On Detailed Network Systems Configuration Management Automation using Python

[1]ARMANDO ELEZI, [1,2]DIMITRIOS A. KARRAS
[1]Canadian Institute of Technology, Tirana, ALBANIA
[2]National and Kapodistrian University of Athens (NKUA), GREECE

Abstract: - In nowadays, communications are expanding in very high rates. New technologies are being born and some of them are taking so much importance in people's lives. In a situation where people's needs are getting more complicated and everybody's lifestyle is advancing to another level, bigger and better infrastructure is needed. Managing all this process can't be anymore a step by step process. It is strictly needed to evolve in automated process. Network configuration and reconfiguration may be a repetitive process, time consuming, and error prone process. To address this problem this paper is going to shed light on the benefits of an automated configuration and topology verification process. To this end, a proof of concept system, Netmiko, has been used in a case study. Netmiko scripts are able to read the current network state, can apply predefined configurations loaded from text-files or csv files, and automatically verify the network state. The goals of this paper are to demonstrate the development of Netmiko scripts, to illustrate the simplicity in the implementation, to compare the automated network system reconfiguration to a fully manual one, and, finally, to discuss potential pros and cons in switching to an automated network configuration process in everyday practice. A simple Packet Tracer simulation in joint with a GNS3 simulation are involved to evaluate this proof of concept. In Packet tracer a manual network configuration is performed while in GNS3 an automated network configuration is developed.

Key-Words: - network configuration management, automated configuration management, computer networks, packet tracer, GNS3, Cisco IOS XR, Python, Netmiko Library.

Received: May 15, 2022. Revised: January 6, 2023. Accepted: February 14, 2023. Published: March 17, 2023.

1 Introduction

In the area of network communications, automation is related to the configuration of network devices and components that realize the communication between different users and services (Red Hat and Sisay Tadesse, Claire Naiga Serugunda Fabrizio Granelli et. al. (2021)). This is challenging, difficult and requires hard work, because not only one needs to verify the configuration of a device, but also the entire network component configuration that results the connection of all the devices together. When working with automated network configuration tools or scripts it is therefore, important to detect bugs in the software, that may lead to delays in committed configurations.

Many network automation engineers have stated that today's systems must be able to detect errors and must be able to auto-correct themselves in a way that they restore the system's stability. To detect such changes or deviations, automation capabilities include the concept of knowing and understanding of a "steady state" where the system fulfills the requirements of a system in health.

A steady state requires that firstly the interfaces and ports must be properly configured, desired protocols are enabled and properly configured on devices and the desired function in a correct way.

This could be done in many ways, but the main most used ones are: (i) return the device configuration to previous configuration, (ii) correct, or smartly correct the actual configuration and restore the "steady state" of the device so that it may connect successfully to the rest of the network and complete correctly all of its assigned tasks.

The goal of this paper is to: use and evaluate a tool that automates network configuration, compares the automated process to a completely manual network configuration with regards to factors, like time and costs, and discuss potential benefits or problems in a manual network configuration transition to an automatic network configuration.

.

The proposed automated network configuration scripts should be easy to implement, should be able to generate accurate network desired configurations, and should be easily understandable. The aim of this paper is to obviously state the potential pros and cons in automating network configurations processes.

1.1 Challenges

The number of possible states that appear in real networks is huge, and it is hard to predict and handle all these states in advance and mitigate actions for the future (Sisay Tadesse, Claire Naiga Serugunda Fabrizio Granelli et. al. (2021)). Verification of the correctness of network automation processes is required, hence their systematic execution appears to be a big challenge. First, the large number of devices in a real environment that need to be configured might affect the time required to configure the entire network. Second, the type of each device for example: CISCO, Juniper, Fortinet, Checkpoint, affects the design and implementation of the automation process. For example, if the automated tool has to support a wide range of devices such as Cisco, Juniper, HP, Alcatel- Lucent, and Aruba, it affects the implementation effort, time and labor needed to be spent to make the whole schema workable. Third, the scripts may be prone to bugs which need to be fixed, a time-consuming task, almost always need efficiency improvements, and there is always the need to make sure that they can be adapted for many other network configuration cases. Fourth, the type of configuration that needs to be managed on the devices also affects the completion time of the configuration process, because not all protocols have the same converging time.

2 Technical Background and Tools Hierarchy needed for Network Configuration Management

Remote administration is beneficial while seeking to facilitate communications to devices and gadgets which can be geographically distant, especially when having to acquire access to many devices without delay, not having to attach via cable every tool and device (Jason Edelman, Scott S. Lowe, Matt Oswalt (2018)). A CLI interface (Command Line Interface) might be a way of interacting with PC systems, like Operating Systems (OS) and networks. Common protocols used for CLI based total network management are for example Telnet and SSH which both allow far from access to community devices. Telnet runs on pinnacle of the connection orientated Transmission Control Protocol (TCP) while speaking with far flung devices and presents more reliable communication than a connectionless protocol, like User Datagram Protocol. To manage Telnet limitations, the SSH protocol is one of the various foremost recognized protocols for stable remote community services over an insecure community, supplying encryption, cryptographic host authentication, and integrity protection. A device is accessed through a remotely located procedure via a stable channel provided by using the SSH protocol. Like Telnet, SSH runs on top of TCP, however with security measurements providing a secure connection for tool management over an insecure community, like the internet. Basic issues involved in the herein planned remote control for network configuration management are

