Department of Computer Science
Computer Networks Research Group

Project Report

Management of ServiCes Across MultipLE clouds

Authors:

Arkajit Dhar
Ashwin Prasad Shivarpatna Venkatesh
Bhargavi Mohan
Deeksha Mysore Ramesh
Sanket Kumar Gupta
Suheel Shrirangapura Nazeersab
Vivek Jaganath

Supervisors:

Prof. Dr. Holger Karl | Sevil Dräxler | Hadi Razzaghi Kouchaksaraei

Paderborn, September 25, 2019

Contents

TODOs

List of Figures

List of Tables

1 Motivation

The rapid growth of mobile data services driven by mobile internet has led to substantial challenges of high availability, low latency, high bit rate and performances in networks. The recent development of Network Function Virtualization (NFV) and Software-Defined Networks (SDN) have emerged as key enablers for 5G networks. There has been a paradigm shift in networking with the recent developments in network virtualization technology. NFV involves decoupling of the hardware components from the software components of a network function. NFV requires a central management and orchestrations (MANO) framework in order to fully deliver end-to-end services of an application. There are multiple open source as well as industrial implementations of MANO framework in the market.

End-to-end network service delivery requires chaining of the Virtual Network Functions (VNFs) across different Internet Service Providers (ISPs) which in turn have their own MANO frameworks. In order to seamlessly create a network service by utilizing the VNF within each of the MANO frameworks, the need of interoperability among different MANO frameworks is of utmost importance.

1.1 Problem Description

European Telecommunications Standards Institute (ETSI) defines the reference architecture for a MANO framework. Each network service is composed of multiple network functions virtually linked and orchestrated by a MANO framework. The network services require a descriptor which contains the details of all the VNFs, virtual links and forwarding graph of VNFs. Each of the VNFs contains it's own Virtual Network Function Descriptor (VNFD) and the information about the number of virtual machines it requires. Different MANO frameworks have their own descriptor schemata pertaining to the standard defined by ETSI. This framework-specific Network Service Descriptor (NSD) hinders the orchestration and management of VNFs between different MANO frameworks.

Firstly, the project aims at tackling the above mentioned problem with the implementation of translator and splitter engines, which would help translate the NSD and divide the VNFs to be deployed on different MANO frameworks, thus creating a framework-independent network service chain. Secondly, the project aims at the implementation of a MANO adaptor, that allows interaction between different MANO frameworks, exposes the network service instantiation interfaces of the underlying MANO frameworks and retrieves monitoring information about the network service status. The adaptor will mainly address MANO scalability challenges and perform state management. Lastly, the project aims at integrating these individual modules in order to provide an end-to-end network service delivery across different MANO frameworks.

2 Technologies

2.1 MANO Frameworks

In this section, a few MANO frameworks and their NSD schemata are listed(list of a few parameters and their description). Among several frameworks, the plan is to select a few and set them up locally, deploy network services and support them in the project. This list is subjected to future additions or removals.

2.1.1 Pishahang

Pishahang is a framework consolidated from state-of-the-art NFV and Cloud management and orchestration tools and technologies. Pishahang is a framework for jointly managing and orchestrating virtual network functions and cloud-based microservices.

Pishahang deploys, manages and orchestrates complex services by consolidating and extending current Cloud and NFV tools and technologies [KDK18].

Network Service Descriptor Section

Parameter	**Description**
Vendor	Identifies the network service uniquely across all network service vendors.
Name	Name of the network service without its version.
Version	Names the version of the NSD.
Author	It's an optional parameter. It describes the author of NSD
Description	It's an optional parameter, provides an arbitrary description of the network service.

Table 2.1: Pishahang: Network Service Descriptor Section

Network Functions Section

Parameter	Description
network_functions	Contains all the VNFs that are handled by the network service.
Vnf_id	Represents a unique identifier within the scope of the NSD
Vnf_vendor	As part of the primary key, the vendor parameter identifies the VNF Descriptor
Vnf_name	As part of the primary key, the name parameter identifies the VNF Descriptor
Vnf_version	As part of the primary key, the version parameter identifies the VNF Descriptor
Vnf_description	It's an optional parameter, a human-readable description of the VNF

Table 2.2: Pishahang: Network Functions Section

2.1.2 Open Source MANO(OSM)

OSM is an open source management and orchestration stack in compliant with ETSI NFV information models. OSM architecture splits between resource orchestrators and service orchestrators [dSPR+18]. Table 2.3 lists the parameters of NSD schema. [Doc]

Parameter	Description
id	Identifier for the NSD
name	NSD name
Short-name	Short name which appears on the UI
vendor	Vendor of the NSD
logo	File path for the vendor specific logo. For example, icons/mylogo.png. The logo should be a part of the network service.
description	Description of the NSD
version	Version of the NSD

Table 2.3: OSM: Network Service Descriptor

2.2 Virtualized Infrastructure Manager (VIM)

VIM is one of the three functional blocks specified in the Network Functions Virtualization Management and Orchestration (NFV-MANO) architecture. VIM is responsible for controlling and managing the NFV Infrastructure (NFVI), by provisioning and optimizing the allocation of physical resources to the virtual resources in the NFVI. Performance and error monitoring is also a key role of the VIM. Popular VIMs are discussed in the following sections.

2.2.1 OpenStack

OpenStack[1] is a community-driven open source cloud resource management platform. Compute, storage, and networking resources in a data center can be provisioned using Application Program Interfaces (APIs) or web dashboard provided by OpenStack component, for instance, NOVA is a component which can provide access to compute resources, such as virtual machines and containers. Network management is enabled by NEUTRON component which handles the creation and management of a virtual networking infrastructure like switches and routers. SWIFT component provides a storage system. By making use of many such components, OpenStack can deliver complex services by utilizing an underlying pool of resources.

2.2.2 Kubernetes

Kubernetes[2] (K8s) is an open-source platform for automation and management of containerized services, it manages computing, networking, and storage infrastructure. Kubernetes was initially developed by Google and now under Cloud Native Computing Foundation. Kubernetes Architecture consists of (1) Master server components – it is the control plane of the cluster and act as the gateway for administrators (2) Node Server Components – servers which are performing work by using containers that are called nodes, they communicate with the master component for instructions to run the workload assigned to them. Kubernetes provides comprehensive APIs which are used to communicate between the components and with the external user.

[1] https://www.openstack.org/
[2] https://kubernetes.io/

3

Scramble architecture

Scramble is a tool to realize scalability with hierarchical orchestration. While the higher level orchestrator (Parent MANO) still controls the life-cycle of the deployed network services, scramble acts as a bridge between parent MANO and lower level orchestrator (child MANO) thus allowing interconnection between different MANO's to provide end-to-end service.

3.1 Architecture

The services of scramble resides as a plugin within MANO frameworks such as SOTANA/Pishahang and Open Source MANO (OSM) and enables cross MANO communication. Service requests received by the higher level MANOs in the hierarchy can be routed to lower level MANOs using scramble plugin based on monitoring parameters.

Each child MANO's in-turn contains scramble plugin which allows multiple other MANO's to be added as a child hence achieving hierarchical orchestration (See Figure **??**).

Scramble is composed of three main services.

- Translator
- Splitter
- Wrapper(adaptor)

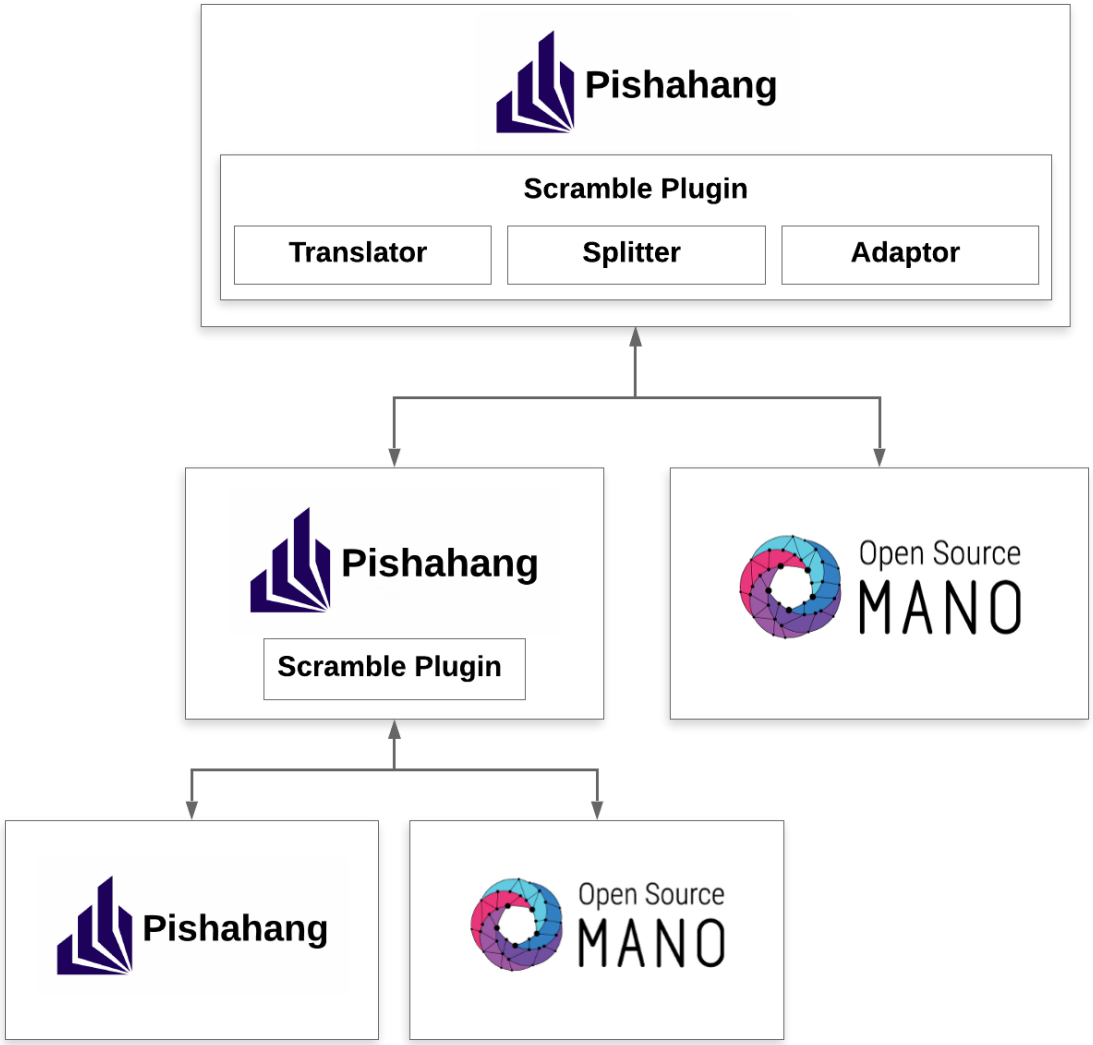

Figure 3.1: Scramble Architecture

4 Work Packages

4.1 Translator

In a hierarchical architecture involving different MANOs, there is a need of conversion of network descriptors to schemas of respective MANO. Service Descriptor Translator (SDT) serves the purpose of translating network descriptors, namely NSDs and VNFDs from schema of SONATA Pishahang to that of OSM and vice versa.

In a scenario, where a parent MANO, say Pishahang decides to deploy one of the network services in its lower hierarchy MANO, say OSM, the NSD and VNFD(s) need to be converted to the descriptor schema of OSM. In such an event, the Scramble plugin calls the translator service and sends the descriptors to the SDT, where the translation of the descriptors takes place and the translated descriptors are sent to Adaptor utility for deployment in appropriate MANO. Figure 4.1 gives a high level view of Translator.

Scramble plug-in, installed within the parent MANO, forwards the network descriptors with service request to Scramble Main-Engine. Main-Engine checks the service request and sends the network descriptors to Translator along with the information of destination schema. On receiving the network descriptors, SDT translates the same to requested schema and calls the validator function to validate the translated network descriptors. Once validation is complete, the translated descriptors will be sent to Adaptor for deployment.

4.1.1 Architecture & Work flow

The translator engine reads a json formatted descriptor and first converts it into a pandas *DataFrame*. The *DataFrame* is constructed to have the following columns.

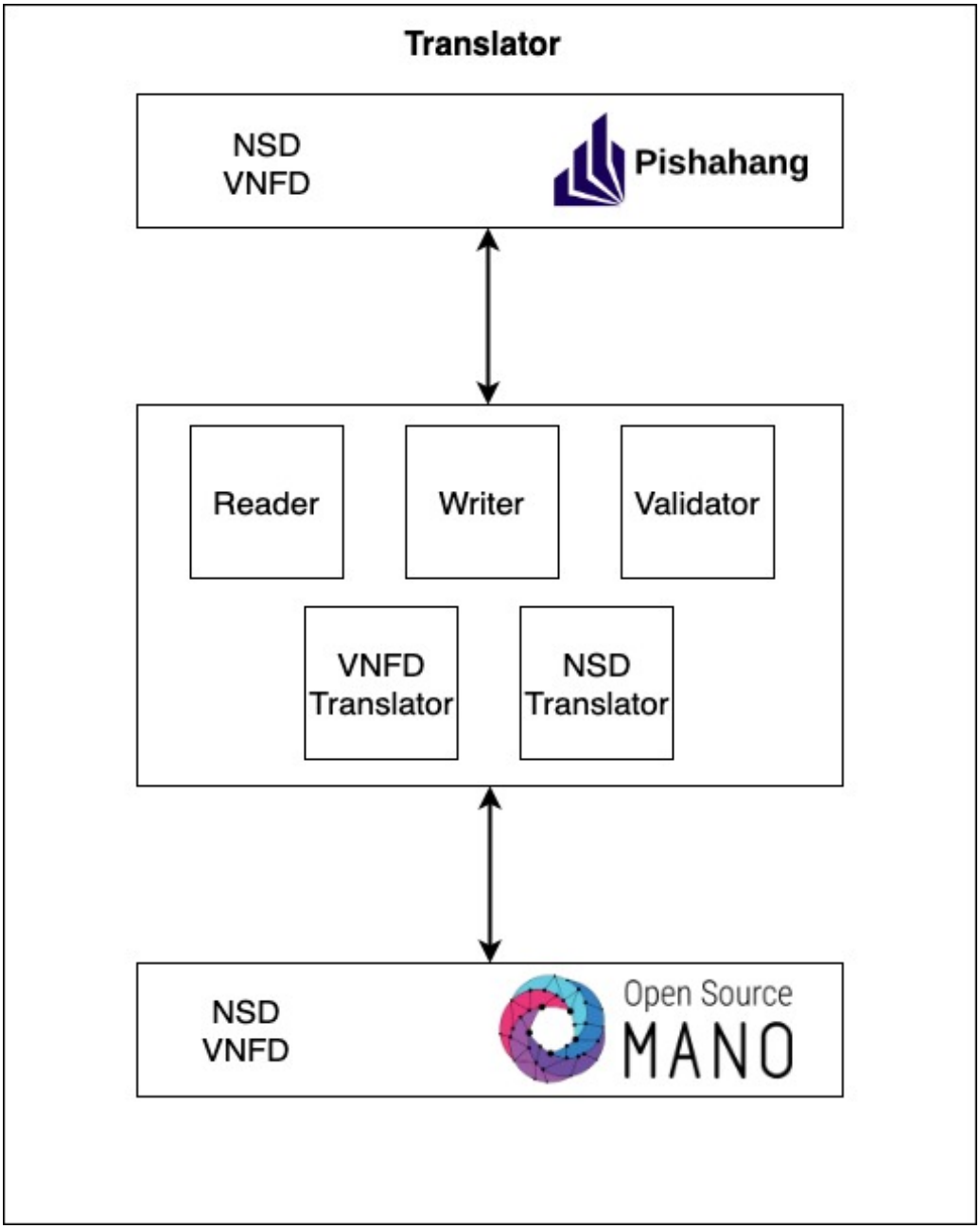

Figure 4.1: Translator

Table 4.1: Descriptor represented DataFrame.

