

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 30, 2019

Contents

1	Introduction and Motivation	3
1.1	Problem Description	3
2	Use Cases	4
2.0.1	Cross-MANO Framework Interaction	4
2.0.2	Hierarchical Orchestration	4
3	Scramble architecture	6
3.1	Architecture	6
4	Scramble Services Implementation	9
4.1	Translator	9
4.1.1	Architecture & Work flow	9
4.1.2	Modules	10
4.1.3	Challenges	13
4.1.4	Usage	13
4.1.5	Future Work	13
4.2	Splitter	17
4.2.1	Architecture and Work flow	17
4.2.2	Usage	23
4.2.3	Challenges	26
4.2.4	Future work	27
4.3	Adaptor	28
4.3.1	Architecture & Work flow	28
4.3.2	Installation and usage	32
4.3.3	Challenges	32
4.3.4	Future work	33
4.4	Pishahang-Scramble Integration	34
4.4.1	SLM_mapping_scramble	35
4.5	Installation of Pishahang with Scramble	38
4.6	Scramble GUI for Pishahang	38
5	MANO Scalability	42
5.1	Scalability Plugin	43
5.1.1	Architecture	43
5.1.2	Workflow	45
5.2	Experiments	49

5.2.1	Testbed	49
5.2.2	OSM Results	51
5.2.3	Pishahang Results	55
5.2.4	Limitations of the experiment	59
5.3	MANO Benchmarking Framework	60
5.3.1	Design	60
5.3.2	Parameters and KPIs	60
5.3.3	Key Performance Indicators	61
5.3.4	Steps for experiment run	61
5.3.5	Example Use Cases	62
5.4	Discussions	67
5.4.1	Understanding OSM lifecycle graphs	70
6	Related Work	73
6.1	Standards and Specifications	73
6.2	Network Service Description and Interoperability	73
6.3	Scalability and Hierarchical Orchestration	74
7	Conclusion	75
A	Appendix	76
A.1	Translator	77
A.1.1	Validation	77
A.2	Splitter	80
A.2.1	Pishahang	80
A.2.2	OSM	80
A.3	Scaling Plugin	81
A.3.1	Pishahang	81
A.3.2	OSM	82
	Bibliography	83

List of Figures

2.1	Use Case Diagram	5
3.1	High-level scramble architecture	7
3.2	Scramble Architecture	8
4.1	Translator sequence diagram	15
4.2	Translator class diagram	16
4.3	OSM Schema Class Diagram	18
4.4	OSM Splitter Class diagram	19
4.5	Pishahang Schema Class Diagram	20
4.6	Pishahang Splitter Class diagram	21
4.7	Splitter Sequence diagram	22
4.8	PWM high level architecture	29
4.9	CommonInterface Abstract Base Classes defined in PWM	30
4.10	OSM Wrappers implemented based on the CommonInterface base classes	31
4.11	Sonata and Pishahang Wrappers implemented based on the CommonInterface base classes	31
4.12	Integration sequence diagram	35
4.13	Scramble_pishahang GUI	39
4.14	Scramble_pishahang GUI	40
4.15	Scramble_pishahang GUI	40
4.16	Scramble_pishahang GUI	41
5.1	High-level SONATA MANO plugin architecture	43
5.2	Scaling plugin architecture	46
5.3	Service Lifecycle Manager with Scaling Plugin Workflow	47
5.4	Scaling Plugin Workflow	48
5.5	OSM CPU	52
5.6	OSM MEM	53
5.7	OSM LS	54
5.8	Pishahang CPU	56
5.9	Pishahang MEM	57
5.10	Pish LS	58
5.11	CPU usage of OSM microservice LCM	62
5.12	CPU usage of OSM microservice RO	63
5.13	Time distribution in MANO and VIM	64
5.14	System resource utilization	65
5.15	Parent MANO instance lifecycle graph	66

5.16	Child MANO instance lifecycle graph	66
5.17	CPU usage of Pishahang microservice son-sp-infrabstract	68
5.18	CPU usage of Pishahang microservice servicelifecyclemanagement	69
5.19	Lifecycle graph for 90 instances	71
5.20	OSM Frequency distribution of functions	72

List of Tables

4.1	Descriptor represented DataFrame.	9
4.2	Helper functions.	36
A.1	Schema Parameters of Pishahang Considered for Splitting	80
A.2	Schema Parameters of OSM Considered for Splitting	80

Introduction and 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) makes it easier to tackle these challenges by providing software based management of network functions. This software based management is possible with a central management and orchestration (MANO) framework. 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 its 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 aids hierarchical orchestration by exposing the MANO IPs. Lastly, the project aims at integrating these individual modules in order to provide an end-to-end network service delivery across different MANO frameworks.

In this chapter, the use cases are discussed.

2.0.1 Cross-MANO Framework Interaction

The MANO frameworks used by every network service provider varies from one another. Scramble with the help of translator and splitter 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. 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.

2.0.2 Hierarchical Orchestration

By using MANO adaptor, 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.

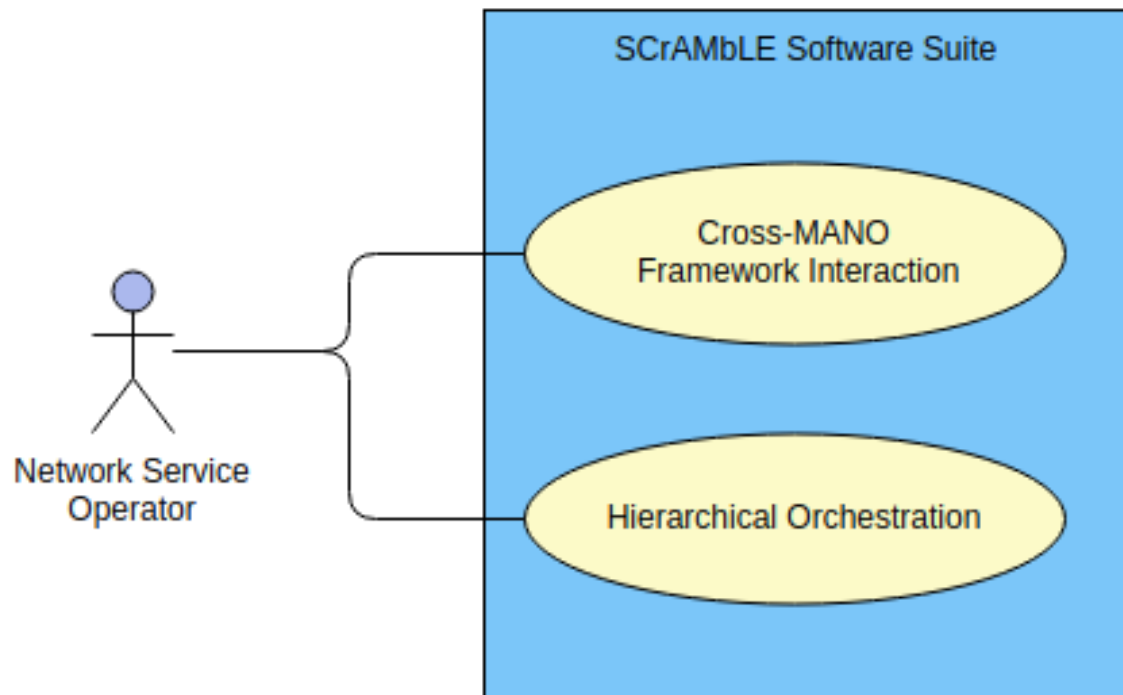


Figure 2.1: Use Case Diagram

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

Scramble architecture

Scramble is a tool to realize inter-operability between different MANO frameworks. To achieve this, scramble adds support for translation and splitting of descriptors and provides python wrappers for REST APIs for OSM, SONATA and Pishahang.

3.1 Architecture

The services of scramble resides as a plugin within Pishahang MANO framework. In the scope of this project, the plugin has been implemented in Pishahang. The plugin enables cross MANO communication.

With scramble, hierarchical orchestration is possible. In the figure 3.1, the high-level architecture of such a scenario is shown. Here, Pishahang instances are installed with the scramble plugin thus enabling it to communicate and manage child MANO instances. Service requests received by the higher level Pishahang MANO instances in the hierarchy can be redirected to lower level MANOs i.e, to either OSM or Pishahang.

Scramble is composed of three main services listed below. These are discussed in detail in the following sections.

- Translator
- Splitter
- Wrapper(adaptor)

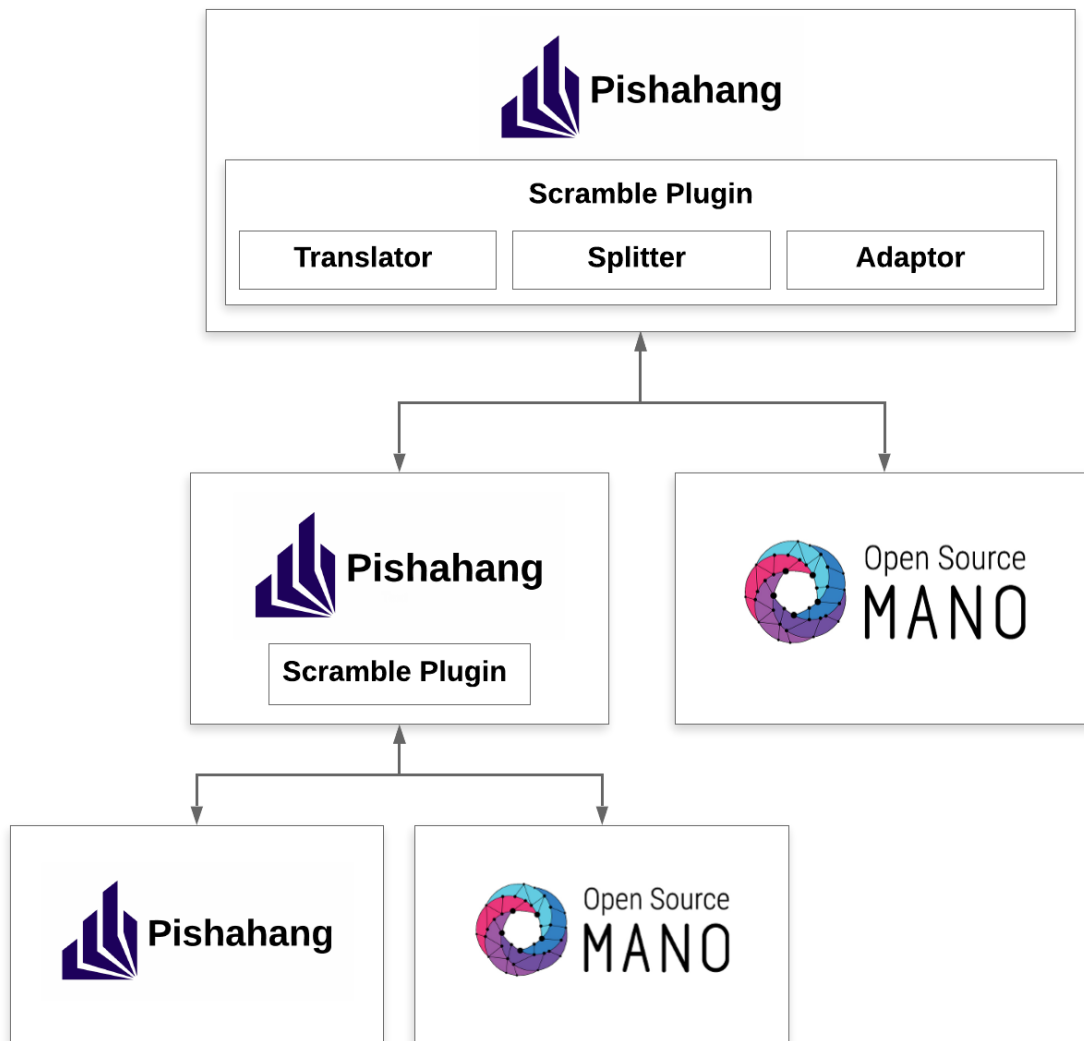


Figure 3.1: High-level scramble architecture

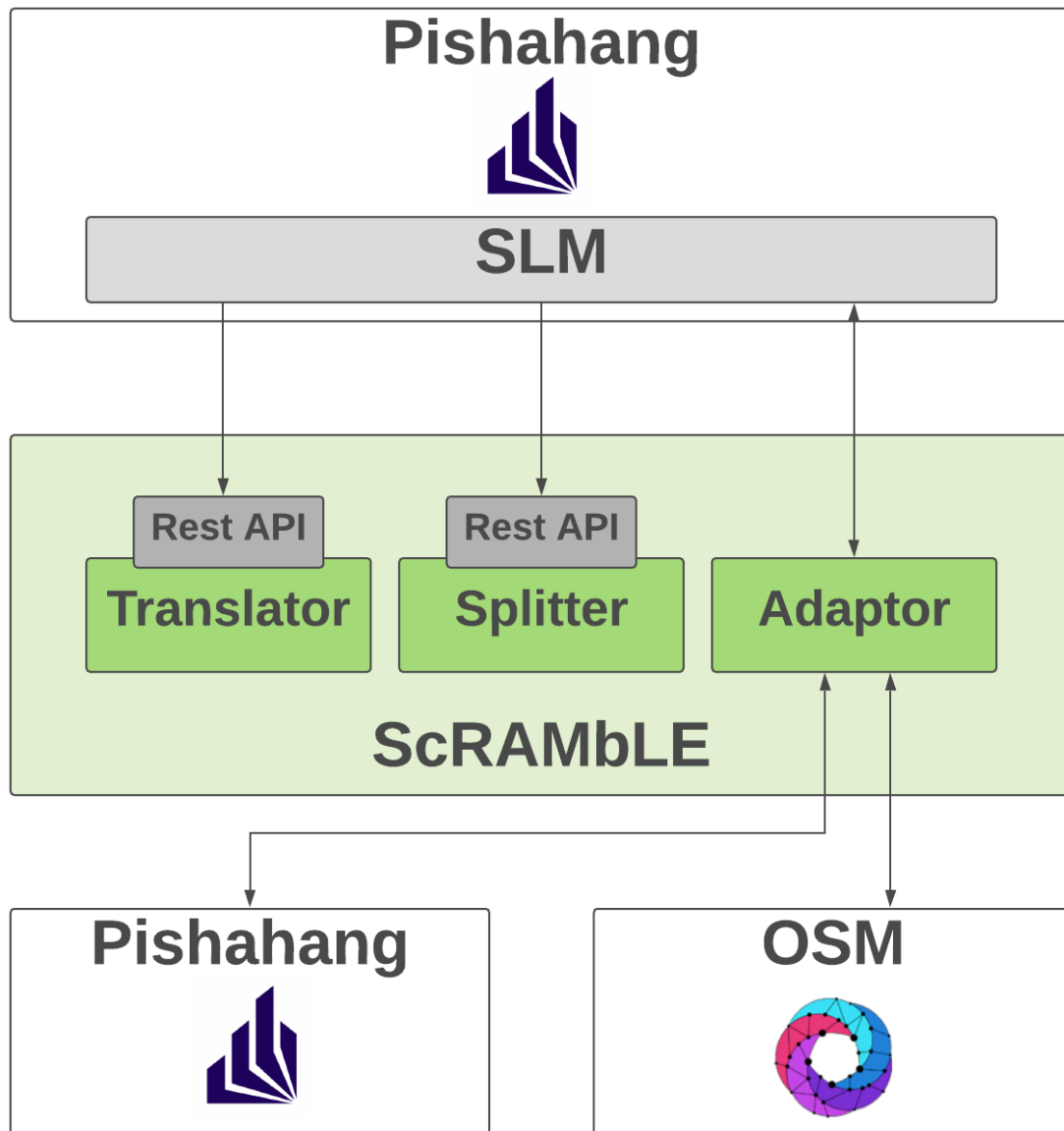


Figure 3.2: Scramble Architecture

Scramble Services Implementation

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.

4.1.1 Architecture & Work flow

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

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. It is useful to store the nested information

The *setup* and *transformation* classes of the translator engine carries out all the transformation, needed to translate between OSM and Pishahang descriptors, on the *DataFrame* object. Once all the transformation are over, the resulting *DataFrame* is then converted again into a json (refer Figure 4.1).

4.1.2 Modules

The Translator engine consists of the following modules (refer Figure 4.2):

1. descriptorReader
 - class: read_dict
2. descriptorWriter
 - class: write_dict
3. utilities
 - class: setup
 - class: transformation
 - class: insert_into_db
4. translator
 - class: TranslatorService
5. validator

descriptorReader

The class *read_dict* 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.

```

1 from descriptorReader import read_dict
2
3 pishahang = pishahang_descriptor ## the descriptor as a json or dict object
4
5 ### reading a dict/ json content into a pandas dataframe
6 reader = read_dict()
7
8 pishahang_dataset = pd.DataFrame(
9 reader.dict_parser(pishahang, 'root', 1, '0|preroot|0'),
10 columns=['parent_level', 'parent_key', 'level', 'key', 'value', 'lineage'])
11
12
13 pishahang_dataset.sort_values(ascending=True, by=['level', 'parent_key'], inplace=True)
14 pishahang_dataset.fillna('NULL', inplace=True)
15 pishahang_dataset.reset_index(drop=True, inplace=True)

```

Listing 4.1: reader to read a json into a DataFrame

descriptorWriter

The class *write_dict* is responsible to read a python pandas DataFrame input and output a nested json/dictionary maintaining the nested structure in the dictionary.

```

1 from descriptorWriter import write_dict
2
3 ### writing from a pandas dataframe to a dict/json object
4 writer = write_dict()
5 pishahang_descriptor = writer.translate(pishahang_dataset.sort_values(by='lineage'))

```

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

utilities

The class *setup* is responsible for transforming the keys and map the corresponding values between sonata and osm descriptors. The class includes 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()`

The class *transformation* acts as a helper class for the task of transforming the dataframe between sonata and OSM structures.

translator

The class *TranslatorService* is the interface where the actual translation request comes in. After a translation 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 and vice-versa.

```

1 import pymongo
2 from validate import validator
3 from utilities import setup
4
5
6 class TranslatorService():
7
8     def __init__(self, client = pymongo.MongoClient("mongodb://mongo:27017")):
9         self.setup_obj = setup(client)
10        self.validate_obj = validator()
11
12
13    def toSonata(self, received_file):
14
15        if 'vnfd:vnfd-catalog' in received_file:
16
17            doc = self.setup_obj.db_descriptors["translated_vnfd"]
18            translated = self.setup_obj.translate_to_sonata_vnfd(received_file)
19
20            check = self.validate_obj.sonata_vnfd_validate(translated)
21
22            if check == "True":
23                temp = doc.insert_one(translated)
24                translated_ref = temp.inserted_id
25
26        elif 'nsd:nsd-catalog' in received_file:
27
28            doc = self.setup_obj.db_descriptors["translated_nsd"]
29            translated = self.setup_obj.translate_to_sonata_nsd(received_file)
30
31            check = self.validate_obj.sonata_nsd_validate(translated)
32
33            if check == "True":
34                temp = doc.insert_one(translated)
35                translated_ref = temp.inserted_id
36

```

```

37 return {"descriptor":translated , "VALIDATE STATUS" :check}
38
39 def toOsm(self ,received_file):
40
41 if 'network_functions' in received_file:
42
43 doc = self.setup_obj.db_descriptors["translated_nsd"]
44 translated = self.setup_obj.translate_to_osm_nsd(received_file)
45
46 check= self.validate_obj.osm_validator(translated)
47
48 if check == "True":
49 temp = doc.insert_one(translated)
50 translated_ref = temp.inserted_id
51
52 elif 'virtual_deployment_units' in received_file:
53
54 doc = self.setup_obj.db_descriptors["translated_vnfd"]
55 translated = self.setup_obj.translate_to_osm_vnfd(received_file)
56
57 check= self.validate_obj.osm_validator(translated)
58
59 if check == "True":
60 temp = doc.insert_one(translated)
61 translated_ref = temp.inserted_id
62
63 return {"descriptor":translated , "VALIDATE STATUS" :check}

```

Listing 4.3: Translating descriptor between Pishahang and OSM

validator

The validator validates the descriptors presented for translation. The simplest form of validation of the descriptors begins with validating the syntax of the given descriptors, implemented using python library called *jsonschema.draft4validator*, this library compares the given descriptor with corresponding schema provided and if errors are found then the error and the path of the error is printed. This error path is handy and is important, as a typical descriptor has many keys with identical names.

