



SUPERFLUIDITY

a super-fluid, cloud-native, converged edge system

Research and Innovation Action GA 671566

Deliverable D6.1:

System Orchestration and Management Design and Implementation

DELIVERABLE TYPE:	REPORT
DISSEMINATION LEVEL:	PU
CONTRACTUAL DATE OF DELIVERY TO THE EU:	31 JAN 2018
ACTUAL DATE OF DELIVERY TO THE EU:	31 MARCH 2018
WORKPACKAGE CONTRIBUTING TO THE DELIVERABLE:	WP6
EDITOR(S):	LUIS TOMAS (RED HAT) DANIEL MELLADO (RED HAT)
AUTHOR(S):	CARLOS PARADA, ISABEL BORGES, FRANCISCO FONTES (ALTICE LABS), GEORGE TSOLIS (CITRIX), MICHAEL MCGRATH, VINCENZO RICCOBENE (INTEL), JOHN THOMSON, JULIAN CHESTERFIELD, JOEL ATHERLEY, MANOS RAGIADAKOS (ONAPP), LUIS TOMAS, LIVNAT PEER, DANIEL MELLADO (RED HAT), EREZ BITON (ALU-IL), CLAUDIO PISA, FRANCESCO LOMBARDO, STEFANO SALSANO, LUCA CHIARAVIGLIO, MOHAMMAD SHOJAFAR, LAVINIA AMOROSI (CNIT), COSTIN RAICIU, RADU STOENESCU, MATEI POPOVICI, DRAGOS DUMITRESCU (UPB)
Internal Reviewer(s)	STEFANO SALSANO, MARIA BIANCO (CNIT)



Abstract: This deliverable reports the efforts of the three WP6 Tasks. It concludes all the efforts done at WP6, finalizing the control framework, the SLA-base deployment design, and completes the symbolic execution checking and anomaly detection tools implementation.

Keyword List: Orchestration, Management, VMs, Containers



Executive Summary

The document provides the results of all the work carried out by the WP6 of the Superfluidity Project. The WP6, named *“System Orchestration and Management Tools”*, focused on orchestration and management actions taken at a distributed system level in the Superfluidity architecture. The WP6 activities were split in three tasks, namely T6.1 *“Provisioning and Control Framework”*, T6.2 *“Access-Agnostic SLA-Based Network Service Deployment”*, T6.3 *“Automated Security Verification Framework”*. This single deliverable reports on all the activities and the results of the three tasks for the whole project duration. In order to help the reader to understand the main results without being overwhelmed by the details, we have structured this deliverable as follows. The initial part of the document (around 50 pages from section 1 to section 5) provides a comprehensive overview of the results. This should be detailed enough to understand the value of the contributions and how these contributions are related to the Superfluidity architecture and in general to the evolution of 5G networking. Then we provide three annexes, one for each task, gathering the detailed reports on the activities performed in the project lifetime and on the achieved results. In these annexes, we also include information that was presented in the previous internal deliverables, considering that this document is the only public deliverable of the WP6.



INDEX

EXECUTIVE SUMMARY	3
GLOSSARY	11
1 INTRODUCTION	13
2 SYSTEM ORCHESTRATION AND MANAGEMENT TOOLS ARCHITECTURE	14
2.1 VIM OPTIONS: OPENSTACK, KUBERNETES AND KURYR	15
2.2 VNFMS	16
2.3 NFVO	17
2.4 DEPLOYMENT	17
3 TASK 6.1: PROVISIONING AND CONTROL FRAMEWORK	21
3.1 REQUIREMENTS ANALYSIS AND STATE OF THE ART	21
3.2 PROVISION AND CONTROL ARCHITECTURE DESIGN	21
3.3 KURYR	22
3.3.1 Nested containers on VMs without double encapsulation	22
3.3.2 Ports Pool Optimisation	23
3.3.3 Load Balancer Integration	23
3.3.4 Integration with Exemplar SDNs: OVN, ODL, DragonFlow	23
3.3.5 CNI-Split for performance improvement	24
3.3.6 Containerized Components (controller and CNI)	24
3.3.7 RDO-packaging + Upstream CI	24
3.4 LOAD BALANCING AND SERVICE FUNCTION CHAINING (SFC)	25
3.5 OSM EVALUATION AND INTEGRATION	26
3.6 MANAGEIQ AS A NVFO	27
3.6.1 Ansible execution support	27
3.6.2 Multi-site support	27
3.7 OPENSIFT-ANSIBLE	28
3.7.1 Integration with ManageIQ	28
3.7.2 Baremetal containers support	28
3.8 RDCL 3D	28
3.9 SERVICE CHARACTERISATION FRAMEWORK AND DEPLOYMENT TEMPLATE OPTIMISATION	29
4 TASK 6.2: ACCESS-AGNOSTIC SLA-BASED NETWORK SERVICE DEPLOYMENT	31
4.1 SLA-BASED DESCRIPTORS	31



4.1.1	NEMO modeling language	31
4.1.2	Nested execution environments	32
4.2	CROSS MANAGEMENT DOMAINS RESOURCE ALLOCATION AND PLACEMENT	32
4.2.1	Data center resource Allocation and Placement.....	32
4.2.2	The Operational Cost of Switching.....	35
4.2.3	Optimized Operation Cost Placement	37
4.2.4	Resource allocation in Mobile Edge Computing	39
4.3	DYNAMIC SCALING, RESOURCE ALLOCATION AND LOAD BALANCING	41
4.3.1	VM Scaling and Scheduling via Cost Optimal MDP solution	41
4.3.2	Load Balancing as a Service	42
4.4	OPTIMAL DESIGN AND MANAGEMENT OF RFBs OVER A SUPERFLUID 5G NETWORK	42
5	TASK 6.3: AUTOMATED SECURITY VERIFICATION FRAMEWORK.....	44
5.1	VERIFYING HIGH-LEVEL SERVICE CONFIGURATIONS	44
5.1.1	Translating RFBs to SEFL	45
5.1.2	Verifying policy compliance of SEFL dataplanes using Symnet and NetCTL.....	46
5.2	VERIFYING LOW-LEVEL IMPLEMENTATIONS	48
5.2.1	Finding bugs in P4 programs	49
5.2.2	Symbolic execution equivalence and its applications	51
5.3	ANOMALY DETECTION.....	53
6	COLLABORATION WITH 5G-PPP	54
	ANNEX A: PROVISION AND CONTROL FRAMEWORK - TASK 6.1.....	55
A1.	REQUIREMENTS ANALYSIS.....	56
A1.1	NFV TECHNICAL REQUIREMENTS	56
A1.2	MEC TECHNICAL REQUIREMENTS.....	61
A1.3	C-RAN TECHNICAL REQUIREMENTS.....	64
A1.4	NFV vs. MEC COMPARISON.....	67
A2.	STATE OF THE ART	69
A2.1	VIM OPENSTACK VIRTUAL INFRASTRUCTURE MANAGEMENT	69
A2.2	VNFM/NFVO.....	70
A2.2.1	Cloudband	70
A2.2.2	OpenMano	73
A2.2.3	Open Baton	73
A2.2.4	OSM.....	78
A2.2.5	Cloudify	81



A2.2.6	Tacker	85
A2.2.7	ManageIQ	87
A2.2.8	Evaluation	88
A2.3	COMPARISON BETWEEN ORCHESTRATORS	88
A2.4	MANAGEMENT AND ORCHESTRATION DESIGN	89
A2.4.1	Cloud Infrastructure	89
A2.4.2	Cloud Infrastructure Management	91
A2.4.3	Cloud Management and Orchestration.....	94
A2.4.4	Orchestration Layer	96
A3.	SUPERFLUIDITY CONTRIBUTIONS.....	98
A3.1	KURYR	98
A3.1.1	Side by side OpenStack and OpenShift/Kubernetes deployment.....	100
A3.1.2	Nested deployment: OpenShift/Kubernetes on top of OpenStack	101
A3.1.3	Ports Pool Optimization	103
A3.1.4	Load Balancer as a Service (LBaaS) Integration	106
A3.1.5	Support for different Software Defined Networks (SDNs).....	108
A3.1.6	CNI Split.....	110
A3.1.7	Containerized Kuryr Components.....	111
A3.1.8	RDO Package and CI.....	112
A3.2	MISTRAL ORCHESTRATION	116
A3.3	OSM	117
A3.4	MANAGEIQ	119
A3.4.1	Ansible execution support.....	120
A3.4.2	Multi-site support.....	122
A3.5	LOAD BALANCING AS A SERVICE	122
A3.6	SERVICE FUNCTION CHAINING	124
A3.6.1	Service Function Chaining with IPv6 Segment Routing (SRv6)	126
A3.7	OPENSIFT-ANSIBLE.....	126
A3.7.1	ManageIQ integration	127
A3.7.2	Baremetal Support	127
A3.8	RDCL 3D.....	129
A3.8.1	Integration between RDCL 3D and ManageIQ.....	131
A3.8.2	Software Architecture	132
A3.9	OPTIMIZATION OF SERVICE DEPLOYMENT TEMPLATES USING A SERVICE CHARACTERISATION FRAMEWORK	134



A3.9.1	Service On-Boarding Characterisation	135
A3.9.2	Characterisation Lifecycle	136
A3.9.3	Exploration of the Deployment Configuration Space for a Virtualised Media Processing Function 139	
A3.9.4	Generalised and Automated Generation of Optimised Service Deployment Templates....	143
A3.9.5	Service Characterisation Framework Implementation	146
A3.9.6	Automated Methodology Results.....	148
A3.9.7	Superfluidity System Integration	151
A3.10	MICROVISOR ORCHESTRATION	152
A3.10.1	UI design for managing a large collection of resources	152
ANNEX B: ACCESS-AGNOSTIC SLA-BASED NETWORK SERVICE DEPLOYMENT – TASK 6.2		157
B1	NEMO ENHANCEMENTS – IMPLEMENTATION DETAILS	158
B2	SUPPORT FOR HETEROGENEOUS AND NESTED EXECUTION ENVIRONMENTS.....	159
B2.1	NOTES ON KUBERNETES NESTING.....	161
B3	CORE DATA-CENTER PLACEMENT	164
B4	MOBILE EDGE COMPUTING	165
B4.1	PLACEMENT	165
B4.2	SERVICE MIGRATION & MOBILITY	166
B4.2.1	Mobility in MEC scope.....	166
B4.2.2	MEC relocation types	168
B4.2.3	UE’s mobility detection	169
B4.2.4	Relocation need detection	169
B4.2.5	Proposed processes	170
B4.2.6	Mobility API.....	171
B5	OPTIMAL SCALING AND LOAD BALANCING BASED ON MDP MODELS	172
B5.1	INTRODUCTION.....	172
B5.1.1	Numerical results	174
B5.2	MACHINE LEARNING TECHNIQUES FOR THE LCM FOR CONTAINERIZED WORKLOAD.....	176
B5.2.1	Introduction.....	176
B5.2.2	Kubernetes scaling mechanism.....	176
B5.2.3	Machine learning based scaling	176
B5.2.4	Offline implementation – video streaming application	177
B5.2.5	Implementation results	179
B6	LOAD BALANCING AS A SERVICE	180



B6.1	LOAD BALANCING PRINCIPLES	180
B6.2	HA PROXY AND LBAAS IN OPENSTACK	180
B6.3	CITRIX NETSCALER ADC AND MAS	180
B6.4	NETSCALER MAS INSTALLATION	182
B6.5	NETSCALER DRIVER SOFTWARE INSTALLATION	183
B6.6	REGISTERING OPENSTACK WITH NETSCALER MAS	183
B6.7	ADDING OPENSTACK TENANTS IN NETSCALER MAS	184
B6.8	PROVISIONING NETSCALER VPX INSTANCE IN OPENSTACK	184
ANNEX C: AUTOMATED SECURITY VERIFICATION FRAMEWORK – TASK 6.3		186
C1	DEBUGGING P4 PROGRAMS WITH VERA	186
C2	EQUIVALENCE AND ITS APPLICATIONS TO NETWORK VERIFICATION	187
REFERENCES		188
PAPER – 1: OPTIMIZING NFV CHAIN DEPLOYMENT THROUGH MINIMIZING THE COST OF VIRTUAL SWITCHING..		191
PAPER – 2: DEBUGGING P4 PROGRAMS WITH VERA.....		192
PAPER – 3: EQUIVALENCE AND ITS APPLICATIONS TO NETWORK VERIFICATION.....		193
ANNEX: INTERNAL DELIVERABLE I6.3		194
ANNEX: INTERNAL DELIVERABLE I6.3B.....		195



List of Figures

Figure 1: NFV Overview	14
Figure 2: Superfluidity Orchestration Framework	15
Figure 3: Multi-site Deployment	18
Figure 4: VMs and Containers mix example	20
Figure 5: Example of possible deployment of four service chains on top of three	34
Figure 6: Server S_j deployed with r sub-chains from possibly different service chains	36
Figure 7: Optimal service chain placement step	38
Figure 8 - ETSI MEC Architecture	39
Figure 9: Symnet dataplane verification	45
Figure 10: ETSI NFV reference architecture	56
Figure 11: ETSI NFV MANO reference architecture	57
Figure 12: ETSI MEC reference architecture	61
Figure 13: Affinity graph between different C-RAN functional blocks	65
Figure 14: Openstack based generic VNF management system	71
Figure 15: VNF lifecycle operation	71
Figure 16: The deployment workflow	72
Figure 17: OpenBaton integration into OpenStack	74
Figure 18: Open Baton architecture	74
Figure 19: OSM Architecture	79
Figure 20: Cloudify architecture	82
Figure 21: Cloudify components integration	83
Figure 22: Tacker Architecture	85
Figure 23: One NFVI per service	90
Figure 24: Common NFVI from all services	91
Figure 25: One local VIM per NFVI	92
Figure 26: Centralized VIM for all NFVIs	93
Figure 27: Hybrid Option	94
Figure 28: Single orchestrator	95
Figure 29: One orchestrator per service	95
Figure 30: Top orchestrator	96
Figure 31: east-west orchestrator	97
Figure 32: Hybrid orchestrator	97
Figure 33: Kuryr Baremetal	98
Figure 34: Kuryr Nested	98
Figure 35: Kuryr VIF-Binding	99
Figure 36: Double encapsulation problem	100
Figure 37: Kuryr components integration	100



Figure 38: Sequence diagram: Pod creation	101
Figure 39: Time from pod creation to running status	105
Figure 40: Sequence diagram: nested pod creation.....	107
Figure 41: Sequence diagram: Service (LBaaS) creation	108
Figure 42: Sequence Diagram: Pod creation with CNI Daemon (Split).....	110
Figure 43: RDO packaging process	113
Figure 44: Upstream CI workflow	115
Figure 45: Onboarding of MEC Apps on OSM.....	118
Figure 46: Ansible playbook execution.....	121
Figure 47: ManageIQ State Machine	121
Figure 48: Muti-Site playbook execution	122
Figure 49: NetScaler ADC at NFV architecture.....	123
Figure 50: OpenShift-Ansible integration.....	127
Figure 51: Provision Interactions Overview	128
Figure 52: Network Service design using the RDCL 3D GUI	130
Figure 53: Positioning RDCL 3D in the ETSI MANO architecture	131
Figure 54: RDCL 3D Software Architecture	133
Figure 55: Characterisation Phases	136
Figure 56: Experimental Configuration	140
Figure 57: Throughput results for different configurations.....	141
Figure 58: Latency results for different configurations.....	142
Figure 59: Template Optimisation Methodology Pipeline	145
Figure 60: Service Characterisation Framework high-level architecture.	147
Figure 61: Visualization UI.....	153
Figure 62: Mock-up diagram showing a UI that relates virtual to physical resources.....	153
Figure 63: Mock-up diagram showing the rack utilization.....	154
Figure 64: Mock-up diagram showing the storage utilization in the management UI	155
Figure 65: Mock-up showing the network planner UI.....	156
Figure 66: Impact of the allocated VNF cost.....	174
Figure 67: Impact of delay cost.....	175
Figure 68: setup topology	177
Figure 69: packet errors	179
Figure 70: Throughput measurements	179
Figure 71: NetScaler LBaaS Integration	182
Figure 72: NetScaler MAS - OpenStack Integration Workflow [36]	184



Glossary

SUPERFLUIDITY DICTIONARY	
TERM	DEFINITION
API	Application Programming Interface
CNI	Container Network Interface
DNS	Domain Name System
EC2	Elastic Compute Cloud
EPC	Enhanced Packet Core
ETSI	European Telecommunications Standards Institute
GTP	GPRS Tunneling Protocol
GUI	Graphical User Interface
HOT	Heat Orchestration Template
HTTP	Hypertext Transfer Protocol
ISG	Industry Specification Group
JSON	Javascript Object Notation
KPI	Key Performance Indicator
KVM	Kernel-based Virtual Machine
LBaaS	Load Balancing as-a-Service
LB	Load Balancer
LTE	Long Term Evolution
MAC	Media Access Control
MANO	Management and Orchestration
MEC	Multi-access Edge Computing
MEO	MEC Orchestrator
NAT	Network Address Translation
NEMO	Network Modeling
NFV	Network Function Virtualization
NIC	Network Interface Controller
NFV	Network Function Virtualization
NFVO	NFV Orchestrator



NFVI	NFV Infrastructure
NS	Network Service
NSD	Network Service Descriptor
OSM	Open Source MANO
OSS	Operation Support Systems
OvS	Open virtual Switch
QoS	Quality of Service
RAN	Radio Access Network
RDCL	RFB Description and Composition Language
REST	Representational State Transfer
REE	RFB Execution Environment
RFB	Reusable Functional Blocks
SDN	Software Defined Networking
SEFL	Symbolic Execution Friendly Language
SLA	Service Level Agreement
SSH	Secure Shell
TOF	Traffic Offloading Function
TOSCA	Topology and Orchestration Specification for Cloud Applications
UE	User Equipment
URI	Uniform Resource Identifier
VDU	Virtual Deployment Unit
VLAN	Virtual Local Area Network
VIM	Virtual Infrastructure Manager
VIP	Virtual IP
VNF	Virtualized Network Function
VNFD	VNF Descriptor
VNFM	VNF Manager
VM	Virtual Machine
YAML	YAML Ain't Markup Language

Table 1: SUPERFLUIDITY Dictionary.



1 Introduction

WP6 focuses on the orchestration and management actions at a distributed system level, building upon WP5 advances. The WP main objectives are resource provisioning, and control and management of network functions as well as applications located at the edge (in this case MEC). To achieve this we propose a framework that targets dynamic scaling and resource allocation, traffic load balancing between virtual functions, or automatic recovery upon hardware failure, among others.

The main objectives from the DoW that the WP's tasks target are:

- OBJ1: design of SLA based network function descriptors
- OBJ2: design of cross management domains resource allocation and placement algorithms
- OBJ3: design the control framework for dynamic scaling, resource allocation and load balancing of tasks in the overall system
- OBJ4: development of platform middleboxes and services
- OBJ5: design of a security framework that allows efficient enforcement of operator policies

There are several points where Superfluidity have contributed/focused to achieve these goals:

- Service behaviour models to support autonomous policy management reacting to current status of the system. For instance, detecting an application having noisy neighbor problems and reacting to it by either performing (QoS) bandwidth limitation, migration or load balancing actions.
- Make OpenStack suitable for C-RAN/MEC components by improving network performance as well as allowing mixed VM and Container environments as well as the possibility of running containers inside VMs or on baremetal nodes.
- Management and Placement in a distributed environment with a system-wide overview as well as with synchronization between the different sites.
- Security framework that enables networking policy assurance checkings, as well as bugs finding.

The document is structured as follows. First an overview of the proposed architecture and its components is presented in section 2. After that, the sections 3, 4 and 5 highlight the main achievements of tasks 6.1, 6.2 and 6.3, respectively. On top of that, there are 3 annexes (1 for each task) attached to present with more details the work carried out as part of each task, as well as the information presented in the previous internal deliverables.



2 System Orchestration and Management Tools Architecture

Some major international operators and vendors started the ETSI ISG (Industry Specification Group) on Mobile Edge Computing (MEC), recently re-branded as “Multi-access Edge Computing”. The MEC paradigm advocates for the deployment of virtualized network services on distributed access infrastructures, which are placed next to base stations and aggregation points, and run on x86 commodity servers. In other words, MEC enables services to run at the edge of the network, so that they can benefit from higher bandwidth and low latency.

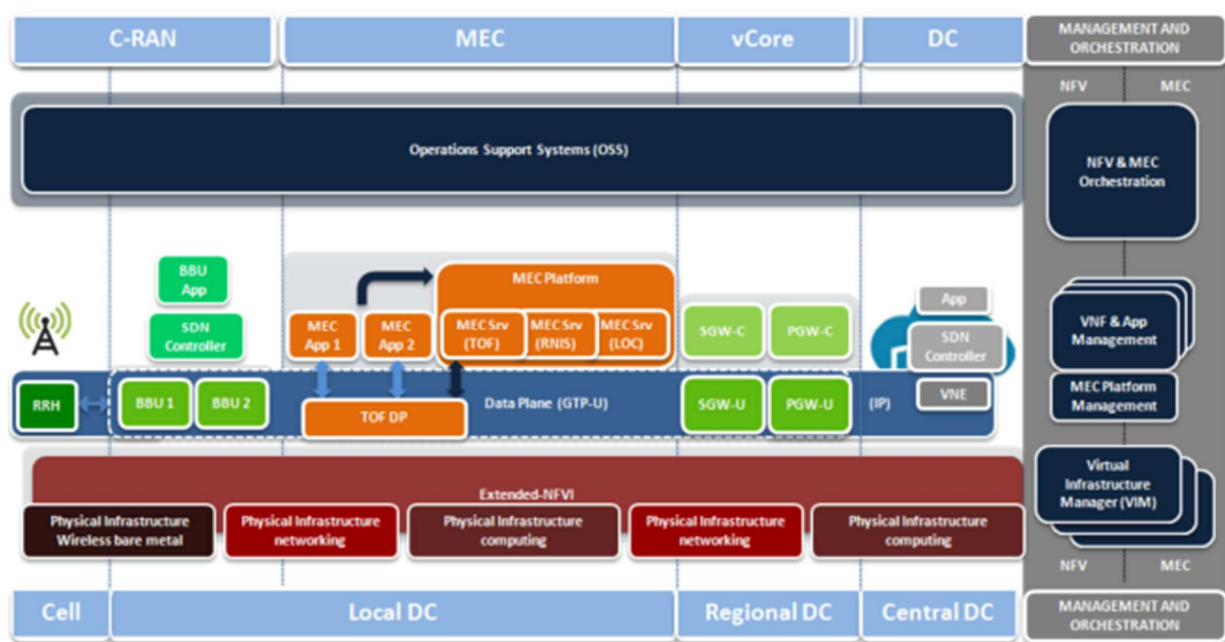


Figure 1: NFV Overview

In such scenario, some network services and applications may be possibly deployed in a specific edge of the network (MEC App and MEC Srv in orange boxes in the above figure). This creates new challenges. On the one hand, the network services reaction to current situations (e.g., spikes in the amount of traffic handled by some specific VNFs/NS) needs to be extremely fast. The application lifecycle management, including instantiation, migration, scaling, and so on, needs to be quick enough to provide a good user experience. On the other hand, the amount of available computational resources at the edge is notably limited when compared to central data centers. Therefore, they must be used efficiently, which results in careful planning of virtualization overheads (time-wise and resource-wise).

Based on the current status of available upstream components (OpenStack, Kubernetes, OSM, ManageIQ, ODL, etc.), the requirement analysis, and the management and orchestration study carried out (see **Annex A, Section A2**, specially section A2.4), we think VMs may not always be a proper



approach for all the needs. Instead, other solutions such as the unikernel VMs and containers should be used. So, we forecast a mixed containers and VMs scenario, at least for the following years. Note, even though there is a high interest in moving more and more functionality to containers over the next years, the priority so far is still set on new applications rather than the legacy ones. And not all the applications will/can be migrated at the same time. On top of that, there is a belief that containers and virtualization are essentially the same thing, while they are not. Although they have a lot in common, they have some differences too. They should be seen as complementary, rather than competitive technologies. For example, VMs can be a perfect environment for running containerized workload (it is already fairly common to run Kubernetes or OpenShift on top of OpenStack VMs), providing a more secure environment for running containers, as well as higher flexibility and even improved fault tolerance, and also taking advantage of accelerated application deployment and management through containers. This is commonly referred to as "nested containers".

Considering the gathered requirements and the state of the art revision, as well as this blend of VMs and Containers, we have proposed and evolved at Superfluidity the next orchestration framework composed of different components at the different actuation levels:

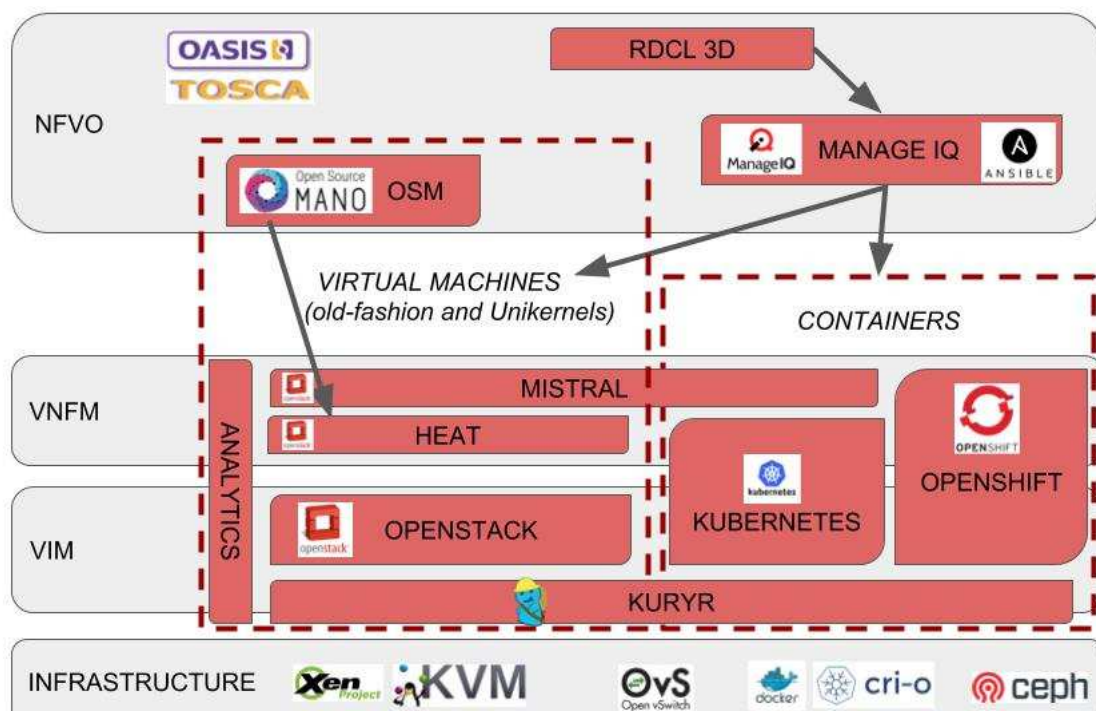


Figure 2: Superfluidity Orchestration Framework

2.1 VIM Options: OpenStack, Kubernetes and Kuryr

Both OpenStack and Kubernetes are the most commonly used VIMs for VMs and Containers, respectively. In addition, Openshift is another well-known framework for container management, leveraging kubernetes power to provide extra dev-ops functionality.



To provide a common infrastructure for both VMs and containers, the problem is not just how to create computational resources, be it VMs or containers, but also how to connect these computational resources among themselves and to the users, in other words, networking. Regarding the VMs in OpenStack, the Neutron project already has a very rich ecosystem of plug-ins and drivers which provide the networking solutions and services, like load-balancing-as-a-service (LBaaS/Octavia), virtual-private-network-as-a-service (VPNaaS) and firewall-as-a-service (FWaaS). By contrast, in container networking there is no standard networking API and implementation. So each solution tries to reinvent the wheel — overlapping with other existing solutions. This is especially true in hybrid environments including blends of containers and VMs. As an example, OpenStack Magnum had to introduce abstraction layers for different libnetwork drivers depending on the Container Orchestration Engine (COE).

Therefore, there is a need to further advance in the container networking and its integration in the OpenStack environment. To accomplish this, we have used and worked on a recent project in OpenStack named **Kuryr**, which tries to leverage the abstraction and all the hard work previously done in Neutron, and its plug-ins and services. In a nutshell, Kuryr aims to be the “integration bridge” between the two communities, containers and VMs networking, avoiding that each Neutron plug-in or solution needs to find and close the gaps independently. Kuryr allows to map the container networking abstraction to the Neutron API, enabling the consumers to choose the vendor and keep one high quality API free of vendor lock-in, which in turn allows to bring container and VM networking together under one API. So all in all, it allows:

- A single community sourced networking whether you run containers, VMs or both
- Leveraging vendor OpenStack support experience in the container space
- A quicker path to Kubernetes & OpenShift for users of Neutron networking
- Ability to transition workloads to containers/microservices at your own pace.

2.2 VNFM

To manage VM-base VNFs, we used and extended two already available OpenStack components. We use HEAT to make the deployment actions through templates. And these templates are managed by another OpenStack component designed for that end, named Mistral, to be able to adapt to the given workflows. Besides this, an analytics module is developed as part of Superfluidity to gather information about the VMs performance and build models based on that. This models can later be used to generate optimized versions of the HEAT templates that will better maintain their QoS needs.



As for container based VNFs, there are different options. Kubernetes itself already provides certain VNF management functionality. On top of that, for kubernetes deployments on top of OpenStack VMs, we could make use of the Magnum OpenStack component, that provides extra capabilities for container management.

On the other hand, OpenShift already has other extra container management functionalities on top of kubernetes that can be used through its API, such as Source-to-Image (S2I) that allows users to build their containers/applications directly from git repositories, and even push new containers everytime a new commit gets merge on the git repository -- also enabling easy rollback in case something went wrong. Moreover, as OpenShift leverages Kubernetes functionality, the management capabilities available in kubernetes can be used on Openshift deployments directly.

2.3 NFVO

Finally, in the upper level, we may make use of different NFV orchestrators. There is no complete solution yet, and, as detailed before in this document, different options have been studied. Among them, we decided to make Superfluidity project to target the 2 most promising ones. On the one hand, OSM seems to have some momentum and has a large support by the NFV community. On the other hand, we decided to explore/extend ManagelQ as it is the only one capable of dealing with different VM and Container providers, which, as mentioned before, is needed for the 5G deployments. In addition, ManagelQ allows to easily handle in a single point multiple deployments, as it is the target of Superfluidity, where we can have different cloud deployments at the edges, and the core. Moreover, it provides enough flexibility to include new orchestration actions by relying on ansible playbooks to trigger/execute new required actions. Finally, there is a component named RDCL 3D at the top. This component provides an UI to create the RFB and translate them from (Superfluidity) TOSCA templates to ansible playbooks. These playbooks are uploaded to git repositories consumed by ManagelQ, and they contains the Heat templates needed to handle the OpenStack VMs (and other OpenStack resources) and kubernetes/openshift templates needed for the containers (e.g., pods, replica controller, services, ...). After they are uploaded to the selected git repository, they can be processed by the NFVOs and pushed to the lower layers in their respective template formats. Note that OSM will be deployed through ManagelQ too as a service running in one of the managed clouds.

2.4 Deployment

Once we have reviewed the components at each hierarchy level (VIM, VNFM, and NFVO), as well as the 'glue' between VMs and containers (Kuryr), it is important to highlight the different deployment



options. As highlighted in <https://ltomasbo.wordpress.com/2017/01/24/superfluidity-containers-and-vms-deployment-for-the-mobile-network-part-2>, the VM and Container deployments can be done in a side-by-side or in a nested way (or a mix of them). Some applications (MEC Apps) or Virtual Network Functions (VNFs) may need really fast scaling or spawn responses and require therefore to be run directly on bare metal deployments. In this case, they will run inside containers to take the advantage of their easy portability and the life cycle management, unlike the old-fashioned bare metal installations and configurations. On the other hand, there are other applications and VNFs that do not require such fast scaling or spawn times. On the contrary, they may require higher network performance (latency, throughput) but still retain the flexibility given by containers or VMs, thus requiring a VM with SRIOV or DPDK. Finally, there may be other applications or VNFs that benefit from extra manageability, consequently taking advantage of running in nested containers, with stronger isolation (and thus improved security), and where some extra information about the status of the applications is known (both the hosting VM and the nested containers). This approach also allows other types of orchestration actions over the applications. One example being the functionality provided by Magnum OpenStack project which allows to install Kubernetes on top of the OpenStack VMs, as well as some extra orchestration actions over the containers deployed through the virtualized infrastructure.

A multi-side deployment example fitting the Superfluidity use cases is presented in the next figure.

