Project Report
For
Capstone Project
On
Model-driven device provisioning

Presented by
Mayank Marwaha

As Part of
Master of Science in Internetworking
University of Alberta

Under the Supervision
of
Dr. Ali Tizghadam

Table of Contents

1 INTRODUCTION 6

1.1 CONTEXT 6
1.2 MOTIVATION 6
1.3 OBJECTIVES 6
1.4 STRUCTURE OF THE PROJECT 6

2 BACKGROUND INFORMATION 7

2.1 SOFTWARE DEFINED NETWORKING (SDN) 7
2.1.1 WHAT IS SOFTWARE DEFINED NETWORKING? 7
2.1.2 WHY SHIFT TO SDN? WHAT ARE THE ADVANTAGES OF SDN? 8
2.2 SNMP (SIMPLE NETWORK MANAGEMENT PROTOCOL) 9
2.2.1 COMPONENTS OF SNMP 9
2.2.2 DIFFERENT VERSIONS OF SNMP 11
2.2.3 SNMP OPERATIONS 11
2.2.4 IS SNMP STILL A GOOD OPTION? WHAT ARE ITS LIMITATIONS? 12
2.3 OPERATIONAL DATA VS CONFIGURATION DATA VS STATISTICAL DATA 12
2.4 DATA MODEL-DRIVEN MANAGEMENT 13
2.4.1 WHAT IS A DATA MODEL? 13
2.4.2 WHY DO WE NEED DATA MODELS? 14
2.5 YANG: THE DATA MODELING LANGUAGE 15
2.5.1 UNDERSTANDING YANG DATA MODEL WITH AN EXAMPLE 16
2.5.2 YANG FOR NETWORK OPERATORS 19
2.6 NETCONF 20
2.6.1 PROTOCOL FUNDAMENTALS 20
2.6.2 NETWORK MANAGEMENT DATASTORE ARCHITECTURE (NMDA) 22
2.6.3 NETCONF PROTOCOL OPERATIONS 25
2.7 RESTCONF 29
2.7.1 RESTCONF VS NETCONF 31
2.8 CISCO NSO 32
2.8.1 NSO ARCHITECTURE 33
2.8.2 ADVANTAGES OF USING NSO 34

3 PROPOSED SOLUTION 35

3.1 THE COMPONENTS 35
3.2 IMPLEMENTATION 37
3.2.1 STEP 1: COMPILING SNMP MIBS AND IMPLEMENTING SNMP NEDS IN NSO 37
3.2.2 STEP 2: ADDING THE DEVICES TO BE MANAGED TO NSO 39
3.2.3 STEP 3: ADDING THE DEVICES TO A GROUP WITHIN NSO 42
3.2.4 STEP 4: IMPLEMENTING SNMP COLLECTOR 43
3.2.5 STEP 5: IMPLEMENTING THE PUBLISHER 48
3.2.6 STEP 6: IMPLEMENTING DB QUERIER 55

4 LIMITATIONS 62

5 BIBLIOGRAPHY.. 63

TABLE OF FIGURES

FIGURE 2.1: DATA PLANE AND CONTROL PLANE [2] 7
FIGURE 2.2: CENTRALISED CONTROL PLANE AND A DISTRIBUTED DATA PLANE [2] 8
FIGURE 2.3: COMPONENTS OF SNMP [8] 9
FIGURE 2.4: MIB-II SUBTREE [9] 10
FIGURE 2.5: IF-TABLE [10] 12
FIGURE 2.6: IETF-INTERFACES.YANG IN TREE FORMAT 14
FIGURE 2.7: YANG AS API CONTRACT [14] 15
FIGURE 2.8: NETCONF PROTOCOL LAYERS [19] 20
FIGURE 2.9: EXAMPLE OF RPC GET REQUEST [20] 22
FIGURE 2.10: RPC REPLY TO THE RPC GET REQUEST [20] 22
FIGURE 2.11: ARCHITECTURAL MODEL OF DATASTORES [14] 24
FIGURE 2.12: EXAMPLE OF NETCONF GET OPERATION [17] 25
FIGURE 2.13: GET-CONFIG OPERATION [17] 26
FIGURE 2.14: EDIT-CONFIG OPERATION [17] 27
FIGURE 2.15: COPY-CONFIG OPERATION [17] 27
FIGURE 2.16: DELETE-CONFIG OPERATION [17] 27
FIGURE 2.17: LOCK OPERATION [17] 28
FIGURE 2.18: UNLOCK OPERATION [17] 28
FIGURE 2.19: CLOSE-SESSION OPERATION [17] 28
FIGURE 2.20: KILL-SESSION OPERATION [17] 29
FIGURE 2.21: COMMIT OPERATION [17] 29
FIGURE 2.22: RESTCONF OPERATIONS RELATED TO NETCONF PROTOCOL OPERATIONS [26] 30
FIGURE 2.23: GET REPLY FOR WELL-KNOWN HOST-META INFORMATION 30
FIGURE 2.24: GET REQUEST ON THE ROOT URL OF RESTCONF SERVER 31
FIGURE 2.25: USE CASE FOR RESTCONF AND NETCONF [14] 32
FIGURE 2.26: CISCO NSO ARCHITECTURE [28] 33
FIGURE 3.1: MAPE-K LOOP [29] 35
FIGURE 3.2: PROPOSED SOLUTION 35
FIGURE 3.3: MIB CORRESPONDING TO OID [29] 37
FIGURE 3.4: IMPORT SECTION OF IP-MIB.MIB 37
FIGURE 3.5: CONTENTS OF MIB DIRECTORY 38
FIGURE 3.6: COMMANDS TO COMPILE MIBS TO SNMP NED 38
FIGURE 3.7: OUTPUT OF MAKE COMMAND 38
FIGURE 3.8: CONTENTS OF /VAR/OPT/NCS/PACKAGES 39
FIGURE 3.9: PACKAGES RELOAD IN NSO 39
FIGURE 3.10: CREATING SNMP-GROUP AUTHGROUP 40
FIGURE 3.11: ADDING A DEVICE TO NSO 40
FIGURE 3.12: CONNECTION STATUS OF DEVICES ADDED TO NSO 41
FIGURE 3.13: PERFORMING LIVE STATUS ON A DEVICE FROM NSO 41
FIGURE 3.14: LIVE STATUS COMMAND FROM NSO ON A DEVICE WITH OUTPUT IN JSON FORMAT 42
FIGURE 3.15: ADDING DEVICES TO A GROUP IN NSO 42
FIGURE 3.16: RESTCONF QUERY TO THE GET DEVICES PART OF SNMP GROUP 43
FIGURE 3.17: CHECKING CONNECTION BETWEEN NSO AND THE DEVICE 43
FIGURE 3.18: CURL OPERATION TO GET THE ARP INFORMATION FROM SNMP-NOKIA-2 44
FIGURE 3.19: DEFINITION OF GET_DEVICES_FROM_NSO() 45
FIGURE 3.20: GET_ARP(DEVICE_NAME) FUNCTION 45
FIGURE 3.21: REST ENDPOINT FOR SNMP-COLLECTOR 46
FIGURE 3.22: GET OPERATION ON /DEVICES 47
FIGURE 3.23: RELATIONSHIP DIAGRAM OF DATABASE 48
FIGURE 3.24: SQLALCHEMY MODEL FOR OPERATIONALDATATYPE 49
FIGURE 3.25: SQLALCHEMY MODEL FOR TIMEDTRANSACTIONS 49
FIGURE 3.26: SQLALCHEMY MODEL FOR ARP 50
FIGURE 3.27: CRUD_OPERATIONAL_DATA_TYPE.PY 50
FIGURE 3.28: CRUD_TIMED_TRANSACTIONS.PY 51

Figure 3.29: crud_arp.py 52
Figure 3.30: publisher.py 53
Figure 3.31: OperationalDataType table 54
Figure 3.32: TimedTransactions table 54
Figure 3.33: arp_info table 55
Figure 3.34: Swagger endpoints for arp.yang 58
Figure 3.35: function get_arp_all_devices() 58
Figure 3.36: all the records of TimedTransactions 59
Figure 3.37: GET operation on /devices endpoint 59
Figure 3.38: Function get_arp_single_device(device) 60
Figure 3.39: Curl on arp:devices=snmp-xr/ipNetToMediaTable/ipNetToMediaEntry endpoint 60
Figure 3.40: Function get_arp_filter_for_ip_and_index() 61
Figure 3.41: GET on third endpoint 61

1 INTRODUCTION

This chapter describes the context, motivation, objective and limitations of the study.

1.1 Context

The traditional network OAM (Operations, Administration and Management) had network engineers as the consumers who used the data collected from network OAM to monitor and diagnose the networks, but these tools are not enough for the modern networks where the consumer of all this operational and management data is not humans but rather other applications or services [1]. This is where term Network Telemetry comes in. Network telemetry means using automated data collection processes for collecting operational data from the network devices, analyzing this data and further use this data for different use cases like anomaly detection, network performance etc. which can be further used to improve network optimization techniques like real time load balancing, traffic engineering and network planning [1].

1.2 Motivation

With the increased adoption of SDN and machine learning based artificial intelligence, data obtained from the networks using network telemetry can be used to detect network faults, anomalies, policy violations and can further be used to prevent these incidents [1]. With such use cases, the aim is to reduce the human labor, provide better services to the customer and to efficiently use the network resources in place [1]. Network operators have both new as well as legacy devices in their network. The presence of legacy devices in the network poses a challenge to implementing network telemetry throughout the network. The unstructured data from conventional OAM techniques like CLI hinder the tool automation and application extensibility [1]. SNMP seems to be the only choice when it comes to getting structured data from the legacy devices. Another major concern is to come up with a solution that will be compatible with devices from different vendors.

1.3 Objectives

This project aims to discuss various technologies that can be used together to collect operational data from network devices irrespective of the vendor. The major focus of this project will be on collecting operational data from the legacy devices. Keeping the above-mentioned challenges in mind. This framework will be capable of collecting data from the legacy as well as the modern network devices that can further be exposed using API to any northbound applications which can utilize this data to provision new services or network devices.

1.4 Structure of the Project

The project is divided into 5 chapters:

- Introduction: This chapter provides the context, motivation and objects of the project
- Background Information: This chapter provides a deep insight into the various technologies that I studied during the course of project. Each technology discusses its advantage over the other and some technical details.
- Proposed Solution: This chapter contains the proposed solution and implementation details of the project.
- Limitations: This chapter discusses the limitations of the approach.
- Bibliography: This chapter provides references to all the resources referred during the project.

2 BACKGROUND INFORMATION

This chapter discusses the various technologies and terms that needs to be understood before understanding the network telemetry framework developed later in this study.

2.1 Software Defined Networking (SDN)

2.1.1 What is Software Defined Networking?

In order to understand SDN in a better way we need to understand what are the main functions that a networking device performs. Whatever a network device does can be categorized as being in a particular plane. There are three planes that define the functionality of the networking device completely and these are:

a. **The Data Plane**: Data plane consists of everything a router or switch might do when receiving, processing and forwarding a message. It involves de-encapsulating and re-encapsulating the packet in a data link frame, matching the frame's destination MAC address with the MAC address table, matching the destination IP address to the IP routing table and then re-encapsulating the packet with the information necessary to forward the packet [2].

b. **The Control Plane:** Control plane supplies the information that the data plane needs to know beforehand so that it can work properly. It involves creating tables like IP routing tables, MAC address tables etc. and modifying the entries in these tables. For example, the OSPF process running on each router in the network runs in the control plane and makes changes to the routing table. Based on these changes in the routing table, the data plane further makes the packet forwarding decision. Some other control plane protocols are EIGRP, BGP, ARP, IPV6 NDP, STP etc. [2]

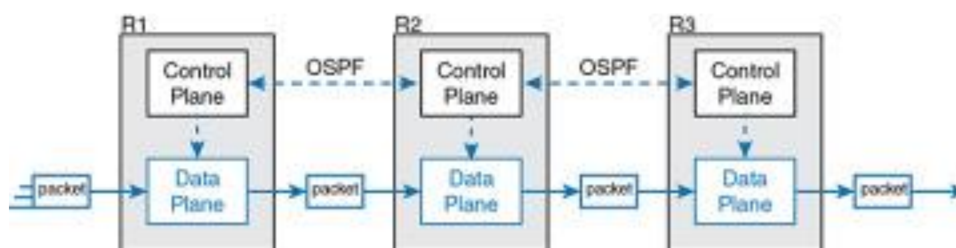

Figure 2.1: Data plane and Control plane [2]

c. **The Management Plane:** Management plane includes the protocols that allows the network engineers to manage the device. Telnet and SSH are two of the most commonly used management protocols [2].

Traditional networks used distributed control planes where each device had their own instance of control plane process that communicated with each other using protocol messages. Now imagine if instead of having a distributed control planes, there was a centralized software acting as the control plane that used protocol messages to learn information from the devices but did all the processing of the information at a centralized location. This is the key factor in defining Software Defined Networks [2].

According to Open Network foundation:

> *"In the **SDN architecture**, the control and data planes are decoupled, network intelligence and state are logically centralized, and the underlying network infrastructure is abstracted from the applications. As a result, enterprises and carriers gain unprecedented programmability, automation, and network control, enabling them to build highly scalable, flexible networks that readily adapt to changing business needs."* [3]

The centralized control plane in this case is called the controller. The controller itself can create a centralized repository of all the useful information about the network.
In figure 2.1 which depicts a centralized control plane and a distributed data plane, all the network devices sit below the controller. There exists a software interface or API which lets the program on controller to communicate with the program on networking devices to let the controller program the data plane of networking devices. Since in most networking and architecture diagrams, this interface sits below the controller hence the interface is called as a Southbound Interface. [2]

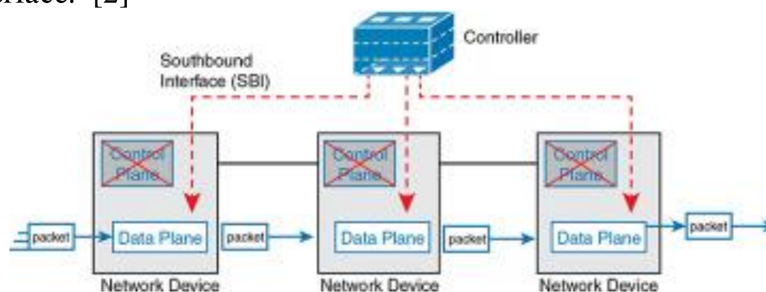

Figure 2.2: Centralised Control Plane and a Distributed Data Plane [2]

In order to make this architecture truly programmable, other applications and programs should be able to program the network. To facilitate that, controller is exposed to other applications and services by another interface or API which is called Northbound Interface since it sits on top of the controller.

2.1.2 Why shift to SDN? What are the advantages of SDN?

In the traditional networks, once the network has been designed and implemented, the only way to make an adjustment to the flow is via changes to the configuration of devices which demand expert skills, change requests, maintenance window and sometimes manual rollback. This makes the network operations complex with the need to maintain, manage and upgrade heterogenous infrastructure. On the other hand, SDN architecture offers a centralized, programmable network that can support the dynamic nature of future networks. [4] [5]
Here are some of the benefits that come along with adopting the SDN architecture:

a. Direct Programmability: Automation tools like OpenStack, Ansible, Puppet, Chef etc. can be used to programmatically configure the network which is only possible because the control plane is decoupled from the data plane. [6]
b. Centralized management: The SDN controller maintains the global view of the network and therefore it is easy to implement to make changes to services without the need to remotely login to every device on the network. [6]
c. Reduced Capex: SDN follows a pay-as-you-grow model when it comes to scalability. Most of the networking devices in the market these days support SDN capabilities and hence in order to scale the network, all you need is a infrastructure composed of devices with SDN capabilities. [6]
d. Reduced Opex: Since the updates can be pushed automatically to the network software uniformly, there is no need to change the infrastructure when the business need arises. Also, most of the monotonous tasks which were earlier being done by the network administrators can be automated which not only reduces the chances of human error but also makes the network management efficient timewise. [6]
e. Agility and flexibility: SDN can help organizations rapidly deploy new applications, services and infrastructure to quickly meet changing business goals and objectives as a simple deploy will be enough to make the changes network wide. [6]

f. Openness: All of the SDN controllers support API which can be used for a wide range of applications like cloud orchestration, business-critical networked apps. Using the APIs, the network operators can write services that can utilize the SDN APIs to give the applications control over the network behavior. These network aware applications can monitor network and automatically adapt to the network conditions as needed. [7]

2.2 SNMP (Simple Network Management Protocol)

SNMP originated in 1988 when the number of IP devices in networks was growing in size. SNMP is a network management protocol that allows network administrators to remotely access or change the management information on a network device. SNMP operates on the application layer.

2.2.1 Components of SNMP

SNMP consists of the following components:

a. **SNMP Manager:** Manager is a device that runs the Network Management System (NMS) and polls SNMP agents running on the managed devices periodically to collect information from the devices or receives traps from the management agents in the network. This information can then be presented to the network administrator using GUI or can be used to take appropriate action [8] [9]. Polling is a pull-based method of collecting information from the management agents where the manager queries for information whereas traps are push based and are sent by the management agents in case some alarming event is noticed on the side of management agents.
b. **Managed Device:** Network devices like PCs, servers, routers and gateways which are managed by the manager are called managed devices. These devices house the management agents.
c. **Management agents:** Management agent is a software running on managed devices that stores information related to the device. It responds to the manager's queries and generates traps to inform the manager about certain events. [8] Since trap are push based, they are considered a better and efficient way to collect information as there is no delay in reporting the event and it does not consume as much bandwidth or resources as polling does.
d. **Protocol:** SNMP is the protocol. It uses UDP as the transport layer protocol. Traps are sent to manager with the destination port of 162. The manager sends queries to management agents with the destination port of 161 and a random source port.

