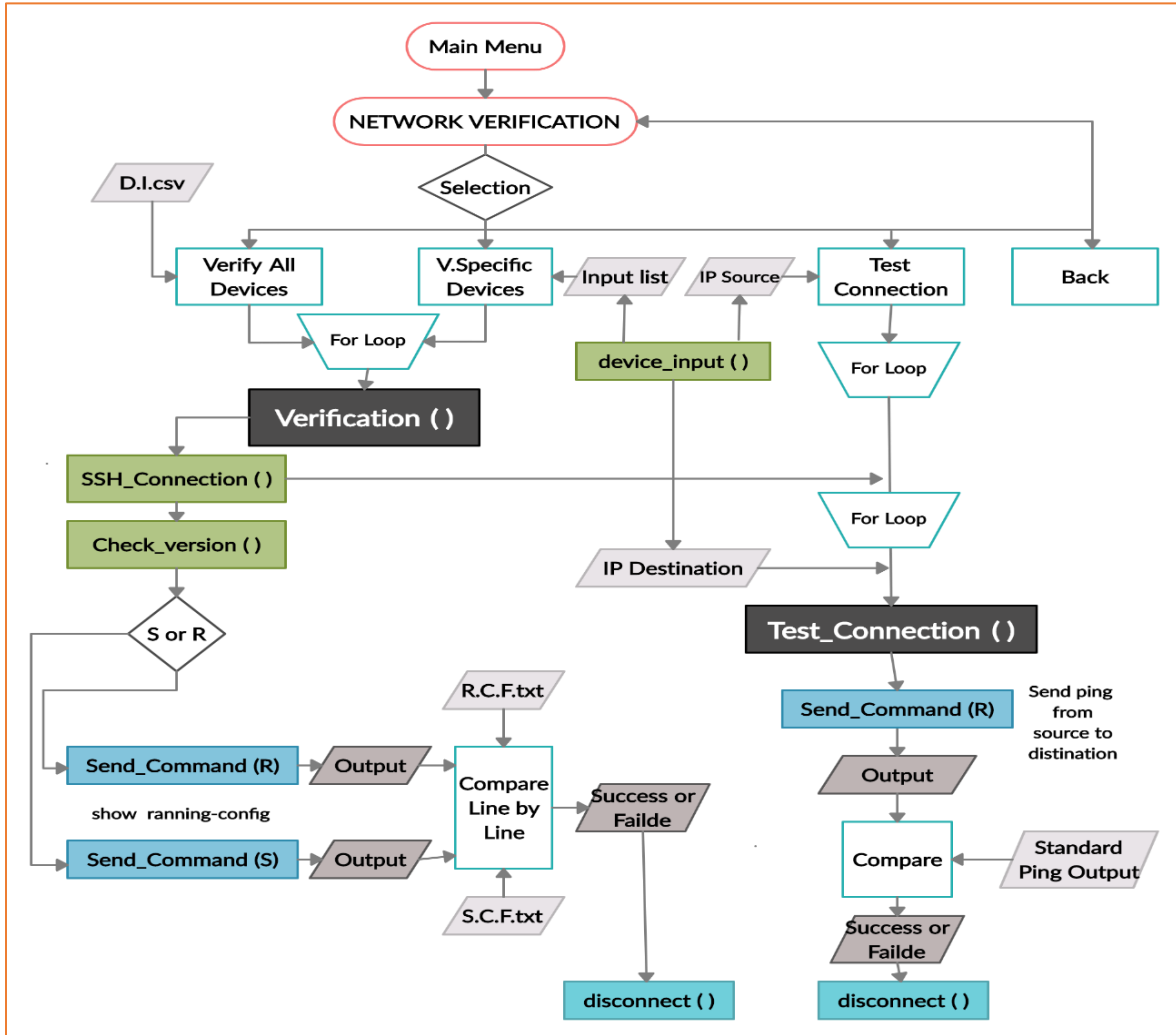


Figure III. 8: Network verification and testing connection stage flowchart.

Once we choose one of the verification options, the platform will call the appropriate function and apply it over all devices given as input, the *verification ()* function will call two auxiliary functions, *ssh\_connection ()* and *check\_version ()* to do the required, a conditional statement was passed to detect the device type in order to select the appropriate configuration file for the routers and switches, after that, we send *show running-config* command down to the selected devices applying the *send\_command ()* Netmiko methods, where we stored the output in *running\_config* string variable, at this point a for loop over each command line on the picked configuration file depending on the device type was used to compare it by the *running\_config* variable, if all found, the program will print out a successful operation, if not, the program will print out in the terminal this specific command line notifying the user that it does not exist. As usual, SSH channel must be closed in each device using *disconnect ()*. The source code of this

function is available in GitHub ([https://github.com/AbdelhakBoumezrag97/NMP\\_Project](https://github.com/AbdelhakBoumezrag97/NMP_Project)).

As we said before, this stage offers other important option, which is the operation of testing the connectivity over the network, under this option the platform will execute an independent function which is `test_connection ()` as we can see in the previous flowchart in Figure III.8, Here, and unlike the other function, this one will take as parameter three variables, the *sources* IP address, the *destinations* IP address, and the *connection* and we must manually give each IP to the program before calling it using the preceding `device_input ()` auxiliary function, once we provide the source IP (it can be more than one), a for loop over each device was applied to get in it using once again the `ssh_connection ()` auxiliary function, under all of this we must provide also the destination IP manually (it can be more than one also), and we apply a for loop again on each destination device, after that we call the `test_connection ()` function shown its source code in the Figure III.9 to do the desired.

```
133 def test_connection(source, destination, connection):
134     ip_source = source [0]
135     name_source = source [1]
136     ip_destination = destination[0]
137     name_destination = destination[1]
138     command = 'ping '+ ip_destination
139     output_ping = connection.send_command(command) #delay_factor = 1)
140     check_list = ['Success rate is 80 percent (4/5)', 'Success rate is 100 percent (5/5)']
141     if any (item in output_ping for item in check_list):
142         print (Back.GREEN+'Connection from '+name_source+' to '+
143               |name_destination+' is reachable==> Success rate'+Style.RESET_ALL)
144     else:
145         print (Back.RED+'Connection from '+name_source+' to '+name_destination
146               +' is unreachable ==> Check Interfaces and protocols !'+Style.RESET_ALL)
```

**Figure III. 9:** The source code of test connection function.

It is already known that in any accessibility test case we should use the `ping` command from each network interface to another, and this what this function does using the `send_command ()` Netmiko methods, from the source device we passing in it the `command` concatenated variable between `ping` and the `destination` IP, after that we passed two standard possibilities in the `check_list` variable in order to compare them by the `output_ping` using a for loop over each item in the list, so if any items exist on the `output_ping`, immediately the connection between the source network device and the destination is reachable, else this condition the connection will not be reachable.

### III.5.3 Network Confirmation and Backups

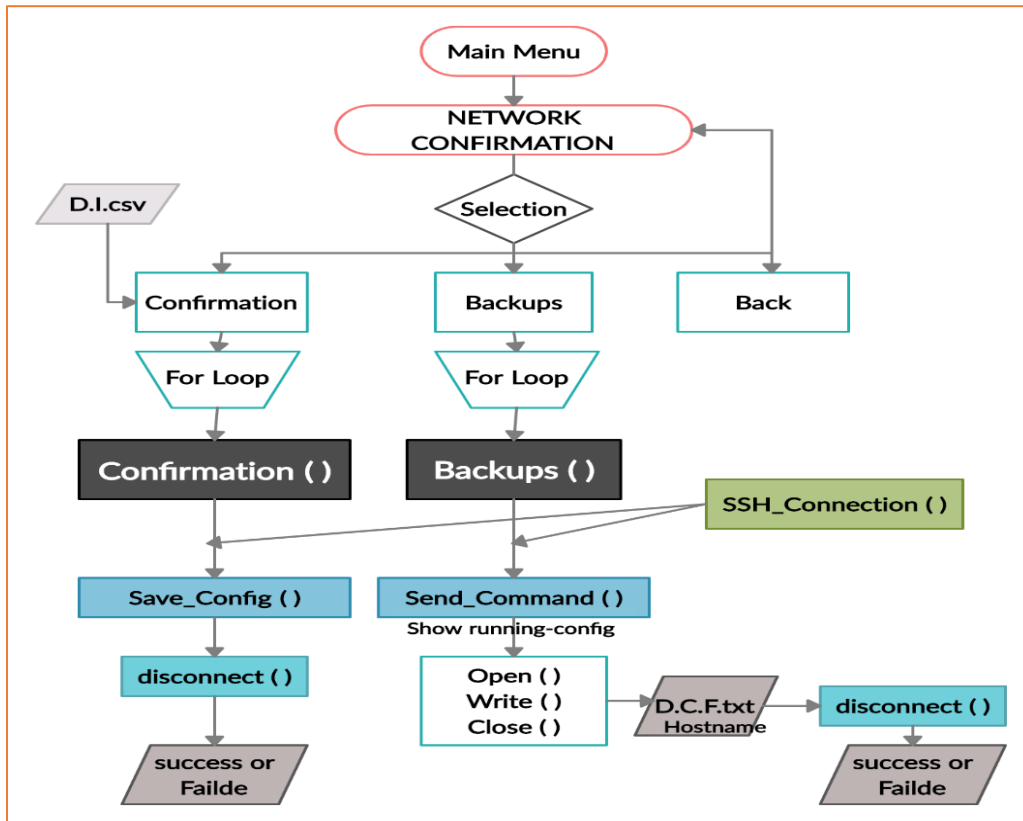
This stage was separated because it has the decision of confirming what we did before, so if the verification process does not match the required need, we can come back to correcting and validating from what we send to the network before, if it matches, we move directly to the confirmation stage, this fully explains the presence of an independent *save\_config ()* Netmiko methods. The stage has two option running two functions as we can see in the flowchart in Figure III.11, the first option is about confirming changes over the network and it runs directly the *confirmation ()* function shown in Figure III.10 over each network device, it will call the *ssh\_connection ()* for opening the SSH channel and applies *Saves\_config ()* method, that will be send simply *write memory* down to all existing devices, after that it will directly disconnect.

```
151 #####
152                                     #CONFIRMATION
153 #####
154 ▾ def confirmation(device):
155     ip = device [0]
156     name = device [1]
157     connection = ssh_connection(device)
158     print (Back.GREEN+'\nconnection to %s ' %name + Style.RESET_ALL)
159     saving = connection.save_config()
160     print(saving + '\n----- Succesful-- Saving-----')
161     connection.disconnect()
162 #####
163                                     #BUCKUPS
164 #####
165 ▾ def backups(device):
166     ip = device [0]
167     name = device [1]
168     connection = ssh_connection(device)
169     Backup = connection.send_command("show running-config")
170     file = open("%s_backup.txt" %name , "w")
171     file.write(Backup)
172     file.close()
173     print(Back.GREEN+"\nBackup for %s is done" %name + Style.RESET_ALL)
174     connection.disconnect()
```