The function in appendix A.1.1 validates the syntax of provided and translated OSM and Sonata descriptors with its corresponding schemas.

The next stage of validation is checking the semantics of the descriptors, for Sonata this is achieved by validating the Integrity of the given descriptors.

1. Integrity Validation: For validation of integrity of a NSD the corresponding VNFD's are required, which then checks the connection points and virtual links in NSD and VNFD's. If there is a correlations between the connection points and virtual links in NSD and VNFD's then the validation holds true.
 - (a) The Function `def sonata_nsd_validate(self,descriptor, vnfd = None)`, accepts NSD and its corresponding zero or more VNFD's. If one or more VNFD's are provided the the integrity and topology validation occurs else just the syntax is checked.

For checking the semantics of the OSM descriptors, python object class which is generated from the .Yang files provided by OSM. The provided descriptors are then passed as parameters to this python object class with the help of python library called *pybindJSONDecoder.load_ietf_json()*. The resulting output will be true if successfully validated else the error in the descriptor is given.

1. In the class *def osm_validator(self,descriptor)*, the descriptor in the form of python dictionary is taken as input and syntax is checked first, later the descriptor is passed to class "osm_dep_validator(descriptor)" to check semantics.
2. in class *osm_dep_validator(descriptor_to_validate)*, *pybindJSONDecoder.load_ietf_json(data, None, None, obj=mynsd)* is used where data is the given descriptor, obj is the Python object class in file osmdata.

And thus a descriptor which is validated true is not only error free but also will be for sure accepted by Sonata, Pishahang and OSM as a valid descriptor.

4.1.3 Challenges

1. 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 work.
2. The challenging part for the development of validator was for OSM. Currently available validator for OSM, is a package developed based on python object code and the problem is that only the error is printed. For example: if a error is found in key "ID", then the error message would be "invalid ID" and a typical descriptor contains multiple "ID" keys and it would be hard to find where exactly the error lies. Hence using various Python libraries a Json schema for OSM was developed with which not just the error but the exact path of the error is also given.

4.1.4 Usage

The class *TranslatorService* takes input of two simple requirements.

1. The first input is a descriptor in the form of json file or a python dictionary. The descriptor can be Pishahang or OSM.
2. The second input is a parameter to let the translator know how the translation should take place. The parameters are
 - (a) *osm_to_sonata* for translation of OSM descriptor to Pishahang.
 - (b) *sonata_to_OSM* for translation of Pishahang descriptor to OSM.

The output will be a valid translated descriptor in the form of python dictionary.

4.1.5 Future Work

Translator engine currently does not 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 was not currently feasible during our implementation period in both OSM and Pishahang.

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 unpacking 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 not feasible in OSM during our implementation period, so we sidelined for future scope.

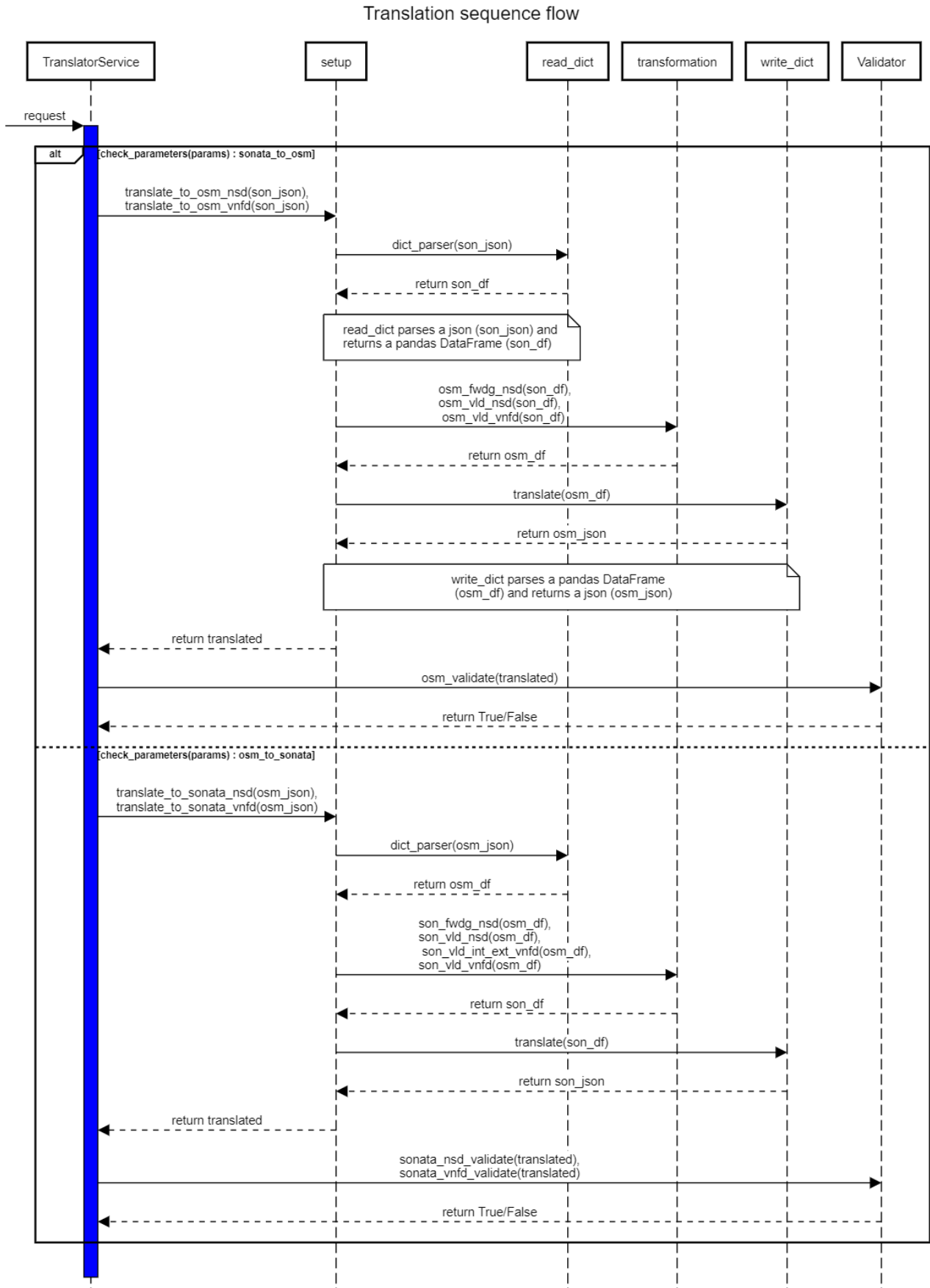


Figure 4.1: Translator sequence diagram



Figure 4.2: Translator class diagram

4.2 Splitter

Splitter helps in splitting a Network Service (NS) into multiple sub Network services which can be deployed and instantiated individually on Internet Service Providers (ISPs) located over a vast geographical region spanning multiple domains and are orchestrated by different MANO frameworks. Splitter calls Service Descriptor Translator if there is a need to translate the NS if it is to be deployed on a different MANO framework. In this work package, a Service Descriptor Splitter (SDS) is implemented which splits the NSD of a network service. SDS takes NSD as an input that contains all the information elements which can be extracted to generate separate NSDs. In the proposed approach, the service graph is extracted from the input NSD and is split into subgraphs that result in a separate NSD which includes a set of elements such as VNFs, Virtual Links (VLs), forwarding graphs of VNFs etc, according to the specific MANO framework.

4.2.1 Architecture and Work flow

Figure 4.3, 4.4, 4.5 and 4.6 represents class diagrams of OSM and Pishahang. 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. Figure 4.7 shows the sequence diagram of Pishahang splitter. 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.
- **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:** 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. The current implementation can split a NSD with three or less VNFs. To maintain the topology of the original NSD, Splitter needs to extract virtual links from the main NSD and create new virtual links for sub NSDs in such a way that the topology of the main NSD is maintained.
- **Create Sub-NSDs:** The last step is to return the sub-nsds created out of NSD objects.