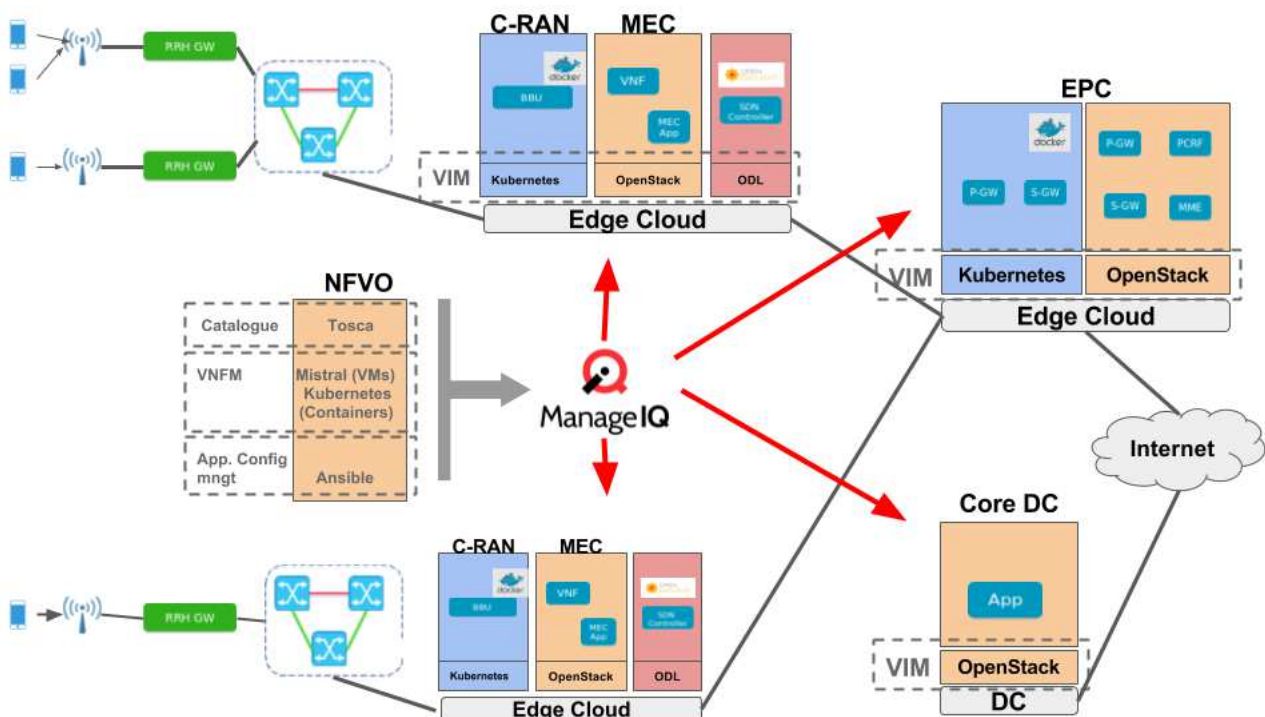


Figure 3: Multi-site Deployment



As it can be seen, there may be several edge clouds in the mobile network, with different roles, such as C-RAN and MEC (co-located in the same edge cloud), or EPC (at the edge of the mobile network). Moreover, outside of the mobile network, there will be other(s) clouds -- known as data centers. In that example, ManageIQ is the NVF orchestration (using Mistral/Heat and kubernetes as VNFMs for VMs and Containers, respectively), and it has a global view of the entire system. Note other local ManageIQ (or OSMs) could also be deployed locally or at some point of the network.

There is a VIM in all of them, but it could be different in different clouds. For instance, it is more common to have OpenStack as a VIM in the big data centers, as the virtualization overhead is not a big concern at that scale. By contrast, at the edge side, the amount of resources is more limited, but the responsiveness need to be higher. Therefore, containers are needed, but so VMs are. In such case the VIM is composed of both OpenStack and Kubernetes/OpenShift, and even SDN controllers (which may also run inside VMs, containers, or baremetal).

Next figure zooms in into one of the C-RAN/MEC edge clouds. There are some servers with Kubernetes and others with OpenStack, and yet use kuryr so that all of them can make use of the Neutron functionality. Thanks to that, some components of the Network Service (NS) or VNF can be running on containers, while others on VMs, and still have layer-2 direct connectivity. For example, this may be required by the firewall load balancer, where one part may need to be on the C-RAN side, and the other on the MEC. There are other use cases where this could also be an advantage, such as the vBRAS, where the routing component could still be a VM with some networking acceleration (DPDK, SR-IOV) for the data plane management, and have a container connected to it in charge of the control plane, i.e., changing the rules applied at the VM when performing the routing actions. Similarly, some MEC Apps, or VNFs may require to be run in containers, due to scaling requirements, or capacity limitations of the edge, while others may need to run on VMs, for instance due to not being yet containerized. This in fact is one of the main advantages of supporting both VMs and Containers: not all the MEC App/VNFs will be containerized at the same time, and some of them may not even ever be.

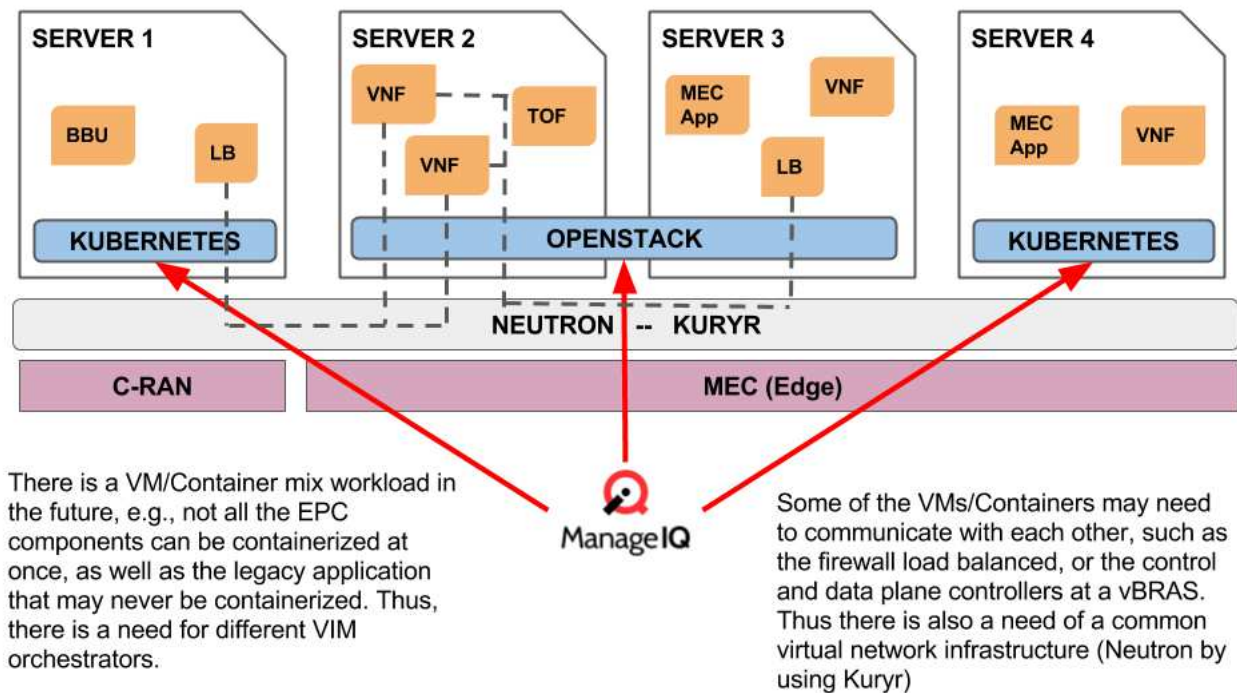


Figure 4: VMs and Containers mix example



3 Task 6.1: Provisioning and Control Framework

The main target of this task is to design and develop a provisioning and control framework and evolve its different components so that:

- it is capable of dynamically scaling network functions as well as enabling their quick instantiation,
- it can perform resource allocation across different sites,
- it enables OpenStack as a suitable VIM for NVF architectures,
- It provides high performance and low latency,
- It can create models of service behavior that can be use as a feedback for the system to improve placement/scheduling/scaling decisions

Having the above points in mind, the Superfluidity efforts have focused on different (correlated) aspects. The main achievements are highlighted next, but more details can be found on **Annex A**.

3.1 Requirements analysis and state of the art

We have gathered and analyzed the requirements for both NFV Orchestration and MEC engine as an initial step for the design of Superfluidity control and provisioning framework. After that we have research and analyzed different existing tools to decide on top of which one of them to build upon. We compared different orchestrators and tools such as OpenBaton, ManageIQ, OSM, Cloudity or Tacker. The outcome of this evaluation was to select OSM as MEC Manager and ManageIQ as NFVO as our target tools for the Superfluidity control and provisioning framework. More details about the requirements can be found at **Annex A, section A1**. The details about the state of the art research and orchestration tools comparison can be found at **Annex A, section A2**.

3.2 Provision and Control Architecture Design

After gathering the requirements and the state of the art, we iteratively designed the architecture of the Superfluidity framework, deciding on the different components that should be part of the ecosystem, as well as the the points where they can be improved to overcome some of their current limitations. The resulting architecture is presented in Figure 2.

The main points to take from this figure are the next:

- We forecast a mixed environment of VMs and containers, as VMs may be more suitable for the Core, while containers are a must at the edge due to its quicker deployment times and higher density. Yet, VMs may also be needed for isolation or supporting legacy applications. What is more, even if containers are more successful in the near future, not all the



applications will be containerized at the same time, therefore even more strong support for the mixed environment assumption.

- The previous assumption lead us to focus on the need for a mixed environment, therefore the focus on Kuryr to enable VMs and containers coexistence at the lower level (VIM level), as well as on ManagelQ at the highest orchestration level (NFVO level) for the multi site environments.
- Moreover, we have worked on enabling 2 type of deployments: containers and VMs deployment side by side (i.e., VMs running on OpenStack, and containers on Kubernetes installed on baremetal server); and nested deployments where containers run inside VMs (i.e., Kubernetes deployments on top of OpenStack VMs).
- Ultimately we have also worked on the option of having a mixed environment where containers can run both on baremetal nodes or on top of OpenStack VMs, and where the infrastructure (VIM level) is provisioned with OpenStack, i.e, OpenStack is deploying the baremetal nodes that will contain the Kubernetes/OpenShift baremetal deployment. More information about this can be found at **Annex A, section A3.7**. Note this work was presented at DevConf 2018 as a keynote talk.

3.3 Kuryr

Considering that Superfluidity project targets quick provisioning at 5G deployments we decided on the need of containers. However, as there is still need for VMs, we worked on a recent project in OpenStack named Kuryr, which provides production grade networking for container base use cases and that allows the mixed VMs and Containers environment described above. To make it more suitable for the Superfluidity purposes, the next work has been pushed upstream:

3.3.1 Nested containers on VMs without double encapsulation

In order to enable the nested container use case new bindings and controller actions were added to Kuryr. The main idea behind this is to avoid double encapsulation due to the nested environment, with a three-fold objective: 1) reduce overhead due to the double encapsulation; 2) enable an easier debugging in case of networking problems; 3) enable the option of nested containers being on the same Neutron network than other VMs, and even other containers running on baremetal. To enable this behavior we leverage on the Neutron trunk ports feature and have developed new kuryr nested drivers as well as the cni plugins (i.e., Container Networking Interface) to enable them. More information at **Annex A, section A3.1.2**.



3.3.2 Ports Pool Optimisation

Every time a container is created or deleted Kuryr makes a call to Neutron to create or remove the port used by the container (even some extra calls for the nested case). Although the time needed for this (up to a couple of seconds) is reasonable for VMs, it is not for containers. Consequently, to minimize the impact of Neutron interactions during container lifecycle management (for example containers boot up), we proposed a blueprint and the follow up development implementation to maintain a pool of pre-created Neutron resources that are ready to be used by the containers. Thus avoiding the Neutron interactions during the container boot up process (as pool repopulation is performed as a background activity), with the consequent speed up. In turns it reduces the load on the Neutron server when many containers are booted up at once, also helping to indirectly speed up other actions such as creating the load balancer needed for the kubernetes/openshift services. More information about the pools implementation can be found at **Annex A, section A3.1.3**.

3.3.2.1 Performance Evaluation

We have also evaluated the performance benefit of the pool implementation at scale, and with different SDN backends, in this case OVN and ODL. For both of them the results were similar and led to the conclusion that pools are a must for medium/big environments, achieving improvements. For more detailed information see **Annex A, section A3.1.3.1**.

3.3.3 Load Balancer Integration

In addition to the effort for supporting nested containers as well as for speeding up container booting time on OpenStack-based deployments, we have also worked on making available the OpenStack LoadBalancer components to Kubernetes/OpenShift. This means that whenever a Kubernetes/OpenShift service is created, Kuryr will create an OpenStack load balancer for it and add the pods (i.e., their associated neutron ports) as members of it. We have integrated in Kuryr both Neutron LBaaSv2 as well as the new Octavia LBaaS OpenStack project (including their 2 modes, for layer-2 and layer-3 load balancing). For more information on this check **Annex A, section A3.1.4**.

3.3.4 Integration with Exemplar SDNs: OVN, ODL, DragonFlow

In addition to the previous kuryr extensions, we also focused on supporting different SDN backends so that we can leverage kuryr functionality on OpenStack clouds regardless of their SDN selection. In addition to the standard ML2/OVS driver that comes with OpenStack out of the box, we have tested and make the needed changes to also support OVN, ODL and DragonFlow SDN controllers. In addition, we even perform our ports pool performance testing at scale with two of those SDNs: OVN and ODL. For more information see **Annex A, section A3.1.5**.



3.3.5 CNI-Split for performance improvement

Originally, Kuryr's CNI (Container Network Interface) consisted on an executable entry point that performed several functions, such as start a watcher on the pod CNI requests or handle the events until it sees the vif annotation. This makes it scale in a bad way as it would require one new https connections to the K8s API per pod. In order to solve this we have split Kuryr CNI into two components:

- CNI driver: Kuryr kubernetes integration takes advantage of the kubernetes CNI plugin and introduces Kuryr-K8s CNI Driver. Based on design decision, kuryr-kubernetes CNI Driver should get all information required to plug and bind Pod via kubernetes control plane and should not depend on Neutron.
- CNI daemon: It runs on every Kubernetes node. It is responsible for watching pod events on the node it's running on, answering calls from CNI Driver and attaching VIFs when they are ready.

As the driver would communicate via socket with the daemon and it's much more lightweight, the scaling issue gets solved by this split. More information at **Annex A, section A3.1.6**.

3.3.6 Containerized Components (controller and CNI)

It is possible to install kuryr-controller and kuryr-cni on Kubernetes as pods. This follows the approach from Kubeadm linux installation guide.

This functionality allows Kuryr to be installed as a Kubernetes component:

- Kuryr CNI as daemon set
- Kuryr Controller as a pod

This would allow to get all the benefits from a container lifecycle in Kuryr and deeper integration within Kubernetes. Also, it would allow you to test any new code change just by rebuilding the images. Further information can be seen at **Annex A, section A3.1.7**.

3.3.7 RDO-packaging + Upstream CI

In order to integrate Kuryr with RDO OpenStack distribution (to make it fully/easy available for everyone as well as facilitate testing), we have produced rpm packages to ease up the installation and testing. In addition, we have triggered upstream CI to ensure quality testing of the produced code leveraging on the OpenStack functional testing framework, Tempest.



In addition to this, we've been setting our own CI/CD system, based on what the OpenStack community has been setting up and we've developed a testing framework plugin for the upstream testing system, Tempest. More information about this is detailed at **Annex A, section A3.1.8**.

3.4 Load Balancing and Service Function Chaining (SFC)

Based on the requirements analysis of the use cases, we identified Load Balancing and Service Function Chaining (SFC) as important capabilities of the Superfluidity platform. After outlining the requirements and specifications of these two areas (table in **Annex A, section A1.1**), we have:



- Analyzed the state-of-the-art of both Load Balancing (**Annex A, section A3.5**) and SFC (**Annex A, section A3.6**), in terms of adoption by the open source projects (OpenStack, OPNFV and OVS).
- On the Load Balancing topic, invested in the Load Balancing as a Service (LBaaS) framework of the OpenStack VIM. Complementing the Kuryr container networking integration (see section 4.3.3), we integrated a commercial Application Delivery Controller (ADC), Citrix NetScaler, using an open source OpenStack LBaaS plugin, which we certified against all OpenStack versions that were released over the course of the project. As part of WP7 activities, we have compared the NetScaler Load Balancing backend against the open source ones that are part of OpenStack. Finally, we implemented open source Ansible modules for NetScaler ADC, which automate the post-deployment configuration of the NetScaler servers and their services, achieving perfect alignment with the relevant support that was introduced to the ManageIQ NFVO (see section 3.6). **Annex A, section A3.5** provides more details on these activities.
- On the SFC topic, we have introduced support for Network Service Header (NSH) in NetScaler. As a result, NetScaler ADC can play the role of the Service Function in the SFC architecture. The NetScaler instance receives packets with Network Service headers and, upon performing the service, modifies the NSH bits in the response packet to indicate that the service has been performed. In that role, the NetScaler appliance supports symmetric service chaining with specific features, for example, INAT, TCP and UDP load balancing services, and routing. Considering the complexity of the current SFC solutions and the initial maturity level of their implementations, we have also investigated a promising innovative approach to support SFC, based on IPv6 Segment Routing (SRv6). Please refer to **Annex A, section A3.6** for more information on these activities.

3.5 OSM Evaluation and Integration

Open Source MANO (OSM) is a project supported by the ETSI standardization body, aiming to develop open source software that implements the main MANO components of the ETSI NFV framework: the VNF Manager (VNFM) and the NFV Orchestration (NFVO). Despite the OSM relation with the ETSI NFV framework, this tool has been used by the Superfluidity project basically to implement the components of the ETSI MEC (Multi-access Edge Computing). In particular, to implement the Mobile/Multi-access Edge Orchestrator (MEO) and the Mobile/Multi-access Edge Platform Manager (MEPM), which have clear similarities with the NFVO and VNFM, respectively. The work performed by the Superfluidity project in this scope was related to the customization of the OSM tool to the



specificities of the ETSI MEC framework, as well as the differences between the Applications and the VNF/NS concepts.

The status of OSM project has been reviewed, leading to new requirements and a more fine grain information about the strong points as well as the missing features. In addition, integration actions into the Superfluidity framework has taken place, being provisioned from the ManagelQ, and being in charge of the MEC orchestration. For more information, see **Annex A, section A2.2.4, Section A2.3 and Section A3.3.**

3.6 ManagelQ as a NVFO

ManagelQ is a management project that enables managing containers, virtual machines, networks and storage from a single platform, connecting and managing different existing clouds: OpenStack, Amazon EC2, Azure, Google Compute Engine, VMware, Kubernetes, OpenShift, etc.

The main reason for choosing ManagelQ as an NFVO is due to being able to work with both VMs and Container providers. However, after feature analysis, we discover a few gaps that needed to be address for the Superfluidity purposes.

3.6.1 Ansible execution support

The main concern about the existing NFVO tools was the lack of applications life-cycle management actions for container. Therefore we worked on an extension to fix this gap on ManagelQ. The proposed extension is based on supporting ansible playbook execution within ManagelQ. Therefore, any action (as ansible is agent-less) can be triggered in any of the providers, for example, deployment of a HEAT template for the OpenStack/VMs case, or deployment of kubernetes/Openshift templates for the Kubernetes/Containers case. The user then only need to push their desired playbooks to their associated git repositories, and then select them from the ManagelQ UI to execute them in the proper cloud/provider. More details available at **Annex A, section A3.4.1.**

3.6.2 Multi-site support

As a follow up extension for the previous point, we see the need of synchronizing actions across different sides. Hence we worked at supporting multi-site deployment from ManagelQ. Up to now, with ManagelQ you can manage different providers (i.e., different sites), but not execute a set of actions that involves several of them at the same time. We have added to the ansible support at ManagelQ the option to include a host file where you can specify the different sites where the playbook need to be executed -- just by also including into the ansible playbook tasks the info about where (of those sites) they need to be executed. For more information check **Annex A, section A3.4.2.**



3.7 OpenShift-Ansible

In order to easily consume all the above features, as well as to better integrate with current cloud environments, we have contributed to a set of playbooks that allow you to install OpenShift on top of different cloud infrastructures: OpenStack, Amazon, Azure, and Google Compute Engine.

Our efforts have focused on implementing kuryr roles inside the openshift-ansible playbooks so that kuryr components can be deployed on the OpenShift installation on OpenStack, in a containerized way. Thanks to this, it is possible to install OpenShift in an existing OpenStack deployment with kuryr configured by just executing one playbook. More details at **Annex A, section A3.7**.

3.7.1 Integration with ManageIQ

In addition to the openshift-ansible contributions, we have also integrated it into ManageIQ for an even more simple user experience. Thanks to the integration, a tenant can simply ask for an OpenShift deployment from ManageIQ UI (e.g., ask for a deployment with 1 master node, 1 infra node and 5 worker nodes) and it will be automatically deployed (by executing the openshift-ansible playbook with the configured parameters) on the selected OpenStack provider, on top of OpenStack VMs, giving the user a functionality similar to have: OpenShift as a Service. More details at **Annex A, section A3.7.1**.

3.7.2 Baremetal containers support

Finally, we extended the openshift-ansible playbooks to also support provisioning baremetal nodes. This adds the flexibility of having an OpenShift/Kubernetes installation that runs both on top of OpenStack VMs as well as on baremetal nodes. This is specially relevant for NVF deployments where some components may require to run (containerized) on baremetal nodes for increase performance, but still being on OpenStack neutron networks so that they can talk to other VMs or nested containers transparently. This work also enables the previous ManageIQ integration to ask for the number of VMs and baremetal nodes that will be used for deploying the OpenShift components. This contribution was presented at a keynote talk at DevConf 2018. More details are provided at **Annex A, section A3.7.2**.

3.8 RDCL 3D

As shown in Figure 2, the RDCL 3D tool is a part of the NFVO layer in the Superfluidity provisioning and control architecture. The RDCL 3D tool offers a GUI to the user (i.e. the service designer / network operator). Its role is to manipulate the information models of network services and service components (VNFs and more in general RFBs) and to interact with Orchestrators. The RDCL 3D tool



per se is agnostic to the information model, and can be specialized to support different ones. In particular, following the Superfluidity architecture based on the concept of RFBs and of multiple RFB Execution Environments, we have considered information models that are more complex than the state-of-the-art models considered for NFV. The new features that we have considered include the “nested” decomposition of VNFs into more granular RFBs, and the support of traditional VMs, containers, Unikernel VMs.

In the context of WP6, The work on RDCL 3D tool performed in other WPs has been extended to be integrated with the WP6 reference architecture, in particular for the support of -- mainly regarding the integration with ManagelQ and Ansible.

The functionality developed in RDCL 3D allows has included a script translating the Superfluidity data model descriptors into Ansible playbooks that are uploaded to a Git repository where ManagelQ will consume them. More details about this integration effort can be found at **Annex A, section A3.8**.

3.9 Service Characterisation Framework and Deployment Template Optimisation

Resource allocation is important in the context of “virtualisation” of network services as it plays an important role in both service assurance and Total Cost of Ownership (TCO). In order to provide scalable rationalisation of service resources allocations and TOC we developed a framework that provides orchestration of an experimental lifecycle, management of data collection and analysis of collected data. The output of the framework are optimised deployment templates to deliver specific levels of performance in compliance with required service level objectives (SLOs).

The service characterisation framework was applied to the optimisation of a deployment template for the Unified Origin video transmuxing workload using the WP7 Superfluidity demonstrator implementation. Resource and configuration options focused on vCPU's, RAM, vNIC and Memory Page Size. The results obtained show that varying the combination of the four parameters it is possible to identify three unique performance classes:

- Class A: Throughput > 180 Mbps and Latency < 0.92 ms
- Class B: Throughput > 150 Mbps and Latency < 1.05 ms
- Class C: Best effort (no specific requirements) for both throughput and latency

Analysis of the results in relation to the configuration parameters identified that using SR-IOV can positively impact both throughput and latency and this configuration on its own offers guarantees of good level of performance, placing the service performance within SLA Class A range. At the same time, this represents the most expensive solution from an infrastructure perspective, which might



lead the service provider to choose an option corresponding to a lower cost in case the user requirements fall into SLA Classes B.

In summary, the work carried out was as follows:

- Defined a high-level architecture of a Deployment Template Optimisation Framework.
- Refactored the framework developed in Task 4.1 in order to support different application plugins such as template optimisation.
- Implemented support for Open Source MANO Orchestrator API, provided by Riftware (rift.io) via the framework in order to perform automated deployment and termination of VMs.
- Conducted experimental campaign to identify an optimised template for the Unified Origin workload based on different allocations for resources and configuration options.

More details on this work can be found in **Annex A, section A3.9**.



4 Task 6.2: Access-Agnostic SLA-Based Network Service Deployment

The work of task 6.2 targeted 3 objectives of work package 6, namely,

- OBJ1: design of SLA based network function descriptors,
- OBJ2: design of cross management domains resource allocation and placement algorithms,
- OBJ3: design the control framework for dynamic scaling, resource allocation and load balancing of tasks and entity in the overall system,

We further structure this section that summarizes the work in Task 6.2 according to the above objectives. While the following sections provide overview of the work and its results, more detailed information is available in **ANNEX B: Access-Agnostic SLA-Based Network Service Deployment**.

4.1 SLA-Based descriptors

4.1.1 NEMO modeling language

Superfluidity has considered NEMO as a modelling language and promoted its ideas via extensions. NEMO [1] is a human-readable command language used for Network Modelling. It is placed by the authors in the scope of Intent-Based Networking (IBN), since it is more descriptive and prescriptive. The NEMO project has launched a series of efforts to get the language standardised. As such, the IBNEMO project within the OpenDaylight (ODL) community is classified in the Intent-Based northbound interfaces (NBIs) group.

NEMO provides basic network commands (Node, Link, Flow, Policy) to describe the infrastructure and controller communication commands to interact with the controller.

We introduced extensions to the Node definition command to import TOSCA or OSM based descriptors as Node definitions. Since Node models can make use of previously defined node models, the resulting language would be recursive and therefore support our notion of (recursive) reusable function blocks.

This concept has been proposed as an Internet draft at the NFV research group (NFV-RG) of the IRTF [2]. In addition to the import process proper, two additional features are defined in NeMo: 1.- the ConnectionPoint to map the VNF's interfaces that are significant in the function description and the connections between them, and 2.- the CONNECTION as a way to express the relations between the enhanced VNFCs (here NodeModels) in a service graph.

Importing OSM VNF Descriptors is proposed in the draft as a two-step process, where the descriptor is first imported and then used to provide the pointers to the connection points. The proposed syntax to import VNFDs is:



```
CREATE NodeModel sample_vnf
  VNFD https://github.com/nfvlabs/openmano.git
  /openmano/vnfs/examples/dataplaneVNF1.yaml;
  ConnectionPoint data_inside at VNFD: ge0;
  ConnectionPoint data_outside at VNFD: ge;
```

The proposed way to define service graphs in NeMo is:

```
CREATE NodeModel      complex_node
  Node    input_vnf    Type sample_vnf;
  Node    output_vnf   Type  shaper_vnf;
  ConnectionPoint input;
  ConnectionPoint output;
  Connection input_connection Type p2p EndNodes input,  input_vnf.data_inside;
  Connection output_connection Type p2p EndNodes output, output_vnf.wa ;
  Connection internal Type p2p EndNodes input_vnf.data_outside, output_vnf.lan;
```

The implementation details are available in Annex B.

4.1.2 Nested execution environments

In this work we extended the ETSI NFV ISG specification [3][4] to support nested VDUs using heterogeneous technologies.

Specifically, we have extended the VDU information element contained in the VNF Descriptors to reference different types of Execution Environments that can be instantiated within a VDU and described by means of some descriptor.

Further details are available in **Annex B, section B2**.

4.2 Cross management domains resource allocation and placement

4.2.1 Data center resource Allocation and Placement

Despite the ever-growing popularity of Network Function Virtualization (NFV), we are still far away from having large scale fully operational NFV networks. One of the main obstacles on this path is the performance of the network functions in the virtual environment. The hardware middleboxes that are in use by network operators today are specifically designed to provide the needed high performance (and high reliability), but getting the same level of performance from commercially off-the-shelf hardware is much more challenging. Hardware accelerators (such as DPDK and SRIOV) were



developed specifically for this purpose, yet the deployment of high performance service chains in a virtual environment remains a complex handcrafted process (e.g., as indicated by Intel's performance reports).

Service chain management in such scenarios is mostly static and operators lose one of the main attractive features of NFV -- the ability to dynamically allocate resources according to the current need. Such a dynamic mechanism would allow a much more efficient utilization of resources, since the same physical resource can be used by different VNFs when needed. Thus, achieving both high performance and agility, by being able to dynamically change the resource allocation of service chains, remains a great challenge.

Identifying near optimal deployment mechanisms for NFV service chains has recently received significant attention from both academia and industry. However, to the best of our knowledge, existing studies do not consider the cost of resources required to steer the traffic within a chain, which is non-negligible for deployment of packet intensive chains such as in the domain of NFV. Therefore, typical models (e.g., in NFV orchestrators) might either lead to infeasible solutions (e.g., in terms of CPU requirements) or suffer high penalties on the expected performance.

In our previous contribution, we focused on evaluating and modeling the virtual switching cost in NFV-based infrastructure. Virtual switching is an essential building block that enables flexible communication between VNFs but it also comes with an extra cost in terms of computing resources that are allocated specifically to software switching to steer the traffic through running services (in addition to computing resources required by the VNFs). This cost depends primarily on the way the VNFs are internally chained, packet processing requirements, and accelerating technologies (such as DPDK).

Example

Figure 5 illustrates a possible deployment of four service chains on three identical physical servers (A, B and C). As one can see, service chain φ^1 is composed of three VNFs - $\varphi^1 = \langle \varphi_1^1, \varphi_2^1, \varphi_3^1 \rangle$ -, φ^2 is composed of four VNFs, φ^3 is composed of five, and φ^4 is composed of two VNFs. In the depicted deployment (shown in Figure 5), servers A and C have the same number of deployed VNFs and thus may have the same computing resource requirement for processing. However, determining the amount of processing resources (CPU) needed for switching inside these server is far from being straightforward. In some cases, the deployment can be infeasible due to lack of sufficient computing resources for the switching task. The amount of computing resources needed for the internal switching depends on the structure of the chaining in the server, and the amount of traffic associated with each chain.

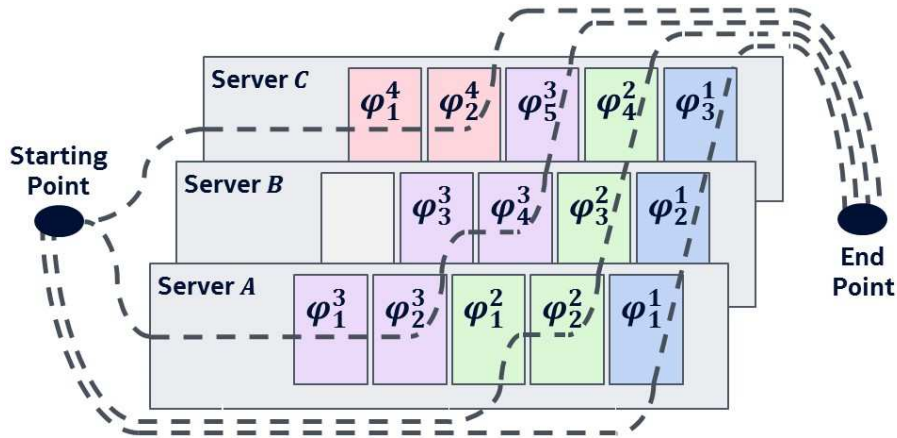


Figure 5: Example of possible deployment of four service chains on top of three

Existing work that measure and evaluate the performance of the internal switching mainly focus on two types of deployment strategies: a **distribute** strategy where each VNF in the chain resides on a different server (for example, service chain φ^1 in Figure), and a **gather** strategy where the entire chain is grouped on the same server (for example, service chain φ^4 in Figure). Some works refer to these two deployment strategies are respectively referred to as "north/south" and "east/west" deployments. We have analysed these two placement strategies, and developed a cost function that predicts the amount of computing resources needed for the internal switching. Interestingly, OpenStack/Nova global policies also follows these two deployment strategies. Namely, after a basic filtering step that clears resources according to local-server constraints (e.g., CPU cores, memory, affinity, etc.), it implements two types of global decisions that boil down to: **load balancing** of a certain metric (i.e., spread VNFs across servers), and; **energy saving** of a certain metric (i.e., stack VNFs up to the limit).

In the following, we extend this work and describe a novel SLA based resource allocation scheme for network services.

We consider arbitrary deployments (like the one described in Figure 5) and develop a general service chain resource allocation strategy that considers both the actual performance of the service chains as well as the needed internal switching resource. This is done by decomposing service chains into sub-chains and deploying each such sub-chain on a (possibly different) physical server, in a way that minimizes the total switching overhead cost. Of course, there are exponentially many ways to map the sub-chains to servers. We introduce a novel algorithm based on an extension of the well-known reduction from the weighted matching to the min-cost flow problem, and show that it gives a near optimal solution with a better running time than exhaustive search.

We evaluate the performance of our algorithm against the fully distribute or fully gather solutions, which are very similar to the placement of the de-facto standard mechanism commonly utilized on



cloud schedulers (e.g., OpenStack/Nova with load balancing or energy conserving weights) and show that our algorithm significantly outperforms these heuristics (up to a factor of 4 in some cases) with respect to operational cost and the ability to support additional network functions.