Columns	Description
parent_level	It stores immediate parent key's depth-level
parent_key	It stores immediate parent key
level	It stores current key's depth-level
key	It stores current key
value	It stores current key's value
lineage	It stores current key's entire lineage from the root depth-level

4.1.2 Usage

The Translator engine consists of the following modules:

1. descriptorReader
2. descriptorWriter
3. utilities
4. translator
5. validator

descriptorReader

This module is responsible to read a json/dictionary input (NSD/VNFD) and iterate over the keys and return a generator of an object to the calling program. This generator can be transformed to any python data structure for ease of use and navigation.

The input to this module is a json or dictionary object.

```
from descriptorReader import read_dict

pishahang = pishahang_descriptor ## the descriptor as a json or dict object

### reading a dict/ json content into a pandas dataframe
reader = read_dict()

pishahang_dataset = pd.DataFrame(
reader.dict_parser(pishahang,'root', 1, '0|preroot|0'),
columns=['parent_level', 'parent_key', 'level', 'key', 'value', 'lineage'])


pishahang_dataset.sort_values(ascending=True, by=['level', 'parent_key'],inplace=True)
pishahang_dataset.fillna('NULL', inplace=True)
pishahang_dataset.reset_index(drop=True,inplace=True)
```

Listing 4.1: reader to read a json into a DataFrame

descriptorWriter

This module is responsible to read a python pandas Dataframe input and output a nested json/dictionary maintaining the nested structure in the dictionary.

```
from descriptorWriter import write_dict

### writing from a pandas dataframe to a dict/json object
writer = write_dict()
pishahang_descriptor = writer.translate(pishahang_dataset.sort_values(by='lineage'))
```

Listing 4.2: writer to write a translated DataFrame into a json

utilities

This module is responsible for transforming the keys and map the corresponding values between sonata and osm descriptors. The module includes the 4 functions for translating between sonata and OSM descriptors.

1. translate_to_osm_nsd()
2. translate_to_osm_vnfd()
3. translate_to_sonata_nsd()
4. translate_to_sonata_vnfd()

translator

This is the interface where the actual translation request comes in. After a tranlsation request is received along with a descriptor, it calls the above modules translate and validate the descriptors (NSD/VNFD).

This following function translates OSM descriptor to Pishahang.

```
import pymongo
from validate import validator
from utilities import setup

def toSonata(received_file):

client = pymongo.MongoClient("mongodb://mongo:27017")
setup_obj = setup(client)
validate_obj = validator()

if 'vnfd:vnfd-catalog' in received_file:

## mongoDB document object to store the translated vnfd
doc = setup_obj.db_descriptors["translated_vnfd"]

## getting a translated sonata vnfd
translated = setup_obj.translate_to_sonata_vnfd(received_file)

## validating the translated descriptor which returns a "True"/ "False" string
check = validate_obj.sonata_vnfd_validate(translated)

if check == "True":
## insert into mongoDB as a Document
temp = doc.insert_one(translated)
translated_ref = temp.inserted_id

elif 'nsd:nsd-catalog' in received_file:

## mongoDB document object to store the translated nsd
```

```
30 doc = setup_obj.db_descriptors["translated_nsd"]
31
32 ## getting a translated sonata nsd
33 translated = setup_obj.translate_to_sonata_nsd(received_file)
34
35 ## validating the translated descriptor which returns a "True"/ "False" string
36 check = validate_obj.sonata_nsd_validate(translated)
37
38 if check == "True"
39 ## insert into mongoDB as a Document
40 temp = doc.insert_one(translated)
41 translated_ref = temp.inserted_id
42
43 return {"descriptor":translated ,"VALIDATE STATUS" :check}
```

Listing 4.3: Translating OSM descriptor to Pishahang

The following function translates Pishahang descriptor to OSM.

```
import pymongo
from validate import validator
from utilities import setup

def toOsm(received_file):

client = pymongo.MongoClient("mongodb://mongo:27017")
setup_obj = setup(client)
validate_obj = validator()

if 'network_functions' in received_file:

## mongoDB document object to store the translated nsd
doc = setup_obj.db_descriptors["translated_nsd"]

## getting a translated osm nsd
translated = setup_obj.translate_to_osm_nsd(received_file)

## validating the translated descriptor which returns a "True"/ "False" string
check= validate_obj.osm_validator(translated)

if check == "True":
## insert into mongoDB as a Document
temp = doc.insert_one(translated)
translated_ref = temp.inserted_id

elif 'virtual_deployment_units' in received_file:

## mongoDB document object to store the translated vnfd
doc = setup_obj.db_descriptors["translated_vnfd"]

## getting a translated osm vnfd
translated = setup_obj.translate_to_osm_vnfd(received_file)

## validating the translated vnfd which returns a "True"/ "False" string
check= validate_obj.osm_validator(translated)

if check == "True":
## insert into mongoDB as a Document
temp = doc.insert_one(translated)
translated_ref = temp.inserted_id

return {"descriptor":translated ,"VALIDATE STATUS" :check}
```

Listing 4.4: Translating Pishahang descriptor to OSM

validator

The validator validates the descriptors presented for translation. It differentiates between OSM, Pishahang and Sonata descriptors and also between NSD's and VNFD's. The sonata descriptor validation is mainly based on schema, implemented using python library jsonschema.draft4validator, this library helps by proving the complete path of the error which comes very handy while debugging. The Validation for OSM is slightly complex than that compared to sonata validator, OSM validator can be divided into two parts. the first part validates the descriptors using schema with the same setup as that of sonata and the second part uses python object class for validation, this is done to replicate the validation method used by OSM hence to make sure that any descriptor that is validated by translator will be accepted by OSM.

4.1.3 Challenges

The initial challenges faced while designing the translator was mapping the "*required*" keys between OSM and Pishahang descriptors. Figuring out the common functionalities of the respective "*required*" keys in OSM and Pishahang was the priority and a mapping was created. The other "*optional*" keys which were exclusive for each MANOs were also identified and sidelined for future scope.

4.1.4 Future scope of this work package

Translator engine currently doesnot support the following:

1. Forwarding Graph
2. Juju charms in OSM
3. Monitoring Parameters

Forwarding Graph

Although translation of forwarding graphs between OSM and Pishahang has been implemented, we could not verify the translation as the forwarding graph logic is not currently feasible in both MANOs.

Juju charms in OSM

MANOs provide programmable and flexible management and orchestration of VNFs. OSM provides this flexibility through juju charms and Pishahang provides this through SSM/FSM (a container based solution). Because of these technological differences, direct translation between juju charms in OSM, and SSM (Service Specific Manager) and FSM (Function Specific Manager) in Pishahang is not possible.

As an alternative, it is possible to add charms functionality to Pishahang so that a descriptor containing juju charms can be deployed in both Pishahang and OSM with direct translation. This can be achieved by adding or modifying below things in Pishahang.

1. Modify packaging and unpackaging techniques in Pishahang to accept charm package along with descriptors
2. Add additional keys in descriptors to mention charm name, actions and vnf index similar to OSM
3. Update Pishahang installation code to install juju and charm programs and tools
4. Create an interface to execute actions on VNFs
5. Create new container or component to perform actions on specified VNFs
6. Manage removal or deletion of charms after life cycle of Network Service

Monitoring Parameters

Translation could be extended to include the monitoring parameters as well. However verifying monitoring parameters were currently not feasible in OSM, so we sidelined for future scope.

4.2 Splitter

Figure **??** graphically represents the splitting architecture of SDS. SDS can split either OSM NSD or Pishahang NSD. Python base classes are used for different sections of a NSD which encapsulate all the attributes and its values into a single unit which makes it very easy to process. Once the objects are set they are passed to different splitting functions based on there type. We have two different processing units for OSM and SONATA. Following are some functions responsible for splitting the NSD.

- **Validate:** Validation of the incoming request from MANO for splitting happens is done before actual splitting. Validation checks if the request has correct VNF ids. It also checks if the list of VNFs specified in the request is matching with the list of VNFs in the original NSD file.
- **Set connection point reference for virtual functions:**After validation, as per the incoming request multiple set of empty NSD objects are created. This step updates the empty NSD objects with connection points. These connection points are either of type "external" or "management". Each sub NSD will have its own connection points just like its parent NSD.
- **Split Network Functions:** After creating connection points for each sub NSDs, VNF objects are set in the updated NSD objects as per the request from MANO. Each set of VNFs from the request is set in each of the NSD objects.
- **Set Connection Points:**
- **Split Virtual Links:** When a NSD is splitted into different parts, its topology changes. Change in topology results in changing of Virtual Links. For example if A, B and C are three Nfs and we are splitting them in such a way so that A and B remain in one NSD and C in separate NSD. A virtual link between B and C now does not make sense. So this link should be broken down and B's output should be connected to the external end point which was connected to C's input earlier. This function splits these kind of Virtual Links.
- **Split Forwarding Graph:** As explained in the above section, once the topology changes, the respective Forwarding graph also changes. Split forwarding graph pulls out the set of connection points and newly created virtual links and sets them in the sub NSDs.
- **Set General Information:**
- **Create Sub-NSDs:**

Once the Splitting is done, create file is responsible for creating YAML files depending on the number of sub NSDs created. These files are saved in the file system which can be downloaded or moved forward to the adopter for deployment purpose. Following figure **??** graphically represents the splitting architecture.

4.2.1 Usage

SDS is implemented as a micro-service which can be used independently from Translator or Wrapper by making a post call to the SDS. Following code snippet describes how to call SDS using POST call.

```

splitter_url=http://$HOST:8003/Main_splitter/split

# Body: descriptor contains NSD, vnfid_set contains set of VNF ids
nsd = { 'descriptor' : descriptor, 'sets': vnfid_set}
```

```
6
7 LOG.info("Calling Scramble Splitter..." )
8 response  = requests.post(splitter_url,
9 data=json.dumps(nsd_to_split))
10
11 print(response)
```

Listing 4.5: POST call to SDS

Following are some of the important functions which helps SDS in splitting the NSD with respective code snippet.

Basic Python classes for NSD Schema

Splitting

"splitsonata" calls the splitting function one by one to split the list of objects created out of NSD. Following code snippet shows the sequence of function calls.

```
1
2 def split_sonata(self):
3 if self.validate() is not False:
4 self.create_new_function_sets()
5 self.set_connection_point_refs_for_virtual_functions()
6 self.split_network_function()
7 self.set_connection_points()
8 self.split_virtual_links()
9 self.split_forwarding_path()
10 self.set_general_information()
11 return self.create_files()
12 else:
13 print("Validation Failed!!")
```

Listing 4.6: Sequence of function calls

Validate

Validate method validates the request coming from the MANOs. For example, if MANO is requesting a NSD to be split into three parts but the original NSD contains just two VNFs then the SDS will throw validation error.

```
1
2 def validate(self):
3 size = 0
4 list_network_function = []
5 for network_function_set in self.network_function_sets:
6 size = size + len(network_function_set)
7 for network_function in network_function_set:
8 list_network_function.append(network_function)
9
10 if size != len(self.utilityFunctions.list_nf):
11 return False
12 if len(list_network_function) != len(set(list_network_function)):
13 return False
```

Listing 4.7: Splitting Request Validation

Split Network Functions

This function updates the sub NSDs with set of network functions and there properties provided in the request.

```
1
2  def split_network_function(self):
3
4  for network_function_set in self.network_function_sets:
5
6  sub_nsd = SonataSchema.NSD("", "", "", "", "", "", [], [], [], [])
7
8  network_function_list = []
9
10 for network_function in network_function_set:
11
12 network_function_list.append(self.get_network_function_object(network_function))
13
14 sub_nsd.networkFunctions = network_function_list
15
16 self.NSDs.append(sub_nsd)
```

Listing 4.8: Network Function Splitting

Split Forwarding Graph

Content to be added

4.2.2 Challenges

The NSD schema of Pishahang and OSM contains a lot of elements. However the challenge we faced was choosing which elements to include for splitting. We tackled it by including mandatory elements and few optional elements from the schema which were present in the input NSD.

4.2.3 Future scope of Service Descriptor Splitter

SDS can currently split NSD of Pishahang and OSM. SDS is built in such a way that it can be implemented for MANO frameworks as well. To implement SDS for a new MANO framework one can refer the implementation of either Pishahang or OSM. First step would be to create basic python classes from the NSD schema of the MANO framework then writing the utility functions to pull the information from the NSD file and store it in the objects of the basic python classes. Lastly writing splitting functions to actually split the list of objects in two or more parts.

Also, the current implementation considers all mandatory elements and a few optional elements from a NSD schema for splitting which can be extended to include other fields (Provided they are present in the input NSD for splitting).

Current implementation of SDS can split a forwarding graph of a NSD (Pishahang) with just three VNFs. Splitting of a forwarding graph is implemented by keeping future implementation for more than three VNFs in mind (Refer Splitting of Forwarding graph section)

4.3 Adaptor

Facilitating easy communication between MANOs is an important aspect of scramble. Adaptor is a component that enables communication between MANOs by wrapping the REST APIs of MANOs in python code.

Python MANO Wrappers (PMW) is a uniform python wrapper library for various implementations of NFV Management and Network Orchestration (MANO) REST APIs. PMW is intended to ease the communication between python and MANO by providing a unified, convenient and standards oriented access to MANO API.

To achieve this, PMW follows the conventions from the ETSI GS NFV-SOL 005 (SOL005) RESTful protocols specification. This makes it easy to follow and the developers can use similar processes when communicating with a variety of MANO implementations.

PMW is easy to install, use and well documented. Code usage examples are available along with the detailed documentation at the following link `https://python-mano-wrappers.readthedocs.io/en/adaptor/`.

PMW is planned and released as an independent library. In scramble, PWM helps in inter communication of different instances of MANO, thereby creating opportunity for more advanced feature set, for example, hierarchical scaling. Operations such as on-boarding of NSD and VNFD, instantiation and termination of NS can be performed with ease.

4.3.1 Architecture & Work flow

Standards based approach is a fundamental design principle behind PMW's design. A Common interface template is defined in compliance with SOL005 which contains the blueprint for all the methods mentioned in the standards. These methods are divided into different sections as per SOL005 into the following:

- **auth:** Authorization API
- **nsd:** NSD Management API
- **nsfm:** NS Fault Management API
- **nslcm:** Lifecycle Management API
- **nspm:** NS Performance Management API
- **vnfpkgm:** VNF Package Management API

In the figure 4.2, different sections of PMW are visualized. As part of the scramble project, support for Open Source MANO (OSM) and Sonata was implemented based on the common interface. This is represented by the dotted lines to OSM and Sonata modules. These modules are based on the common interface and implement the methods it has defined. Wrappers also support additional functionalities of Pishahang, which is an extension of Sonata.