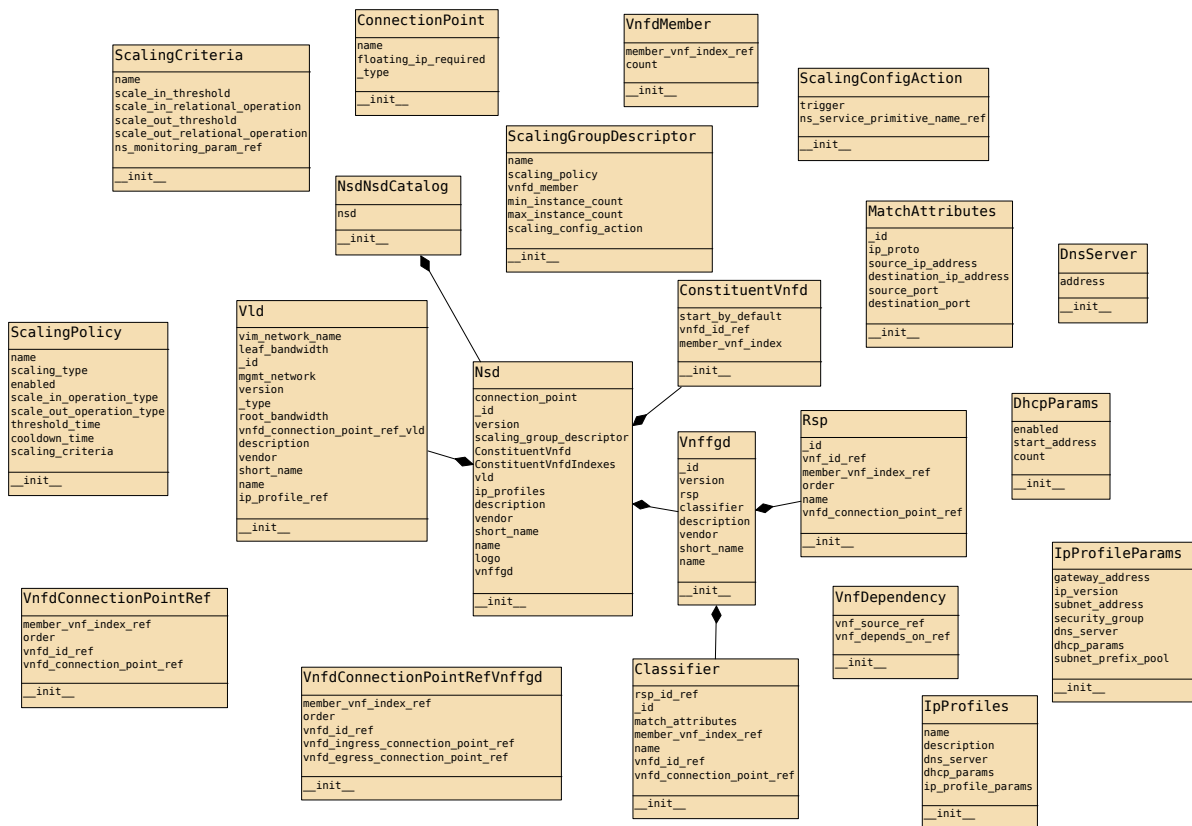


Figure 4.3: OSM Schema Class Diagram

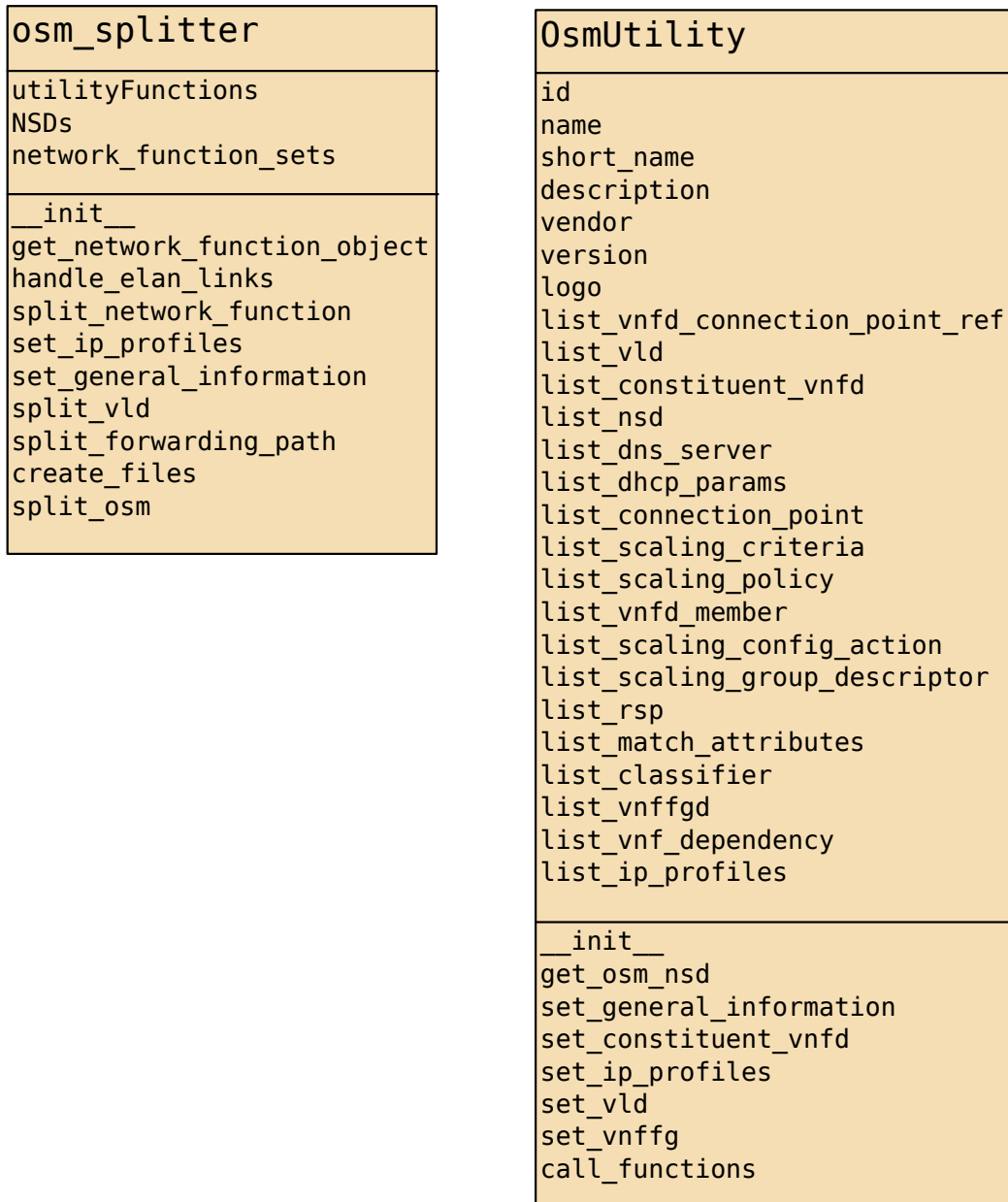


Figure 4.4: OSM Splitter Class diagram

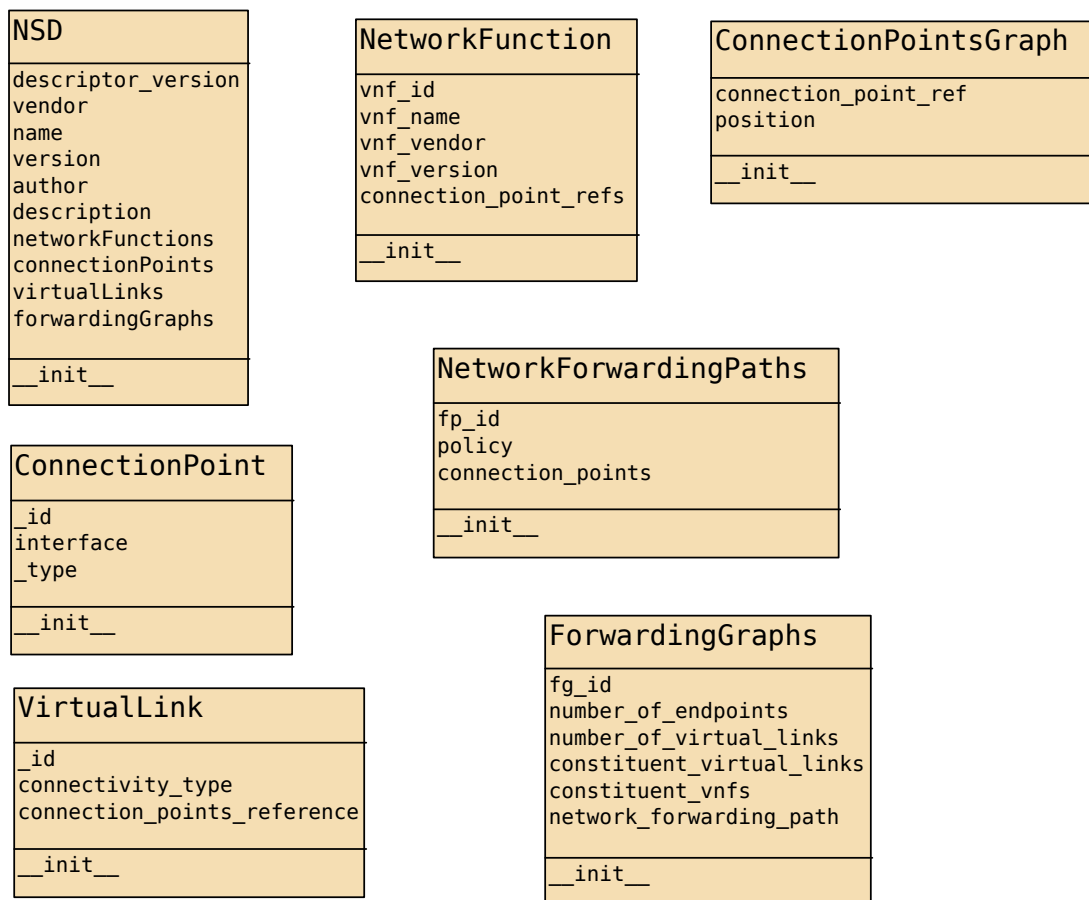


Figure 4.5: Pishahang Schema Class Diagram

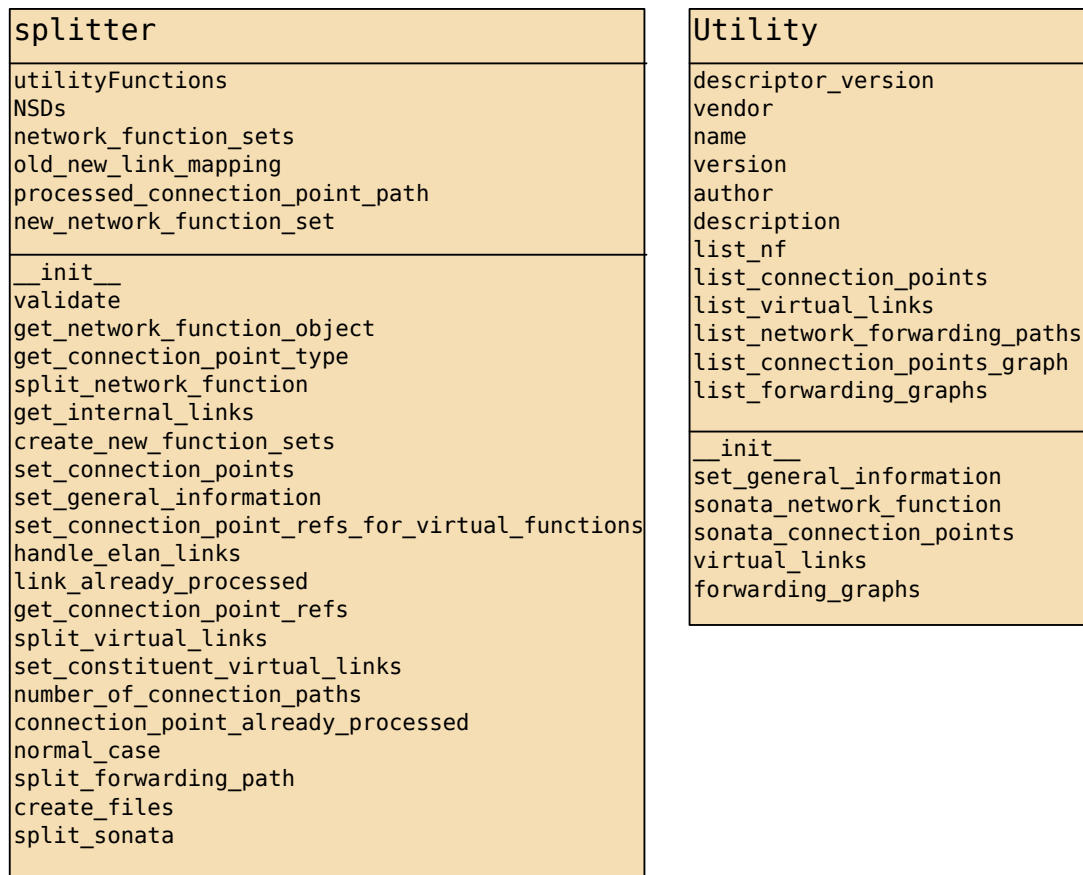


Figure 4.6: Pishahang Splitter Class diagram

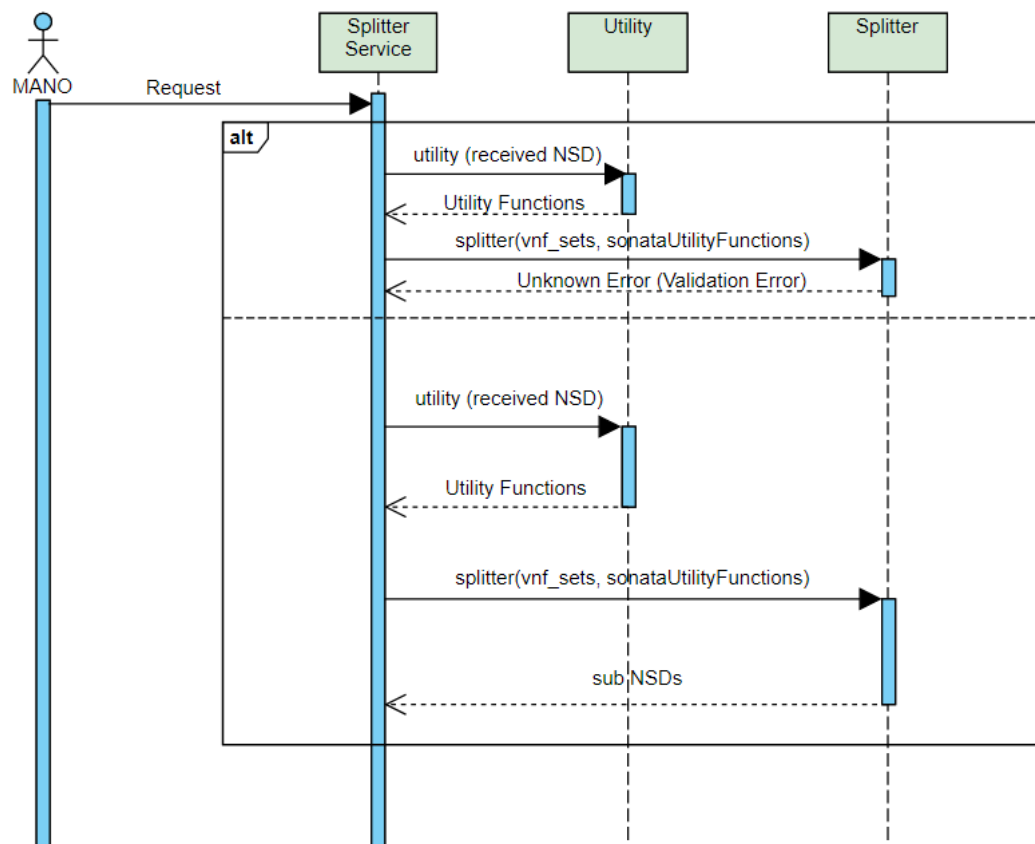


Figure 4.7: Splitter Sequence diagram

4.2.2 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.

```

1 splitter_url=http://$HOST:8003/Main_splitter/split
2
3
4 # Body: descriptor contains NSD, vnfid_set contains set of VNF ids
5 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 4.4: POST call to SDS

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

Python Base classes for NSD Schema

Following code snippet shows how NSD schema is mapped to various Python classes.

```

1
2 class NetworkFunction: # class representing a network function and its properties
3     vnf_id = ""
4     vnf_name = ""
5     vnf_vendor = ""
6     vnf_version = ""
7     connection_point_refs = []
8
9     def __init__(self , vnf_id , vnf_vendor , vnf_name , vnf_version , connection_point_refs
10 ):
11         self.vnf_id = vnf_id
12         self.vnf_vendor = vnf_vendor
13         self.vnf_name = vnf_name
14         self.vnf_version = vnf_version
15         self.connection_point_refs = connection_point_refs
16
17 class ConnectionPoint: # class representing a connection point and its properties
18     _id = ""
19     interface = ""
20     _type = ""
21
22     def __init__(self , _id , interface , _type):
23         self._id = _id
24         self.interface = interface
25         self._type = _type
26
27
28 class VirtualLink: # class representing a virtual link and its properties
29     _id = ""
30     # connectivity type can be 'E-LAN' representing many to many connectivity or
31     # 'E-Line' representing one to one
32     connectivity_type = ""
33     connection_points_reference = []
34

```

```

35     def __init__(self, _id, connectivity_type, connection_points_reference):
36         self._id = _id
37         self.connectivity_type = connectivity_type
38         self.connection_points_reference = connection_points_reference

```

Listing 4.5: Python Base classes

Splitting

"split sonata" 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

The current implementation of splitter can successfully split a NSD with three or less VNFs. To maintain the topology of the main NSD, the graph has to consider the virtual links between VNFs present in the main NSD. The main challenge in splitting a forwarding graph is to maintain the topology. In case of four or more VNFs, the possible scenarios for splitting increases which increases the complexity of splitter. It can be achieved by maintaining a mapping of original one to one virtual links while processing the virtual links and then creating new virtual link set based on the request from MANO. Example, consider three VNFs, A, B and C. Packet is supposed to flow from A to B to C. After splitting, the sequence of packet flow should not be altered. If there is a request from MANO to split the NSD into two parts with [A, C] and [B], then to insure the topology is not changed, the NSD has to be splitted into three NSDs instead of two. In reference to the above example and following code snippet, normal scenario refers to when the request from MANO is to split it in such a way which does not result in creation of new virtual links which are not there in the original NSD.

```

1
2 """
3 Method splits the forwarding path.
4 """
5 def split_forwarding_path(self):
6     for i in range(len(self.NSDs)):
7         nsd_fg = self.NSDs[i]
8         del self.processed_connection_point_path[:]
9         for fg in self.utilityFunctions.list_forwarding_graphs:
10
11             fg_inner = SonataSchema.ForwardingGraphs(fg.fg_id, fg.
12                                                         number_of_endpoints,
13                                                         len(self.set_constituent_virtual_links(
14                                                             nsd_fg, fg)), self.
15                                                         network_function_sets[i],
16                                                         self.set_constituent_virtual_links(
17                                                             nsd_fg, fg), [])
18
19             for path in fg.network_forwarding_path:
20                 if self.number_of_connection_paths(self.
21                                                         set_constituent_virtual_links(nsd_fg, fg)) > 1:
22                     for j in range(self.number_of_connection_paths(self.
23                                                         set_constituent_virtual_links(nsd_fg, fg))):
24                         path_inner = SonataSchema.NetworkForwardingPaths(path.fp_id
25                                     + "_" + str(j), path.policy, [])
26                         x = 0
27                         for cp in path.connection_points:

```

```

20         if self.connection_point_already_processed(cp.
21            connection_point_ref) is False:
22             found = 0
23             if cp.connection_point_ref in self.
24                get_connection_point_refs(nsd_fg.
25                networkFunctions):
26                 x = x + 1
27                 point = SonataSchema.ConnectionPointsGraph(cp.
28                    connection_point_ref, x)
29                 path_inner.connection_points.append(point)
30                 found = 1
31                 self.processed_connection_point_path.append([cp
32                    .connection_point_ref, 1])
33             else:
34                 for connection_point in nsd_fg.connectionPoints
35                    :
36                     if cp.connection_point_ref ==
37                        connection_point._id:
38                         x = x + 1
39                         point = SonataSchema.
40                            ConnectionPointsGraph(cp.
41                            connection_point_ref, x)
42                         path_inner.connection_points.append(
43                            point)
44                         found = 1
45                         self.processed_connection_point_path.
46                            append([cp.connection_point_ref,
47                            1])
48             if found == 0:
49                 string = cp.connection_point_ref.split(":")
50                 if string[1] == "input":
51                     x = x + 1
52                     point = SonataSchema.ConnectionPointsGraph(
53                        "output", x)
54                     path_inner.connection_points.append(point)
55                     self.processed_connection_point_path.append
56                        ([cp.connection_point_ref, 1])
57                     break
58                 if string[1] == "output":
59                     x = 1
60                     point = SonataSchema.ConnectionPointsGraph(
61                        "input", x)
62                     path_inner.connection_points.append(point)
63                     self.processed_connection_point_path.append
64                        ([cp.connection_point_ref, 1])
65                 fg_inner.network_forwarding_path.append(path_inner)
66             else:
67                 fg_inner.network_forwarding_path.append(self.normal_case(path,
68                    nsd_fg))
69             nsd_fg.forwardingGraphs.append(fg_inner)
70             self.NSDs[i] = nsd_fg

```

Listing 4.9: Forwarding Graph Splitting

4.2.3 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.

4.2.4 Future work

SDS can currently split NSD of Pishahang and OSM. SDS is built in such a way that it can be implemented for new 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. (4.2.2)

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 ETSI SOL005 which contains the blueprint for all the methods mentioned in the standards. These methods are divided into different sections as per ETSI SOL005 into the following:

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

In the figure 4.8, a high level architecture of PWM is shown. As part of the scramble project, support for Open Source MANO (OSM), Sonata and Pishahang are implemented based on the common interface provided by PWM. Wrappers also support additional functionalities of Pishahang, which is an extension of Sonata.

In the figures 4.10 and 4.11, the class diagram of OSM, Sonata and Pishahang PWM implementation is shown respectively. Note the additional "Admin" functionalities supported by PWM for both OSM and Sonata, these are individual non-standard APIs of the respective MANOs. In the figure 4.11, support for Pishahang specific APIs are also implemented along with the database APIs which are part of pg-scramble.

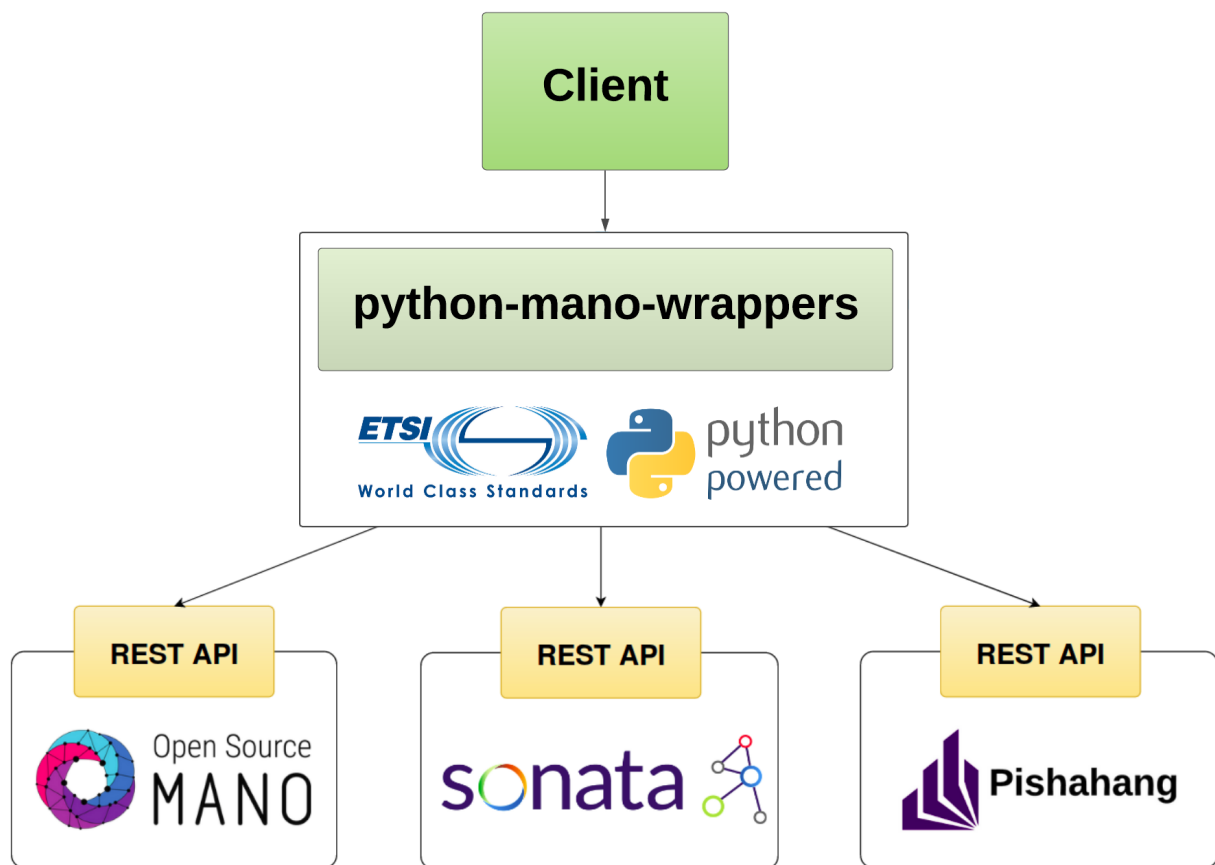


Figure 4.8: PWM high level architecture

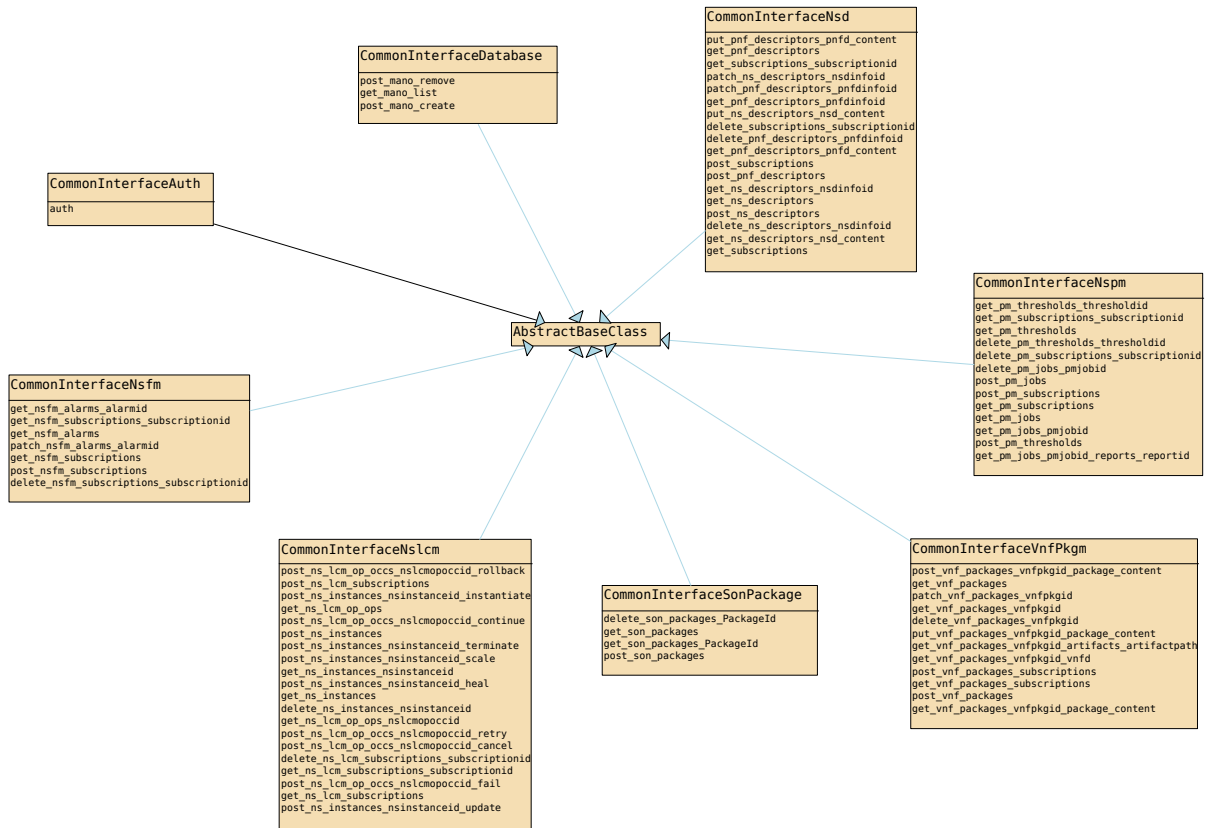


Figure 4.9: CommonInterface Abstract Base Classes defined in PWM

CHAPTER 4. SCRAMBLE SERVICES IMPLEMENTATION

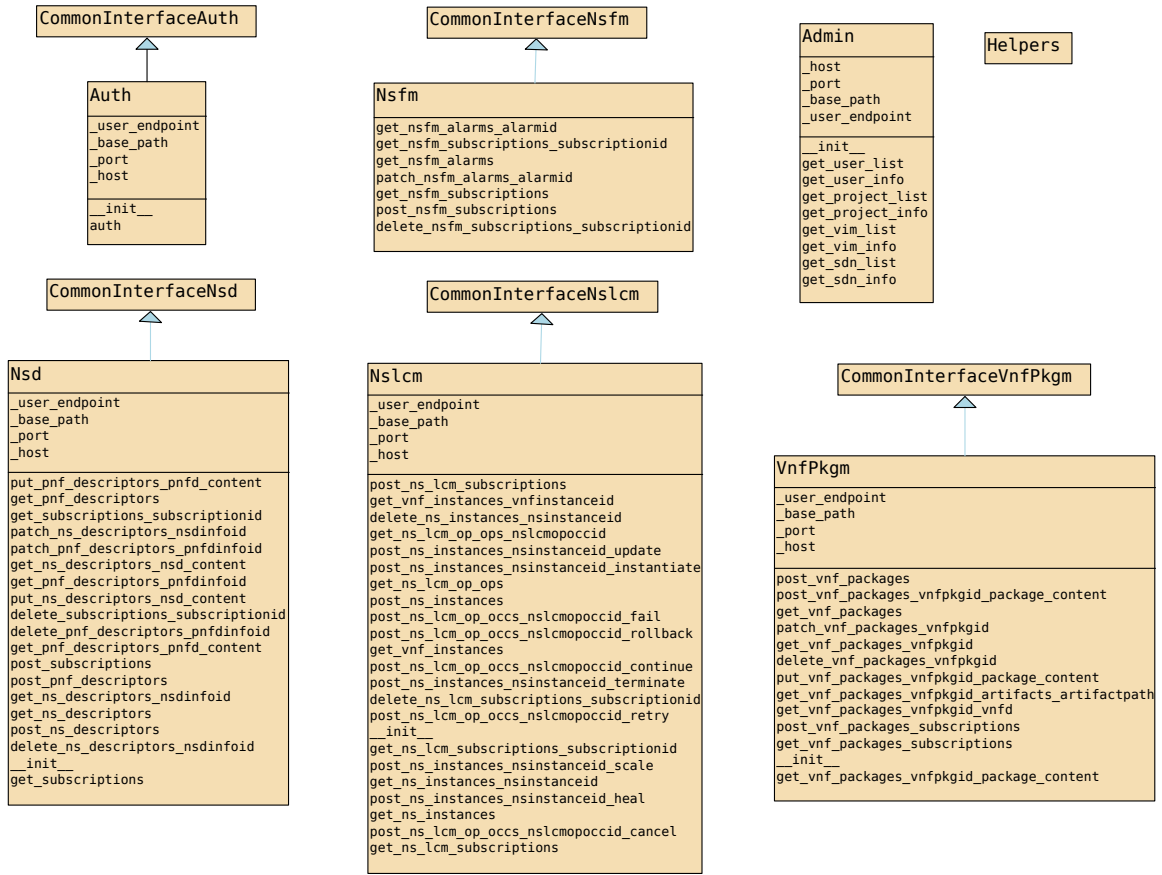


Figure 4.10: OSM Wrappers implemented based on the CommonInterface base classes

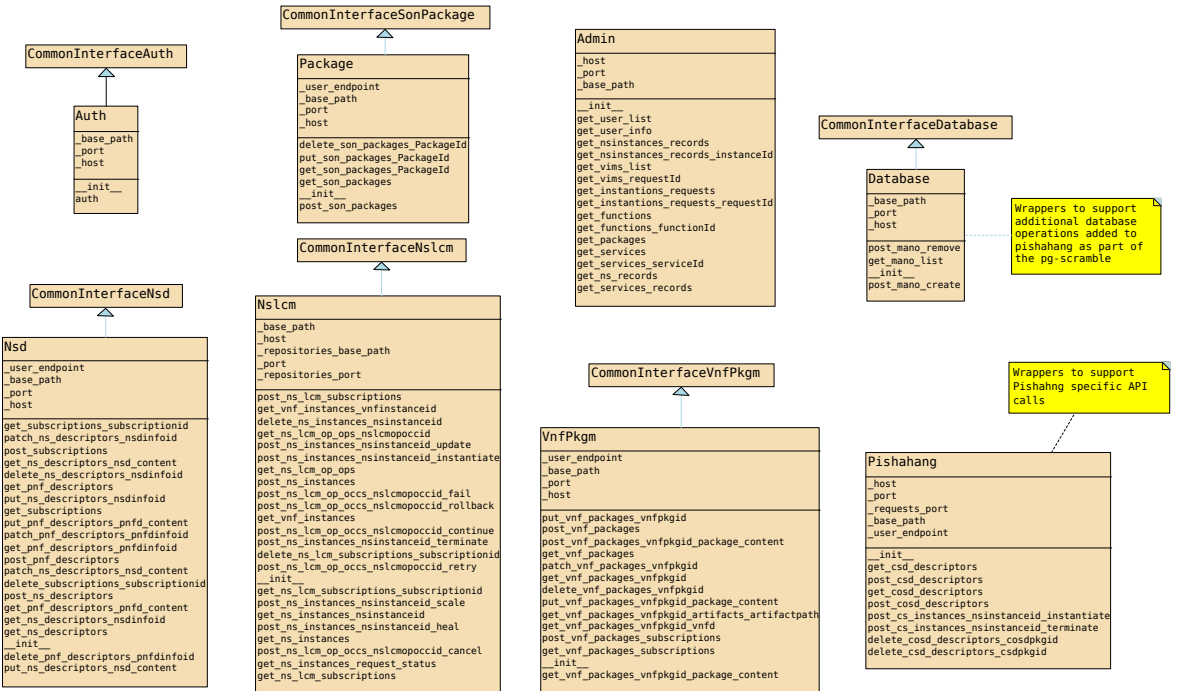


Figure 4.11: Sonata and Pishahang Wrappers implemented based on the CommonInterface base classes

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 4.10, 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 4.11, 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.