Our main contributions to this work are:

- **NFV deployment cost model.** We develop a general switching cost model that predicts the switching related CPU cost for arbitrary deployments and evaluate its accuracy over a real NFV environment.
- **Optimal deployment mechanism.** We develop an efficient online placement algorithm (OCM - Operational Cost Minimization) that uses this new cost model to minimize the switching cost of service chain requests, thus allowing more network functions to run on the same NFV infrastructure in a more efficient way. We evaluate the expected performance of this novel algorithm and show that it can significantly increase utilization.

These contributions enable NFV providers to design a new pricing models for service chaining, that quantifies the actual consumed resources by the infrastructure (in addition to the resources consumed directly by the VNFs).

4.2.2 The Operational Cost of Switching

We first define a model that captures the operational cost of virtual switching for a given server. Namely, given a placement function P that deploys all service chains in $\Phi = \langle \varphi^1, \varphi^2, \dots, \varphi^m \rangle$ on the set of servers $S = \{S_1, S_2, \dots, S_k\}$, we can infer the set of sub-chains (originating from possibly different service chains) to deploy on server S_j . Our goal is to develop a function that predicts the CPU cost of server S_j .

4.2.2.1 Virtual Switching

In virtualization-intense environments, virtual switching is an essential functionality that provides isolation, scalability, and mainly flexibility. However, the functionality provided by software switching also introduces a non-negligible operational cost making it much harder to guarantee a reasonable level of network performance, which is a key requirement for the success of the NFV paradigm. Assessing and understanding this operational cost and particularly the cost associated with virtual switching is a crucial step towards driving cost-efficient service deployments.

Several recent publications addressed the performance of intensive traffic applications in NFV chaining settings. In most industry, related works the goal is to define the setting that provides the best performance on a specific hardware, and not on the switching cost of a given service chain under



a certain setting. Our previous contribution is different as it does try to evaluate the switching cost but it does so only for the special cases of gather and distribute.

Yet the results of these recent studies indicate that the operational cost of deploying service chains depends on the installed OvS (either kernel OvS or DPDK-OvS), the required amount of traffic to process, the length of the service chain, and the placement strategy. Moreover, it appears that there is no single strategy that is always superior, with respect to the operational cost, and that the best strategy depends on the system parameters and the characterization of the deployed service chains.

4.2.2.2 The Cost of Virtual Switching

Let $\{\varphi_{s_1 \rightarrow e_1}^1, \dots, \varphi_{s_r \rightarrow e_r}^r\}$ be a set of r sub-chains, each is part of a decomposition from possibly different service chain. Each sub-chain $\varphi_{s_w \rightarrow e_w}^w$ (for $1 \leq w \leq r$) might carry different traffic requirements, that are defined by service chain $\varphi^w \in \Phi$. Figure 6 illustrates such a deployment on server S_j . The total cpu-cost consumed by the guests (denoted by $C_j^v(P)$) is just the sum of the required CPU for each VNF, but calculating the hypervisor switching cpu-cost $C_j^h(P)$ is much more involved.

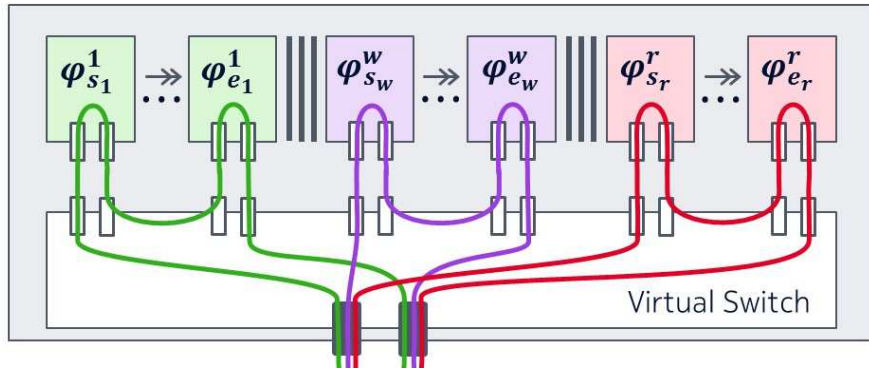


Figure 6: Server S_j deployed with r sub-chains from possibly different service chains

$$\{\varphi_{s_1 \rightarrow e_1}^1, \dots, \varphi_{s_w \rightarrow e_w}^w, \dots, \varphi_{s_r \rightarrow e_r}^r\}.$$

For a single sub-chain, the switching cost is exactly the gather cost of a chain deployed on server S_j and can be obtained directly from function F defined in our previous contribution. However, when we deploy more than one sub-chain, the overall cost is not the sum of the separate deployments. The cost of deploying two sub-chains is much smaller than twice the cost of deploying one sub-chain. Thus, in concurrent chain deployments the switching cost is amortized and the total cost is smaller than the sum of two separate costs.

To quantify this, we define $\|\varphi_{s \rightarrow e}\|^r$ to be the **concurrent deployment** of r copies of subchain $\varphi_{s \rightarrow e}$, and $\Delta(\varphi_{s \rightarrow e}, r)$ to be the **switching cost delta** between r times deploying a single sub-chain $\varphi_{s \rightarrow e}$ and the concurrent deployment of r copies of $\varphi_{s \rightarrow e}$, i.e.,



$$\Delta(\varphi_{s \rightarrow e}, r) = (r \cdot F(\varphi_{s \rightarrow e}) - F(\|\varphi_{s \rightarrow e}\|^r))$$

Given a set of r sub-chains $\{\varphi_{s_1 \rightarrow e_1}^1, \dots, \varphi_{s_r \rightarrow e_r}^r\}$ that are deployed on server S_j (as illustrated in Figure 6), we can now compute the cpu-cost as follows:

$$C_j^h = \sum_{w=1}^r F(\varphi_{s_w \rightarrow e_w}^w) - (r-1) \cdot \left(\sum_{w=1}^r \Delta(\varphi_{s_w \rightarrow e_w}^w, r) \right) / r$$

The left-hand side of C_j^h is the sum of the cpu-cost associated with the gather deployment of each sub-chain, and the right-hand side reflects the saving due to the concurrent deployment of r chains which is $(r-1)$ times the average switching cost delta. Note that for $r=1$ we do not have any savings and the cost is just the gather cost of deploying a single sub-chain.

4.2.3 Optimized Operation Cost Placement

We are now ready to present our next contribution, that is a placement algorithm for Operational (switching) Cost Minimization (OCM). Our algorithm entails a strategy to deploy the service chains onto the set of physical servers. Per each service chain in the sequence, we find a partition of the chain into sub-chains and an allocation of a server to each sub-chain in a way that minimizes the total switching cost. This on-line handling does not guarantee global minimum switching cost but as we show in this section, it does reduce the switching CPU cost and allow deploying significantly more services.

4.2.3.1 Operational Cost Minimization Algorithm

The **Operational Cost Minimization** (OCM) algorithm receives as an input a set of servers $S = \{S_1, S_2, \dots, S_k\}$, and a sequence of service chains $\Phi = \langle \varphi^1, \varphi^2, \dots, \varphi^m \rangle$. For each service chains $\varphi \in \Phi$, the OCM invokes the optimal service chain placement step shown in Figure 7.



Algorithm optimal service chain placement step

Input: $S \leftarrow \{S_1, S_2, \dots, S_k\}$: set of servers
 $\varphi \leftarrow \langle \varphi_1 \rightarrow \varphi_2 \dots \rightarrow \varphi_n \rangle$: service chain
 1: $min-cost \leftarrow \infty$: minimum cost found so far
 2: $deploy-map \leftarrow NIL$: maps all VNFs to servers in S
 3: $\mathcal{A} \leftarrow$ all possible set partitioning (or decompositions)
 4: **for** every set partition $a \in \mathcal{A}$ **do**
 5: **for** every partition $p \in a$, and server $S_j \in S$ **do**
 6: $C_j^h, C_j^v \leftarrow$ compute the cost of deploying p on S_j
 7: $G \leftarrow$ reduce to minimum cost flow in a graph
 8: **if** $min(G) \leq min-cost$ **then**
 9: $min-cost \leftarrow min(G)$
 10: $deploy-map \leftarrow$ extract solution from G
 11: **return** $deploy-map$

Figure 7: Optimal service chain placement step.

Each placement step is made of three building blocks:

- i. List all set partitions of the VNFs in φ . In a relaxed version of the algorithm, we consider all decompositions in which every service chain goes through a server at most once.
- ii. Given a set partition build the objective function, namely a cost function that predicts the operational cost of network traffic switching per each server;
- iii. Given an objective function build a reduction to minimum-weight matching in bipartite graphs between partitions and servers, where the weights are given by the objective function.

We denote by P_o (and P_r) the **optimal** placement function (and the **relaxed** placement function) that implements the optimal OCM algorithm (and respectively the relaxed version of the OCM algorithm). To conclude, we introduced the Operational Cost Minimization (OCM) placement algorithm, a performance-oriented deployment mechanism that minimizes internal switching CPU overhead and improves network utilization. This algorithm uses a novel cost model that captures the operational cost of the internal virtual switching for a given server. We provided empirical evidence, using a real NFV-based environment, indicating that our cost model is accurate comparing to actual deployment measurements (lower than 5%). Using this cost model, we introduced an efficient online placement algorithm that minimizes the switching cost of service chain requests. We show that OCM significantly reduces the operational costs and increases utilization, when compared to commonly used deployment strategies (up to a factor of 4 in the extreme case and 20% - 40% in typical cases). Our work opens opportunities to design new pricing models for service chaining by NFV providers. More details are available at the attached INFOCOM paper in Annex B.



4.2.4 Resource allocation in Mobile Edge Computing

In a Mobile Edge Computing (MEC) System, applications will run at ME (Mobile Edge) Hosts in a virtualized environment, as VDUs (*Virtualization Deployment Units*), e.g. VM or Container. For that purpose, and in line with ETSI NFV, management and orchestration components have been added to the MEC system, composed by “Mobile Edge Host Level” and “Mobile Edge System Level”. That is, while the latter has a global view of the entire MEC System, including all ME Hosts, the former acts at the ME Hosts level, with the functions: Element Manager for the ME Host, Rules and Requirements management for MEC applications, and MEC Applications LCM operations (similar to VNFM for NFV). At each ME Host, the ME Platform component may or may not be deployed over the virtualization infrastructure, possibly using specific HW and SW. Figure 8 depicts the full ETSI MEC architecture.

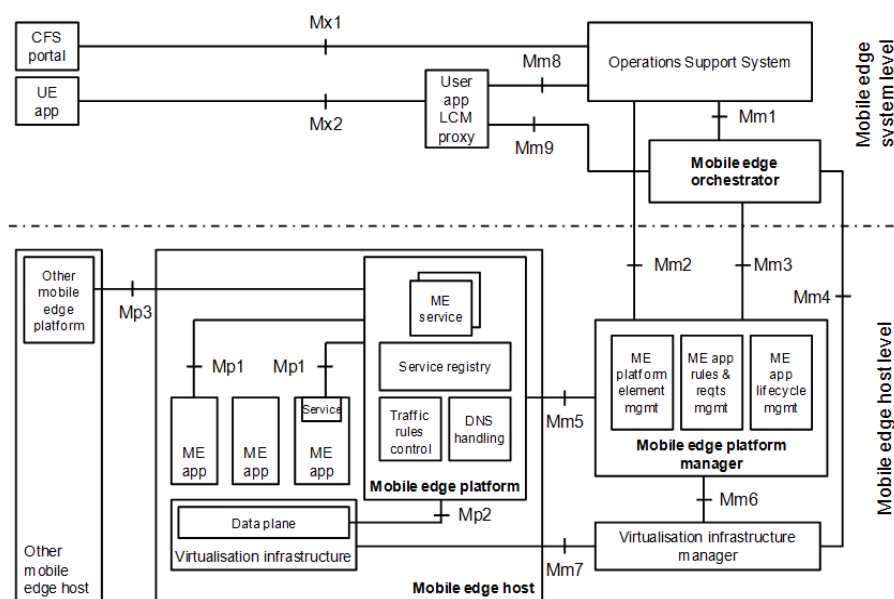


Figure 8 - ETSI MEC Architecture

In this context, resources allocation to MEC Applications will occur on the virtualization infrastructure, existing at each ME Host. This will be controlled by the ME Orchestrator, as part of ME Applications Life-Cycle Management (LCM) operations. SUPERFLUIDITY internal document I6.1 “Initial Design for Control Framework” documents MEC management and orchestration requirements as well as respective flows.

The following entities, hosted at the MEO, are required for ME Applications resources allocation:

- MEC App Catalogue



- Stores the catalogue of deployable MEC Applications, with associated images and MEC Descriptor (where applicable rules and requirements are specified)
- MEC App Descriptor
 - Even if not yet defined by the ETSI MEC ISG, MEC Applications will have a descriptor associated to them. This will be in line with the VNF Descriptor, including specific information like, delay budget, required MEC Services to run and required virtualization resources.
- MEC Infrastructure
 - Keeps track of available, reserved and in use resources at all ME Hosts, for usage by MEC Applications
- MEC Hosts Inventory
 - Lists and describes the ME Hosts under MEO control. This shall include, per ME Host, a mapping with served eNB and available services (LOC, RNIS, DNS, etc.)

MEC Apps instantiation may occur:

1. Triggered by MEO
 - Selection of ME Hosts to deploy MEC Apps shall be automatic and based on existing information at the MEO and associated MEC Applications' Descriptors. Thus, from no need for allocate resources (only on-boarding), to allocate resources at all ME Hosts, all scenarios are possible.
2. Triggered by management (OSS)
 - Selection of ME Hosts for Applications instantiation will be the decided by a third-party entity, eventually including the analysis of the information contained in MEC Application Descriptors.
3. Triggered by UE
 - Via the MEC "User App LCM Proxy" component, entities at the UE may request the instantiation of specific MEC Apps. Application requirements (e.g. latency, compute resources, storage resources, location, network capability, security condition etc.) will be analysed in order to select a host fulfilling all the requirements.

In any of the previous situations, upon identification of the ME Hosts for Application deployment, resources at the corresponding virtualized infrastructures, must be reserved, involving the defined ME Platform Manager and the VIM, in a similar way to what happens with VNFs.

Even if current ETSI ISG MEC work does not address MEC Applications scale in/out or up/down, there is no reason to exclude that as part of LCM operations, in the context of resources allocation. In the same sense, ETSI ISG MEC considers that individual MEC Applications will be made of single VDUs. Similarly to VNF, there is no strong reason for not considering that MEC Applications may be the result of the composition of several VMs.



Resources allocation for MEC Applications will be managed by the interactions between MEO, MEPM and VIM.

More information on the placement, service migration and migration in MEC is detailed in **Annex B, section B4**.

4.3 Dynamic scaling, resource allocation and load balancing

4.3.1 VM Scaling and Scheduling via Cost Optimal MDP solution

4.3.1.1 Modelling y flexible queuing system

We address cost optimization problem of VM scaling and scheduling which considers both the exogenous and internal cost constraints. The existing VM scaling tools as e.g., in AWS, provide only limited instrumentation for cost optimization and are not designed to account for any additional exogenous constraints. To account for this, we model the problem as cost-optimal task

scheduling to a queuing system with flexible number of queues, where a queue stands for a VM, hence it can be deployed, can have a task scheduled to and can be terminated. We aim to capture the impact of a variety of constraints which includes decision making on queue deployment and termination, considers delay cost, cost associated with having deployed VM (even if idle), task processing incentives which are modeled as scheduling reward and rejection fine.

4.3.1.2 Solution by Markov Decision Process

We analyse the system by Markov decision process (MDP) and numerically solve it to find the optimal policy. The solution we provide captures the aforementioned constraints. We show that the optimal policy possesses decision thresholds, which depend on the system parameters. In particular, distinctive impacts of VM deployment cost, cost of having an idle VM, and the cost associated with a delay are observed, graphically presented, and the corresponding insights are analysed. In addition, to be practically sound, we investigate the impact of average VM deployment time.

4.3.1.3 Validation of the policy found by MDP through AWS setting

We validate policies found by MDP, through directing an exogenous computational tasks flow with known statistics to a set-up implemented on AWS. The policy is implemented by setting accordingly the AWS Elastic Load Balancer (ELB) thresholds for VM deployment and termination. The results clearly confirm the superiority of the optimal policy over any other heuristically applied ELB management. Note that the policy which we find by the presented here method can be adopted by any cloud infrastructure.



More details are found in Annex B: section B5, where we also provide a machine learning based technique for the Life Cycle Management of containerized workloads.

4.3.2 Load Balancing as a Service

Elaborating on the activities covered in Section 3.4, particularly to focus on the Load Balancing capabilities, we proceeded to integrate NetScaler ADC, a commercial-grade load balancer, with OpenStack and the respective Load Balancing as a Service (LBaaS) extensible framework. For that purpose, and in compliance with the ETSI NFV reference architecture, we utilized the respective EMS element, NetScaler Management and Analytics System (MAS). To prepare for the integration and validation activities (WP7), we deployed and verified the concept in the Superfluidity staging environment. For a detailed description of the procedures, please refer to **Annex B, Section B6**.

4.4 Optimal design and management of RFBs over a Superfluid 5G network

We have first focused on the problem of designing an RFB-based 5G networks, by targeting the reduction of the installation costs for the physical nodes. In [5] we have provided an optimization model that allows the minimization of the total costs, under the RFBs placement on the physical 5G nodes. The presented formulation is able to take into account multiple RFBs types, ranging from the low-level ones, devoted to the setting of the communication channel to the users, to the high level ones, which are able to provide application layer features. The goal is then to optimally select the set of installed 5G nodes, as well to properly dimension each of them in terms of commodity and dedicated HardWare (HW). Results, obtained over a representative 5G case study, demonstrate the efficacy of the proposed approach.

Then, we have faced the problem of the optimized management of a set of RFBs over a 5G network. In this case, the problem is related to the dynamic allocation of the RFB components, which has to be performed in real time, in order to match the required levels of flexibility and agility. Initially, we have focused on the problem modelling by means of optimization tools [6]. The goal is to target a given KPI (e.g., maximization of the user traffic, or minimization of the number of powered on 5G nodes), while ensuring users traffic, RFB placement, and availability of physical resources on the 5G nodes. In the following, we have provided an efficient algorithm, based on bio-inspired techniques [P5G-Globecom2017], which is able to sub-optimally solve the problem in a reasonable amount of time. In both cases, our results clearly indicate that the management of a RFB-based network is feasible, and that the performance achieved by users depends on the specific KPI chosen by the operator. Currently, an extended version of [6] is under revision for a journal (Wiley International Journal of Network Management).



Another important problem that we have considered is the reduction of energy consumption for the 5G nodes, while ensuring Quality of Service constraints to users. To this aim, in [7] we have considered a special case of RFB, where this component is realized by means of a VNF. We have then focused on the problem of managing set of VNF chains, that have to be deployed on physical nodes, in order to provide the service to users. The goal is then to reduce the total energy consumed by the 5G nodes. To this aim, different formulations of the problem are provided, together with a set of fast algorithms. Results show that the energy consumption can be wisely preserved, while still ensuring the service to users.

Finally, we have also provided economic indications about the RFB-based solution at a global level. [8] presents a stakeholder analysis, as well as a Strengths Weaknesses Opportunities Threats (SWOT) analysis. In addition, in [9] we focus on the profitability of a RFB-based 5G network in two cities, i.e., Bologna (Italy) and San Francisco (CA). Results show that the initial investment incurred by the operator can be compensated by properly setting the users' monthly subscription fee. Overall, the RFB-based 5G deployment appears to be profitable for the operator from an economic point of view.



5 Task 6.3: Automated Security Verification Framework

Network security and its correct operation are two sides of the same coin. Misconfigured routers allow attacks to hosts or routers in the network, thus lack of correctness reduces security. The converse also holds true: a network cannot behave correctly, as specified by its operator, if there are low level attacks on the software on any of its components, for instance routers or network functions; such attacks allow the attacker to inject arbitrary traffic, thus breaking the correctness of the network.

5G networks enable operators to quickly deploy new functionality in the form of network functions, and this will allow networks to keep up with the development pace of applications. However, dynamically instantiated network functions make it significantly more difficult to ensure that the network is behaving correctly and that it is secure.

In Superfluidity, we propose an integrated approach that ensures both network security and correctness, at the same time. Our approach leverages network dataplane verification, in particular the Symnet symbolic execution tool and the SEFL language developed in a prior project (Trilogy II) and the initial part of the Superfluidity. The solution has three distinct parts that are used at different times in the service deployment life-cycle:

- 1) *[before deployment]* **Verifying that the high level service configuration meets the correctness requirements set out by the operator.**
- 2) *[before, after deployment]* **Verifying that the low-level implementation is bug-free.**
- 3) *[at runtime]* **Detecting in real time possible attacks using anomaly detection.**

We now provide an overview of these three components; for details please refer to Annex C.

5.1 Verifying high-level service configurations

To change network functionality, the network operator or third-parties will provide a service configuration. These configurations could be written in NEMO, RDCL3D or other similar languages, and they are, essentially, directed graphs connecting various network functions (vertices in the graph). The network functions are, in Superfluidity terminology, Reusable Functional Blocks (RFBs) and their processing can be specified either by selecting from a catalogue of functions (e.g. Click modular router elements or Openstack Neutron primitives) or by providing a program that specifies the processing to be done by the RFB and can be directly run in the dataplane. Such programs could be expressed in programming languages such as P4 or Openstate (an output of the Superfluidity and Beba H2020 projects, from CNIT/Uni Roma Tor Vergata); such programs can be run directly in the dataplane, at very high processing speeds. Recently Barefoot Tofino, the first commercial P4-enabled switch, has started shipping.



Quick service instantiation and reconfiguration is the cornerstone of 5G, but it fosters many risks; in particular, how can one ensure that the new service about to be deployed will not harm the operator's network, i.e. already running network services? Another question is how can we ensure that the new service behaves as its user expects.

To answer such questions, one possible approach is to use active testing but this approach scales poorly and thus has limited coverage. Another possible approach is to use formal verification techniques, which is the approach we take. In particular, before instantiation, we need to ensure that service configurations conform to the policy of the network operator and/or the party deploying the configuration. The policy can be seen as a correctness specification and can include reachability requirements, the way packets will be changed, isolation requirements, and so on.

We use exhaustive symbolic execution of dataplane code to test such policies. In our approach, the network dataplane is a directed graph connecting network functions expressed in the SEFL language. We inject *symbolic packets* at selected vantage points in the network and track how they are processed by the network. A symbolic packet is a packet whose header fields can take any possible value.

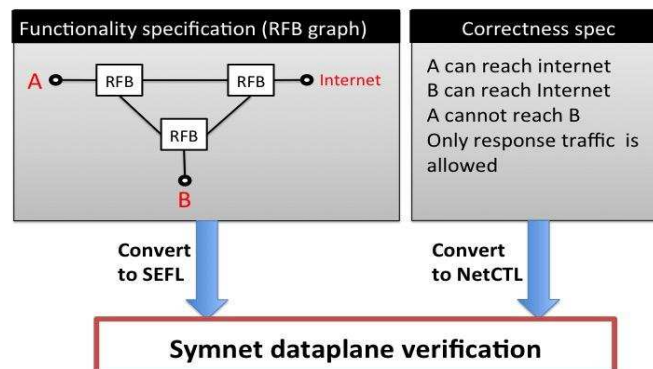


Figure 9: Symnet dataplane verification

To verify that a high-level RFB graph corresponds to the policy specified by the operator we start by translating RFB processing to SEFL and using the Symnet symbolic execution tool to verify that the policy holds. This step is captured in Figure shown top the right: the inputs to our verification tools includes the RFB graph, as well as high-level policies we want the resulting network configuration to obey. We discuss these in greater detail next.

5.1.1 Translating RFBs to SEFL

We have implemented translators for a many types of RFBs, shown below.



- Click modular router elements - over 50 basic Click elements can be translated to SEFL. The elements we do not translate are the ones relating to performance, not functionality (i.e. traffic shaper, rate limiter, etc.)
- Router FIBs - given a snapshot of a router FIB, we can generate SEFL code that performs the same functionality and is optimized for verification.
- *Openstack Neutron network primitives* - including firewall, router, load balancer and VPN.
- Openflow rule snapshots - given a dump of Openflow rules taken from a software or hardware Openflow-enabled switch, our code outputs equivalent SEFL code.
- Iptables rules - given iptables rules, we generate equivalent SEFL code. Our code also models stateful firewall processing and NATs.
- P4 programs - we have developed the Vera verification tool which includes a translator that parses the P4.14 version of P4 and automatically generates equivalent SEFL code. See Annex C for a detailed description.

In summary, given an RFB graph using RFB described in the list above, we can automatically generate equivalent SEFL code. Note that, at this point, the data acquisition must be performed offline; our translators expect the data in the required format, do not contact the switches themselves. In our future work we automatically collect data from a production environment (such as OpenMano or Openstack).

5.1.2 Verifying policy compliance of SEFL dataplanes using Symnet and NetCTL

Symbolic execution simulates how all these packets may be processed, without iteratively testing each concrete packet in isolation because this approach does not scale. Symbolic execution for networks tracks classes of packets that are treated in the same way. When symbolic execution finishes, the result is a list of symbolic packets with constraints or concrete values on their header fields, along with the set of boxes and instructions executed by that packet.

To verify whether a policy holds, we use basic symbolic execution offered by Symnet as follows:



- We convert the policy to NetCTL and then decide which packets to inject and where, as well as to decide which header fields should be made symbolic.
- We use the policy to guide symbolic execution, checking on each path that the policy holds. If it does not, we stop exploration and offer a counterexample that violates the policy.

The full description of NetCTL is included in the internal deliverable I6.3b which is attached to this document. Hereafter we provide a brief overview of our approach.

For any given SEFL program, symbolic execution will explore a large number of paths, many of which are successful. In our evaluation, we typically see hundreds such paths. Examining them manually to decide whether the behavior is correct is time consuming and error-prone. We wish to specify desirable properties and check them automatically.

The specification must combine packet constraints at specific ports of the network (or state properties) with constraints over the possible paths which the packets may take between ports (or path properties). We can already express state properties via SEFL instructions. For instance, the property ‘destination IP is always X at port out’ can be verified by placing the SEFL instruction `Dest-IP != X at port out` and observing the successful paths from port out.

In order to express path properties, we have considered a wide range of SDN policy languages, e.g. the Kinetic family, FatTire, NetPlumber, as well as approaches relying on logic programming (e.g. Shenker’s FML). We have found that all such languages are limited in their ability to express compositional constraints.

We have thus turned to Computation Tree Logic (CTL). In CTL, temporal operators such as F (i.e. sometime in the future) and G (i.e. always in the future) are combined with path quantifiers: \exists (on some path) and \forall (on all paths). For instance, the policy: $\forall F \text{destTCP} == 80$ evaluated at some port P of a box, expresses that on all possible packet paths from P, `destTCP` will eventually become 80.

The syntax of NetCTL is given below:

$$\phi ::= \text{SEFL} \mid \neg\phi \mid \phi \wedge \phi \mid XY\phi$$

where $X \in \{\exists, \forall\}, Y \in \{F, G\}$.

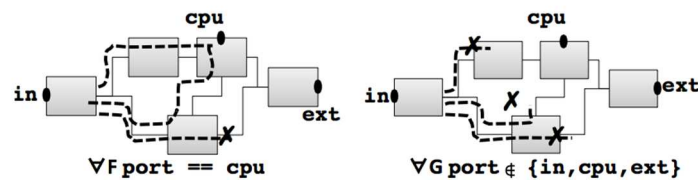
Unlike Merlin, FatTree or NetPlumber, NetCTL is compositional: starting from simple properties, we can construct more complex ones. For instance, we can express that “whenever the IP destination of a packet becomes a public address, port P is reachable” via the formula: $\forall G(\text{ip} != 192.168.0.0/16 \rightarrow \exists F \text{port} == \text{Internet})$. CTL can express many other properties including TCP connectivity and invariance across tunnels.



In principle, checking NetCTL formulae can easily be implemented after exhaustive symbolic execution by checking the outputs. However, this approach is also inefficient because in many cases we can check a property without exploring all possible paths.

That is why we integrate NetCTL verification with symbolic execution. In our implementation, NetCTL verification is performed as added checks on the packets after each SEFL instruction is executed; the overhead of these checks is very small in practice. After every check we can decide to prioritize a certain path or stop execution altogether. Because of this, in most cases, our approach checks NetCTL properties faster than exhaustive symbolic execution. In Annex C.1 we provide a correctness verification of a NAT implemented in P4, showing how this approach can reduce the amount of work performed by symbolic execution by a factor of two.

In the figure to the right, we briefly illustrate NetCTL verification. The figure describes two symbolic execution traces performed on the same topology — a simplistic illustration of the P4 NAT model,



described in more detail in the subsequent sections. Boxes represent SEFL code blocks and solid lines — links between boxes. Dashed lines describe the paths explored by our verifier. The formula $\phi_1 = \forall F(\text{port} == \text{cpu})$ (left) expresses that all paths eventually reach port cpu. In order to evaluate it, our checker performs symbolic execution starting at port in. The checker will explore each encountered path until port == cpu is satisfied, the path ends, or it becomes unsatisfiable. Suppose the checker explores three paths, as shown in the figure. Since the formula $F(\text{port} == \text{cpu})$ is true on the first two paths only, ϕ_1 is false. The formula $\phi_2 = \forall G(\text{port} \notin \{\text{in}, \text{cpu}, \text{ext}\})$ (right) expresses that all packets are dropped by the NAT. To verify it, our checker will determine if $\text{port} \notin \{\text{in}, \text{cpu}, \text{ext}\}$ is true on each execution path, and after each SEFL code-block. In our example, this is indeed the case, thus the policy is true.

5.2 Verifying low-level implementations

In most cases, the SEFL model is very close in functionality to the actual processing of the box: this is the case for router FIBs and Openflow rules. This is because the functionality of the switch dataplane is simple enough that after passing basic tests, we can trust that both the hardware and the model to be correct. The same is not true for a range of RFBs, including:



- P4 programs which are surprisingly complex to develop and debug.
- Openstack dataplane rules which are created by the very complex Neutron software starting from tenant configurations.
- IPtables rules which are implemented in C and offer a wide range of functionality, beyond basic filtering.
- Click element implementations (written in C++).

To find bugs in P4 programs, our approach relies on the strength of exhaustive symbolic execution to catch a variety of bugs, which are frequent in practice. To find problems with the other RFB implementations, we rely on the novel notion of ***symbolic execution equivalence***. We discuss these in turn.

5.2.1 Finding bugs in P4 programs

We have developed a P4 verification tool that translates P4 to SEFL and integrates NetCTL verification with Symnet. Even without a specification, by default, Vera inserts checks that capture a wide range bugs in P4 programs and flags such bugs to the user as failed paths. For each failed path, Vera also generates a concrete packet that matches the path constraints, which can be used to test the bug in P4 switches, be they software or hardware. Below are the types of errors that Vera catches automatically:



- **Implicit drops** are flagged when a packet reaches the buffering mechanism without having an `egress_spec` set. Vera catches this by adding an assertion that the `egress_spec` must be non-zero when it reaches the buffer.in port.
- **Table rules that match dropped packets** are flagged as errors by adding an assertion that `egress_spec != 511` in the preamble of all actions.
- **Invalid memory accesses** are frequent P4 mistakes, when users do not test the validity of a header before using its fields. Vera relies on Symnet's memory safety guarantees to capture these errors; when accessing an unallocated field, Symnet will fail the current path.
- **Header errors** Malformed headers are captured during parsing by using the exists SEFL instruction. Adding an existing header or removing an inexistent one are also caught automatically as deparsing errors.
- **Scoping and unallowed writes** Certain metadata values are read-only in P4, yet the P4 compiler allows the program to write them (e.g. the `egress_port` metadata). Further, static registers can only be read from one table according to the spec, yet the compiler allows such reads. Vera catches such errors during the translation process and reports them to the user.
- **Out-of-bounds array accesses** are caught automatically by Vera by adding, before each array access, an out-of- bounds check for the index. At runtime, the solver will check if the constraint is satisfiable and if it is the user will get a failed path providing an example packet that triggers a possible out-of-bounds access.
- **Field overflows/underflows** are the only arithmetic exceptions possible in P4 (because division is not supported) and Vera catches them by adding a check before each addition/subtraction operation.

Loops are also caught automatically. The loop detector runs by default on the parser input port and on the egress input port which are the two places where packets can be redirected backwards in the P4 pipeline. Whenever a packet enters visits one of these ports, Vera remembers the entire memory state (i.e. the values and constraints or all the meta- data and header fields). When a packet revisits the same port, its memory state is compared to all the previous saved memory states. Two memory states are different if and only if at least one symbol has a different value in the two states. Note that we compare not only concrete values, but also symbolic ones: if a metadata is bound to the same symbolic value in both states, it is deemed to be equal. Whenever Vera discovers two memory states that are equal, it fails the current path with the "loop detected" message.

Vera and its evaluation is provided in detail in Annex C.1, as part of a paper under submission. Here we simply list the set of bugs Vera has caught in public P4 programs.