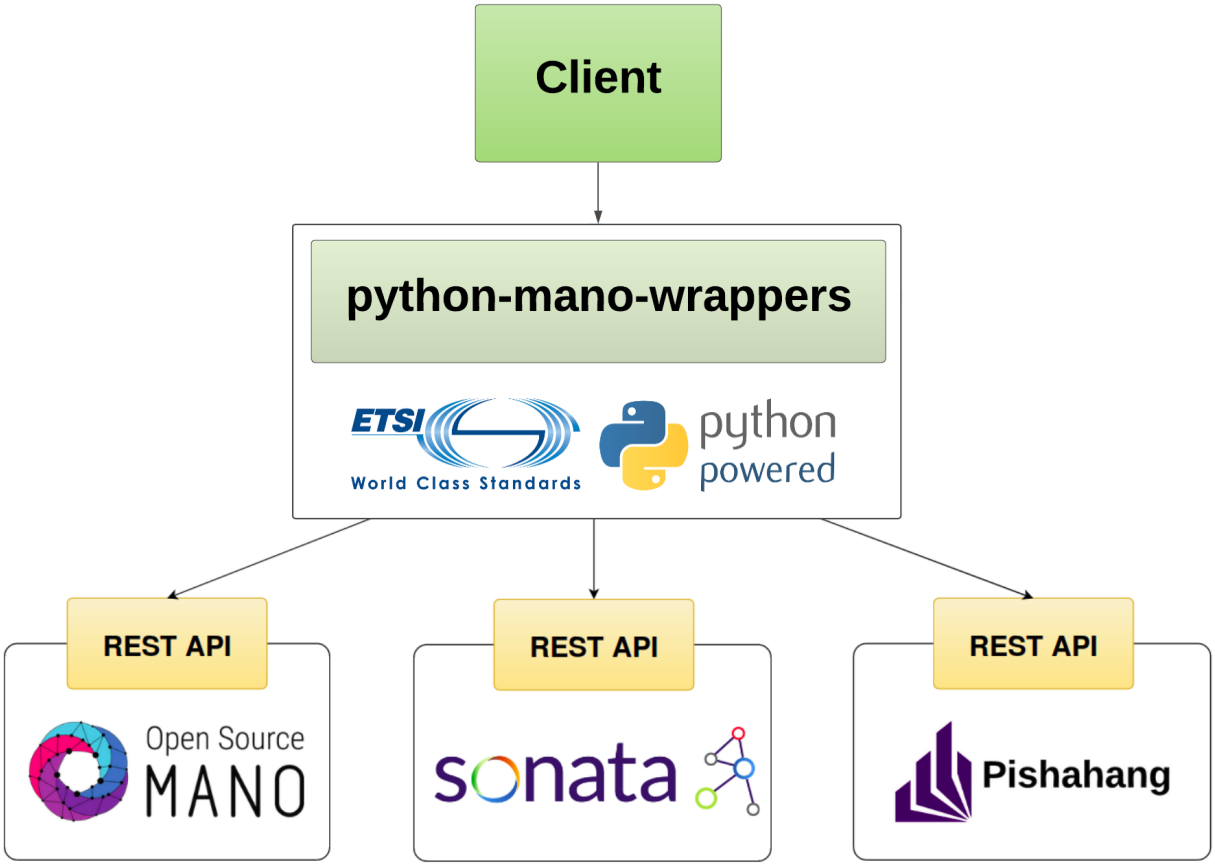

Figure 4.2: PWM Common interfaces

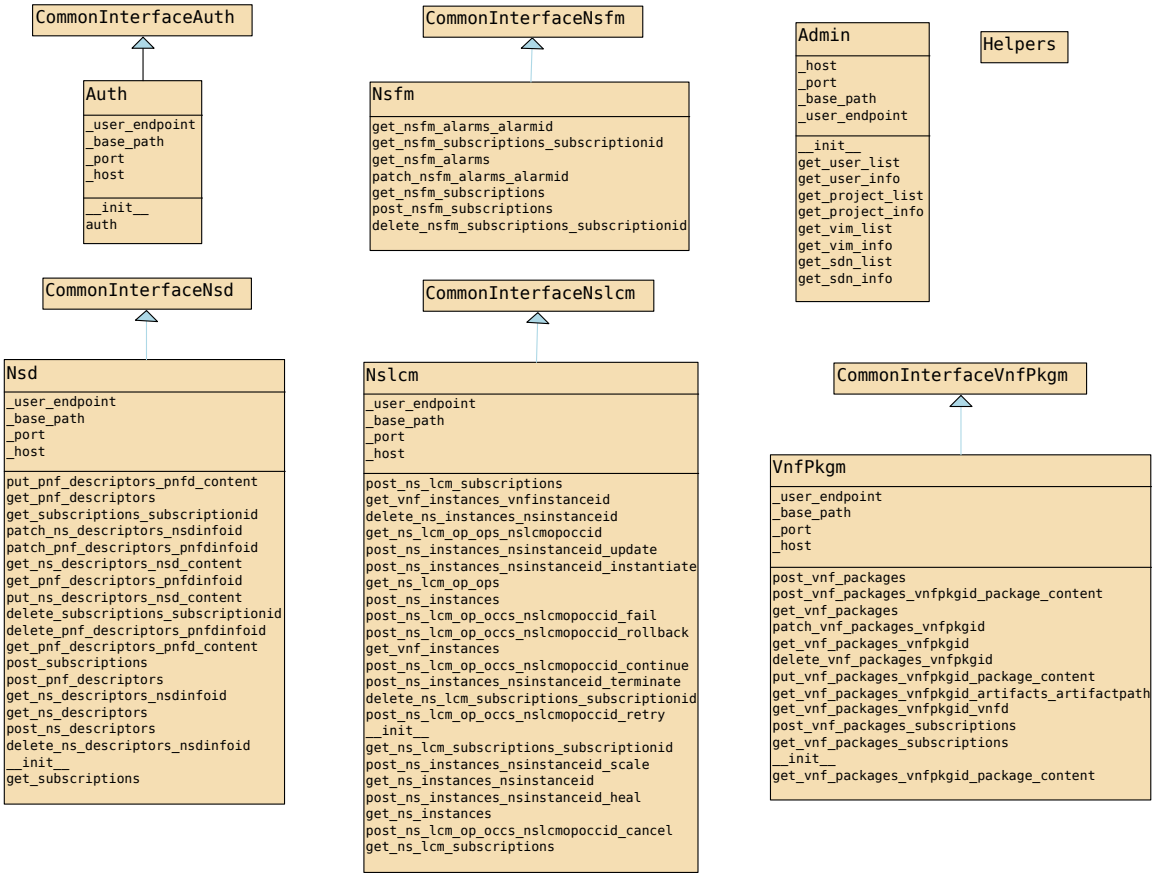

Figure 4.3: OSM Wrappers implemented based on the CommonInterface base classes

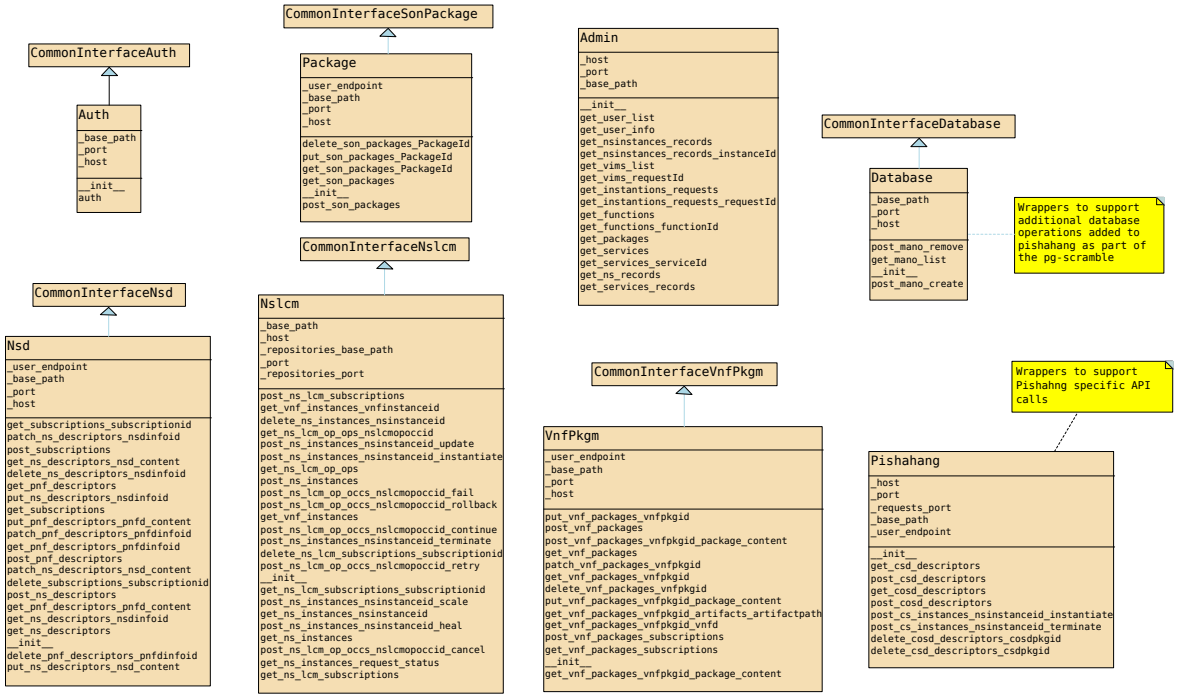

Figure 4.4: Pishahang Wrappers implemented based on the CommonInterface base classes

4.3.2 Installation and usage

PWM can be installed using pip by using this command `pip install python-mano-wrappers`.

A simple script to get started with PWM is shown in the Listing 4.9, here, the wrappers are imported and a client object is created according to the MANO type. Currently supported are OSM and Sonata. Such a client object can be used to make REST calls relevant to the MANO type. An example usage to retrieve all the network service descriptors of OSM can be seen from the listing 4.10, here, the OSMClient module is used to first fetch an auth token and further using the auth token to fetch the relevant information, in this case NSD descriptors.

```
import wrappers

username = "admin"
password = "admin"
mano = "osm"
# mano = "sonata"
host = "osmmanodemo.com"

if mano == "osm":
  _client = wrappers.OSMClient.Auth(host)
elif mano == "sonata":
  _client = wrappers.SONATAClient.Auth(host)

response = _client.auth(
               username=username, password=password)

print(response)
```

Listing 4.9: Simple wrapper code to fetch token

```
import wrappers

osm_nsd = wrappers.OSMClient.Nsd(HOST_URL)
osm_auth = wrappers.OSMClient.Auth(HOST_URL)

_token = json.loads(osm_auth.auth(
                    username=USERNAME,
                    password=PASSWORD))

_token = json.loads(_token["data"])

response = json.loads(osm_nsd.get_ns_descriptors(
                        token=_token["id"]))
response = json.loads(response["data"])
```

Listing 4.10: Code to fetch all NSDs in OSM

4.3.3 Challenges

Implementing such a python wrapper for a REST API is straight forward from the implementation perspective. However, the challenges that we faced are when identifying the required functional documentation from the respective MANOs. OSM and Sonata do not yet fully support the ETSI suggested endpoints and this combined with the lack of unified documentation, made it difficult in the begining to decide on the scope of supported functionalities.

4.3.4 Future scope of wrappers

PWM is built with easy maintanability and feature addition in mind. PWM makes it easy to add support for other MANOs. We expect MANO developers to use the common interface that we have suggested to add support to their REST APIs in python.

4.4 Pishahang-Scramble Integration

Service Life-cycle Management(SLM) component of Pishahang carries out the main task of orchestration. As a result it was all the more relevant to add one more aspect to it for integrating and handling Scramble components.

The main class of SLM, *ServiceLifecycleManager*, contains a list of member functions for carrying out the entire orchestration. One of the many member functions, *SLM_mapping*, is responsible for handling the descriptors payload and creating a mapping of network funtions to the available VIMs. Keeping the original flow of SLM intact, we extended the main class to include a new member function (*SLM_mapping_scramble*) to handle request addressed for mapping the network functions to the available MANOs.

When a request to instantiate the network service from the BSS (son-bss) is made, the gatekeeper (son-gkeeper) gets the request payload from BSS and creates a instantiation request to hand it over to SLM. For differentiating the instantiation request between a "*normal*" call and a "*scramble*" call, we added a "*scramble*" button in BSS.

```
instantiateScramble:function(id, ingresses, egresses, ENV, selectedmanos, manodetails){
var defer=$q.defer();

{...} ## unchanged

var data={"service_uuid":id, "ingresses": ingresses, "egresses": egresses, "scramble":
    true, "selectedmanos":selectedmanos, "manoips":manodetails};
$http.post(ENV.apiEndpoint+"/requests",data)
.then(function successCallback(result){defer.resolve(result)})
.catch(function errorCallback(error){defer.reject(error)});

return defer.promise;
},
```

Listing 4.11: BSS instantiateScramble function

This function sends the gatekeeper a payload which consists of a token "*scramble*", which set to true, and a list of mano details.

The gatekeeper also ensures the payload contains this additional package when it creates a new instantiation request and informs the SLM.

```

post '/requests/?' do
log_msg = MODULE + '::POST /requests'
original_body = request.body.read
logger.debug(log_msg) {"entered with original_body=#{original_body}"}
params = JSON.parse(original_body, quirks_mode: true)
logger.debug(log_msg) {"with params=#{params}"}

# we're not storing egresses or ingresses
egresses = params.delete 'egresses' if params['egresses']
ingresses = params.delete 'ingresses' if params['ingresses']
user_data = params.delete 'user_data' if params['user_data']

begin
{...} ## unchanged
start_request['egresses'] = egresses
start_request['ingresses'] = ingresses
start_request['user_data'] = user_data

if params['scramble'] == true
start_request['scramble'] = true
```

```
start_request['selectedmanos'] = params['selectedmanos']
start_request['manoips'] = params['manoips']

end
{...} ## unchanged
end
```

Listing 4.12: create instantiation request in gatekeeper

4.4.1 SLM_mapping_scramble

This member function, together with four other helper functions, is responsible for treating requests routed for translation, splitting and sending the descriptor for instantiating in other MANOs. The 4 other helper functions used by it are as follows:

Table 4.2: Helper functions.

Functions	Description
get_network_functions	It gets the network function names and ids from the service descriptor.
random_combination	It maps different network functions to different MANOs in random combination.
send_to_osm	This function sends one part of the splitted service descriptor and its network functions to OSM Mano instance.
send_to_pishahang	This function sends one part of the splitted service descriptor and its network functions to PISHAHANG Mano instance.
inform_gk_instantiation_scramble	This function is used to inform the gatekeeper to create a dummy NSR in the only case when none of the network functions, of the original NSD, are instantiated by this SLM. This way the instantiation request is not rolledback

This function is created by extending the original **SLM_mapping** function to include the logic to map each network functions to a MANO and then send and instantiate them in their mapped MANOs:

```
def SLM_mapping_scramble(self, serv_id):
"""
This method is used if the SLM is responsible for the placement.
:param serv_id: The instance uuid of the service
"""
corr_id = str(uuid.uuid4())
self.services[serv_id]['act_corr_id'] = corr_id

LOG.info("Service " + serv_id + ": Calculating the placement ")
topology = self.services[serv_id]['infrastructure']['topology']
```

```
11
12 ## getting all manos information from payload
13 mano_dict = self.services[serv_id]['payload']['selectedmanos']
14 mano_details = self.services[serv_id]['payload']['manoips']
15 mano_list = []
16
17 ## creating a list of selected manos and its corresponding details
18 for key, val in mano_dict.items():
19 for manos in mano_details:
20 if manos['name']==key and val == True:
21 mano_list.append(manos)
22
23 ## original flow with scramble portion added
24 if 'nsd' in self.services[serv_id]['service']:
25
26 descriptor = self.services[serv_id]['service']['nsd']
27 functions = self.services[serv_id]['function']
28 original_nsd_uuid = descriptor['uuid']
29
30 ##-------------------------------------------------------------------##
31 ##-----------------------SCRAMBLE PART-------------------------------##
32 ##-------------------------------------------------------------------##
33
34 # create a set of vnfs for different MANO frameworks through random logic
35 # Number of splits is by default 2 except if the number of MANOs and number of VNFs are
      equal.
36
37 function_list = self.get_network_functions(descriptor)
38 rndm_sets = self.random_combination(function_list, mano_list)
39
40 if(len(rndm_sets) > 1): # if there are more than 1 MANOs, SCRAMBLE-splitter is called
     to split the NSD
41 vnfid_set = [sets[0] for sets in rndm_sets]# vnf-ids of sets 1 and 2
42
43 # send the random vnf split to SCRAMBLE Splitter and get back sub NSDs for each split.
44 splitter_url = os.environ['splitter_url']
45 nsd_to_split = { 'descriptor' : descriptor, 'sets': vnfid_set}
46
47 response  = requests.post(splitter_url,
48 data=json.dumps(nsd_to_split))
49
50 nsds_splitted = json.loads(response.text) # get back 2 sets of sub-nsds
51
52 else:
53
54 nsds_splitted = {"message" : [descriptor]}
55
56
57 # logic to check which vnf is to be send to which MANO
58
59 function_pish = [] # list to store vnfs for MAIN_PISHAHANG
60 main_pish_nsd = {} # string to store nsd for MAIN_PISHAHANG
61
62 for i,sets in enumerate(rndm_sets):
63
64 if sets[2][0]['type'] == 'MAIN_PISHAHANG':
65
66 main_pish_nsd = nsds_splitted['message'][i]
67
68 for vnf in functions:
69 if(vnf['vnfd']['name'] in sets[1]):
```

```
70 function_pish.append(vnf)
71
72
73 elif sets[2][0]['type'] == 'PISHAHANG':
74
75 self.send_to_pishahang(serv_id, sets, functions, nsds_splitted['message'][i])
76
77 elif sets[2][0]['type'] == 'OSM':
78
79 self.send_to_osm(serv_id, sets, functions, nsds_splitted['message'][i])
80
81 # remove the vnfs which are sent to other MANO from self.services[serv_id]['function']
82 NSD = main_pish_nsd
83 functions = function_pish
84 NSD['uuid'] = original_nsd_uuid
85
86 self.services[serv_id]['service']['nsd'] = NSD
87 self.services[serv_id]['function'] = functions
88
89 if(functions == []):
90
91 ## put up a dummy nsr when there is no network functions is available for this mano. So
       as to keep the unique UUID of this instantiation request in ledger instead of
      forced rollback.
92
93 self.inform_gk_instantiation_scramble(serv_id)
94
95 else:
96 content = {'nsd': NSD,
97 'functions': functions,
98 'topology': topology,
99 'serv_id': serv_id}
100 else:
101 {...} ## unchanged
```

Listing 4.13: Extended **SLM_mapping_scramble** function

5 Use Cases

In this chapter, the use cases are discussed.