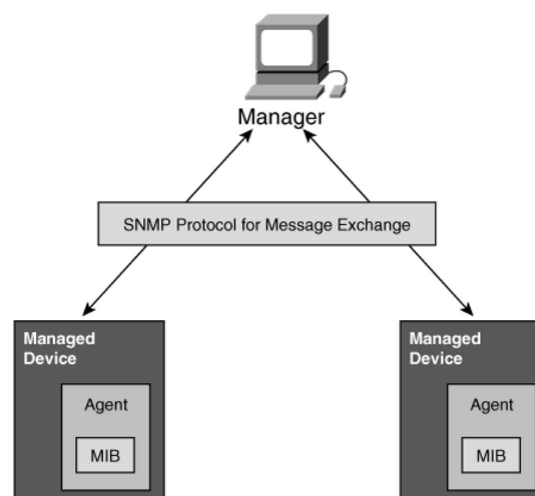

Figure 2.3: Components of SNMP [8]

e. Management Information Base (MIB): MIB is the database that stores the information related to all the managed objects that an agent track. The Structure of Managed Information (SMI) file defines how the managed objects are named and provides their associated datatypes whereas the managed objects are themselves defined in the MIB. Different agents from different vendors implement different MIBs but all the network devices have to compulsorily implement MIB-II or RFC 1213 which contains general information related to TCP/IP. [9]

It is very essential to understand the MIB and the structure of MIB in order to understand the proposed solution implemented later. Managed objects are defined using these three attributes:

i. Name: Managed objects are organized in a tree-like structure. Each managed object is uniquely defined by a OID or object ID. The object ID is made of integers which represent the different nodes as we traverse the Object tree. The Object ID can also be represented in a human readable format which is nothing, but the name of each node separated by dots as we traverse the Object tree. [9]

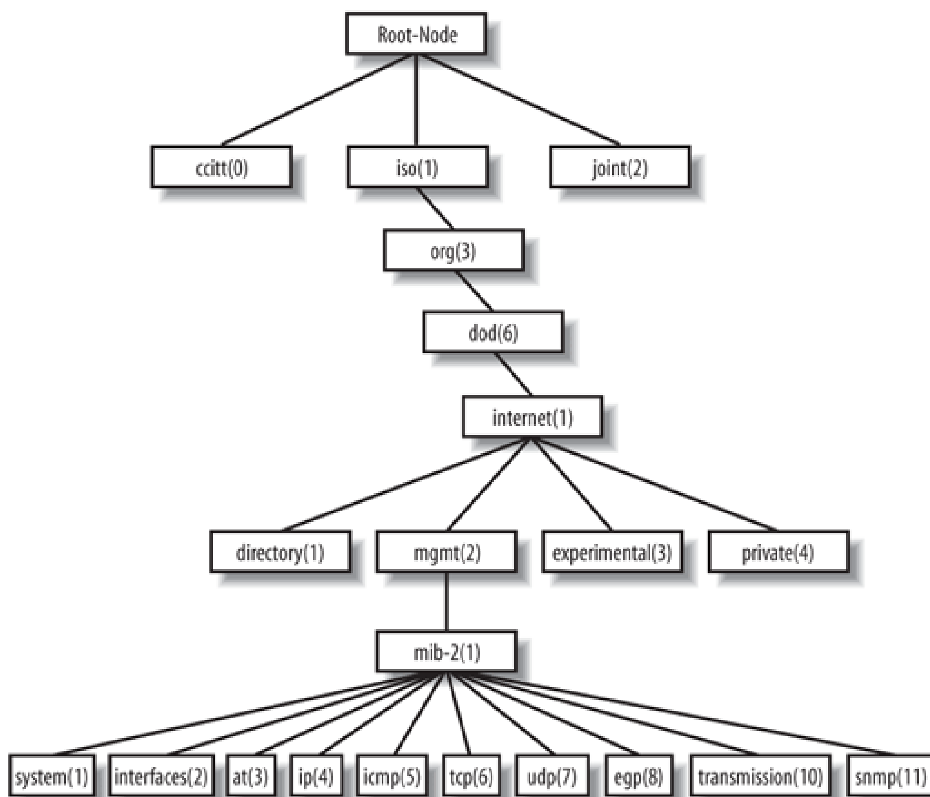

Figure 2.4: MIB-II Subtree [9]

Just like in any tree, the node at the top of Object tree is called the root node and anything under that is called the subtree. Figure 2.4 shows the MIB-II management group. Every device that supports SNMP has to provide the MIB-II management group. So IP management group under MIB-II can be accessed as 1.3.6.1.2.1.4 or iso.org.dod.internet.mgmt.1.4

ii. Type and Syntax: Some of Abstract Syntax Notation One (ASN.1) are used to define the data type of a managed object. ASN.1 specifies how data is represented and transmitted between the managers and agents. ASN.1 is machine independent and so we need not worry about byte ordering when a Unix machine is communicating with the windows machine. [9]

iii. Encoding: Managed objects are encoded and decoded using the Basic Encoding Rules (BER) for transmission over a transport medium. [9]

2.2.2 Different Versions of SNMP

a) **SNMPv1** is the first version of the protocol that was introduced in 1988. It is described in RFC 1157. SNMPv1 uses community strings to establish the trust between the manager and the agent. Community strings are nothing but passwords. Each managed agent is configured with three community strings, each serving the following purpose: read-only, read-write and trap. Read-only string allows the manager to just access the managed objects with read-only access rights. Read-write string allows the manager to change the values as well and trap string is used to receive traps. Even though SNMPv1 has been replaced by SNMPv2 and SNMPv3 it is still the most used of all the other versions. SNMPv1 supports Get, GetNext, Set and Trap operations [9]

b) **SNMPv2** was the next release of SNMP protocol. Not much was changed in the SNMPv2. Just like SNMPv1 it uses community strings as well. GetBulk, Inform were added as new operations to the protocol and the PDU format of trap was modified to be the same as that of Set and Get. [9]

c) The most recent version **SNMPv3** supports strong authentication between the managed entities. SNMPv3 uses USM by default. Another operation Report was added to the protocol in this release [9]

2.2.3 SNMP Operations

Manager and the managed agents communicate the information using a set of operations. SNMP supports the following operations:

a. Get: The get request is issued by the manager to the agent. The get request contains the snmp version, community string or authentication details along with the object ID of the object that the manager wants to access. The managed agent replies with a getresponse back to the manager. This type of request is best suited for retrieving a single MIB object at a time. [9]

b. Getnext: When the manager wants to retrieve a group of values from the managed agent, it issues a getnext command for each MIB object and the managed agent replies with a separate getresponse for each getnext. Just like the get request, the manager provides the snmp version and accordingly either provides the community string or authentication details along with the managed object. [9]

c. Getbulk: The getbulk operation is only available in SNMPv2 and SNMPv3. It allows the manager to request a section of a table at once. The request contains the snmp version along with community string or authentication details, max repetition and nonrepeater values. [9]

d. Set: The set operation is used to make changes to the managed object. The set request contains the snmp version along with the community string or authentication details, managed object id and the new value you want to set it to. [9]

e. Getresponse: the managed agent replies with getresponse every time a manager requests something with the get, getnext or getbulk operation. [9]

f. Trap: The trap originates at the managed agent and is sent to the trap destination which in most of the cases is the manager. The manager should be capable of interpreting the trap. The trap numbers can vary from 0 to 6. For the traps 0 to 5, the knowledge of what the trap contains is defined within the NMS itself. The trap number 6 is enterprise specific which is defined by the manufacturer of product. [9]

g. Notification: Since the pdu format of trap is different from the get and set request in case of SNMPv1 thus SNMPv2 defined Notification which has the same pdu format as that of the get and set operation. [9]
h. Inform: SNMP inform is exclusive to SNMPv2 and SNMPv3. Whenever an event is sent, the receiver acknowledges that it received the event by sending a SNMP inform. [9]
i. Report: SNMP Report was introduced as a draft in SNMPv2 but was officially implemented in SNMPv3. It was developed to provide a mechanism for communication between SNMP engines. [9]

2.2.4 Is SNMP still a good option? What are its limitations?

With SDN gaining popularity and the number of network devices in the network of an organization continuously increasing, it has become very hard to manage all these devices with SNMP. SNMP has helped the network administrators a lot when the Internet was gaining popularity, but it has failed to keep up with the scale and speed that modern networks require. [10]

SNMP polling is based on pull. To get large amounts of data from the management agents, SNMP uses GetBulk operation which further sends a series of GetNext operations. The poller receives as many entries from an object table as the IP packet can carry across the network. It keeps sending the GetNext requests until the poller detects that the management agent is now sending the entries from the next object table. In this case each request is individually processed by the router. More than that if you have more than one poller then that further creates delay as now the router has to keep the record of all the incoming requests separate and thus the SNMP response gets slow. [10]

ifIndex	ifDescr	ifType	ifMTU	ifSpeed
3	GigibitEthernet0	6	1514	1000000000
4	GigabitEthernet1	6	1514	1000000000
5	Loopback0	24	1500	0

Figure 2.5: IF-Table [10]

Another issue is the way data is ordered in the object table. Let us consider the above table in figure 2.5. If a GetBulk request is made to obtain the table, the internals of router fulfil the request by going through each column (returning a list of ifIndex, followed by ifDescr and so on) one by one. Even though methods like indexing can be implemented in the router object table but implementing such mechanisms means more processing work as well as increased retrieval time which makes the data stale. [10]

AI and Machine learning techniques can be used to perform data analysis on the operational data obtained from the network devices. This data analysis can further be put to use for anomaly detection and improving network performance etc. but due to slow response times of SNMP, this is not possible in a large network.

2.3 Operational data vs Configuration data vs Statistical data

Configuration data is the changes done to a network device to change the initial state of the device to its current state. [11]

Operational state data is the data that has been obtained from the network device at runtime and unlike configuration data is modified by interactions with internal components or by other systems using specialized protocols. [11]

Statistical data is read only and provides information about the performance of system as well as the various components in the system. [11]

Let us consider the following examples to make the difference clear:

a) IP Routing table: When you statically configure a route in the router that will be considered as the configuration data. When a router learns about other networks available in the topology due to a routing protocol running on it, then the learned information about network is called the operational data. In addition, the routing process on a router may also provide information like how often the routing entry was used. [11]
b) MTU value on Interface: Every port has a default speed or MTU value that it uses by default when it detects a particular type of cable plugged in. This value becomes the operational state associated with the interface when the interface is being used. The value of MTU or speed can be configured on the interface. This will be configuration data for the interface. The interface can record statistics for the number of bytes, packets received on the interface and this will be considered the statistical data. [11]
c) Account Information: Configurations about the accounts locally on the system. Further additional information can be obtained using protocols like LDAP dynamically which counts as the operational state data. Information like account usage will be considered statistical data. [11]

2.4 Data Model-Driven Management

2.4.1 What is a Data model?

In simple words, a Data model is nothing but an agreed upon and commonly understood method to describe something. Consider that you have to describe a person, the simple data model for this person will be [12]

--Person

    --Gender: male, female, other
    --Height: in feet, inches or centimeter
    --Weight: kgs or lbs
    --Hair Color: Brown, Blonde, Black, other
    --Eye Color: Brown, Black, Green, other

According to RFC 3444, Data Models define managed objects at a lower level of abstraction. They include implementation- and protocol-specific details, e.g. rules that explain how to map managed objects onto lower-level protocol constructs. [13]

Data models can be represented using the following formats:

- Yang
- XML
- JSON
- HTML/JavaScript

Yang is most commonly used language for making Data models in networking world these days. Different organizations are working together to come up with the industry standard yang data models as well as the specific-to-vendor data models.
The yang data models from standard organizations like IETF, open-source projects like Open Daylight, vendors can be accessed at: https://github.com/YangModels/yang

A Data model can be used to describe the

- Device Data Models: Interface, VLAN, Device ACL, Tunnel, OSPF, etc. [12]
- Service Data Models: MPLS VPN, BGP, VRF, etc. [12]

```
Desktop $ pyang -f tree ietf-interfaces.yang
module: ietf-interfaces
  +--rw interfaces
  |  +--rw interface* [name]
  |     +--rw name                        string
  |     +--rw description?                string
  |     +--rw type                        identityref
  |     +--rw enabled?                    boolean
  |     +--rw link-up-down-trap-enable?   enumeration {if-mib}?
  +--ro interfaces-state
     +--ro interface* [name]
        +--ro name               string
        +--ro type               identityref
        +--ro admin-status       enumeration {if-mib}?
        +--ro oper-status        enumeration
        +--ro last-change?       yang:date-and-time
        +--ro if-index           int32 {if-mib}?
        +--ro phys-address?      yang:phys-address
        +--ro higher-layer-if*   interface-state-ref
        +--ro lower-layer-if*    interface-state-ref
        +--ro speed?             yang:gauge64
        +--ro statistics
           +--ro discontinuity-time    yang:date-and-time
           +--ro in-octets?            yang:counter64
           +--ro in-unicast-pkts?      yang:counter64
           +--ro in-broadcast-pkts?    yang:counter64
           +--ro in-multicast-pkts?    yang:counter64
           +--ro in-discards?          yang:counter32
           +--ro in-errors?            yang:counter32
```

Figure 2.6: IETF-INTERFACES.yang in tree format

Figure 2.6 shows the ietf-interfaces.yang which is the IETF standard data model for an interface in a tree format. The output is obtained using a python library called PYANG which is used to validate the yang data models. As you can see the Data model describes the various properties of an interface and also mentions if they are RO – read only or RW – read write. RO is for the operational or statistical data whereas RW is for the configuration data related to the object.

2.4.2 Why do we need Data Models?

The need for Data models came from the need of automation in the networking equipment. Even though CLI and SNMP were being used for configuration and management purposes, let's see where they fail, and this will help us understand why there was a need for data models.

The cli of a networking device is meant to be user-friendly for the network engineers. While cli may be the most interactive way of handling networking equipment, it most certainly is not the most efficient way to introduce new service in the network. For network operators, launching a new service sometimes involves doing changes to thousands of devices sometimes which would mean that a network expert would have to login to each and every one of these devices to make the changes. Using highly trained network experts for inserting commands into the network devices is a wastage of resource and time as well. More than that since the commands are being typed by a network professional manually, there is a high probability of fat-finger typing which can bring the whole network down. What if a need arises to make the changes to this network service? Will this entire process be repeated again? Using tools like expect and libraries like paramiko, the cli can be automated but only to a limit because of the following reasons:

a) The cli is not standardized across all the vendors. Different vendors use different operating system and thus different set of commands for a particular configuration. [14]

b) Dependency issues while configuring the devices using cli is another issue. For example, a vlan has to be created in the networking device before an interface can be

assigned to the vlan. The configuration fails and, in some cases, it is partially completed. [14]

c) Cli provide no error reporting while configuring the devices using scripts. This makes it difficult to cover the edge cases with scripts. [14]
d) The output obtained from cli is not structured. The only way to extract data from the output of scripts is using regex or screen scraping which is not the best method. A slightest change in the output can break the script. [14]

As explained in the SNMP section above, snmp is not a good option for bulk transfers and is not efficient when fetching information from the device. More than that SNMP can only be used for monitoring but not for making the changes to configuration.

In order to make the management and configuration of network devices easier model driven architecture was introduced. Adopting the model driven architecture provides an API that has the following benefits:

a) Abstraction: a programmable API abstracts the underlying implementation complexities. Thus, making configuration changes does not involve making changes in a particular order. It is more like filling a checklist. [14]
b) Data specification: This API provides how the data is organized and what the type of data is. [14]
c) Means of accessing data: the API provides a standardized way of reading and manipulating device data. [14]

Thus, Data model-driven management builds on the idea of specifying in models the semantics, the syntax, the structure, and constraints of management objects. [14] The advantage is that, as long as the models are updated in a backward-compatible way, the previous set of APIs is still valid. [14]
[15]

2.5 YANG: The Data Modeling Language

As per RFC 7950 YANG, Yet Another Next Generation language, can be defined as:
"YANG is a data modeling language used to model configuration data, state data, Remote Procedure Calls, and notifications for network management protocols."

Yang is an API contract language that is used to write the specifications of an interface between a client(controller or an application) and a server(network element or an application) on a particular topic. A Yang module consists of a specification and a set of Yang modules forms the Yang model. [14]

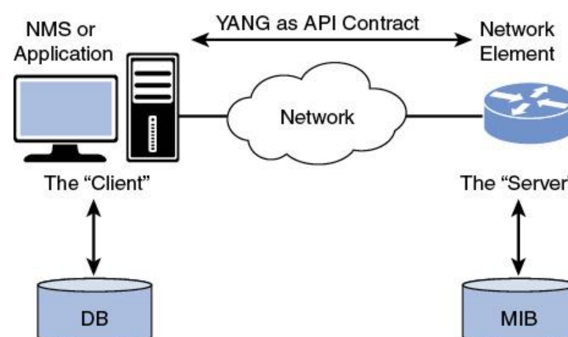

Figure 2.7: Yang as API Contract [14]

As shown in figure 2.7, the Yang based server publishes its Yang modules which taken together form the server's Yang model. On the basis of what the yang model contains, these yang models can be used to configure, monitor status, receive notifications from server and invoke actions on the server. [14]

2.5.1 Understanding YANG data model with an example

Let us consider table 1 as an ARP table obtained from the device

IfIndex	PhysAddress	NetAddress	MediaType
1	6c:9c:ed:11:73:a5	10.10.55.3	dynamic
2	f8:6b:d9:10:98:28	172.25.1.1	static
102	78:ba:f9:11:47:89	172.25.139.249	dynamic
201	f4:b5:2f:40:b8:e1	10.1.2.3	other

Table 1: ARP Table

The Yang model corresponding to the ARP table is:

```
module ArpTable{
    yang-version 1.1;
    namespace http://www.example.com/netconf/yang/arp;
    prefix arp;

import ietf-inet-types {
    prefix inet;
  }

import ietf-yang-types {
    prefix yang;
  }

revision 2021-02-08 {
    description
      "Initial revision of ARP table";
  }

container ArpTable {
        config false;
        list ArpEntry {
          key "NetAddress";

          leaf IfIndex {
            description
               "Interface index";
            type int32
            {range "1..2147483647";}
          }

          leaf PhysAddress {
            description
               "Mac Address associated with the IP address";
            type yang:phys-address;
```

```
          }

          leaf NetAddress
          {
            description
               "IP address associated with the MAC address";
            type inet:ipv4-address;
          }

          leaf MediaType {
            description
               "specifies how the binding was learned";
            type enumeration {
              enum "other"
                    {
                    value 1;
                    }
              enum "invalid"
                    {
                    value 2;
                    }
              enum "dynamic"
                    {
                    value 3;
                    }
              enum "static"
                    {
                    value 4;
                    }
             }
          }
       }
    }
 }
```

Please note that the data model described here is just to demonstrate an example and is not complete.
Each of the Yang module starts with the keyword `module` which is followed by the module name. In our case the name of the module is `ArpTable`. The name of the file must be same as the module name followed with a `.yang` extension. [14] A module is the smallest unit for defining models in YANG. A module contains only one data model. A module can also include other existing data models. [15]

The `yang-version` states that this particular module is written in yang version 1.1. `namespace` is used to give a unique name to the module. No two modules can have the same namespace and it is up to the yang developer to make sure of that. Ideal suggestion is to include the domain name of the organisation in the namespace. [14]

The `prefix` statement defines the prefix associated with the particular module. Since the namespace section is long, it becomes hard to refer to the definitions inside the module with

the namespace and thus there is the prefix which is the abbreviation of the module name. the prefix name should be unique among the set of yang modules being used in the organisation otherwise it leads to confusion while reading as well as importing. In our case we have used to the prefix arp. [14]

The `import` statement makes all the definitions and types in the imported module available to the current module. [14] The prefix statement is mandatory as it is used to assign a prefix to the definitions in the imported module in order to define it in the scope of the calling module. [15] In the above example we are importing two modules namely: `ietf-inet-types` and `ietf-yang-types`. The `revision` statement is optional, it is not required but it is a good practise to add the revision date to the module as it is published. The revision statement contains the revision history of the model. The format of date string is "YYYY-MM-DD". You can further add a description section to define the changes in each revision. [15]

Now comes the definition part where the specifications of the managed object is defined. A Yang `container` is a collection of information elements that belong together [14]. A container has only child nodes and no value. It can contain any number of child nodes which can be leafs, lists, containers etc. In our case `ArpTable` is the container for all the arp entries of a device. The config is set to false which means that the data under this container is read only and cannot be changed. Config set to false represents the operational data. If config is set to true then that means that the data elements under that particular node can be changed and hence the model represents the configuration data.

A Yang `list` works like a container and is a sequence of list entries. [14] A list can have many child nodes. Each entry in the list is uniquely identified by its key leafs [15]. In our case a single `ArpTable` will have many `ArpEntry`. The key leaf is just like the primary key in the database. The key can be from one child leaf node or multiple child nodes. [15]

Under the list `ArpEntry` we see many `leaf` nodes. Each leaf node represents a column of the Arp table shown in table 1. A leaf has no child nodes and is of a particular type [15]. By default all the leafs are optional unless they are marked as key or mandatory. [14] Each `leaf` has a `type` statement which describes the kind of data this `leaf` can hold. You can define the type as one of the 19 built in types or define the derived type which are made from built-in types. Some of the example built-in types are Boolean, bits, binary, enumeration, int8, int32, int64, string, uint16, uint32, uint64 etc. [15] In the data model of ArpTable we have used `int32` for `IfIndex` with the values in the range of 1 to 2147483647, a derived type called `phys-address` from the module `ietf-yang-types` for PhysAddress, a derived type called `ipv4-address` from the module `ietf-inet-types` for the IP address and enumeration type where the values `other, invalid, dynamic, static` are given the integer values 1,2,3,4 respectively.