- Accessing SSH terminals with paramiko
- Transferring files thru SFTP
- Transferring documents with the help of FTP

In the proposed herein project for automated network configuration management several third party packages, like paramiko, pysnmp, and so on are involved, to facilitate developments using as a platform secure shell python. (Eric Chou, Abhishek Ratan, Pradeeban Kathiravelu (2019), M. O. FaruqueSarker, Sam Washington (2015) , José Manuel Ortega (2018) , Kirk Byers (2016))

SSH presents an exquisite encrypted communications among sender and receiver , so unrelated third-events can't see the content material of the info in the course of the transmission medium. Details of the SSH protocol are regularly discovered in these RFC documents: RFC4251-RFC4254, obtainable at http://www.Rfc-editor.Org/rfc/ rfc4251.Txt.

Python's paramiko library affords a clever support for the SSH-primarily based network communication. Python scripts could be used to investigate the benefits of SSH-based remote administration, just like the remote command-line login, command execution, and consequently the extraordinary steady community offerings among networked computers. (https://pypi.Python.Org pypi/pysftp/)

The SSH may be used as a client/server protocol where each of the parties use the SSH key pairs to setup the communication link. Each key pair has one private and one public key, as known in PKI infrastructures. The SSH public and private keys are regularly generated and digitally signed via an outside or an inside certificates authority, but this brings big overheads to a little enterprise. So, instead, the keys are frequently generated haphazardly via software tools, like ssh-keygen. The general public key should be available to all or any participating parties. As soon as the SSH patron connects to the server for the first time, it registers the general public key of the server on a special file acknowledged as ~/.Ssh/known_hosts. Of course, if you re-build the machines, just like the server device, then the old public key of the server might not in shape thereupon of the one stored within the ~/.Ssh/known_hosts report. So, the SSH client will issue an exception and stop from connection.

We can use the paramiko module to make an SSH patron and connect it to the SSH server. This module will supply the SSHClient() elegance. The instance of this client will robotically reject the unknown host keys. So, the user will be capable of initiating a coverage for accepting the unknown host keys. The built-in AutoAddPolicy() method will add the host keys as soon as they are discovered. In the sequel the user could employ the set_missing_host_key_policy() technique collectively with the subsequent argument at the ssh_client object as follows, Ssh_client.Set_missing_host_key_policy(paramiko. AutoAddPolicy()). It is additionally possible to add the device host keys by means of the use of the load_system_host_keys() approach.

It might be very interesting to intercept the network packets change among the client and the server. To this end we could use many commands but it's easy to use both the local tcpdump command and the Wireshark tool to capture network packets. With tcpdump, we'll be capable of specifying the target network interface (-i lo) and also the port (port 22) options. within the following packet exchange session, where 5 packet exchanges are shown during an SSH client/server conversation, as captured, also, through Wireshark in the following screenshot:

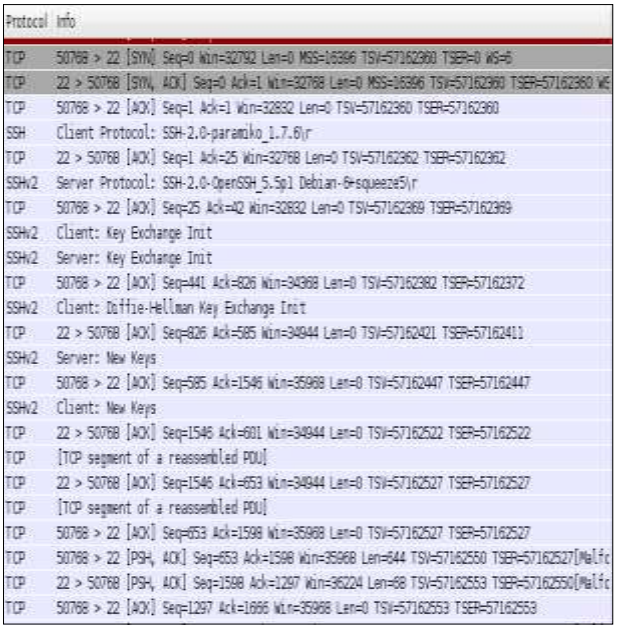


Fig. 1 Inspecting Packets

After completion of the TCP handshake session, the SSH packets that follow negotiate the relationship between the client and the server and determine how the client and the server negotiate the encryption protocols. During this example, the client port is #50768 and the server port is #22.