5.0.1 Cross-MANO Framework Interaction

The MANO frameworks used by every network service provider varies from one another. The software suite with the help of SDT and SDS enables the deployment of network services across different frameworks.

For instance: Consider two network service Operators using different MANO frameworks. One of them uses Sonata framework [DKP+17] and another operator uses OSM framework [Ers13]. These frameworks have different NSD schemata(refer 2.1.1 and 2.1.2). NSD schemata contain VNFs, virtual links, and VNF forwarding graphs and also describes the deployment of a network service. By using a translator and splitter, these NSD schemata can be translated and split into a framework-specific schema. With this, operators can deploy and manage network services across different MANO implementations.

5.0.2 Hierarchical Orchestration

By using MA, dynamic instantiation of multiple MANO instances and inter-operability between different MANO frameworks can be achieved. The operator will be able to handle the resources in an efficient manner, as one MANO framework can manage a limited number of service requests, operators can explore options to include additional MANO instances under the existing MANO instance to mitigate the traffic load on a single instance. The resources can be provisioned based on the number of requests. This helps the operator in extending their profitability.

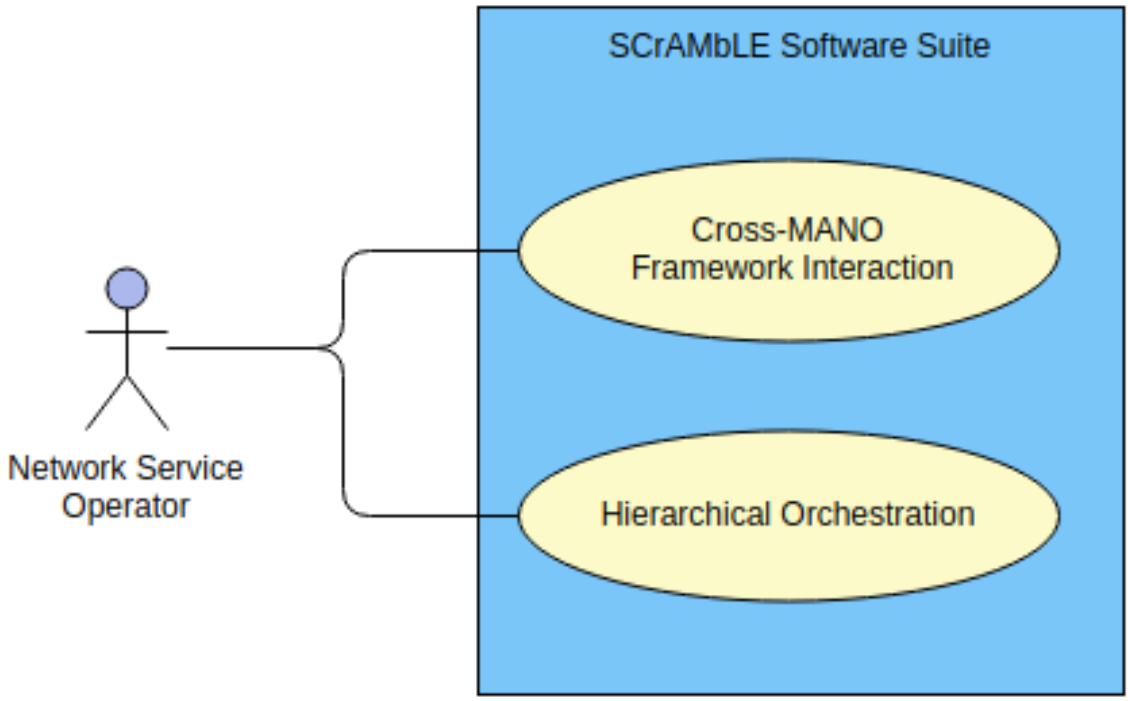

Figure 5.1: Use Case Diagram

Actors : The Network Service Providers who would use features of SCrAMbLE.

6

MANO Scalability

In this chapter we discuss the two directions we explored to investigate MANO orchestrator scalability. First, we discuss the scalability plugin that was added to pishahang. The scalability plugin adds 3 main functionalities to pishahang, 1) spawn new child instances of pishahang by allocating new physical resources, 2) Redirecting requests from parent MANO to the child instances and 3) managing the state of child instances. Second, we discuss the experiments conducted on OSM and Pishahang to understand the resource utilization. We also propose a more generic framework to characterize and analyze MANO under load.

6.1 Scalability Plugin

MANO framework faces scalability challenges in large-scale deployments. The amount of infrastructure a single instance of MANO framework can manage is limited. The scaling of MANO should be addressed to avoid it from being the bottleneck in the NFV paradigm.

We propose a hierarchical scaling mechanism for MANO by implementing and evaluating a scalability plugin (SPL) in pishahang. SPL adds the following features to pishahang 1) Spawn/Terminate child MANO instances, 2) State management between instances, 3) Redirection of requests from parent instance to child instances and 4) Monitoring of system load on all instances

6.1.1 Architecture

SPL is built based on the "base plugin framework" of SONATA. The figure 6.1 from SONATA documentation shows the high-level architecture of the SONATA MANO framework. The plugins communicate over the message broker using AMQP with other plugins and together facilitate the functions of MANO.

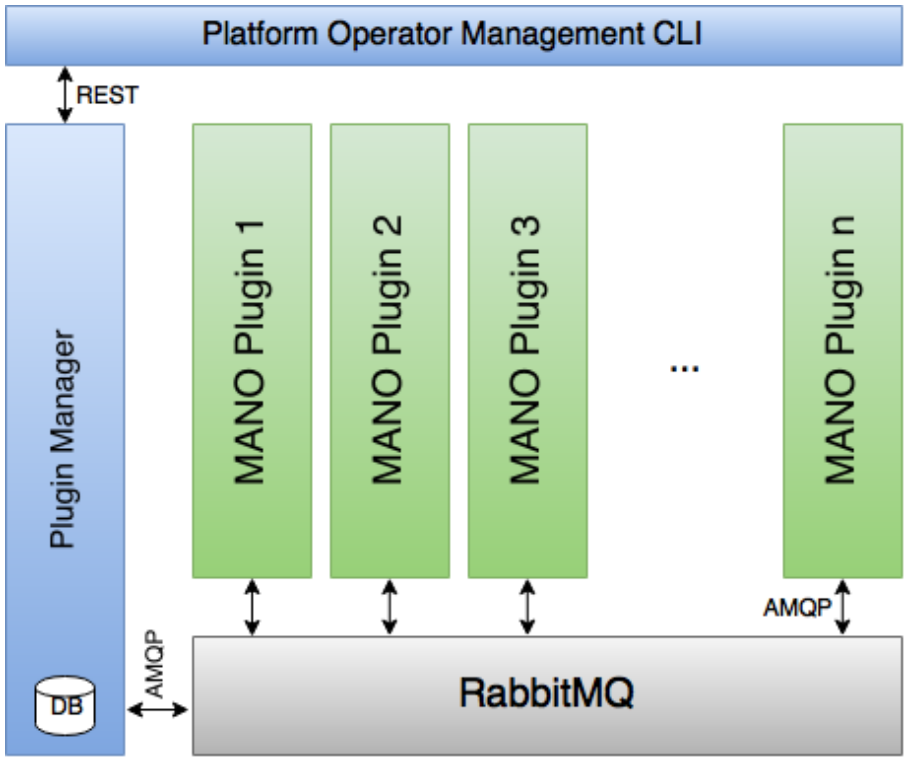

Figure 6.1: SONATA MANO plugin architecture

The architecture of the scalability plugin is shown in the figure 6.2. SPL mainly consists of 1) **Scaling Manager:** is responsible for the overall workflow of the SPL, which includes monitoring, spawning, termination and state management of MANO instances. 2) **MANO Manager:** is responsible for sending the on-boarding and instantiation requests to the parent MANO using python-mano-wrappers. SPL co-ordinates with the service lifecycle management plugin in pishahang to provide the features discussed above. The workflow of this is discussed in the further sections.

Scaling Manager

Features of the scaling manager are discussed briefly in this section.

Monitoring

SPL needs to continuously monitor the system resource utilization and take scaling decisions based on that. We are using the *average system load* values given by the linux kernel for taking the scaling decisions. System load is the number of processes which are being executed by the CPU or in waiting state. The linux kernel provides 3 average values of system load calculated over a period of 1, 5 and 15 minutes.

As a proof of concept, SPL uses *5m* and *15m* moving averages as a heuristic for scaling decisions. *1m* average is not considered to avoid taking decisions on a short lived spike in system load. SPL has the following simple rule set listed below, note that the configurable threshold value is used to take relevant actions. For simplicity, we consider "0.7" as the threshold value as it is generally used in the wild by system administrators.

- **Rule 1**
  If *5m* load average is more than the *threshold* value, then, SPL will consider this as a warning sign and start preparing a child MANO instance.
- **Rule 2**
  If *15m* load average is more than the *threshold* value, then, SPL will consider this as a critical sign and start redirecting service requests on the parent mano to the child MANO instance.
- **Rule 3**
  If *15m* load average of both the parent and child instances are less than the *threshold* value, then, SPL will consider this as decreasing load and terminate the child MANO instance.

List 6.1: Scaling Plugin Rules

SLM Communication

The monitoring of the load information happens in the SPL, however, the service requests are handled by the SLM plugin. SLP communicates information about the current load of the instances to the SLM and based on this response, SLM decides whether to continue serving the request or to forward it to one of the child MANOs.

The forwarding logic is implemented on the SLM plugin of pishahang. The forwarding logic includes on-boarding the VNFD/NSD and instantiating it.

Spawning

Spawning refers to the action of instantiating a child instance of pishahang by allocating new physical resources on existing infrastructure. According to Rule 1 from 6.1, when the threshold is reached, a request to instantiate a new instance is sent to the Mano Manager. Mano Manager will instantiate a new instance of pishahang and returns the IP address of the newly deployed instance. This new instance is now added to the list of child MANO instances and continuously monitored.

Termination

Once the load on the parent MANO and child MANO instances are reducing, it becomes unnecessary and expensive to use the allocated physical resources of the child MANO. According to Rule 3 from 6.1, the termination request of the child MANO is sent to the MANO Manager and the metadata generated by the child instances are saved on the parent instance. Metadata that is saved includes the VNF records and network service records.

MANO Manager

MANO Manager is responsible for the end communication with the parent MANO. MANO Manager receives an instantiation request from the Scaling Manager and uses python-mano-wrappers to make an instantiation request to the parent MANO to instantiate a child MANO instance.

MANO Manager will wait for successful instantiation of the instance and return the IP address of the instance to the Scaling Manager.

Service Descriptors

We designed a basic function descriptor and service descriptor in the schema supported by the pishahang MANO framework. The descriptors are stored as part of the scaling plugin and used by the MANO Manager to on-board and instantiate upon request. We have created the descriptors for both OSM and Pishahang.

MANO Image

A Virtual Machine (VM) image with MANO pre-installed is used along with the service descriptors to spawn new instances of MANO. As a proof of concept, we have created images with OSM and pishahang. However, the pishahang image that was created was unstable and has unexpected behavior. These images are first created locally and later uploaded to the VIM where the instances will be created. In our tests we used OpenStack as the VIM.

6.1.2 Workflow

SLM with SLP

The workflow of how the SLM and SLP interacts is shown in the figure 6.3. At first, an instantiation request comes into the parent mano instance and this is first seen at SLM. SLM will then request SLP for the current system load status. If the parent MANO is loaded, SLP will reply with the details of the child MANO instance, which will then be used by SLM to redirect the incoming request by on-boarding and instantiating the network service on the child instance. If the parent MANO is not loaded, then, SLM will continue its normal flow to deploy the network service

Scaling Plugin Workflow

In the figure 6.4, we visualize the rules set of SLP from the List 6.1. SLP is always in the monitoring loop and takes actions according to the system load and the rules set.