The definition of `phys-address` from `ietf-yang-types` and `ipv4-address` from `ietf-inet-types` is available as part of RFC 6021 [16] as :

```
typedef phys-address {
      type string {
        pattern
          '([0-9a-fA-F]{2}(:[0-9a-fA-F]{2})*)?';
```

```
      }
      description
        "Represents media- or physical-level addresses
represented as a sequence octets, each octet represented by
two hexadecimal numbers. Octets are separated by colons.  The
canonical representation uses lowercase characters. In the
value set and its semantics, this type is equivalent to the
PhysAddress textual convention of the SMIv2.";
      reference
        "RFC 2579: Textual Conventions for SMIv2";

    } [16]
```

```
typedef ipv4-address {
     type string {
       pattern
         '(([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-
5])\.){3}' + '([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-
5])' + '(%[\p{N}\p{L}]+)?';}
     description
       "The ipv4-address type represents an IPv4 address in
dotted-quad notation.  The IPv4 address may include a zone
index, separated by a % sign. The zone index is used to
disambiguate identical address values.  For link-local
addresses, the zone index will typically be the interface
index number or the name of an interface.  If the zone index
is not present, the default zone of the device will be used.
The canonical format for the zone index is the numerical
format";
} [16]
```

Both the derived type `phys-address` and `ipv4-address` are formed from built-in type `string` using regular expression `pattern` statement.

In order to get the ARP entry corresponding to a particular IP address using the above data model, we would use something called as the XPath. Using the XPath /ArpTable/ArpEntry[NetAddress="10.10.55.3"]/ we could access the ArpEntry for net address of 10.10.55.3

2.5.2 YANG for Network Operators

During the 2002 IAB Network Management Workshop, many network operators raised issues related to network management architectures that were being used. Strengths and weaknesses of various technologies were assessed, and network operators came up with a number of requirements for the management architecture that the industry must follow. YANG was one of the outcomes of that workshop and it addresses the following requirements:

a) Separation between the configuration data and state data: Yang makes it easier to differentiate the configuration data from the state data with the help of `config` statement. If the `config` is set to True, that would imply that the data can be changed or configured but if it is set to False then that makes the data as just read only

or operational. All the child nodes under a `container` or `list` with `config` as false will be considered as part of the operational data. [14]

b) Easy to Use: The syntax of Yang is human friendly and easily readable just like some of the programming languages. Many of the issues remain unresolved in Yang but its ease of usage is attracting many developers. [14]

c) Generation of Deltas: Since every managed object is represented using a yang data model, it is very easy to generate the delta between the two configuration data sets. This makes it easier to switch back to old configuration quickly in case the new configuration creates problems. [14]

d) Task Oriented: Using data models, not only can you map the leafs nodes with the underlying network data, but you can also define specific tasks as RPC operations which can act on this underlying data. [14]

e) Full coverage: All the capabilities of the network device can be accessed with a proper data model. Due to all this, there is no more need to CLI based tools. [14]

f) Simple data modeling language: Yang data models make it very easier to integrate the network devices into the infrastructure or with another application. [14]

g) Timeliness: It is very easy to tie the cli operations to a yang module, thus making all the operations available immediately. [14]

h) Implementation difficulty: The ease of implementation of new modules in Yang makes it is very easy to add new features and integrate them with the current data models. [14]

2.6 NETCONF

NETCONF is a network management protocol defined by IETF in RFC 6241 to "install, manipulate, and delete the configuration of network devices" [17]. It provides a set of operations built on top of remote procedure calls (RPC) using xml encoding and are used to get and edit the configuration of network devices. [18] NETCONF was the result of complaints brought forward during the 2002 IAB Network Management Workshop by network operators who were struggling with the network management tools at that time.

2.6.1 Protocol Fundamentals

NETCONF uses a client server architecture based on the RPC calls. In this case NETCONF client is the application or script running as part of the network manager whereas the NETCONF server is the managed network device [17]. NETCONF client communicates with the NETCONF server over a session using a series of XML structured messages. The messages in this case are either the remote procedure call or RPC reply [14].

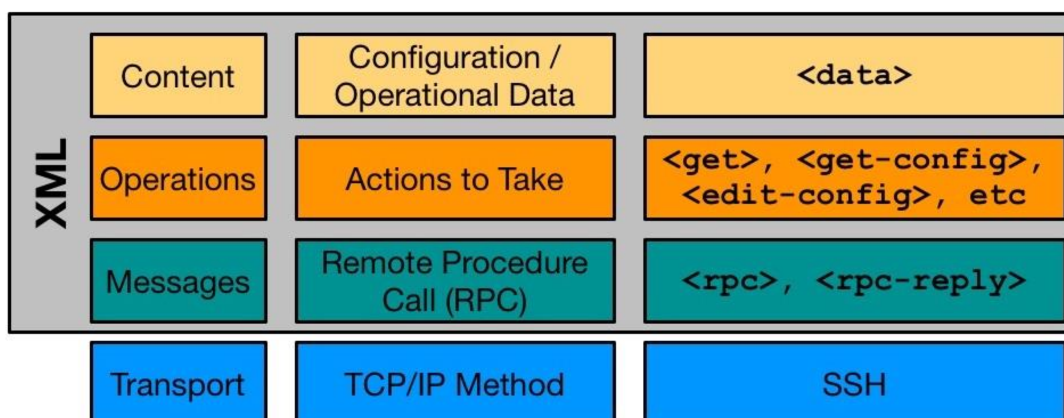

Figure 2.8: NETCONF protocol layers [19]

All this communication occurs on top of SSH session using the concept of SSH subsystem. Figure 2.8 depicts the various layers of the NETCONF protocol. IANA has assigned port 830 as the default port for NETCONF. As per RFC 6241:
"*A device MUST support at least one NETCONF session and SHOULD support multiple sessions. Global configuration attributes can be changed during any authorized session, and the effects are visible in all sessions. Session-specific attributes affect only the session in which they are changed.*" [17]
When a client wants to connect to the NETCONF server, it opens only one channel. If the client wants to perform multiple things in parallel on a particular device, it can create another channel to perform the particular activity. For example, the client can have three SSH channels in a single SSH connection, one to configure the device, another to read the operational status from the device and the third to read notifications in case of any change in the operational state of a device. [14]
When a NETCONF session is opened between a NETCONF client and NETCONF server, both the sides share their capabilities as part of the hello message. The server replying with the hello message must include the session id for this particular NETCONF session in the message. The client does not include a session id in the hello message sent from its side. [17]
As part of the hello message, following capabilities are declared:

- The version of NETCONF supported by server and client (1.0, 1.1 or both).
- The YANG data models supported by server and client.
- The optional capabilities supported by both the parties.

In order to connect to a NETCONF capable device from the unix/linux terminal, enter the following command:

```
ssh -2 -l root -p 830 ios-xe-mgmt.cisco.com
```

In the above command `root` is the username and `ios-xe-mgmt.cisco.com` is the device acting as NETCONF server.

The hello message received from the device looks something like this:

```
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    <capability>urn:ietf:params:netconf:base:1.1</capability>
    <capability>urn:ietf:params:netconf:capability:writable-
running:1.0</capability>
...
 </capabilities>
 <session-id>1132</session-id>
</hello>
```

Once the client receives the capabilities from the NETCONF server as part of the hello message, the client knows the data models supported by the server(device). The client communicates with the server using Remote Procedural Calls. The client sends RPC requests to the NETCONF server and the server replies with RPC replies. The RPC reply can be a simple OK message or an rpc-error. The server tracks each RPC with the message id and replies back the same message id in the rpc reply. NETCONF supports different RPC operations which will be explained later. For an example let's consider the get operation as shown in figure 2.9 in order to understand the structure of a rpc request.

```
<rpc message-id="1" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
  <get>
    <filter xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
      <interfaces>
        <interface>
          <name>eth0</name>
        </interface>
      </interfaces>
    </filter>
  </get>
</rpc>
```

Figure 2.9: Example of RPC get request [20]

In the above RPC request, the message-id is 1. Xmlns is used to specify the capability that the server supports and thus this particular yang module is referenced to get the information from the NETCONF server. In the above example we are also using the filtering capabilities to get the information corresponding to the interface eth0 only.

```
<rpc-reply message-id="1"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
    <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
      <interface>
        <name>eth0</name>
        <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-
type">ianaift:ethernetCsmacd</type>
        <enabled>true</enabled>
        <ipv6 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
          <address>
            <ip>2001:db8:c18:1::3</ip>
            <prefix-length>128</prefix-length>
          </address>
        </ipv6>
      </interface>
    </interfaces>
  </data>
</rpc-reply>
```

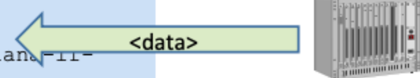

Figure 2.10: RPC reply to the RPC get request [20]

Figure 2.10 shows the RPC reply to the RPC get request. You can notice that the server replies with the same message id as the client originally sent in the rpc request.

2.6.2 Network Management Datastore Architecture (NMDA)

RFC 6241 that provides the specification of NETCONF also introduces the concept of one or more configuration datastores and the various configuration operations that can be performed on each datastore. As defined in RFC 6241:
"A configuration datastore is defined as the complete set of configuration data that is required to get a device from its initial default state into a desired operational state." [17]
As mentioned earlier, one of the major requirements put forward by the network operators during the 2002 IAB Network Management Workshop was the need to separate the operational data and configurational data. NETCONF originally did not have any mechanism to differentiate between the configuration data and operational data as it treated both of them as plain data. Yang made it easier to differentiate between the two types of data by using the `config` flag. Since a yang list has to be either true or false for the data under the list element

to be considered as configurational or operational data, the initial Yang data models had two `list` elements – one for the configuration data that was readable and could be changed as well, the other one for the operational data that could only be read. [14]
One of the examples of this type of data model is the initial version of ietf-interfaces as described in the RFC 7223 [21]:

```
+--rw interfaces
      |  +--rw interface* [name]
      |     +--rw name                        string
      |     +--rw description?                string
      |     +--rw type                        identityref
      |     +--rw enabled?                    boolean
      |     +--rw link-up-down-trap-enable?   enumeration
      +--ro interfaces-state
         +--ro interface* [name]
            +--ro name               string
            +--ro type               identityref
            +--ro admin-status       enumeration
            +--ro oper-status        enumeration
            +--ro last-change?       yang:date-and-time
            +--ro if-index           int32
            +--ro phys-address?      yang:phys-address
            +--ro higher-layer-if*   interface-state-ref
            +--ro lower-layer-if*    interface-state-ref
            +--ro speed?             yang:gauge64
```

This approach was not clear on how to navigate between the operational data and configurational data. For example, looking at the data model described above, it may be easy to say that for an interface "x", the configuration data can be accessed at */interfaces/interface [name= "x"]* and the operation data can be accessed at */interfaces/interface-state [name= "x"]* but what it actually means is that any configuration list *abc* should have an operational list called *abc-state* and that this should be like a naming convention. This approach is not scalable, and you can see it involves duplication of data in both lists. [14]
Thus, another approach which suggested using new datastore to represent the operational data was suggested. The only problem with this approach is making changes to the already developed Yang data models which is currently being done at IETF. In this revised NMDA model, the data objects are defined only once in the Yang data model but different datastores can be used to store the independent instantiations. [14]
The modified form of ietf-interfaces compliant with the new NMDA model is described in RFC 8343 [22] as:

```
+--rw interfaces
   +--rw interface* [name]
      +--rw name                        string
      +--rw description?                string
      +--rw type                        identityref
      +--rw enabled?                    boolean
      +--rw link-up-down-trap-enable?   enumeration {if-mib}?
      +--ro admin-status                enumeration {if-mib}?
      +--ro oper-status                 enumeration
      +--ro last-change?                yang:date-and-time
```

```
      +--ro if-index                          int32 {if-mib}?
      +--ro phys-address?                     yang:phys-address
      +--ro higher-layer-if*                  interface-ref
      +--ro lower-layer-if*                   interface-ref
      +--ro speed?                            yang:gauge64
```

With this new architecture in place, the length of models decreased significantly and there was no need for any kind of naming convention.
The conceptual model of datastores as per the NMDA is as follows:

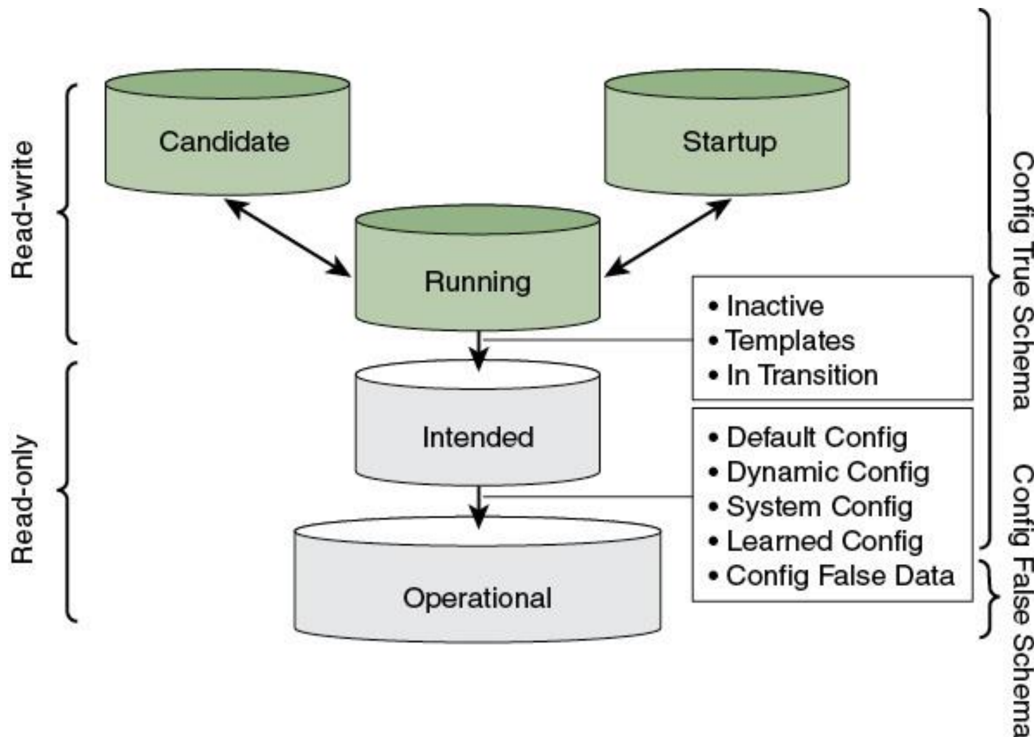

Figure 2.11: Architectural Model of Datastores [14]

As you can see in figure 2.11, the NMDA includes the following datastores:

2.6.2.1 The Startup Configuration Datastore<startup>

<startup> is the configuration datastore that contains the configuration that the device loads when it boots up. This datastore is optional and may not be implemented by all the implementations of protocol. If the device supports non-volatile memory, the contents of this datastore are persistent across reboots. In this case the configuration from <startup> datastore is copied into the <running> datastore when the device boots up. [23]

2.6.2.2 The Candidate Configuration Datastore<candidate>

<candidate> is the configuration datastore that can be altered without changing the device's current configuration. These configurations can later be committed to the <running> datastore. <candidate> datastore is optional and is not supported by most of the implementations of protocol. [23]

2.6.2.3 The Running Configuration Datastore<running>

<running> is the datastore that holds the current valid configuration of the device. <running> datastore is not optional and must be supported by the device. If the device does not have

<startup> datastore implemented and a non-volatile memory is available, then the device will use that non-volatile memory to save the <running> configuration across reboots. [23]

2.6.2.4 The Intended Configuration Datastore<intended>

<intended> datastore holds the configuration that is intended to be used by the device. It is nothing but the <running> configuration after all the configuration transformations have been performed on it. In most implementation, the <intended> is similar to <running> but it can be updated independently as well, provided that the configuration it holds is valid. This datastore is optional and does not have to be persistent across reboots. [23]

2.6.2.5 The Operational State Datastore<operational>

<operational> contains the state of the system. Request to <operational> datastore returns the value in use for a node. This datastore is read only and consists the nodes with config set as true or false. <operational> datastore does not persist across reboots. [23]

2.6.3 NETCONF Protocol Operations

NETCONF supports a set of base operations to manage the configuration of device and to retrieve the operational state data. Some devices support some additional operations depending on the capabilities of the device.
The basic protocol operations are:

2.6.3.1 `get`

The `get` operation is used to get the configuration as well as the operational data from a datastore. Further parameters like `source` to specify the datastore being queried and `filter` to identify the portion of the data being queried can be used. [17]
Figure 2.12 provides an example of RPC with `get` operation and the RPC reply returned by the device. In the example, a `filter` to only retrieve data corresponding to interface with name “eth0” has been supplied.

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
    <filter type="subtree">
      <t:top xmlns:t="http://example.com/schema/1.2/stats">
        <t:interfaces>
          <t:interface t:ifName="eth0"/>
        </t:interfaces>
      </t:top>
    </filter>
  </get>
</rpc>

<rpc-reply message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
    <t:top xmlns:t="http://example.com/schema/1.2/stats">
      <t:interfaces>
        <t:interface t:ifName="eth0">
          <t:ifInOctets>45621</t:ifInOctets>
          <t:ifOutOctets>774344</t:ifOutOctets>
        </t:interface>
      </t:interfaces>
    </t:top>
  </data>
</rpc-reply>
```

Figure 2.12: Example of NETCONF get operation [17]

2.6.3.2 `get-config`

the `get-config` is similar to the `get` operation but is used to only fetch the configuration data from a particular datastore. Like `get`, different parameters like `source` and `filter` can be used to retrieve certain parts of a configuration datastore. [17]

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <running/>
    </source>
    <filter type="subtree">
      <top xmlns="http://example.com/schema/1.2/config">
        <users/>
      </top>
    </filter>
  </get-config>
</rpc>
```

Figure 2.13: get-config operation [17]

Figure 2.13 provides an example of `get-config` operation. Notice how `<source>` tag is used to specify the datastore which should be queried. The `filter` tags are used to get the configuration related to the `users`.

2.6.3.3 edit-config

`edit-config` is one of the most important features of NETCONF because of its transactional nature. `edit-config` changes all or a part of the configuration in a datastore and all this happens in a single transaction. There is no concept of time in a transaction. As a result, there is no need to sequence the commands as you would do while configuring the device manually from a cli. In case if the changes requested are invalid or if the server fails to process these changes, it provides a feature `rollback-on-error`. This feature guarantees that if any of the changes fail then all the other changes that were part of the same transaction are discarded and the server returns to its initial state before the transaction. This mechanism is built into the protocol and as a result does not require implementing code for detecting failure and rollback on the client side. [14]
As part of the edit-config operation you can specify the datastore as target on which you want to perform the operation. The operation attribute is used to set the kind of operation that should be performed on the configuration as part of the edit-config operation. These can be:

- merge: the configuration data sent as part of the `edit-config`, which contains this attribute, is merged with the configuration present at a particular level of the datastore identified by the `<target>` [17].
- replace: the configuration data sent as part of the `edit-config`, which contains this attribute, replaces the configuration present at a certain level in the datastore identified by `<target>`. If the configuration data does not exist, it is created. [17]
- create: if the configuration data which contains this attribute is not present as configuration on the datastore identified by the `<target>`, then and only then will this configuration be added as configuration otherwise an `<rpc-error>` element with the `<error-tag>` value of “data-exists” will be returned. [17]
- delete: if the configuration data which contains this attribute is present as configuration on the datastore identified by the `<target>`, then and only then will this configuration be deleted from the configuration datastore otherwise an `<rpc-error>` element with the `<error-tag>` value of “data-missing” will be returned. [17]
- remove: if the configuration data identified by the element that contains this attribute exists in the configuration datastore identified by the `<target>`, then the configuration data is deleted, otherwise the operation is silently ignored. [17]

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <running/>
    </target>
    <config xmlns:xc="urn:ietf:params:xml:ns:netconf:base:1.0">
      <top xmlns="http://example.com/schema/1.2/config">
        <interface xc:operation="replace">
          <name>Ethernet0/0</name>
          <mtu>1500</mtu>
          <address>
            <name>192.0.2.4</name>
            <prefix-length>24</prefix-length>
          </address>
        </interface>
      </top>
    </config>
  </edit-config>
</rpc>

<rpc-reply message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

Figure 2.14: edit-config operation [17]

Figure 2.14 provides an example of `edit-config` operation which will replace configuration data under `Ethernet0/0` interface with the configuration specified as part of this operation.