Apart for SSH protocol, the tools needed to manage this development project include, additionally, involvement of the Cisco IOS® software. Cisco IOS is a multitasking software bundle for Cisco-primarily based community elements that provides services, like routing, switching, internetworking, and specific telecommunications features. consistent with Cisco. IOS is presently working on numerous active Cisco devices, making it the maximum universally leveraged community infrastructure software application, offering a set of commands to configure Cisco devices. Cisco IOS presents excellent configuration modes for various user privileges. For international configuration mode allows input of instructions with the flexibleness to alternate the device configurations, while the setup mode permits the configuration of greater unique features, like interfaces and protocols, in an interactive way (request-reply). Cisco network elements may additionally even be configured via loading a data document with configurations immediately into the tool, instead of having to input the complete configuration in the kind of instructions. Based on Cisco IOS the OSFP routing

protocol in networking topologies between Cisco devices can be easily configured.

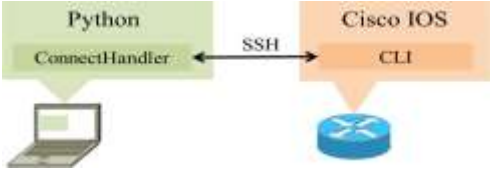
Cisco IOS XR is a detail of the Cisco IOS family that is made for building modular, and allotted middle routers. Such routers are typically positioned within the center or fringe of a backbone service company network wherein more robust solutions of device mirroring are required. The modularity is shown into the memory control wherein IOS XR has protected memory safety between processes, while routing policies run in separate memory space. A crashed BGP method will therefore no longer affect a parallel OSPF way, consequently developing a facilitating context for the multiple devices manipulation that the central network of an ISP requires. Additional functions supported thru IOS XR are hardware redundancy support, packet based software program distribution version, and optional functions, permitting multicast routing and Multiprotocol Label Switching (MPLS) to be configured at the same time the router is even in commission, without interrupting any functionality.

In IOS, the modifications you create to the configuration of the networking devices are carried out at once. IOS XR based devices have to be committed (the use of the dedicate command) in advance before being implemented. Syntax could include commands for defining characteristic abilities, as can be visible in Table 1 below, in which common capabilities are indexed and defined. The commits are saved regionally within the tool, together with an autogenerated devote ID, and can be displayed whilst strolling the configuration commands listing. The commits in the database involve rollback points, permitting preceding configurations to be activated all over again with the rollback command.¨

Table 1. Basic BGP configuration for Cisco IOS routers and Cisco IOS XR.

Cisco IOS (Router ID: 192.168.1.5)	**Cisco IOS XR (Router ID: 192.168.1.8)**
router bgp 3402 no synchronization bgp log-neighbor-changes neighbor 192.168.1.8 remote-as 3402 neighbor 192.168.1.8 update-source Loopback0 no auto-summary end	!! IOS XR Configuration 5.3.3 router bgp 3402 neighbor 192.168.1.5 remote-as 3402 update-source Loopback0 ! ! End

The architecture and hierarch of tools needed to implement the aforementioned problem of automated network configuration management includes, moreover Python scripting. Python is a multi-functional programming language. Python gives high-degree syntax that permits programmers to precise standards in fewer traces of code than is viable with other programming languages like Java. The power of Python relies on the efficient use of its specialized libraries with minimal effort using very short scripting commands. The library used herein is Netmiko. Python provides the Netmiko library, which simplifies SSH manipulation of network devices (Kirk Byers (2016) Linux Journal, Packet Hub, Learning Oreilly), returns a rich text output to the developer, permitting him to pay attention on the configuration of the device in place in terms of low-degree SSH details. Netmiko supports more than one framework including a large set of Cisco frameworks, HP and Juniper ones. From the Netmiko library, a set of factory capabilities are regularly imported, one being the ConnectHandler class selecting the right Netmiko class based upon the tool specified and being a library typical to devices from multiple companies. The following figure shows an example on a way to use the Netmiko library to open an SSH connection at the Cisco IOS tool and verify the connectivity thru sending a command to spark off the variety of active interfaces at the device, and for that reason verifying connectivity with back output. As depicted in the following Listing of fig. 2, the ConnectHandler needs the tool type as an input, which in its familiar way, is regularly set to an large variety of devices from vendors, like Cisco IOS, Cisco IOS XR, Juniper, HP and Huawei.




```
[1]>>>net_connect=ConnectHandler(device
   _type=' cisco_ios',ip='10.10.10.227'
      ,
         username='pyclass',password='p
      sw')
[2]>>>net_connect.send_command("show
   ip int brief")
```

Fig. 2: SSH using Netmiko connection to a device, via Python interpreter .

The ConnectHandler opens and keeps the SSH session with the CLI tool. Moreover, the pyIOSXR library is specialized to facilitate the communication with Cisco IOS XR devices, thru using an Extensible Markup Language (XML) agent. XML, being a framework and community meta language, facilitates the sharing of information among the Cisco IOS XR factors and the Python language.

```
device = IOSXR(hostname='10.10.10.227',username='***',password='***',
    port='22',timeout=120)
device.open()
device.load_candidate_config(filename='
/path/..')
device.compare_replace_config()
```

Fig. 3: Connecting to Cisco IOS XR, via Python interpreter

The XML agent permits for exclusive processes of sending instructions to devices, enabling commands, as proven in Listing fig. 3, wherein the script first opens an SSH connection to device, sets up the relationship within the device variables (line one), after which enters XML mode for the execution of the additional commands. The commands in Listing fig. 3 first ensure that the communication channel is open (line two), that a target configuration from a specified route is setup (line 3), comparing this to other configurations of the device (line 4), and finally committing the new configuration with the device (line 5), keeping the specific parts and replacing only the configurations that are not possible.