**Figure III. 10:** The source code of confirmation and backups functions.

If we need to extract backups from the network devices to conserve it, so we never lose critical network data of the desired state. we just can improve network security by automating backups and save it in our locale workstation, and this what *backups ()* function under the second option does for us illustrated in Figure III.10, it will be executed over each network device using, as usual, a for loop and calls the *ssh\_connection ()* function again to open the channel and forward

a *show running-config* command using once again *send\_command ()* method, then, we stored the output in a variable in order to extract it in an opened file, it gives a changeable name depending on the device hostname using the prebuilt in Python three methods, which are: *open ()*, *write ()*, and *close ()*, then we have to disconnect the SSH channel down from each network device.

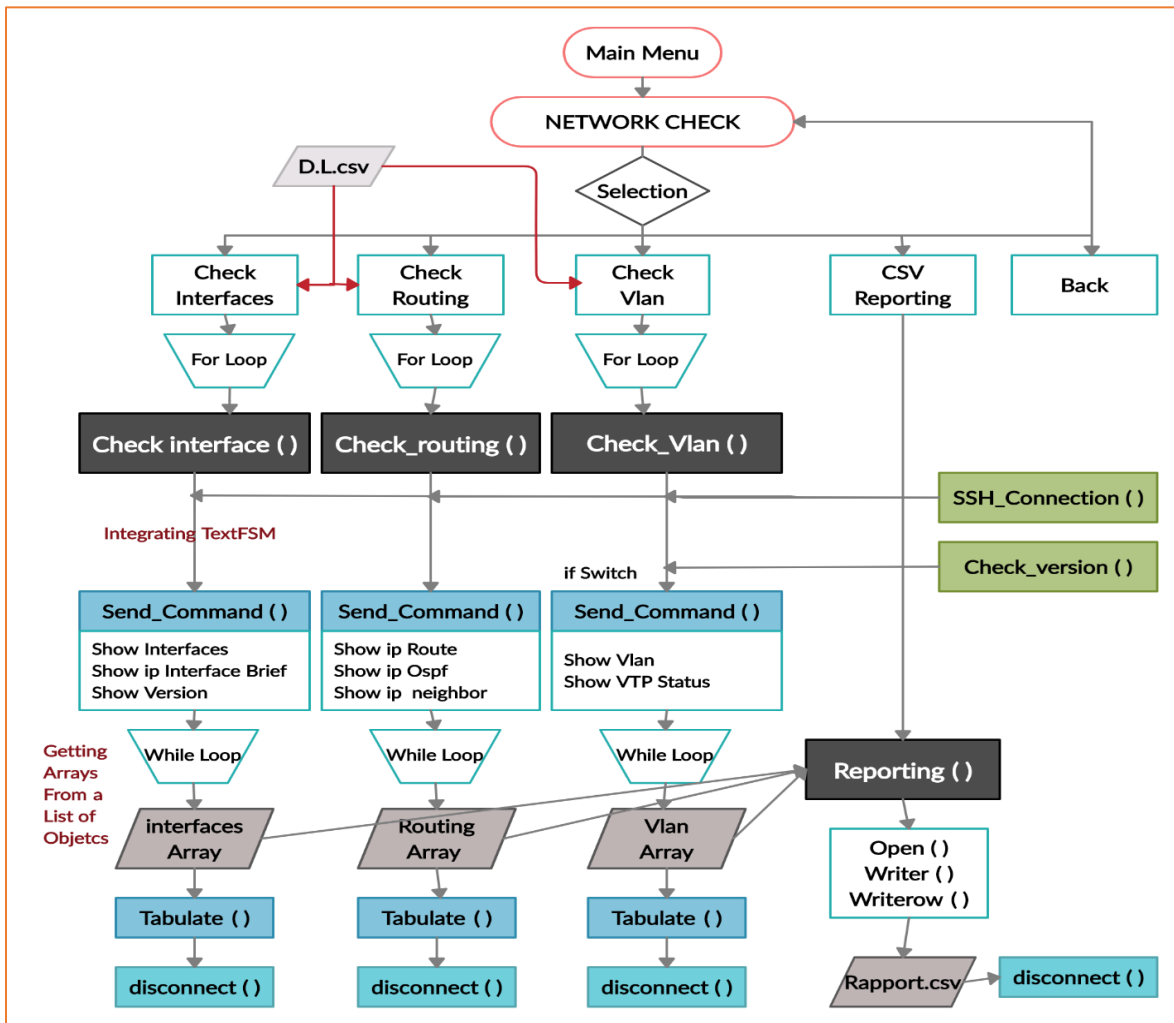


**Figure III. 11:** Network confirmation and backup Stage flowchart.

### III.5.4 Check Network and Report Generation

One of the tedious tasks that a network administrator does in his daily work is check, monitor, and troubleshoot networking devices in order to keep it up in health, for that the last stage was created to provide an automated way instead of the classic method, it has three options that facilitates the checking of interfaces, VLANs, and routing protocols using three other functions and they are as follows: *check\_interfaces ()*, *check\_vlan ()*, *check\_routing ()* as we can see in the flowchart depicted in Figure III.12. Each function will be applied on each network device using a for loop and the *ssh\_connection ()* to open the channel again, also, they send a bunch of show commands which will fulfill the purpose for the case of the tasks mentioned in the beginning.

Hence, if we decide to check interfaces the program will automatically send *show interfaces*, *show ip interface brief*, and *show version* to all the network devices, and if we decide to check routing protocol the program will automatically send *show ip route*, *show ip ospf database*, and *show ip ospf neighbor* to all the network devices once again, and if choose the third option which is checking VLAN, here, and after we check the version the program will send *show vlan*, and *show vtp status* over only the switches because router does not run VLANs. We integrated TextFSM templates (*ntc\_template*) that we explained in the second chapter, the reason from that is picking up from each output structured data some important information and stored in an arrays which will help us to convert it into easy and readable spreadsheets and print them directly in the terminal using *tabulate ()* method from the imported tabulate library.



**Figure III. 12:** Checking network and report generation stage flowchart.

The fourth option provides an automated workflow that will do the work of reports

generating and collecting data for us instead of logging into an application and running reports every time, once we choose this option it will immediately run the fourth function that we call it *reporting ()* which will take advantage from the output of the previous three checking functions, this function will open a CSV file which is the one of best way to store extracted data, and write each list from the output preceding arrays in one line to reshape the previous spreadsheets using *writerow ()* method, also we can add standard information like the date, the time, and separate sections by notes. At the end, we get one CSV file that contains all collected data and it will be ready to read and check. The source code of these functions are available in GitHub ([https://github.com/AbdelhakBoumezrag97/NMP\\_Project](https://github.com/AbdelhakBoumezrag97/NMP_Project)).

### III.5.5 Multithreading Integration

we discussed before in the second chapter, that we can improve the speed in the execution time using multithreading, we added this functionality in our platform by just importing “Thread” module then we call it on each network device item under the for loop, passing the targeted function which is *configuration ()* in our example, and arguments to create each device thread individually, then we append every single thread in an empty list of *threads* variable in order to store them, by calling the *start ()* methods that will start the execution of each thread instance, then a for loop must apply on each object device to call the *join ()* methods that will wait until the thread terminates execution. We are setting up the time in order to calculate the time spent in running this part, Figure III.13, show as the configuration function integrated with the threading functionality.

```
startTime = time()
threads=[]
for device in devices_list:
    t = Thread(target=configuration, args= (device,))
    t.start()
    threads.append(t)
for t in threads:
    t.join()
print(time() - startTime)
```

**Figure III. 13:** The configuration function integrated with threading functionality.

In the coming sections, we are going to implement and test one by one the previous stages of this management platform with at least one example, using threading, surely after building a campus network use case in GNS3, and explore what's behind the result in term of performance and efficiency in different corners.

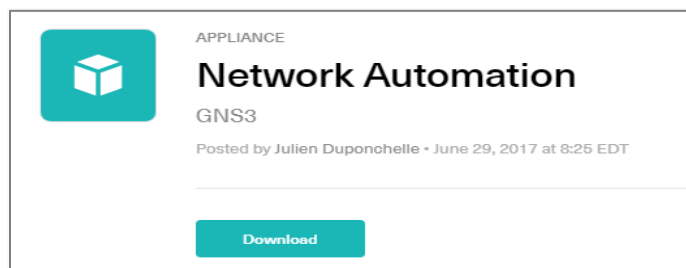
### III.6 Use Case Testing with NMP on GNS3

The test operation of this platform was partial, meaning that when we get a separated code part completed we get in the test directly on one or more network device as needed in GNS3 in order to iterate and hide the development complexity, once we get the complete platform ready, we built a real case network then we examined all the part and stages to see what it can add as values, after discussing the results.