2.6.3.4 copy-config

`copy-config` is used to copy or replace an entire configuration datastore. `<target>` and `<source>` tags are sent as attributes as part of the operation to specify the destination and source datastores. [17] Figure 2.15 below gives an example of copy-config operation.

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <copy-config>
    <target>
      <running/>
    </target>
    <source>
      <url>https://user:password@example.com/cfg/new.txt</url>
    </source>
  </copy-config>
</rpc>
```

Figure 2.15: copy-config operation [17]

2.6.3.5 delete-config

`delete-config` is used to delete an entire configuration datastore. The running configuration datastore cannot be deleted. `<target>` tag is used to specify the datastore that has to be deleted. [17] Figure 2.16 provides an example of `delete-config`.

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <delete-config>
    <target>
      <startup/>
    </target>
  </delete-config>
</rpc>
```

Figure 2.16: delete-config operation [17]

2.6.3.6 `lock`

The `lock` operation locks the configuration datastore for a brief period of time so that no other NETCONF client can make the changes to the configuration datastore in that period. An attempt to lock the configuration datastore fails if it is already locked by another client. The `lock` is maintained until it is released or until the session closes. The `<target>` tag is used to specify the datastore which has to be locked. [17] The figure below provides an example to lock the running configuration datastore.

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <lock>
    <target>
      <running/>
    </target>
  </lock>
</rpc>
```

Figure 2.17: lock operation [17]

2.6.3.7 `unlock`

The `unlock` operation is used to release the `lock` on a configuration datastore. The operation fails in case the `lock` is already released or in case the `lock` was acquired in a different session. [17] The figure below provides an example of the `unlock` operation.

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <unlock>
    <target>
     <running/>
    </target>
  </unlock>
</rpc>
```

Figure 2.18: unlock operation [17]

2.6.3.8 `close-session`

`close-session` is used to gracefully close a NETCONF session. When a `close-session` request is received, the server releases any lock or resources associated with the session and closes the session. [17] The figure below provides an example of close-session operation.

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <close-session/>
</rpc>
```

Figure 2.19: close-session operation [17]

2.6.3.9 `kill-session`

`kill-session` is used to forcefully terminate the NETCONF session. When a server receives the `kill-session` request, the server aborts the current operation, releases all the locks and resources associated with the session. If the server receives a `kill-session` while the server is processing a commit, it must restore the configuration to the state before the commit was issued. The session id is specified for the session that has to be killed. [17] figure 2.20 provides an example of kill-session operation performed on session with session-id of 4.

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <kill-session>
    <session-id>4</session-id>
  </kill-session>
</rpc>
```

Figure 2.20: kill-session operation [17]

2.6.3.10 `commit`

The `commit` operation is used to commit the configuration data in the `<candidate>` datastore to the `<running>` datastore. If the device fails to commit all the changes then the `running` configuration datastore is not changed. This operation is only available if the `candidate` datastore is available [17]. Figure 2.21 provides an example of commit operation.

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <commit/>
</rpc>
```

Figure 2.21: commit operation [17]

2.7 RESTCONF

Representational State Transfer Configuration protocol is based on REST. REST is not a protocol but a design pattern. REST is a very popular RPC mechanism based over HTTP and HTTPS. REST is the most popular choice of interaction between multiple web services and can be implemented using any programming language. Due to its popularity, there was a demand among the network automation community for a HTTP interface that follows the principle of REST and is compatible with NETCONF. This led to IETF eventually coming up with RESTCONF in early January 2017 when RFC 8040 specifying the details of RESTCONF was published. [14] RESTCONF follows the following constraints that make up the REST architectural style:

- Client-Server: Implementing the client server architecture makes the system more scalable and makes it easier to evolve the components on either side to evolve independently. [24]
- Statelessness: The communication between the client and server should be stateless in nature. This means that every request from the client must contain all the necessary information that the server needs to respond to the request. [24]
- Cacheability: The data in the response from server can be labelled as cacheable or non-cacheable. If the data is cacheable, the client can reuse the data for a particular period of time after which it received the response. [24]
- Layered System: The client cannot tell if it is communicating directly to the server or through a proxy server. This constraint makes it easier for the applications to scale properly and use techniques like load balancing to distribute the load evenly on the servers. This also provides a way to implement an extra layer of security. [24]
- Uniform Interface: The architecture emphasizes on a uniform interface between different components of the system. It does so by resource identification in requests(each request has a resource id which provides the resource it operates on), resource manipulations through representations(the information needed to create, update or delete the resource are part of the request), self-descriptive messages(each request enough information to describe how to process the message), hypermedia as

the engine of application state (responses from the server should provide the client with the dynamically available resources available to a client through hyperlink). [14] [25] [24]

Just like REST, RESTCONF has standardized requests like GET, POST, PUT, PATCH, DELETE.

RESTCONF is a schema-driven API based on the YANG modules. If the client knows about the yang modules used by the server, it can easily derive all the resource URLS and also the structure of all the RESTCONF requests as well as responses. The Yang modules supported by the server are listed in the module named "ietf-yang-library" [26].

RESTCONF is not a replacement of NETCONF and neither does it replicate the way NETCONF is implemented. In fact, it just provides an HTTP interface to implement the equivalent of NETCONF operations. Thus, basic CRUD operations on the data in different resources are possible with the HTTP GET, POST, PUT, PATCH, DELETE operations. [26]
The figure 2.22 below shows how the RESTCONF operations are related to the NETCONF protocol operations.

RESTCONF	NETCONF
OPTIONS	none
HEAD	<get-config>, <get>
GET	<get-config>, <get>
POST	<edit-config> (nc:operation="create")
POST	invoke an RPC operation
PUT	<copy-config> (PUT on datastore)
PUT	<edit-config> (nc:operation="create/replace")
PATCH	<edit-config> (nc:operation depends on PATCH content)
DELETE	<edit-config> (nc:operation="delete")

Figure 2.22: RESTCONF operations related to NETCONF protocol operations [26]

If you know you have a RESTCONF server running on localhost port 8080, you can enter the following command to find the root URL of this server:

```
curl -i -X GET http://localhost:8080/.well-known/host-meta --header
"Accept: application/xrd+xml" -u username:password
```

```
HTTP/1.1 200 OK
Date: Wed, 17 Feb 2021 16:08:54 GMT
Content-Length: 107
Content-Type: application/xrd+xml
Content-Security-Policy: default-src 'self'; block-all-mixed-content; base-uri 'self'; frame-ancestors 'none';
Strict-Transport-Security: max-age=15552000; includeSubDomains
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block

<XRD xmlns='http://docs.oasis-open.org/ns/xri/xrd-1.0'>
    <Link rel='restconf' href='/restconf'/>
</XRD>
```

Figure 2.23: GET Reply for Well-Known host-meta Information

The server reply to the well known host-meta information can be seen in figure 2.23. The href response [`'/restconf'`] can be concatenated to the server address [`'http://localhost:8080'`] to get the root URL for this server. From the reply, we can say that the root URL for the server is `http://localhost:8080/restconf`.

Running a GET request on the root URL of the RESTCONF server using the following command:

```
curl -i -X GET http://localhost:8080/restconf -u username:password
```

gives the output shown in figure 2.24

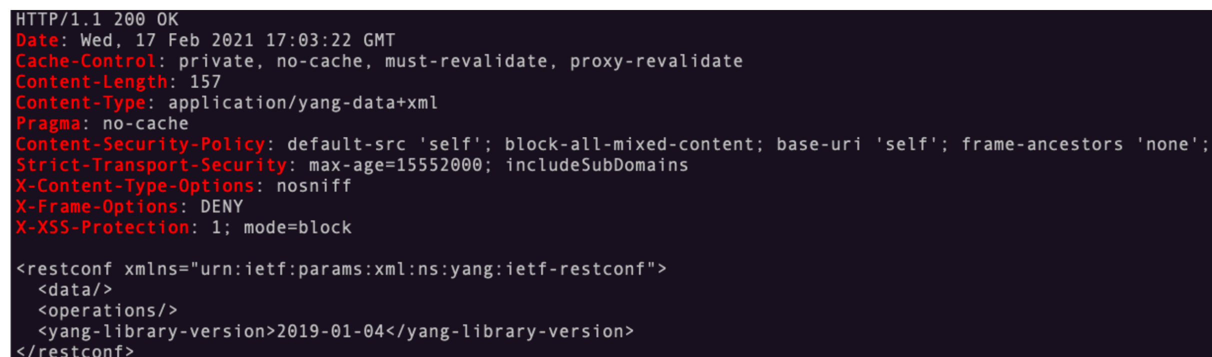

Figure 2.24: GET request on the root URL of RESTCONF server

We can notice that the `/restconf` root resource has the following child resources:

- `restconf/data`: This is a mandatory resource and has to be present in every implementation of RESTCONF. This resource contains both the configuration and state data of the device. [26]
- `restconf/operations`: This is an optional resource, and an implementation of protocol can choose to avoid implementing this feature. This resource provides access to all the RPCs depending on the data models supported by the server. [26]
- `restconf/yang-library-version`: This resource identifies the revision date of the 'ietf-yang-library' module implemented by the server. [26]

2.7.1 RESTCONF vs NETCONF

RESTCONF is guided by the principles of REST. Even though NETCONF follows some of the REST principles like client-server architecture, layered system approach, it differs from REST in some perspectives. The main difference is the stateless server principle. In case of NETCONF, in order to perform a number of `edit-config` operations on server, the client first makes a connection with the server and then performs all the `edit-config` operations. Failure of a single `edit-config` operation means that the server has to roll back to the previous stable configuration. This is not possible to achieve in RESTCONF keeping the stateless server principle of REST in mind as this would mean that all the requests from the client to server should be sent in a single message (which is not possible) and should be acted upon by the server immediately. Implementing more than one changes in the entire network would mean sending multiple requests. Since each request will treated as one transaction this would mean that a network wide transaction is not possible in case of RESTCONF. This means the key feature of network-wide transactions in NETCONF is not possible in case of RESTCONF. *[14]*

Unlike NETCONF which deals with multiple datastores there is no concept of datastore in RESTCONF. In RESTCONF the resource under `restconf/data` behaves like a datastore which imitates the functionality of `<running>` datastore in NETCONF. *[14]*

Also, RESTCONF does not have a concept of `lock`. If a datastore is locked by the NETCONF session, there is no provision in RESTCONF to check if the datastore is locked or to acquire the lock. *[14]*

Keeping the above-mentioned differences in mind, it will be easier to conclude that RESTCONF should be used in cases when the client is managing a single system, but in case when the client is required to manage multiple system, NETCONF is a better option. Thus, RESTONF is the preferred option for web application running on top of the network orchestrator but NETCONF is a better option for communication between the orchestrator and network elements. *[14]*

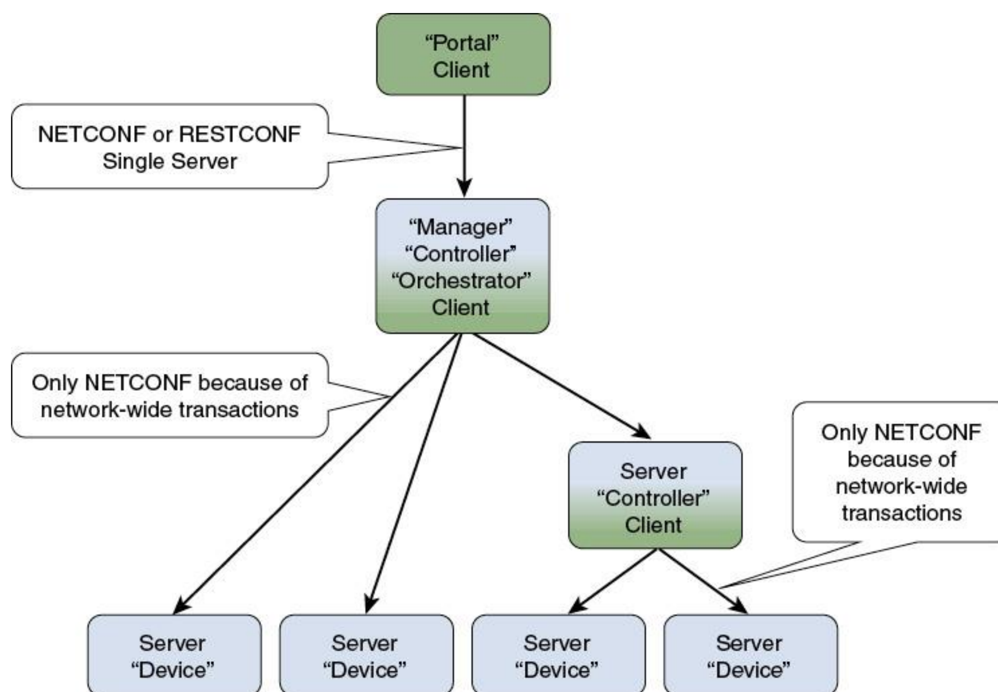

Figure 2.25: Use case for RESTCONF and NETCONF [14]

2.8 Cisco NSO

Cisco NSO or Network Service Orchestrator is a tool developed by Tail-f (now owned by Cisco). It is an orchestration tool to manage hybrid networks. It is designed to deliver high-quality services in an easy and faster way. [27]

NSO uses YANG as a modelling language to manage the devices as well as the services. This makes NSO model driven. NSO also provides a set of Northbound interfaces like CLI, GUI for human interaction and programmable interfaces like RESTCONF, NETCONF and language bindings for Java, Python.

NSO can be used to access and manage the entire network irrespective of the vendors or the type of device (legacy or modern). All the devices and services can be managed from a single CLI. Whenever a user logs in to the NSO CLI, a session is created, and this session provides a local copy of configuration of the entire network. This configuration can be changed by the user which is then validated before committing them to the devices. On making a commit, the changes are copied to the local copy of configuration and the deltas are then pushed out to the network devices. If the changes do not produce any validation errors or network failures, then

the changes are committed to the devices otherwise they are rolled back into the previous stable state. Each commit is a single transaction. A commit either succeeds or fails and thus the changes are either implemented or rolled back. [27]

2.8.1 NSO Architecture

Refer to figure 2.26 to take a look at the architecture of NSO. NSO consists of Service manager, Device Manager, Configuration Database (CDB) and Network Element Drivers (NED).

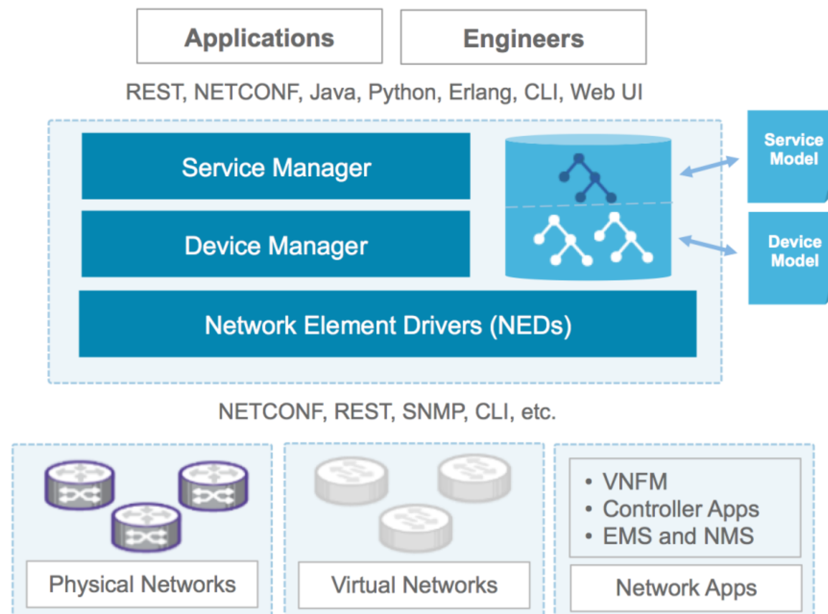

Figure 2.26: Cisco NSO Architecture [28]

Device manager manages the device configuration in a transactional manner. It supports features like fine grained configuration commands, device groups and templates, compliance reporting. [28]
The service manager makes it possible for NSO to support creating and managing high level services in the network. An example of this will be creation and configuring of Vlan across certain devices in the network. Service manager will itself compute the configuration changes for each device and push the changes to them. [28]
While making changes to the devices using NSO, you don't have to worry about the sequence of the commands or the syntax of the commands. This is taken care by the NSO. The life cycle of the service is covered by FASTMAP. While creating a service using NSO, NSO stores the reverse of the device configuration changes as a result of the service alongside the service instance. If someone makes some other changes to the service using NSO, NSO first applies the reverse difference of the previous service creation and then runs the logic to create the service again and finds the difference with the current configuration. This difference is then sent to the devices. [27]
Configuration Database (CDB) is used to store complete network configuration as seen by NSO. All the data stored in the CDB is validated against the YANG model for the devices or services. Any changes done to the configuration of device from anywhere but NSO are considered out of band changes. Doing out of band changes to NSO makes the configuration in NSO CDB to be out of sync with the actual device configuration. Further functionality to synchronize the NSO CDB configuration with the device configuration by either writing NSOs view to the device or reading device configuration into NSO has been implemented. [27]

NSO can communicate to the NETCONF devices southbound provided it has the data models for these devices but for the devices that do not support NETCONF and support only CLI or SNMP, NSO uses Network Element Drivers (NED). NED can render the commands and operations for the devices using this data model. This way if NSO contains the NED for a particular type of device, it should be able to communicate with the device. For NSO to communicate with the different type of OS, respective NED will have to be loaded for the OS. For example, you will need a separate NED for Cisco IOS, Cisco XR and Juniper Junos. [27] [28]

2.8.2 Advantages of using NSO

- NSO is completely model driven thus making it easier to create and manage services as well as devices using data models
- NSO support multi-vendor environment since it is based on data models and standard protocols like NETCONF and RESTCONF
- NSO provides quick mechanism to create service from the scratch to managing to deleting of service.
- Just like NETCONF, all the configuration changes are pushed out NSO as a single transaction. Making it easy to rollback even if a single change is not valid.

3 PROPOSED SOLUTION

3.1 The Components

TINAA platform is based on the principles of **MAPE-K** loop which involves **Monitoring** to collect information from managed devices, **Analyzing** the collected data to see if adaptation is needed, if it is then the steps to meet the target condition are defined in the Plan stage. Finally, the steps defined in the Plan stage are executed in **Execute** stage to reach the desired condition. This loop uses a certain set of adaption parameters which are specified in the **Knowledge** component of MAPE-K loop. [29]

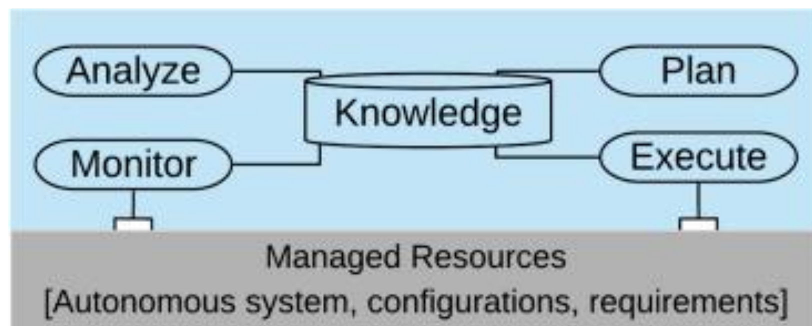

Figure 3.1: MAPE-K loop [29]

The major focus of the project will be to collect the operational data from the legacy network devices which can communicate only using CLI or SNMP. This will fit in the **Monitor** stage of the MAPE-K loop implemented within the TINAA platform. The information collected will be used in both the **analyzing** and **plan** stage. The solution abstracts the southbound protocols used to interact with devices. Even though the project uses SNMP to communicate with the devices southbound, it has the capability to accommodate protocols like NETCONF, RESTCONF for the newer generation devices.

The proposed solution has been used to extract the ARP information from various devices of different vendors over SNMP but can be used to extract other data as well. Please refer to the below diagram Figure 3.2 to understand the proposed solution:

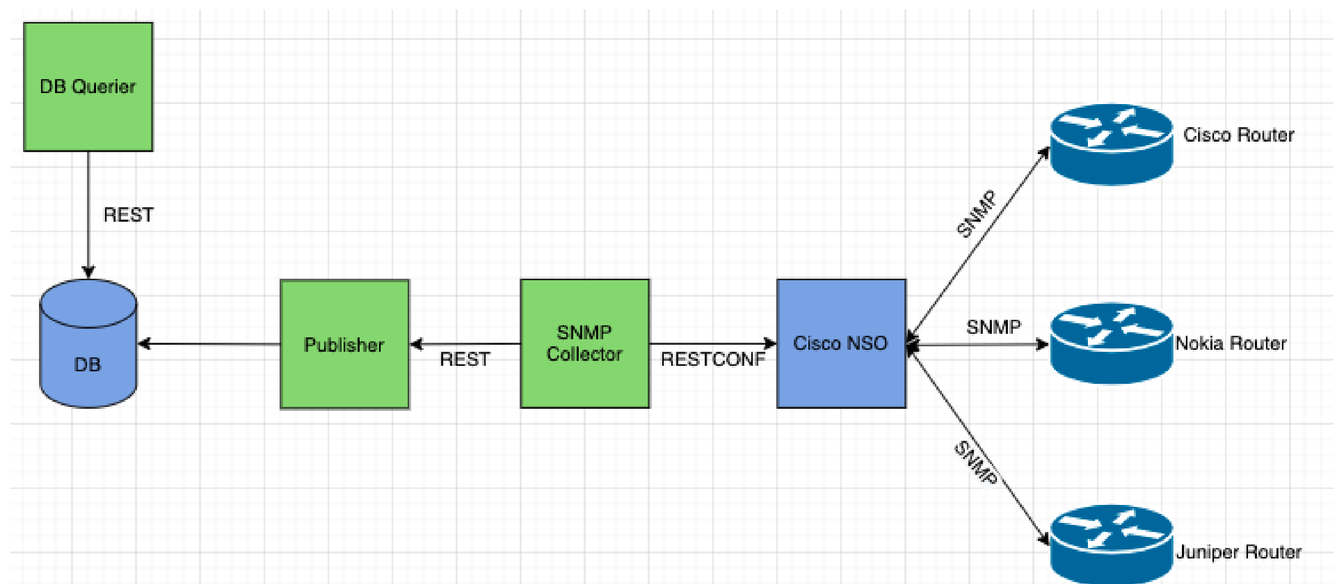

Figure 3.2: Proposed Solution

The solution consists of three major components:

- **SNMP Collector**: The SNMP collector will be responsible for polling data from the devices. SNMP Collector will be communicating with the Cisco NSO using RESTCONF over the YANG data models. Cisco NSO will communicate with the network devices present southbound using Network Element Driver based on SNMP. Since the focus of the project is on legacy devices, we will be using SNMP NEDs. In case if the devices were modern, the task becomes simple as we can communicate with them using NETCONF NED.
- **Publisher**: The publisher collects the data from the SNMP collector over the REST API and pushes the data on to the database.
- **DB Querier**: DB querier will provide an interface to the northbound applications to access the data that has been pushed into the database by publisher. The endpoints of this API are based on the yang data model for ARP.

The SNMP collector can be automated to poll the data from devices after regular intervals of time. The data collected every time the SNMP collector runs could be pushed on to the message queue. The publisher will be listening on the other end of this message queue and as soon as the publisher receives the data on the queue, it will start pushing the data onto the database. Due to the shortage of time, I was not able to implement this mechanism. In the current implementation, Publisher calls the SNMP collector and retrieves the data from the SNMP collector over REST API and then publishes the data to the database.

3.2 Implementation

As part of the implementation, everything from setting up the communication between NSO and network devices to the DB querier will be explained in a step by step manner.

3.2.1 Step 1: Compiling SNMP MIBs and implementing SNMP NEDs in NSO

ARP data is stored in **`ipNetToMediaTable`** SNMP object. The Object ID for **`ipNetToMediaTable`** is 1.3.6.1.2.1.4.22. In order to find the MIB that handles this data, the tool made available by Cisco can be used [30]:
https://snmp.cloudapps.cisco.com/Support/SNMP/do/BrowseOID.do?objectInput=1.3.6.1.2.1.4.22&translate=Translate&submitValue=SUBMIT

Translate OID into object name or object name into OID to receive object details

Enter OID or object name: 1.3.6.1.2.1.4.22 [Translate]

examples -
OID: 1.3.6.1.4.1.9.9.27
Object Name: ifIndex

Object Information

Specific Object Information	
Object	ipNetToMediaTable
OID	1.3.6.1.2.1.4.22
Type	SEQUENCE
Permission	not-accessible
Status	deprecated
MIB	IP-MIB ; - View Supporting Images
Description	"The IPv4 Address Translation table used for mapping from IPv4 addresses to physical addresses. This table has been deprecated, as a new IP version-neutral table has been added. It is loosely replaced by the ipNetToPhysicalTable."

Figure 3.3: MIB corresponding to OID [30]

Since this OID falls under MIB-II, as explained earlier, the table has to be implemented by all the vendors that implement SNMP. Further the highlighted section in figure 3.3 shows IP-MIB as the one responsible for handling **`ipNetToMediaTable`**. IP-MIB is IETF standard MIB and can be either downloaded from the ftp server provided by Cisco at [31]: ftp://ftp.cisco.com/pub/mibs/v2/. The MIB file will be present by the name IP-MIB.my. Download the IP-MIB.my file and rename it to IP-MIB.mib.
On opening the IP-MIB.mib, the import section of the MIB contains the following entries:

```
IMPORTS
    MODULE-IDENTITY, OBJECT-TYPE,
    Integer32, Counter32, IpAddress,
    mib-2, Unsigned32, Counter64,
    zeroDotZero                          FROM SNMPv2-SMI
    PhysAddress, TruthValue,
    TimeStamp, RowPointer,
    TEXTUAL-CONVENTION, TestAndIncr,
    RowStatus, StorageType               FROM SNMPv2-TC
    MODULE-COMPLIANCE, OBJECT-GROUP      FROM SNMPv2-CONF
    InetAddress, InetAddressType,
    InetAddressPrefixLength,
    InetVersion, InetZoneIndex           FROM INET-ADDRESS-MIB
    InterfaceIndex                       FROM IF-MIB;
```

Figure 3.4: Import section of IP-MIB.mib

Figure 3.4 shows the various other MIBs that IP-MIB depends on. We can avoid SNMPv2-SMI, SNMPv2-TC, SNMPv2-CONF since they are related to definitions of SNMPv2 but the other MIBs i.e. IF-MIB and INET-ADDRESS-MIB will have to be downloaded in order to compile the IP-MIB to form the SNMP NED. These mibs can also be downloaded from the Cisco FTP server [31]. Once IF-MIB and INET-ADDRESS-MIB are downloaded, their imports sections can be checked to find the other dependencies which can be downloaded from the same source. In this case there is just one more MIB called IANAifType-MIB that needs to be downloaded.
Move all the MIBs to a directory named MIB

```
[mayank@nsov5-dev-01 ~]$ ls
MIB
[mayank@nsov5-dev-01 ~]$ cd MIB
[mayank@nsov5-dev-01 MIB]$ ls
IANAifType-MIB.mib  IF-MIB.mib  INET-ADDRESS-MIB.mib  IP-MIB.mib
```

Figure 3.5: Contents of MIB directory

Move out of the MIB directory and run the following command to compile the MIBs into SNMP NED package called `acme`:

```
ncs-make-package --snmp-ned ./MIB/ acme
```

Now we `make` the SNMP NED using the following command:

```
cd acme/src; make;
```

```
[mayank@nsov5-dev-01 MIB]$ cd ..
[mayank@nsov5-dev-01 ~]$ ls
MIB
[mayank@nsov5-dev-01 ~]$ ncs-make-package --snmp-ned ./MIB/ acme
[mayank@nsov5-dev-01 ~]$ ls
acme  MIB
[mayank@nsov5-dev-01 ~]$ cd acme/src; make;
```

Figure 3.6: Commands to compile MIBs to SNMP NED

```
BUILD SUCCESSFUL
Total time: 0 seconds
make -C ../netsim  all
make[1]: Entering directory '/home/mayank/acme/netsim'
make INET-ADDRESS-MIB.fxs IP-MIB.fxs IANAifType-MIB.fxs IF-MIB.fxs INET-ADDRESS-MIB.bin IP-MIB.bin IANAifType-MIB.bi
make[2]: Entering directory '/home/mayank/acme/netsim'
env SMIPATH=../src/mibs /opt/ncs/current/netsim/confd/bin/confdc --mib2yang ../src/mibs/INET-ADDRESS-MIB.mib > INET-
ng
a=INET-ADDRESS-MIB-ann.yang; \
/opt/ncs/current/netsim/confd/bin/confdc --yangpath /opt/ncs/current/netsim/confd/src/confd/snmp/yang \
  -c -o INET-ADDRESS-MIB.fxs  `[ -e $a ] && echo "-a $a"` INET-ADDRESS-MIB.yang
env SMIPATH=../src/mibs /opt/ncs/current/netsim/confd/bin/confdc --mib2yang ../src/mibs/IANAifType-MIB.mib > IANAifT
env SMIPATH=../src/mibs /opt/ncs/current/netsim/confd/bin/confdc --mib2yang ../src/mibs/IF-MIB.mib > IF-MIB.yang
env SMIPATH=../src/mibs /opt/ncs/current/netsim/confd/bin/confdc --mib2yang ../src/mibs/IP-MIB.mib > IP-MIB.yang
a=IANAifType-MIB-ann.yang; \
/opt/ncs/current/netsim/confd/bin/confdc --yangpath /opt/ncs/current/netsim/confd/src/confd/snmp/yang \
  -c -o IANAifType-MIB.fxs  `[ -e $a ] && echo "-a $a"` IANAifType-MIB.yang
a=IF-MIB-ann.yang; \
/opt/ncs/current/netsim/confd/bin/confdc --yangpath /opt/ncs/current/netsim/confd/src/confd/snmp/yang \
  -c -o IF-MIB.fxs  `[ -e $a ] && echo "-a $a"` IF-MIB.yang
a=IP-MIB-ann.yang; \
/opt/ncs/current/netsim/confd/bin/confdc --yangpath /opt/ncs/current/netsim/confd/src/confd/snmp/yang \
  -c -o IP-MIB.fxs  `[ -e $a ] && echo "-a $a"` IP-MIB.yang
make[2]: 'IANAifType-MIB.fxs' is up to date.
make[2]: 'IF-MIB.fxs' is up to date.
/opt/ncs/current/netsim/confd/bin/confdc -c ../src/mibs/INET-ADDRESS-MIB.mib  \
INET-ADDRESS-MIB.fxs -f . -f /opt/ncs/current/netsim/confd/etc/confd -f /opt/ncs/current/netsim/confd/etc/confd/snmp
/opt/ncs/current/netsim/confd/bin/confdc -c ../src/mibs/IANAifType-MIB.mib  \
IANAifType-MIB.fxs -f . -f /opt/ncs/current/netsim/confd/etc/confd -f /opt/ncs/current/netsim/confd/etc/confd/snmp
/opt/ncs/current/netsim/confd/bin/confdc -c ../src/mibs/IF-MIB.mib  \
IF-MIB.fxs -f . -f /opt/ncs/current/netsim/confd/etc/confd -f /opt/ncs/current/netsim/confd/etc/confd/snmp
/opt/ncs/current/netsim/confd/bin/confdc -c ../src/mibs/IP-MIB.mib  \
IP-MIB.fxs -f . -f /opt/ncs/current/netsim/confd/etc/confd -f /opt/ncs/current/netsim/confd/etc/confd/snmp
make[2]: 'IANAifType-MIB.bin' is up to date.
make[2]: 'IF-MIB.bin' is up to date.
make[2]: Leaving directory '/home/mayank/acme/netsim'
make[1]: Leaving directory '/home/mayank/acme/netsim'
```

Figure 3.7: Output of make command

Figure 3.7 provides the output of `make` command. Notice how it says that the build was successful. We have successfully compiled the MIBs to form a SNMP NED Package called `acme`.
The next step is to load this SNMP NED package in NSO. In order to do that we will copy the `acme` package to the `/var/opt/ncs/packages` directory by the following command:

```
cp -r acme /var/opt/ncs/packages/
```

```
[mayank@nsov5-dev-01 ~]$ cd /var/opt/ncs/packages/
[mayank@nsov5-dev-01 packages]$ ls
acme                 cisco-ios-cli-6.67     lcdcpe-audit-rf
alu-sr-cli-8.11      cisco-iosxr-cli-7.30   lcdcpe-audit-tracker
```

Figure 3.8: Contents of /var/opt/ncs/packages

We can see that the acme has been placed in the packages directory of NSO.
Login into NSO with the command:

```
ncs_cli -C -u admin
```

and do a `packages reload`

```
[mayank@nsov5-dev-01 packages]$ ncs_cli -C -u admin

admin connected from 172.19.196.9 using ssh on nsov5-dev-01.osp01.srbh.tinaa.tlabs.ca
admin@ncs# packages reload
reload-result {
    package acme-snmp-1.0
    result true
}
reload-result {
    package alu-sr-cli-8.11
    result true
}
reload-result {
    package alu-sr-cli-8.13
    result true
}
reload-result {
    package alusros15r13-nc-1.0
    result true
}
reload-result {
    package auth-store
    result true
}
```

Figure 3.9: Packages Reload in NSO

Figure 3.9 shows the output of packages reload command. Notice that the output mentions the package `acme` as `acme-snmp-1.0`. The addition of snmp to the name is because NSO recognizes the kind of NED it is and 1.0 is the version of package.

3.2.2 Step 2: Adding the devices to be managed to NSO

Since the SNMP NED package is based on IP-MIB which is IETF standard for all the vendors, we can use the same package for Cisco, Juniper or Nokia devices. The next step involves adding the devices to NSO using the SNMP NEDs. Using `acme` package, we will be able to communicate to the devices for any kind of information present in IP-MIB but nothing else.

Before adding the device, we will create a snmp-authgroup by entering the config mode in NSO. Following commands will be used:

```
admin@ncs(config)devices authgroups snmp-group mayank umap
admin community-name ctolab

admin@ncs(config-umap-admin)devices authgroups snmp-group
mayank umap mayank community-name ctolab

admin@ncs(config-umap-mayank)commit
```

```
[mayank@nsov5-dev-01 ~]$ ncs_cli -C -u admin

admin connected from 172.19.196.9 using ssh on nsov5-dev-01.osp01.srbh.tinaa.tlabs.ca
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# devices authgroups snmp-group mayank umap admin community-name ctolab
admin@ncs(config-umap-admin)# devices authgroups snmp-group mayank umap mayank community-name ctolab
admin@ncs(config-umap-mayank)# commit
```

Figure 3.10: Creating snmp-group authgroup

In the above command, you can see that `community-name` has been entered as `ctolab`. This is the community string for this particular device. By default, the `umap` mapping for `admin` has to be created while creating the `snmp-group` before creating the mapping for remote user.

Now we add the device to NSO and map it to the snmp-group we created above by entering the following commands in NSO config mode:

```
admin@ncs(config)# devices device snmp-nokia-2 address
172.25.205.146 port 161 device-type snmp ned-id acme-snmp-1.0
snmp-authgroup mayank version v2c

admin@ncs(config-device-snmp-nokia-2)# state admin-state
unlocked

admin@ncs(config-device-snmp-nokia-2)# commit
```

```
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# devices device snmp-nokia-2 address 172.25.205.146 port 161 device-type snmp ned-id acme-snmp-1.0 snmp-authgrou
p mayank version v2c
admin@ncs(config-device-snmp-nokia-2)# state admin-state unlocked
admin@ncs(config-device-snmp-nokia-2)# commit
Commit complete.
admin@ncs(config-device-snmp-nokia-2)# exit
admin@ncs(config)#
admin@ncs(config)# devices device snmp-nokia-2 connect
result true
info (admin) Connected to snmp-nokia-2 - 172.25.205.146:161
```

Figure 3.11: Adding a device to NSO

The device connection can be checked using the following command:

```
admin@ncs(config)# devices device snmp-nokia-2 connect
```

As shown in figure 3.11, the result in this case is `True` which means the connection was established.

In the same way other devices can be added to NSO. In our case we have named the devices as `snmp-juniper`, `snmp-nokia-3` and `snmp-xr`. As explained earlier, since the acme

package is for IETF standard IP-MIB, thus the same package can also be used to communicate with other devices. There will be a need to create different authgroups depending on if the share the same community strings or not.
The below screenshot provides the connection status of these devices added to NSO

```
admin@ncs(config)# devices device snmp-juniper connect
result true
info (admin) Connected to snmp-juniper - 172.25.130.105:161
admin@ncs(config)# devices device snmp-nokia-3 connect
result true
info (admin) Connected to snmp-nokia-3 - 172.25.205.150:161
admin@ncs(config)# devices device snmp-xr connect
result true
info (admin) Connected to snmp-xr - 172.25.130.152:161
```

Figure 3.12: connection status of devices added to NSO

Let's try to obtain the ARP information from `snmp-nokia-2` device now. For that we will use the following command:

```
admin@ncs# show devices device snmp-nokia-2 live-status IP-MIB ipNetToMediaTable
```

which provides us with the desired information. Please check figure 3.13 for the same.