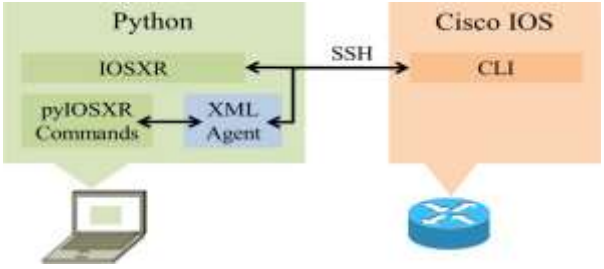


Fig. 4: SSH connection to a Cisco IOS XR

As depicted in the Figure 4, the messages exchange session remains channeled via the lively SSH connection,
to the device CLI, but the XML agent enables the process development. Note that the CLI manages the exchange session with the actual tool, as defined above. For instance, the device.Commit_replace_config(label='labelname') method corresponds to the fusion of the dedicate replace and devote label labelname. Cisco command noted in the IOSXR framework corresponds to the ConnectHandler feature used inside the Netmiko library. The Netmiko and pyIOSXR connection Handlers both lock the device under the entire connection, assuring that no configuration adjustments, by means of every other person, is merged into the devote class.

Within this object oriented framework, involved in defining the hierarchy and architecture of the system of tools for automated network configuration management.

JavaScript Object notation (JSON) plays a significant role too. JSON is a compact, textual content, language independent, layout used for statistical analysis of transactions, which can be written manually or as an instance generated from a Java object. JSON permits for dependent facts sets management, just like XML, and is considered to be simpler to apply than its noted alternatives. This concept is primarily based on that in JSON shorter syntax is used with a simplified utilization of tags, which makes it faster to realize and develop. With regards to Python, the JSON library may be imported, permitting parsing between documents in JSON layout together with a Python dictionary or listing.

3 Related Work and Technologies in Automating Network Configuration Management

Netmiko is used to establish SSH connections to the devices and confirm that the connections have been setup properly. The devices have been stored in Python dictionaries corresponding to a community tool. The script that applied all above instructions, for a simple network topology, consisted of 93 lines of code and required approximately 46 seconds to execute Cisco IOS XR devices configuration as it will be demonstrated in the next sections. Through this computerized manner, decreasing tedious work in repetitive protocol configurations is made possible. If it is evolved and made everyday practice, for general and non vendor specific networking solutions, such scripting could serve as a completely automatic BGP configuration machine. Several technologies and protocols are herein evolved and are presented in the sequel.

3.1 Dynamic Host Configuration Protocol

Dynamic Host Configuration Protocol (DHCP) is a community (network) protocol used to offer network configurations to community elements with the aid of allocating IP addresses in an automatic fashion. Consequently, IP addresses will not be manually configured on community devices inclusive of workstations, printers, game consoles, and private computers. The automatically dispensed IP addresses are selected from a pool of addresses, set with the aid of the server administrator and assigned to the clients thru the DHCP scheme. The client declares its presence inside the community through broadcasting a Discovery message. The server sends returned an Offer of an IP address and the client responds by way of sending a Request to hold the address. The address is taken in use by the client first when receiving the Acknowledge message dispatched via the server, ensuring that no other device inside the network has claimed the deal with. The purpose of the protocol is to deliver all configuration statistics needed, for a computer or network device, to access the community. This computerized host configuration provides dependable IP addresses coping with configuration, hence minimizing the occurrence of configuration mistakes inclusive of typographical errors.

The data supplied by DHCP is IP addresses, subnet mask, default gateway etc. The most effective information to be given dynamically is the IP address deal with the remaining facts nonetheless needed to be manually configured on the DHCP servers. If the guide configuration contains faulty records, allotted to all of the community elements, the devices will no longer be able to communicate. For instance, if incorrect IP address is given to a host by means of the DHCP server, the host will be no longer capable of resolving DNS names due to the use of wrong DNS servers, hence the host will not be able to reach the ideal IP addresses.

Due to a centralized and automated TCP and IP configuration, and the handling of modifications for mobile devices, such as portable non-public computers, DHCP reduces community administration. This dynamic protocol is yet another instance of automated community configuration that reduces manual management and errors through the tedious but statically assigning IP addresses process.

3.2 Network Configuration Protocol

In the early 21st century, it became clear that the Simple Network Management Protocol (SNMP), was used for monitoring networks mainly than for managing them The Internet Architecture Board and Members of the Internet Engineering Task Force (IETF) found that the network control changed through CLI scripting, which became, as noted, restricted in dependent error management, transaction management and susceptible to changeable syntax of instructions. The Network Configuration Protocol (NETCONF), was developed to address the shortcomings of existing approaches, imparting exclusive mechanisms to simplify the installation, manipulation and deletion of configuration on community devices, as compared to SNMP.