#### III.6.1 Environment Setup

It is essential to have a testing environment that is as close as possible to the reality in order to obtain realistic results when experimenting with the network management platform, so as we mentioned in the introduction we used GNS3 emulator which is a tool for designing, testing and troubleshooting complex networks, capable now also to connect to external networks and allowing integration with virtual images or Docker Containers. It extremely helps in testing and developing network automation software.

After providing the requirements listed before we configured the environment to be ready for the use, all what we need is to uploaded Cisco images, in our case we used an *IOSv-15.6 (2) T* which is a VIRT image released for a Cisco router, and *IOSv12-15.2.1* which is an image released for layer two Cisco switches. Cisco Virtual Internet Routing Lab or VIRT is a powerful technology, easy to use, extensible network modeling and simulation environment, they are recommended for using in GNS3 because they are much like real IOS [57]. Then we downloaded the *Network automation Docker container* appliance from the GNS3 marketplace that can be seen in Figure III.14, it is a lightweight, standalone, executable package of software that includes everything needed to run an application, and it comes with python and some popular tools used for network automation and programmability like Netmiko, NAPALM, and Ansible preinstalled.



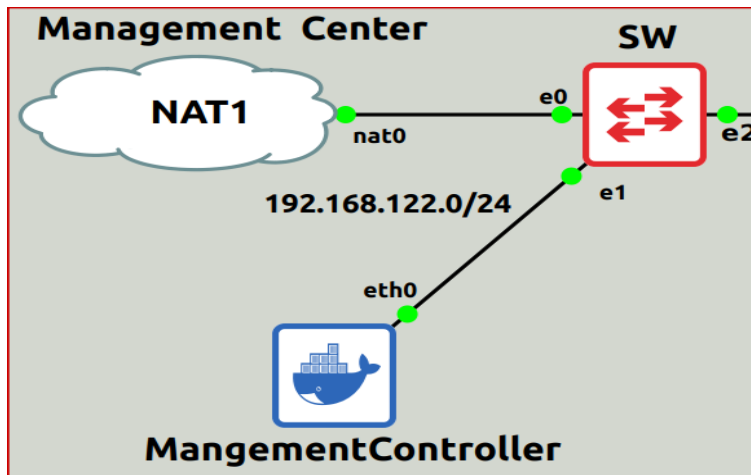
**Figure III. 14:** The network automation appliance in GNS3 marketplace.



In order to prove how we can successfully set a programmable test on the devices using the network management platform, we created a use case network topology. The coming section will explain the steps to build a simple campus network use case scenario with only CISCO IOS devices routers and switches.

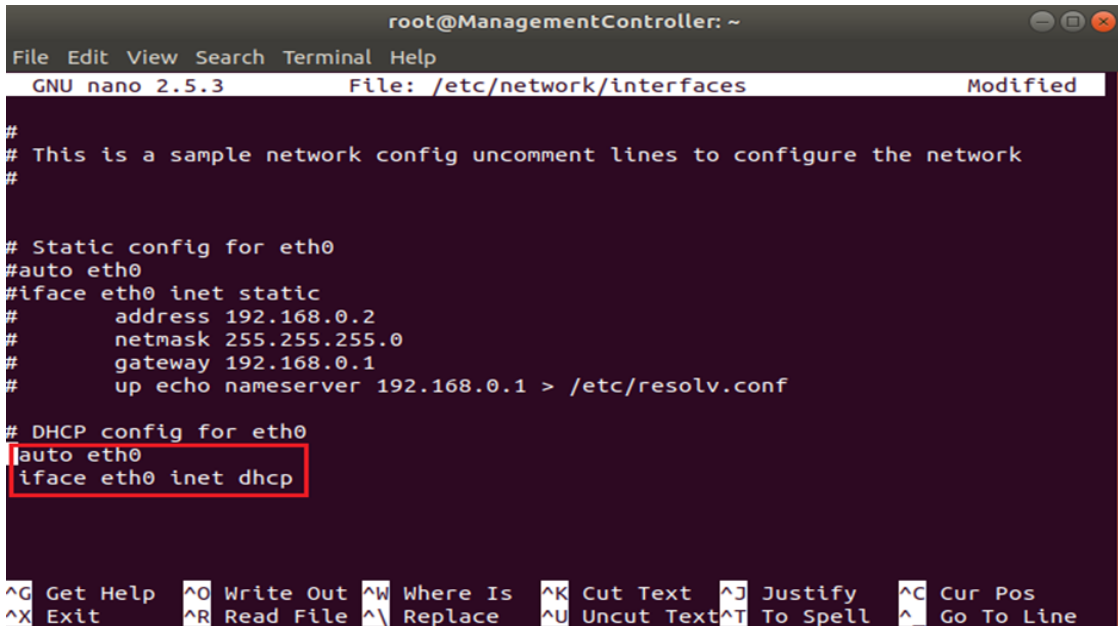
### III.6.2 Network Topology Setup

After we installed the *network automation Docker Container* in GNS3 that make the process of testing very easy, we dragged to the work space, it acted as management controller in central point, then we drag an Ethernet switch with the NAT node that allowed the Docker for getting an IP address, also having internet connectivity from to local computer, because by default the NAT node runs a DHCP server with a predefined pool in the **192.168.122.0/24** range. Figure III.15, shows as these components.



**Figure III. 15:** Docker container connected to the NAT node on the workspace in GNS3

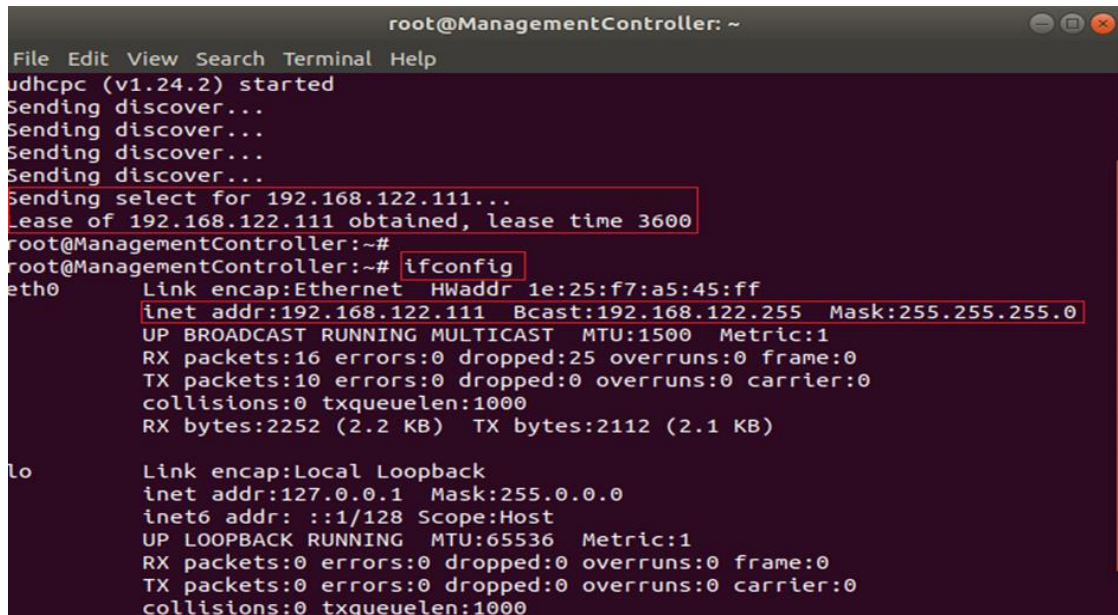
When we started up the topology, we opened up the console from the Docker container once it booted up, then we set it up to use DHCP, so `nano /etc/network/interfaces` command was passed into the console in order to edit this file and uncomment the last two lines then saved as we can see in Figure III.16.



```
root@ManagementController: ~
File Edit View Search Terminal Help
GNU nano 2.5.3 File: /etc/network/interfaces Modified
#
# This is a sample network config uncomment lines to configure the network
#
# Static config for eth0
#auto eth0
#iface eth0 inet static
#   address 192.168.0.2
#   netmask 255.255.255.0
#   gateway 192.168.0.1
#   up echo nameserver 192.168.0.1 > /etc/resolv.conf
# DHCP config for eth0
#auto eth0
#iface eth0 inet dhcp
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line
```

**Figure III. 16:** Configuring the Docker network interface using Nano editor.

After that, we reloaded the Docker node, to allow the DHCP client on it for sending discover messages to the NAT node that will offer an IP address, as we can see in Figure III.17. To confirm *ifconfig* command must be passed in the console.