<i>Program</i>	<i>Size (LOC)</i>	<i>Verification time (sec)</i>	<i>Implicit drop</i>	<i>Parsing</i>	<i>Deparsing</i>	<i>Header ops.</i>	<i>Invalid access</i>	<i>Underflow / overflow</i>	<i>Loop</i>	<i>Processing dropped packets</i>
copy-to-cpu	70	0.1		•		•				
resubmit	70	0.4							•	
encap	130	0.45	•		•	•				
simple router	145	0.55	•							
simple NAT	290	1.25	•			•				
simple router + ACL	200	0.8	•							•
Axon	100	14		•			•			
Switch	6000	5-15/sym.pkt.				•	•			
Beamer mux[25]	340	1.4	•		•		•			
NDP switch[12]	210	0.8					•			
P4xos[7]	650	13.4	•				•			

Table 2: Bugs caught by Vera

5.2.2 Symbolic execution equivalence and its applications

Automatically finding bugs in C or C++ implementations of RFBs is much more tricky because exhaustive symbolic execution is typically not feasible. Nor is verifying that Neutron correctly implements tenant configurations into the dataplane (e.g. as Openflow and iptables rules).

In the case of Neutron, we do however know that the tenant configuration - the abstract specification of what the network should do - must be equivalent to the resulting dataplane rules. So instead of verifying Neutron per se, we verify that its output is equivalent to the input.

Checking equivalence is a very useful primitive to have in the context of network verification. If we had such an algorithm, we could check whether Neutron is indeed behaving correctly for any given input (a tenant configuration). This would still not constitute a proof of correctness of Neutron for all inputs; it merely validates that for a given input Neutron behaves correctly. Intriguingly, we could apply the same approach in the case of C implementations of RFBs. Given an instantiated RFB (e.g. a firewall together with its rules) in SEFL and in C, we could check they behave similarly, i.e. their output is the same when presented the same input.

Unfortunately, checking equivalence is undecidable for general programs because it can be reduced to the halting problem. However, equivalence is decidable for programs that always terminate and have bounded inputs: to check it, enumerate all possible inputs, run both programs and check that the outputs match. Network dataplanes are such programs: they have bounded inputs (packets) and rarely loop (P4 does not include traditional loops). Loops can appear, especially network-wide, but they can be caught automatically.

We show that it is possible to quickly and automatically decide if two programs describing network dataplanes are equivalent. To make our approach scalable, we rely on exhaustive symbolic execution instead of testing all possible inputs.

Symbolic execution outputs can be used to check that two programs are equivalent but the algorithm to do so is far from trivial. We discuss here informally three basic approaches, which we call input, output and functional equivalence, and explain why they all need to be implemented simultaneously



to ensure correctness. For a description of our algorithm and a formal proof of its correctness, please refer to **Annex C, section C2**.

The simplest way to check equivalence is to compare which packets can exit any given port by examining the feasible values for each header field—we call this co- domain or output equivalence. Say ports p_1 and p_2 belong to two different programs, but they should be equivalent. The algorithm to check for output equivalence is simple: compute the reunion of possible values for each header field for each of these ports, and then check whether the resulting sets of packets are identical (this can easily be achieved using set operations).

Consider now two programs where one constraints packets to have $TTL > 0$ and then sets TTL to zero, while the other program simply sets $TTL = 0$. The two models are not equivalent, but checking just output equivalence is not enough to capture this problem.

The next step is to also check the constraints applied on the original (input) values of the header fields, before any modifications are made; when combined with output equivalence checking, we are now checking input/output equivalence. With input/output equivalence, we will find that the first model only allows packets to pass when $TTL \geq 1$ while the optimized model allows packets when $TTL \geq 0$; the two ranges are not the same, so the two models are not equivalent.

Checking for input/output equivalence is necessary to find bugs, but on its own it is still not sufficient. To see why this is the case, consider two trivial programs where one leaves the TTL field unchanged, while the other executes the instruction $TTL = 255 - TTL$. Both the input values (0-255) and the possible output values (0-255) of the two models are the same, yet they are obviously not equivalent. What we also need is functional equivalence: regardless of the initial value of the TTL, the two values of the TTL after executing the two programs should always be equal. In our example, functional equivalence is not true because the condition $TTL = 255 - TTL$ does not always hold. In fact, all these three checks are simultaneously needed to ensure equivalence: removing a single check leads to wrong results.

We have implemented this algorithm we call EQ and have applied it to find Openstack Neutron bugs. In our testbed, we installed a minimal Openstack deployment that included two compute nodes and one network node. In each experiment, we had one or multiple tenants deploy VMs and network configurations using Openstack, and then ran `sdiff` on another machine that had snapshots of the tenant configuration and a snapshot of the dataplane rules installed by Neutron (including openvswitch rules and iptables rules); acquisition was performed with scripts that dumped and copied the relevant data. The largest Neutron configuration we analyzed with `sdiff` had 16VMs belonging to six tenants, and each tenant had configured three VLANs and one router to connect them. Equivalence testing for this deployment took around 200s.

EQ is able to detect issues in the implementation of Neutron on the underlying compute and network nodes, and it also points out the dataplane rule which breaks the expected tenant-level behaviour.



We caught 5 software bugs introduced by Neutron's middle-ware implementation, 2 configuration bugs introduced by a misconfiguration on machines in the deployment - and 1 tenant-level misconfiguration. We do describe these bugs in detail in **Annex C, section C2**.

In our future work, we will apply EQ to check equivalence between SEFL and C code via symbolic execution (with Symnet and Klee respectively).

5.3 Anomaly detection

To complement cases where verification cannot offer guarantees, such as legacy, deployed boxes that cannot be verified, Superfluidity also includes an online anomaly detection component that aims to identify suspicious behaviour. This work was completed and reported in the two internal documents I6.3 and I6.3b, which are attached to this document; we only give a brief outline here.

In short, we proposed a novel universal anomaly detection algorithm, which was able to learn the normal behaviour of systems and alert for abnormalities, without any prior knowledge on the system model, nor any knowledge on the characteristics of the attack. The suggested method utilized the Lempel-Ziv universal compression algorithm in order to optimally give probability assignments for normal behaviour (during learning), then estimate the likelihood of new data (during operation) and classify it accordingly.

We have also evaluated the algorithm on real-world data and network traces, showing how a universal, low complexity identification system can be built, with high detection rates and low false-alarm probabilities. We have applied the detection algorithms to the problems of malicious tools detection via system calls monitoring and data leakage identification. We also provided results for detecting anomalous HTTP traffic, e.g., Command and Control channels of Botnets.



6 Collaboration with 5G-PPP

Superfluidity project is contributing mainly in 5G Architecture Working Group (WG) and Software Network WG, besides to coordination with 5G-PPP Steering Board and Technology Board.

This deliverable introduces a set of innovations that the project built in orchestration, provisioning and control frameworks. The advances made by Superfluidity in ETSI MANO, NFVO and VNFM are described in the recent version of the 5G Architecture white paper produced by the 5G Architecture WG and released in the Mobile Wireless Congress 2018, in which Superfluidity is very active. Mainly, Section 5 of the white paper referring to the softwarization and 5G service management captures the work of the Superfluidity project in TOSCA, OpenStack and VNF on-boarding. Some of the content of this document is also introduced in the Software Network WG and part of the white paper published in January 2017 entitled: '5G-PPP Vision on Software Networks' in which Superfluidity actors took the lead of editing section 4.

It is important to highlight the fact that most of phase 1 projects architecture are based on ETSI MANO and VM technology, which explains that many Superfluidity contributions are not yet captured in 5G-PPP ecosystem. However, as some of the partners of Superfluidity are still active in phase 2, e.g. Nokia Bell-Labs France and Nokia IL, we expect that more and more the results of Superfluidity will be disseminated and leveraged even after the completion of the project which demonstrates the high impact that Superfluidity made in Network softwarization and cloudification areas. To corroborate this fact, we can cite that many contributions made by Superfluidity in Container, Kuryr, Ansible, RDCL/RFB etc areas is leveraged in NGPaaS project (<http://ngpaas.eu>) which is a phase 2 project. To facilitate this technology transfer, the Software Network WG, chaired now by Nokia Bell-Labs France, traces 5G-PPP phase 1 projects outcomes and their reuse in phase 2 projects.



ANNEX A: Provision and Control Framework - TASK 6.1

WP6 focuses on the orchestration and management actions at a distributed system level, building upon WP5 advances. The provision and control framework main objectives are resource provisioning, and control and management of network functions as well as applications located at the edge (in this case MEC). To achieve that this framework targets dynamic scaling and resource allocation, traffic load balancing between virtual functions, or automatic recovery upon hardware failure, among others.

The main objectives from the DoW that task T6.1 targets are:

- OBJ2: design of cross management domains resource allocation and placement algorithms
- OBJ3: design the control framework for dynamic scaling, resource allocation and load balancing of tasks in the overall system
- OBJ4: development of platform middleboxes and services

There are several points where Superfluidity have contributed/focused to achieve these goals:

- Service behaviour models to support autonomous policy management reacting to current status of the system. For instance, detecting an application having interference problems and reacting to it by either performing (QoS) bandwidth limitation, migration or load balancing actions.
- Make OpenStack suitable for C-RAN/MEC components by improving network performance as well as allowing mixed VM and Container environments.
- Placement in a distributed environment with a system-wide overview

The following sections capture the requirement analysis performed for the different scenarios, i.e., NFV, MEC and C-RAN; presents an overview of state of the art solutions at different levels of the controller/orchestration hierarchy; review different orchestration options stating the preferences; and presents the proposed framework and components used at Superfluidity, followed by a list of contributions achieved at Superfluidity to support the proposed framework.



A1. Requirements analysis

In order to tackle the challenge, our approach was split into several steps. As a first step we started by analysing the use cases from WP2 as our input. The objective was the identification of shared attributes and the identification of common requirements that the use cases shared. After doing so, we had the next step ready – investigation of the aforementioned requirements' support in existing orchestration solutions. As a last step we identify the gaps between the requirements and each solution capabilities.

A1.1 NFV Technical Requirements

The following two figures depict the relevant ETSI NFV architectures: the main ETSI NFV and the MANO (Management AND Orchestration). This MANO architecture highlights the management and orchestration components (dashed box), identifying in more detail the management and orchestration interfaces, and other sub-components, like Catalogues and Services/Resources Repositories.

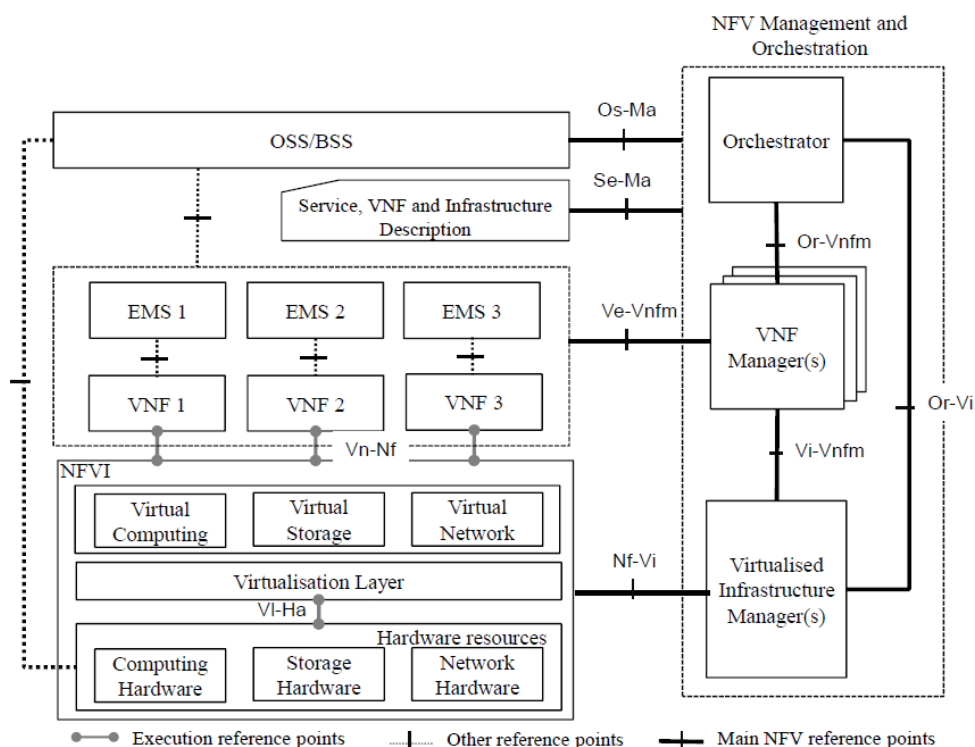


Figure 10: ETSI NFV reference architecture.



Type	Description
Onboarding	[Onboarding-01] The MANO framework MUST support the on-boarding of VNFs and NSs, respectively into the NFV Catalogue and NS Catalogues, making them available for instantiation
	[Onboarding-02] The MANO framework SHOULD perform other actions than on-boarding regarding VNF and NS packages: Disable, Enable, Update, Query and Delete.
Application lifecycle	<p>[Lifecycle-01] The MANO framework MUST support the following VNF and NS lifecycle management (LCM) operations:</p> <ul style="list-style-type: none"> ● Instantiation ● Scaling ● Modification ● Termination
	[Lifecycle-02] The MANO framework MUST provide API to the OSS or a UE application to process application LCM requests



	[Lifecycle-03] The MANO framework MUST be able to identify the VNF/NS running requirements. This will be the input for the decision on which location VNFs/NSs shall be provisioned.
Application scheduling and instantiation	[Instantiation-01] The MANO framework MUST support the indication of the following virtualized resources: <ul style="list-style-type: none"> • Compute (cpu and memory) • Storage • Network resources • Specific hardware support
	[Instantiation-02] The MANO framework MAY support the indication of the following requirements, such as: <ul style="list-style-type: none"> • Latency • Jitter • Bandwidth
	[Instantiation-03] The MANO framework MUST support the indication of physical location (PoP-DC).
	[Instantiation-04] The MANO framework MAY consider cost requirements, which can be a translation of the operator's estimation for the deployment costs.
KPI's support	[Monitoring-01] The MANO framework MUST be able to collect infrastructure and service monitoring information, in order to feed KPI-based automated management and orchestration features.
Application Scaling	[Scaling-01] The MANO framework MUST be able to scale a VNF and/or NS, on OSS request or automatically based on KPIs, in order to increase/decrease the capacity.
Load Balancing	[LB-01] The MANO framework SHOULD support load balancing function as part of the NFVI/VIM infrastructure. This requires integration with the application lifecycle and scaling functions.
	[LB-02] The MANO framework SHOULD support standard load balancing features. OpenStack LBaaS captures these requirements at https://wiki.openstack.org/wiki/Neutron/LBaaS/requirements .



	<p>[LB-03] The MANO framework SHOULD ideally support firewall load balancing mode. However, this MAY require addressing gaps in NFVI/VIM (OpenStack LBaaS doesn't appear to support this case).</p>
Service Function Chaining	<p>[SFC-01] The MANO framework MUST support the creation of Service Function Chains (SFCs), consisting of an ordered sequence of Service Functions (SFs). SFs are virtual machines, or even physical devices, that perform a network function such as firewall, content filter, content cache, or any other function that requires processing of packets in a flow.</p>
	<p>[SFC-02] The MANO framework MUST support SFCs with both simple (i.e. single SF) and complex (i.e. sequence of multiple SFs) Service Functions Paths (SFPs). Materialisation of SFCs requires the cooperation of the NFV Orchestrator, VIM and SDN controller. The NFV-O provides the VNFFG definition (please refer to relevant requirements in this document), the VIM creates the SFC by attaching the SF VM instances to network ports and the SDN controller configures the network overlay fabric that interconnects these network attachment points. According to the OPNFV SFC project (https://wiki.opnfv.org/display/sfc), SFC also depends on the VNF Manager: http://artifacts.opnfv.org/sfc/docs/design/architecture.html#vnf-manager</p>
	<p>[SFC-03] The MANO VIM MUST support the attachment of SF VM instances to network ports to construct SFPs (for more details on how OpenStack aims to implement this capability, please refer to https://docs.openstack.org/networking-sfc/latest/contributor/system_design_and_workflow.html and https://docs.openstack.org/networking-sfc/latest/contributor/api.html).</p>
	<p>[SFC-04] A Service Function (SF) MAY consist of a cluster of VM instances, which can be used for load balancing (please also see 2.2.2.5). The load balancing function MUST have the option to be sticky (i.e. sessions in progress must be sent through the same SF VM instance). The load balancing function MUST also have the option to ensure symmetric return traffic.</p>
	<p>[SFC-05] The MANO VIM MUST be extensible to support the creation ("rendering") of SFPs in conjunction with different SDN controllers and renderers (e.g. OpenFlow, NETCONF, etc.).</p>



	<p>The support of SFC-related requirements by the OpenDaylight SDN controller is described below:</p> <p>https://wiki.opendaylight.org/view/Service_Function_Chaining:Main</p>
	<p>[SFC-06] The MANO VIM MAY support a network overlay function that is part of the NFV infrastructure (OpenStack will provide a reference implementation using Open vSwitch).</p> <p>For a complete implementation of SFC, the MANO framework would need to also support orchestration of the SFC Classifier, Service Function Forwarder (SFF) and SFC Proxy building blocks. For more information on how OpenStack aims to support these SFC functions, please refer to https://docs.openstack.org/networking-sfc/latest/contributor/ovs_driver_and_agent_workflow.html).</p>
	<p>[SFC-07] The MANO VIM SHOULD support orchestration of SFC Classifiers. The MANO VIM MAY offer an implementation of an SFC Classifier that is part of the NFV infrastructure (OpenStack will provide a reference implementation using Open vSwitch).</p>
	<p>[SFC-08] The MANO VIM SHOULD support the orchestration of Service Function Forwarder (SFF). The MANO VIM MAY also offer an implementation that is part of NFV infrastructure (OpenStack will provide a reference implementation using Open vSwitch).</p>
	<p>[SFC-09] The MANO VIM SHOULD support orchestration of SFC Proxies. The MANO VIM MAY offer an implementation of an SFC Proxy that is part of the NFV infrastructure (OpenStack will provide a reference implementation using Open vSwitch).</p>
	<p>[SFC-10] The reference implementation of the SFF, SFC Classifier and SFC Proxy (if available) SHOULD support the preferred SFC encapsulation scheme, NSH (please see IETF draft-ietf-sfc-nsh).</p> <p>Please note that an initial implementation of a subset of the SFC requirements above was made available in OPNFV Brahmaputra, as a combination of OpenDaylight, OpenStack and Open vSwitch:</p> <p>https://wiki.opnfv.org/display/PROJ/Project+Proposals+Service+Function+Chaining</p> <p>An overview of how OPNFV Brahmaputra puts all the pieces together:</p>



<http://artifacts.opnfv.org/sfc/brahmaputra/docs/design/index.html>
 Further progress was achieved, as part of the OPNFV Colorado release:
<http://artifacts.opnfv.org/sfc/colorado/docs/design/index.html>
 Finally, the requirements for supporting VNF Forwarding Graphs are outlined below:
<https://wiki.opnfv.org/display/VFG/Openstack+Based+VNF+Forwarding+Graph>

Table 3: NFV requirements

A1.2 MEC Technical requirements

The following Figure depicts the relevant ETSI MEC architecture. This architecture describes how a MEC environment should be organised, namely regarding the deployment of MEC App on top of a cloud environment, as well as the whole management and orchestration functions to support this operation.

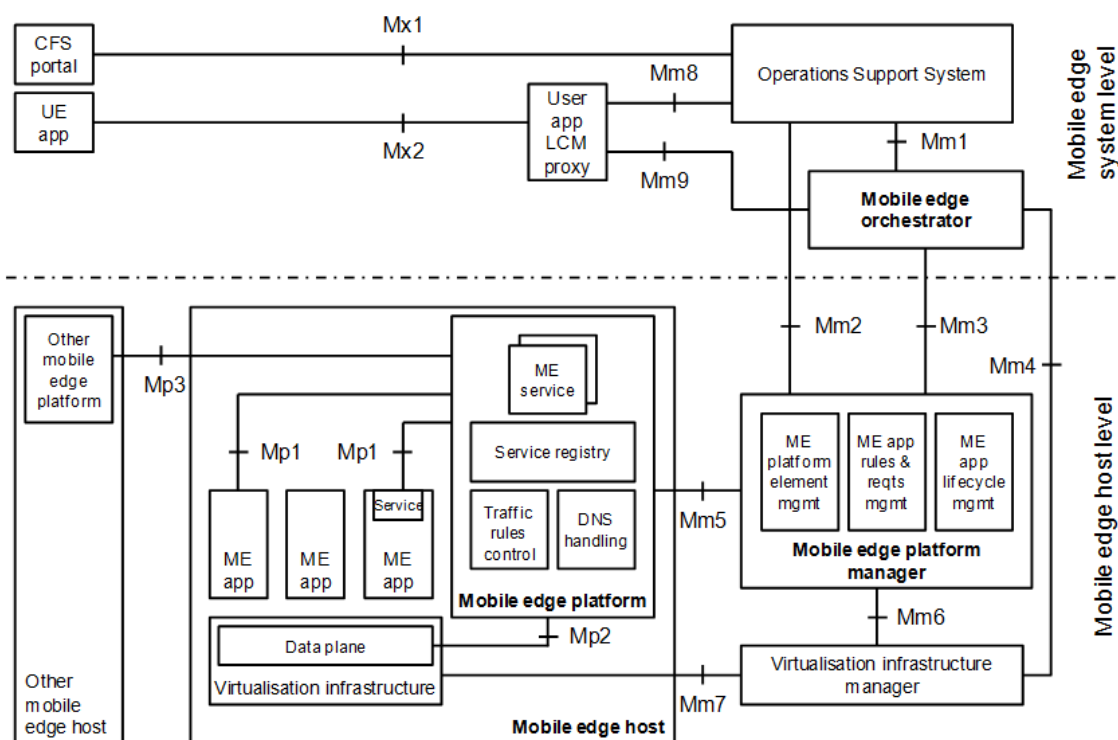


Figure 12: ETSI MEC reference architecture

The next table describes high-level technical requirements for a MEC management and orchestration system.

Type	Requirements description
------	--------------------------



Application lifecycle	<p>[Lifecycle-01] The management system MUST support the following application lifecycle management (LCM) operations:</p> <ul style="list-style-type: none"> ● Instantiation ● Scaling ● Relocation ● Modification ● Termination
	<p>[Lifecycle-02] The management system MUST be able to receive and process application LCM requests:</p> <ul style="list-style-type: none"> ● From the OSS, a third-party, or a UE application ● Based on LCM rules.
	<p>[Lifecycle-03] The management system MUST be able to identify the mobile edge features and services an application requires to run. This will be the input for the decision on which mobile edge host to provision the application.</p>
	<p>[Lifecycle-04] The management system shall support the instantiation and termination of a running application when required by the operator.</p>
Application scheduling and instantiation	<p>[Instantiation-01] The management system MUST be able to deploy the application on mobile edge hosts in various locations, both in a central data center and at the edge of the Core Network.</p>
	<p>[Instantiation-02] The management system MUST support the following deployment application models:</p> <ul style="list-style-type: none"> ● One App instance per MEC Host, serving multiple users ● Multiple App instances per MEC Host, each serving a single user
	<p>[Instantiation-03] The management system MUST support the indication of the following virtualized resources:</p> <ul style="list-style-type: none"> ● Compute ● Storage ● Network resources ● Specific hardware support
	<p>[Instantiation-04] The management system MUST support the indication of the following network connectivity resources:</p>



	<ul style="list-style-type: none"> ● Connectivity to local networks ● External connectivity to the Internet ● Access to user traffic
	<p>[Instantiation-05] The management system MUST support the indication of the following latency requirements:</p> <ul style="list-style-type: none"> ● Maximum ● Expected
	<p>[Instantiation-06] The management system MUST support the indication of physical location (edge).</p>
	<p>[Instantiation-07] The management system MUST support the indication of service requirements:</p> <ul style="list-style-type: none"> ● Mandatory - for MEC Apps to be able to operate. ● Optional - for MEC Apps can benefit from, if available.
	<p>[Instantiation-8] The management system MUST consider cost requirements, which can be a translation of the operator's estimation for the deployment costs.</p>
Mobility support	<p>[Mobility-01] The management system MUST support multiple MEC Hosts in different locations, including radio sites, aggregation points, or at the edge of the Core Network.</p>
	<p>[Mobility-02] The MEC system MUST guarantee service continuity while the UE moves across the network (between different edges and cells).</p>
	<p>[Mobility-03] The MEC system MUST be able to perform application instance relocation for MEC Apps dedicated to a single user.</p>
	<p>[Mobility-04] The MEC system MUST be able to perform application state relocation for MEC Apps serving multiple users.</p>
KPI's support	<p>[KpiTemplate-01] – The system MUST be able to dynamically define a workload deployment template to ensure that resource allocations can support required SLA's and SLO's.</p>



	[KpiScaling- 01] – The system MUST be able to determine the number and types of resources to support workload scaling in order to maintain KPI's and SLO's.
	[Monitoring-01] – The MEC system MUST be able to collect infrastructure and service monitoring information, in order to feed KPI-based automated management and orchestration features.
Network traffic control	[TControl-01] The management system must be able to provide provisioned MEC platforms with guaranteed network bandwidth.
	[TControl-02] The management system must be able to rate limit the provisioned MEC platforms traffic flows.
	[TControl-03] The management system must have the ability to selectively apply the traffic control on different types of traffic, and have the ability of traffic classification.
	[TControl-04] Within the constraints set by the orchestration and management, an authorized mobile edge application shall be able to request the activation, update and deactivation of the mobile edge application traffic rules dynamically.
Scaling	[Scaling-01] – The MEC system MUST be able to scale a MEC App, on OSS request or automatically based on KPIs, in order to increase/decrease the capacity.
	[Scaling-02] The MEC system MUST be able to terminate a MEC App whenever it is no longer required to serve users.

Table 4: MEC Requirements

A1.3 C-Ran Technical Requirements

Delivery D2.3 decomposes C-RAN into RFBs and further discuss the affinity of those RFBs. For completeness, the affinity graph between the different proposed RFBs is presented next.

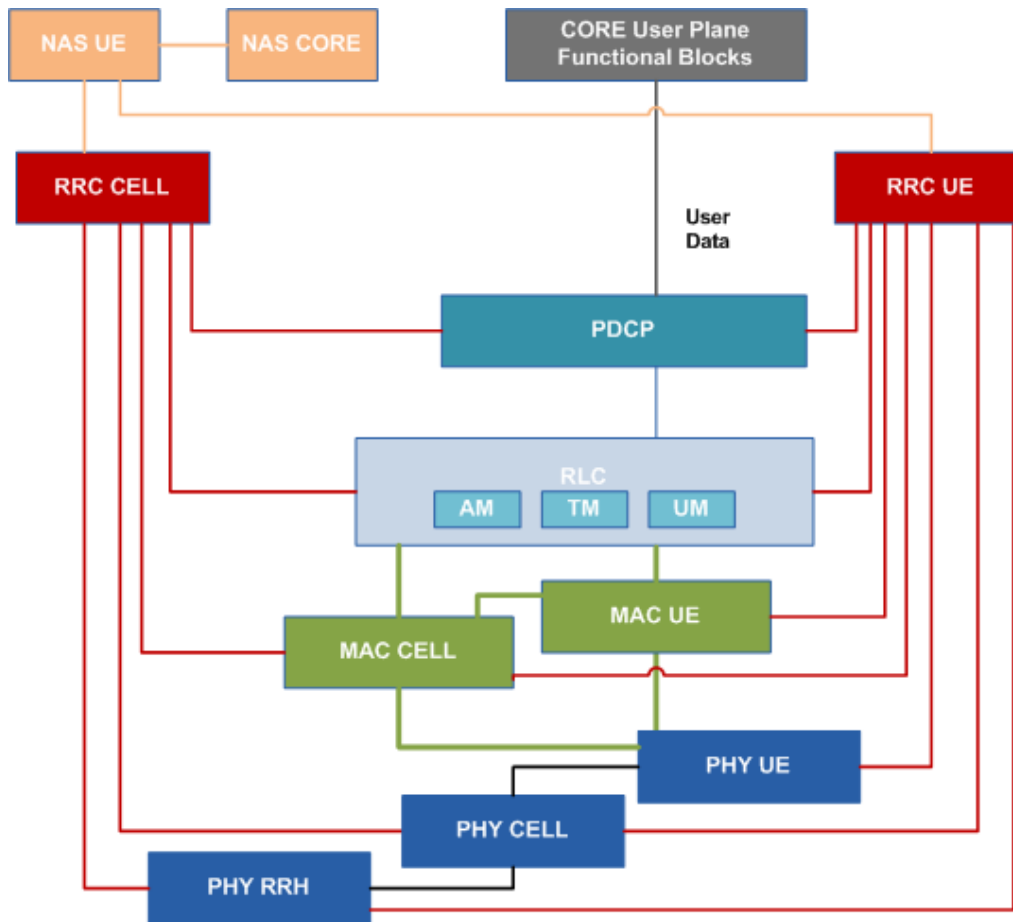


Figure 13: Affinity graph between different C-RAN functional blocks

Here, in the next table, we further analyse the location, event handling capacity and scaling requirements from those functional blocks.

FUNCTIONAL BLOCK (FB)	EXAMPLES OF FB DECOMPOSITION	FB DEPLOYMENT LOCATION	EVENT HANDLING CAPACITY ()	APPLICATION SCALING REQUIREMENT
PHY RRH	Physical NF – not virtualized	Antenna site		Not Scalable as application
PHY Cell	all the processes executed for one cell, e.g. FFT/iFFT, Modulation, Cyclic prefix	Antenna site or Front-End Cloud	every 10 ms (LTE radio frame length)	scaling decision may be reactive (based on computational latency of previous frame). Less than 10 ms requirement.
	Joint Multiuser Detection – jointly process the received		New UE could arrive or leave	Scale in/out may be dependent on UE mobility.



	signals from multiple UE from more than one RAP (MTPD, INS)		asynchronously. Scaling decision should be based on current computational latency and next state prediction	About 5-10 seconds worst case (bus, or train travelling between RAPs)
PHY User (UE)	HARQ must be sent 3 ms after receiving the frame	Front-End Cloud or EDGE cloud	new frame every 10 ms (LTE radio frame length) , but events that results capacity dependent on UE mobility.	Scale in/out may be dependent on UE mobility. About 5-10 seconds worst case (bus, or train travelling between RAPs)
	Convolution coding			
MAC Cell/Scheduling Real Time	ICIC (InterCell Interference Coordination)	Front-End Cloud or EDGE cloud	every 10 ms (LTE radio frame length), Works with a cluster of RAP's, scaling events not coming in peaks.	number of minutes in most cases, dependent on UE mobility. About 5-10 sec
	link adaptive part	antenna site	Dependent on current antenna measurements, need to be executed locally on antenna site, latency sensitive	10 ms If implemented in proactive fashion could be less time sensitive
MAC User (UE)	UE Power control	EDGE cloud	LTE case it can happen maximum 1000 times within a second per ue. capacity is Number of users in 1ms	not coming in peaks, 5-10 seconds worst case
RLC	It includes processes related to segmentation/concatenation of PDCP PDUs based on information exchange with MAC and PDCP. Several	EDGE cloud	<i>depends on the mobility and traffic intensity of UE. For the EDGE cloud slow change in</i>	number of minutes to scale



	modes are supported: Transparent, Acknowledged and Unacknowledged. Each case could be a separate FB		<i>number of ue associated with it.</i>	
PDCP Packet Data Convergence Protocol	transfer of user plane data, transfer of control plane data, header compression, ciphering, integrity protection.	EDGE cloud or Central cloud	Depends on ue activity levels, would change through the day in predictable manner (peak in the morning, less activity in the night, etc)	scaling not strict time constrained, and predictable. number of times in a day
RRC Cell		EDGE cloud or Central cloud		
RRC User (UE)	Handover UE measurements reporting, QoS management, paging	EDGE cloud or Central cloud	about 30% of UE are in the handover state, so with central deployment number of scaling events in a day	scaling not strict time constrained, number of times in a day
NAS User (UE)	It refers to the user procedures related to signaling between the UE and MME	EDGE cloud or Central cloud	Asynchronous, depends on user mobility. Because of deployment on central cloud slow change in number of the users in the whole network	scaling not strict time constrained, number of times in a day
NAS Core	MMEs load balancing, MME overload control, GTP-C signaling load control...	EDGE cloud or Central cloud		

Table 5: C-RAN RFB requirements

A1.4 NFV vs. MEC Comparison

NFV	MEC
-----	-----



NFVO only orchestrates Network Services (NS), not VNFs (for those are VNFM)	MEO orchestrates MEC Apps (MEC has no combination of MEC Apps as NSs combine VNFs)
NFV has no services platform to provide services	MEC has a service platform to provide services to Apps, which must be managed (access, auth, etc.)
The deployment details of NSs (e.g. location) can be decided by the NFVO, but also by the VNFM	The deployment details of a MEC App is only determined by the MEO
Mobility issues are not very relevant (although in some cases may arise)	Mobility issues (state movement) are relevant
Location issues are not always relevant (although in some cases may happen)	Location issues are always relevant

Table 6: NFV vs MEC comparison



A2. State of the art

A2.1 VIM OpenStack Virtual Infrastructure Management

This section provides a summary of the capabilities exposed by the virtual infrastructure, which are relevant to the orchestration layer.