NETCONF uses Remote Procedure Calls (RPC) to permit a network administrator to change network configuration facts with the managed devices. The RPC based totally on messages exchange model works in a request-reply way using XML, supplying the NETCONF protocol with transport protocol unbiased framing, and the usage of elements for the communication between the client and the server.

The NETCONF protocol transactions allow error handling of incomplete configurations that the Cisco specific, non transactional approach does no longer offer. In a Cisco-unique method, inclusive of the OSPF configuration, an incomplete configuration will not be defined as a mistake, but as an alternative and the device will wait for extra instructions to finish the configuration. In a transaction primarily based communication, the inadequate configuration may be described as an error and comments will be provided to the administrator that the protocol isn't properly configured. Thus, a transaction primarily based framework will enable the network manager to recognize the management of the services within the network, as opposed to the management of devices configurations. Provided that a Cisco device involves the NETCONF protocol, a Cisco-unique technique and NETCONF approach can be used collectively, so long as the configurations aren't contradictory.

3.3 Software-Defined Networking

Software-Defined Networking (SDN) is a modern approach to manipulate and control networks, using suitable software systems by abstracting network

and networking devices performance statistics and managing all network factors. In SDN, the manipulating planes of all community factors are managed through a logically centralized SDN controller, permitting configurations of big area networks to be driven by managing network factors in order to easily integrate, as an example, new network services. The SDN controller communicates as a network backbone with all the devices, accordingly creating a control plane in the network wherein all the devices may be managed through it.

There are specific software system standards adapted for the management of the abstracted planes of an SDN, as an example including NETCONF and OpenFlow, which provides the capability for unique OpenFlow enabled Switches to be controlled or updated centrally and for that reason shaping an integral SDN. McKeown et al. (McKeown et al.2008) defines an OpenFlow Switch as consisting of at least three components: "(1) A Flow Table, with a specific action related to each entry, to inform the switch the way to process the system component, (2) A Secure Channel that connects the transfer of control commands to an overseas control technique (known as the controller), allowing instructions and packets to be dispatched among a controller and the relevant switch. (3) The OpenFlow Protocol that offers an open and preferred way for a controller to talk with a transfer." McKeown emphasizes that through specifying the protocol in which entries within the Flow Table are manipulated and defined exactly, then, the switch itself does not have to be programmed and configured and this can be easily managed via the protocol.

Further, McKeown draws the belief that: "OpenFlow may be a realistic compromise that permits researchers to run experiments on heterogeneous switches and routers in the course of a uniform way, without setting the requirement for providers to expose the internal workings of their merchandise, or for researchers to code and install vendor-unique manipulation software". This statement regarding OpenFlow type protocols implies clearly the facilitation of network configurations on the basis of a uniformly defined network, but, additionally, implies the significance of the investigation for uniform network configurations despite integration of vendor-specific networking factors.

4 A Proposed Methodology for Automated Network Configuration Management through Netmiko Python Library

The proposed herein methodology is based on the capabilities offered by the Netmiko python library. Netmiko is a multi-dealing SSH Python library that produces connectivity to community devices thru SSH protocol. This library adds certain important functionalities to the paramiko library, which is the de-facto SSH library in Python. Netmiko simplifies the connectivity to a networking community device via SSH permitting the use of a smooth method for issuing remote calls like the “send_command”, in order to be able to execute sets of commands on a device as well as to properly analyze the consistency of its response with the tool being linked to python. Netmiko is characterized as an open software supply with all code publicly available on GitHub and it's absolutely easy to start using it.

Netmiko facilitates a growing list of networking products integration and configuration into complex network topologies. Such lists could be found within Netmiko documentation. Some providers offer device control with a variety of different PC application commands. As an instance, Holler provides two kinds of such commands for the same device: dell_force10 and dell_powerconnect; and Cisco offers many software systems variations at several product strains, like sidecisco_ios, cisco_nxos and cisco_asa. The authentic Netmiko code and documentation is at https://github.Com/ktbyers/netmiko. Moreover, this paper is extending the results of [16], presenting a step by step methodology for automating network configuration management using solely Netmiko , illustrating the corresponding framework in implementation detail.

4.1 Login to the Router

Here’s a simple script to log in to the router (at IP 192.168.255.249 with a username and countersign of cisco) and show the version:

```
from netmiko import ConnectHandler
device = ConnectHandler(device_type=’cisco_ios’, ip=’192.168.255.249', username=’cisco’, password=’cisco’)
output = tool.Send_command(“show version”)
print (output)
tool.Disconnect()
```

The output of the execution of code with regards to a router is as follows. As we're going to see within the pattern code, we name the ConnectHandler characteristic based on the Netmiko library, which takes four inputs (platform kind, IP address of tool etc.)

Depending upon the selection of the platform kind, Netmiko can understand the back spark off and hence to issue the proper command through SSH into the particular component. Once the relationship is formed, we're going to send commands to the component through the use of the send_command approach.