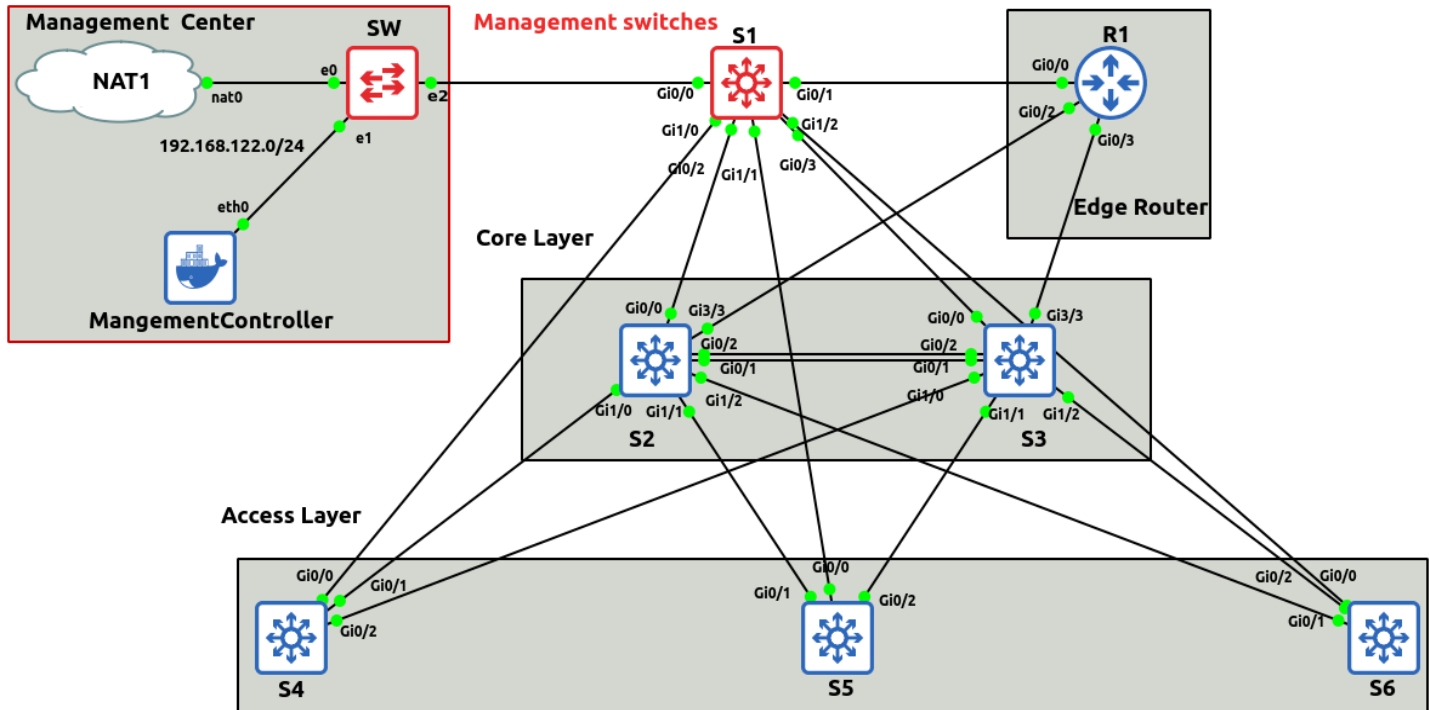
```
root@ManagementController: ~
File Edit View Search Terminal Help
dhclient (v1.24.2) started
Sending discover...
Sending discover...
Sending discover...
Sending discover...
Sending select for 192.168.122.111...
Lease of 192.168.122.111 obtained, lease time 3600
root@ManagementController:~#
root@ManagementController:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 1e:25:f7:a5:45:ff
          inet addr:192.168.122.111 Bcast:192.168.122.255 Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:16 errors:0 dropped:25 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2252 (2.2 KB)  TX bytes:2112 (2.1 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
```

**Figure III. 17:** Docker obtained an IP address from DHCP. Confirm using *Ifconfig*.

As we mentioned above, the devices we used to automate and experiment on them is Cisco VIRL, *IOSvL2* switches and Cisco *IOSv* routers, so we added six switches *S1* to *S6*, and one edge

router *R1* on the previous topology, and connecting all of the nodes by virtual cabling available on GNS3 in a very similar way to the hierarchical Cisco design campus networks, we assumed that we were using an out-of-band management network which mean separating the management traffic from the production traffic, the problem in in-band management network is that we were had issues where sometimes spanning tree and other protocols blocked links and stopped the script communicating with network devices, so in this topology shown in Figure III.18, the downlinks of *S1* to the other switches are part of management network and other links between the other devices are data network, we were assumed that *S2* and *S3* are core switches, and that *S4* to *S6* are access switches, also we placed *R1* as an edge router. Figure III.18, depicts the entire topology.



**Figure III. 18:** The experimented campus network on GNS3.

Now, after we had a linked network we pushed some commands to get initial settings on the devices, so on each device in this network we created management interface, and specify an IP address in the same network subnet of the management controller to reach out the connectivity from it, also enabling SSH client to accept remote connection via the terminal and set credentials information. The following configuration steps were used in all of the devices:

### 1- Configuring management interfaces and set IP addresses:

### a- On the switches:

```
S1 (config) #interface vlan1
```

```
S1 (config-if) #ip address 192.168.122.201 255.255.255.0
```

```
S1 (config-if) #no shutdown
```

The same commands passed, but the IP address schema used is the following:

- **S2:** 192.168.122.202
- **S3:** 192.168.122.203
- **S4:** 192.168.122.204
- **S5:** 192.168.122.205
- **S6:** 192.168.122.206

### b- On the router:

```
R1 (config) #interface Gigabit-Ethernet 0/0
```

```
R1 (config-if) #ip address 192.168.122.211 255.255.255.0
```

```
R1 (config-if) #no shutdown
```

### 2- Configuring credential information used for accessing these devices:

```
S (config) #username admin privilege 15 password 2020
```

```
S (config) #enable password cisco
```

```
S (config) #hostname [from 1 to 6]
```

### 3- Set an enable password in all devices:

```
S (config) #enables password cisco
```

### 4- Set up an SSH client on all devices:

```
S (config)#ip domain-name cisco.com
```

```
S (config) #crypto key generate rsa
```

```
S (config) #ip ssh time-out 120
```

```
S (config) #line vty 0 4
```

```
S (config-line) #login local
```

```
S (config-line) #transport input all
```

Once we did that, we had to verify the accessibility between the management controller and all of the devices on the network topology using SSH protocol, the command `ssh admin@IP-add` was used for testing remote accessing from the Docker terminal, only generating the proper output if the connection to the device is established. Figure III.19, illustrate to us an example of S4 switch.



```
root@MangementController: ~
File Edit View Search Terminal Help
root@MangementController:~# ssh admin@192.168.122.202
*****
* IOSv is strictly limited to use for evaluation, demonstration and IOS *
* education. IOSv is provided as-is and is not supported by Cisco's *
* Technical Advisory Center. Any use or disclosure, in whole or in part, *
* of the IOSv Software or Documentation to any third party for any *
* purposes is expressly prohibited except as otherwise authorized by *
* Cisco in writing. *
*****Password:
*****
* IOSv is strictly limited to use for evaluation, demonstration and IOS *
* education. IOSv is provided as-is and is not supported by Cisco's *
* Technical Advisory Center. Any use or disclosure, in whole or in part, *
* of the IOSv Software or Documentation to any third party for any *
* purposes is expressly prohibited except as otherwise authorized by *
* Cisco in writing. *
*****
S2#
S2#
S2#
```

**Figure III. 19:** Example of verifying remote access on S2, using SSH protocol.

It may appear another window asking us to confirm by yes, which is normal, or a window that warns us when we change our credential information or we updated the SSH key and says that: *REMOTE HOST IDENTIFICATION HAS CHANGED!* In this case we just use `ssh-keygen` to delete the invalid key and generate another using this command:

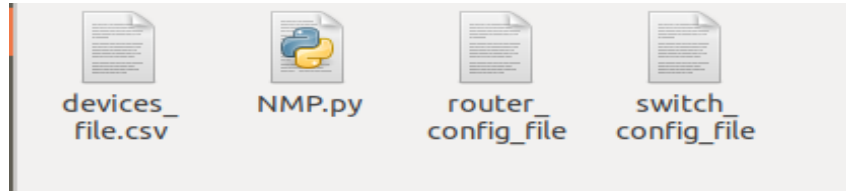
- `ssh-keygen -R "hostname or IP"`

After confirming a successful login on all devices, we passed directly into testing the different stages with evaluation and results analyzing.

### III.6.3 Use Case Testing and Results

As we referred before, we built this platform to deal with files from the local laptop alongside the Python file, in Figure III.20, we can see three essential files, the one with a CSV extension is constant and it is similar to the one in Figure III.2, it contains devices information from the IP

address to the host name.



**Figure III. 20:** Needed files to execute the NMP in local laptop

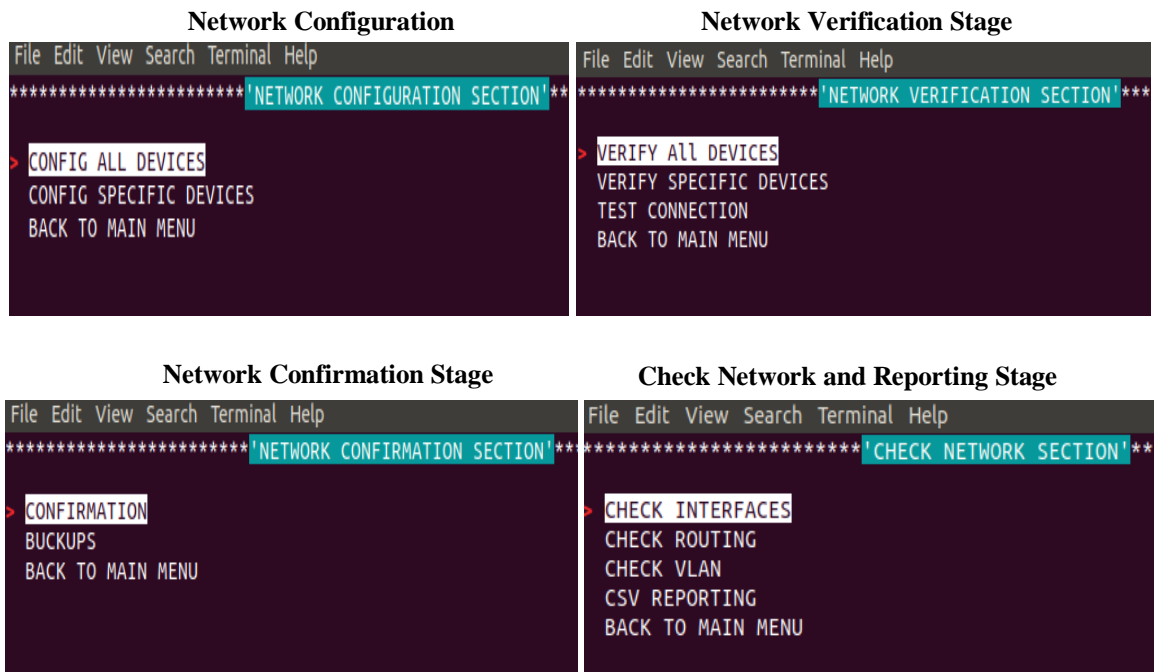
The two other text file contain all commands line that we will pushed as configuration, one for the switches and one for the routers, it is changeable based on what we need to configure on the day task or in one round, the following table presents what we send as commands over the network in this test, we assumed that those would achieve the desired network state, as example we choose to configure IP addresses on the router interfaces, and setting up the NAT service between the outside and the inside interfaces with the default route to the external IP address, in the switches also we setting up services like NTP, switch port trunk, and OSPF routing protocol, the complete configuration in these two files shown in Table III.1.