Network Traffic Control

Neutron has become OpenStack's 'networking as a service' de facto project, and provides multiple networking services, QoS is being one of the key features provided. The supported traffic control requirements in Mitaka release are rate limiting answering [TControl-02], and the dynamic activation/deactivation upon request [TControl-04]. However, on the downside the missing features are bandwidth guarantee [TControl-01] and having a more mature traffic classification capability [TControl-03] (e.g. layer 7), with the latter becoming an active discussion at last OpenStack summits.

Scheduling parameters

In order for the orchestration layer to be able of making a 'smart' scheduling decision, the VIM has to expose the required set of parameters for the orchestrator to take into account. However, at this point in time, most of the aren't supported. On the upper side - requirements [AppSched-05] (description of the virtualized resources) can be satisfied by the usage of templates provided by such projects as Heat and Tacker as well as [AppSched-06] (Required network connectivity description). However, on the downside requirement [AppSched-08] (Physical location requirements) is hardly fulfilled. The possible solutions to accomplish that can be by made by the usage of OpenStack's Nova (compute project) regions and cells. For the containers scheduling on Kubernetes/OpenShift, more options are provided and more fine grain scheduling (through labels) can be done.

Mobility support

While the OpenStack Nova (compute) project provides support for a subset of functionality for migrating VM instances from one physical host to another, it lacks some of the properties required for full mobility support: [Mobility-01], [Mobility-02]. The user of the migration feature in its current form cannot specify the physical host on which the VM will be migrated, as this decision is left out to the scheduler. For the container COE, the idea is to quickly recreate the container rather than migrate it.

KPI Support

A KPI is a metric used to evaluate factors that are crucial to the performance of a workload or service. Operationally KPIs act as a simple set of indicators to measure data against -- a sort-of service success gauge. In order to appropriately monitor and measure KPIs requires quantitative and qualitative metrics. These metrics are typically captured through the use of telemetry providing both platform and service level data.



Current service orchestration approaches are based on the use of predefined configurations for the node(s) hosting the workloads. The Orchestrator then requests instantiation of the predefined configuration to bring the workload into service on specific hardware platform, for instance through usage of pre-compiled deployment templates (i.e. OpenStack Heat Orchestration Templates (HOT), TOSCA descriptors, etc.). These templates are managed by orchestration platforms through the use of catalogues, (for instance, OpenStack Murano project can be used to store and manage HOT templates for OpenStack Heat). However, this approach does not scale efficiently. As the number of different services to be supported by the platform increases as well as the granularity of service specific KPIs (Key Platform Indicators) and SLOs (Service Level Objectives) it results in a huge number of deployment templates to supported deployment of services. A more effect approach may be based around the use of dynamic template definitions at deployment time to meet specified KPI's as described in Section A3.9.3.

A2.2 VNFM/NFVO

A2.2.1 Cloudband

Cloudband management system is based on two main components, VNFM (VNF management) and NFVO (NFV orchestrator). In the following we would focus on the VNFM.

Cloudband VNF management system is mostly based on OpenStack and open source services. Specifically, on top of OpenStack main projects (NOVA, Neutron, Cinder and Glance) Heat is utilized for VNF deployment and resource allocation. To further allow VNF lifecycle management we utilized Mistral workflow engine that operates in conjunction with Heat. We note that the selection of a workflow engine for a generic VNF management has been identified as an efficient approach in terms of providing a quite broad generic management capabilities and with relatively low complexity (Odini, Marie-Paule. "Short Paper: Lightweight VNF manager solution for virtual functions." Intelligence in Next Generation Networks (ICIN), 2015 18th International Conference on. IEEE, 2015).

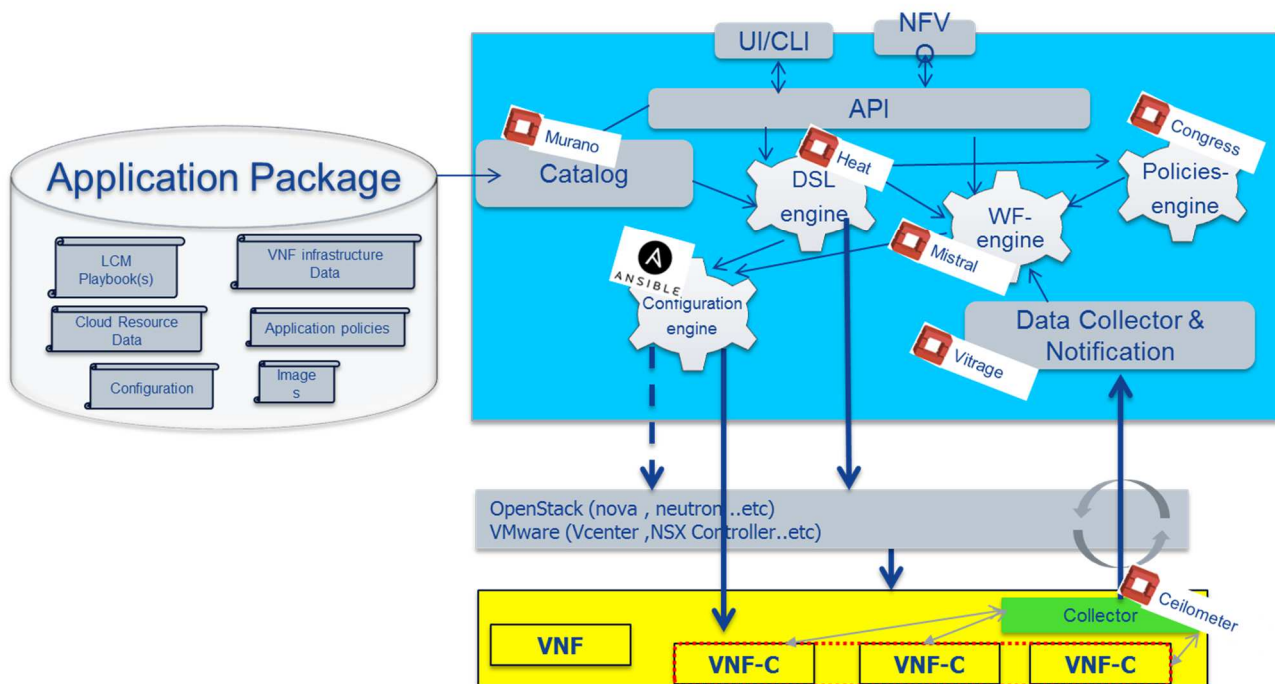


Figure 14: Openstack based generic VNF management system

Figure 14 depicts the architecture for the VNF management system. As indicated, the architecture is based on OpenStack services, such as: Heat, Mistral, Murano, Ceilometer, Vitrage and possibly Congress. In addition, it utilizes Ansible as an open source configuration management. This architecture can support all of the operations that are required for a VNF lifecycle management, including deployment, monitoring, scaling healing and termination (as depicted in Figure 15).

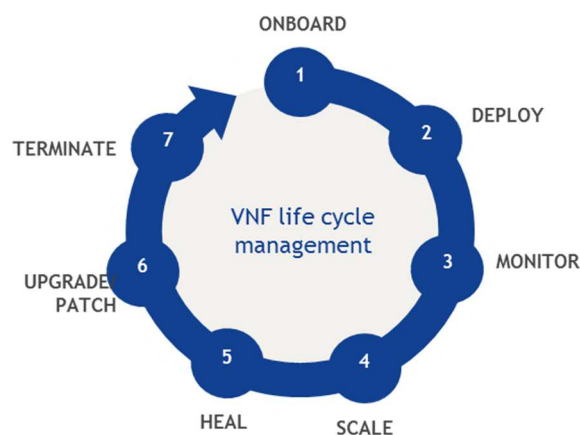


Figure 15: VNF lifecycle operation



For example, Deployment takes place once the onboarding process is complete. Deployment entails ensuring that the newly-introduced application is deployed with its name and the correct environment, on the correct VMs, with the right IPs, etc.

After the onboarding process is complete, the second LCM stage—Deployment takes place (Figure 16).

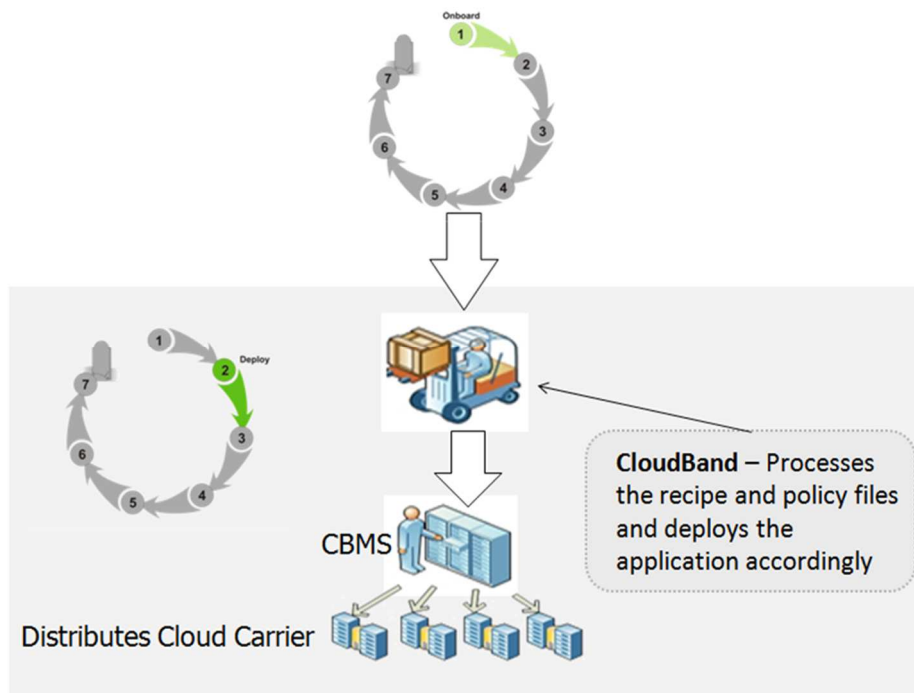


Figure 16: The deployment workflow

Only the customer user can deploy applications. There are two ways to deploy:

- From the Catalog (add application blueprint to the Catalog specified in onboarding)
- Direct deployment of Deploy Stack Directly on OpenStack Node

The HOT template is validated by OpenStack during deployment. No validation is performed when the HOT template is onboarded.

After an application is deployed, a service will be created in the MY CLOUD > DEPLOYMENTS. Under the service the customer user can see the stacks of the application.

For each deployment, a job will be created.

For the deployment to succeed, one should ensure that the Hot is valid and that all the required resources for the stack are on the node (for example, the image).



A2.2.2 OpenMano

OpenMANO implements components from the ETSI NFV MANO stack. Currently, the situation with regards to the requirements outlined in this Annex is the following:

Network Traffic Control

OpenMANO supports the definition of link parameters in the VNFD descriptor as well as in the Network Scenario Descriptors (NSDs). They include the type of link (point-to-point, LAN-type, etc.) as well as quality of service parameters

Scheduling parameters

Currently, OpenMANO does not support scheduling internally. However, the OpenMANO component in the OpenMANO project controls a VIM where NFV services are offered including the creation and deletion of VNF templates, VNF instances, network service templates and network service instances using the openmano API. This can be used by other components to implement scheduling.

Mobility Support

Currently, OpenMANO concentrates on creating NFV-based scenarios. As such, the VNFDs are static and do not provide hooks to define mobility for the virtual machines (VMs) that are included in a VNFD.

KPI Support

OpenMANO offers a northbound interface, based on REST ([openvim API](#)), where enhanced cloud services are offered including the creation, deletion and management of images, flavours, instances and networks. The implementation follows the recommendations in [NFV-PER001](#).

A2.2.3 Open Baton

Open Baton (<http://openbaton.github.io>) is an ETSI NFV compliant MANO framework. It enables virtual Network Services deployments on top of heterogeneous NFV Infrastructures. In the release 3, Open Baton significantly increased the number of available components that are part of the ecosystem and included new functionalities for simplifying the way Network Service developers deploy their services.

Open Baton is easily extensible. It integrates with Openstack infrastructure and provides a plugin-based mechanism for supporting additional VIM types. It supports Network Service orchestration either using a generic VNFM or interoperating with VNF-specific VNFM. It uses different mechanisms (REST or PUB/SUB) for interoperating with the VNFMs. It integrates with additional components for the runtime management of a Network Service. For instance, it provides auto-scaling and fault management based on monitoring information coming from the monitoring system available at the NFVI level.

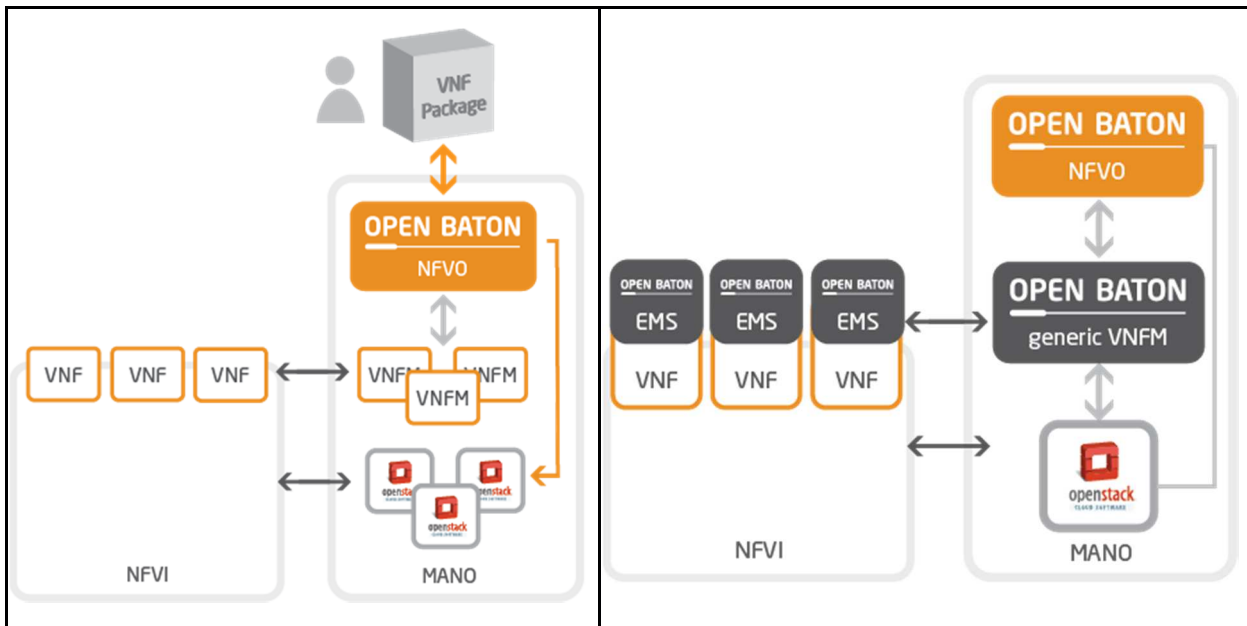


Figure 17: OpenBaton integration into OpenStack

The Figure below depicts the Open Baton architecture.

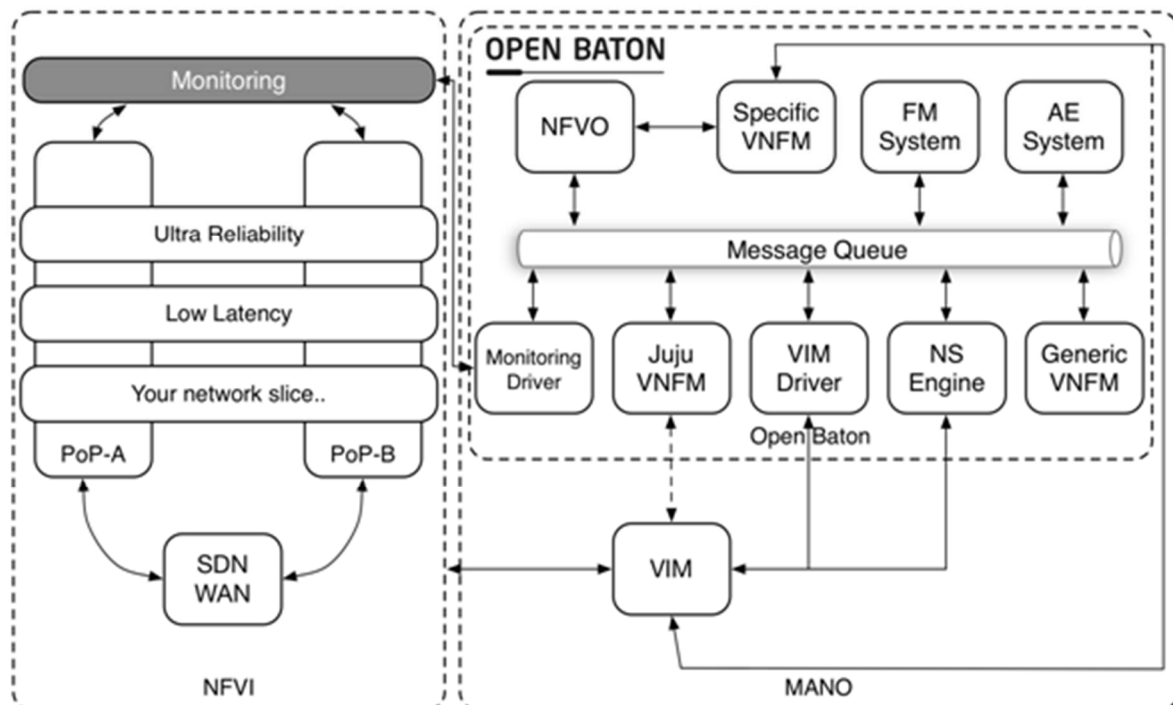


Figure 18: Open Baton architecture



A2.2.3.1 Features

The Open Baton implementation includes the following list of features:



- A Network Function Virtualization Orchestrator (NFVO) completely designed and implemented following the ETSI MANO specification.
- A generic Virtual Network Function Manager extendable (VNFM) able to manage the lifecycle of VNFs based on their descriptors. The Generic VNFM can execute the following operations:
 - Request to the NFVO the allocation of specific resources for the virtual network instance (using a granting mechanism)
 - Can request from the NFVO the instantiation, modification, starting and stopping of the virtual services (or directly to the VIM)
 - Instructs the generic Open Baton EMS to save and to execute specific configuration scripts within the virtual machine instances
- A Juju VNFM Adapter in order to deploy Juju Charms or Open Baton VNF Packages using the Juju VNFM.
- A driver mechanism for adding and removing different type of VIMs without having to re-write anything in your orchestration logic.
- A powerful event engine useful based on a pub/sub mechanism for the dispatching of lifecycle events execution.
- An auto-scaling engine which can be used for automatic runtime management of the scaling operation operations of your VNFs.
- A fault management system, which can be used for automatic runtime management of faults which may occur at any level.
- It integrates with the Zabbix monitoring system.
- A set of libraries (the openbaton-libs) which could be used for building your own VNFM. A Marketplace useful for downloading VNFs compatible with the Open Baton NFVO and VNFMs.
- A user-user friendly dashboard, which enables the management of the complete environment.
- It provides also a set of mechanisms, which enable the support of external VNFMs. This can be done in the following ways:
 - Publish/Subscribe mechanism using a message queue based on AMQP.



- REST APIs.
- Open Baton integrates via Plugins to different VIM. By default an OpenStack plugin is provided.
- Docker-based Element Management System
 - It is possible to instantiate a VNF on top of a docker container using the GenericVNFM and VNF Package approach;
 - The compute node need to use the nova-docker driver.
- Identity Management:
 - Possibility of defining different projects;
 - Possibility of registering users and assign them different roles;
 - Possibility of registering users and assign them to different projects.
- Network Slicing Engine (NSE)
 - The NSE instantiates rules on physical networks for allocating dedicated bandwidth as per Network Service specific requirements;
 - The NSE provides an abstracted view of the inter-datacenter networking topology allowing the instantiation of guaranteed bandwidth levels on top of the physical network elements.

A2.2.3.2 Evaluation

Strongest Points:

- Aligned with NFV;
- TOSCA aligned with ETSI NFV;
- Ability of auto-healing ;
- Ability of auto-scaling with configuration;
- Use the QoS capabilities of Mitaka to make network slicing;
- Work with docker or VMs, depending on openstack configuration;
- Extendable in the most of components.



Weakest Points:

- Does not work with the more recent release of openstack, it follow the openstack releases with delay of one or two (now Mitaka with release 3);
- The documentation is not sufficient for all the features;
- The Open Baton EMS needs to have inside the VM, and act as an agent.
- It isn't stable (tested on release 2);
- Have performance issues in auto-scale mechanism (tested on release 2).

A2.2.4 OSM

Open Source MANO (OSM) (<https://osm.etsi.org>) is the ETSI open source community which aims to deliver a production-quality MANO stack for NFV, capable of consuming openly published information models, available to everyone, suitable for all VNFs, operationally significant and VIM-independent. OSM is aligned to NFV information models, while providing first-hand feedback based on its implementation experience.

The OSM implementation is built on top of 3 main components:

1. RIFT.ware, a service orchestrator (NSO).
2. OpenMANO, a resource orchestrator (RO).
3. Juju, a configuration manager (CM).

The integration of those components is the main job of the OSM developers. The Figure below depicts the basics of this integration.

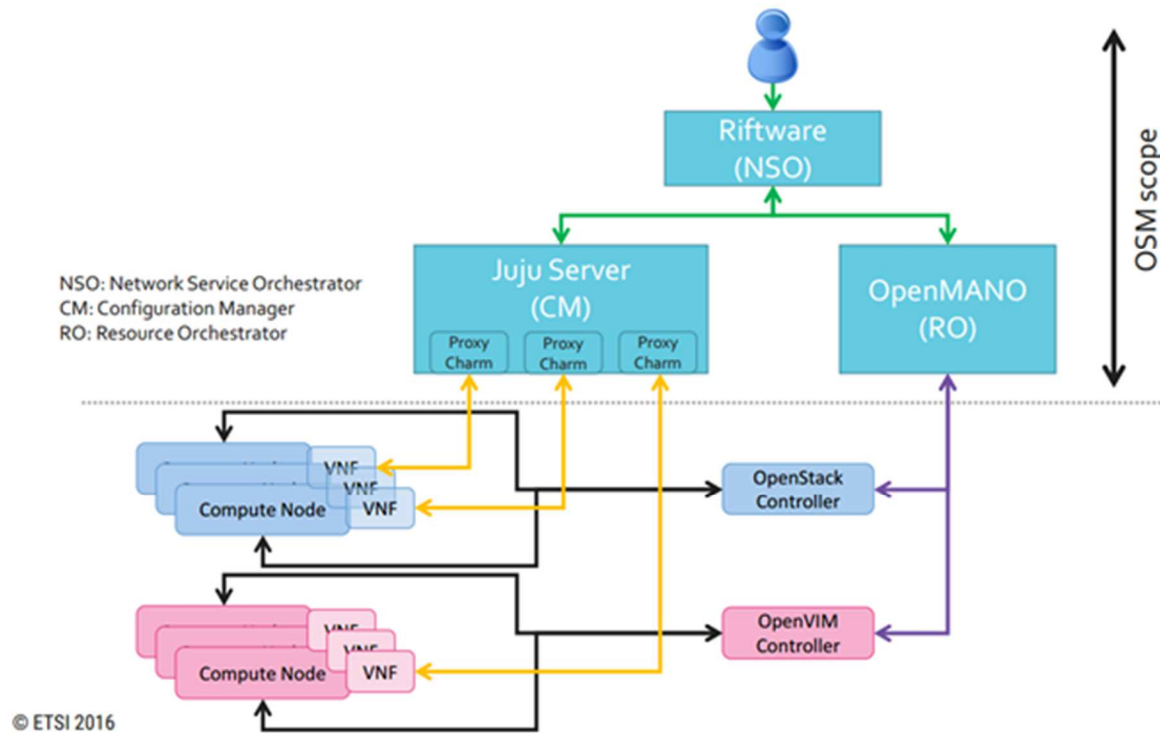


Figure 19: OSM Architecture

A2.2.4.1 Features

The OSM implementation includes the following list of features (based on R2):

- On-boarding experience & VNF Packaging;
 - Allow cloud-init configuration;
 - Create networks at VIM;
 - Remove NSD datacenter network reference;
 - Detailed feedback when deploying and configuring & Error Message from RO & VCA;
 - Distinction between template, particularization and instance for NS;
 - Composer should display descriptors in YAML format;
 - Enhance Visual Differentiation Between NS Catalog and VNF Catalog;
 - Restructure layout of service primitive page;
 - Package creation command line utility;
- EPA based resource allocation;
 - Not explicitly captured;



- May be included as part of an upgrade to OpenStack Mitaka, likely push to R2 timeframe;
- (Networking) Service Modelling;
 - Juju-2.x;
 - Network types in RO;
 - Allow IP parameters for networks;
 - Configuration/Service Primitive model enhancements;
- Multi-VIM;
 - New VIM connector for VMware vCloud Director;
 - Openvim as reference VIM with EPA capabilities;
 - Datacenter capabilities;
 - Support for VIM Accounts;
- Multi-Site;
 - Multi-site NS.

A2.2.4.2 Evaluation

Strongest Points (OSM R2):

- Is in active development by ETSI group;
- ETSI NFV aligned implementation;
- Roadmap well defined and going in the right direction;
- It works with the more recent release of RIFT.ware (version 4.3.3);
- Open Source community behind;
- Large documentation available (although sometimes not enough);
- Integration with (external) monitoring.

Weakest Points (OSM R2):



- Limited TOSCA adoption, resorting to YAML (NSD and VNFD);
- The documentation is not complete, missing complex examples;
- Limited set of functionalities (even less than RiftWare alone, one of the pieces);
- Some limitations identified, e.g.:
 - Missing an integrated monitoring;
 - Not supporting scaling operations;
 - Not supporting VIM images onboarding;
- Evolutions and corrections come slowly sometimes;
- Not very stable and reliable yet.

A2.2.5 Cloudify

Cloudify (<http://getcloudify.org>) is an open source cloud orchestration framework that allows users to model applications and services and automate their entire life cycle, including deployment on any cloud or data center environment, monitoring all aspects of the deployed application, detecting issues and failure, manually or automatically remediating them and handle ongoing maintenance tasks. The Figure below depicts the implementation architecture.

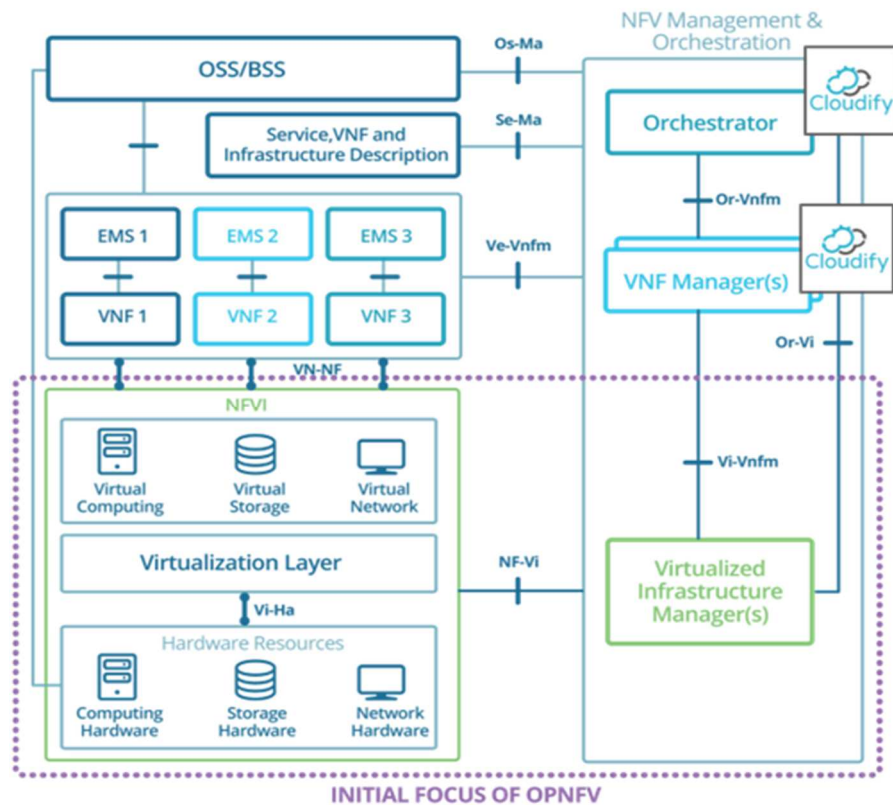


Figure 20: Cloudify architecture.

The VNFM (Virtual Network Function Manager) is responsible for the VNF lifecycle management - e.g. it takes action on instantiation, termination, failover, scaling in and out, and more. Cloudify serves as a generic VNF manager (G-VNFM) and enables full automation of all lifecycle stages for any network function.

The NFVO (NFV Orchestrator), as its name implies, basically serves the purposes of orchestrating and managing end to end network services, through the complex NFV architecture, including integration with SDN controllers, OSS/BSS systems, and more. Cloudify provides a fully open NFVO.

The Cloudify solution is based on the integration of many basic components for data storing, messaging, logging, monitoring and many others. The Figure below depicts this ecosystem.

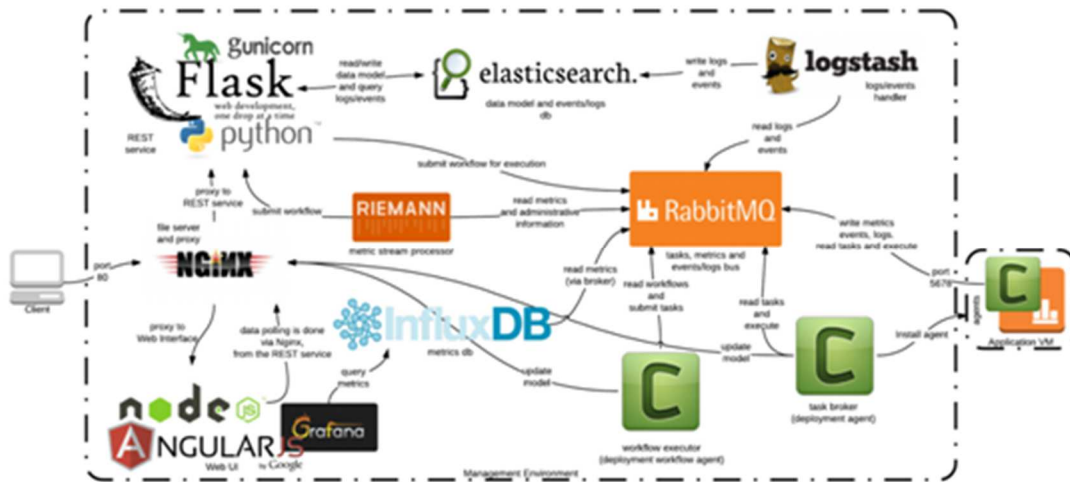


Figure 21: Cloudify components integration

A2.2.5.1 Features

The Cloudify implementation includes the following list of features:

- Full Application Lifecycle Orchestration with Cloudify:
- Pure-play orchestration of applications on the cloud:
- Infrastructure & Tooling Agnostic:
- NFV Orchestration: The only pure-play technology agnostic MANO implementation:
- Hybrid Cloud & Workloads with Cloudify:
- Cloudify Simplifies Cloud Migration & Portability for Enterprises:
- Blueprint Composition:
- Cloudify Telecom Edition: The Telecom Edition is an NFV-specific offering of Cloudify's TOSCA-based orchestration framework. This is the most advanced open source NFV MANO offering and includes built-in multi-VIM support with added extensibility to hybrid workloads and micro services. Included in this release:
 - Multi-VIM - Now fully open source vCloud and vSphere plugins, alongside OpenStack native support;
 - Overlay Service Chaining;
 - Netconf plugin;
 - TOSCA/YANG data modelling interoperability;
 - Network service management;
 - Clearwater vIMS blueprints and plugins;
 - F5 BIG-IP plugin;



- VNF updates for running VNFs and services;
- NIC ordering;
- Enhanced Platform Awareness coupled with Data Plane Acceleration through integration with Intel;
- Drag and drop Cloudify Composer with VNF-specific components;
- Using ARIA as the kernel for TOSCA Orchestration;
- Support installation within environments with no internet access;
- Support for Cloud Native services through Kubernetes plugin.

A2.2.5.2 Evaluation

Strongest Points:

- Active development by GigaSpaces and will be upgrade in future releases, with public roadmap;
- Well-defined roadmap;
- Overall mature solution;
- TOSCA-based (although non-NFV compliant);
- Multi-Vim;
- Support for containerized and non-containerized VNFs;
- Overlay Service Chaining;
- Built-in auto-healing and auto-scaling policies;
- VNF updates for running VNFs and services;
- Metrics Queuing, Aggregation and Analysis.

Weakest Points:

- Roadmap defined by a single organization;
- It not work with the more recent release of openstack, it follow the openstack releases with delay of one or two (now Liberty with version 3.4);
- It's not multi-tenancy;
- Composer and Web UI are premium (not included in the free distribution).



A2.2.6 Tacker

Tacker is the Openstack project devoted to cover the Management and Orchestration functions of the ETSI NFV architecture (MANO). The main purpose is to manage the lifecycle of VNFs and orchestrate NSs. The Tacker basic architecture is depicted in the following Figure.