```

1 import wrappers
2
3 username = "admin"
4 password = "admin"
5 mano = "osm"
6 # mano = "sonata"
7 host = "osmmanodemo.com"
8
9 if mano == "osm":
10     _client = wrappers.OSMClient.Auth(host)
11 elif mano == "sonata":
12     _client = wrappers.SONATAClient.Auth(host)
13
14 response = _client.auth(
15     username=username, password=password)
16
17 print(response)

```

Listing 4.10: Simple wrapper code to fetch token

```

1 import wrappers
2
3 osm_nsd = wrappers.OSMClient.Nsd(HOST_URL)
4 osm_auth = wrappers.OSMClient.Auth(HOST_URL)
5
6 _token = json.loads(osm_auth.auth(
7     username=USERNAME,
8     password=PASSWORD))
9
10 _token = json.loads(_token["data"])
11
12 response = json.loads(osm_nsd.get_ns_descriptors(
13     token=_token["id"]))
14 response = json.loads(response["data"])

```

Listing 4.11: 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 beginning to decide on the scope of supported functionalities.

4.3.4 Future work

PWM is built with easy maintainability 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 functions 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.

```

1 instantiateScramble:function(id, ingresses, egresses, ENV, selectedmanos, manodetails){
2   var defer=$q.defer();
3
4   {...} ## unchanged
5
6   var data={"service_uuid":id, "ingresses": ingresses, "egresses": egresses, "scramble":
       true, "selectedmanos":selectedmanos, "manoips":manodetails};
7   $http.post(ENV.apiEndpoint+"/requests",data)
8   .then(function successCallback(result){defer.resolve(result)})
9   .catch(function errorCallback(error){defer.reject(error)});
10
11  return defer.promise;
12 },

```

Listing 4.12: 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.

```

1
2 post '/requests/?' do
3   log_msg = MODULE + '::POST /requests'
4   original_body = request.body.read
5   logger.debug(log_msg) {"entered with original_body=#{original_body}"}
6   params = JSON.parse(original_body, quirks_mode: true)
7   logger.debug(log_msg) {"with params=#{params}"}
8
9   # we're not storing egresses or ingresses
10  egresses = params.delete 'egresses' if params['egresses']
11  ingresses = params.delete 'ingresses' if params['ingresses']
12  user_data = params.delete 'user_data' if params['user_data']
13
14  begin
15    {...} ## unchanged
16
17    if params['scramble'] == true
18      start_request['scramble'] = true
19      start_request['selectedmanos'] = params['selectedmanos']
20      start_request['manoips'] = params['manoips']

```

```

21
22 end
23 {...} ## unchanged
24 end

```

Listing 4.13: create instantiation request in gatekeeper(request.rb)

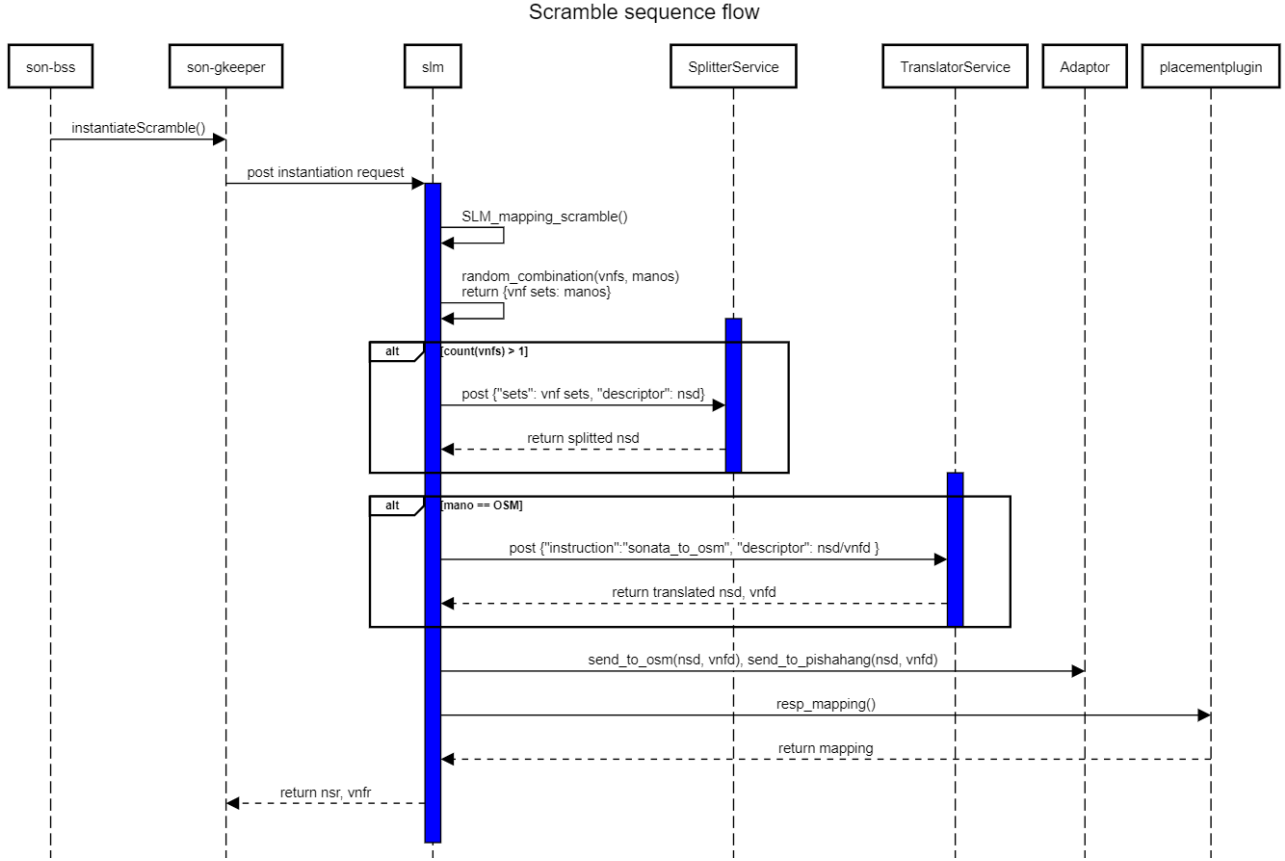


Figure 4.12: Integration sequence diagram

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:

```

1 def SLM_mapping_scramble(self, serv_id):
2     """
3     This method is used if the SLM is responsible for the placement.
4     :param serv_id: The instance uuid of the service
5     """
6     corr_id = str(uuid.uuid4())
7     self.services[serv_id]['act_corr_id'] = corr_id
8
9     LOG.info("Service " + serv_id + ": Calculating the placement ")
10    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 nsdsSplitted = json.loads(response.text) # get back 2 sets of sub-nsds
51
52 else:
53
54 nsdsSplitted = {"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 = nsdsSplitted['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, nsdsSplitted['message'][i])
76
77 elif sets[2][0]['type'] == 'OSM':
78
79 self.send_to_osm(serv_id, sets, functions, nsdsSplitted['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
92     ## as to keep the unique UUID of this instantiation request in ledger instead of
93     ## forced rollback.
94
95 self.inform_gk_instantiation_scramble(serv_id)
96
97 else:
98     content = {'nsd': NSD,
99               'functions': functions,
100               'topology': topology,
101               'serv_id': serv_id}
102 else:
103     {...} ## unchanged

```

Listing 4.14: Extended `SLM_mapping_scramble` function

4.5 Installation of Pishahang with Scramble

In order to use and exploit the functionalities of all the work packages of scramble described above, additional steps needs to be executed after a clean installation of Pishahang. The steps are listed below in the script.