**Table III. 1:** Configuration files content on the test case.

Switch Config File	Router Config File
ip name-server 8.8.8.8 ntp update-calendar clock timezone PST -8	interface GigabitEthernet0/2 ip address 10.10.10.2 255.0.0.0 no shutdown
int range g0/1 - 2 switchport trunk encapsulation dot1q switchport mode trunk switchport nonegotiate switchport trunk allowed vlan all	interface GigabitEthernet0/3 ip address 11.10.10.2 255.0.0.0 no shutdown
int range g1/0 - 3 switchport trunk encapsulation dot1q switchport mode trunk switchport nonegotiate switchport trunk allowed vlan all	interface GigabitEthernet0/1 ip address 192.168.1.42 255.255.255.0 no shutdown ip nat outside
	interface GigabitEthernet0/0 ip nat inside ip nat inside source list 1 interface GigabitEthernet0/1 overload access-list 1 permit any

	<pre>ip route 0.0.0.0 0.0.0.0 192.168.1.0 ip name-server 8.8.8.8  router ospf 100  network 10.0.0.0 0.255.255.255 area 0  network 11.0.0.0 0.255.255.255 area 0</pre>
--	---

Now, after we prepared the configurations files in the correct order, we ran the python file, then we passed the username and password of the network administrator in the terminal, that are identical for accessing network devices from the SSH client, we faced the network management platform in the terminal window as Figure II.9 depicts, just by up and down selection using the keyboard and hitting enter, we got sub-menu for each stage with options shown in the Figure III.21.



**Figure III. 21:** Sub-menus of the NMP stages shown in the terminal.

### 1- Network Configuration Stage Testing:

#### a- Config all devices option:

In the configuration stage testing, we firstly selected *config all devices* option in order to loop over each device exist in the CSV file from the main program, and run the *configuration ()* function,

with setting up the time to calculate the duration, the result would be the output in the terminal window as we can see in Figure III.22, that illustrate element by element the progress of the execution, expressed by print statements in a nice view using *Colorama* module, The first capture in the left represents the result of the configuration process over six devices without multithreading integration. The capture on the right depicts the same process, but with parallel execution using multithreading integration.

Without Multithreading	With Multithreading
<pre>File Edit View Search Terminal Help *****'NETWORK CONFIGURATION SECTION'***** Config All Devices Has Been Selected connection to R1 is up Running Router config_file ... connection to S2 is up Running Switch config file ... connection to S3 is up Running Switch config file ... connection to S4 is up Running Switch config file ... connection to S5 is up Running Switch config file ... connection to S6 is up Running Switch config file ... time in second is = 46.076061725616455</pre>	<pre>File Edit View Search Terminal Help *****'NETWORK CONFIGURATION SECTION'***** Config All Devices Has Been Selected connection to R1 is up connection to S3 is up connection to S2 is up connection to S5 is up connection to S6 is up connection to S4 is up Running Router config_file ... Running Switch config file ... Running Switch config file ... Running Switch config file ... Running Switch config file ... Running Switch config file ... time in second is = 9.472570657730103</pre>

**Figure III. 22:** Configuration stage, execution progress, with and without multithreading.

The output on the left indicates the sequential assignment, which means going to each device CLI individually after providing SSH channel, then running the associated *config\_file* depending on the device type in a single thread. The total duration was 46 seconds with an average of 7.67 seconds per device, which is good comparing with the manual configuration, but it will be very inefficient when our complexity and scalability grows.

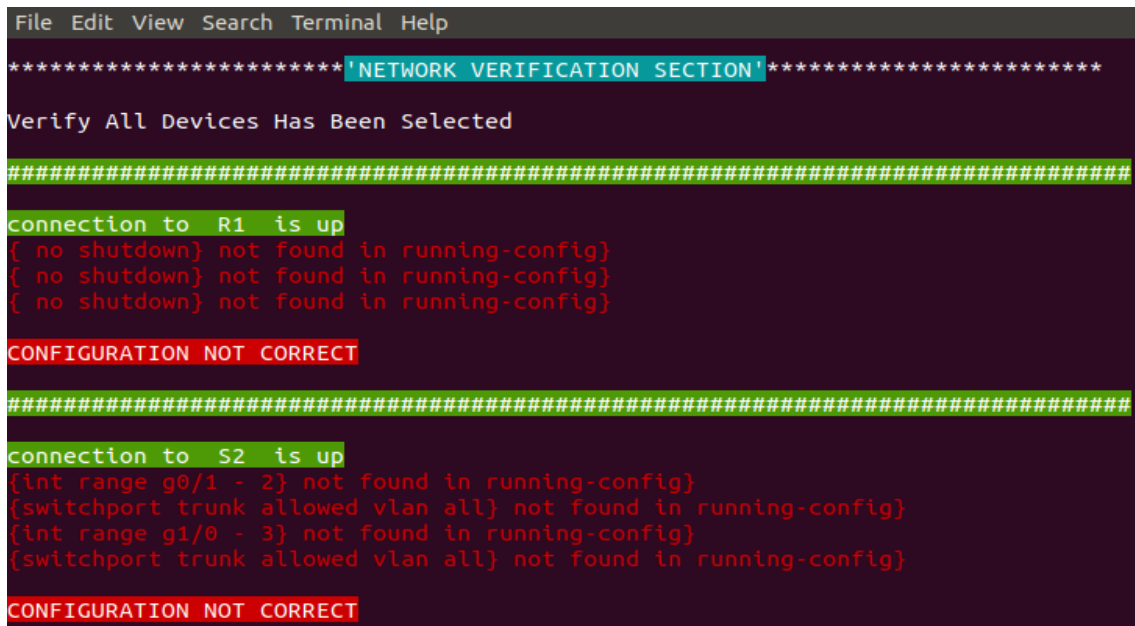
The output on the right indicate the parallel assignment, which means going to all devices CLI at once, then running the associated *config\_file* in a multiple thread. The total duration was 9.5 seconds with an average of 1.6 seconds per device, which is good comparing with the previous state.



### 2- Network Verification Stage Testing:

#### a- Verify all devices option:

In this test, *verify all devices* option has been selected to verify if the configuration done correctly using the *verification ()* function, which will take the previous inputted router and switch configuration files and compare them with the running-config in network devices, in the figure III.23, we can see the output in the terminal capturing the verification process for R1 and S1.



```
File Edit View Search Terminal Help
***** 'NETWORK VERIFICATION SECTION' *****
Verify All Devices Has Been Selected
#####
connection to R1 is up
{ no shutdown} not found in running-config
{ no shutdown} not found in running-config
{ no shutdown} not found in running-config
CONFIGURATION NOT CORRECT
#####
connection to S2 is up
{int range g0/1 - 2} not found in running-config
{switchport trunk allowed vlan all} not found in running-config
{int range g1/0 - 3} not found in running-config
{switchport trunk allowed vlan all} not found in running-config
CONFIGURATION NOT CORRECT
```

**Figure III. 23:** Execution progress of *Verify all devices* option.

The output indicates that the previous configuration stage is not correct in R1 and S1 as well, actually, it is not correct in all network devices from the code logic perspective. After checking the configuration files again, we noticed that *no shutdown* command cannot be found in the running-config by default, but it was passing correctly through each CLI, also, when we send command shortcuts, like *int range g0/1 – 2*, which stands for *interface range GigabitEthernet 0/1 -2*. The logic of the code takes each command-line exists in the config-files as a string, and then searches for it in the running-config returned to the program as a block of string as well, hence, any fault syntax in those commands can make those kinds of errors that appear in the terminal, even if it was understandable for the CLI and passed as correct command. In this case, we must add exceptions to the *verification ()* function to skip those few commands and errors.