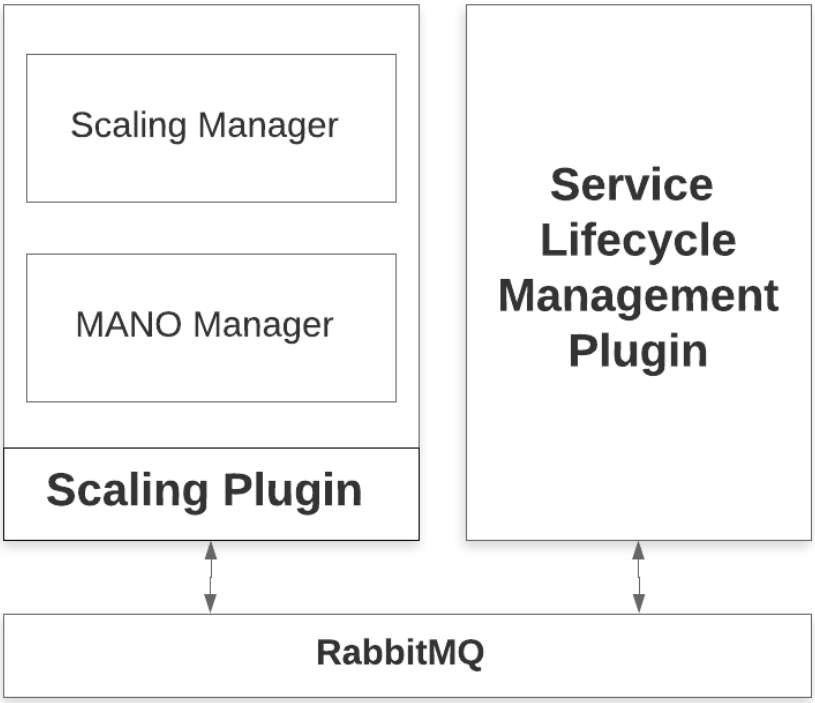

Figure 6.2: Scaling plugin architecture

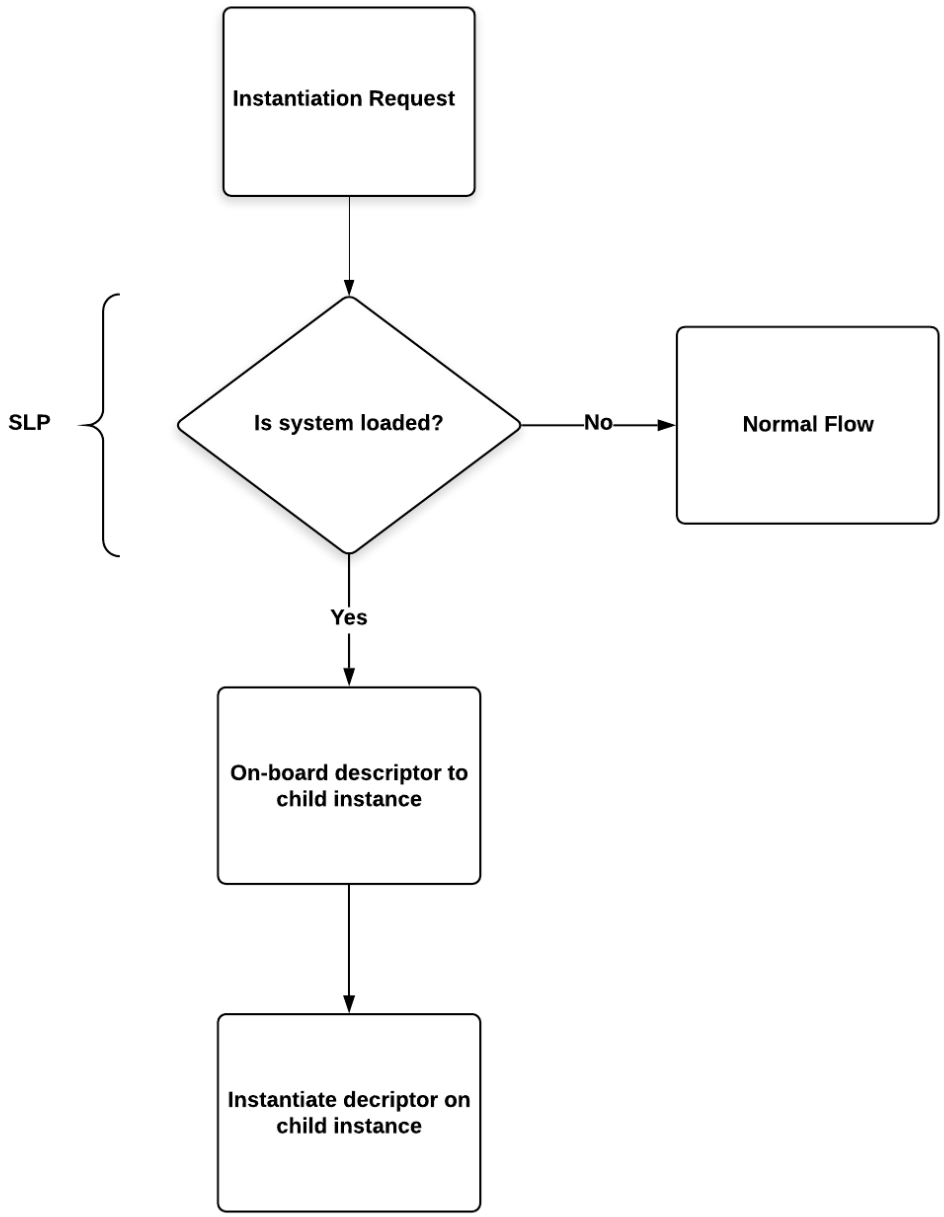

Figure 6.3: Service Lifecycle Manager with Scaling Plugin Workflow

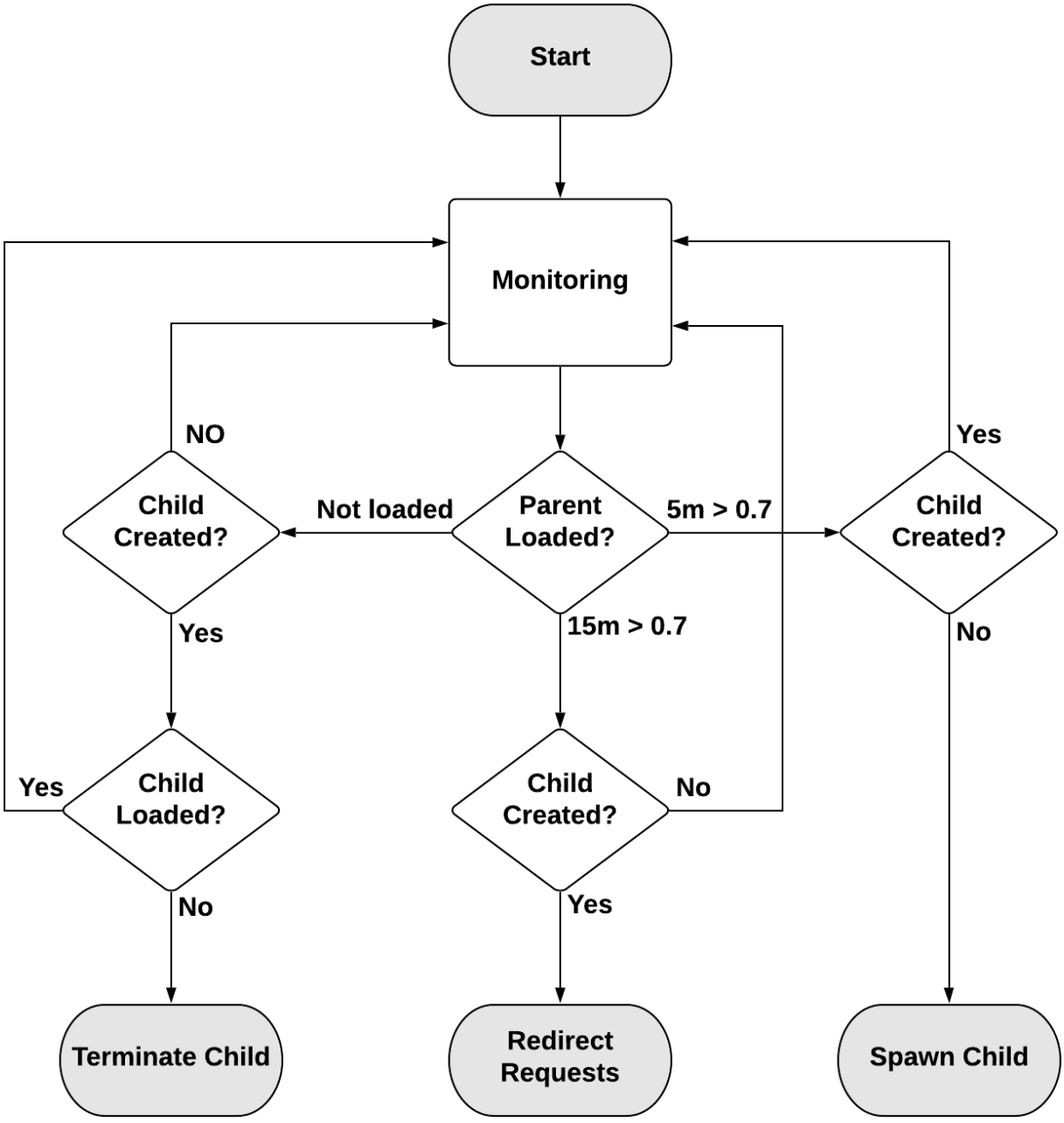

Figure 6.4: Scaling Plugin Workflow

6.2 Experiments

One other task under the MANO scalability investigation was to observe the resource utilization in OSM and Pishahang. To do this we used our own python-mano wrappers to instantiate multiple requests at a time. The next step was to decide on the number of service requests to instantiate. We could simply instantiate any number (like 1,000 or 5000) of service requests but we had to make sure all those requests get successfully instantiated. For this, the infrastructure was the only crucial factor. We used a 16 core virtual machine for installing VIM(Openstack). Since one virtual CPU equals 16 physical CPUs, we have 16 * 16 = 256 cores. Hence we first assigned 1 CPU to one service request. We instantiated 256 service requests. They were not successfully instantiated. Next, we came up with another parameter called Requests Per Minute(RPM). The first attempt was setting the RPM to be 60 and instantiating 256 service requests. This run was not successful. The next run was 250 service requests at 60 RPM. The requests failed to get instantiated. The third run was 200 instances at 30 RPM, which was also a failure. The idea was to find the right combination of number of service requests and RPM. Finally 180 requests at 30 RPM was found to be suitable for our experiment.

Next, we had to decide which VNF image to use for our experiment. We used a basic cirros VNF, we instantiated 180 such service requests. We designed their NSDs to have a single VNFD.

6.2.1 Testbed

This subsection describes the experimental setup that was required for observing resource utilization in scalability analysis.

Infrastructure

We used 5 servers, two were installed with OSM and pishahang. The other two were installed with two openstack versions and the remaining was installed with kubernetes. OSM was connected to one openstack. Pishahang was connected to kubernetes and openstack. The servers had the following machine configuration:

- **OSM (server 1)**
  Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz with 8 core CPU and 64 GB memory
- **Pishahang (server 2)**
  Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz with 8 core CPU and 64 GB memory
- **Openstack (server 3)**
  Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz with 16 core CPU and 128 GB memory
- **Openstack (server 4)**
  Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz with 16 core CPU and 128 GB memory
- **Kubernetes (server 5)**
  Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz with 16 core CPU and 128 GB memory

need to verify

Experiment script

We used a python script to instantiate 180 requests of a cirros image continuously at 30 RPM with the help of python-mano-wrappers. Once all the requests were instantiated, we once again used the wrappers to send the termination requests all at once. We conducted the same experiment three times i.e we had three runs of this experiment to see the variance. This experiment was conducted both on OSM and pishahang.The results of the experiment is discussed in the following sections.

6.2.2 OSM Results

The 6.5, 6.6 and 6.7 shows the CPU utilization, memory utilization and CPU utilization throught the life cycle of the experiment respectively. Let us consider the important dockers in each case and list their functionalities to realize why they have consumed maximum CPU.

CPU utilization

The 6.5 shows CPU utilization. The first 5 OSM dockers are:

- **osm_ro:** This is the Resource Orchestrator for OSM. The resource orchestrator is responsible for co-ordinating resource allocation across multiple geo-distributed VIM types. It is responsible in processing the resource-allocation requirements of the VNF as per parts of the VNFD and driving the VIM to allocate appropriate compute, network, and storage resources for the deployment of VNFs with their interconnection.
- **osm_lcm:** This is the Life Cycle Management module for OSM. This docker is responsible for set of operations related to the life cycle of a VNF and NS
  - *NS LCM operations:* 1) On-board Network Service 2) Instantiate Network Service 3) Scale Network Service 4) Update Network Service by supporting Network Service configuration changes Create, delete, query, and update of VNFFGs associated to a Network Service. 5) Terminate Network Services
  - *VNF LCM operations:* 1) Instantiate VNF (create a VNF using the VNF on-boarding artefacts) 2) Scale VNF (increase or reduce the capacity of the VNF). 3) Update and/or Upgrade VNF (support VNF software and/or configuration changes of various complexity). 4) Terminate VNF (release VNF-associated NFVI resources and return it to NFVI resource pool)
- **osm_mon:** This is the monitoring module for OSM. The main task of this docker is to retreive metrics from VIM. It keeps polling VIM every 30 seconds.
- **osm_ro_db:** This is called RO database module. RO module in OSM maintains its own database called ro-db. (RO module is usually edited to enable or disable this particular module) This module takes care of database related operations. According to the osm/ro.git, ro-db stores different IDs like vnfd id ,osm id, tenant id, vim id etc. There is something called scenario. ro db stores these scenarios and scenario ID and then a scenario gets executed when the scenario id matches with the right tenant id, osm id and vnfd id. It stores vim actions, wim actions and they are processed sequentially.
- **osm_nbi:** North Bound Interface of OSM. Restful server that follows ETSI SOL005 interface. It is the unique entry point for all the interactions with the OSS/UI system. This serves as the interface for MANO operations. (OSM's NBI offers all the necessary abstractions to allow clients for the complete control, operation and supervision od the NS.)

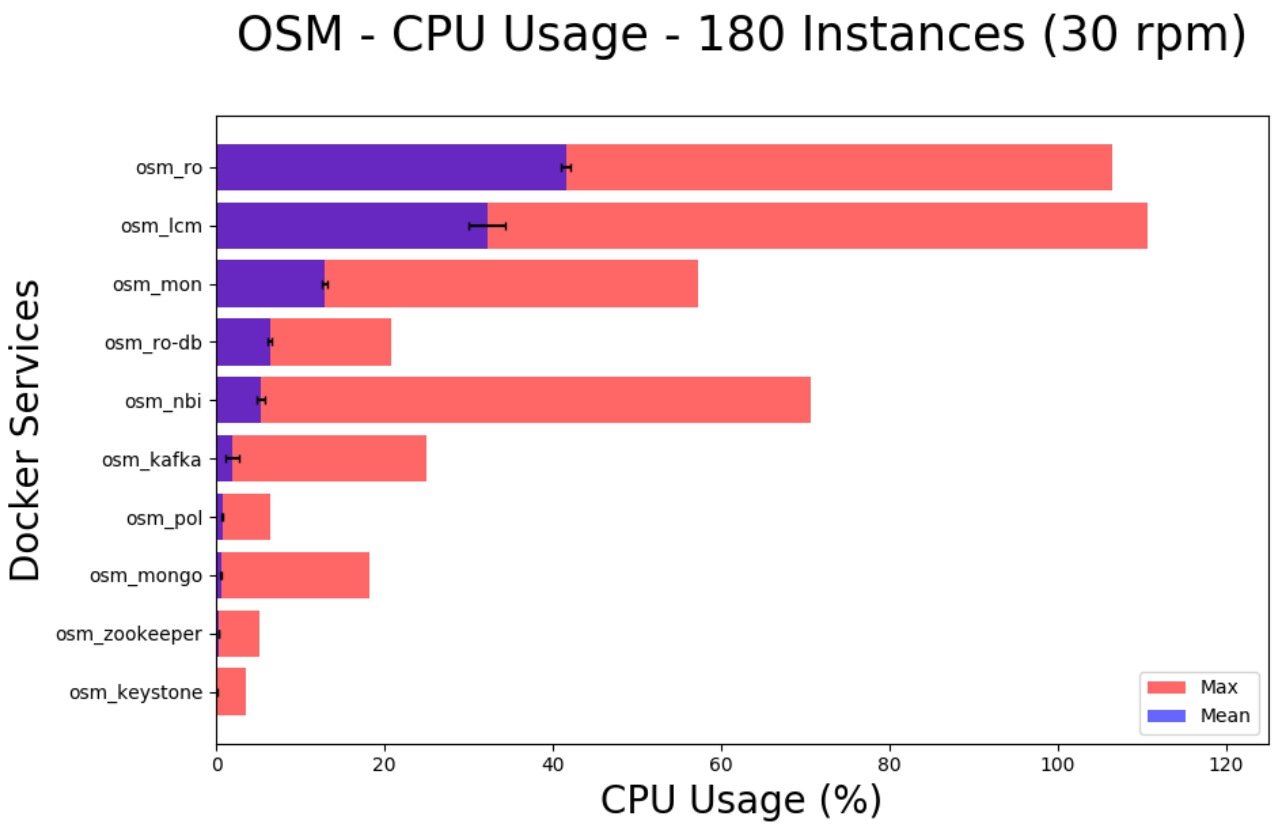

Figure 6.5: OSM CPU

Memory Utilization

The first 5 OSM dockers in memory utilization graph are:

- **osm_mongo:** This is a common non relational database for OSM modules.
- **osm_kafka:** This module provides a Kafka bus used for OSM communication.
- **osm_nbi:** This is the north bound interface for OSM module. The functionalities of this module remains the same(as explained in the previous subsection)
- **osm_light-ui:** This docker is an implementation of a web GUI to interact with the North-bound API. (The framework allows editing, validating, visualizing the descriptors of services and components both textually and graphically)
- **osm_ro_db:** This is the RO database. The functionalities of this module remains the same (as explained in the previous subsection)

Lifecycle

We now have the life cycle graphs of the entire experiment. This shows the distribution of the CPU usage among the top 3 dockers throughout the experiment. The experiment lasted for about 10 minutes.