```

1 sudo apt-get install -y software-properties-common
2 sudo apt-add-repository -y ppa:ansible/ansible
3 sudo apt-get update
4 sudo apt-get install -y ansible
5 sudo apt-get install -y git
6 git clone --single-branch --branch scramble-pishahang https://github.com/CN-UPB/pg-
   scramble.git
7 cd pg-scramble/pishahang/Pishahang-master/son-install
8
9 git checkout install-pishahang
10 mkdir ~/.ssh
11 echo sonata | tee ~/.ssh/.vault_pass
12
13 ansible-playbook utils/deploy/sp.yml -e "target=localhost public_ip=<<ip-address>>" -v
14
15 cd pg-scramble
16 sudo ./run_scramble.sh <<ip-address>>

```

Listing 4.15: install Pishahang with scramble

4.6 Scramble GUI for Pishahang

The scramble-gui is integrated with both pishahang-gui and in pishahang-BSS developed with Angularjs for front end and mangodb deployed using python for back-end., enabling users to enter the details of child-MANO's and also select the desired MANO's to instantiate a service.

With Scramble-pishahang GUI the users can enter details of child-MANO's as shown in images below.

Step 1: Select *MANO Settings* and then *Add MANO*

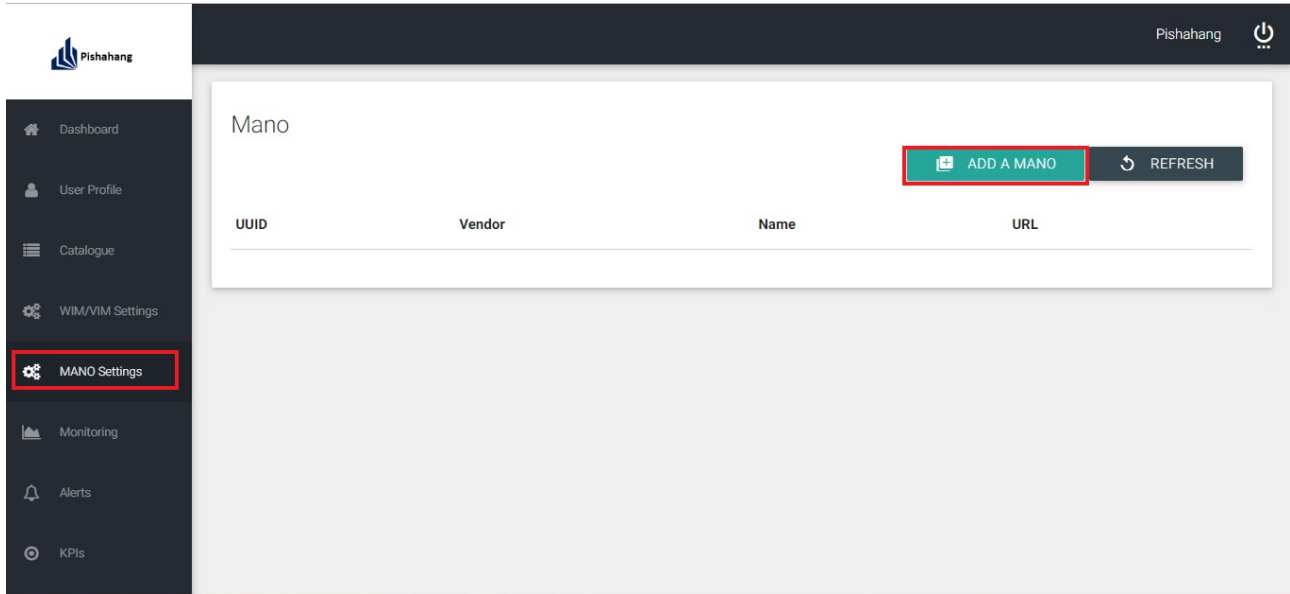
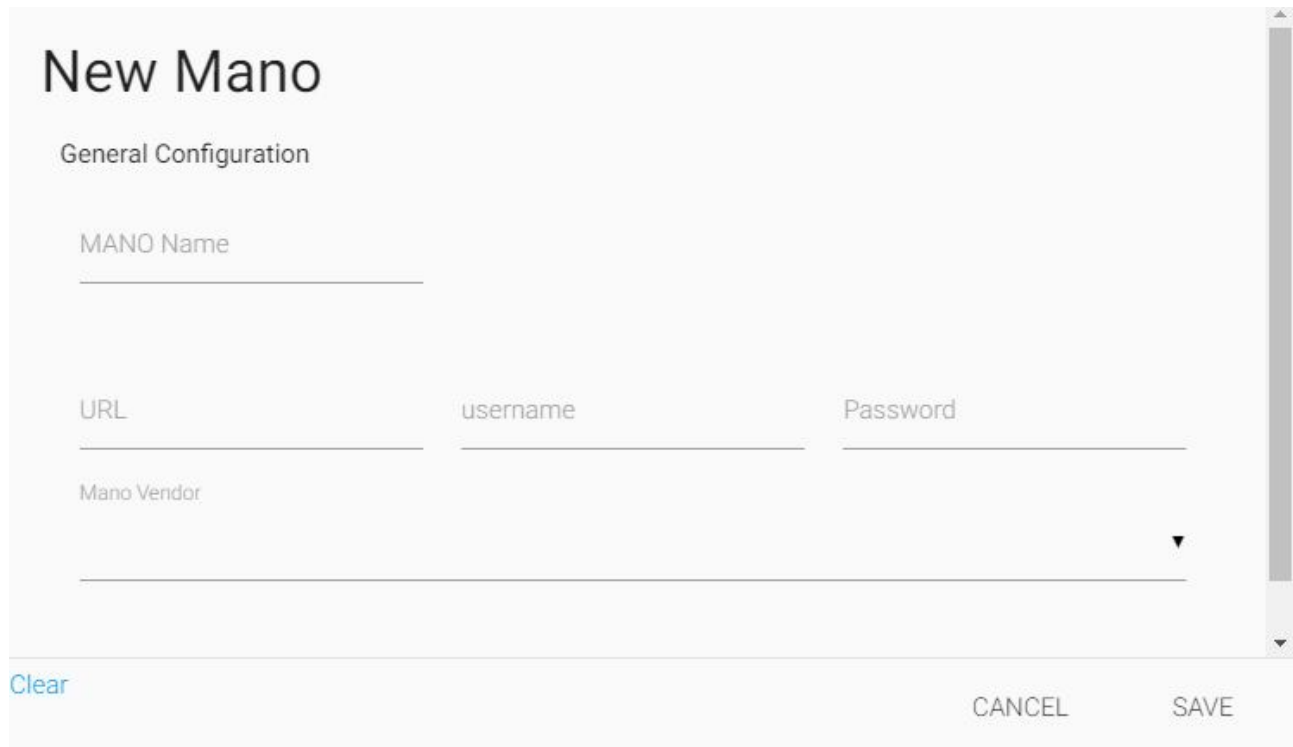


Figure 4.13: Scramble_pishahang GUI

Step 2: Enter MANO details, The required MANO details are

1. MANO Name: The name is take so that the user can easily differentiate between MANO's and can be any name desired by user.
2. URL: Is the URL to access the MANO
3. Username and Password: Is the credentials required to access the MANO
4. MANO Vendor: Is a choice between OSM and Pishahang to recognize the type of child-mano added.



New Mano

General Configuration

MANO Name

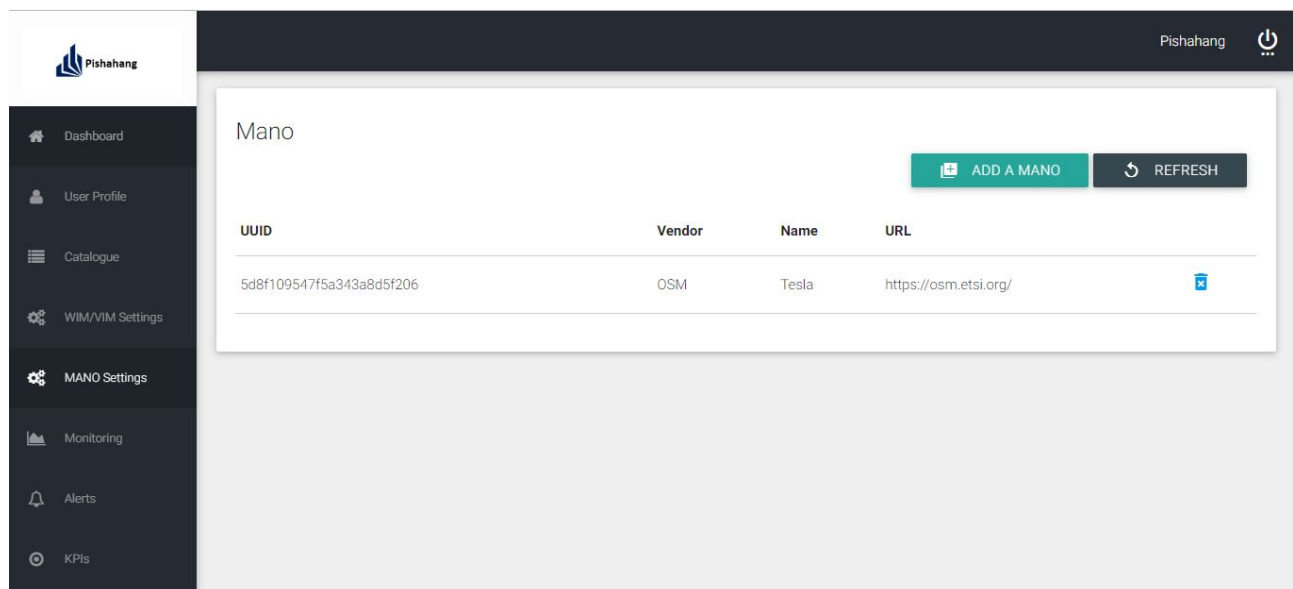
URL username Password

Mano Vendor

[Clear](#) [CANCEL](#) [SAVE](#)

Figure 4.14: Scramble_pishahang GUI

Once the MANO details are entered and the list of MANO's added are displayed.



Mano

[ADD A MANO](#) [REFRESH](#)

UUID	Vendor	Name	URL
5d8f109547f5a343a8d5f206	OSM	Tesla	https://osm.etsi.org/

Figure 4.15: Scramble_pishahang GUI

Once the child-MANO's are added, in the pishahang-bss the added MANO's are listed. The user can select the MANO's in which the service can be instantiated and also if user does not need to instantiate the service in child MANO, he can instantiate the service just in the parent MANO as shown below.

CHAPTER 4. SCRAMBLE SERVICES IMPLEMENTATION

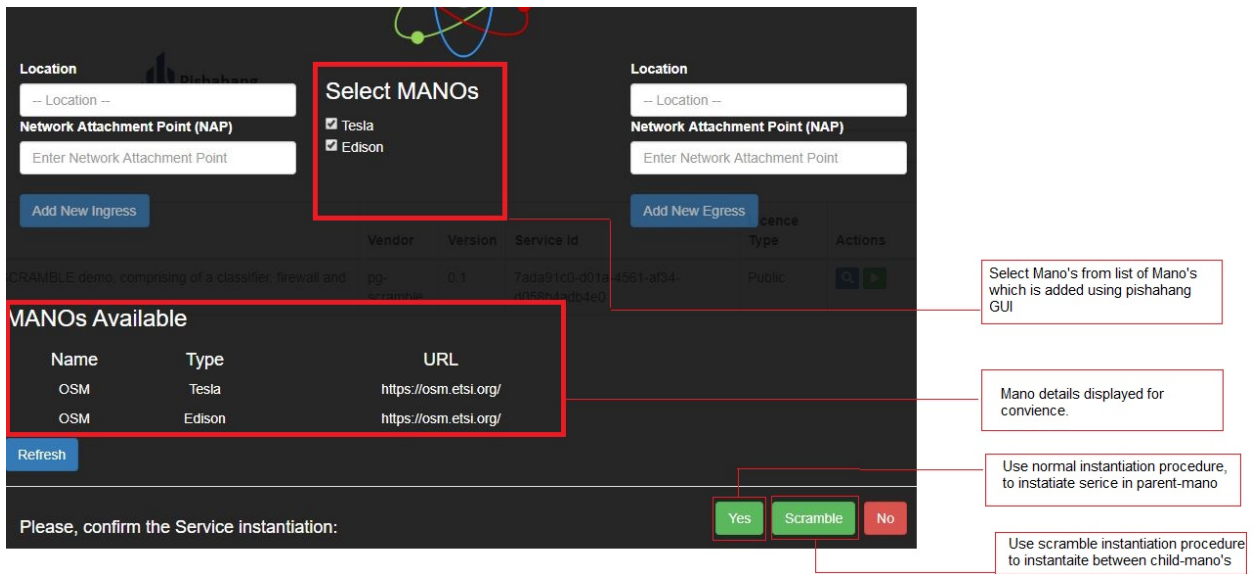


Figure 4.16: Scramble_pishahang GUI

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.

5.1 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

5.1.1 Architecture

SPL is built based on the "base plugin framework" of SONATA. The figure 5.1 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.

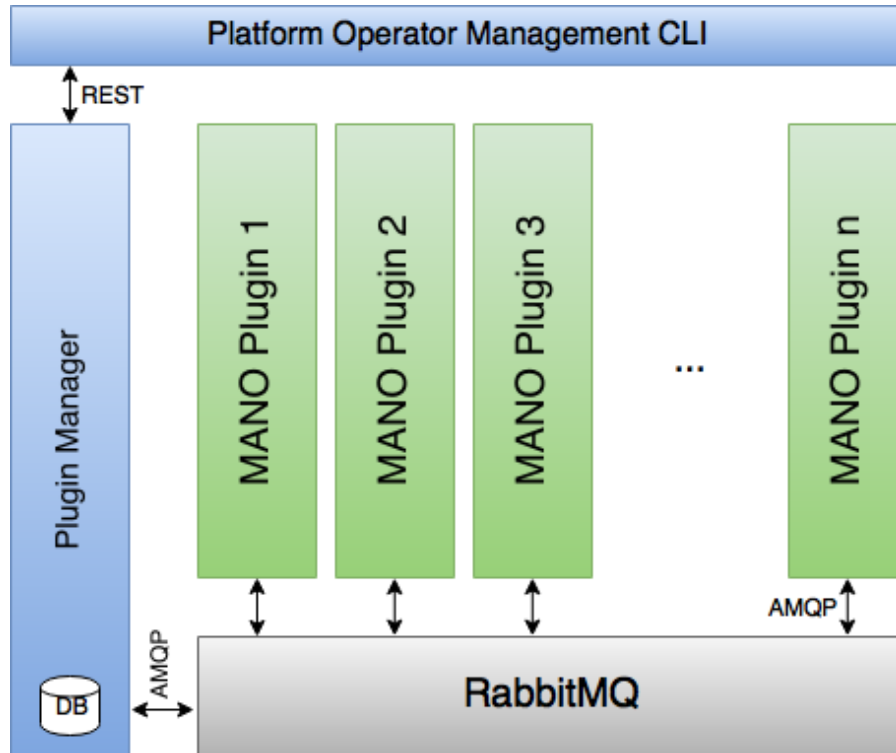


Figure 5.1: High-level SONATA MANO plugin architecture

The architecture of the scalability plugin is shown in the figure 5.2. 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-manowrappers. 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.

1. 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. Some informal discussions about this are available here ^{1 2}.

- **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 5.1: Scaling Plugin Rules

2. 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.

3. 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 5.1, 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.

¹<https://support.itrsgroup.com/hc/en-us/articles/360020251273-check-load-thresholds>

²<https://medium.com/devops-process-and-tools/how-to-measure-cpu-load-v-cpu-percentage-and-examples-da6057e39111>

4.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 5.1, 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.

1.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, the descriptors are listed in the appendix A.3.

2.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.

5.1.2 Workflow

SLM with SLP

The workflow of how the SLM and SLP interacts is shown in the figure 5.3. 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 5.4, we visualize the rules set of SLP from the List 5.1. SLP is always in the monitoring loop and takes actions according to the system load and the rules set.

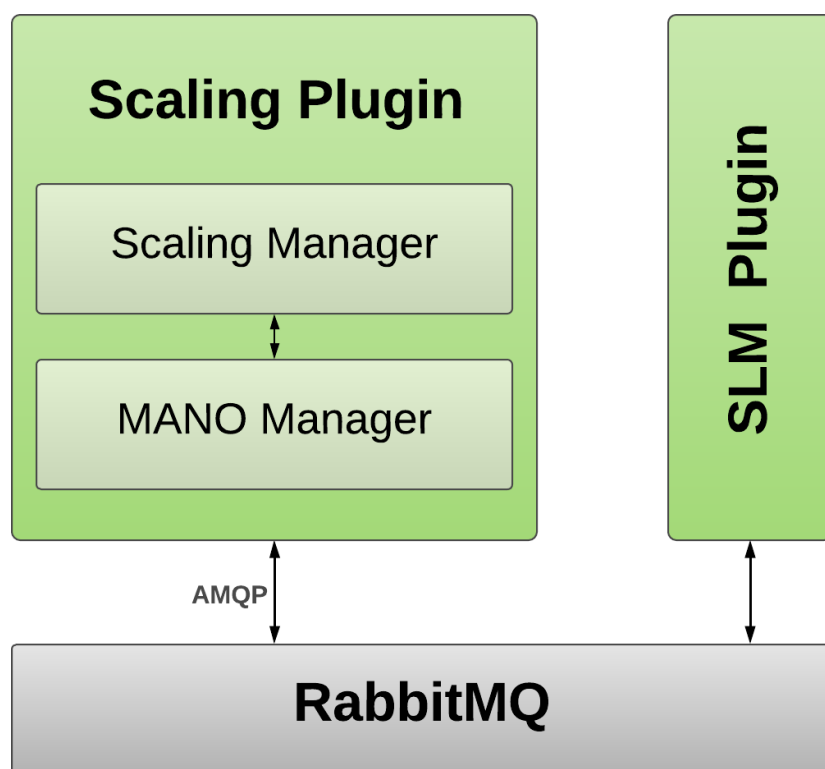


Figure 5.2: Scaling plugin architecture

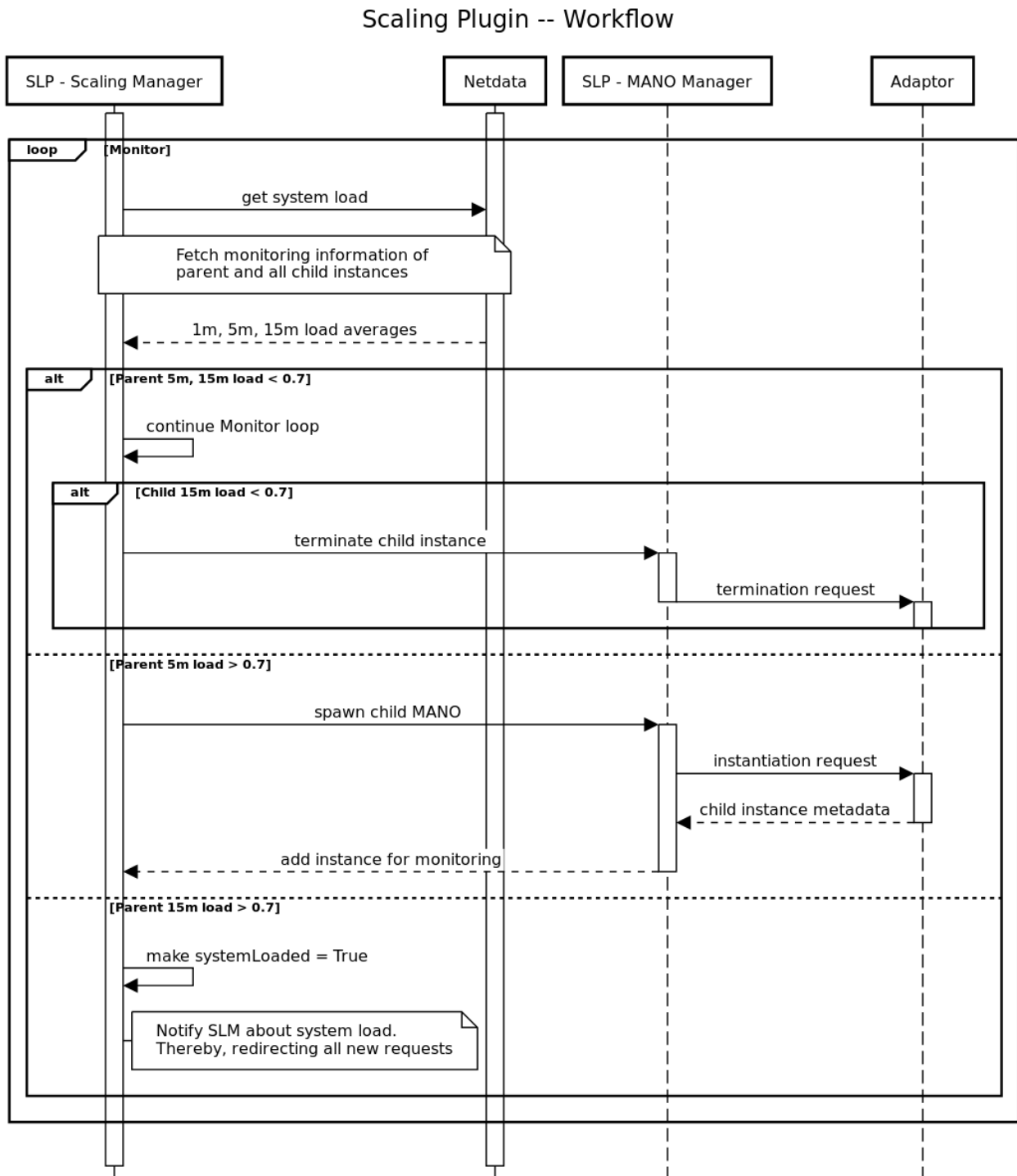


Figure 5.3: Service Lifecycle Manager with Scaling Plugin Workflow

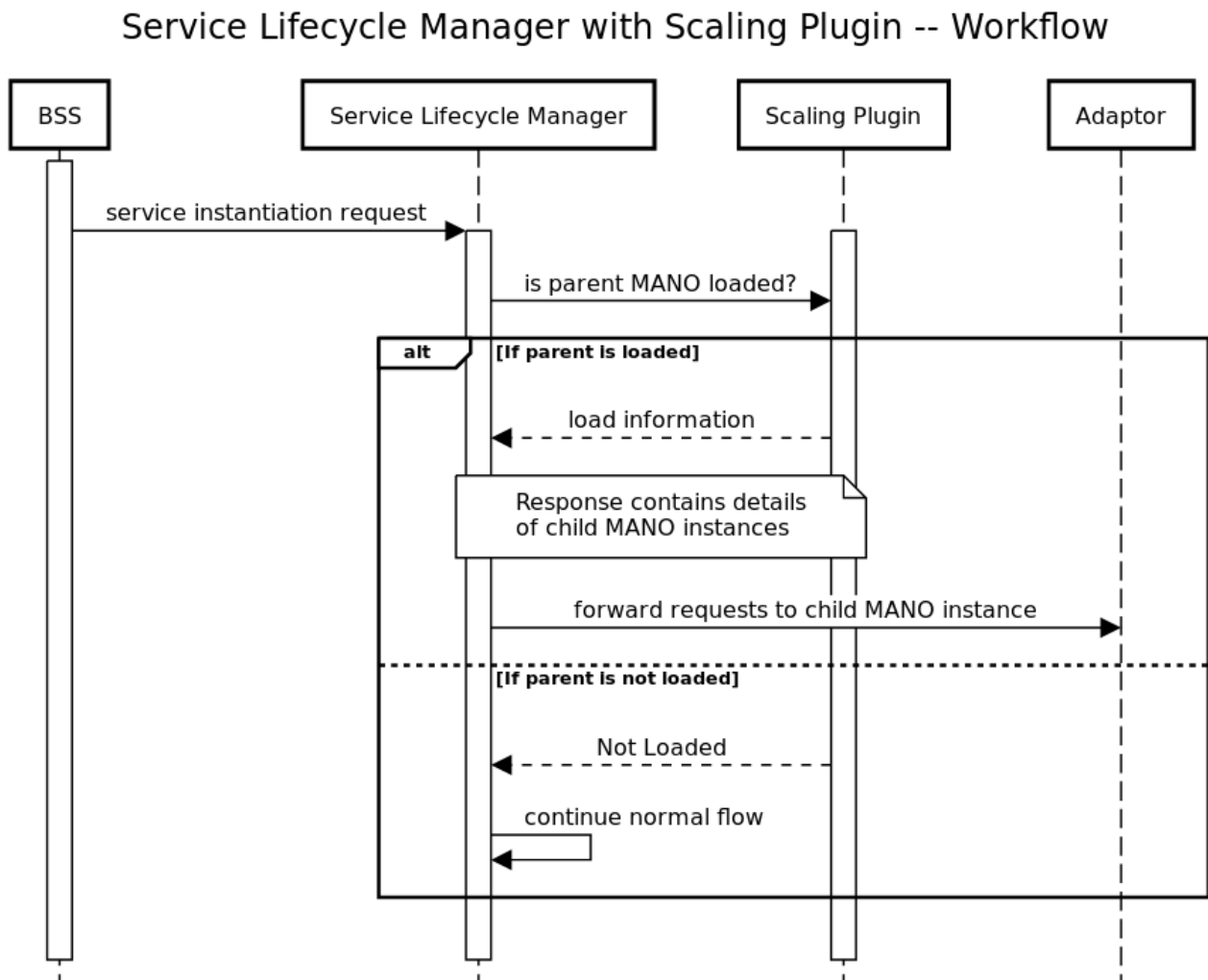


Figure 5.4: Scaling Plugin Workflow

5.2 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). The recommended ratio of physical CPU to virtual CPU is 1:16, we have 16, thus $16 \times 16 = 256$ cores³. Hence we first assigned 1 CPU to one service request.

We instantiated 256 service requests on OSM. 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. This trial and error was done in OSM. Once we got the appropriate requests-RPM combination, the experiment was conducted both in OSM and Pishahang.

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.

Apart from this, we also wanted to capture the CPU utilization throughout the experiment. We call this lifecycle graphs. The lifecycle of the experiment involves on-boarding, instantiating and terminating network services. This was conducted both on OSM and Pishahang.

5.2.1 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

³<https://docs.openstack.org/arch-design/design-compute/design-compute-overcommit.html>

- **Kubernetes (server 5)**

Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz with 16 core CPU and 100 GB memory

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.

5.2.2 OSM Results

The 5.5, 5.6 and 5.7 shows the CPU utilization, memory utilization and CPU utilization through out 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 5.5 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-coordinating 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 artifacts) 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 retrieve 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 of the NS.)

OSM - CPU Usage - 180 Instances (30 rpm)

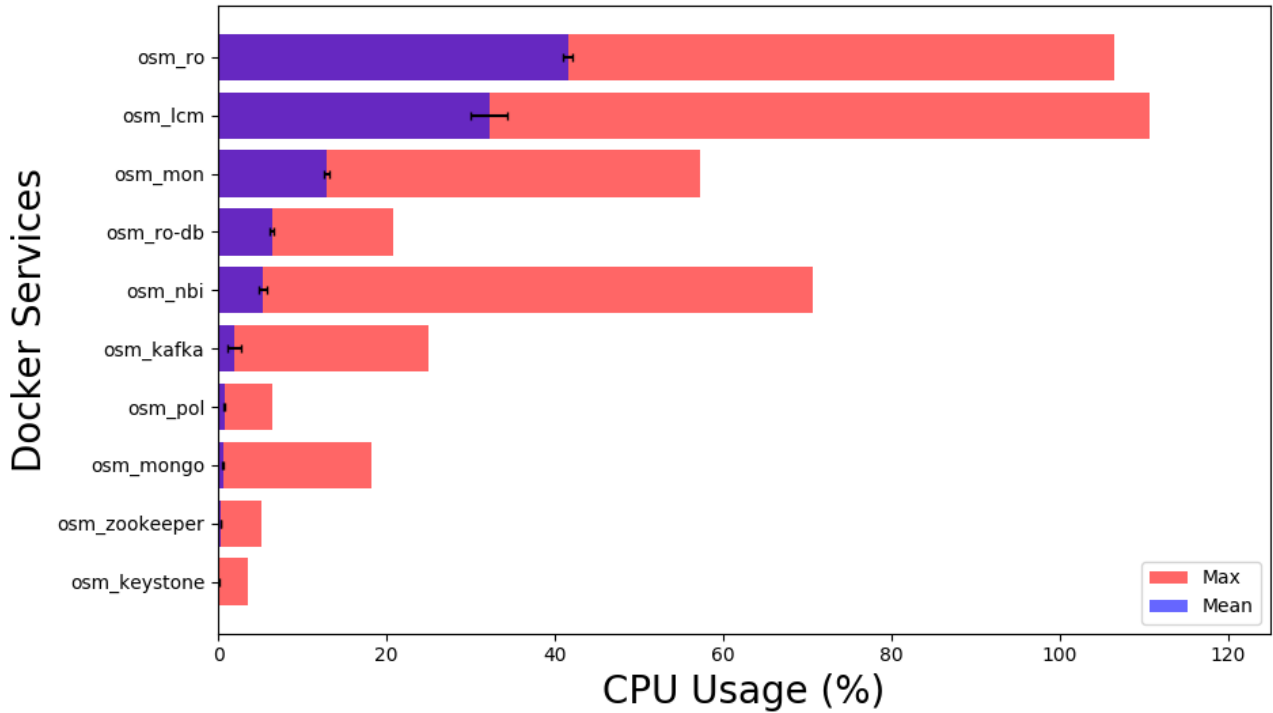


Figure 5.5: OSM CPU

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 capture idle metrics for about a minute and then the service requests starts getting instantiated at 30 RPM.

OSM - Memory Usage - 180 Instances (30 rpm)