### b- Test connection option:

In this test, *test connection* option has been selected, to test the connectivity between network devices, we gave the required source and destination IP addresses to the platform, using *device\_input ()* auxiliary function, and then applying the main *test\_ connection ()* function over them, just like we see in the Figure III.24 and Figure III.25.

```
File Edit View Search Terminal Help
***** 'NETWORK VERIFICATION SECTION' *****
Test Connection Has Been Selected
Get source ip =>:
Enter the IP address of the device: 192.168.122.211
Enter the hostname : R1
Do you want more devices? answer by 'y' or 'n'! : y
Enter the IP address of the device: 192.168.122.201
Enter the hostname : S1
Do you want more devices? answer by 'y' or 'n'! : n
Get destination ip =>:
Enter the IP address of the device: 192.168.122.204
Enter the hostname : S4
Do you want more devices? answer by 'y' or 'n'! : y
Enter the IP address of the device: 192.168.122.205
```

**Figure III. 24:** Manually entering Source and destination IPs.

The capture depicts that we entered two sources IP address and hostname; R1 and S1, and three destinations IP address and hostname; S4, S5, and S10 (notice that S10 does not exist on the topology, we entered it just to check the validity of the code). Once we enter *n* for no more devices we get this terminal window shown in the Figure III.25.

```
Connection to R1
Connection from R1 to S4 is reachable==> Success rate
Connection from R1 to S5 is reachable==> Success rate
Connection from R1 to S10 is unreachable ==> Check Interfaces and protocols !
Connection to S1
Connection from S1 to S4 is reachable==> Success rate
Connection from S1 to S5 is reachable==> Success rate
Connection from S1 to S10 is unreachable ==> Check Interfaces and protocols !
```

**Figure III. 25:** Testing test connection process and the output result.

After the `ssh_connection ()` function was implemented over each source IP, the program sent `ping` statement from it to the destination devices. The output illustrates that the connectivity from R1 to S4 and S5, was reachable, and from R1 to S10 was not reachable which make sense as long as does not exist on the topology.

**Note:** The test of the two options under the previous stages, Config and Verify specific devices was passed successfully with the same steps, the only difference was with integrating `device_input ()` function rather than taking it from the CSV file, to interact with only those entered devices.

```
*****'NETWORK CONFIGURATION SECTION'*****
Config Specific Devices Has Been Selected
Enter the IP address of the device: 192.168.122.204
Enter the hostname : S4

Do you want more devices? answer by 'y' or 'n'! : y

Enter the IP address of the device: 192.168.122.205
Enter the hostname : S5

Do you want more devices? answer by 'y' or 'n'! : y

Enter the IP address of the device: 192.168.122.206
Enter the hostname : S6

Do you want more devices? answer by 'y' or 'n'! : |
```

**Figure III. 26:** Entering device information manually for specific device interaction.

### 3- Network Confirmation Stage Testing:

#### a- Confirmation option:

In the confirmation stage testing, we firstly selected the `confirmation` option, and we directly got this output shown is the Figure III.27.

As we said before, this option is about running the `confirmation ()` function over each network device on the topology, as the figure depicts, R1, and S1 are copying their running-config to the startup-config using `write memory` command successfully.

```
File Edit View Search Terminal Help
*****NETWORK CONFIRMATION SECTION*****
Confirm All Devices Has Been Selected

connection to R1 is up
write mem
Building configuration...

[OK]
R1#
----- Successful-- Saving-----

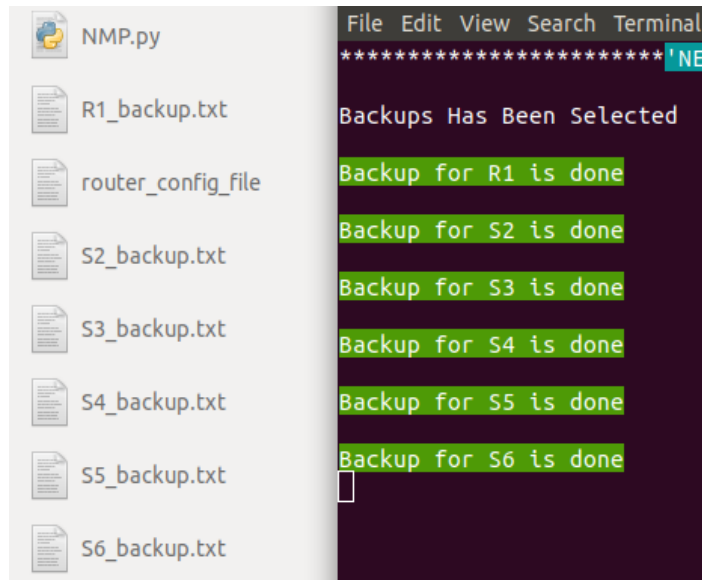
connection to S2 is up
write mem
Building configuration...

Compressed configuration from 4295 bytes to 1952 bytes[OK]
S2#
----- Successful-- Saving-----
□
```

**Figure III. 27:** Testing the confirmation option and the terminal output.

### b- Backups option:

Secondly, we have chosen *backups* option, and we directly got this output shown in the local directory and in the terminal as we can see in the Figure III.28.



**Figure III. 28:** Testing backups option and the output in the terminal and in local directory.

The *backups ()* function generate automatically six configuration backup text files, for the safety in which we assumed that they were achieved the desired state of the network topology, stored in the local directory alongside with the NMP python file.

### 4- Check Network Stage Testing:

#### a- Check interface, routing, VLAN options:

In this test, these options; checking interfaces, routing and VLAN of the fourth stage were selected, then we got successively those three spreadsheets of network devices contains a very useful information and statistics when it comes to the monitoring and troubleshooting process, presented in Figure III.29, 30, 31. Each option ran its own function integrated with TextFSM templates (*ntc\_template*).

In the first spreadsheet, we've got information about each interface on each device on the topology, the interface name, IP address, protocol, status, uptime, in and out packets, in and out errors. All are brief and organized key information.

```
*****'CHECK NETWORK SECTION'*****
Check interfaces Has Been Selected
connection to R1 is up
```

R1=>15.6(2)T	Interface	IP-Address	Protocol	Status	Uptime	In Error	In Pckts	Out Error	Out Pckts
R1	Gi0/0	192.168.122.211	up	up	2 hours, 28 minutes	0	6000	0	6366
R1	Gi0/1	192.168.1.42	down	down	2 hours, 28 minutes	0	0	0	0
R1	Gi0/2	10.10.10.2	up	up	2 hours, 28 minutes	0	6	0	513
R1	Gi0/3	11.10.10.2	up	up	2 hours, 28 minutes	0	6	0	513
R1	NVI0	192.168.122.211	up	up	2 hours, 28 minutes	0	0	0	0

**Figure III. 29:** Check interfaces option testing and the output.

In the check routing option output shown in Figure III.30, we've got information about routing protocol implemented on the topology, in this case we have OSPF up and running, this information modeled in two separated spreadsheets, the first is about routing table, and the second is for OSPF database.

## Chapter III: Network Management Platform for Network Automation

Check Routing Has Been Selected

connection to R1 is up

Ospf R-table=>R1	Network	Mask	Next-Hop	Protocol	Neighbor
	10.0.0.0	8	GigabitEthernet0/2	C	No neighbor
	10.10.10.2	32	GigabitEthernet0/2	L	No neighbor
	11.0.0.0	8	GigabitEthernet0/3	C	No neighbor
	11.10.10.2	32	GigabitEthernet0/3	L	No neighbor
	192.168.122.0	24	GigabitEthernet0/0	C	No neighbor
	192.168.122.211	32	GigabitEthernet0/0	L	No neighbor

Ospf Data-base=>R1	adv_router	age	area	link_count	link_id	process_id	router_id
	192.168.122.211	1521	0	2	192.168.122.211	100	192.168.122.211

connection to S2 is up

**Figure III. 30:** Check routing option testing for OSPF protocol and the output.

The last checking was concerned about VLAN, specific only for the switches, as we can see in the spreadsheet depicted in Figure III.31, we've got information about the state of each VLAN on each switch, like the VLAN name, ID, and all interfaces associated with it.

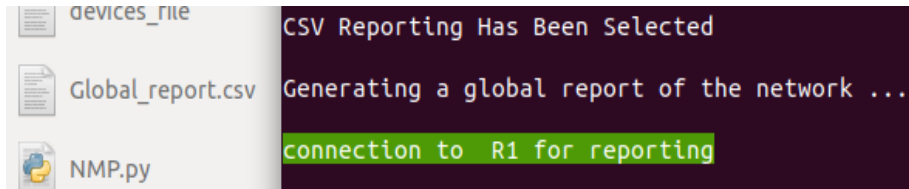
connection to S6 is up

S6	Interfaces	Name	Status	Vlan id	VTP-Mode
	Gi0/0,Gi0/3,Gi1/0,Gi1/1,Gi1/2,Gi1/3,Gi2/0,Gi2/1,Gi3/0,Gi3/1,Gi3/2,Gi3/3	default	active	1	Transparent
	Gi2/2,Gi2/3	Data	active	100	Transparent
	Gi2/2,Gi2/3	Voice	active	101	Transparent
		Test	active	102	Transparent
		fddi-default	act/unsup	1002	Transparent
		token-ring-default	act/unsup	1003	Transparent
		fddinet-default	act/unsup	1004	Transparent
		trnet-default	act/unsup	1005	Transparent

**Figure III. 31:** Check VLAN testing and output.

**b- CSV reporting option:**

Once we selected this last option, we've got the *Global\_report.csv* file in the local directory, and indications about the execution progress in the terminal, as we can see in Figure III.32, after the *reporting ()* function done from the execution, we opened the output file using Excel, which we depicted in Figure III.33.



**Figure III. 32:** CSV report output in local laptop and in the terminal.

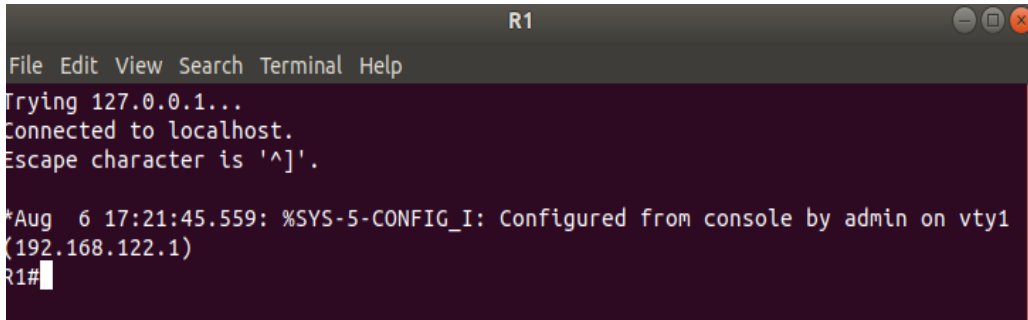
As we mention before, this option is based on the execution outputs of the previous function; *Chek\_interface ()*, *Check\_routing ()*, and *check\_vlan ()*, in which this option function stored the previous spreadsheet on separated CSV file successively, with adding some information like the time and the spreadsheet names.