We can observe that the ro module of OSM got continuous requests over the time and at the end all the termination requests came at once. Also LCM , mon modules continously processed the requests.

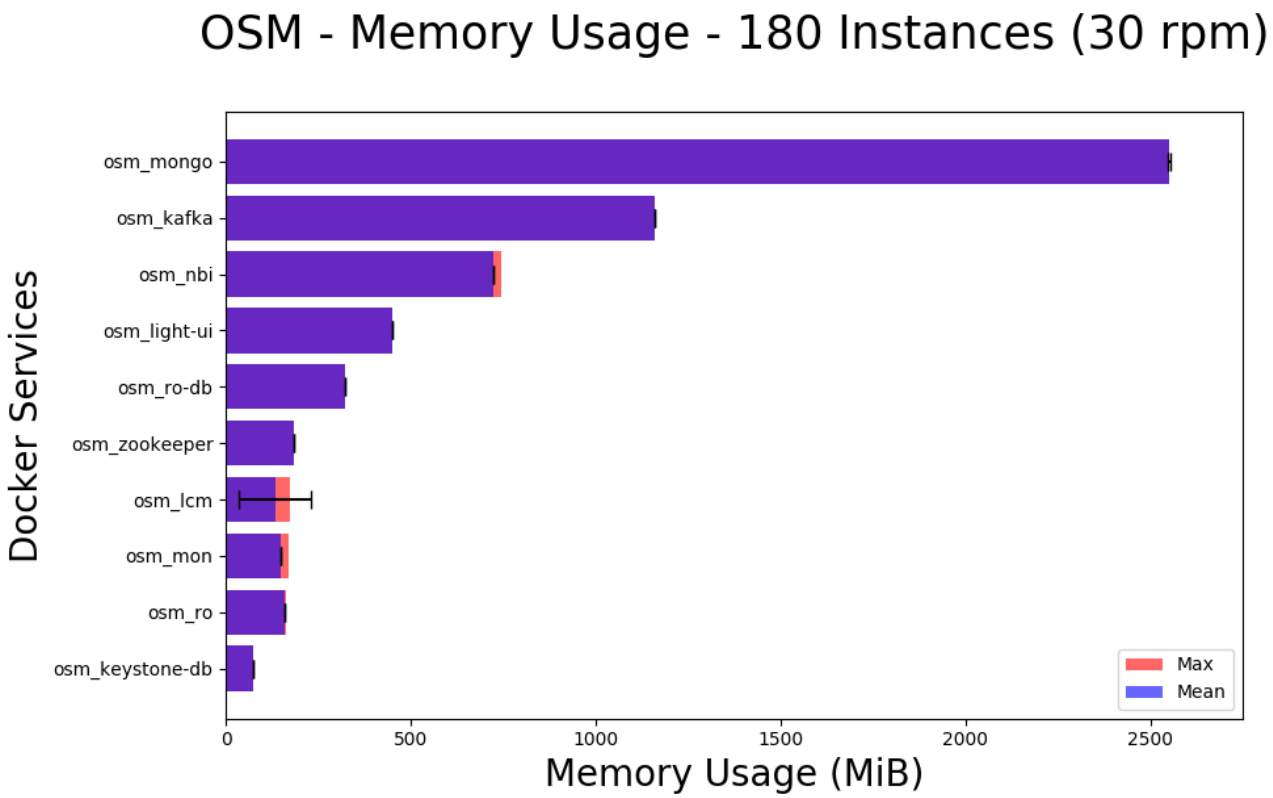

Figure 6.6: OSM MEM

mon module retrieved metric information continuously and the same trend remains throughout the experiment.

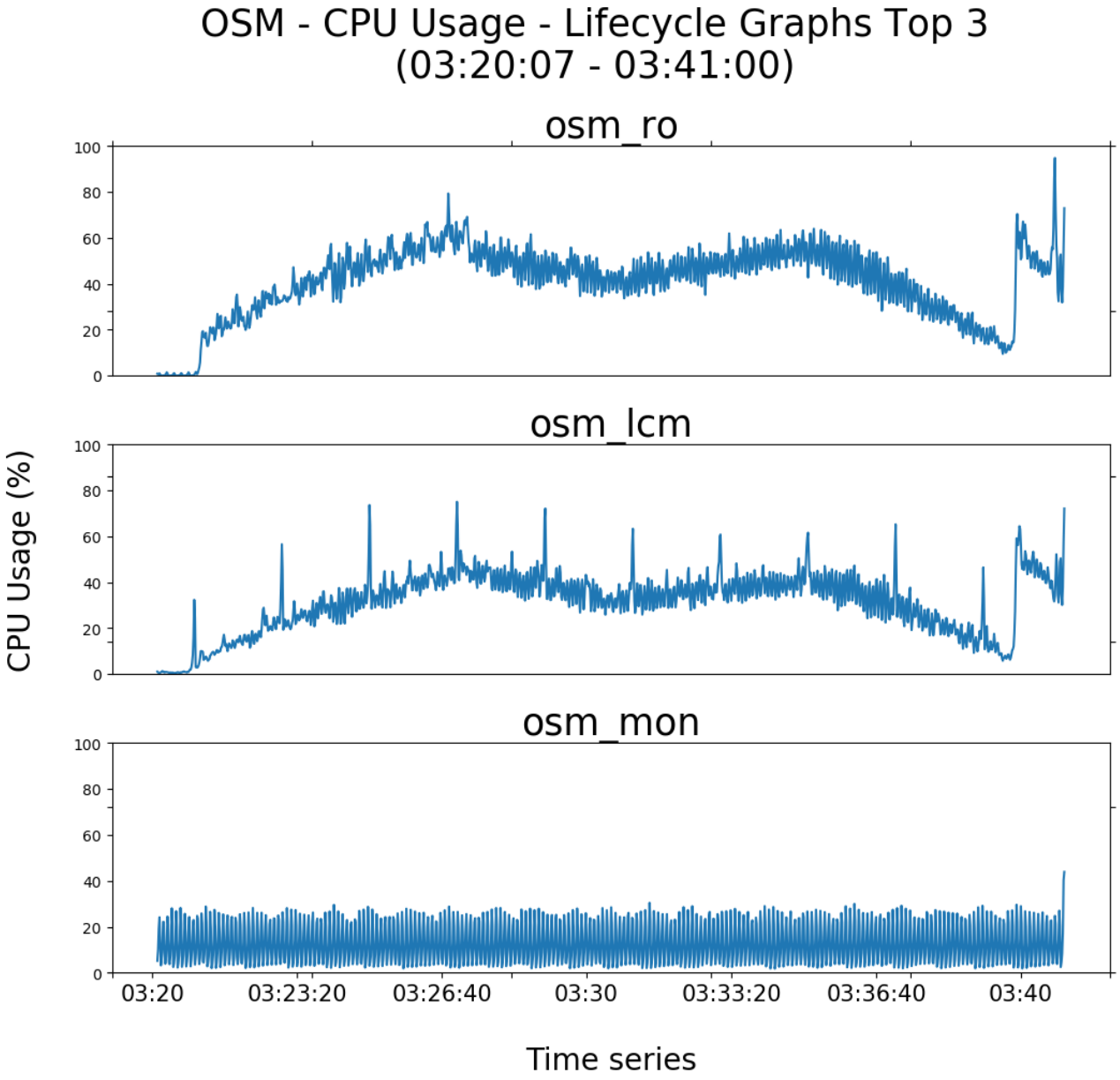

Figure 6.7: OSM LS

6.2.3 Pishahang Results

The figures 6.8 6.9 and 6.10 shows the CPU utilization, memory utilization and CPU utilization through out the life cycle of Pishahang dockers during the experiment respectively. Let us consider the important dockers in each case and list their functionalities to realize why they have consumed maximum CPU.

CPU

The figure 6.5 shows CPU utilization. The first 6 Pishahang dockers are:

- **son-sp-infrabstract:** This docker plays the role of an abstraction layer between the MANO framework and the underlying virtualised infrastructure. It exposes the interfaces to manage services and the VNF instances of all of these 180 instances by reserving resources for their service deployment. It also receives monitoring information about the infrastructure status. Hence, it occupies much of the CPU.
- **sevicelifecyclemanagement:** This is responsible for orchestrating the entire lifecycle of every service that is being deployed with Pishahang service platform. The lifecycle operations of NS include
  - *NS operations:* 1) On-board Network Service 2) Instantiate Network Service 3) Scale Network Service 4) Update Network Service by supporting Network Service configuration changes Create, delete, query, and update of VNFFGs associated to a Network Service. 5) Terminate Network Services
- **son-broker:** The Pishahang service platform consists of micro services that use a message broker to communicate, building a flexible orchestration system.

  The load on all of the other microservices in Pishahang service platform is distributed and the difference between their mean usage is negligible. Hence, the order of all other microservices can be insignificant.
- **specificmanageregistry:** The role of this docker is to manage lifecycle of FSM (function-specific manager) and SSM(service-specific manager). Lifecycle operations of FSM and SSM include instantiation, registeration, updation and termination. ex: to obtain onboarding SSM request from SLM.
- **cloudservicelifecyclemanagement:**This docker is responsible for lifecycle management of cloud network services on kubernates.
- **functionallifecyclemanagement:** This docker manages the lifecycle events of each VNF in these 180 network instances.
  - *VNF operations:* 1) Instantiate VNF (create a VNF using the VNF on-boarding artefacts) 2) Scale VNF (increase or reduce the capacity of the VNF). 3) Update and/or Upgrade VNF (support VNF software and/or configuration changes of various complexity). 4) Terminate VNF (release VNF-associated NFVI resources and return it to NFVI resource pool)

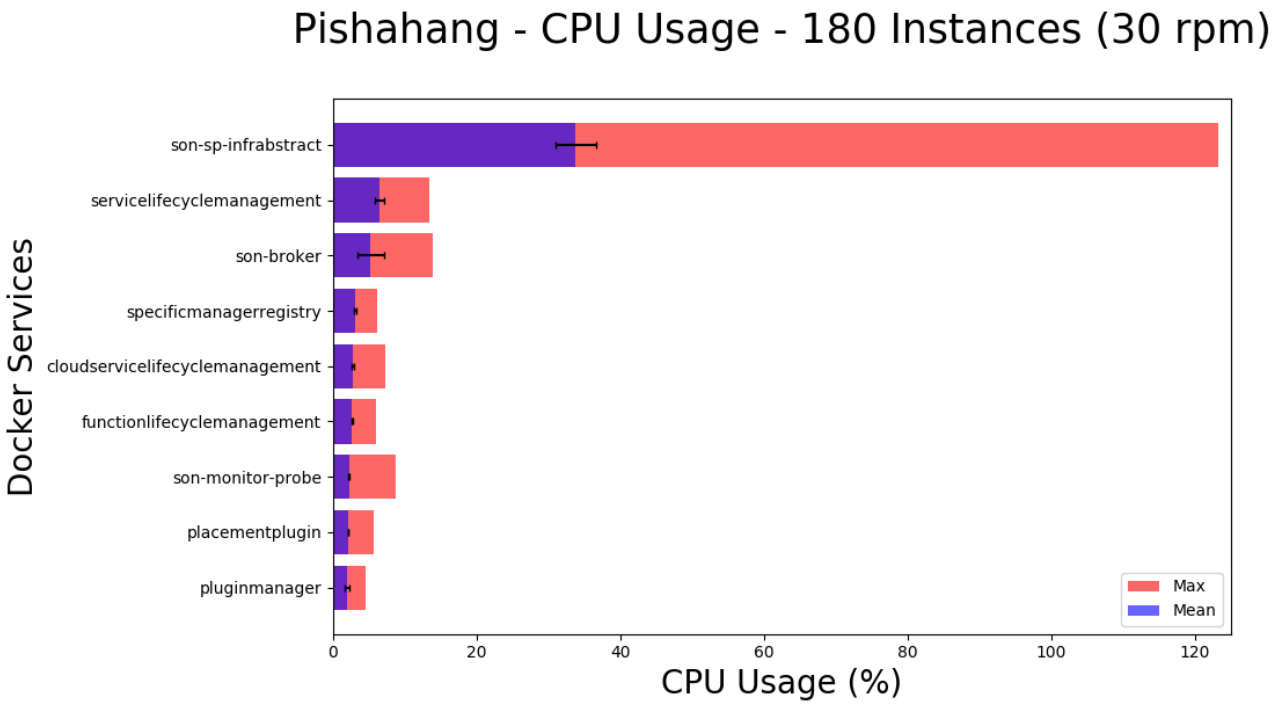

Figure 6.8: Pishahang CPU

Memory Utilization

The first 5 Pishahang dockers in memory utilization graph from figure 6.9 are:

- **son-keyclock:** This docker service provides access management and identity management of these micro services.
- **son-monitor-influxdb:** This monitoring plugin records metrics on the internal runtime and service performance and writes it to database.
- **son-sp-infrabstract:** This docker plays the role of an abstraction layer between the MANO framework and the underlying virtualised infrastructure. It exposes the interfaces to manage services and the VNF instances of all of these 180 instances by reserving resources for their service deployment. It also receives monitoring information about the infrastructure status. Hence, it occupies much of the memory.
- **WIM adaptor:** The role of this microservice is to provide connectivity over the physical network. The WIM adaptor acts as a north-bound interface to the higher layers,T eg., NFVO to provide connectivity services between NFVI-POPs or to physical network functions. This also invokes the underlying NFVI network southbound interfaces, whether they are network controllers or NFs, to construct the service within the domain.
- **son-gtksrv:** This is Pishahang's gatekeeper service management micro-service.

The most important inference from 6.6 and 6.9 is that, the memory utilisation by Pishahang is much lighter than OSM for the top docker services.

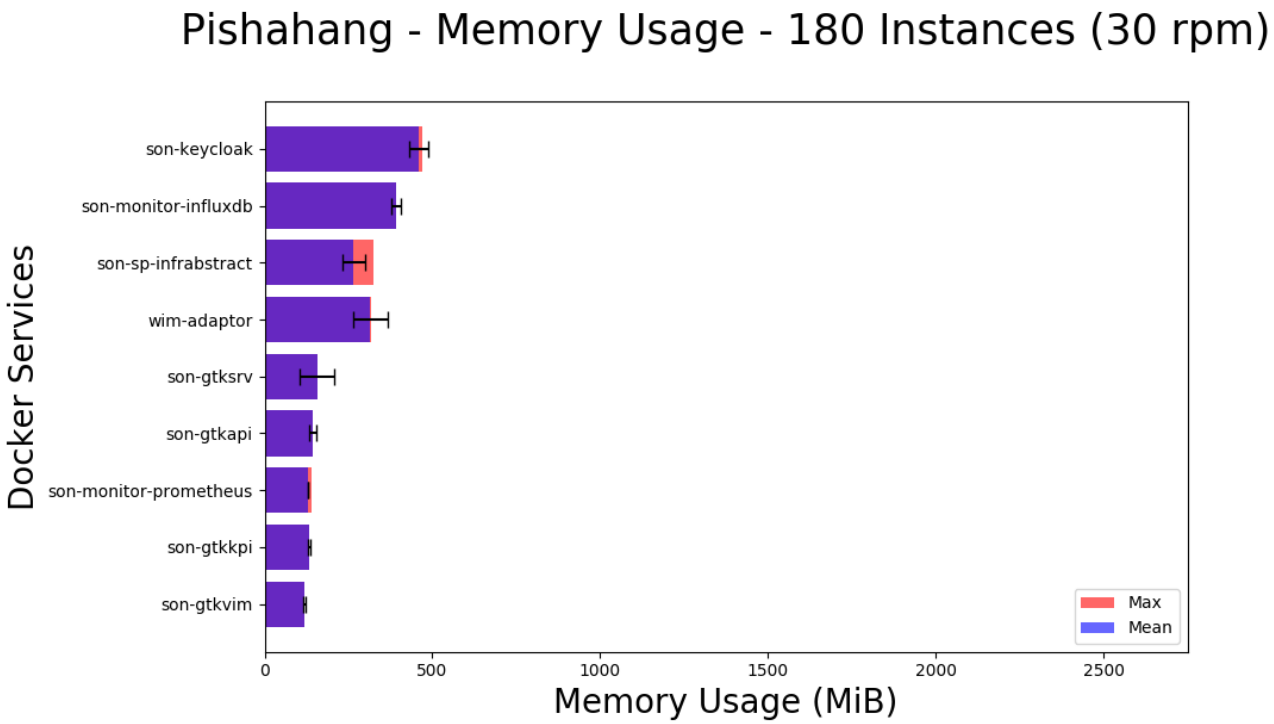

Figure 6.9: Pishahang MEM

Lifecycle

We now have the life cycle graphs of the entire experiment. The figure 6.10 shows the distribution of the CPU usage among the top 3 dockers throughout the experiment. The experiment lasted for about 10 minutes.