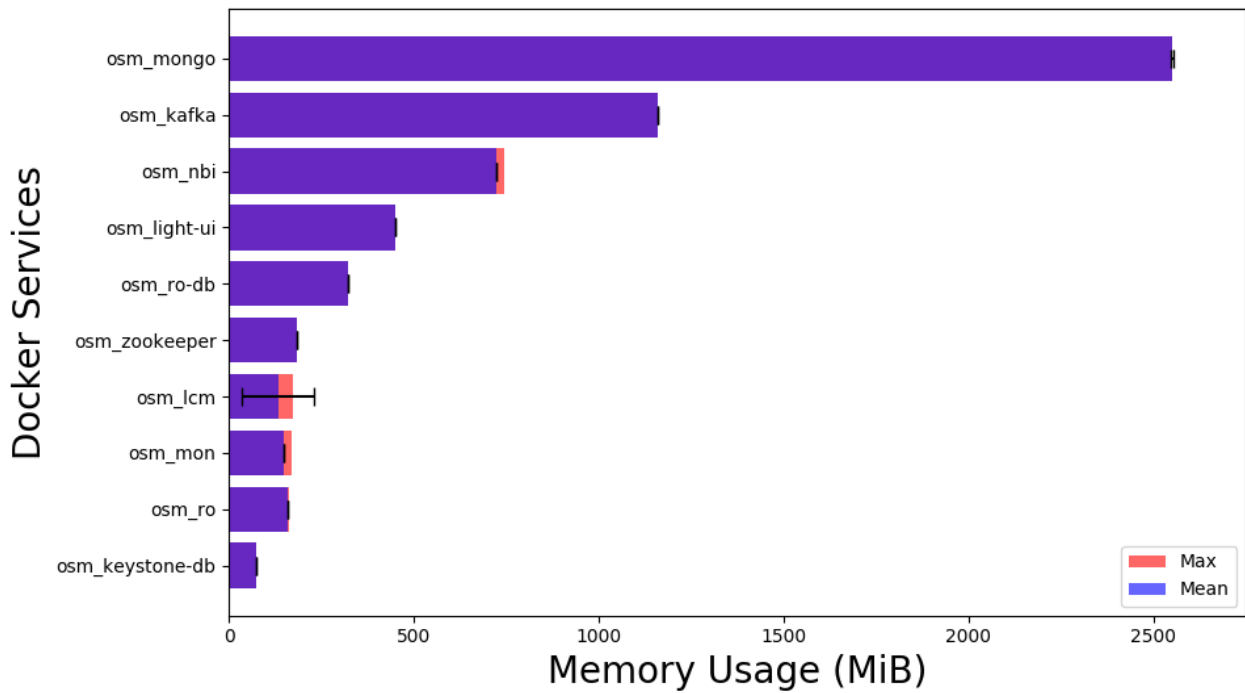


Figure 5.6: OSM MEM

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 continuously processed the requests. mon module retrieved metric information continuously and the same trend remains throughout the experiment. Once all the instances are successfully instantiated, the termination requests are executed.

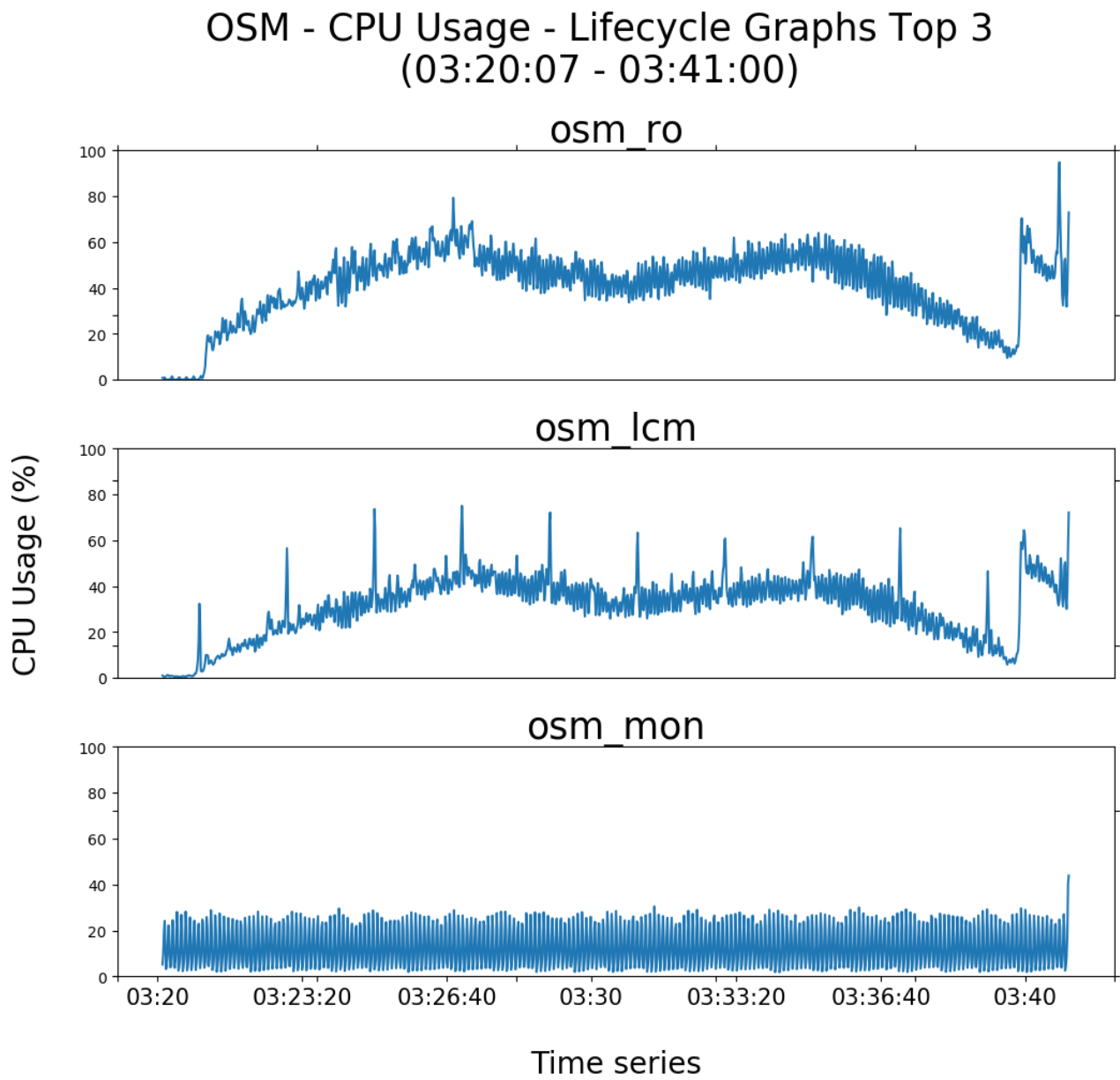


Figure 5.7: OSM LS

5.2.3 Pishahang Results

The figures 5.8 5.9 and 5.10 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 5.5 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 virtualization 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, registration, updation and termination. ex: to obtain onboarding SSM request from SLM.
- **cloudservicelifecyclemanagement:** This docker is responsible for lifecycle management of cloud network services on kubernetes.
- **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)

Pishahang - CPU Usage - 180 Instances (30 rpm)

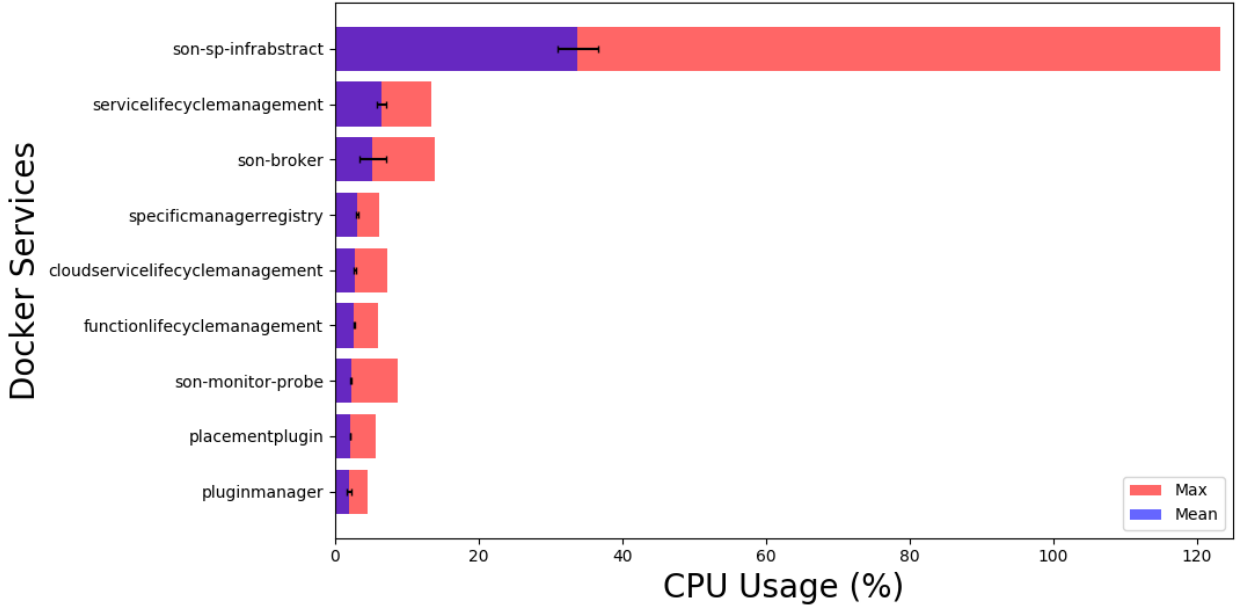


Figure 5.8: Pishahang CPU

Memory Utilization

The first 5 Pishahang dockers in memory utilization graph from figure 5.9 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 virtualization 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 5.6 and 5.9 is that, the memory utilization by Pishahang is much lighter than OSM for the top docker services.

Pishahang - Memory Usage - 180 Instances (30 rpm)

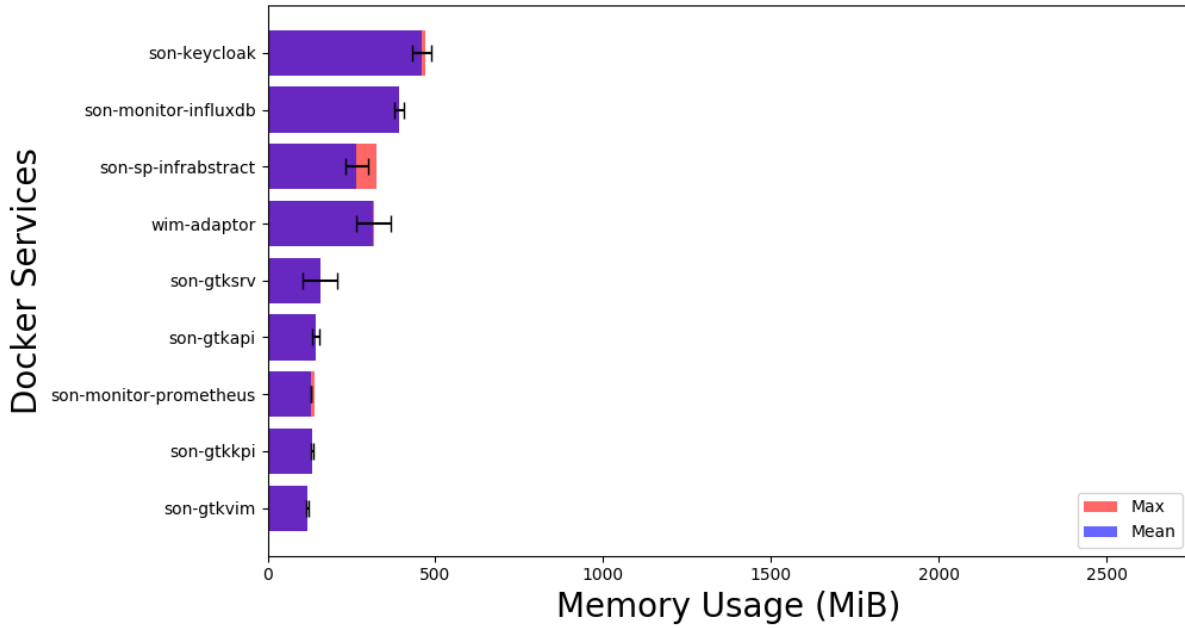


Figure 5.9: Pishahang MEM

Lifecycle

We now have the life cycle graphs of the entire experiment. The figure 5.10 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 5.10 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.

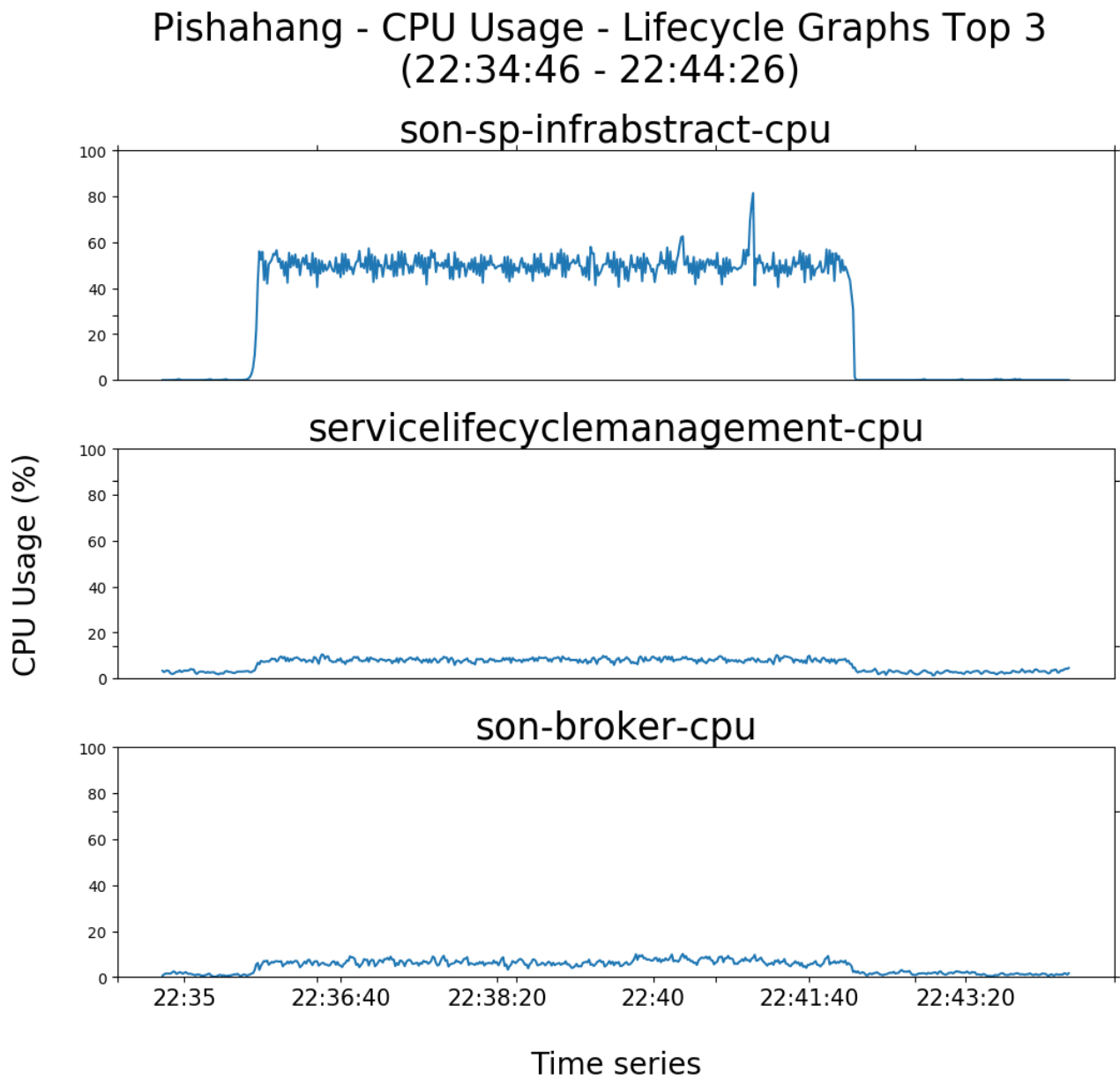


Figure 5.10: Pish LS

5.2.4 Limitations of the 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
- Pishahang doesn't have stable support for VM orchestration.
- No service function chaining or forwarding graph support in MANO frameworks yet

5.3 MANO Benchmarking Framework

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.

5.3.1 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**⁴ 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.

5.3.2 Parameters and KPIs

Parameters

MBF has experiment parameters that can be altered. The following parameters 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.
- **Skip experiment on error** if set to true, the current run is skipped due to a failed instance on the VIM.

⁴<https://github.com/netdata/netdata>

5.3.3 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.

5.3.4 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-branch⁵
2. Build and start the docker container
3. Change experiment variables in the relevant scripts for respective MANOs
 - **OSM** – *run-experiment-osm.py*
 - **Sonata and Pishahang** – *run-experiment-sonata.py*
 - **Pishahang Container Orchestration** – *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.*

⁵<https://github.com/CN-UPB/MANO-Benchmarking-Framework>

5.3.5 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 5.11 and 5.12 show the CPU utilization of the OSM microservice LCM and RO respectively.

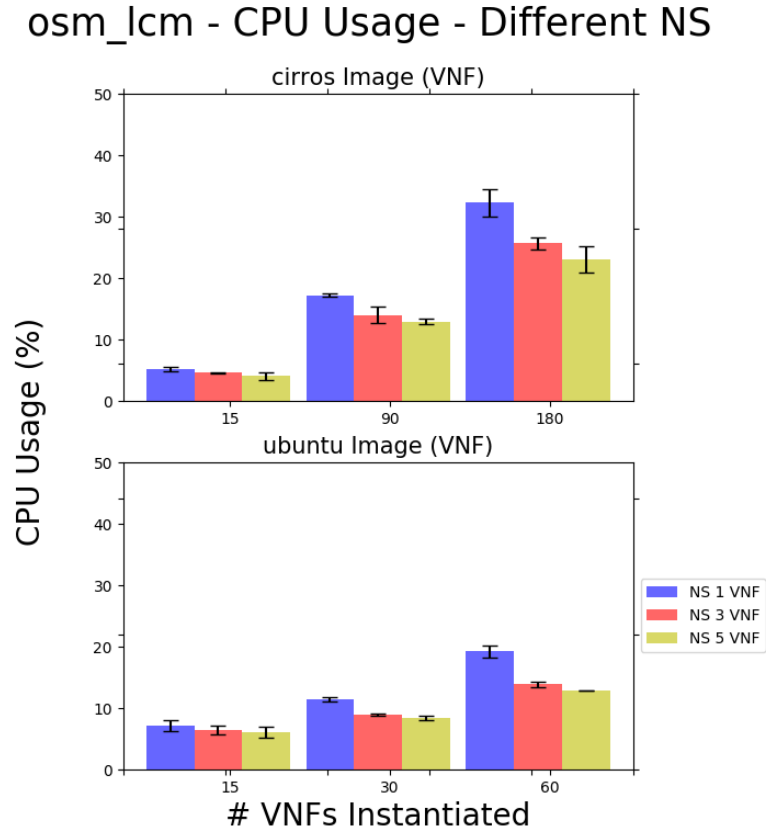


Figure 5.11: CPU usage of OSM microservice LCM

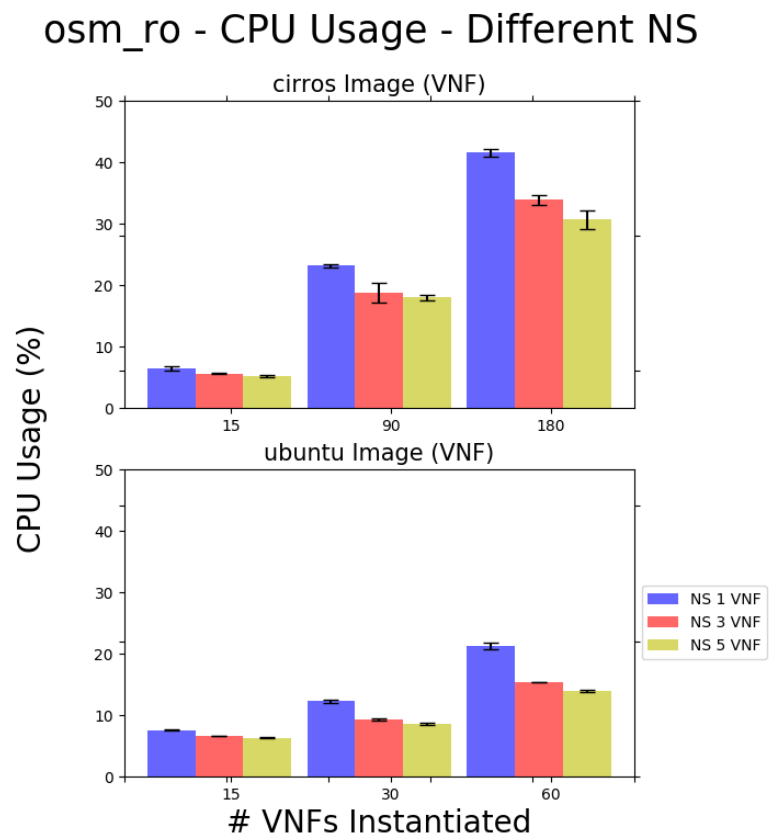


Figure 5.12: CPU usage of OSM microservice RO

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 5.13, 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 5.13 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.

OSM (VM) vs Pishahang (Docker) - 90 Instances

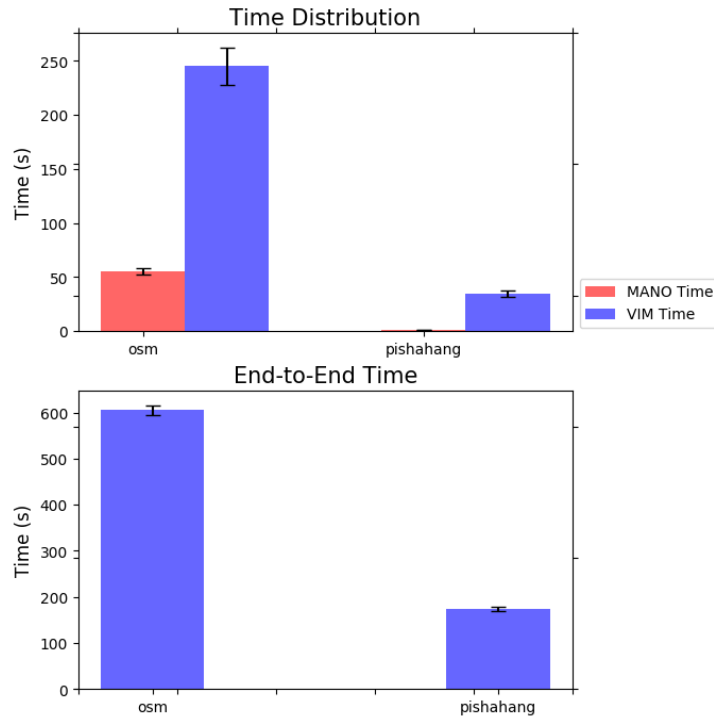


Figure 5.13: Time distribution in MANO and VIM

Scaling Plugin Evaluation

We utilized MBF to evaluate our Scaling Plugin discussed in the section 5.1. 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 5.1. 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.

OSM (VM) vs Pishahang (Docker) - 90 Instances

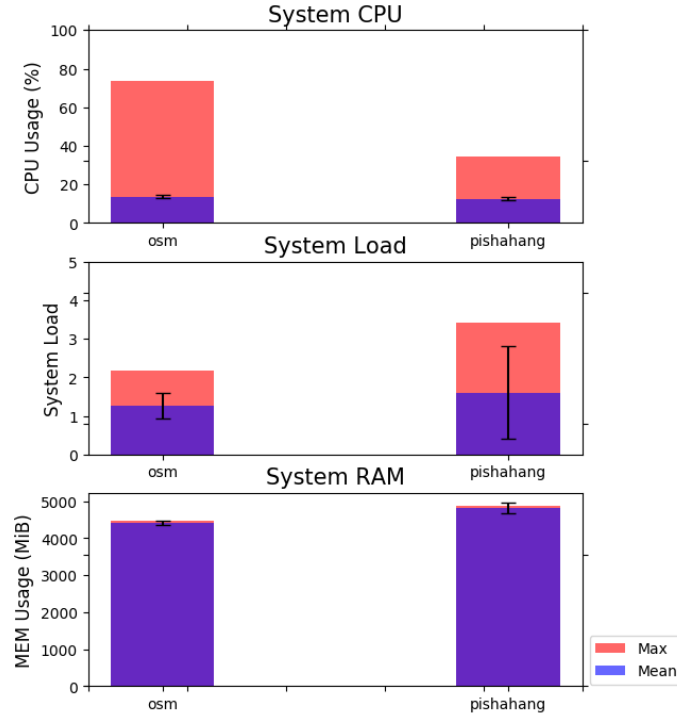


Figure 5.14: System resource utilization

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 5.1. 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 5.15. 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 5.16.

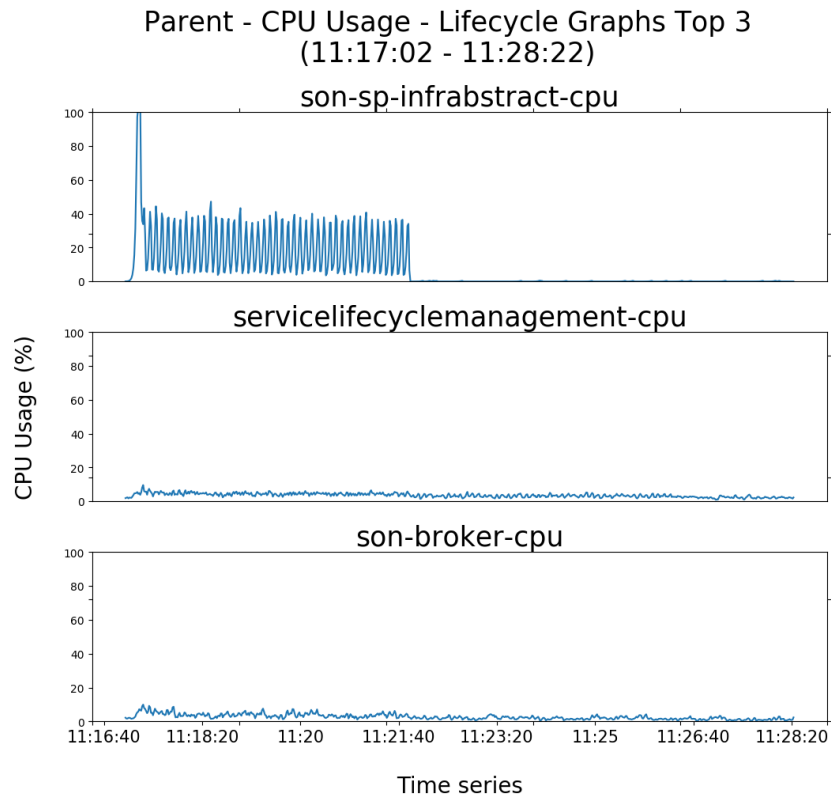


Figure 5.15: Parent MANO instance lifecycle graph

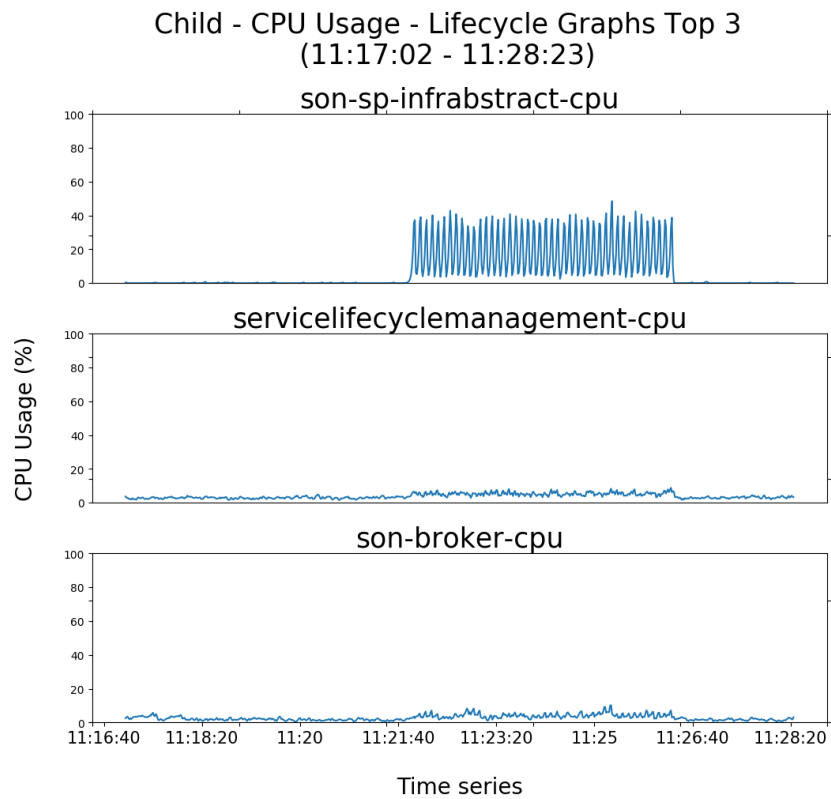


Figure 5.16: Child MANO instance lifecycle graph

5.4 Discussions

In this section, we discuss the observations and inferences drawn from all the experiments conducted on OSM and Pishahang.

Requests per minute (RPM)

We observed that the number of successful instantiation of network services and RPM are inversely proportional. For this, we instantiated 90 cirros instances at various RPMs and compared the successful instantiations of VNFs. We carried out this experiment on OSM as the support for VM orchestration is only a proof of concept in Pishahang and we could not instantiate more than 10 instances at once.

We saw that when the service requests are sent at 30, 60 and 120 RPMs all the VNFs were successfully instantiated in VIM. But, when the requests were sent at 240 RPMs, we saw on average, 67 successful instantiations, 5 resulted in build errors on the VIM and the remaining requests were lost at the MANO side due to timeouts and internal server errors.

An interesting observation here is the end-to-end deployment times of various RPMs, i.e, the time taken between the start of instantiation requests and all the instances getting successfully instantiated. We see from the table below that the end-to-end time actually increased with RPM. Even though the requests from the MANO are sent faster, VIM is taking much longer to actually process the requests.

RPM	End-to-End time
30	555
60	550
120	691

NSDs with different number of VNFDs

To observe the differences in resource utilization for different network services, we designed three different NS. These 3 NS differed in the number of VNFs, i.e, 1, 3 and 5 VNFs respectively. All the VNFs are cirros images which is used primarily for testing. We also created 3 more identical NS with Ubuntu cloud image as the VNF. The NS are all instantiated in such a way that the number of instantiations remain the same irrespective of the number of VNFs. For example, if the first NS that has one VNF is instantiated 15 times, the second NS is instantiated five times because it has 3 VNFs. The third NS is instantiated three times as it has 5 VNFs, so that all the three NS will result in 15 VNF instantiations on the VIM. Similarly, we also had 90 and 180 instantiations.