A	B	C	D	
1	CHECKING NETWORK IN:	2020-08-07 00:25:11.721018		
2				
3				
4	THIS IS A SPREADSHEET	TO GET INTERFACES <u>INFOS</u> FROM R1		
5				
6	R1=>15.6(2)T	Interface	IP-Address	Protocol
7	R1	Gi0/0	192.168.122.211	up
8	R1	Gi0/1	192.168.1.42	down
9	R1	Gi0/2	10.10.10.2	up
10	R1	Gi0/3	11.10.10.2	up
11	R1	NV10	192.168.122.211	up
12				
13	THIS IS A SPREADSHEET	TO GET ROUTING <u>INFOS</u> FROM R1		
14				
15	Ospf R-table=>R1	Network	Mask	Next-Hop
16		10.0.0.0		8 GigabitEthernet0/2 C
17		10.10.10.2		32 GigabitEthernet0/2 I

**Figure III. 33:** Global report spreadsheet opened with Excel .

To be sure that the previous instructions from the NMP were implemented successfully on each network device, we opened the console after each execution and we noticed the Syslog messages indicates that the admin was connected in a specific time, as Figure III.34 illustrate. As well as expected changes on the devices, verified using show commands with the normal way.

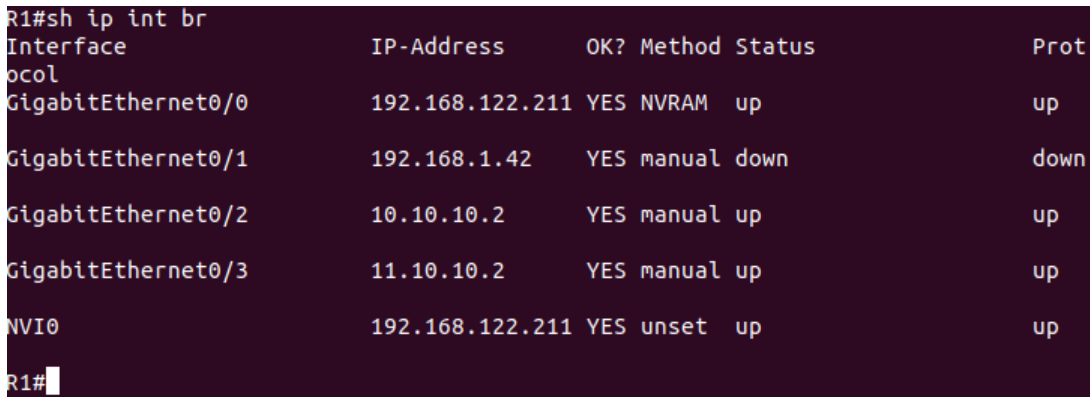
Figure III.35, depicts the output of *show ip interface brief* on the CLI of R1 as an example for verifying if the IP addresses were assigned successfully from the configuration stage.



```
R1
File Edit View Search Terminal Help
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

*Aug  6 17:21:45.559: %SYS-5-CONFIG_I: Configured from console by admin on vty1
(192.168.122.1)
R1#
```

**Figure III. 34:** Syslog message indicates the connection between R1 and the admin.



```
R1#sh ip int br
Interface                IP-Address      OK? Method Status  Prot
ocol
GigabitEthernet0/0      192.168.122.211 YES NVRAM  up      up
GigabitEthernet0/1      192.168.1.42   YES manual  down    down
GigabitEthernet0/2      10.10.10.2     YES manual  up      up
GigabitEthernet0/3      11.10.10.2     YES manual  up      up
NVI0                     192.168.122.211 YES unset   up      up
R1#
```

**Figure III. 35:** The output of *show ip interface brief* after executing the configuration stage.

### III.7 Discussion

Netmiko module provided a lot of prebuilt mechanisms and methods that interface with the CLIs programmatically. As the results indicate, it facilitated the SSH communication and sending instructions from the management platform program down to the Cisco network elements, with the ability of handling errors. The NMP are divided into several dependent functions to ensure reusability. The NMP execution time are vary based on the numbers of instructions and calculation on the functions, generally it is highly fast comparing with the manual procedure, especially when executing in multithreads fashion. TextFSM also helps in retrieving essential data easily back from the CLI output by providing template for structuring data of show commands. The stages order, (i) Configuration, (ii) Verification, (iii) Confirmation, (iiii) Checking, and reporting, is the same when we interact using the classical methods.



### III.7.1 Benefits, Drawbacks and limitations

The following benefits are noticed in this use case testing with the management platform:

- It covers most of the network administration tasks.
- Ease of using and selecting stages of from the UI menu.
- The platform reduces human errors that slow down network performance.
- Relatively Abstracted from the command line interface outputs.
- Centralized workflows and more network visibility.
- It is super-fast and can quickly done lot of needs.
- Decrease the effort and manual intervention.
- Reusable, extensible and open for integration and extending capabilities.
- Executing major stages and tasks separately.
- Solution for operational expenses saving.

Like any software, the NMP has also a set of drawbacks and limitation remarked from the execution process.

- The lack of repetitive commands in the same configuration file.
- The lack of the unstructured data back from the CLI (show commands that does not has TextFSM template yet).
- No *idempotency* and validating for the desired states yet.
- Does not respond to network events and notification yet.
- Some state of the CLI output can be changing which can break the code execution when get in data parsing.

### III.7.2 Future works

There are many possibilities to proposed as future works to enhance performance, since we produced this platform for extensibility and continuous development in mind, some of which are derived and relevant to the drawbacks and limitations that are referred to previously. Those proposed future works are as follow:

- Adding Jinja templating language for avoiding repetitive commands in the same config

files and to focusing only on changeable values.

- Building more TextFSM templates to avoid parsing raw text.
- Performing mechanisms to perform events driven.
- Adding more security enhancements; encrypting sensitive data.
- Adding support for REST and NETCONF device APIs by integrating *request* and *ncclient* modules.
- Adding support for other network devices vendors; juniper, Arista ...etc.
- Integrating web application frameworks like *Django* to provide a nice GUI Dashboard.
- Integration with database applications to store different information; credentials, IP addresses, backups ...etc.

### III.8 Conclusion

This chapter demonstrates the competency of network automation using the primary intended management interface (CLI) on network devices, and how it is useful when creating predefined code based on Python to eliminate the need for following long management and configuration steps.

The proposed Network Management Platform uses the Netmiko library that enables the SSH connection between the application Python file and the CLIs on network devices. The NMP consists of four main stages that match the classical network management tasks order, under each stage, we've got options that execute various functions which then send various instructions to Cisco IOS network devices in our example, relevant to the configuration process and other administrative tasks, as verification, backups aggregation, checking the network and reporting in less time and human effort. After we explained the workflow of each function and placed some code source of some of them, then passing through the test case of each task successfully on the GNS3 network topology, and based on the noticed outputs, we can conclude that the NMP has proven its value and the good performance in the network through a number of advantages. The NMP ensures efficiency, reusability, extensibility, scalability, and ease of using it over the network using just the CLI and SSH. The emerged drawbacks and limitations that can be improved by adding the suggested enhancements. The complete project and the Python code are available on my GitHub page ([https://github.com/AbdelhakBoumezrag97/NMP\\_Project](https://github.com/AbdelhakBoumezrag97/NMP_Project)).

# **GENERAL CONCLUSION**

### General Conclusion

This work proved the value of network automation and programmability over the network infrastructure, especially with the continuous increase in the number of devices and their associated configurations. Every mentioned automation approach, has a mechanism that aims to simplify the administration tasks difficulties involved in the manual network management, which can be implemented depending on what can the network devices support. The CLI-based types, must respond to the dynamic changing due to the emerging of other programmability procedures that try to eliminate definitely the vendor-dependency as SDN and the idea of commanding the network using a controller, and exercising programmatically the data path via APIs like OpenFlow, in order to have more visibility and abstracting the low-level interactions. Network automation is very possible with the CLIs to configure and operate the network.

The strength of using Python in the networking context is that it is not bound by someone else pre-built framework as Ansible, we can choose libraries according to needed desires and we can extend capabilities as much as possible. Netmiko facilitates the SSH protocol establishment between the Python program and network devices, gives also a set of methods that can send instructions down to the network devices CLIs easily. In addition, we can easily parse string output data and pulling useful information using TextFSM module and *ntc\_template*. The execution time are essential especially when the program going to scale, for this we introduce the idea of parallel execution as an added value using the threading module.

Our Network Management Platform (NMP) is a super simple user interface based network automation application, developed using Python, it demonstrated his efficiency in the previous test case on GNS3 over vendor-specific which is Cisco IOS switches and routers, it uses the Netmiko package as the primary module with other modules for enhancements and the terminal interface design as *simple\_term\_menu*. We have covered each aspect from the design to the development process ending by the examination in the experimental part. the NMP consist of four stages run many functions that match the classical management order. these stages are network configuration, network verification, network confirmation, as well as the checking and CSV reporting, it has many benefits and drawbacks that can give a regular evaluation, some of these drawbacks can be eliminated in the journey of integrations that we mentioned in the future work section. this work and this Platform fulfill the project objectives of avoiding the manual configuration interaction over the traditional network devices and its related problems.