Initially, the metrics of docker containers are recorded for one minute, which is visualized in the figure 6.10 with almost no significant change in the CPU occupancy. The Pishahang infrastructure was reserved for all the 180 instances and with termination requests over the time, the CPU usage was reduced like before. This is observed by the graph of son-sp-infrabstract docker. The graph also depicts the CPU occupancy of other top docker services throughout the experiment.

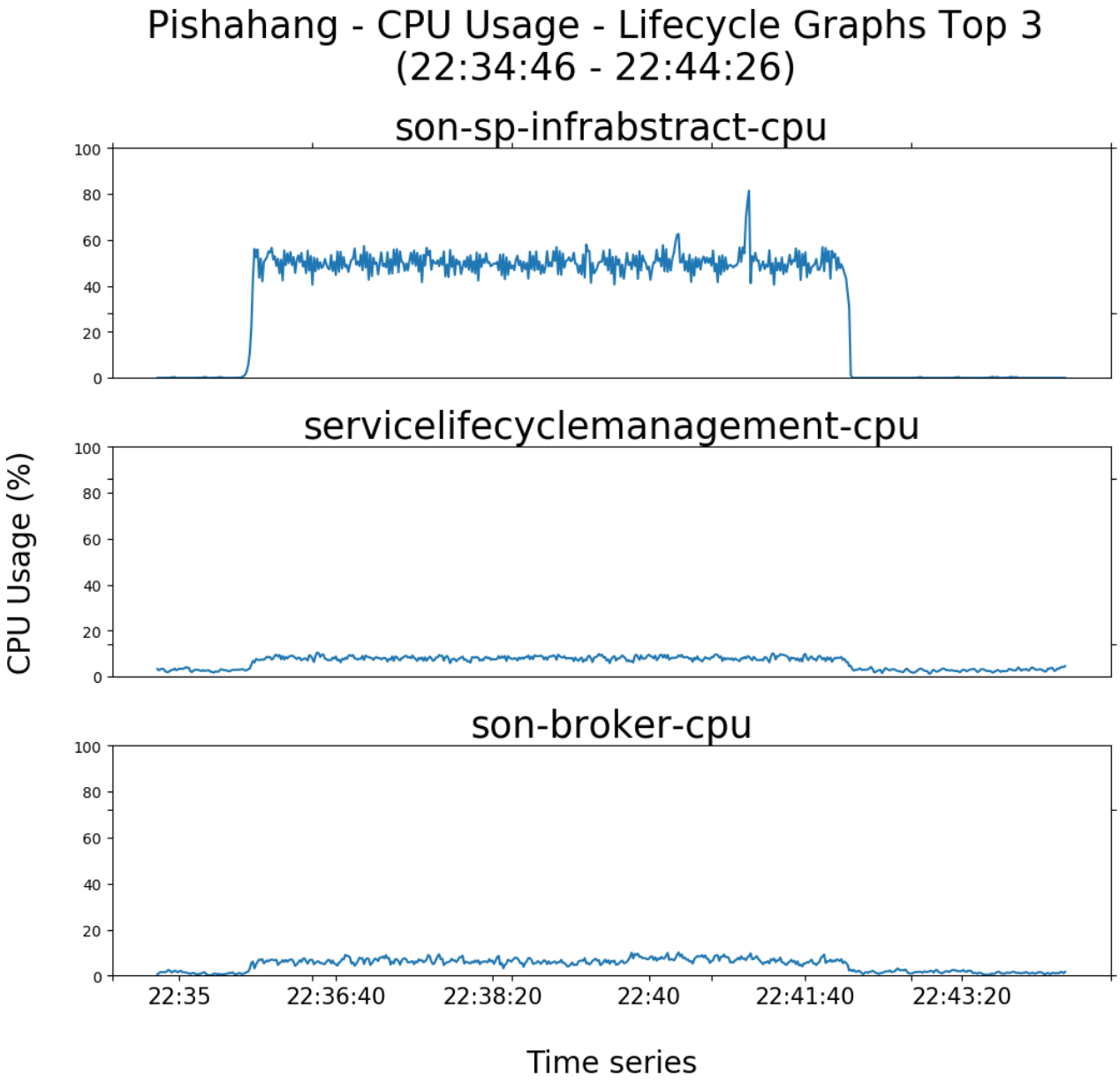

Figure 6.10: Pish LS

6.2.4 Summary of issues of the experiment

The limitations of this experiment:

- The VM in which we installed the VIM(openstack) was supposed to use a configuration with only a 16 core CPU.
- Unlike for cirros image, there were issues with successfully instantiating 90 and 180 service requests of ubuntu image despite multiple runs because ubuntu image occupied more memory than the cirros image
- Higher the RPM, lower the success rate.
- There is no VM v/s VM or container v/s container comparison i.e. this experiment conducted on OSM and pishahang could not be a fair comparison because OSM doenst not support containerized orchestration and pishahang does not have a stable support for openstack. Hence, we have used openstack for OSM and kubernetes for pishahang.

6.3 MANO Benchmarking Framework

6.3.1 Introduction

MANO Benchmarking Framework (MBF) is a result of a small script that was used to run the experiments discussed in the previous sections. The idea of MBF is to provide MANO developers with a generic framework for running experiments on MANO. MBF mainly provides the following 1) Easy interfacing with MANO instances by using python-mano-wrappers, 2) Ability to run experiments with different service descriptors, 3) Collection of performance metrics in convenient data format and 4) Flexible graphing mechanism of the collected data.

6.3.2 Design

MBF is designed for ease of use and low barrier to entry for developers. We explain the choice of tools that are used in MBF in the following list.

- **Netdata**[1] is the metrics monitoring system for MBF. Netdata captures relevant system metrics and provide powerful APIs to query the recorded data in a suitable format.
- **Python** as the choice of scripting language was obvious as the MANOs itself are implemented in python.
- **python-mano-wrappers** is used to provide access to REST APIs of MANOs from python.
- **Docker** is used to containerize MBF, thus making it easy to distribute and portable.
- **Matplotlib** is the graphing library for MBF due to its flexibility and ease of use.
- **Flask** as a python server that can be used to provide additional interactions with the experiment runner.

6.3.3 Parameters and KPIs

Parameters

MBF has experiment parameters that can be altered. The following paramenters are supported

- **Descriptors** NSDs and VNFDs can be changed. a list of NSDs/VNFDs is also supported. When a list of descriptors are provided, the experiment will be run for each of the descriptors.
- **Number of instances** Total number of instantiation requests to be sent to the MANO.
- **Number of runs** number of re-runs of the same experiment to be run. This is performed to measure the variance in results.
- **Requests per minute (RPM)** The rate at which the instantiation requests are sent to the MANO
- **Observation Time** The observation time after the instantiation requests are sent. This can be used to collect metrics post instantiation to observe how MANO behaves in the monitoring phase.
- **Inter-experiment Delay** is the time between experiment runs. This is altered to give enough time for the VIM to terminate and cleanup instances from the previous experiment runs if any.

[1] https://github.com/netdata/netdata

- **Skip experiment on error** if set to true, the current run is skipped due to a failed instance on the VIM.

6.3.4 Key Performance Indicators

MBF stores resource utilization metrics during the experiment and generates graphs to visualize the results. However, these are only examples and the further possibilities are supported by the framework. The metrics are stored as CSV files.

- **CPU** Overall system CPU usage is recorded as well as the individual docker micro service CPU usage metrics are stored.
- **Memory** Overall system memory usage along with the individual docker micro service memory usage is stored.
- **System Load** The 1m, 5m and 15m moving averages of system load values provided by the linux kernel is stored.
- **Status Tracking** The status of all instances are stored by polling the VIM every 5 seconds over the experiment lifetime. This enables to track the status change over time.
- **End-to-end Deployment Time** is the time elapsed to deploy all the instances on the VIM.
- **Individual Deployment Time** is the time taken by each instance for deployment. This is also split into time taken by MANO and VIM.

6.3.5 Steps for experiment run

The steps to run a basic experiment is detailed in this section. The following instruction is to run an experiment with 90 instances on OSM using an network service with 1 VNF.

1. Git clone the experiments-branch[2]
2. Build and start the docker container
3. Change experiment variables in the relevant scripts for respective MANOs
   - **OSM** – *run-experiment-osm.py*
   - **Sonata (Pishahang)** – *run-experiment-sonata.py*
   - **Sonata Container Orchestration (Pishahang)** – *run-experiment-sonata-k8*
4. Run the relevant script from inside the container. The experiment will now run and stores result files in the same directory
5. Use the result parser from the `experiments/results/csv-result-parser.py` to parse the results and store it in a format suitable for graphing
6. Use the graph plotter on the parsed files to generate the graphs `experiments/results/plot-graphs.py`

The commands required to run these are listed in the readme file here,
`https://github.com/CN-UPB/MANO-Benchmarking-Framework/blob/master/README.md`

Note: *This process is being streamlined to reduce some redundant steps and will be released in the next version of MBF.*

[2]https://github.com/CN-UPB/MANO-Benchmarking-Framework

6.3.6 Example Use Cases

In this section we demonstrate a few use cases of MBF. The framework facilitated easy experimentation, collection and analysis of metrics in the following cases. However, in-depth analysis of what the metrics and graphs mean are out of scope of this document.

Comparison of different network services

We compared the performance metrics of different NSDs/VNFDs and visualized it. For this, we provided 6 different network service descriptors to the experiment runner. First 3 NS consisted of a cirros image as a VNF with 1, 3 and 5 VNFs per NSD and the other 3 NS consisted of an ubuntu image as a VNF with 1, 3 and 5 VNFs per NSD.

The experiment stores the KPIs listed in the previous section, along with the graphs visualizing the differences. This graph is produced for each micro-service showing the resource utilization of each NS under different number of instantiations. As an example, the figures 6.11 and 6.12 show the CPU utilization of the OSM microservice LCM and RO respectively.

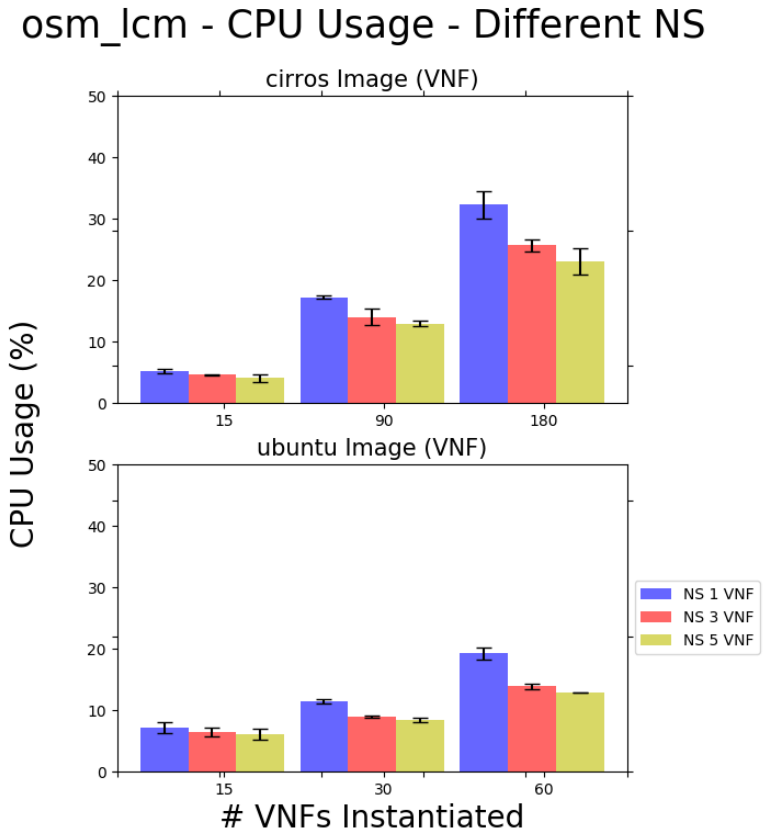

Figure 6.11: CPU usage of OSM microservice LCM

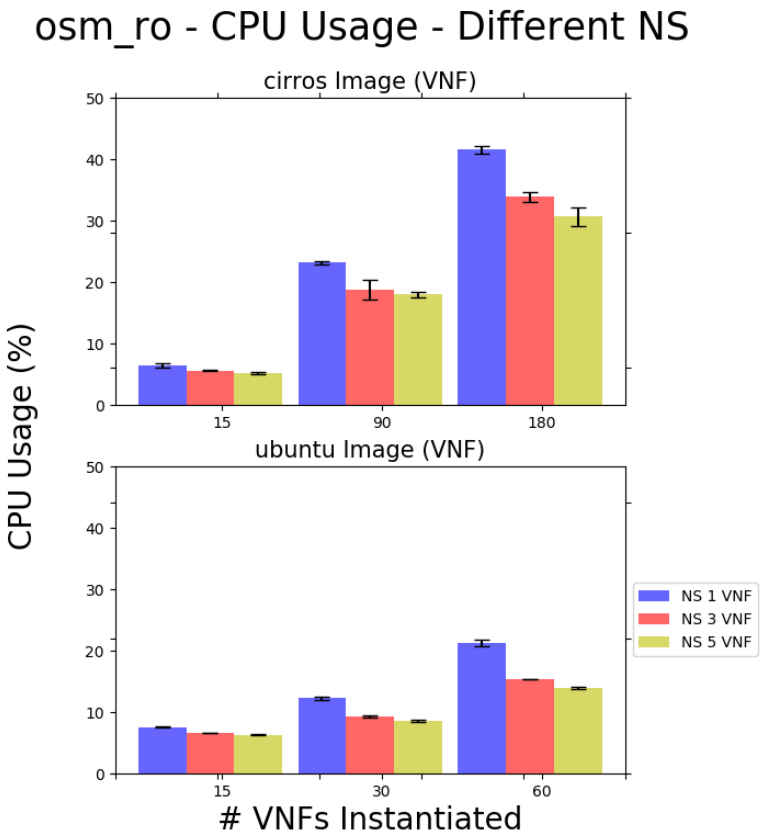

Figure 6.12: CPU usage of OSM microservice RO

Container vs VM Orchestration

Pishahang supports container orchestration on kubernetes and OSM supports VM orchestration on OpenStack. We compared the performance of orchestrating similar network services on VM and containers. In the figure 6.13, the graph on top shows the average time distribution between MANO and VIM for deploying one network service with one VNF. The bottom graph in the figure 6.13 shows the total time taken to deploy 90 instances of a network with one VNF. The time taken to deploy similar VNFs is significantly less for containers, which is expected as containers are light weight compared to a VMs.

Note: Pishahang support for VM orchestration is not stable, hence, we could not perform similar VM experiments on Pishahang for this comparison.