We observed that NS with more number of VNFs consumed more CPU than the ones with lesser number of VNFs. The image type (cirros or Ubuntu) does not make a difference on the resource utilization on the MANO but it does have an effect on the VIM side due to the more physical resources required by the Ubuntu image.

Results of the top 2 CPU consuming microservices for OSM are shown in the figure 5.12 and 5.11. Results of top 2 CPU consuming microservices for Pishahang are shown in the figure 5.17 and 5.18

Container v/s VM

We wanted to compare the resource utilization of two MANO frameworks for orchestration of similar NS. But, Pishahang's support for VM orchestration is unstable. Hence, we were not able to successfully instantiate more than a few (i.e. about 10) NS continuously. It was out of scope of our work to look

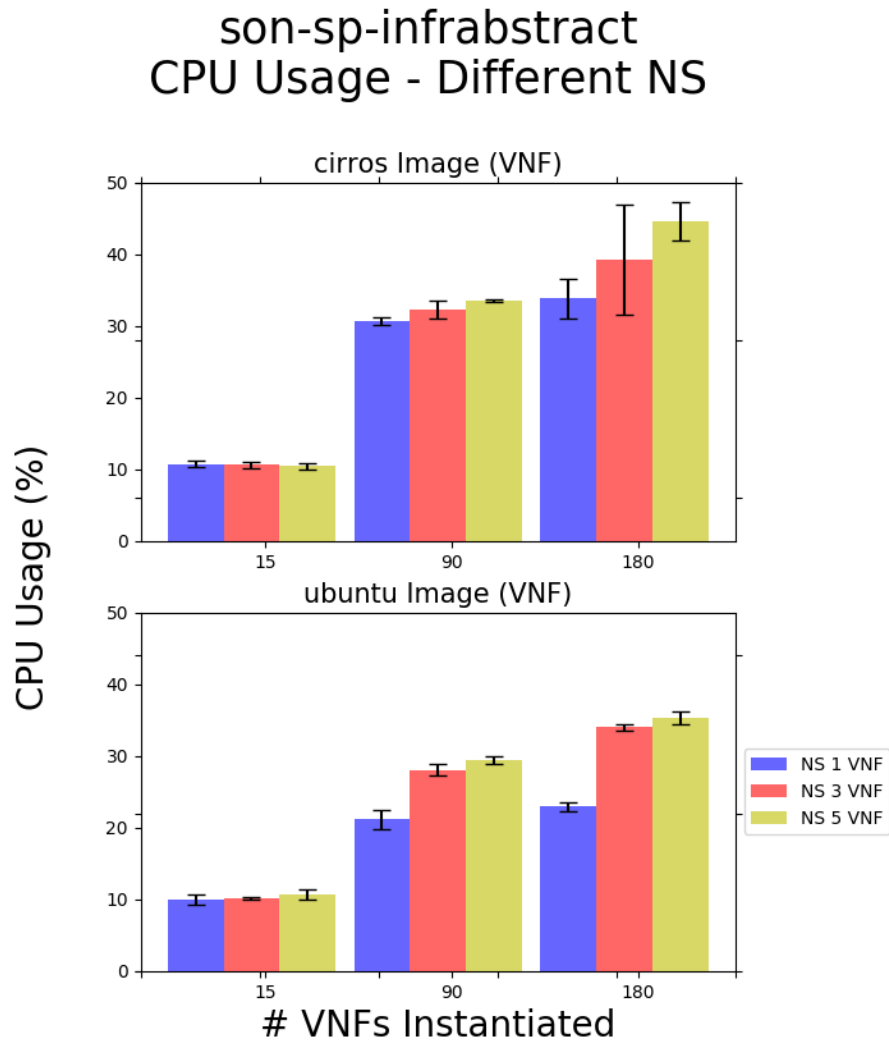


Figure 5.17: CPU usage of Pishahang microservice son-sp-infrabstract

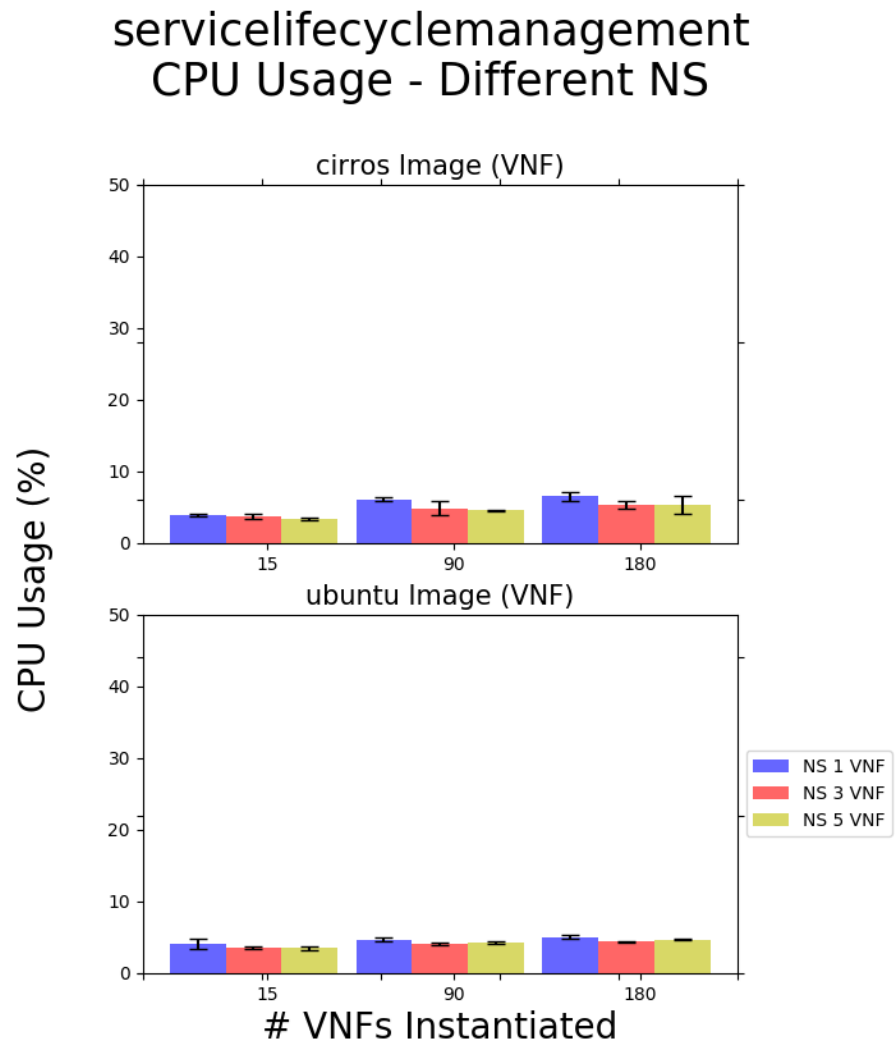


Figure 5.18: CPU usage of Pishahang microservice servicelifecyclemanagement

into the issue and fix them. However, Pishahang has a stable support for container orchestration. We thus decided to compare containers and VMs. The results are discussed in the section 5.3.5.

5.4.1 Understanding OSM lifecycle graphs

The figure 5.19 shows the lifecycle graphs of the experiment for instantiating 90 instances of the basic cirros NS at 30 RPM. As discussed in the previous sections, the top 3 microservices in terms of CPU usage are RO, LCM and MON respectively.

To further understand the uneven peaks and the general trend of RO and LCM graphs, we used a performance profiling tool called Pyflame⁶. It takes snapshots of the Python call stack to profile a program without modifying its source code. We used Pyflame on the RO and LCM python processes running inside their respective docker containers and started our experiment. We captured flame graphs every 30 seconds and analysed the frequency distribution of the function calls in these containers. This will give us a better understanding of the functionality that is taking the CPU resource.

We identified the major function calls of RO and LCM by analyzing the flame graphs. We looked into the source code to understand the functionality and the list containing the references is given below.

- **RO Functions**

- `_refres_elements`⁷
- `_proccess_pending_tasks`⁸
- `new_vm`⁹
- `del_vm`¹⁰

- **LCM Functions**

- `instantiate`¹¹
- `terminate`¹²
- `kafka_ping`¹³

We plotted the frequency distribution of these functions over experiment time, the resulting graph can be seen in the figure 5.20. The instantiation requests are coming into OSM until 9:32:54, by looking at the figure 5.19 and 5.20 it can be seen that all the 4 RO functions listed above are active during this period causing the spikes. It is similar in the LCM graph, the `instantiate` and `terminate`

⁶<https://github.com/uber/pyflame>

⁷https://osm.etsi.org/gitweb/?p=osm/RO.git;a=blob;f=osm_ro/vim_thread.py;h=d5301f1a6eafa3bf55fe3b4bea17537527dc731d;hb=refs/heads/v5.0#l253

⁸https://osm.etsi.org/gitweb/?p=osm/RO.git;a=blob;f=osm_ro/vim_thread.py;h=d5301f1a6eafa3bf55fe3b4bea17537527dc731d;hb=refs/heads/v5.0#l522

⁹https://osm.etsi.org/gitweb/?p=osm/RO.git;a=blob;f=osm_ro/vim_thread.py;hb=73ce6ce177937bac038f892e75dfe5a95dc8feeb#l810

¹⁰https://osm.etsi.org/gitweb/?p=osm/RO.git;a=blob;f=osm_ro/vim_thread.py;hb=73ce6ce177937bac038f892e75dfe5a95dc8feeb#l863

¹¹https://osm.etsi.org/gitweb/?p=osm/LCM.git;a=blob;f=osm_lcm/ns.py;h=83e7147234c66cb348f1c78877fd602781b1b987;hb=refs/heads/v5.0#l597

¹²https://osm.etsi.org/gitweb/?p=osm/LCM.git;a=blob;f=osm_lcm/ns.py;h=83e7147234c66cb348f1c78877fd602781b1b987;hb=refs/heads/v5.0#l1125

¹³https://osm.etsi.org/gitweb/?p=osm/LCM.git;a=blob;f=osm_lcm/lcm.py;hb=f609c16f40cef9c60d73c842d858ba457ab3549b#l190

functions took the most CPU and the spikes that are seen every 120 seconds in the LCM graph is caused by the `kafka_ping` function, which is used to keep the connection to the message broker active.

When the requests stop coming in, i.e. from 9:32:54, RO gradually processes all the remaining instantiation tasks and this can be seen from the graphs.

At 9:38:59, the termination requests start coming into OSM and a spike in RO and LCM can be seen from the graphs.

The MON microservice however, is responsible for continuously monitoring the health of VNFs from the VIMs. We did not profile the MON processes in this experiment.

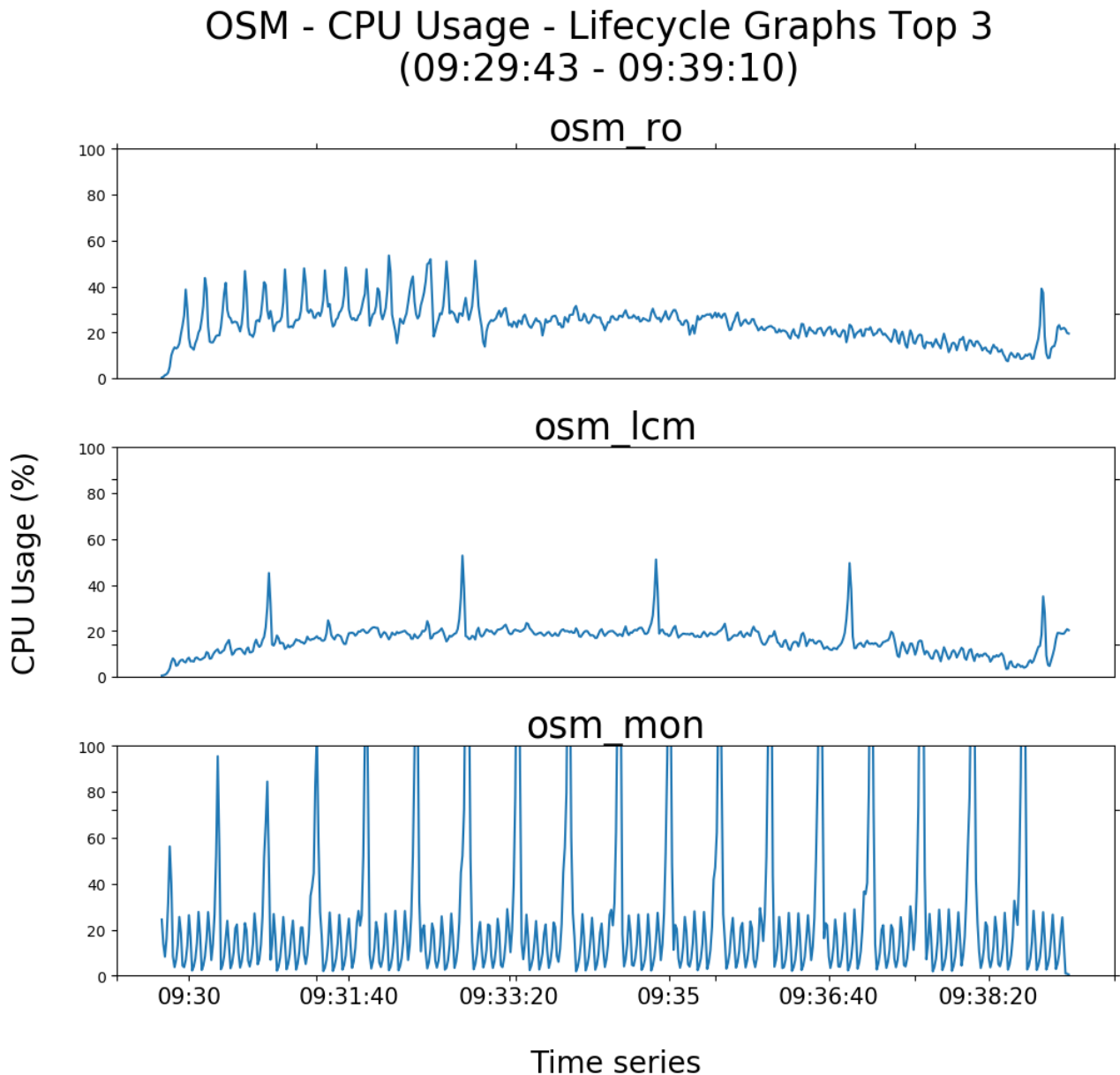


Figure 5.19: Lifecycle graph for 90 instances

Frequency Distribution of Functions

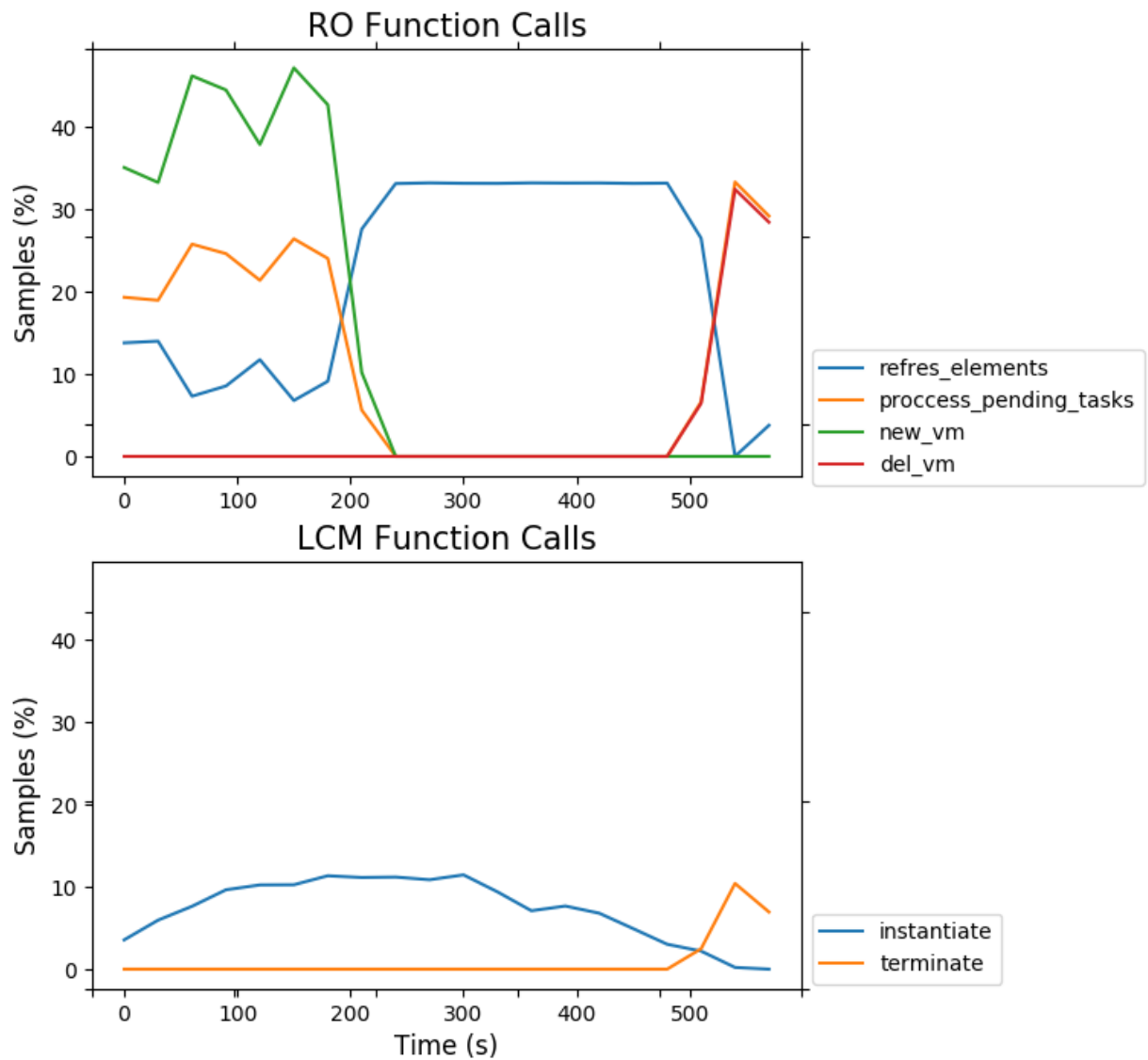


Figure 5.20: OSM Frequency distribution of functions

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 6.1. The fundamental aspect of a service deployment is the NSD, in the section 6.2 the trends and options of NSDs and research papers that try to mitigate the interoperability challenges between different MANO frameworks are discussed. Section 6.3 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.

6.1 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].

6.2 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 inter-operate by comparing the NSD templates by OpenStack (HOT¹) and OASIS (TOSCA²). 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.

6.3 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.

¹Heat orchestration template (HOT) specification:

http://docs.openstack.org/heat/rocky/template_guide/hot_spec.html

²Topology and Orchestration Specification for Cloud Applications (2013):

<http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html>

Conclusion

Pishahang-scramble integration enables the end users to instantiate a network service across multiple MANO frameworks. Each of the services developed by work-packages are functioning as per requirements and have been tested under various scenarios. The services are exposed, integrated with each other and with pishahang. The independent work packages themselves are capable of being used separately by other related projects. We believe that the work we carried out in this project will be useful for other MANO developers.

A

Appendix

A.1 Translator

A.1.1 Validation