Fig. 5: Retrieving records

4.2. Send Command

Once we get the return value, the value is saved in the output variable as displayed below, thus, is saved in the string output of the command that we sent to the device, as the device response. The final line, which makes use of the disconnect feature, ensures that the connection is terminated clearly as soon as the task ends. For configuration (for example, it is desired to offer connection to the FastEthernet 0/0 router interface), we use Netmiko, as shown in the subsequent example:

```
from netmiko import ConnectHandler
print (“Before config push”)
tool = ConnectHandler(device_type=’cisco_ios’, ip=’192.168.255.249', username=’cisco’, password=’cisco’)
output = device.Send_command(“display walking-config interface fastEthernet zero/zero”)
print (output)

configcmds=[“interface fastEthernet 0/0”, “description my test”]
device.Send_config_set(configcmds)
print (“After config push”)
output = device.Send_command(“show walking-config interface fastEthernet 0/zero”)
print (output)
tool.Disconnect()
```

As we are capable to see, for config push, we don't need to perform any extra configurations but simply specify the instructions in the same order as we send them manually to the router through a listing, and pass that list as argument to the send_config_set feature. The output at Before config push may be a direct output of the FastEthernet0/zero interface, however, the output under After config push (in the figure 6 below), defines configurations managed using the input listing of instructions. In an exceedingly similar way, we are able to skip multiple instructions to the router, and Netmiko will get into configuration mode, write those instructions to the router, and exit config mode.

If we would like to ensure that the router writes the newly driven configuration to memory configuration, we use the following command after the send_config_set command:

device.Send_command("write memory")

.

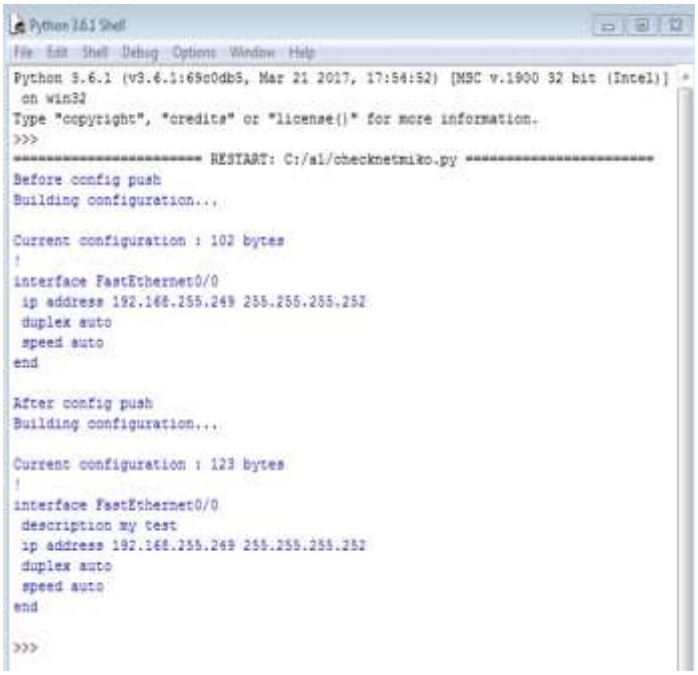


Fig. 6: Send Command Output

4.3 Network Configuration Management through Templates Usage

With all the routers accessible and at hand via SSH, it is permitted to configure a base template that sends the Syslog to a Syslog server and additionally ensures that simplest statistics logs are dispatched to the Syslog server. Also, after configuration management, a validation could be obtained to make sure that logs are being sent to the Syslog server.

The logging server information is as follows:

- Logging server IP: 192.168.20.5
- Logging port: 514
- Logging protocol: TCP

Additionally, a loopback interface (loopback 30) is desired to be configured with the loopback interface description.

The code traces for such a template are as follows:

logging host 192.168.20.five delivery tcp port 514
logging lure 6
interface loopback 30
description " loopback interface"

To validate that the Syslog server is accessible for the logs sent, we use the display logging command. within the event with the output of the command containing the text:

```
from netmiko import ConnectHandler
template="""logging host 192.168.20.5 transport tcp port 514
logging entice 6
interface loopback 30
description " loopback interface"""
username = 'test'
password="test"
#step 1
#fetch the hostname of the router for the template
for n in range(1, five):
ip="192.168.20.".format(n)
device = ConnectHandler(device_type='cisco_ios', ip=ip, username='test', password='test')
output = device.Send_command("show in hostname")
output=output.Split(" ")
hostname=output[1]
generatedconfig=template.update("",hostname)
#step 2
#push the generated config on router
#create a listing for generateconfig
generatedconfig=generatedconfig.Split("n")
tool.Send_config_set(generatedconfig)
#step 3:
#carry out validations
print ("*********")
print ("Performing validation for :",hostname+"n")
output=device.Send_command("display logging")
if ("encryption disabled, link up"):
print ("Syslog is configured and reachable")
else:
print ("Syslog is not configured and NOT accessible")
if ("Trap logging: level informational" in output):
print ("Logging set for informational logs")
else:
print ("Logging not set for informational logs")
print ("nLoopback interface status:")
output=tool.Send_command("show in loopback interface")
print (output)
print ("*************n")
```

```
RESTART: C:/gdrive/book2/github/edition2/python_automation/config_syslog.py
********
Performing validation for : rtr1

Syslog is configured and reachable
Logging set for informational logs

Loopback interface status:
Lo30                   up            up      "rtr1 loopback interface"
************

********
Performing validation for : rtr2

Syslog is configured and reachable
Logging set for informational logs

Loopback interface status:
Lo30                   up            up      "rtr2 loopback interface"
************

********
Performing validation for : rtr3

Syslog is configured and reachable
Logging set for informational logs

Loopback interface status:
Lo30                   up            up      "rtr3 loopback interface"
************

********
Performing validation for : rtr4

Syslog is configured and reachable
Logging set for informational logs

Loopback interface status:
Lo30                   up            up      "rtr4 loopback interface"
************
```