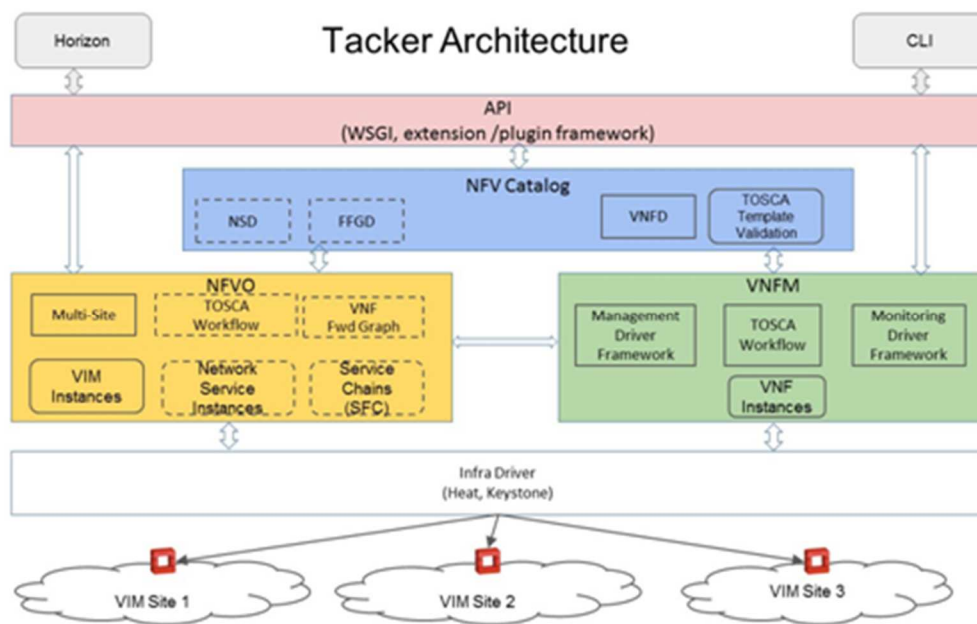


Figure 22: Tacker Architecture

A2.2.6.1 Features

The Tacker implementation includes the following list of features:

- Tacker VNF Catalog
 - Repository of VNF Descriptors (VNFD)
 - VNF definition using TOSCA templates
 - Support for multiple VMs per VNF (VDUs)
 - Tacker APIs to on-board and maintain VNF Catalog
 - VNFDs are stored in Tacker DB
- VNFD using TOSCA
 - Describes the VNF attributes
 - Glance image IDs
 - Nova properties
 - Security Groups



- Performance Monitoring Policy
 - Auto-Healing Policy
 - Auto-Scaling Policy
 - Working with Heat-Translator
- VNF Auto Configuration
 - Tacker provides a Management Driver Framework
 - Facilitates VNF configuration based on Service selection
 - Inject initial configuration using:
 - Config-drive
 - custom mgmt-driver (connect using ssh / RESTapi and apply configuration)
 - Update configuration in active state
 - Extendable
- VNF Self-Healing
 - Tacker health check starts as VNF becomes ready
 - Ongoing network connectivity check
 - Auto-restart on failure (based on VNFD policy)
 - Extendable Vendor and Service specific Health Monitoring Driver framework
- VNF Auto-Scaling
 - Auto-Scale VNF based on policy
 - Continuous performance monitoring according to KPI described in VNFD
 - Basic Auto-Scaling using common VM metric
 - CPU threshold
 - Custom Monitoring Metric
 - Alarm-based monitoring driver using Ceilometer
 - Manual-scaling option to scale in/out the VDUs
- Multisite VIM Usage
 - Manage multiple Openstack sites
 - Deploy VNFs in multiple OpenStack sites
- VNF Forwarding Graph
 - TOSCA NFV Profile based FG Descriptor can be uploaded to VNF-FGD Catalog



- VNF-FFGD template describes both Classifier and Forwarding Path across a collection of Connection Points described in VNFDs
- Recently NS support was introduced
 - NSD Catalogue
 - NS instantiation

A2.2.6.2 Evaluation

Strongest Points:

- Active development in openstack environment and will be upgrade in every openstack version;
- Roadmap well defined and large community involved;
- Aligned with Openstack releases;
- Good documentation;
- Aligned with NFV;
- Uses TOSCA;
- Integration with other tools (ManagelQ);
- Auto-healing and auto-scaling capabilities;

Weakest Points:

- Still immature and unstable;
- VNFFG and NS only released very recently;
- Basic scaling features;
- Only support Openstack as VIM;
- A few examples available;

A2.2.7 ManagelQ

ManagelQ is an open-source project that allows administrators to control and manage today's diverse, heterogeneous environments that have many different cloud and container providers and/or instances spread out all over the world. Thus it is a higher layer that build upon the VIM management layer, making it a candidate for the NVFO scope. Its main advantage is that it provides a single pane of glass for all the deployments, simplifying management as well as helping to have a global view of the multi-site system.



A2.2.7.1 Features

- Continuous Discovery: ManagelQ is able to connect to virtualization, container, network and storage management systems and discover their inventory, map relationships, and listen to changes.
- Self-Service: ManagelQ defines bundles of resources and publish them in a service catalog. Users can order them from there and manage the full life cycle of a service, including policy, compliance, chargeback/showback and retirement.
- Compliance: ManagelQ may scan the contents of your VMs, hosts and containers to create advanced security and compliance policies.
- Optimization: ManagelQ captures metrics from the different providers, allowing to better understand the current utilization and normal operating ranges. This data may be used to find unused or overbooked systems, get right-sizing recommendations, do capacity planning, or run what-if scenarios.

A2.2.8 Evaluation

Strongest Points:

- Support for several VIM types and instances;
- Good documentation;
- Integration with other tools;
- Supports both VMs and containers;
- Self-service capabilities;

Weakest Points:

- Not aligned with NFV;
- It doesn't make direct use of TOSCA descriptors;

A2.3 Comparison between Orchestrators

Out of all the options covered above, in this section we compare the two main orchestrators that are being considered for the Superfluidity framework: *OSM* and *ManagelQ*. Because ManagelQ is not compliant with the NFV architecture, it's been analysed in conjunction with other tools such as OpenStack Tacker. In both cases OpenStack has been used as a VIM, though ManagelQ also supports Container Orchestrator Engines, such as Kubernetes or Openshift.



Both orchestrators fulfil the NFV requisites previously mentioned with the exception of scaling already deployed VNFs, specific hardware support and in the case of OSM, there is no consideration of costs previous to deployment (while the deployment in ManageIQ has to be approved by an administrator). Also, in both cases, the creation of Service Function Chains is under recent development.

In the context of Superfluidity there is however another requirement, the use of containers as Virtual Deployment Units. While the ETSI NFV architecture does not require VNFs to be deployed only as virtual machines, most NFVOs such as OSM only contemplate this possibility and are not suited for Superfluidity scenes where the orchestration of containers is required. ManageIQ on the other hand, does allow deploying containers, but Tacker does not. Which means that it is possible to deploy containers using ManageIQ but not using TOSCA descriptors or following the NFV philosophy.

The solution to this problem involves a Superfluidity contribution to ManageIQ and is approached in a later section of this document.

Regarding OSM, in the context of the Superfluidity project, containers support (e.g. containers creation and removal) is achieved by the definition and execution of Juju charms by the Juju Server.

A2.4 Management and Orchestration Design

This section intends to identify and describe the different available options regarding cloud infrastructure, cloud infrastructure management and orchestration. We also discuss the pros and cons and the best approaches to be followed by the project.

A2.4.1 Cloud Infrastructure

The cloud infrastructure is the basis of the emerging cloud technology. It allows to create isolated virtual entities, with compute, storage and networking capabilities, appearing as if they were physical machines. The use of hypervisors (e.g. KVM, ESX) is still the most common virtualization technology. However, container-based technologies (e.g. Kubernetes, Dockers*) are getting momentum. ETSI NFV refers to this as NFV Infrastructure (NFVI); we will use this term from now on.

Independently of the virtualization technology in use, some architectural aspects need to be discussed and decided, in the context of the project, in order to find the best approach that fits with our requirements. Superfluidity shall support two different types of services: network functions (e.g. eNB, EPC) and applications (e.g. MEC).

Option 1: One NFVI per Service

The easiest way to support different services is to use a separated cloud infrastructure (i.e. servers, storage, network) (see Figure 23). However, this leads to an inefficient use of resources, as there are



no synergies between similar infrastructures. Furthermore, for an operator, the management effort is considerably larger, as isolated silos need to be built.

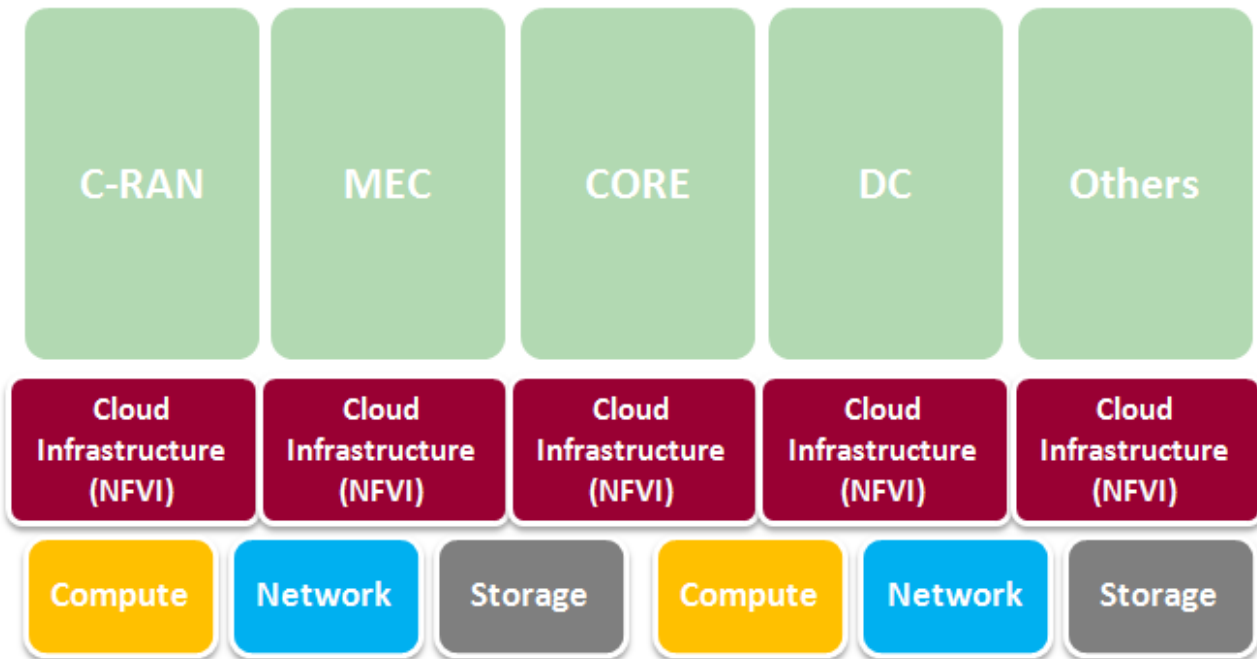


Figure 23: One NFVI per service

Conclusion: Inefficient and complex

Option 2: Common NFVI for all Services eventually locations

To increase efficiency and reduce complexity, it is preferable to have a common infrastructure, which can be used to hold all kinds of services, eventually even in multiple locations (see sections below). For this to be possible, it is required to ensure that all services can rely on similar infrastructure standards. After some discussions among service specialists, we were not able to identify any service specificities that prevent this approach. For this reason, it seems that the best strategy is to have a common cloud infrastructure (NFVI) for all services. This model increases efficiency and simplifies management. The next Figure depicts this view.

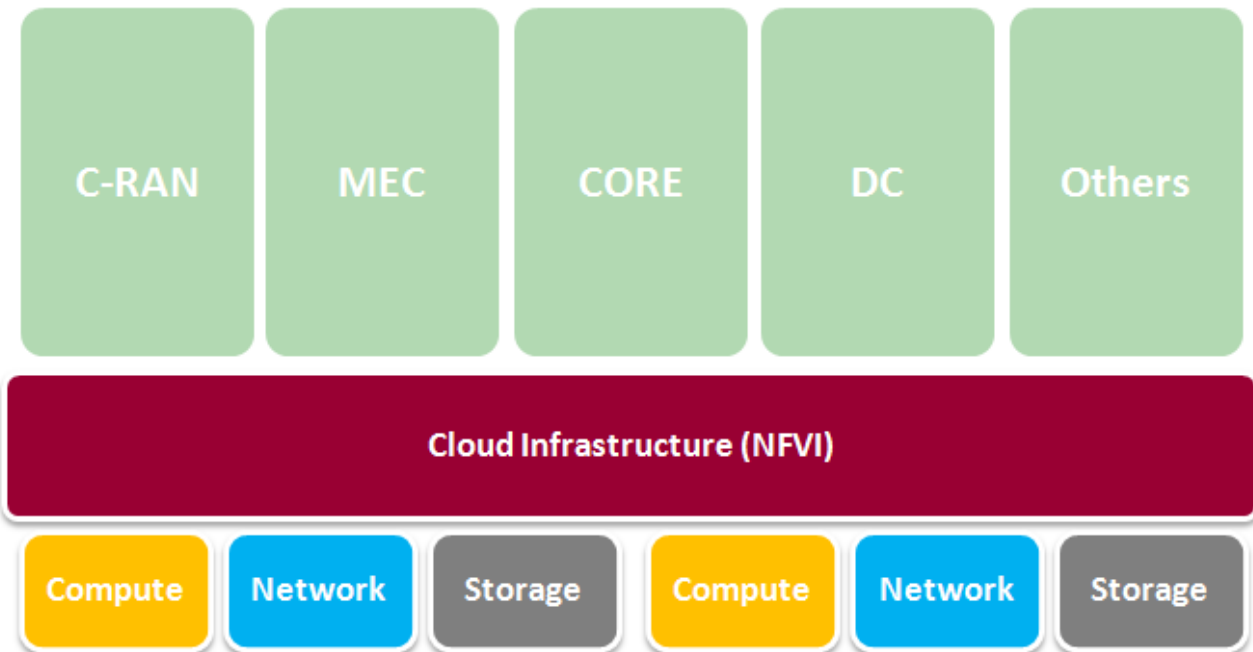


Figure 24: Common NFVI from all services

Conclusion: Preferred

A2.4.2 Cloud Infrastructure Management

To manage a cloud infrastructure (NFVI) a controller/manager is required. This manager is responsible to interact with the hypervisors and provide users with the capacity to manage (create, remove, update and delete virtual resources (compute, storage, network). ETSI NFV refers to this as Virtual Infrastructure Management (VIM); we will use this term from now on. Today, the reference for this component is the open source OpenStack solution. Although there are others like OpenVIM, OpenStack is clearly a de facto standard.

Assuming that a common NFVI can support all services (see section above), it is important to define the strategy to efficiently support the management of resources spread across a large number of datacenters (core and, especially, edges). As described below, there are several options, each with pros and cons.

Option 1: One local VIM per NFVI

The simplest and most common approach is to use one VIM per NFVI, i.e. one manager/controller per cloud infrastructure (datacenter). Following this approach, the VIM function is deployed locally on the datacenter (e.g. edge) and manages all NFVI resources located there (see next Figure). This has the advantage of being a well-known and resilient approach, as inter-datacenter connectivity is not required. However, it has two main disadvantages. Firstly, this may lead to a large number of



VIMs, making the life of the upper Orchestration layer more complex, as it needs to interact with multiple VIMs endpoints. Secondly, the use of multiple VIMs may prevent the use of some capabilities like “VM live migration” among different locations, which may be an important feature. Up to now, it is not clear whether this feature is required and has advantages when compared to other models (e.g. service migration at Orchestration level). Some work still needs to be done to evaluate this.

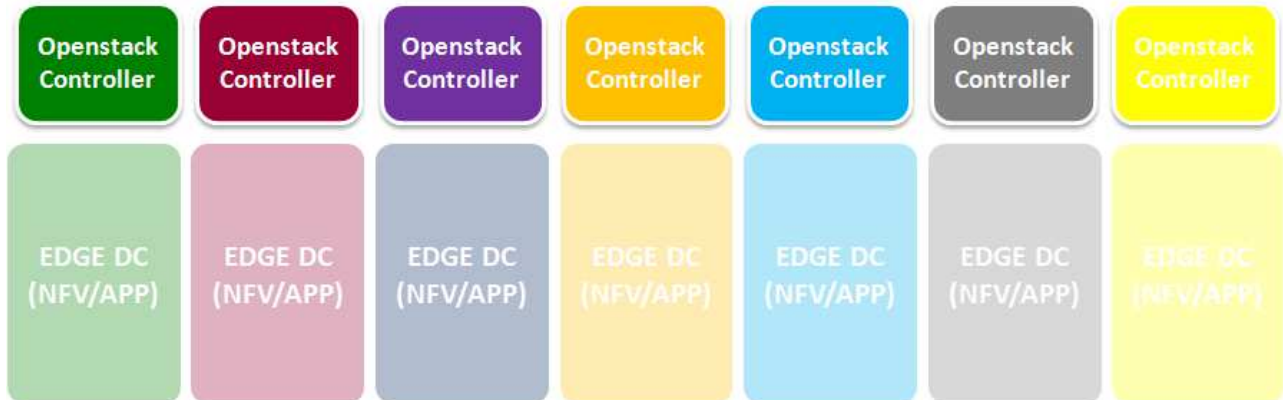


Figure 25: One local VIM per NFVI

Conclusion: Acceptable

Option 2: Single centralized VIM for all NFVIs

The use of a single VIM for all NFVIs located in multiple datacenters (core and edges) is another option to consider. In this case, a single centralized VIM is able to manage all resources located in different locations, providing an external view of a single and federated large datacenter (see next Figure). The different locations can be identified, when needed, based on regions. This approach has the advantage of simplifying the life for the Orchestration layer, as it has a single VIM as endpoint, where all resources can be requested. On the other hand, it allows the use of features like “live migration”, only possible within the same VIM domain, as referred above. However, it has also some disadvantages. From one side, it makes the VIM operation more complex, as it needs to manage a large amount of resources and locations. Furthermore, there may exist some limitations on the number of managed resources. Finally, the manager/controller is no longer local to the NFVI, resulting in traffic increase and delay for the actions to be taken, making also appropriate connectivity a requirement. Anyway, today this seems to not be a hard limitation, as services today already are highly connectivity dependent.

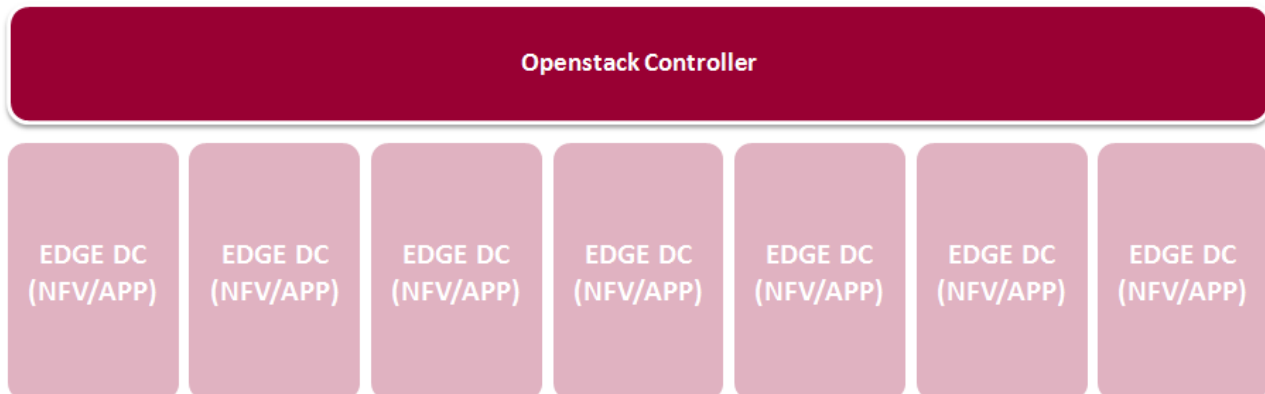


Figure 26: Centralized VIM for all NFVIs

Conclusion: Acceptable

Option 3: Hybrid Option 1 and Option 2

There is still a compromise approach between the two options referred above. In this hybrid approach, multiple datacenters (NFVIs) are grouped into zones and managed by a single VIM (see Figure 27). This option intends to take advantage of the best of both worlds, overtaking some limitations. The group sizing needs still to be defined, but it may depend on a case by case. Compared to option 1, it reduces the number VIM endpoints to a more reasonable number, making the Orchestrator's task easier. On the other hand, it allows users to take advantage of features like "live migration" within the same zone; if groups are properly defined, it can lead to a good tradeoff. Compared to option 2, it can reduce overall complexity and overtake any resource management limitations. In this option, the manager is also no longer local to the NFVI; however, this seems not today a hard limitation as stated above.

Note that in the two extreme cases, this solution is similar to the previous options. If groups are very small, we may lead to groups of a single NFVI, meaning Option 1. On the other extreme, large groups may lead to a single group, meaning Option 2. With this flexibility, it is reasonable to consider this the best option.

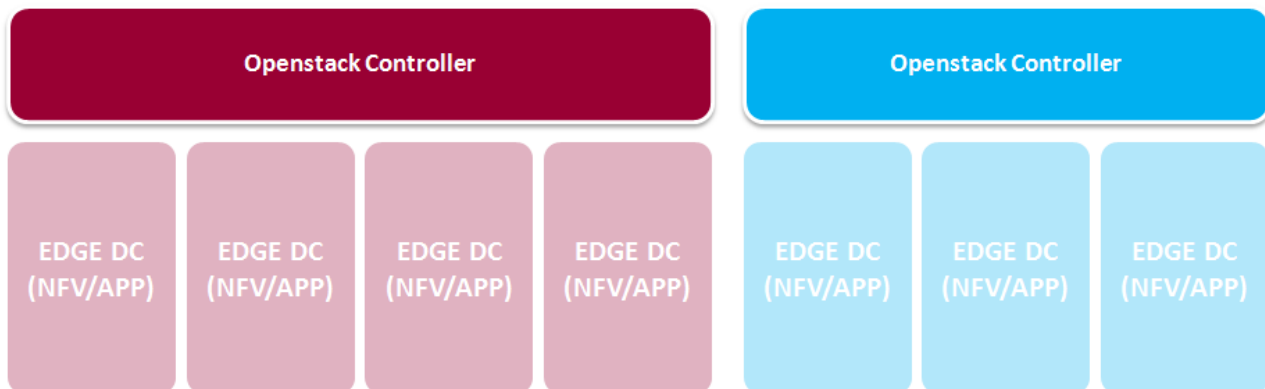


Figure 27: Hybrid Option

Conclusion: Preferred

A2.4.3 Cloud Management and Orchestration

Running on top of Cloud Management, the Orchestration layer is responsible to build complex services by combining and interconnecting the required pieces, on the right locations. Among other things, the Orchestration is able to select the appropriate resources in the right place, based on predetermined constraints. For this, it requires interaction with VIMs. However, as Orchestration can be a very complex task, it will not be simply a single piece, but a set of them, dealing partially with the Orchestration tasks. This section discusses some Orchestration strategies and how do they map to the Infrastructure Management (VIMs).

Option 1: One Orchestrator for all Services and locations

A simple approach to orchestrate all Services in all locations is to use a single Orchestrator. One multi-purpose Orchestrator can deal with all resources and has the advantage of having an overall view of all services, taking eventually advantage of some synergies from that. This model is depicted in next Figure. However, the Orchestrator needs to deal with Service specificities and it may be hard to have a common Orchestrator to handle all that. On the other hand, in real world, different vendors provide different Services, and it is very likely each one brings its own Orchestration for his particular Service. In that case, this solution can be hard to achieve, both technically and commercially.

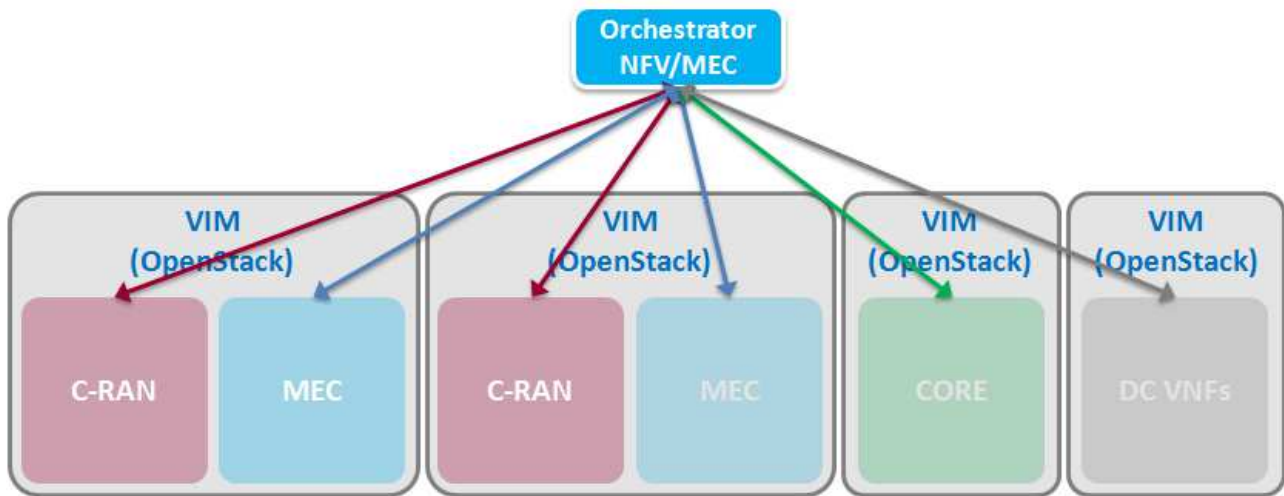


Figure 28: Single orchestrator

Conclusion: Non-realistic

Option 2: One Orchestrator per Service

Another approach is to use different Orchestrators to comprise the overall Orchestration layer. In this case, each Orchestrator is in charge of part of the overall Orchestration tasks (see next Figure). As state above, a dedicated Orchestrator per Service seems a realistic approach; however, other options may also be reasonable. For example, if a vendor provides the C-RAN and the Core, maybe a single Orchestrator can take care of both. Similarly, if an operator has multiple C-RAN vendors, which is common, different Orchestrators may be needed for the same service, one for each particular vendor.

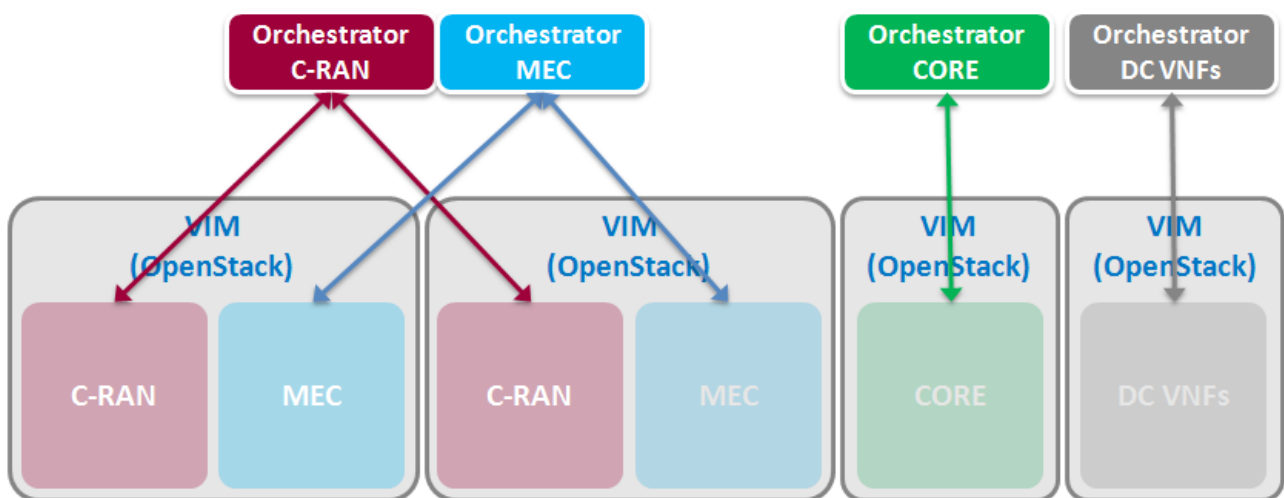


Figure 29: One orchestrator per service



Conclusion: Preferred

A2.4.4 Orchestration Layer

As described in the section above, the Orchestration layer may be composed by multiple Orchestrators, each of them devoted to a particular part of the Orchestration tasks/domains, namely to a particular Service (and from a particular vendor). In this situation, it is relevant to discuss how these Orchestrators can talk to each other and how an operator can have a global view and control about the Services. This section discusses the available options and interfacing models that can be used for this purpose.

Option 1: Northbound and Southbound Interfaces

One possible option leads to the creation of a Top Orchestrator, which integrates all the Service Orchestrators. In this case, Service Orchestrators interact with the Top Orchestration using a Northbound interface (Southbound interface from the Top Orchestrator perspective). For this option, the interaction between Service Orchestrators is not required, as everything is coordinated via the Top Orchestrator. Here, the Operator will own the Top Orchestrator and must integrate it with all Service Orchestrators. The next Figure depicts this hierarchical model.

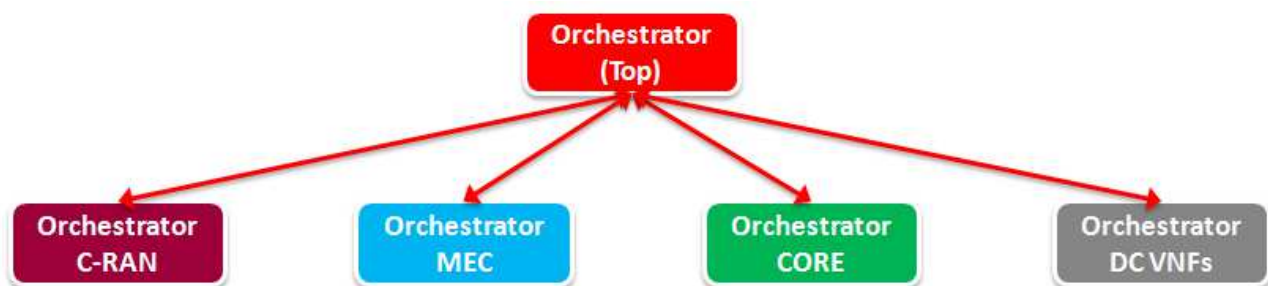


Figure 30: Top orchestrator

Conclusion: Acceptable

Option 2: Eastbound and Westbound Interfaces

Another option is to make Service Orchestrators to integrate with each other's, using East and Westbound interfaces, in order to build an overall service. In this case, Service Orchestrators need to potentially integrate with all (or at least some) of the other Service Orchestrators, making things apparently more difficult and complex (more integrations required – partial/full mesh). On the other hand, the operator does not have any central Orchestration point where he can control the whole



system, but instead multiple Orchestrations, one per Service, which in some cases, may difficult obtaining a global orchestration view. Next Figure depicts this peer-to-peer model.



Figure 31: east-west orchestrator

Conclusion: Difficult

Option 3: Hybrid Option 1 and Option 2

There is still a compromise approach between the two options referred above. In this approach, a Top Orchestrator integrates all Service Orchestrators (interfaces Northbound and Southbound) in a central Orchestration point. This approach reduces the number of integrations required and provides to the operator an overall Orchestration view. Additionally, Eastbound and Westbound interfaces can be used in order to improve the efficiency of the system, in cases where the integration is preferable between Service Orchestrators. The number of interactions among Service Orchestrators and between Service Orchestrators and the Top Orchestrator will depend on the particular cases. The next Figure depicts this hybrid model.

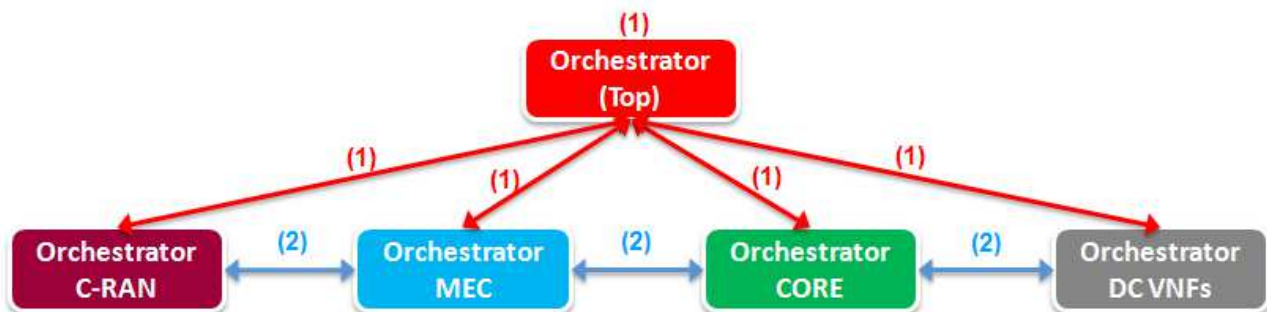


Figure 32: Hybrid orchestrator

Conclusion: Preferred



A3. Superfluidity Contributions

This section covers the contributions within the Superfluidity project that enables the above mentioned deployments and orchestration actions required to enable fluid 5G deployments.