```
admin@ncs# show devices device snmp-nokia-2 live-status IP-MIB ipNetToMediaTable
IP NET
TO                                           IP NET
MEDIA   IP NET TO                            TO
IF      MEDIA NET        IP NET TO MEDIA     MEDIA
INDEX   ADDRESS          PHYS ADDRESS        TYPE
-------------------------------------------------------
1       172.25.141.41    00:21:05:c2:e8:24   other
2       10.160.161.1     00:00:00:00:00:00   other
3       172.25.159.169   00:21:05:c2:e9:6f   other
4       172.25.159.175   00:21:05:c2:e9:6e   other
5       209.91.124.1     00:21:05:c2:e8:24   other
6       172.25.159.206   00:03:fa:52:7a:b3   dynamic
6       172.25.159.207   00:21:05:c2:e9:67   other
7       192.168.114.1    00:21:05:6c:99:50   other
9       10.160.32.1      00:00:00:00:00:00   other
13      10.160.96.5      00:21:05:c2:e8:24   other
15      192.168.150.1    00:21:05:c2:e8:24   other
17      10.158.0.1       00:21:05:c2:e8:24   other
17      10.158.64.1      00:21:05:c2:e8:24   other
17      10.158.128.1     00:21:05:c2:e8:24   other
17      10.158.192.1     00:21:05:c2:e8:24   other
17      10.197.2.1       00:21:05:c2:e8:24   other
17      10.198.36.1      00:21:05:c2:e8:24   other
17      10.198.37.1      00:21:05:c2:e8:24   other
17      10.198.40.1      00:21:05:c2:e8:24   other
17      10.198.125.1     00:21:05:c2:e8:24   other
17      10.199.36.1      00:21:05:c2:e8:24   other
17      10.199.41.1      00:21:05:c2:e8:24   other
```

Figure 3.13: Performing live status on a device from NSO

When you look at the above output from the command, you might think that this is no different from the output we get from cli but that's not the case. The data from cli is not structured, whereas this is. We just need to see it with a different filter. When we enter the below command:

```
admin@ncs# show devices device snmp-nokia-2 live-status IP-MIB
ipNetToMediaTable | display json
```

we get the following output:

```
admin@ncs# show devices device snmp-nokia-2 live-status IP-MIB ipNetToMediaTable | display json
{
  "data": {
    "tailf-ncs:devices": {
      "device": [
        {
          "name": "snmp-nokia-2",
          "live-status": {
            "IP-MIB:IP-MIB": {
              "ipNetToMediaTable": {
                "ipNetToMediaEntry": [
                  {
                    "ipNetToMediaIfIndex": 1,
                    "ipNetToMediaNetAddress": "172.25.141.41",
                    "ipNetToMediaPhysAddress": "00:21:05:c2:e8:24",
                    "ipNetToMediaType": "other"
                  },
                  {
                    "ipNetToMediaIfIndex": 2,
                    "ipNetToMediaNetAddress": "10.160.161.1",
                    "ipNetToMediaPhysAddress": "00:00:00:00:00:00",
                    "ipNetToMediaType": "other"
                  },
                  {
                    "ipNetToMediaIfIndex": 3,
                    "ipNetToMediaNetAddress": "172.25.159.169",
                    "ipNetToMediaPhysAddress": "00:21:05:c2:e9:6f",
                    "ipNetToMediaType": "other"
                  },
                  {
                    "ipNetToMediaIfIndex": 4,
                    "ipNetToMediaNetAddress": "172.25.159.175",
                    "ipNetToMediaPhysAddress": "00:21:05:c2:e9:6e",
                    "ipNetToMediaType": "other"
```

Figure 3.14: Live status command from NSO on a device with output in JSON format

The same data can also be obtained in xml format by just replacing json with xml in the above command.

3.2.3 Step 3: Adding the devices to a group within NSO

The concept of groups in NSO is to group together the devices which have the same NED capabilities. This makes it easier to manage devices and even provides a unique way to access the group from outside using RESTCONF. The importance of implementing this will be made clear later during the implementation.
We can add the devices to a group by entering the following commands at the NSO configuration mode:

```
admin@ncs(config)# devices device-group snmp device-name [
snmp-juniper snmp-nokia-2 snmp-nokia-3 snmp-xr ]
```

```
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# devices device-group snmp device-name [ snmp-juniper snmp-nokia-2 snmp-nokia-3 snmp-xr ]
admin@ncs(config-device-group-snmp)# commit
Commit complete.
```

Figure 3.15: Adding devices to a group in NSO

3.2.4 Step 4: Implementing SNMP collector

SNMP collector will be querying NSO to collect the information from devices. The information collected in this case will be related to ARP. NSO provides many northbound interfaces but in the project, I will be using the RESTCONF interface to contact the device. Since RESTCONF works similar to REST, we can do a curl on NSO to find out the information.

In the above steps, all the devices that were to be managed using `acme` package were added to a group in NSO named `snmp`. In order to find the members of this group, we can do a curl on NSO with the following command, and we get all the devices that are part of the group.

```
curl -i -u mayank:mayank123
‘http://localhost:8080/restconf/data/tailf-ncs:devices/device-
group=’snmp’/member’ -X GET -H “Content-Type:
application/yang-data+json”
```

Note that in the above GET operation “mayank:mayank123” is the username and password combination used. Since I am accessing NSO from the server where it is hosted, I am using localhost in the URI. This can be replaced by the IP address of the server where NSO is hosted, and the curl will still work fine.

```
[mayank@nsov5-dev-01 ~]$ curl -i -u mayank:mayank123 'http://localhost:8080/restconf/data/tailf-ncs:devices/device-group='snmp'/me
mber' -X GET -H "Content-Type: application/yang-data+json"
HTTP/1.1 200 OK
Date: Mon, 22 Feb 2021 16:21:35 GMT
Last-Modified: Mon, 22 Feb 2021 05:29:19 GMT
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Etag: "1613-971759-606928"
Content-Type: application/yang-data+json
Transfer-Encoding: chunked
Pragma: no-cache
Content-Security-Policy: default-src 'self'; block-all-mixed-content; base-uri 'self'; frame-ancestors 'none';
Strict-Transport-Security: max-age=15552000; includeSubDomains
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block

{
  "tailf-ncs:member": ["snmp-juniper", "snmp-nokia-2", "snmp-nokia-3", "snmp-xr"]
}
```

Figure 3.16: RESTCONF query to the get devices part of snmp group

As you can see in figure 3.16, the above curl provides us with all the devices part of snmp group in the form of a json dictionary. We can test the connection between NSO and any of these devices with the following command:

```
curl -i -u mayank:mayank123
'http://localhost:8080/restconf/data/tailf-
ncs:devices/device="snmp-nokia-2"/connect' -X POST -H
"Content-Type: application/yang-data+json"
```

```
[mayank@nsov5-dev-01 ~]$ curl -i -u mayank:mayank123 'http://localhost:8080/restconf/data/tailf-ncs:devices/device="snmp-nokia-2"/
connect' -X POST -H "Content-Type: application/yang-data+json"
HTTP/1.1 200 OK
Date: Mon, 22 Feb 2021 16:32:41 GMT
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Content-Length: 122
Content-Type: application/yang-data+json
Pragma: no-cache
Content-Security-Policy: default-src 'self'; block-all-mixed-content; base-uri 'self'; frame-ancestors 'none';
Strict-Transport-Security: max-age=15552000; includeSubDomains
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block

{
  "tailf-ncs:output": {
    "result": true,
    "info": "(mayank) Connected to snmp-nokia-2 - 172.25.205.146:161"
  }
}
```

Figure 3.17: Checking connection between NSO and the device

Figure 3.17 shows a POST operation on NSO RESTCONF interface to check the connection between NSO and snmp-nokia-2 device. As a result of operation, we can see True being returned by NSO, implying that the connection is active between the two.

In order to get ARP information from snmp-nokia-2, the following curl command can be used.

```
curl -i -u mayank:mayank123
'http://localhost:8080/restconf/data/tailf-
ncs:devices/device="snmp-nokia-2"/live-status/IP-MIB:IP-
MIB/ipNetToMediaTable/ipNetToMediaEntry' -X GET -H "Content-
Type: application/yang-data+json"
```

```
[mayank@nsov5-dev-01 ~]$ curl -i -u mayank:mayank123 'http://localhost:8080/restconf/data/tailf-ncs:devices/device="snmp-nokia-2"/
live-status/IP-MIB:IP-MIB/ipNetToMediaTable/ipNetToMediaEntry' -X GET -H "Content-Type: application/yang-data+json"
HTTP/1.1 200 OK
Date: Mon, 22 Feb 2021 16:43:15 GMT
Last-Modified: Mon, 22 Feb 2021 05:29:19 GMT
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Etag: "1613-971759-606928"
Content-Type: application/yang-data+json
Transfer-Encoding: chunked
Pragma: no-cache
Content-Security-Policy: default-src 'self'; block-all-mixed-content; base-uri 'self'; frame-ancestors 'none';
Strict-Transport-Security: max-age=15552000; includeSubDomains
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block

{
  "IP-MIB:ipNetToMediaEntry": [
    {
      "ipNetToMediaIfIndex": 1,
      "ipNetToMediaNetAddress": "172.25.141.41",
      "ipNetToMediaPhysAddress": "00:21:05:c2:e8:24",
      "ipNetToMediaType": "other"
    },
    {
      "ipNetToMediaIfIndex": 2,
      "ipNetToMediaNetAddress": "10.160.161.1",
      "ipNetToMediaPhysAddress": "00:00:00:00:00:00",
      "ipNetToMediaType": "other"
    },
    {
      "ipNetToMediaIfIndex": 3,
      "ipNetToMediaNetAddress": "172.25.159.169",
      "ipNetToMediaPhysAddress": "00:21:05:c2:e9:6f",
      "ipNetToMediaType": "other"
    },
```

Figure 3.18: Curl operation to get the ARP information from snmp-nokia-2

Figure 3.18 provides an example of response from NSO when queried for the IP-MIB information for device snmp-nokia-2 through RESTCONF interface. In the examples shown above, it is important to know that the device name used in the URI is local to NSO. Above 3 operations, will be the basis of our SNMP collector implementation in python.
As part of the report, I will be pasting snippets of code for some important functions from repository and explain what the code does.
The first function to explain is `get_devices_from_nso` which is present in file `snmp-poller/backend/app/app/poller.py`. This is a helper function which fetches the devices part of `snmp` group in NSO. Thus this function provides the devices managed by `acme` SNMP NED. Figure 3.19 is the screenshot of code. Refer to the screenshot as I explain the code.
The value of `RESTCONF_BASE, RESTCONF_HEADERS, NSO_USER, NSO_PASSWORD` is set in the config.py file present at `snmp-poller/backend/app/app/core/config.py`
This is what the value is set to:

```
RESTCONF_HEADERS = {"Accept": "application/yang-data+json"}
RESTCONF_BASE = "http://172.25.7.39:8080/restconf/data"
NSO_USER = "mayank"
```

```
NSO_PASSWORD = "mayank123"
```

`RESTCONF_BASE` is the Parent URL of NSO RESTCONF server with 172.25.7.39 being the IP where NSO is hosted. `RESTCONF_HEADERS` just signifies the format of data which in our case is JSON. `NSO_USER` and `NSO_PASSWORD` are the username and password for authentication to NSO.
Using `requests` library from python, a `GET` request is being sent to the `interface_url` which is built with the combination of RESTCONF URL and path to access members of `snmp` group.

```
14  def get_devices_from_nso():
15      """
16      This function returns all the devices added in a group
17      which contains devices that can be controlled using the SNMP NEDs.
18      RESTCONF URL to access members of group :
19      "http://{ip_where_NSO_is_hosted}:8080/restconf/data/tailf-ncs:devices/device-group={group_name}/member"
20      """
21      try:
22          interface_url = settings.RESTCONF_BASE + "/tailf-ncs:devices/device-group=snmp/member"
23          response = requests.get(interface_url, headers = settings.RESTCONF_HEADERS, auth=(settings.NSO_USER,settings.NSO_PASSWORD),
24          # success
25          if response.status_code == 200:
26              json_response = response.json()
27              return json_response["tailf-ncs:member"]
28
29          logger.debug(repr(response.status_code))
30          logger.debug(repr(response.content))
31
32          # error
33          json_response = response.json()
34          raise HTTPException(
35              status_code=response.status_code, detail=json_response["error"],
36          )
37      except Exception as e:
38          logger.exception(e)
39          if type(e) != HTTPException:
40              raise HTTPException(status_code=500, detail="Internal server error")
41          else:
42              raise e
```

Figure 3.19: Definition of get_devices_from_nso()

If `response` received from the `GET` operation is success with a response `status_code` of 200, we can access "`tailf-ncs:member`" element of json response and return it to the function calling `get_devices_from_nso`, otherwise we raise an exception.

```
37  @celery_app.task(acks_late=True)
38  def get_arp(device_name):
39      """
40      This function generates the ARP table entries for a device which has been
41      added to NSO using SNMP NED.
42      RESTCONF URL to get the ARP entries:
43      "http://{ip_where_NSO_is_hosted}:8080/restconf/data/tailf-ncs:devices/device={device_name}/live-status/IP-MIB:IP-MIB/ipNetToMediaTable/ipNetToMediaEntry"
44      """
45
46      logger.info(f"Entering ARP ")
47      try:
48          interface_url = settings.RESTCONF_BASE + f"/tailf-ncs:devices/device={device_name}/live-status/IP-MIB:IP-MIB/ipNetToMediaTable/ipNetToMediaEntry"
49          response = requests.get(interface_url, headers = settings.RESTCONF_HEADERS, auth=(settings.NSO_USER,settings.NSO_PASSWORD), verify=False)
50
51          if response.status_code == 200:
52              result = response.json()
53              logger.info(result)
54              return result['IP-MIB:ipNetToMediaEntry']
55
56
57          logger.debug(repr(response.status_code))
58          logger.debug(repr(response.content))
59
60          # error
61          json_response = response.json()
62          raise HTTPException(
63              status_code=response.status_code, detail=json_response["error"],
64          )
65
66      except Exception as e:
67          logger.exception(e)
68          if type(e) != HTTPException:
69              raise HTTPException(status_code=500, detail="Internal server error")
70          else:
71              raise e
```

Figure 3.20: get_arp(device_name) function

Figure 3.20 shows the function `get_arp` present in the file `snmp-poller/backend/app/app/worker.py`. This function is used to fetch the ARP information for a particular device by sending a GET request to `interface_url` using python `requests` library. The function takes the name of device local to NSO as an argument `device_name`. Line number 37 is a wrapper which makes every call to this function to be executed as a different process using the concept of celery workers. Celery is a distributed task queue implemented in python. Celery workers make it possible to fetch the information from more than one device at a time. Thus, making the snmp collector to be scalable. If the response from the request is a success with a `status_code` of 200, this function returns the `'IP-MIB:ipNetToMediaEntry'` element from the Json response. This element contains the ARP entries retrieved from a particular device. If the response is not a success, the function raises an exception.

```
14  @router.get("/devices")
15  def get_all_devices():
16      """
17      This endpoint is used to get the ARP information for all the devices
18      added to NSO using SNMP-NEDs
19      """
20      if settings.AUTO_DISCOVER_DEVICES:
21          try:
22              devices = get_devices_from_nso()
23          except Exception as e:
24              logger.exception(e)
25              if type(e) != HTTPException:
26                  raise HTTPException(status_code=500, detail="Internal server error")
27              else:
28                  raise e
29      else:
30          pass #Function to get the device info from another API should be implemented here
31      result_dict = {}
32      device_data_list = []
33      for device in devices:
34          device_data = {}
35          try:
36              result = celery_app.send_task("app.worker.get_arp_test", args=[device])
37              device_data['device_name'] = device
38              device_data['arp_table'] = result.get()
39              logger.info(f"request to get the arp info for {device} received")
40              device_data_list.append(device_data)
41          except Exception as e:
42              logger.exception(e)
43              if type(e) != HTTPException:
44                  raise HTTPException(status_code=500, detail="Internal server error")
45              else:
46                  raise e
47      result_dict['data'] = device_data_list
48      result_dict['data_type'] = 'Arp'
49      logger.info(result_dict)
50      return result_dict
```

Figure 3.21: REST Endpoint for snmp-collector

Figure 3.21 shows the implementation of `get_all_devices` function present in `/snmp-poller/backend/app/app/api/api_v1/endpoints/poller.py`. The wrapper `router.get('/devices')` turns this function into an API endpoint using

FastAPI library from python. FastAPI is one of the quickest ways to implement RESTful APIs. The endpoint returns the ARP information from all the devices with the help of functions `get_devices_from_nso` and `get_arp`. The if condition in line number 20 checks if the Autodiscover is True or False. If the Autodiscover is True, then the function gets the devices from NSO otherwise it fetches the devices from another API. Once we get the list of devices loaded into NSO. We can do a `get_arp` on each device to get the ARP information from each device. The function returns the ARP information from each device as a dictionary.

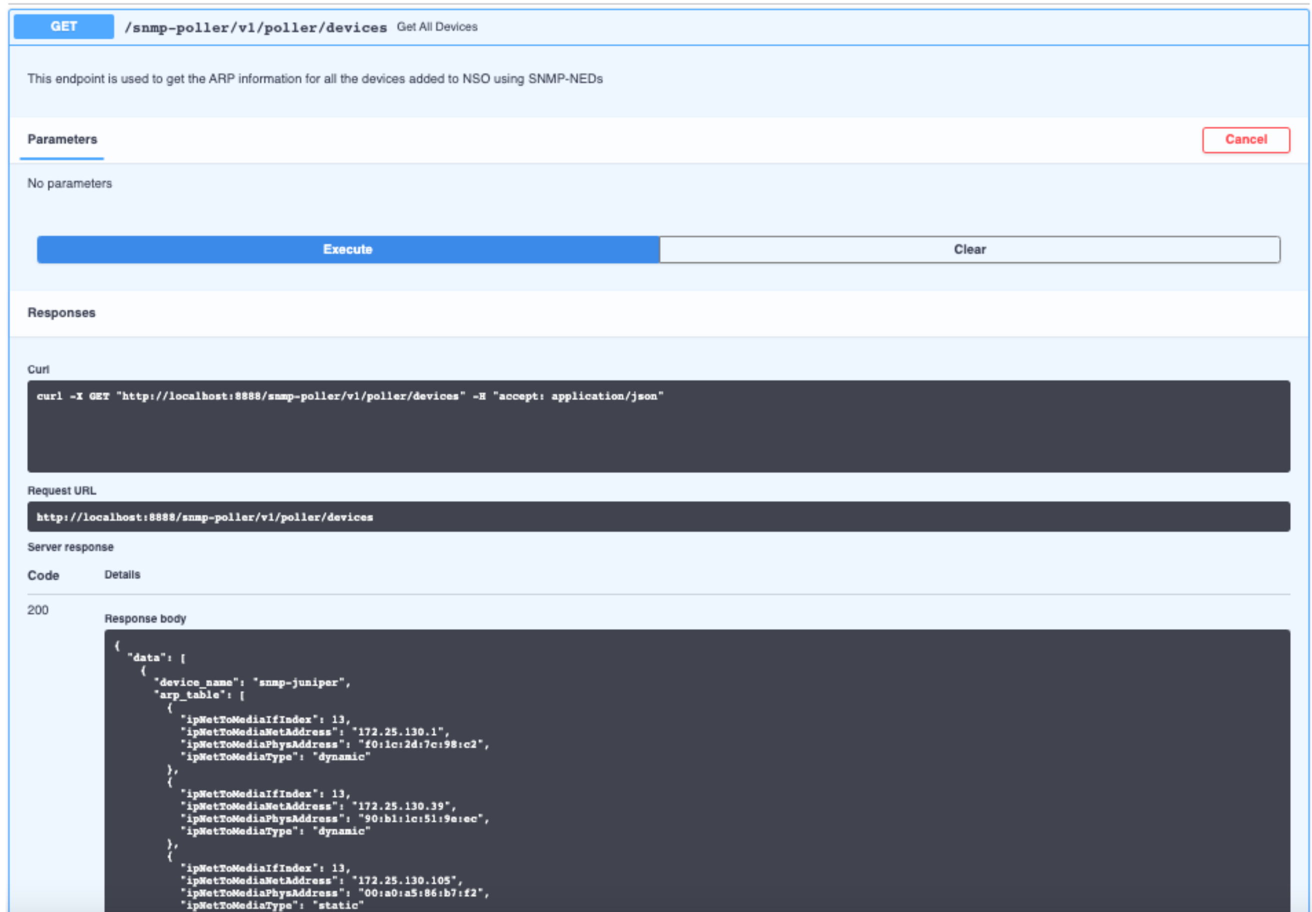

Figure 3.22: Get operation on /devices

The above figure provides the output of GET operation done one /devices endpoint from a swagger page associated with FastAPI app.

Let’s consider we get the data from 2 devices (snmp-juniper, snmp-xr) the return dictionary will look something like this:

```
{
   'data': [
               {
               'device_name': 'snmp-juniper'
               'arp_table': [
                               {
                                "ipNetToMediaIfIndex": 13,
                                "ipNetToMediaNetAddress": "172.25.130.1",
                                "ipNetToMediaPhysAddress":"f0:1c:2d:7c:98:c2",
                                "ipNetToMediaType": "dynamic"
                               },
                               {
                               "ipNetToMediaIfIndex": 13,
                               "ipNetToMediaNetAddress": "172.25.130.39",
```

```
                         "ipNetToMediaPhysAddress": "90:b1:1c:51:9e:ec",
                         "ipNetToMediaType": "dynamic"
                         }
                         ]
            },
            {
            'device_name': 'snmp-xr'
            'arp_table': [
                         {
                          "ipNetToMediaIfIndex": 13,
                          "ipNetToMediaNetAddress": "172.25.130.1",
                          "ipNetToMediaPhysAddress":"f0:1c:2d:7c:98:c2",
                          "ipNetToMediaType": "dynamic"
                         },
                         {
                         "ipNetToMediaIfIndex": 13,
                         "ipNetToMediaNetAddress": "172.25.130.39",
                         "ipNetToMediaPhysAddress": "90:b1:1c:51:9e:ec",
                         "ipNetToMediaType": "dynamic"
                         }
                         ]
            }
      ]
    'data_type': 'Arp'
}
```


3.2.5 Step 5: Implementing the publisher

The function of the publisher is to collect all the data that snmp-collector polls and publish it to a PostgreSQL database. In order to do so, models are created for different tables using SQLAlchemy Object Relational Mapper for python.

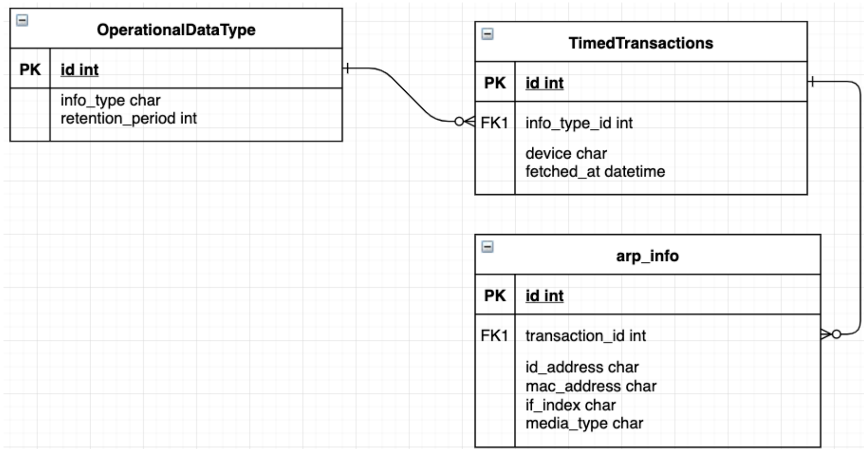

Figure 3.23: Relationship diagram of database

Figure 3.23 provides the relationship between different tables that can be used to hold information for ARP.
`OperationalDataType` holds records for different types of operational data like ARP, route table etc. that the database can hold. Figure 3.24 shows the `OperationalDataType` model present at `snmp-poller/backend/app/app/models/operational_data_type.py`

`TimedTransactions` holds the record about when a particular type of `OperationalDataType` was fetched from a particular device. Figure 3.25 shows the `TimedTransactions` model present at `snmp-poller/backend/app/app/models/timed_transactions.py`

```python
from app.db.base_class import Base
from sqlalchemy import Column, String, Integer, Date, Table, ForeignKey
from sqlalchemy.orm import relationship, backref

class OperationalDataType(Base):
    """
        Contains records of the types of operational data
        we want to poll e.g. route table, ARP table etc.
    """
    __tablename__ = 'OperationalDataType'

    id = Column(Integer, primary_key=True)
    info_type = Column(String)
    retention_period = Column(Integer)

    def __init__(self, info_type, retention_period):
        self.info_type = info_type
        self.retention_period = retention_period
```

Figure 3.24: SQLAlchemy Model for OperationalDataType

```python
import datetime
from sqlalchemy import Column, String, Integer, Date, DateTime, ForeignKey
from sqlalchemy.orm import relationship, backref
from app.db.base_class import Base

class TimedTransactions(Base):
    """
        Each TimedTransaction contains the record of the device that was polled
        and when it was polled. It also contains the record of what kind of operational
        data was polled.
    """
    __tablename__ = 'TimedTransactions'

    id = Column(Integer, primary_key=True)
    device = Column(String)
    fetched_at = Column(DateTime, default=datetime.datetime.utcnow)
    info_type_id = Column(Integer, ForeignKey('OperationalDataType.id'))
    info_type = relationship("OperationalDataType", backref=backref("operational_data_type"))

    def __init__(self, device,timestamp,data_type):
        self.device = device
        self.fetched_at = timestamp
        self.info_type = data_type
```

Figure 3.25: SQLAlchemy model for TimedTransactions

Table `arp_info` holds the arp records associated with each `TimedTransaction`. It contains all the arp entries retrieved from the device in a single `TimedTransaction`. Figure 3.26 contains the `Arp` model which can be located at `snmp-poller/backend/app/app/models/arp.py`

```python
from sqlalchemy import Column, String, Integer, Boolean, ForeignKey
from sqlalchemy.orm import relationship, backref
from app.db.base_class import Base


class Arp(Base):
    """
        This is the model class for ARP entries.
        Each object of this class is an ARP entry.
    """
    __tablename__ = 'arp_info'

    id = Column(Integer, primary_key=True)
    ip_address = Column(String)
    mac_address = Column(String)
    if_index = Column(String)
    media_type = Column(String)
    transaction_id = Column(Integer, ForeignKey('TimedTransactions.id'))
    transaction_type = relationship("TimedTransactions", backref=backref("transation_type"))

    def __init__(self, ip_address, mac_address, if_index, media_type, transaction_type):
        self.ip_address = ip_address
        self.mac_address = mac_address
        self.if_index = if_index
        self.media_type = media_type
        self.transaction_type = transaction_type
```

Figure 3.26: SQLAlchemy model for Arp

For implementing the Publisher and DB Querier there is a need to implement certain CRUD functions for models. The CRUD functions for `OperationalDataType` are implemented in the file `snmp-poller/backend/app/app/crud/crud_operational_data_type.py`

```python
from app.models.operational_data_type import OperationalDataType

def create_operational_data_type(db, info, retention_period):
    """
        This function creates the OperationalDataType record and
        returns it.
    """
    operational_data = OperationalDataType(info, retention_period)
    db.add(operational_data)
    return operational_data

def read_operational_data_type(db, info_type):
    """
        This function returns the OperationalDatatype record with the
        same information type as that of info_type. It returns the same record.
    """
    query = db.query(OperationalDataType).filter(OperationalDataType.info_type == info_type)
    return query.first()
```

Figure 3.27: crud_operational_data_type.py

Figure 3.27 shows the CRUD functions for `OperationalDataType`. Here is what they do:

- `create_operational_data_type(db, info, retention_period)`: This function creates a record of `OperationalDataType`. Here `db` is the database session object, `info` string used to set the value of info_type and

`retention_period` is the integer to set the retention period of a particular `OperationalDataType`.

- `read_operational_data_type(db, info_type)`: This function takes in the database session object `db` and string `info_type` as argument. The string `info_type` is used to filter the record from `OperationalDataType` table and return the `OperationalDataType` object back to the calling function.

```
def create_timed_transaction(db, device_name, timestamp, info):
    """
        This function creates a record of TimedTransaction by taking in
        the name of device as string, Timestamp as datetime object and info
        as the OperationalDataType object.
    """
    transaction = TimedTransactions(device_name, timestamp, info)
    db.add(transaction)
    return transaction

def read_timed_transaction(db, device_name):
    """
        This function returns the most recent TimedTransaction
        record for a particular device.
    """
    query = db.query(TimedTransactions).filter(TimedTransactions.device==device_name).order_by(TimedTransactions.fetched_at.desc())
    return query.first()

def get_last_timestamp_transaction(db, info):
    """
        This function returns the TimedTransaction with the most
        recent timestamp.
    """
    query = db.query(TimedTransactions).filter(TimedTransactions.info_type_id==info.id).order_by(TimedTransactions.fetched_at.desc()
    return query.first()

def all_transactions_for_timestamp(db, timestamp):
    """
        This function returns all the TimedTransaction records
        for a particular timestamp.
    """
    query = db.query(TimedTransactions).filter(TimedTransactions.fetched_at==timestamp).order_by(TimedTransactions.fetched_at.desc()
    return query.all()
```

Figure 3.28: crud_timed_transactions.py

Figure 3.28 shows the CRUD functions for TimedTransactions implemented in file `snmp-poller/backend/app/app/crud/crud_operational_data_type.py`. Here is what they do:

- `create_timed_transaction(db, device_name, timestamp, info)`: This function takes in the database `session` object as `db`, name of device as string `device_name`, time at which the transaction was run as `datetime` object `timestamp` ,operational data that was retrieved in the transaction as `OperationalDataType` object `info` and creates a `TimedTransaction` record.
- `read_timed_transaction(db, device_name)`: this function takes in the database session object `db` and the name of device in string `device_name`. It then uses `device_name` to filter the most recent `TimedTransaction` having the same `device_name` and returns the `TimedTransaction`.
- `get_last_timestamp_transaction(db, info)`: this function takes in the database session object `db` and the type of operational data as `OperationalDataType` object `info`. It filters to give the most recent `TimedTransaction` for this particular `info` and returns it.
- all_transactions_for_timestamp(db, timestamp): this function takes in the database session object `db` and the `datetime` object `timestamp`. It returns all the `TimedTransaction` having the `fetched`_at value same as that of `timestamp`.

```python
def create_arp_entry(db, device_data, device_transaction):
    """
        this function adds a record in the Arp table. It takes in a
        dictionary device_data that has the data corresponding to a single ARP entry
        and the TimedTransaction record with which the Arp entry has to be mapped.
    """
    entry = Arp(device_data['ipNetToMediaNetAddress'], device_data['ipNetToMediaPhysAddress'],
                device_data['ipNetToMediaIfIndex'], device_data['ipNetToMediaType'], device_transaction)
    db.add(entry)


def create_multiple_arp_entry(db, device_data, device_transaction):
    """
        This function adds a record in the Arp table. It takes in a
        dictionary device_data that has a list of ARP entries and the TimedTransaction
        record with which these Arp entries have to be mapped.
    """
    for single_entry in device_data:
        create_arp_entry(db, single_entry, device_transaction)

def get_all_arp_entries_for_transaction(db, transaction):
    """
        Get all the Arp records for a particular TimedTransaction.
    """
    query = db.query(Arp).filter(Arp.transaction_id==transaction.id)
    return query.all()

def filter_for_ip_and_index(db, transaction, ip, index):
    """
       Get all the Arp records for a particular TimedTransaction
       having the same IP address.
    """
    query = db.query(Arp).filter(Arp.transaction_id==transaction.id).filter(Arp.ip_address==ip).filter(Arp.if_index==index)
    return query.all()
```

Figure 3.29: crud_arp.py

Figure 3.29 shows the CRUD functions for Arp implemented in file `snmp-poller/backend/app/app/crud/crud_arp`. Here is what they do:

- `crud_arp_entry(db, device_data, device_transaction)`: this function takes in database session object `db`, a single arp data entry as `device_data` dictionary, the `TimedTransaction` object which was created while fetching the arp entry from network device as `device_transaction` and creates an object or record of `Arp` model class in the `arp_info` table.
- `create_multiple_arp_entry(db, device_data, device_transaction)` : this function takes in database session object `db`, a list with multiple arp data entries as `device_data`, the `TimedTransaction` object which was created while fetching the arp entries from network device as `device_transaction` and creates multiple objects or records of `Arp` model class in the `arp_info` table.
- `get_all_arp_entries_for_transaction(db, transaction)` : this function takes in database session object as `db`, `TimedTransaction` object as `transaction` and returns all the entries in `arp_info` table associated with that `transaction`.
- `filter_for_ip_and_index(db, transaction, ip, index)` : this function takes in database session object as `db`, `TimedTransaction` object as `transaction`, ip address as string `ip` , if-index as string `index` and returns all the entries in `arp_info` table associated with that transaction having the same value of `ip_address` and `if_index` as `ip` and `index` respectively.

After implementing the various CRUD functions for different model classes, we move on to discuss the publisher which is present in snmp-poller/backend/app/app/publisher.py.

```python
30  if __name__ == "__main__":
31      """
32          This functions polls the SNMP poller.
33          The data received from snmp poller is
34          published into the database.
35      """
36      r = requests.get('http://localhost/snmp-poller/v1/poller/devices')
37      logger.info(r.status_code)
38      if r.status_code == 200:
39          result = r.json()
40      else:
41          logger.info("No Response from SNMP poller")
42          sys.exit(0)
43      try:
44          session = SessionLocal()
45          info_type = find_operational_data_type(session, result['data_type'])
46          timestamp = datetime.datetime.utcnow()
47          for i in result['data']:
48              transaction = crud_timed_transactions.create_timed_transaction(session,
49                                                                i['device_name'], timestamp, info_type)
50              crud_arp.create_multiple_arp_entry(session, i['arp_table'], transaction)
51          session.commit()
52      except Exception as e:
53          session.rollback()
54          logger.exception(e)
55      else:
56          logger.info("Data published to the database")
57      finally:
58          session.close()
59          logger.info("Session closed")
```

Figure 3.30: publisher.py

Figure 3.30 shows the `main` function in publisher.py. Line 36 is used to fetch all the arp entries from the snmp-collector by sending a GET request to the `/devices` endpoint. If the request is not successful, the publisher exits. If the request is successful, the json response is saved in a variable called `result` and a database session is created by calling on the `SessionLocal`. Another function called `find_operational_data_type` present in the same file is used to find if there exists an `OperationalDataType` record with the `info_type` value same as that of `result['data_type']`. If it does not exist, the function creates the new `OperationalDataType` record with the value as that of `result['data_type']` and puts it in the variable `info_type`. In line 47 we create a `datetime` object `timestamp` which will be used to create the `TimedTransactions`. Then for every device in `result['data']` element, we create the `TimedTransaction` using `create_timed_transaction` and further use this `TimedTransaction` to create multiple arp entries for each `TimedTransaction` using `create_multiple_arp_entry` function. If there is any exception, the transaction is rolled back and finally the database session is closed.
Running publisher.py from the docker image where the code is hosted results in data being pushed into the database.

The figures below show the data sitting in OperationalDataType, TimedTransactions and arp_info tables in a postgres database.

snmp-poller/postgres@db

Query Editor | Query History

```
1 select * from public."OperationalDataType";
2
3
```

Data Output | Explain | Messages | Notifications

	id [PK] integer	info_type character varying	retention_period integer
1	1	Arp	10

Figure 3.31: OperationalDataType table

snmp-poller/postgres@db

Query Editor | Query History

```
1 select * from public."TimedTransactions";
2
3
```

Data Output | Explain | Messages | Notifications

	id [PK] integer	device character varying	fetched_at timestamp without time zone	info_type_id integer
1	1	snmp-juniper	2021-02-23 01:11:54.89295	1
2	2	snmp-nokia-2	2021-02-23 01:11:54.89295	1
3	3	snmp-nokia-3	2021-02-23 01:11:54.89295	1
4	4	snmp-xr	2021-02-23 01:11:54.89295	1

Figure 3.32: TimedTransactions table

snmp-poller/postgres@db

Query Editor   Query History

```
1  select * from public.arp_info;
2
```

Data Output   Explain   Messages   Notifications

	id [PK] integer	ip_address character varying	mac_address character varying	if_index character varying	media_type character varying	transaction_id integer
127	127	192.168.178.1	40:a6:77:6f:bb:42	123895	static	1
128	128	192.168.178.1	40:a6:77:6f:bb:42	123896	static	1
129	129	10.192.64.1	40:a6:77:6f:ba:ec	124381	static	1
130	130	192.168.139.1	40:a6:77:6f:bb:42	124397	static	1
131	131	192.178.177.1	40:a6:77:6f:bb:42	124400	static	1
132	132	192.168.128.1	40:a6:77:6f:bb:42	124401	static	1
133	133	192.168.189.1	40:a6:77:6f:bb:42	124414	static	1
134	134	12.1.1.1	40:a6:77:6f:b9:04	124416	static	1
135	135	172.25.130.1	f0:1c:2d:7c:98:c2	13	dynamic	2
136	136	172.25.130.39	90:b1:1c:51:9e:ec	13	dynamic	2
137	137	172.25.130.105	00:a0:a5:86:b7:f2	13	static	2
138	138	172.25.130.136	00:03:fa:42:38:ce	13	dynamic	2
139	139	10.0.0.4	02:00:00:00:00:04	18	static	2
140	140	10.0.0.5	02:01:01:00:00:05	18	dynamic	2

Figure 3.33: arp_info table

3.2.6 Step 6: Implementing DB Querier

Before implementing the DB querier, we have to come up with a generic data model that would support fetching the ARP information as the `acme` package compiled in the first step does.

Below is the data model developed as part of the project to support the ARP functionality:

```
module arp{
  namespace "http://telus.com/yang/arp";
  prefix arp;

import ietf-inet-types {
    prefix inet;
  }

import ietf-yang-types {
    prefix yang;
 }
list devices{
        config false;
        key "device";
        leaf device{
         description
         "Device Name";
         type string;
        }
container ipNetToMediaTable {
        description
```

```
        "The IPv4 Address Translation table used for mapping
          from IPv4 addresses to physical addresses";
        list ipNetToMediaEntry {
          key "ipNetToMediaIfIndex ipNetToMediaNetAddress";
          description
          "Each entry contains one IpAddress to `physical'
               address equivalence.";
          leaf ipNetToMediaIfIndex {
            type int32 {
              range "1..2147483647";
            }
            description
              "The interface on which this entry's equivalence
               is effective.";
          }
          leaf ipNetToMediaPhysAddress {
            type yang:phys-address {
              length "0..65535";
            }
            description
              "The media-dependent `physical' address.";
          }
          leaf ipNetToMediaNetAddress {
            type inet:ipv4-address;
            description
            "The IpAddress corresponding to the media-
               dependent physical' address.";
          }
          leaf ipNetToMediaType {
            type enumeration {
              enum "other" {
                value 1;
              }
              enum "invalid" {
                value 2;
              }
              enum "dynamic" {
                value 3;
              }
              enum "static" {
                value 4;
              }
            }
           description
            "The type of mapping.";
          }
        }
      }
  }
}
```

The name of module is `arp` and it should be saved in a file called `arp.yang` for this module to work.

The following dockerfile will build a container which will provide us with the swagger page with the various endpoints that can be supported.