# **REFERENCES**

## References

- [1] J. A. Alex, NETWORK AUTOMATION USING PYTHON 3, 2018.
- [2] Cisco Inc, "What Is Network Automation?," Cisco, [Online]. Available: <https://www.cisco.com/c/en/us/solutions/automation/network-automation.html>. [Accessed August 2020].
- [3] T. Ryan and G. Jason, Programming and Automating Cisco Networks, USA: Cisco Press, 2016.
- [4] A. W. Rheza and R. R. Nur, "PENGEMBANGAN APLIKASI OTOMATISASI ADMINISTRASI JARINGAN BERBASIS WEBSITE MENGGUNAKAN BAHASA PEMROGRAMAN PYTHON," *Journal of Mechanical Engineering, Electrical and Computer Science*, vol. 10, no. 2, pp. 741-752, 2019.
- [5] A. Ratan, E. Chou, P. Kathiravelu and D. M. O. F. Sarker, Python Network Programming, UK: Packt Publishing, 2019.
- [6] F. Xenofon, K. M. Mahesh and K. Kimon, "Software Defined Networking Concepts," in *Software Defined Mobile Networks (SDMN): Beyond LTE Network Architecture, First Edition.* , John Wiley & Sons, Ltd., 2015, pp. 21-44.
- [7] K. Hyojoon and F. Nick, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114-119, 2013.
- [8] M. Paul, B. Titus, C. Radu and S. Florin, "Network Automation and Abstraction using Python Programming Methods," *MACRo*, vol. 2, no. 1, pp. 95-103, 2017.
- [9] U. Brian and K. Gary, *Software Defined Networking For Dummies*, New Jersey: John Wiley & Sons, Inc., 2015.
- [10] O. WENDELL, CCNA 200-301 Official Cert Guide, Volume 2, San Jose, CA: Cisco Press, 2020.
- [11] F. Nick, R. Jennifer and Z. Ellen, "The Road to SDN: An Intellectual History of Programmable Networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87-98, 2014.
- [12] K. Diego, M. V. R. Fernando, V. Paulo, E. R. Christian, A. Siamak and U. Steve, "Software-Defined Networking: A comprehensive survey.," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14-76, 2014.
- [13] G. Dewang and P. Levi, "A Centralized Network Management Application for Academia and Small Business Networks," *Information Technology in Industry Journal*, vol. 6, no. 3, pp. 1-10, 2018.

## References

---

- [14] E. Salib and J. Lester, "Hands-on Labs and Tools for Teaching Software Defined Network (SDN) to Undergraduates.," in *American Society for Engineering Education*, Salt Lake city, 2018.
- [15] J. Manar, S. Taranpreet, S. Abdallah, RasoolAsal and L. Yiming, "Software-Defined Networking: State of the Art and Research Challenges.," *Computer Networks*, vol. 72, pp. 74-98, 2014.
- [16] Open Networking Foundation, "Software-Defined Networking: The New Norm for Networks," ONF ONF White Paper, CA, 2012.
- [17] A. Siamak, *Software Defined Networking with OpenFlow*, Packt Publishing, 2013.
- [18] M. Nick, A. Tom, B. Hari, P. Guru, P. Larry, R. Jennifer, S. Scott and T. Jonathan, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69-74, 2008.
- [19] W. Braun and M. Michael, "Software-defined networking using OpenFlow: Protocols, applications and architectural design choices," *Future Internet* , vol. 6, no. 2, pp. 302-336, 2014.
- [20] C. Ching-Hao and D. Y. D. Lin, "OpenFlow Version Roadmap," pp. 1-15, 2015.
- [21] A. Yashi and K. Uma, "Software Defined Networking: Basic Architecture & Its Uses In Enterprises," in *International Conference on "Computing: Communication, Network and Security"*, 2018.
- [22] E. Jason, "Network automation with Ansible," O'Reilly Media, Inc, USA, 2016.
- [23] G. Jason, H. Roddie and V. Srilatha, *Cisco Software-Defined Access*, Cisco Press, 2020.
- [24] Cisco Public, "Cisco DNA Center Solution Overview," 9 July 2020. [Online]. Available: <https://www.cisco.com/c/en/us/products/collateral/cloud-systems-management/dna-center/nb-06-dna-center-so-cte-en.html>. [Accessed August 2020].
- [25] P. Hank, Writer, *Configuration Management and the Network, A Network Programmability Basics Presentation*. [Performance]. Cisco DevNet, 2017.
- [26] E. Jason, S. L. Scott and O. Matt, *Network Programmability and Automation, Skills for the Next-Generation Network Engineer*, USA: O'Reilly Media, Inc, 2018.
- [27] O. Karim, *Network Automation Cookbook*, UK: Packt Publishing Ltd., 2020.
- [28] B. Kyle and D. Kevin, Writers, *Ansible Network Automation*. [Performance]. Red Hat, Inc, 2018.
- [29] Red Hat, Inc, "Ansible for Network Automation, Documentaion," Red Hat, Inc, 2019. [Online]. [Accessed August 2020].

## References

---

- [30] C. Sean and B. Andrius, Writers, *What's new with Ansible Network*. [Performance]. Red Hat, Inc ; Cisco DevNet, Inc.
- [31] D. Gerald, Writer, *Network Resource Modules*. [Performance]. Red Hat, Inc.
- [32] P. Hank, Writer, *APIs are Everywhere...but what are they?, A Network Programmability Basics Presentation*. [Performance]. Cisco Devnet, 2017.
- [33] "REST API Tutorial," [Online]. Available: <https://restfulapi.net/>. [Accessed August 2020].
- [34] E. Brad, G. R. Ramiro, H. David and G. Jason, CCNP and CCIE Enterprise Core ENCOR 350-401 Official Cert Guide, USA: Cisco Press, 2020.
- [35] E. Jason and T. Roman, Writers, *Model Driven Network Automation with IOS-XE*. [Performance]. Cisco Public, 2017.
- [36] P. Hank, Writer, *REST APIs Part 1: HTTP is for more than Web Browsing A Network Programmability Basics Presentation*. [Performance]. Cisco DevNet, 2017.
- [37] Y. James and A. A. Imad, "An Empirical Study of the NETCONF Protocol," *2010 Sixth International Conference on Networking and Services, IEEE*, pp. 253-258, 2010.
- [38] H. Brian, A. Watwe and S. Siddharth, "Protocol Efficiencies of NETCONF versus SNMP for Configuration Management Functions," *interdisciplinary Telecommunications at the University of Colorado, Boulder*, pp. 1-13, 2011.
- [39] R. Enns, M. Bjorklund, J. Schoenwaelder and A. Bierman, "Network Configuration Protocol (NETCONF), RFC 6241," June 2011. [Online]. Available: <https://tools.ietf.org/html/rfc6241>. [Accessed August 2020].
- [40] C. Moberg, Writer, *NETCONF by Example*. [Performance]. IETF, 2015.
- [41] B. Byrne, Writer, *Deep Dive Into Model Driven, Programmability with NETCONF and YANG*. [Performance]. Cisco DevNet, 2018.
- [42] Python Software Foundation, "ncclient 0.6.9," [Online]. Available: <https://pypi.org/project/ncclient/>. [Accessed August 2020].
- [43] Marius-Ioan, CANDREA-BOZGA and C. Petrică, "Integrated Management of Transport and Commutation Resources over the Network Layer," *Journal of Military Technology*, vol. 2, no. 1, pp. 1-4, 2019.
- [44] PathSolutions , "What is Network Troubleshooting?," PathSolutions Inc, February 2019. [Online]. Available: <https://www.pathsolutions.com/blog/what-is-network-troubleshooting>. [Accessed August 2020].



## References

---

- [45] Python Software Foundation, "What is Python? Executive Summary," [Online]. Available: <https://www.python.org/doc/essays/blurb/>. [Accessed August 2020].
- [46] B. Aly, Hands-On Enterprise Automation with Python, UK: Packt Publishing Ltd., 2018.
- [47] A. Ratan, E. Chou, P. Kathiravelu and D. M. O. F. Sarker, Python Network Programming, UK: Packt Publishing Ltd, 2019.
- [48] R. Abhishek, Practical Network Automation Second Edition, Uk: Packt Publishing Ltd., 2017.
- [49] K. Byers, "Netmiko Library," 02 01 2019. [Online]. Available: <https://pynet.twb-tech.com/blog/automation/netmiko.html>. [Accessed August 2020].
- [50] C. Eric, Mastering Python Networking, Third Edition, UK: Packt Publishing Ltd, 2020.
- [51] "Netmiko API Documentation," pdoc, [Online]. Available: [https://ktbyers.github.io/netmiko/docs/netmiko/base\\_connection.html](https://ktbyers.github.io/netmiko/docs/netmiko/base_connection.html). [Accessed August 2020].
- [52] Google, "README.md TextFSM," Google, [Online]. Available: <https://github.com/google/textfsm>. [Accessed August 2020].
- [53] K. Byers, "Netmiko and TextFSM," 06 2018. [Online]. Available: <https://pynet.twb-tech.com/blog/automation/netmiko-textfsm.html>. [Accessed August 2020].
- [54] R. Abhishek, Practical Network Automation Leverage the power of Python and Ansible to optimize your network, UK: Packt Publishing Ltd., 2017.
- [55] IngoMeyer, "simple-term-menu 0.6.7," Python Foundation software, [Online]. Available: <https://pypi.org/project/simple-term-menu/>. [Accessed August 2020].
- [56] Sublime HQ Pty Ltd, [Online]. Available: <https://www.sublimetext.com/>. [Accessed August 2020].
- [57] Cisco Inc, "Cisco Virtual Internet Routing Lab," [Online]. Available: <http://virl.cisco.com/>. [Accessed August 2020].