A3.1 Kuryr

Considering that Superfluidity project targets quick provisioning at 5G deployments, there is a need to further advance in the container networking and its integration in OpenStack environment. To accomplish this, we worked on a recent project in OpenStack named Kuryr, which tries to leverage the abstraction and all the hard work previously done in Neutron, and its plugins and services, and use that to provide production grade networking for containers use cases. Hence with a two fold objective: a) make use of neutron functionality in containers deployments; and b) being able to connect both VMs and Containers in hybrid deployments.

In order to map Docker libnetwork to Neutron API, Kuryr is in charge of creating the appropriate objects in Neutron, so that every solution that implements Neutron API can be used for container networking. In this way, all the additional Neutron features can be applied directly to containers ports, such as security groups or floating IPs. To do this, the kuryr service works as an intermediary between the Docker network service (or Kubernetes) and the Neutron server, as shown in next figures, which also include the nested container use case.

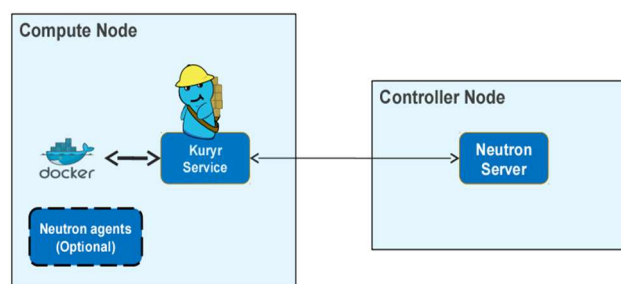


Figure 33: Kuryr Baremetal

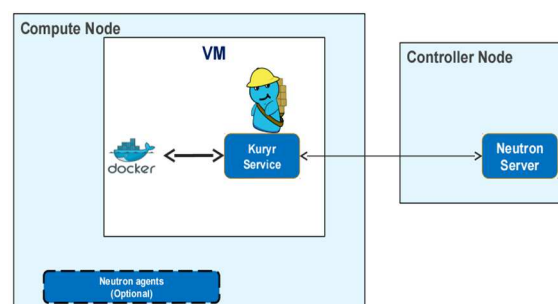


Figure 34: Kuryr Nested



Note that the potential of Kuryr does not need to stop at core API or basic extensions, but also to more advanced networking services such as enabling Load Balancing as a Service for Kubernetes services. What is more, it drives changes to Neutron community, such as the ‘tags’ addition to Neutron resources in order to allow API clients (like Kuryr) to store mapping data and port forwarding. Kuryr uses this to store the mapping between the Docker and Neutron networks. This information is used, among others, to know if a network must be removed from Neutron when it is removed from Docker (depending on who created the network or if it is being used).

Besides the interaction with the Neutron API, it is needed to provide binding actions for the containers so that they can be linked to the network. This is one of the common problems for Neutron solutions supporting containers networking as there is a lack of nova port binding infrastructure and no libvirt support. To address this, Kuryr provides a generic VIF binding mechanisms that takes the port types received from Docker namespace end and attach it to the networking solution infrastructure as highlighted in next figure.

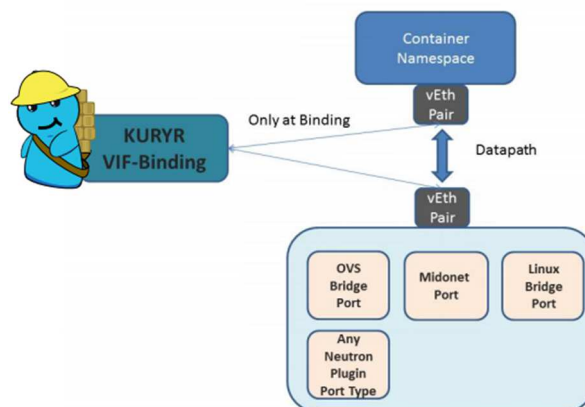


Figure 35: Kuryr VIF-Binding

Additionally, Kuryr provides a way to avoid double encapsulation as is the case in current nested deployments, for example when the containers are running inside VMs deployed on OpenStack. As we can see in next figure, when using docker inside the OpenStack VMs, there is a double encapsulation: one for the Neutron overlay network and another one on top of that for the containers network (e.g., flannel overlay). This creates an overhead that needs to be removed for the 5G scenario target by Superfluidity.

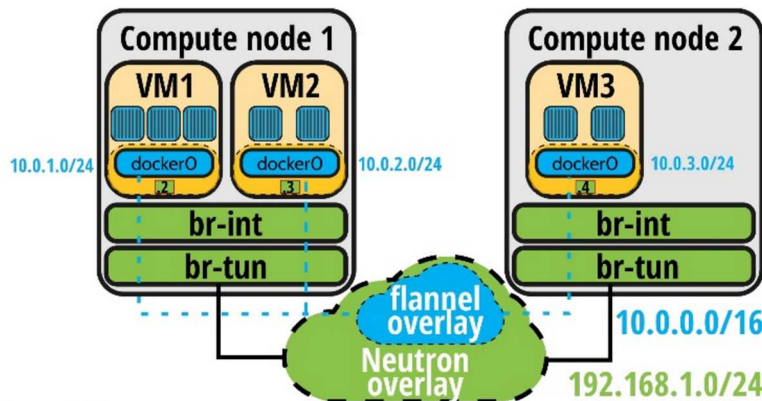


Figure 36: Double encapsulation problem

A3.1.1 Side by side OpenStack and OpenShift/Kubernetes deployment

To enable side by side deployments through Kuryr, a few components had to be added to handle the OpenShift (and similarly the Kubernetes) container creation and networking. An overview of the components is presented in the next image

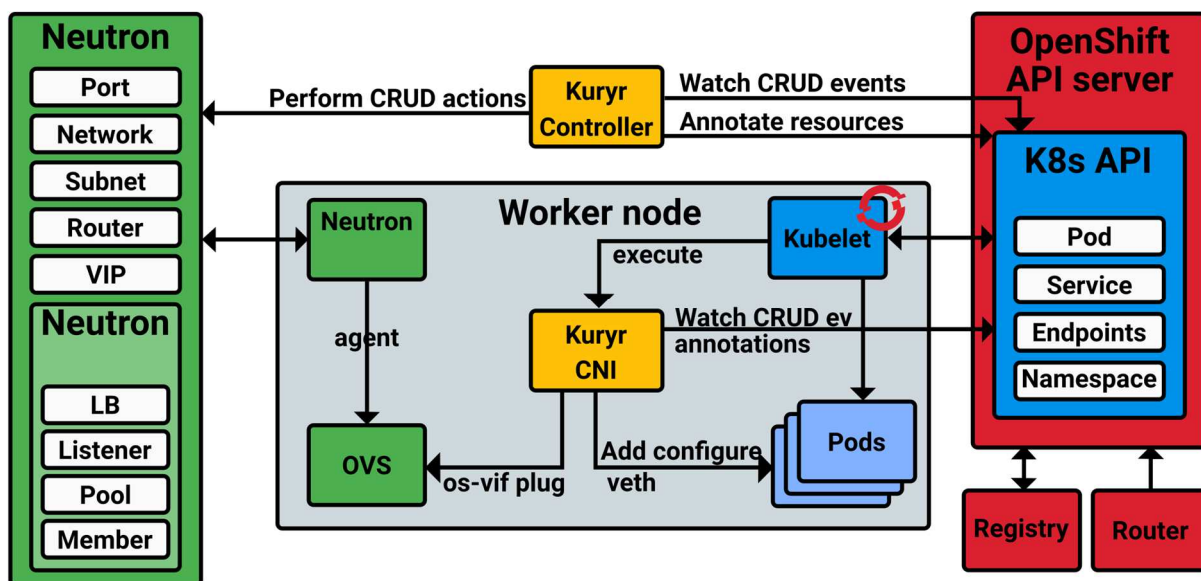


Figure 37: Kuryr components integration

The main Kuryr components are highlighted in yellow. The Kuryr-Controllers is a service in charge of the interactions with the OpenShift (and similarly Kubernetes) API server, as well as the Neutron one. By contrast, the Kuryr CNI is in charge of the networking binding for the containers and pods at each worker node, therefore, there will be one kuryr CNI instance in each one of them.

The interaction process between these components, i.e., the Kubernetes, OpenShift and Neutron components, is depicted in the following sequence diagram.

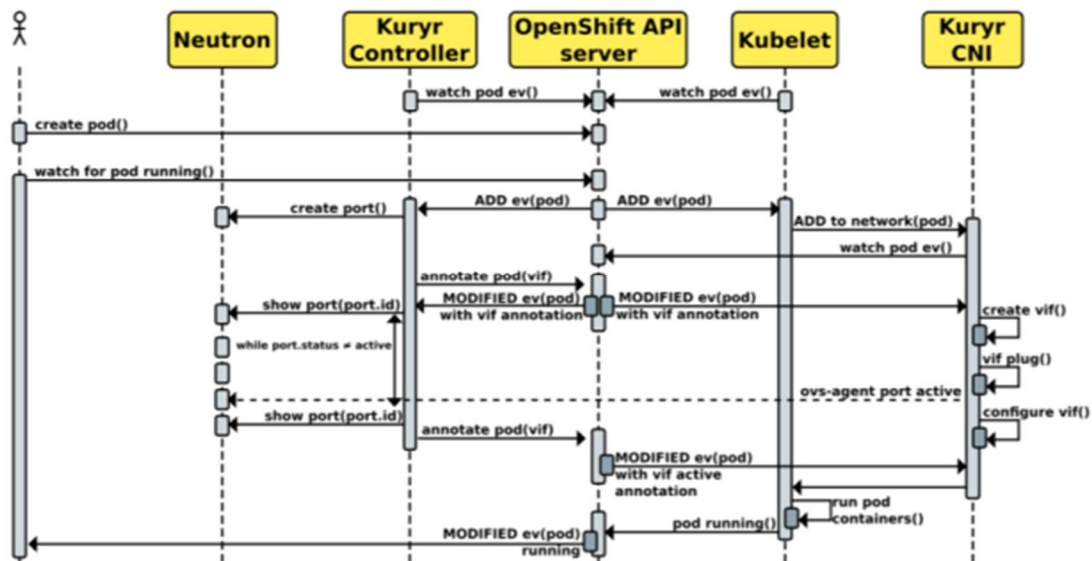


Figure 38: Sequence diagram: Pod creation

Similarly to Kubelet, the Kuryr-Controller is watching over the OpenShift API server (or Kubernetes API server). When a user request to create a pod reaches the API server, a notifications is sent to both, Kubelet and the Kuryr-Controller. The Kuryr-Controller then interacts with Neutron to create a Neutron port that will be used by the container later. It calls Neutron to create the port, and notifies the API server with the information about the created port (pod(vif)), while it is waiting for the Neutron server to notify it about the status of the port becoming active. Finally, when that happens, it notifies the API server about it. On the other hand, when Kubelet receives the notification about the pod creation request, it calls the Kuryr-CNI to handle the local bindings between the container and the network. The Kuryr-CNI waits for the notification with the information about the port and then starts the necessary steps to attach the container to the Neutron subnet. These consist of creating a veth device and attaching one of its ends to the OVS bridge (br-int) while leaving the other end for the pod. Once the notifications about the port being active arrives, the Kuryr-CNI finishes its task and the Kubelet component creates a container with the provided veth device end, and connects it to the Neutron network.

A3.1.2 Nested deployment: OpenShift/Kubernetes on top of OpenStack

There were still some gaps regarding nested containers that we are addressing at the Superfluidity project. Note these VMs are not managed directly by Nova and hence do not have an OpenStack agent inside them. This means we need a mechanism to perform the VIF binding inside the VM and



it needs to be initiated by the local Docker remote driver or Kuryr-Kubernetes CNI for the kubernetes case.

We have extended kuryr to leverage on the new **TrunkPort** functionality provided by Neutron (also known as VLAN-Aware-VMs) to be able to attach subports that are later bound to the containers inside the VMs, running a Kuryr Controller to interact with the Neutron server. This enables better isolation between the containers co-located in the same VM, even if they belong to the same subnet as the network traffic will belong to different (local) VLANs.

To make Kuryr working in nested environment, a few modifications and extensions were needed. These modifications have been contributed to the Kuryr upstream branch, both for Docker and Kubernetes/OpenShift support:

- (Docker) <https://review.openstack.org/#/c/402462/>
- (Kubernetes) <https://review.openstack.org/#/c/410578/>

The way the containers are connected to the outside Neutron subnets is by using a new feature included in Neutron, named Trunk Ports (<https://wiki.openstack.org/wiki/Neutron/TrunkPort>). The VM, where the containers are deployed, is booted with a Trunk Port, and then, for each container created inside the VM, a new subport is attached to that VM, therefore having a different encapsulation (VLAN) for different containers running inside the VM. They also differ from the own VM traffic, which leaves the VM untagged. Note that the subports do not have to be on the same subnet as the host VM. This thus allows containers both in the same and in different Neutron subnets to be created in the same VM.

To continue the previous example based on Kubernetes/OpenShift, a few changes were to be made to the two main components described above, Kuryr-Controller and Kuryr-CNI. As for the Kuryr-Controller, one of the main changes is regarding how the ports, which will be used by the containers, are created. Instead of just asking Neutron for a new port, there are two more steps to be performed once the port is created:

- Obtaining a VLAN ID to be used for encapsulating containers traffic inside the VM.
- Calling neutron to attach the created port to the VM's trunk port by using VLAN as a segmentation type, and the previously obtained VLAN ID. This way, the port will be attached as a subport to the VM, and can be later used by the container.

Furthermore, the modifications at the Kuryr-CNI (and kuryr-libnetwork for the docker case) are targeting the new way to bind the containers to the network, as in this case, instead of being added to the OVS (br-int) bridge, they are connected to the VM's vNIC in the specific vlan provided by the Kuryr-Controller (subport).



For the nested deployment with Kubernetes/OpenShift the interactions as well as the components are mainly the same. The main difference is how the components are distributed. Now, as the OpenShift/Kubernetes environment is installed inside VMs, the Kuryr-Controller also needs to run on a VM so that it is reachable from the OpenShift/Kubernetes nodes running in other VMs on the same Neutron network. With regards to the Kuryr-CNI, instead of being located on the servers, they need to be located inside the VMs actuating as worker nodes, so that they can plug the container to the vNIC on the VM on which they are running.

A3.1.3 Ports Pool Optimization

Every time a container is created or deleted, Kuryr makes a call to Neutron to create or remove the port used by the container. Interactions between Kuryr and Neutron may take more time than it is desired from the container management perspective, and specially from the speed needed by the target superfluidity scenarios.

Fortunately, some of these interactions can be optimized or even avoided. For instance, by maintaining a pre-created pool of Neutron resources instead of asking for their creation during pod lifecycle pipeline. As an example, every time a container is created or deleted, there is a call from Kuryr to Neutron to create/remove the port used by the container. To optimize this interaction and speed up both container creation and deletion, we propose a new Pool management driver at kuryr-kubernetes that takes care of both: Neutron ports creation beforehand, and Neutron ports deletion afterwards. This will consequently remove the waiting time for:

- Creating ports and waiting for them to become active when booting containers
- Deleting ports when removing containers

The main idea behind the proposed pool management driver resides on when and how the Neutron resources are managed, i.e., handling the Neutron resource creation, deletion and updates outside the container lifecycle pipeline -- when possible.

The proposed pool management driver handles different pools of Neutron ports:



- Available pools: There will be a pool of ports for each tenant, host (or trunk port for the nested case), and security group, ready to be used by the new pods being created. Note at the beginning there is no pools, and once a pod is created at a given host/VM by a tenant, with a specific security group, a corresponding pool gets created, and populated with the desired minimum amount of ports.
- Recyclable pool: Instead of deleting the port during pods removal, the port will be included into this pool. The ports in this pool will be later recycled by this driver and put them back into the corresponding available pool, after reapplying security groups to avoid any security breach.

The logic behind this pool driver ensures that at least X ports are ready to be used at each pool, i.e., for each security group and tenant. To provide this functionality, a new **VIF Pool driver** has been designed (one for the baremetal and one for the nested deployment types) that manages the ports pools upon pods creation and deletion events. It makes sure that at least a certain number of available ports exist in each pool (i.e., for each security group, host or trunk, and tenant) which already has a pod on it. The ports in each Available_pool are created in batches, i.e., instead of creating one port at a time, a configurable amount of them are created at once through Neutron bulk calls. The pool management driver checks for each pod creation that the remaining number of ports in the specific pool is above X. Otherwise it creates Y extra ports for that pool (with the specific tenant and security group). Note both X and Y are configurable and need to consider neutron quotas.

Having the ports ready at the Available_pools during the container creation process will speed up the process. Instead of calling Neutron port_create and then waiting for the activation of the port, it will be taken from the *available_pool* (hence, no need to call Neutron) and only the port info will be updated later with the proper container name (i.e., call Neutron port_update). Consequently, at least two calls to Neutron can be skipped (to create a port and wait for port to become ACTIVE), in favour of one extra step (the port name update), that is faster than the others. On the other hand, if the corresponding pool is empty, a *ResourceNotReady* exception is triggered and the pool is repopulated. After that, a port can be taken from that pool and used for another pod.

Similarly, the pool driver ensures that ports are regularly recycled after pods are deleted and put back in the corresponding *available_pool* pool to be reused. Therefore, Neutron calls are skipped as there is no need to delete and create another port for a future pod. The port cleanup actions return ports to the corresponding available_pool after re-applying security groups -- only if they have been changed during its attachment, otherwise there is no need to call Neutron at all during the deletion. A



maximum limit for the pool can be specified to ensure that once the corresponding available_pool reach a certain size, the ports above this number get deleted instead of recycled. This upper limit can be disabled by setting it to 0.

More information about the upstream design and implementation of these capabilities can be found at the next document:

- Blueprint: <https://blueprints.launchpad.net/kuryr-kubernetes/+spec/ports-pool>
- DevRef: <https://review.openstack.org/#/c/427681/>

More instructions about how to enable it and use it are available at:

<https://ltomasbo.wordpress.com/2017/05/09/kuryr-ports-pool-speeding-up-containers-booting-time-on-neutron-networks/>

A3.1.3.1 Ports Pool Performance Evaluation

Thanks to the above mentioned effort on pool management driver, the time needed to create containers, both in baremetal and in nested deployment when using kuryr is remarkably decreased as it can be seen in the figure below. In Figure 39, Upstream Baremetal and Upstream nested means the current available version of Kuryr for the generic and nested cases, respectively. On the other hand, the Pool Driver ones, represent the optimizations carried out as part of Superfluidity efforts.

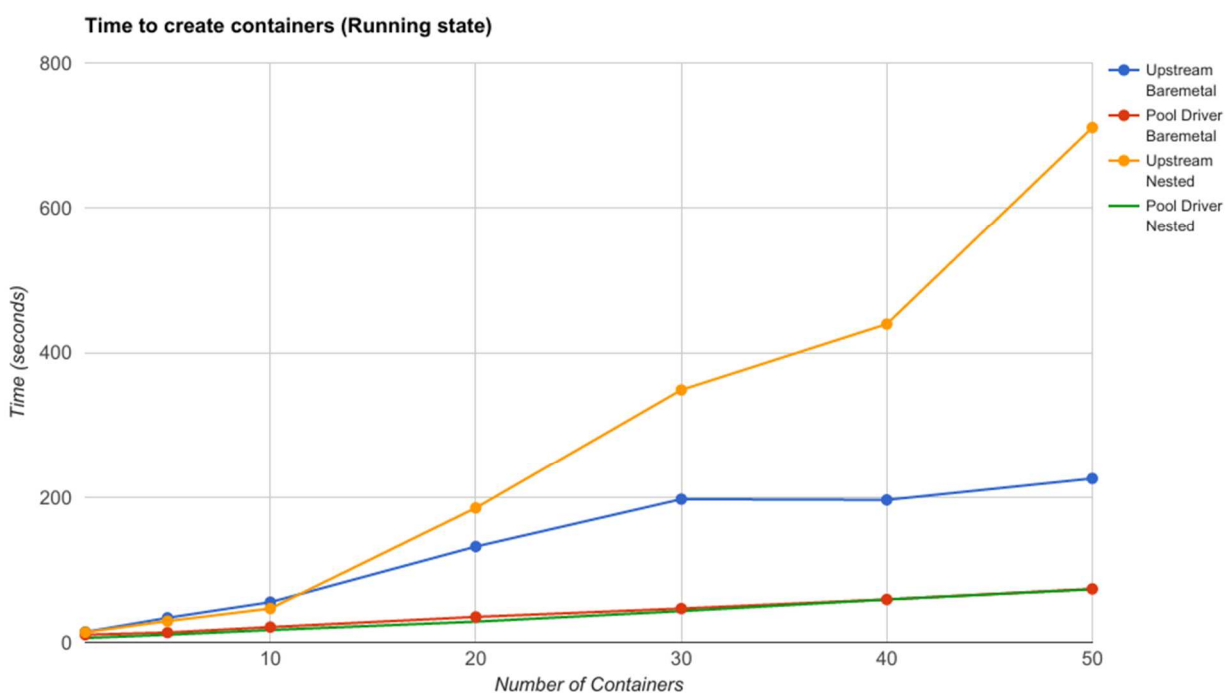


Figure 39: Time from pod creation to running status



As it can be seen, thanks to that, the creation times remain fairly constants, and slowly increase as the number of containers increased, but just do to the actual time to create the containers regardless of the network. Another important point to highlight is that there is no actual difference on performance of creating the containers in baremetal or inside VMs, from the booting time perspective.

In addition to that, thanks to reducing calls to Neutron, the load impose by container networking management on Neutron server is reduced too. Consequently, the ports pool feature helps to improve the scale at what Neutron can work, as well as speeds up other neutron actions unrelated to pod networking.

More details about scale testing of ports pool feature with different SDNs is presented at D7.3.

A3.1.4 Load Balancer as a Service (LBaaS) Integration

A Kubernetes Service (and consequently an OpenShift Service) is an abstraction which defines a logical set of Pods and a policy by which to access them. Whenever this set of Pods is updated, the Service gets updated too. Kubernetes service in its essence is a Load Balancer across Pods that fit the service selection. Kuryr's choice is to support Kubernetes services by using Neutron LBaaS service. The initial implementation is based on the OpenStack LBaaSv2 API, so compatible with any LBaaSv2 API provider. In order to be compatible with Kubernetes networking, Kuryr-Kubernetes makes sure that services Load Balancers have access to Pods Neutron ports.

More specifically, Kubernetes service is mapped to the LBaaSv2 Load Balancer with associated Listeners and Pools based on the protocol and specified exposed ports. Service endpoints are then mapped to Load Balancer Pool members.

As regards to the implementation details, two different kubernetes event handlers has been added to the kuryr controller pipeline to listen to the proper kubernetes events and trigger the related Neutron LBaaS operations:



- LBaaSSpecHandler manages Kubernetes Service creation and modification events. Based on the service spec and metadata details, it annotates the service endpoints entity with details to be used for translation to LBaaSV2 model, such as tenant-id, subnet-id, ip address and security groups.
- LoadBalancerHandler manages Kubernetes endpoints events. It manages LoadBalancer, LoadBalancerListener, LoadBalancerPool and LoadBalancerPool members to reflect and keep in sync with the K8s service. It keeps details of Neutron resources by annotating the Kubernetes Endpoints object.

Both Handlers use Project, Subnet and SecurityGroup service drivers to get details for service mapping. LBaaS Driver is added to manage service translation to the LBaaSV2-like API.

Next figures show the sequence diagram for the pod creation and the load balancer creation, respectively. In the first diagram the interactions between Kuryr, Kubernetes and Neutron components are detailed, for the nested case.

Similarly, in the second diagram, the interactions with the Neutron LBaaSV2 are presented. Note there is no interaction with the CNI part as there is no modifications to be made to how the containers are plugged into the network, just about how they are exposed at the load balancer level. Thus, they just need to be included as members at the pool associated to the corresponding load balancer.

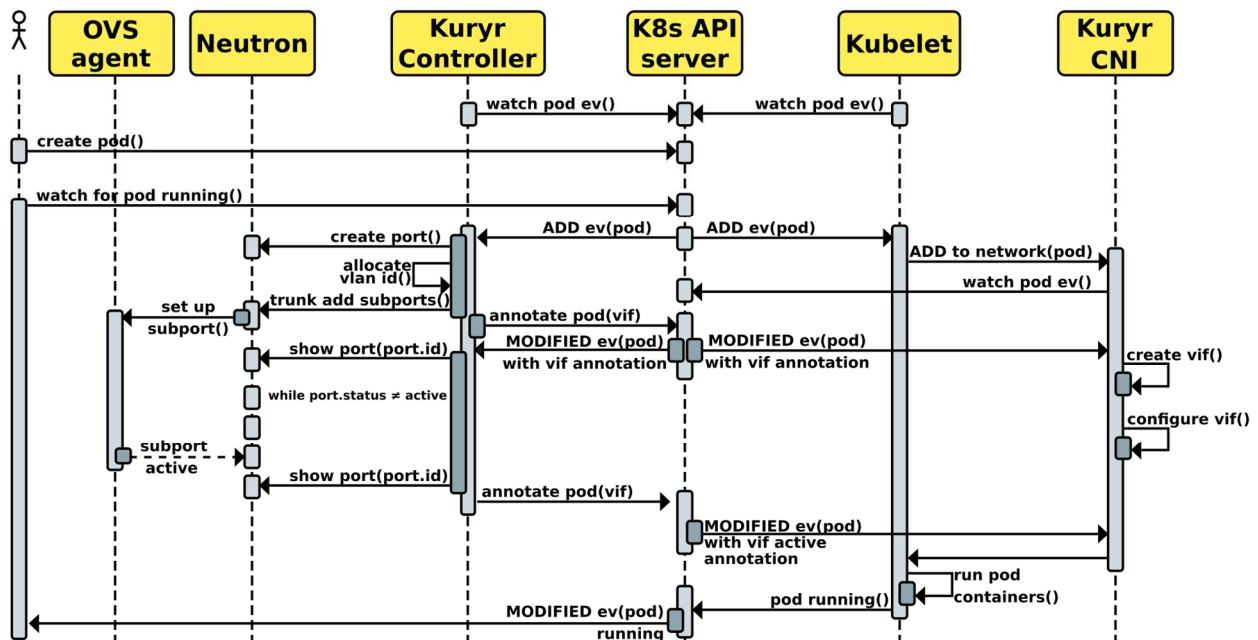


Figure 40: Sequence diagram: nested pod creation

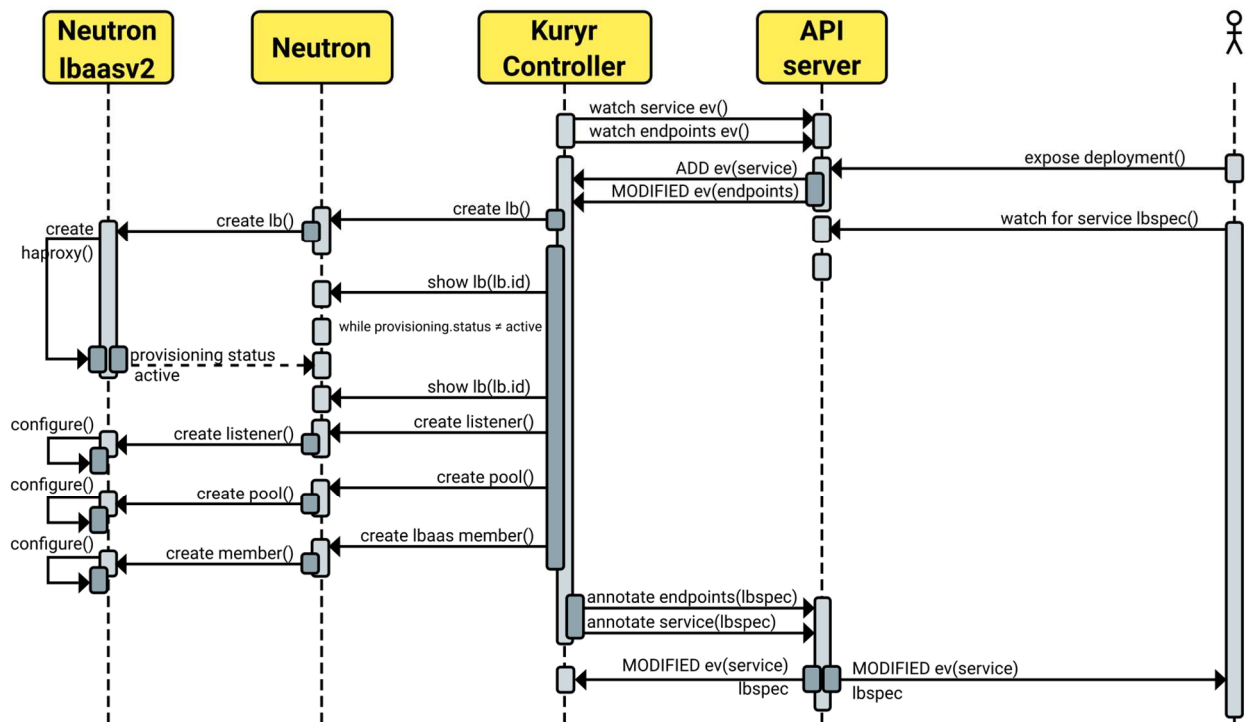


Figure 41: Sequence diagram: Service (LBaaS) creation

In addition to the integration between Kuryr and LBaaS V2, we have also work on extending the support to also integrate the new OpenStack load balancer project, named Octavia. A few modifications were needed to fully support Octavia load balancer modes, as for example it also supports L2 mode:

- <https://review.openstack.org/#/c/499103/>

Note just internals were modified as the process as well as the interaction are the same described above.

A3.1.5 Support for different Software Defined Networks (SDNs)

In addition to the previous kuryr extensions, we also focused on supporting different SDN backends so that we can leverage kuryr functionality on OpenStack clouds regardless of their SDN selection. The default OpenStack Neutron ML2 driver is OVS and that is supported (and tested) by Kuryr. However, in order to ensure other SDNs, we need to ensure:



- The Kuryr code works with them
- There is enough documentation to be able to use it
- There is enough testing to ensure the support is not lost at some point

We have focused on adding support for another 3 well-known/common SDNs, in this case OpenDaylight (ODL), Open Virtual Network (OVN), and DragonFlow (DF).

In order to cover the above mentioned points, we have tested and make the needed changes to support the three of them on Kuryr: OVN, ODL and DF. It must be highlighted that, thanks to the Neutron API abstraction, the support from the Kuryr side was mostly working out of the box, but thanks to the integration we helped discovering and fixing different problems on the SDNs side. As an example, we discovered and fixed problems related to the Trunks port support in kuryr. These functionality was tested with default ML2/OVS drivers, but both ODL, OVN, and DF recently added the functionality and had some gaps into their implementation, e.g., they were not setting the subports status to ACTIVE:

- ODL: bug (<https://bugs.launchpad.net/networking-odl/+bug/1707008>) and fix (<https://review.openstack.org/#/c/489517/>)
- OVN: bug (<https://bugs.launchpad.net/networking-ovn/+bug/1707141>) and fix (<https://review.openstack.org/#/c/488354/>)

In addition to ensure proper support for the different SDNs, and specially related to Superfluidity as different SDN solutions may be used at different points on the network, we worked on enhance documentation about how to use the different SDNs with Kuryr, as this will both help adoption as well as finding possible problems that may need fixing:

- <https://review.openstack.org/#/c/487885/>
- <https://review.openstack.org/#/c/497151/>
- <https://review.openstack.org/#/c/487906/>
- <https://review.openstack.org/#/c/543556/>

Finally, in order to ensure that the current support is not lost, we are also working on CI and the creation of new gates upstream that will ensure new Kuryr additions will not break the support for those SDNs.



CNI Daemon is an optional service that should run on every Kubernetes node. It is responsible for watching pod events on the node it's running on, answering calls from CNI Driver and attaching VIFs when they are ready. In the future it will also keep information about pooled ports in memory. This helps to limit the number of processes spawned when creating multiple Pods, as a single Watcher is enough for each node and CNI Driver will only wait on local network socket for response from the Daemon.