```

1  from __future__ import print_function
2  import json
3  import sys
4  from osmdata import vnfd as vnfd_catalog
5  from osmdata import nsd as nsd_catalog
6  from pyangbind.lib.serialise import pybindJSONDecoder
7  import os
8  import logging
9  import uuid
10 import jsonschema import *
11 import networkx as nx
12 import errno
13 import yaml
14 import event
15 from storage import DescriptorStorage
16 from until import read_descriptor_files, list_files, strip_root, build_descriptor_id
17 import pprint
18 import time
19
20
21 log = logging.getLogger(__name__)
22 evtlog = event.get_logger('validator.events')
23 storage = DescriptorStorage()
24
25 class validator():
26
27     def osm_validator(self, descriptor):
28         if 'nsd:nsd-catalog' in descriptor:
29             descriptor_to_validate = descriptor
30             with open(r"osm-schema-nsd.json") as f:
31                 schema = json.load(f)
32                 v = Draft4Validator(schema, format_checker=FormatChecker())
33                 lastidx = 0
34                 for idx, err in enumerate(v.iter_errors(descriptor_to_validate), 1):
35                     lastidx = idx
36                     if idx == 1:
37                         print("\tSCHEMA ERRORS:")
38                         print("\t{0}. {1}\n".format(idx, pprint.pformat(err)))
39                         print("\t{0}. {1}\n".format(idx, pprint.pformat(err.path)))
40                     self.osm_dep_validator(descriptor_to_validate)
41                 if lastidx == 0:
42                     return True
43
44         elif 'vnfd:vnfd-catalog' in descriptor:
45             descriptor_to_validate = descriptor
46             with open(r"osm-vnfd-schema.json") as f:
47                 schema = json.load(f)
48                 v = Draft4Validator(schema, format_checker=FormatChecker())
49                 lastidx = 0
50                 for idx, err in enumerate(v.iter_errors(descriptor_to_validate), 1):
51                     lastidx = idx
52                     if idx == 1:
53                         print("\tSCHEMA ERRORS:")
54                         print("\t{0}. {1}\n".format(idx, pprint.pformat(err)))
55                         print("\t{0}. {1}\n".format(idx, pprint.pformat(err.path)))
56                     self.osm_dep_validator(descriptor_to_validate)

```

```

57     if lastidx == 0:
58         return True
59     else:
60         print("This is not a valid OSM descriptor")
61
62
63     def sonata_nsd_validate(self, descriptor, vnfd = None):
64         if 'network_functions' and "descriptor_version" in descriptor:
65             descriptor_to_validate = descriptor
66             with open(r"nsd-Pishahang.yml") as f:
67                 schema = yaml.load(f)
68                 v = Draft4Validator(schema, format_checker=FormatChecker())
69                 lastidx = 0
70                 for idx, err in enumerate(v.iter_errors(descriptor_to_validate), 1):
71                     lastidx = idx
72                     if idx == 1:
73                         print("\tSCHEMA ERRORS:")
74                         print("\t{0}. {1}\n".format(idx, pprint.pformat(err)))
75                         print("\t{0}. {1}\n".format(idx, pprint.pformat(err.path)))
76                     if lastidx == 0:
77                         return True
78
79         elif "network_functions" in descriptor:
80             service = storage.create_service(descriptor)
81             if not service:
82                 evtlog.log("Invalid service descriptor",
83                     "Failed to read the service descriptor of file '{0}'"
84                     .format(descriptor),
85                     descriptor,
86                     'evt_service_invalid_descriptor')
87             return
88             self.validate_service_syntax(descriptor)
89             if not vnfd:
90                 print("service integrity is not validated as there is no vnfd")
91             elif vnfd != {}:
92                 self.validate_service_integrity(service, vnfd)
93             return True
94         else:
95             print("This is not a valid sonata or Pishahang descriptor")
96
97
98     def sonata_vnfd_validate(self, descriptor):
99         if "virtual_deployment_units" and "descriptor_version" in descriptor:
100             descriptor_to_validate = descriptor
101             with open(r"vnfd-pishahang.yml") as f:
102                 schema = yaml.load(f)
103                 v = Draft4Validator(schema, format_checker=FormatChecker())
104                 lastidx = 0
105                 for idx, err in enumerate(v.iter_errors(descriptor_to_validate), 1):
106                     lastidx = idx
107                     if idx == 1:
108                         print("\tSCHEMA ERRORS:")
109                         print("\t{0}. {1}\n".format(idx, pprint.pformat(err)))
110                         print("\t{0}. {1}\n".format(idx, pprint.pformat(err.path)))
111                     if lastidx == 0:
112                         return True
113         elif "virtual_deployment_units" in descriptor:
114             func = storage.create_function(descriptor)
115             if not func:
116                 evtlog.log("Invalid function descriptor",
117                     "Couldn't store VNF of file '{0}'".format(descriptor),

```

```
118 | descriptor ,
119 | 'evt_function_invalid_descriptor')
120 | return
121 |
122 | self.validate_function_syntax(descriptor)
123 | self.validate_function_integrity(func)
124 | self.validate_function_topology(func)
125 | return True
126 | else:
127 | print("This is not a valid sonata or Pishahang descriptor")
```

Listing A.1: "Validating Sonata Pishahang and OSM descriptors (Both NSD and VNFD's)"

A.2 Splitter

A.2.1 Pishahang

[Docb]

Table A.1: Schema Parameters of Pishahang Considered for Splitting

Parameter	Description
descriptor_version	The version of the description definition used to describe the network descriptor.
vendor	The vendor id allows to identify a VNF descriptor uniquely across all function descriptors.
name	The name of the network service description.
version	The version of the service descriptor.
author	The person or organization that created the NS descriptor.
description	A longer description of the network service.
networkFunctions	The VNFs (their descriptors), that are part of this network service.
connectionPoints	The connection points of the overall NS, that connects the NS to the external world.
virtualLinks	One to One link between two VNFs.
forwardingGraphs	The forwarding graph describing the topology of the network.

A.2.2 OSM

[Doca]

Table A.2: Schema Parameters of OSM Considered for Splitting

Parameter	Description
__id	Identifier for the NSD.
name	NSD name.
short_name	Short name to appear as label on the UI
description	Description of the NSD
vendor	Vendor of the NSD
version	Version of the NSD
connection_point	List for external connection points. Each NS has one or more external connections.
logo	File path for the vendor specific logo
ConstituentVnfd	List of VNFDs that are part of this network service.
scaling __group __descriptor	scaling group descriptor within this network service. The scaling group defines the scaling characteristics.
vnffgd	List of VNF Forwarding Graph Descriptors (VNFFGD).
vld	Virtual Link Descriptor
ip __profiles	List of IP Profiles. IP Profile describes the IP characteristics for the Virtual-IP.

A.3 Scaling Plugin

A.3.1 Pishahang

```

1 ---
2 descriptor_version: '1.0'
3 vendor: pg-scramble
4 name: Pishahang-Service
5 version: '1.0'
6 author: pg-scramble
7 description: 'Pishahang Instance'
8 network_functions:
9     -
10         vnf_id: pishahang-instance
11         vnf_vendor: pg-scramble
12         vnf_name: pishahang-instance
13         vnf_version: '1.0'

```

Listing A.2: Pishahang MANO instance NSD

```

1 ---
2 descriptor_version: vnfd-schema-01
3 description: 'Pishahang Instance'
4 name: pishahang-instance
5 vendor: pg-scramble
6 version: '1.0'
7 author: pg-scramble
8 virtual_deployment_units:
9     -
10         id: pishahang-instance
11         description: 'Pishahang Instance'
12         vm_image: pishahang-instance
13         vm_image_format: qcow2
14         resource_requirements:
15             cpu: {vcpus: 2}
16             memory: {size: 1, size_unit: GB}
17             storage: {size: 1, size_unit: GB}
18         connection_points:
19             - {id: eth0, interface: ipv4, type: external}
20
21 # Image name on openstack should be "pg-scramble_pishahang-
    instance_1.0_pishahang-instance"

```

Listing A.3: Pishahang MANO instance VNFD

A.3.2 OSM

```

1 ---
2 descriptor_version: '1.0'
3 vendor: pg-scramble
4 name: OSM-Service
5 version: '1.0'
6 author: pg-scramble
7 description: 'OSM Instance'
8 network_functions:
9   -
10     vnf_id: osm-instance
11     vnf_vendor: pg-scramble
12     vnf_name: osm-instance
13     vnf_version: '1.0'

```

Listing A.4: OSM MANO instance NSD

```

1 ---
2 descriptor_version: vnfd-schema-01
3 description: 'OSM Instance'
4 name: osm-instance
5 vendor: pg-scramble
6 version: '1.0'
7 author: pg-scramble
8 virtual_deployment_units:
9   -
10     id: osm-instance
11     description: 'OSM Instance'
12     vm_image: osm-instance
13     vm_image_format: qcow2
14     resource_requirements:
15       cpu: {vcpus: 2}
16       memory: {size: 1, size_unit: GB}
17       storage: {size: 1, size_unit: GB}
18     connection_points:
19       - {id: eth0, interface: ipv4, type: external}
20
21 # Image name on openstack should be "pg-scramble_osm-instance_1
    .0_osm-instance"

```

Listing A.5: OSM MANO instance VNFD

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.
- [Doca] OSM Documentation. *OSM Schema Documentation*.
- [Docb] SONATA Documentation. *SONATA Schema Documentation*.
- [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.
- [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.