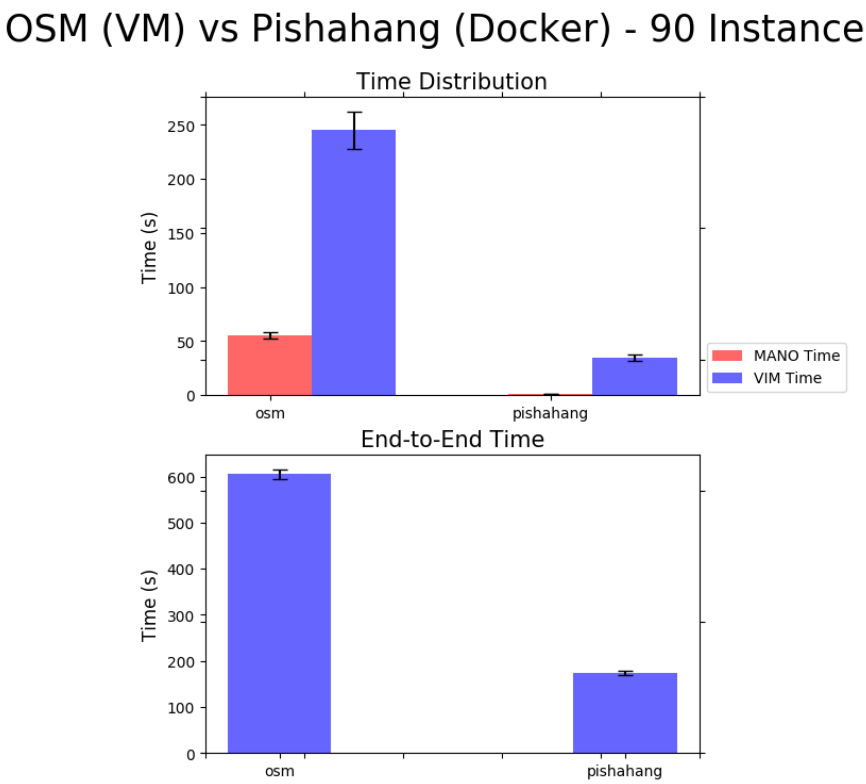

Figure 6.13: Time distribution in MANO and VIM

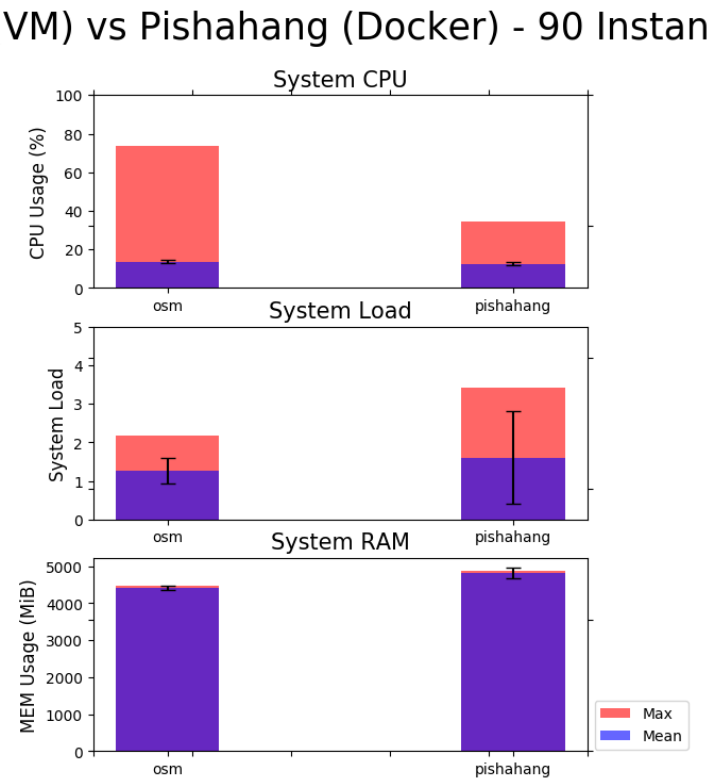

Figure 6.14: System resource utilization

Scaling Plugin Evaluation

We utilized MBF to evaluate our Scaling Plugin discussed in the section 6.1. The Parent Pishahang instance is running the scaling plugin in debug mode, which can be used to mock system load values. We added a REST call to the MBF flask server to mock the system load values in the scaling plugin.

The experiment was designed to instantiate 90 instances of cirros docker containers on Pishahang. After sending 30 instantiation requests, the experiment script alters the *5m* system load to greater than 0.7, which triggers Rule 1 from the SLP rule set 6.1. Thus, creating a new child Pishahang instance. However, in the debug mode, for the scope of this experiment, we had a child instance already instantiated from start.

After instantiating 45 instances, the script mocks the *15m* system load value to greater than 0.7, which triggers Rule 2 from the SLP rule set 6.1. Thus, parent instance will now redirect any further service requests (i.e., 46th instantiation requests on-wards) to the child Pishahang instance.

We use the overall lifecycle data collected by MBF to visualize the CPU usage over time of the top 3 microservices (based on CPU usage).

The experiments ran from 11:17:02 to 11:28:23. Initially the microservices on the parent instance took all the load, as it can be seen from the figure 6.15. At about the half way mark, the requests are redirected to the child instance. The load distributed to the child instance can be seen in the figure 6.16.

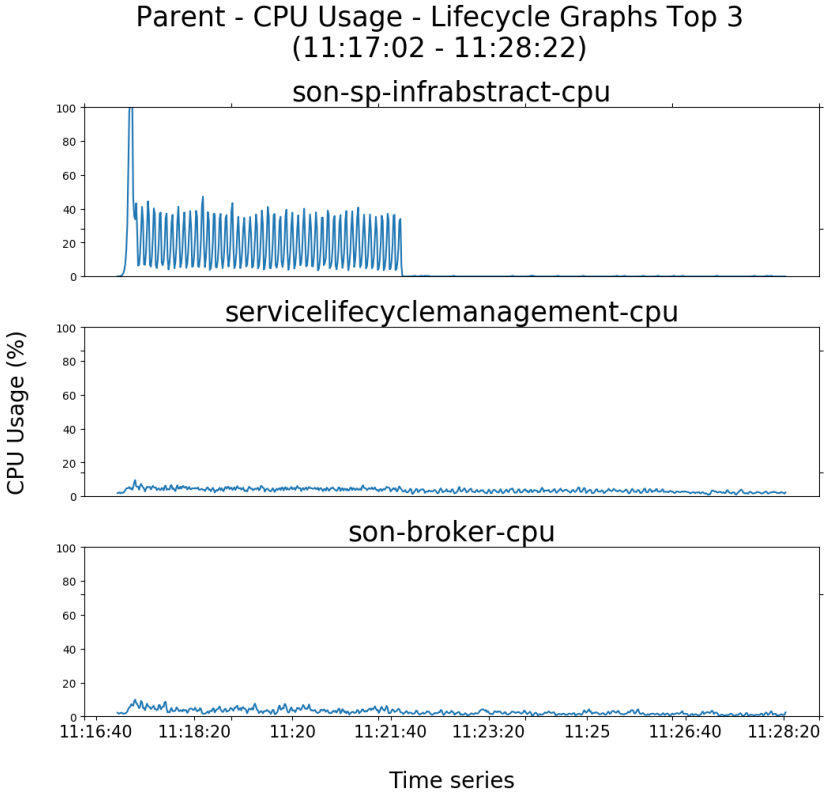

Figure 6.15: Parent MANO instance lifecycle graph

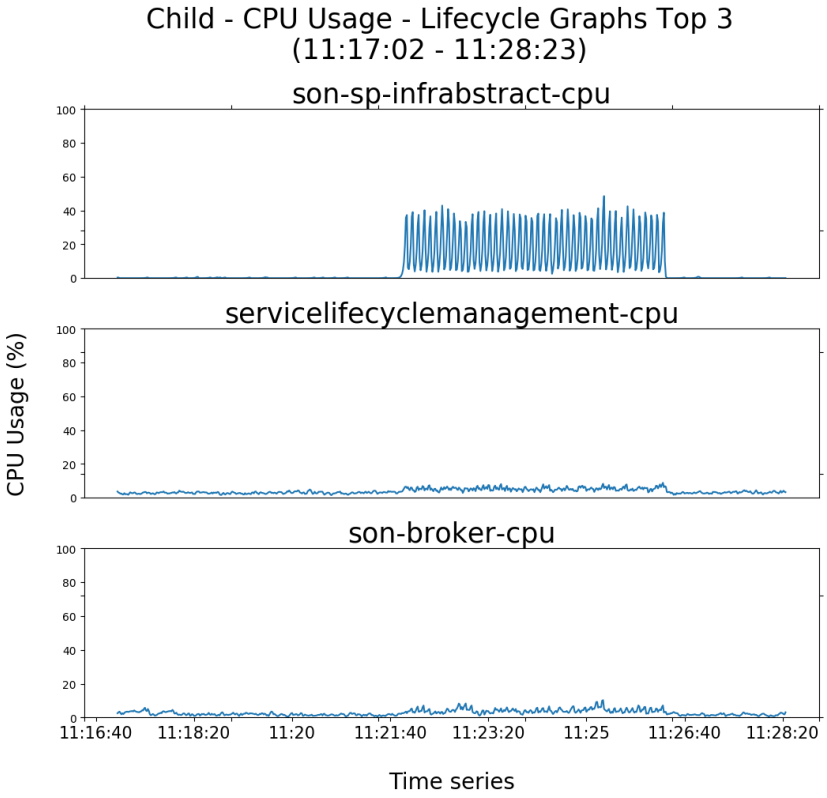

Figure 6.16: Child MANO instance lifecycle graph

6.3.7 Future scope

TODO: scope of MBF

7 Related Work

In this chapter, the relevant research efforts that can be used to achieve the goals are discussed. Firstly, the standards and specifications for orchestration and management of NFV in the section are discussed 7.1. The fundamental aspect of a service deployment is the NSD, in the section 7.2 the trends and options of NSDs and research papers that try to mitigate the interoperability challenges between different MANO frameworks are discussed. Section 7.3 will be a brief account of the MANO scalability problem.

As this is the initial project proposal, the state-of-the-art could change progressively and the approach will be updated accordingly.

7.1 Standards and Specifications

SDN decouples network control from forwarding with programmable ability. With the decoupling and programmability, SDN brings many benefits such as efficient configuration, improved performance, higher flexibility [XWF+15]. ETSI NFV [NFV2] architecture virtualizes network functions and enables dynamic and flexible selection of service functions. In ETSI NFV architecture, Network Function Forwarding Graph (VNF-FG), which consists of multiple network functions, is defined to describe network service. Internet Engineering Task Force (IETF) Service Function Chaining (SFC) working group also proposes the SFC architecture in RFC 7665 [HP15]. An SFC defines an ordered set of network Service Functions (SFs) for delivery of end-to-end services. Reference [QE16] designs a protocol named Network Service Header (NSH) to decouple the service from topology. An intelligent control plane is proposed to construct service function chains but does not consider the multi-domain situation [B+16].

ETSI NFV designs a basic frame for NFV-MANO. It defines VIM, VNF Manager (VNFM) and NFV Orchestrator (NFVO) for management and orchestration of Network Functions Virtualization Infrastructure (NFVI),VNF and network services [ETS14].

7.2 Network Service Description and Interoperability

The description of the network service plays an important role in integration and interoperability of different MANO frameworks. According to ETSI, network service is the "composition of network functions and defined by its functional and behavioral specification." Following this approach, a network service can be defined as a set of VNFs and/or Physical Network Functions (PNFs), with virtual links (VLs) interconnecting them and one or more virtualized network function forwarding graphs(VNFFGs)

describing the topology of the network service.

Garay et al. [GMUJ] emphasize on NSD, required to allow the different components to interoperate by comparing the NSD templates by OpenStack (HOT [1]) and OASIS (TOSCA[2]). A strawman model is proposed in the paper to address the upcoming interoperating challenges. The aim is to build a mechanism to translate the NSDs in order to facilitate the interoperability between different MANO frameworks.

7.3 Scalability and Hierarchical Orchestration

MANO framework faces significant scalability challenges in large-scale deployments. The amount of infrastructure a single instance of MANO framework can manage is limited. Network service scaling with NFV is discussed in paper [AHOLA+18]. It also shows different procedures that the Network Function Virtualization Orchestrator (NFVO) may trigger to scale a network service according to ETSI specifications and how NFVO might automate them. Abu-Lebdeh et al. [ALNGT] explores the effects of placement of MANO on the system performance, scalability and conclude by suggesting hierarchical orchestration architecture to optimize them. They formally define the scalability problem as an integer linear programming and propose a two-step placement algorithm. A horizontal-based multi-domain orchestration framework for Md-SFC(Multi-domain Service Function Chain) in SDN/NFV-enabled satellite and terrestrial networks is proposed in [LZF+]. The authors here address the hierarchical challenges with a distributed approach to calculate the shortest end-to-end inter-domain path.

The main intention is to answer the MANO scalability challenges, by exploring the optimal number of MANO deployments in a system and optimal hierarchical level. Also, how to manage the state of such a system dynamically.

[1] Heat orchestration template (HOT) specification:
http://docs.openstack.org/heat/rocky/template_guide/hot_spec.html

[2] Topology and Orchestration Specification for Cloud Applications (2013):
http://docs:oasis-open:org/tosca/TOSCA/v1:0/os/TOSCA-v1.0-os.html

8 Conclusion

conclusion todo

Bibliography

[AHOLA+18] Oscar Adamuz-Hinojosa, Jose Ordonez-Lucena, Pablo Ameigeiras, Juan J Ramos-Munoz, Diego Lopez, and Jesus Folgueira. Automated network service scaling in nfv: Concepts, mechanisms and scaling workflow. *arXiv preprint arXiv:1804.09089*, 2018.

[ALNGT] Mohammad Abu-Lebdeh, Diala Naboulsi, Roch Glitho, and Constant Wette Tchouati. NFV orchestrator placement for geo-distributed systems.

[B+16] Mohamed Boucadair et al. Service function chaining (sfc) control plane components & requirements. *draft-ietf-sfc-control-plane-06 (work in progress)*, 2016.

[DKP+17] Sevil Dräxler, Holger Karl, Manuel Peuster, Hadi Razzaghi Kouchaksaraei, Michael Bredel, Johannes Lessmann, Thomas Soenen, Wouter Tavernier, Sharon Mendel-Brin, and George Xilouris. Sonata: Service programming and orchestration for virtualized software networks. In *Communications Workshops (ICC Workshops), 2017 IEEE International Conference on*, pages 973–978. IEEE, 2017.

[Doc] OSM Documentation. *OSM Schema Documentation.*

[dSPR+18] Nathan F Saraiva de Sousa, Danny A Lachos Perez, Raphael V Rosa, Mateus AS Santos, and Christian Esteve Rothenberg. Network service orchestration: A survey. *arXiv preprint arXiv:1803.06596*, 2018.

[Ers13] Mehmet Ersue. Etsi nfv management and orchestration-an overview. In *Proc. of 88th IETF meeting*, 2013.

[ETS14] NFVISG ETSI. Gs nfv-man 001 v1. 1.1 network function virtualisation (nfv); management and orchestration, 2014.

[GMUJ] Jokin Garay, Jon Matias, Juanjo Unzilla, and Eduardo Jacob. Service description in the NFV revolution: Trends, challenges and a way forward. 54(3):68–74.

[HP15] Joel Halpern and Carlos Pignataro. Service function chaining (sfc) architecture. Technical report, 2015.

[KDK18] Hadi Razzaghi Kouchaksaraei, Tobias Dierich, and Holger Karl. Pishahang: Joint orchestration of network function chains and distributed cloud applications. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 344–346. IEEE, 2018.

[LZF+] Guanglei Li, Huachun Zhou, Bohao Feng, Guanwen Li, and Qi Xu. Horizontal-based orchestration for multi-domain SFC in SDN/NFV-enabled satellite/terrestrial networks. 15(5):77–91.

[NFV2] GS NFV. Network functions virtualisation (nfv); architectural framework. *NFV ISG*, 2.

[QE16] Paul Quinn and Uri Elzur. Network service header. *Internet Engineering Task Force, Internet-Draft draft-ietf-sfc-nsh-10*, 2016.

[XWF+15] Wenfeng Xia, Yonggang Wen, Chuan Heng Foh, Dusit Niyato, and Haiyong Xie. A survey on software-defined networking. *IEEE Communications Surveys & Tutorials*, 17(1):27–51, 2015.