Fig. 7: Configure Loopback interface

5 Implementation Highlights of Automated Network Configuration Management Generic Cases

In this section we outline the implementation of certain generic configuration instances

5.1 Single Device Configuration

In the figure below a simple script for configuring a single network device is outlined.

```
netmiko_acl.py
from netmiko import ConnectHandler

cisco_device = {
    'device_type': 'cisco_ios',
    'host': '10.1.1.10',
    'username': 'u1',
    'password': 'cisco',
    'port': 22,
    'secret': 'cisco',
    'verbose': True
}

connection = ConnectHandler(**cisco_device)

print('Entering the enable mode ...')
connection.enable()


commands = ['access-list 101 permit tcp any any eq 80',
            'access-list 101 permit tcp any any eq 443',
            'access-list 101 deny ip any any']

print('Sending commands to the device ...')
connection.send_config_set(commands)


print(f'Disconnecting from {cisco_device["host"]}')
connection.disconnect()
```

Fig. 8: Configuring a single device

The script on using an object called cisco_device, where all the information necessary to ssh management is saved to the device. The main information consists of the IP-address of the device, the username and password to login and the port needed for the connection.

The port in most cases used by the ssh protocol to connect a device is frequently port 22

5.2 Multiple Device Configuration

On the figure below a multiple-device configuration is shown, where device information and configuration commands are hard-coded. Each

device has its own configuration table that may or may not differ from the other devices. A for loop is created to iterate through the device list and makes the configurations, one by one.

Fig. 9: Configuring multiple devices

5.3 Configuring Multi-Vendor Devices

```
with open('devices.txt') as f:
    file_content = f.read().splitlines()


devices = list()
for item in file_content:
    tmp = item.split(':')  #tmp is a list
    devices.append(tmp)


for device in devices:
    net_device = {
        'device_type': device[0],
        'ip': device[1],
        'username': device[2],
        'password': device[3],
        'port': 22,
        'secret': device[3],  # this is the enable password
        'verbose': True
    }

    connection = ConnectHandler(** net_device)

    if not connection.check_enable_mode():
        connection.enable()


    #connection.config_mode()
    config_file = input('Enter configuration file for device type ' + device[0] + ' with ip ' + device[1])
    print('Sending commnands to device ...')
    with open(config_file) as config:
        commands = config.read().splitlines()
    # print(commands)

    output = connection.send_config_set(commands)
```

Figure 10. Configuring multi-vendor devices

In the above a script is shown where multi-vendor devices can be configured at the same script, illustrating that we are not obligated to use different automation scripts for different device types.

5.4 Configuring Multiple Devices without Multi-Threading

```
def connect_and_run(device, cmd='show run'):
    print('Connecting to device: ', device['ip'])
    connection = ConnectHandler(**device)

    print('Entering enable mode ...')
    connection.enable()

    print('Executing command: ', cmd)
    output = connection.send_command(cmd)
    print(output)
    print('#' * 40)


if __name__ == '__main__':
    with open('devices.txt') as f:
        devices = f.read().splitlines()

    device_list = list()

    for ip in devices:
        cisco_device = {
            'device_type': 'cisco_ios',
            'ip': ip,
```

Fig. 11: Configuring multiple devices without multithreading

In the figure shown above, the script reads the IP address from a file called devices and configures devices one after the other. It finishes configuring one device and then, starts configuring another from the list of devices.

5.5 Configuring Multiple Devices using Multi-Threading

In the figure below a script is shown to configure multiple devices. The script reads the IP from a device file and proceeds to configuring the devices at the same time. It executes the first step of configuration at one device then, continues to the other device to execute the same step. After finishing the first steps for all the devices, the script passes to the second step of automated configuration management and so on.

```
        'password': 'cisco',
        'port': 22,
        'secret': 'cisco', #this is the enable password
        'verbose': True
    }
    device_list.append(cisco_device)

#print(device_list)

#script starting time
start = time.time()


output_q = Queue()

for device in device_list:
    my_thread = threading.Thread(target=connect_and_run, args=(device, output_q, 'sh run'))
    my_thread.start()


main_thread = threading.current_thread()
for my_thread in threading.enumerate():
    if my_thread != main_thread:
        my_thread.join()

while not output_q.empty():
    output = output_q.get()
    print(output)
    print('#' * 40)
```

Fig. 12: Configuring multiple devices using multithreading

6 Conclusions and Future Work

One of the areas where most organizations have difficulties when it comes the moment to deal with network automation is where to start such process. It is herein strongly recommended that deployment teams should consider starting on small, and most common problems and issues. Using this strategy helps to build the bases and clarify the ideas for a network automation-strategy.