```
ARG MODEL_FILE_NAME=arp.yang
FROM mbj4668/yanger as builder
ARG MODEL_FILE_NAME
COPY ./ /workdir
RUN yanger -t expand \
   -f swagger \
   --swagger-tag-mode resources  \
   --swagger-top-resource data \
   /workdir/${MODEL_FILE_NAME} -o /workdir/swagger.json
FROM node:14 as converter
ARG MODEL_FILE_NAME
# COPY --from=builder /workdir /workdir
COPY --from=builder /workdir/swagger.json /workdir/swagger.json
RUN npm install -g swagger2openapi
RUN swagger2openapi /workdir/swagger.json -o /workdir/oas3.json
FROM swaggerapi/swagger-ui:latest
ARG MODEL_FILE_NAME
COPY --from=converter /workdir/swagger.json /workdir/swagger.json
COPY --from=converter /workdir/oas3.json /workdir/oas3.json
ENV SWAGGER_JSON "/workdir/oas3.json"
```

Place both the dockerfile and arp.yang in the same folder to build the container using the following command:

```
$ docker build -t mayank-yanger .
```

And then you can run the container by using the following command:

```
$ docker container run -d -p 8080:8080 --name swagger mayank-
yanger:latest
```

The swagger endpoints for the above data model can be seen in figure 3.34.

GET /data/arp:devices

GET /data/arp:devices={devices-device}

GET /data/arp:devices={devices-device}/device Device Name

GET /data/arp:devices={devices-device}/ipNetToMediaTable The IPv4 Address Translation table used for mapping from IPv4 addresses to physical addresses

GET /data/arp:devices={devices-device}/ipNetToMediaTable/ipNetToMediaEntry Each entry contains one IpAddress to `physical' address equivalence.

GET /data/arp:devices={devices-device}/ipNetToMediaTable/ipNetToMediaEntry={ipNetToMediaEntry-ipNetToMediaIfIndex},{ipNetToMediaEntry-ipNetToMediaNetAddress} Each entry contains one IpAddress to `physical' address equivalence.

GET /data/arp:devices={devices-device}/ipNetToMediaTable/ipNetToMediaEntry={ipNetToMediaEntry-ipNetToMediaIfIndex},{ipNetToMediaEntry-ipNetToMediaNetAddress}/ipNetToMediaIfIndex The interface on which this entry's equivalence is effective.

GET /data/arp:devices={devices-device}/ipNetToMediaTable/ipNetToMediaEntry={ipNetToMediaEntry-ipNetToMediaIfIndex},{ipNetToMediaEntry-ipNetToMediaNetAddress}/ipNetToMediaPhysAddress The media-dependent `physical' address.

GET /data/arp:devices={devices-device}/ipNetToMediaTable/ipNetToMediaEntry={ipNetToMediaEntry-ipNetToMediaIfIndex},{ipNetToMediaEntry-ipNetToMediaNetAddress}/ipNetToMediaNetAddress The IpAddress corresponding to the media-dependent `physical' address.

GET /data/arp:devices={devices-device}/ipNetToMediaTable/ipNetToMediaEntry={ipNetToMediaEntry-ipNetToMediaIfIndex},{ipNetToMediaEntry-ipNetToMediaNetAddress}/ipNetToMediaType The type of mapping.

Figure 3.34: Swagger endpoints for arp.yang

From all the endpoints in figure 3.34, we will be implementing the one squared in red.

```
26  @router.get("/data/arp:devices")
27  def get_arp_all_devices():
28      """
29          This function returns the Arp records for all the devices
30          using the most recent data published by publisher.
31      """
32      try:
33          db = SessionLocal()
34          info = crud_operational_data_type.read_operational_data_type(db, 'Arp')
35          last_timestamp = crud_timed_transactions.get_last_timestamp_transaction(db, info)
36          transactions = crud_timed_transactions.all_transactions_for_timestamp(db, last_timestamp.fetched_at)
37          return_dict = {}
38          for trns in transactions:
39              entries = crud_arp.get_all_arp_entries_for_transaction(db, trns)
40              return_dict[trns.device] = parse_data_for_each_transaction(entries)
41          db.close()
42      except Exception as e:
43          logger.exception(e)
44          if type(e) != HTTPException:
45              raise HTTPException(status_code=500, detail="Internal server error")
46          else:
47              raise e
48      else:
49          logger.info("Returned the recent ARP entries for all devices")
50          return return_dict
51      finally:
52          db.close()
```

Figure 3.35: function get_arp_all_devices()

Figure 3.35 shows the function `get_arp_all_devices()` implemented in the `snmp-poller/backend/app/app/api/api_v1/endpoints/querier.py` file. The `router.get` is a wrapper function which turns this function to a REST endpoint. This function returns the ARP information from all the devices using the most recent `TimedTransaction`. All this is done using the CRUD functions explained in step 5 of implementation.

If we look at figure 3.36, we get to know that the recent `TimedTransaction` for devices `snmp-juniper and snmp-nokia-2` has id of 5 and 6. Figure 3.37 provides the result of `GET` operation on "`http://localhost:8888/snmp-`

`poller/v1/querier/data/arp:devices`" and we can see the ARP entries with the transaction id of 5 and 6. Due to the screen size, the rest of the devices could not fit into the screenshot.

snmp-poller/postgres@db

Query Editor | Query History

```
1  select * from public."TimedTransactions"
```

Data Output | Explain | Messages | Notifications

	id [PK] integer	device character varying	fetched_at timestamp without time zone	info_type_id integer
1	1	snmp-juniper	2021-02-23 01:11:54.89295	1
2	2	snmp-nokia-2	2021-02-23 01:11:54.89295	1
3	3	snmp-nokia-3	2021-02-23 01:11:54.89295	1
4	4	snmp-xr	2021-02-23 01:11:54.89295	1
5	5	snmp-juniper	2021-02-23 05:07:22.975657	1
6	6	snmp-nokia-2	2021-02-23 05:07:22.975657	1
7	7	snmp-nokia-3	2021-02-23 05:07:22.975657	1
8	8	snmp-xr	2021-02-23 05:07:22.975657	1

Figure 3.36: all the records of TimedTransactions

Curl

```
curl -X GET "http://localhost:8888/snmp-poller/v1/querier/data/arp:devices" -H "accept: application/json"
```

Request URL

```
http://localhost:8888/snmp-poller/v1/querier/data/arp:devices
```

Server response

Code	Details
200	Response body


```
        "Type": "static",
        "Transaction": 5
      },
      {
        "IP": "12.1.1.1",
        "MAC": "40:a6:77:6f:b9:04",
        "Index": "124416",
        "Type": "static",
        "Transaction": 5
      }
    ],
    "snmp-nokia-2": [
      {
        "IP": "172.25.130.1",
        "MAC": "f0:1c:2d:7c:98:c2",
        "Index": "13",
        "Type": "dynamic",
        "Transaction": 6
      },
      {
```


Figure 3.37: GET operation on /devices endpoint

The next function `get_arp_single_device(device)` is used to implement the "`/data/arp:devices={device}/ipNetToMediaTable/ipNetToMediaEntry`" endpoint. This function is also implemented in the same `querier.py` file. This function returns the ARP entries for a particular device associated with the most recent `TimedTransaction`. The function is implemented with the help of CRUD functions defined in step 5.

```python
54  @router.get("/data/arp:devices={device}/ipNetToMediaTable/ipNetToMediaEntry")
55  def get_arp_single_device(device):
56      """
57          This function returns the Arp records for a particular
58          device using the most recent data published by publisher.
59      """
60      try:
61          db = SessionLocal()
62          info = crud_operational_data_type.read_operational_data_type(db, 'Arp')
63          trns = crud_timed_transactions.read_timed_transaction(db, device)
64          entry = crud_arp.get_all_arp_entries_for_transaction(db, trns)
65          return_dict = {}
66          return_dict[trns.device] = parse_data_for_each_transaction(entry)
67          db.close()
68      except Exception as e:
69          logger.exception(e)
70          if type(e) != HTTPException:
71              raise HTTPException(status_code=500, detail="Internal server error")
72          else:
73              raise e
74      else:
75          logger.info("Returned the recent ARP entries for a particular device")
76          return return_dict
77      finally:
78          db.close()
```

Figure 3.38: Function get_arp_single_device(device)

Curl

```
curl -X GET "http://localhost:8888/snmp-poller/v1/querier/data/arp:devices=snmp-xr/ipNetToMediaTable/ipNetToMediaEntry" -H "accept: application/json"
```

Request URL

```
http://localhost:8888/snmp-poller/v1/querier/data/arp:devices=snmp-xr/ipNetToMediaTable/ipNetToMediaEntry
```

Server response

Code	Details
200	Response body


```
{
  "snmp-xr": [
    {
      "IP": "172.25.130.1",
      "MAC": "f0:1c:2d:7c:98:c2",
      "Index": "13",
      "Type": "dynamic",
      "Transaction": 8
    },
    {
      "IP": "172.25.130.39",
      "MAC": "90:b1:1c:51:9e:ec",
      "Index": "13",
      "Type": "dynamic",
      "Transaction": 8
    },
    {
      "IP": "172.25.130.105",
      "MAC": "00:a0:a5:86:b7:f2",
      "Index": "13",
      "Type": "static",
      "Transaction": 8
    },
    {
      "IP": "172.25.130.136",
      "MAC": "00:03:fa:42:38:ce",
      "Index": "13",
      "Type": "dynamic",
      "Transaction": 8
```


Figure 3.39: Curl on arp:devices=snmp-xr/ipNetToMediaTable/ipNetToMediaEntry endpoint

As shown in figure 3.39, the most recent `TimedTransaction` for device `snmp-xr` has id 8. The results of curl operation done on the "`arp:devices=snmp-xr/ipNetToMediaTable/ipNetToMediaEntry`" show the ARP entries from `arp_info` with the `transaction_id` 8.

```python
80  @router.get("/data/arp:devices={device}/ipNetToMediaTable/ipNetToMediaEntry={ip},{index}")
81  def get_arp_filter_for_ip_and_index(device, ip, index):
82      """
83          This function returns the Arp records with a particular ip
84          using the most recent data published by publisher for a single device.
85      """
86      try:
87          db = SessionLocal()
88          info = crud_operational_data_type.read_operational_data_type(db, 'Arp')
89          trns = crud_timed_transactions.read_timed_transaction(db, device)
90          entry = crud_arp.filter_for_ip_and_index(db, trns, ip, index)
91          return_dict = {}
92          return_dict[trns.device] = parse_data_for_each_transaction(entry)
93          db.close()
94      except Exception as e:
95          logger.exception(e)
96          if type(e) != HTTPException:
97              raise HTTPException(status_code=500, detail="Internal server error")
98          else:
99              raise e
100     else:
101         logger.info("Returned the recent ARP entries for a particular device with a filter for a particular IP")
102         return return_dict
103     finally:
104         db.close()
```

Figure 3.40: Function get_arp_filter_for_ip_and_index()

The function get_arp_filter_for_ip_and_index() returns the ARP entry associated with the most recent TimedTransaction containing a particular ip and if-index. The function is defined in the same querier.py file. The function utilizes the CRUD functions defined in step 5 earlier. In figure 3.40, the third entry has an IP address of 172.25.130.105 and index value as 13. Doing a GET operation on the endpoint /data/arp:devices={device}/ipNetToMediaTable /ipNetToMediaEntry={ip},{index} with the value of device, ip and index set to snmp-xr, 172.25.130.105 and 13 returns the same entry with transaction id 8. Check the screenshot in figure 3.41 for the same.

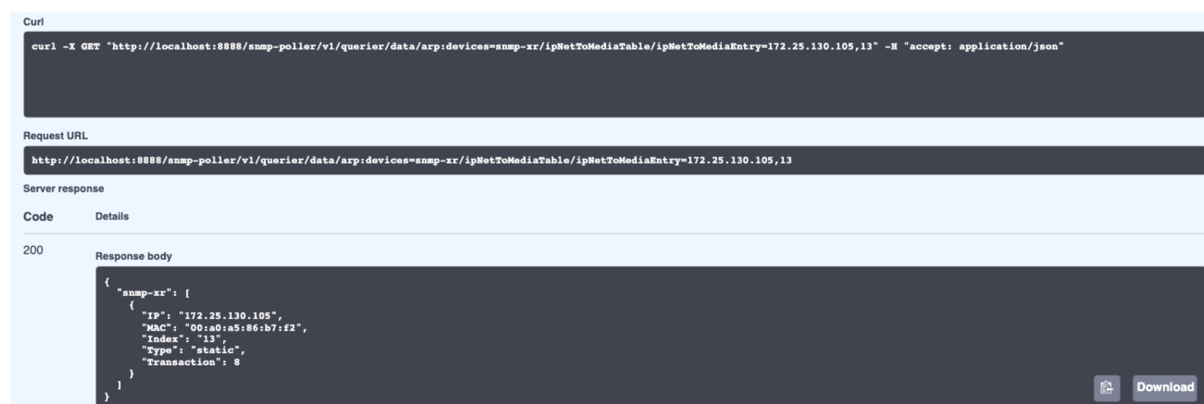

Figure 3.41: GET on third endpoint

4 LIMITATIONS

Cisco NSO can prove out to be a limitation to the scalability of this approach. Even though there are various replies from Tail-f system in Cisco community saying that NSO is capable of handling 10,000 devices on a single server, this approach does not test those limits.

While working with NSO I noticed that sometimes NSO was slow in handling the slow legacy devices. The commands executed from NSO can take up to 2 minutes to get around 1500 records of necessary information from slow old devices.

While working with the RESTCONF interface of NSO, there were slight delays in response sometimes. Even though RESTCONF is fast, but since it is very new it has still got to get better.

5 Bibliography

[1] H. Song, T. Zhou, Z. Li, Z. Li, P. Martinez-Julia, L. Ciavaglia and A. Wang, “Network Telemetry Framework,” 14 December 2018. [Online]. Available: https://tools.ietf.org/id/draft-song-opsawg-ntf-02.html#rfc.section.2.1.

[2] W. Odom, “Introduction to Controller-Based Networking,” 12 Feb 2020. [Online]. Available: https://www.ciscopress.com/articles/article.asp?p=2995354&seqNum=2. [Accessed Feb 2021].

[3] Open Networking Foundation, “Software-Defined Networking: The New Norm for Networks,” 13 April 2012. [Online]. Available: https://opennetworking.org/sdn-resources/whitepapers/software-defined-networking-the-new-norm-for-networks/. [Accessed Februaury 2021].

[4] Wipro Technologies, “Software Defined Networking,” [Online]. Available: https://www.wipro.com/infrastructure/sdn-adoption-in-enterprises/.

[5] S. Sezer, S. Scott-Hayward, P. K. Chouhan, D. Lake, B. Fraser, N. Viljoen, J. Finnegan, N. Rao and M. Miller, “Are We Ready for SDN? Implementation Challenges for Software-Defined Networks,” July 2013. [Online]. Available: https://cs.uwaterloo.ca/~brecht/courses/854-Emerging-2014/readings/sdn/sdn-intro-ieee-comm-2013.pdf. [Accessed February 2021].

[6] SDxCentral Studios, “What Is Software Defined Networking (SDN)? Definition,” 25 August 2016. [Online]. Available: https://www.sdxcentral.com/networking/sdn/definitions/what-the-definition-of-software-defined-networking-sdn/. [Accessed February 2021].

[7] Ciena, “What is SDN?,” [Online]. Available: https://www.ciena.com/insights/what-is/What-Is-SDN.html.

[8] D. Teare, “Structuring and Modularizing the Network with Cisco Enterprise Architecture,” 12 June 2008. [Online]. Available: https://www.ciscopress.com/articles/article.asp?p=1073230&seqNum=4. [Accessed February 2021].

[9] D. Mauro and K. Schmidt, Essential SNMP, 2nd Edition, O'Reilly Media, Inc., 2005.

[10] S. Cadora, “The limits of SNMP,” 10 June 2016. [Online]. Available: https://blogs.cisco.com/sp/the-limits-of-snmp. [Accessed February 2021].

[11] P. Shafer, “An Architecture for Network Management Using NETCONF and YANG,” June 2011. [Online]. Available: https://tools.ietf.org/html/rfc6244#section-4.3.3. [Accessed February 2021].

[12] Cisco, “Deep Dive into Model Driven Programmability with NETCONF and YANG,” 2018. [Online]. Available: https://pubhub.devnetcloud.com/media/netdevops-live/site/files/s01t03.pdf. [Accessed February 2021].

[13] A. Pras and J. Schoenwaelder, “On the Difference between Information Models and Data Models,” January 2003. [Online]. Available: https://tools.ietf.org/html/rfc3444. [Accessed February 2021].

[14] J. Clarke, B. Claise and J. Lindblad, Network Programmability with YANG: The Structure of Network Automation with YANG, NETCONF, RESTCONF, and gNMI, First Edition, Addison-Wesley Professional, 2019.

[15] M. Bjorklund, “The YANG 1.1 Data Modeling Language,” August 2016. [Online]. Available: https://tools.ietf.org/html/rfc7950. [Accessed February 2021].

[16] J. Schoenwaelder, "RFC 6021 : Common YANG Data Types," October 2010. [Online]. Available: https://tools.ietf.org/html/rfc6021. [Accessed February 2021].

[17] R. Enns, M. Bjorklund, J. Schoenwaelder and A. Bierman, "Network Configuration Protocol (NETCONF)," June 2011. [Online]. Available: https://tools.ietf.org/html/rfc6241#page-35. [Accessed February 2021].

[18] Tail-f, "NETCONF OVERVIEW," [Online]. Available: https://www.tail-f.com/what-is-netconf/. [Accessed February 2021].

[19] Cisco Systems, "Breaking down NETCONF communication," [Online]. Available: https://developer.cisco.com/learning/lab/intro-netconf/step/2. [Accessed February 2021].

[20] C. Moberg and C. Justus, "NETCONF by Example," October 2015. [Online]. Available: https://trac.ietf.org/trac/edu/raw-attachment/wiki/IETF94/94-module-3-netconf.pdf. [Accessed February 2021].

[21] M. Bjorklund, "A YANG Data Model for Interface Management," May 2014. [Online]. Available: https://tools.ietf.org/html/rfc7223. [Accessed February 2021].

[22] M. Bjorklund, "A YANG Data Model for Interface Management," March 2018. [Online]. Available: https://tools.ietf.org/html/rfc8343. [Accessed February 2021].

[23] M. Bjorklund, J. Schoenwaelder, P. Shafer, K. Watsen and R. Wilton, "Network Management Datastore Architecture (NMDA)," March 2018. [Online]. Available: https://tools.ietf.org/html/rfc8342. [Accessed February 2021].

[24] R. T. Fielding, "Representational State Transfer (REST)," [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. [Accessed February 2021].

[25] Wikipedia, "Representational state transfer," [Online]. Available: https://en.wikipedia.org/wiki/Representational_state_transfer#Layered_system. [Accessed February 2021].

[26] A. Bierman, M. Bjorklund and K. Watsen, "RESTCONF Protocol," January 2017. [Online]. Available: https://tools.ietf.org/html/rfc8040. [Accessed 2021 February].

[27] Cisco, "NSO Fundamentals," [Online]. Available: https://developer.cisco.com/docs/nso/#!nso-fundamentals/nso-fundamentals. [Accessed February 2021].

[28] Cisco, "NSO Guide," [Online]. Available: https://developer.cisco.com/docs/nso/guides/. [Accessed February 2021].

[29] S. Jahan, I. Riley, C. Walter, R. F. Gamble, M. Pasco, P. K. Mckinley and B. H. Cheng, "MAPE-K/MAPE-SAC: An interaction framework for adaptive systems with security assurance cases," August 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19320527. [Accessed February 2021].

[30] Cisco, "SNMP Object Navigator," [Online]. Available: https://snmp.cloudapps.cisco.com/Support/SNMP/do/BrowseMIB.do?local=en&step=2&mibName=IP-MIB. [Accessed February 2021].

[31] Cisco, "Index of /pub/mibs/v2/," [Online]. Available: ftp://ftp.cisco.com/pub/mibs/v2/. [Accessed February 2021].

[32] J. D. Case, M. Fedor, M. L. Schoffstall and J. R. Davin, "Simple Network Management Protocol," May 1990. [Online]. Available: https://tools.ietf.org/html/rfc1157. [Accessed February 2021].