Beyond the start phase, many companies try to push harder to the topic, by following big and risky steps in the automation process. The so far results from relevant reports indicated that these decisions will make automation attempts far more difficult than it is supposed to be, by wasting precious time and money, and this may set back progress for years.

Starting networking automation with the proposed methodology based on Netmiko library scripting can help groups manipulate their network infrastructure throughout the complete production lifecycle -- from building an initial infrastructure and evolving it throughout integrating multi-dealer products based new topologies and configurations, to managing everyday network automation, development and functionality services and tasks**.**

A few simple points to keep in mind when beginning your network automation journey include:

- Pick the right tool
- Make small steps but meaningful ones
- Fight the ideas to fall back to manual processes.

- Define metrics to track success.

The results, discussions and conclusions drawn from this research report shed light on the advantages of an automated configuration management and topology verification method.

The proposed methodology based on Netmiko library, fulfills the paper's goal of developing and evaluating a method that automates network configuration management.

Netmiko based methodology's runtime performance is in favor of any comparison with the manual techniques discussed and its impact has been herein analyzed. The proposed methodology based on Netmiko library contributes to lowering the manual hard work required by employers to carry out the infrastructure reconfiguration. using automated scripts like the above discussed.

An expensive and time consuming manual configuration management system will eventually be replaced by means of a procedure that causes fewer mistakes, offering to the user the potential to correct easily any mistake monitoring system perfromance.

Network automation may be a solution for this problem, it saves time, labor and costs. Network automation combined with testing and verifications can help and advance all the process of automation, upgrading it to another level.

While automation is combined with testing and verification, the wide variety of errors may be notably reduced.

Several enhancements in this research effort could be discussed and associated with two main areas.

First, the scripts could be upgraded to do more processes, to cover more aspects of the network automation, like creating scripts that make automatic check for the device software or firmware, that find the best adaptive time to install the updates, time that doesn't affect the network performance and ensures the network sustainability.

Second the scripts could be rewritten or enhanced partially to improve the efficiency of execution. Also adding newer and better adapted libraries to the scripts, could make the automation process evolve even further in its coverage. Moreover, towards these goals, the integration of AI (Artificial Intelligence) in the automated network management and its configuration management relevant scripts could be the future of network automation field.

References:

[1] Jason Edelman, Scott S. Lowe, Matt Oswalt (2018) Network Programmability and Automation: Skills for the Next-Generation Network Engineer, O'Reilly Media, 2018

[2] Eric Chou, Abhishek Ratan, Pradeeban Kathiravelu (2019) Python Network Programming: Conquer All Your Networking Challenges with the Powerful Python Language, Packt Publishing Ltd, 2019

[3] M. O. FaruqueSarker, Sam Washington (2015) Python Network Programming: Learning Python Network Programming, Packt Publishing Ltd, 2015

[4] José Manuel Ortega (2018) Mastering Python for Networking and Security: Leverage Python scripts and libraries to overcome networking and security issues, Packt Publishing (Sep 28, 2018)

[5] Kirk Byers (2016), Git HubNetmiko scripting website:https://github.com/ktbyers/netmiko

[6] Linux Journal Netmiko connecting

[7] website: https://www.linuxjournal.com/content/use-case-network-automation

[8] Packet Hub Python interacting with device

[9] website: https://hub.packtpub.com/using-python-automation-to-interact-with-network-devices-tutorial/

[10] Learning Oreilly, Mastering Python for network and security website:

[11] https://learning.oreilly.com/library/view/mastering-python-for/9781788992510/b97d457f-041a-424d-b75d-a7090d9de141.xhtml

[12] Red hat , Why start network automation

[13] website: https://www.redhat.com/en/blog/network-automation-why-organizations-shouldnt-wait-get-started

[14] Sisay Tadesse, Claire Naiga Serugunda Fabrizio Granelli et. al. (2021), A Theoretical Discussion and Survey of Network Automation for IoT: Challenges and Opportunity, August 2021IEEE Internet of Things Journal 8(15):12021-12045, DOI: 10.1109/JIOT.2021.3075901

[15] McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, (2008) http://doi.acm.org/10.1145/1355734.1355746 OpenFlow:, Enabling Innovation in Campus Networks.

[16] Anne Golinski, (2017), Automating Network System Configurations for Vendor-Specific Network Elements., Technical report/Thesis, KTH Royal institute of Technology, Stockholm, Sweden.

Contribution of Individual Authors to the Creation of a Scientific Article (Ghostwriting Policy)

The authors equally contributed in the present research, at all stages from the formulation of the problem to the final findings and solution.

Sources of Funding for Research Presented in a Scientific Article or Scientific Article Itself

No funding was received for conducting this study.

Conflict of Interest

The authors have no conflicts of interest to declare that are relevant to the content of this article.