Communication

```

sequenceDiagram
    participant User
    participant Neutron
    participant Kuryr_Controller as Kuryr Controller
    participant K8s_API as K8s API server
    participant Kubelet
    participant Kuryr_CNI as Kuryr CNI
    participant Daemon_Watcher as Daemon (Watcher)
    participant Daemon_Server as Daemon (Server)

    User->>Neutron: create pod()
    Neutron->>K8s_API: watch for pod running()
    K8s_API->>Kuryr_Controller: watch pod ev()
    K8s_API->>Kubelet: watch pod ev()
    K8s_API->>Daemon_Watcher: watch pod ev()
    Kuryr_Controller->>Neutron: create port()
    Kuryr_Controller->>K8s_API: ADD ev(pod)
    Kuryr_Controller->>Kubelet: ADD ev(pod)
    Kuryr_Controller->>Kuryr_CNI: ADD to network(pod)
    Kuryr_Controller->>Daemon_Server: POST addNetwork
    Kuryr_Controller->>Neutron: show port(port.id)
    Kuryr_Controller->>K8s_API: annotate pod(vif)
    Kuryr_Controller->>K8s_API: MODIFIED ev(pod) with vif annotation
    Kuryr_Controller->>Kubelet: MODIFIED ev(pod) with vif annotation
    Kuryr_Controller->>Kuryr_CNI: MODIFIED ev(pod) with vif annotation
    Kuryr_Controller->>Daemon_Watcher: Get VIF (Manager)
    Kuryr_Controller->>Neutron: while port.status != active
    Kuryr_Controller->>Neutron: show port(port.id)
    Kuryr_Controller->>K8s_API: annotate pod(vif)
    Kuryr_Controller->>K8s_API: MODIFIED ev(pod) with vif active annotation
    Kuryr_Controller->>Kubelet: pod running()
    Kuryr_Controller->>K8s_API: MODIFIED ev(pod) running
    Kubelet->>Kuryr_CNI: run pod containers()
    Kuryr_CNI->>Daemon_Watcher: 201 Accepted
    Daemon_Watcher->>Daemon_Server: Get active VIF
    Daemon_Server->>Daemon_Server: wait vif()
    Daemon_Server->>Daemon_Server: create vif()
    Daemon_Server->>Daemon_Server: vif plug()
    Daemon_Server->>Daemon_Server: configure vif()
    Daemon_Server->>Daemon_Server: wait active
    
```

D6.1: System Orchestration and Management Design and Implementation



A3.1.7 Containerized Kuryr Components

Following what was already explained before. The main idea would be to follow Kubernetes Kubeadm linux installation guide and install pod networking like this:

```
kubectrl apply -f kuryr.yaml
```

This simplifies greatly the deployment of Kuryr-Kubernetes from the end-user side and also creates a easy way to perform upgrades. Instead of redeploying from scratch, a user or operator would only have to launch a new container.

Kuryr-Kubernetes gets split into:

- Kuryr CNI as daemon sets
- Kuryr Controller as a Pod
- Service accounts

The end integration in terms of networking is:

Kubelet (host space) -> kuryr-cni exec (host space) -> socket file -> kuryr-cni daemon (container space) -> netlink (container space) + k8s api

While this is just a simple task of further integration with kubernetes, it also had to be integrated with DevStack (Developer-oriented installation of OpenStack for the upstream testing needs). We have developed a way to generate both images and resource definitions for Kuryr-Kubernetes on devstack:

It includes a tool that lets you generate resource definitions that can be used to Deploy Kuryr on Kubernetes. The script is placed in tools/generate_k8s_resource_definitions.sh and takes up to 3 arguments:

```
$ ./tools/generate_k8s_resource_definitions <output_dir> [<controller_conf_path>]  
[<cni_conf_path>]
```



- `output_dir` - directory where to put yaml files with definitions.
- `controller_conf_path` - path to custom kuryr-controller configuration file.
- `cni_conf_path` - path to custom kuryr-cni configuration file (defaults to `controller_conf_path`).

This should generate 4 files in your `<output_dir>`:

- `config_map.yml`
- `service_account.yml`
- `controller_deployment.yml`
- `cni_ds.yml`

Deploying Kuryr resources on Kubernetes

To deploy the files on your Kubernetes cluster run:

```
$kubectl apply -f config_map.yml -n kube-system
$kubectl apply -f service_account.yml -n kube-system
$kubectl apply -f controller_deployment.yml -n kube-system
$kubectl apply -f cni_ds.yml -n kube-system
```

After successful completion:

- kuryr-controller Deployment object, with single replica count, will get created in kube-system namespace.
- kuryr-cni gets installed as a daemonset object on all the nodes in kube-system namespace

A3.1.8 RDO Package and CI

RDO is a community of people using and deploying OpenStack on CentOS, Fedora, and Red Hat Enterprise Linux. You can say that is the community version of OSP (Red Hat OpenStack Platform). In order to make the consumption of Kuryr easier for end-users, we've gone through the steps of packaging both Kuryr and its tests (Kuryr Tempest Plugin).

RDO produces two set of packages repositories:



- **RDO CloudSIG** repositories provide packages of upstream point releases created through a controlled process using CentOS Community Build System. This is kind of "stable RDO".
- **RDO Trunk** repositories provide packages of latest upstream code without any additional patches. New packages are created on each commit merged on upstream OpenStack projects.

Following diagram shows the global packaging process in RDO.

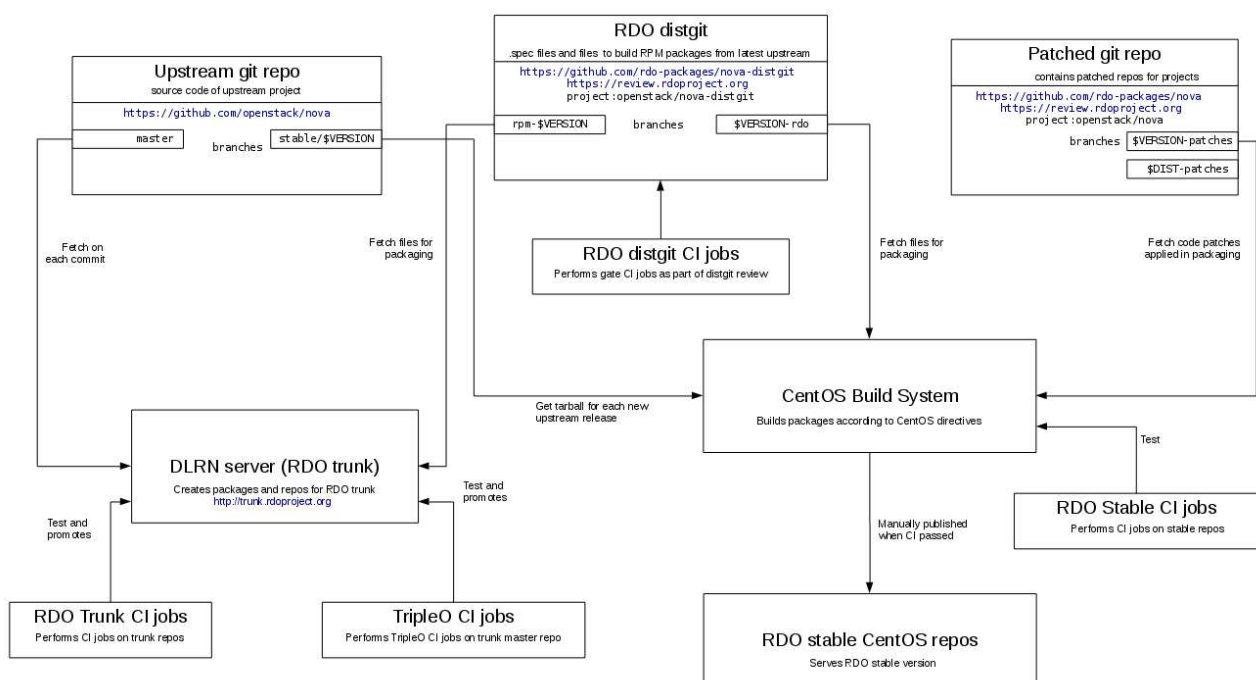


Figure 43: RDO packaging process

While this section does not intend to provide a comprehensive walk-through over the packaging process, we'd like to highlight two main different sections.

distgit - where the .spec file lives

distgit is a git repository which contains `.spec` file used for building a RPM package. It also contains other files needed for building source RPM such as patches to apply, init scripts etc.

RDO packages' distgit repos are hosted on review.rdoproject.org and follow \$PROJECT-distgit naming.

DLRN



DLRN is a tool used to build RPM packages on each commit merged in a set of configurable git repositories. DLRN uses `rdoinfo` to retrieve the metadata and repositories associated with each project in RDO (code and distgit) and `mock` to carry out the actual build in an isolated environment. DLRN is used to build the packages in RDO Trunk repositories that are available from <http://trunk.rdoproject.org>.

NVR for packages generated by DLRN follows some rules:

- Version is set to MAJOR.MINOR.PATCH of the next upstream version.
- Release is 0.<timestamp>.<short commit hash>

For example `openstack-neutron-8.1.1-0.20160531171125.ddfe09c.el7.centos.noarch.rpm`.

With the usage of this tools, we make sure that there are always updated packages available for consumption coming from Kuryr side.

Upstream CI

From the upstream side of things, we do not use packages but source code. The workflow used follows this diagram:

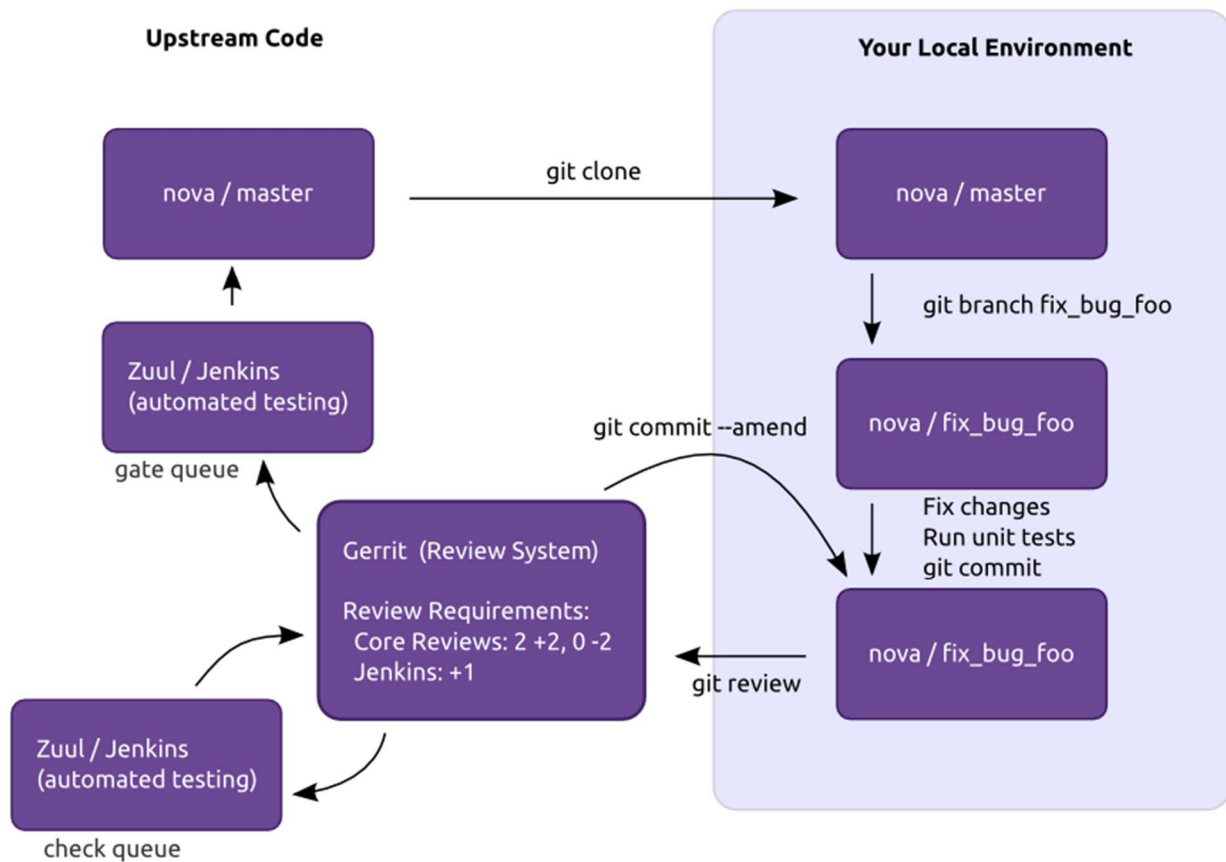


Figure 44: Upstream CI workflow

Once that you have cloned the repo and worked on your patch, it goes through an extensive testing system called Zuul.

Zuul is a program that drives continuous integration, delivery, and deployment systems with a focus on project gating and interrelated projects.

Zuul's configuration is organized around the concept of a *pipeline*. In Zuul, a pipeline encompasses a workflow process which can be applied to one or more projects. For instance, a "check" pipeline might describe the actions which should cause newly proposed changes to projects to be tested. A "gate" pipeline might implement the automation to merge changes to projects only if their tests pass. A "post" pipeline might update published documentation for a project when changes land.

While Zuul focuses on the infrastructure, the tests that are being run into OpenStack's pipelines for Kuryr are based into two different projects.



- Tempest: Is the functional testing framework for OpenStack.
- Kuryr-Tempest-Plugin <https://github.com/openstack/kuryr-tempest-plugin>:
 - We have developed a new upstream depository dedicated to the Integration of kuryr testing into upstream OpenStack's Tempest framework. For that, we have been focusing on pod to vm networking tests, which cover the functionality needed for all the Superfluidity's scenes.

A3.2 Mistral Orchestration

There is no doubt that workflows are a key ingredient for cloud automation, and automation is a key element in orchestration. However, when looking at an end-to-end, Telco grade, full NFV deployment over multiple distributed sites, orchestration starts becoming trickier.

Using TOSCA as the main DSL for the Network Service Descriptor provides a hierarchical view of services, components and their relationships, which is decoupled from the underlying VIM/NFVI. When combining TOSCA interfaces and a Mistral workflow, a full network service lifecycle can be achieved.

Mistral is an OpenStack workflow service. Most life cycle management (LCM) processes consist of multiple distinct interconnected steps that need to be executed in a particular order in a distributed environment. One can describe such LCM process as a set of tasks and task relations and upload such description to Mistral so that it takes care of state management, correct execution order, parallelism, synchronization and high availability. Mistral also provides flexible task scheduling so that we can run a process according to a specified schedule (i.e. every Sunday at 4.00pm) instead of running it immediately.

Although Mistral is quite generic it is built to become a natural part of OpenStack ecosystem. Out of the box Mistral provides "openstack" action pack for using functionality provided by other OpenStack services like Nova, Neutron or Heat. Mistral workflows can also be run from Murano PL.

To fulfil the vision of Superfluidity, and develop a telco grade orchestration based on Mistral, we identified several gaps that we addressed and submitted to the OpenStack Newton and Ocata releases. Specifically, (i) we addressed the performance and stability of Mistral, making it ~100 times faster, (ii) we extended the scope of Mistral to support multi VIM as envisioned in Superfluidity architecture, (iii) improved its reporting and event notification engine, and (iv) improved its usability. In more details, to improve Mistral performance and reduce the LCM operation time, we shortened the database transactions, optimized the databased queries, and applied caching techniques.

Mistral, originally, was configured to execute workflows on a specific cloud VIM. In order to execute workflows on a different VIM, one needed to start a new Mistral instance. To allow a more fluent support for the distributed architecture of Superfluidity, we extended Mistral's workflow execution



parameters to make it possible to target a specific cloud without modifying the configuration of the Mistral service.

Additionally, we improved the reporting and tracking mechanisms in Mistral, e.g., adding an endpoint to track action/workflow executions belong to a certain task.

Finally, to improve usability we added a Mistral dashboard in OpenStack Horizon, as well as developed UI for better experience with writing custom actions.

A3.3 OSM

After an evaluation process of different MANO tools (see section 3.2), the OSM was selected to implement the MEC Orchestrator (MEO) component. Although OSM is built for NFV scenarios (to materialize the NFVO and VNFM components), we performed a work to adapt it to the MEC components.

There are a few differences among NFV and MEC. In the NFV world there are mainly two layers of entities: VNFs and NSs. In the MEC world there is only a single entity: the MEC Applications (MEC Apps). For this reason, we decided to implement MEC Apps as VNFs, creating additionally a NS with a single VNF inside. The reason for doing that is because the OSM does not allow the deployment of VNFs, but only NSs. So this is a simple workaround we found to overtake this difference. Considering this, in order to onboard an MEC Apps, we basically need to create 2 separated descriptors: a VNFD and a NSD.

Using OSM, the on-boarding process is dealt by the Riftware (NSO) component. The MEC App (as NS) is stored at the NS Catalogue as depicted in the Figure below.

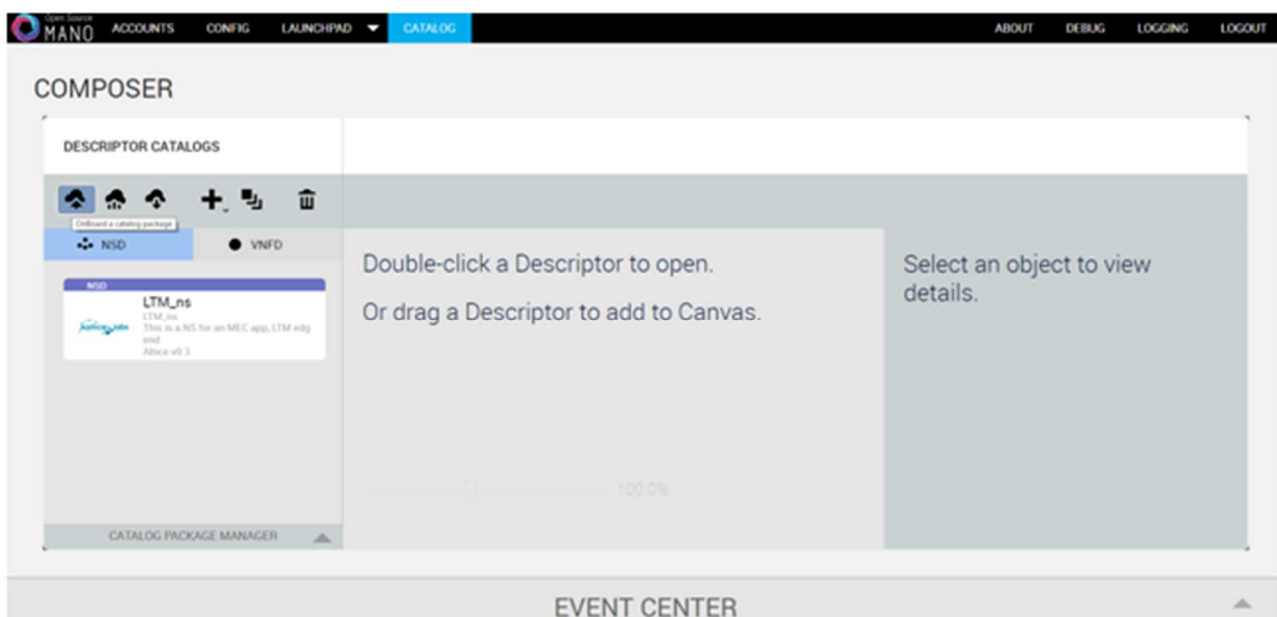




Figure 45: Onboarding of MEC Apps on OSM.

The VNF Catalogue stores the embedded MEC App VNF. Although the MEC App Descriptors and the NFV Descriptors contain different directives, they are very similar, allowing us to reuse the NFV ones. However, in the future we will need to extend it with MEC specific directives, in order to accommodate the requirements defined by MEC. The on-boarding process is performed through a file with the format described below.

The on-boarding process is very simple. It basically requires the selection of the type of descriptor to on-board (VNFD/NSD), and upload a tar.gz file containing the descriptor and other related data. The CSAR (Cloud Service Archive) is a format defined by TOSCA/NFV to upload all the material related to a VNF and NS, e.g. descriptors, lifecycle scripts, etc.

The OSM OpenMano (RO) component is responsible to interact with the VIM and can be mapped directly to MEC environments. As OSM is multi-VIM, it allows us to select the VIM where the MEC App is deployed. In the instantiate process the user can choose one of the available VIMs, previously configured. In MEC environments, assuming a VIM per edge, this permits a single orchestration tool to manage the multiple edges.

The OSM Juju component is used as configuration tool, in order to run all the scripting required for the life cycle management. Juju uses the charm concept to store and run sets of scripts associated to the configurations required for each of the lifecycle stage. Scripts can be written in any language. Juju allows the use of a large number of public charms available (more than 300) for some services. It also allows the user to create new charms, which can also use internally other charms. For example, in our case, the charm we created is composed by the charm 'basic', 'sshproxy' (ssh capabilities) and 'vnfproxy' (common VNF functionality).

The MEC Apps lifecycle management can be performed using the so-called charms hooks associated with events. There are several predefined events that each application goes through: install, configure, start, upgrade and stop. To perform configurations, when those events occur, Juju triggers a charm that can respond by pointing to executable files (hooks); then, Juju executes the specific hook for the appropriate event. For example, in our case, we use the events 'start' and 'stop' to interact with our MEC App. In the start event we provision the MEC App and the TOF piece with traffic offloading rules; and in the stop event we remove the TOF rules.

Charm can receive inputs from the VNFD, if they are configured in the VNFD and in Juju. For example, we use this mechanism to pass to Juju credentials for ssh and some ip/port data.

Juju charms need to be compiled before they are uploaded to Juju. To create a charm, you just need to install the Juju package, and execute one command to create the basic charm layer. Charms are built using the following directory tree.



```
├── config.yaml
├── icon.svg
├── layer.yaml
├── metadata.yaml
├── reactive
│   └── ltm.py
├── README.ex
├── tests
├── 00-setup
└── 10-deploy
```

The *layer.yaml* file specifies the charms available to use; the *metadata.yaml* file describes what the charm does; the *config.yaml* file specifies the inputs received from VNFD. Inside the *reactive* folder are located all the scripts that need to be invoked from the main file (*ltm.py* in the example above), in the correspondent hook event, or putting the code directly within the file (in case of python language). If you need other hooks than the defaults explained before, just create a folder *actions* with the file who get the script from reactive folder, and create an *action* file to specify the action and the parameters required.

The charm is then built and included in the CSAR file. The OSM has a particular project called 'descriptor-packages' to help on that. Using a simple command, it produces the CSAR file (tar.gz) with the following structure. This file is used to on-board the MEC App as described above.

```
ltm_vnf
├── charms
│   └── ltm
├── checksums.txt
├── icons
├── images
├── ltm_vnfd.yaml
├── README
└── scripts
```

A3.4 ManagelQ

ManagelQ is a management project that enables managing containers, virtual machines, networks and storage from a single platform, connecting and managing different existing clouds: OpenStack, Amazon EC2, Azure, Google Compute Engine, VMware, Kubernetes, OpenShift, ...



The main reason for choosing ManageIQ as an NFVO is due to being able to work with both VMs and Container providers. However, after feature analysis, we discover a few gaps that needed to be address for the Superfluidity purposes

A3.4.1 Ansible execution support

The main concern about the existing NFVO tools was the lack of applications life-cycle management actions for container. Therefore we worked on an extension to fix this gap on ManageIQ. The proposed extension is based on supporting ansible playbook execution within ManageIQ. Therefore, any action (as ansible is agent-less) can be triggered in any of the providers.

One of the more powerful capabilities of ManageIQ is the self-service. It allows an administrator to maintain a catalog of requests that can be ordered by regular users, for example, to provision a single VM, a container or an application stack. It starts with an administrator creating a "service bundle," which is a collection of "service items". Each service item is an action that ManageIQ knows how to create/handle. The order in which items in a bundle are provisioned is specified by the administrator, in what is known as the state machine. Services typically require some amount of input. For example, if the request is to provision a VM, then a typical question would be the memory and disk size. This information can be requested from the user through a dialog, which can be created using ManageIQ's built-in dialog editor.

Once the service bundle and the dialog are created, they need to be associated with an "entry point" in the ManageIQ workflow engine (called "Automate"). The entry point defines the process to provision the bundle. With the bundle definition, dialog, and entry point, the request can be published in a service catalog, which then enables users to order the service.

Given the above, Self-service is good for both the administrator and the end user, and it is the capability we have used to add support to run Ansible playbook at Containers deployments, in our case Kubernetes and/or OpenShift -- note it also enables running them at both baremetal or OpenStack deployments. By enabling the execution of playbooks located at git repositories, we provide an easy way to onboard new lifecycle management actions. It is really easy to update, include, or extend different playbooks that take care of different VMs/Containers/Apps lifecycle actions, such as creation, termination, scaling or any other configuration actions. It is as simple as using the common *git commit* and *git push*.

In our ManageIQ catalog item, we have defined a dialog that takes, on the one hand, the ansible playbook to execute (usually a site.yml file), and on the other hand the provider where you want to execute it -- being the only requirement to have ansible installed at the ManageIQ appliance and ssh-passwordless towards the providers master nodes.

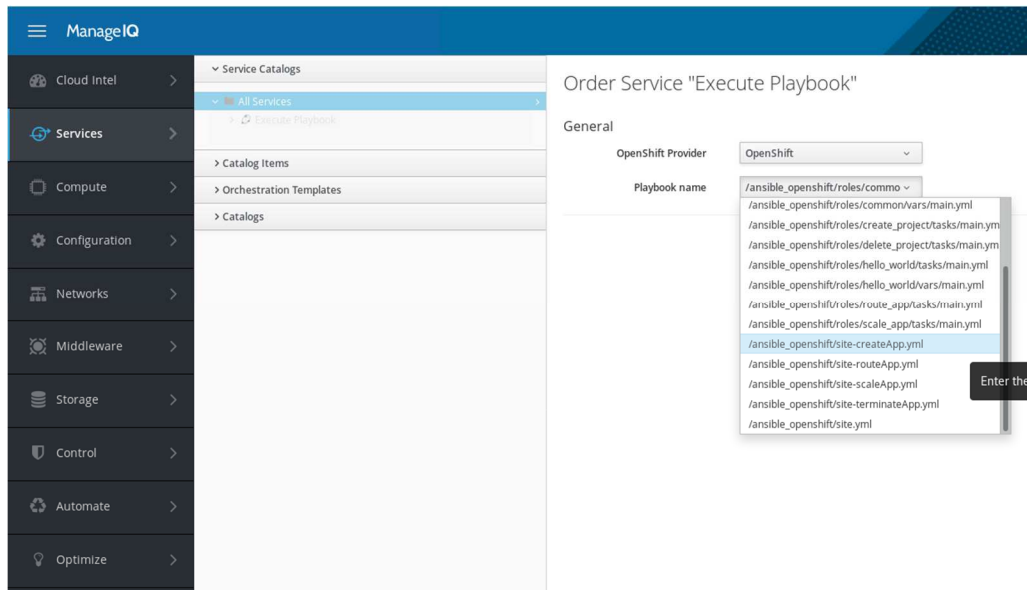


Figure 46: Ansible playbook execution

In the figure above, we can see there are different playbooks to create, delete, scale and scale the Application for a given github repository. Note there may be several of them, as many as cloned in the specified directory in the ManageIQ appliance.

As regards to the service bundle, it contains a series of methods that are organized in a state machine that performs an initial action of reading, checking and processing the input from the user dialogs (named `parse_dialog`). Then, there is a second step where playbook execution is actually triggered (`run_job`).

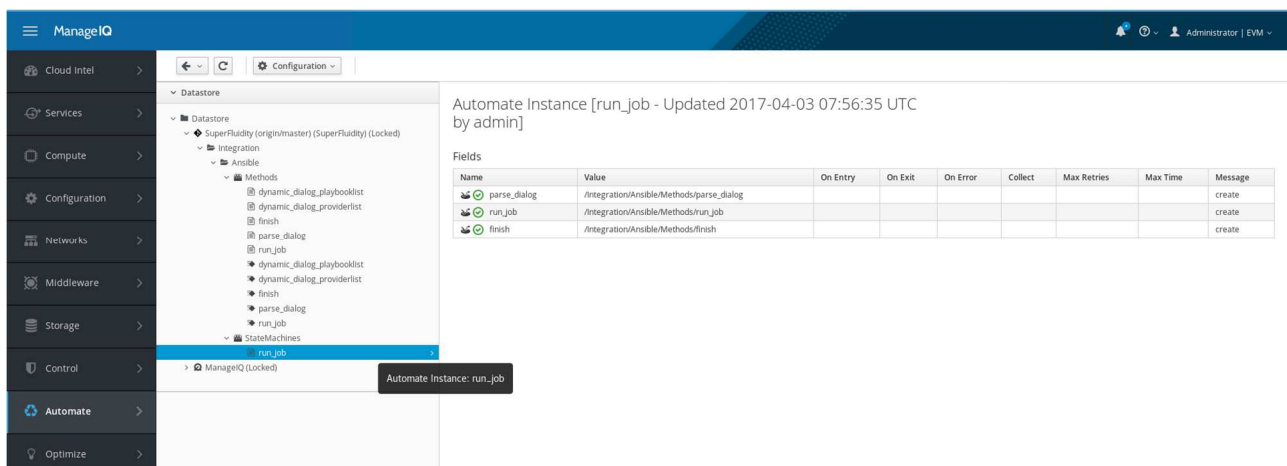


Figure 47: ManagelQ State Machine



A3.4.2 Multi-site support

Building upon the previous point, we added extra support to execute those playbooks across different sites as there was a need for synchronization about tasks being performed at different sites. As an example, when deploying the CRAN, MEC, EPC components, they can share some information or even must know about each other. Thus, it may be convenient to have then in the same playbook and control when the steps take place, e.g., deploying the EPC before the CRAN (or the other way around)

Consequently, we have worked at adding this support for multi-site deployment from ManageIQ. Up to now, with ManageIQ you can manage different providers (i.e., different sites), but not execute a set of actions that involves several of them at the same time. We have added to the ansible support at ManageIQ the option to include a host file where you can specify the different sites where the playbook need to be executed. This way, you can directly specify on the playbooks where each task is meant to be executed: e.g., the EPC related tasks in the EPC cloud (which for instance can be a VM running on an OpenStack deployment), and the CRAN tasks at the CRAN server (which can be a kubernetes cluster or even just a linux server).

Therefore, to make use of this new functionality the only need from the user point of view is to include a host file defining all the hosts (be it OpenStack endpoints, kubernetes master APIs, or single servers) into their git repository and annotate the tasks with the host where it need to be executed. Then, at the ManageIQ UI, the multisite deployment flag needs to be enabled, and the proper host file selected - beside the desired playbook to be executed.

Ansible Run Playbook

The screenshot shows the 'Ansible Run Playbook' interface with the 'General' tab selected. The configuration options are as follows:

General	
MultiSite deployment ⓘ	<input checked="" type="checkbox"/>
OpenShift Provider ⓘ	Select an OpenShift Provider from the list ▼
Inventory ⓘ	/superfluidity-playbooks/cran-ansible-hosts.yml ▼
Playbook name ⓘ	/rdcl-repo/testbed/project_1/nsd_cran/site_deploy.yaml ▼

Figure 48: Muti-Site playbook execution

A3.5 Load Balancing as a Service

As identified in previous deliverables (Deliverable D2.1), many of the use cases of interest depend on common load balancing capabilities, to fulfil the scalability and high-availability requirements. The requirements of Load Balancer, as a functional block, are analyzed further in Deliverable D2.2.



OpenStack offers a few open source options for implementing the infrastructure load balancing capability, most notably HAProxy (upstream open source project) and Octavia (OpenStack project). Deliverable D5.3 provides more information on these options.

Something we wanted to evaluate, as part of Superfluidity, is how they compare with commercial, carrier-grade, options. Citrix NetScaler ADC [20] is such an option. The figure below illustrates how NetScaler ADC fits into the NFV architecture and how it integrates with the MANO elements, namely the VIM (OpenStack) and the SDN Controller.

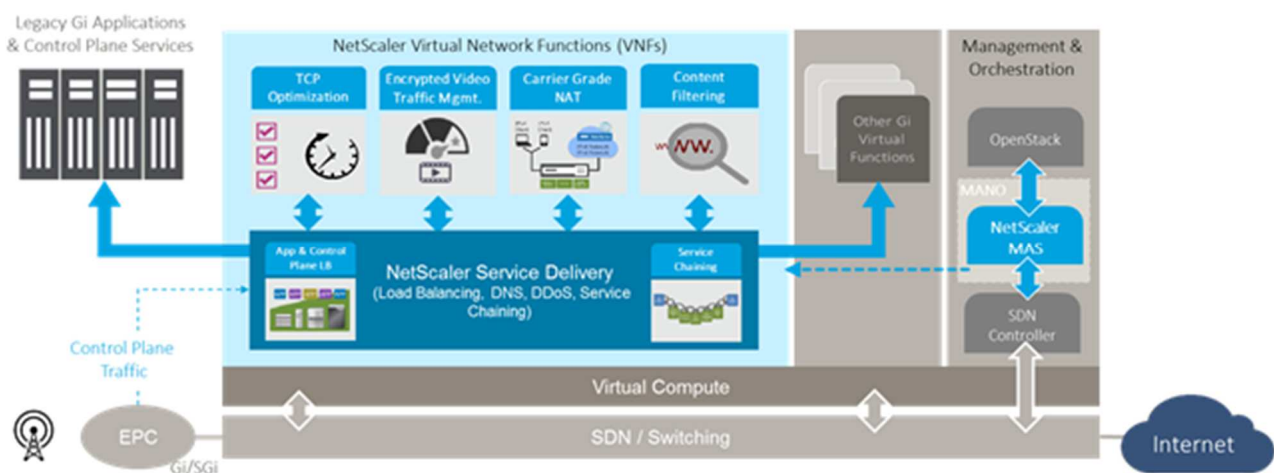


Figure 49: NetScaler ADC at NFV architecture

NetScaler MAS [21] acts as an Element Manager for NetScaler ADC. It integrates using standard APIs with OpenStack and supported SDN Controllers, translating them to the RESTful APIs supported by NetScaler ADC [22]. More information on the integration between NetScaler MAS and OpenStack can be found at [23]

The evaluation we have performed as part of Superfluidity consists of two parts:

- Comparing the capabilities of open source options (initially HAProxy) versus NetScaler ADC
- Validating the integration of NetScaler ADC, NetScaler MAS and OpenStack in the Reference NFV Lab we have deployed to support the Task 5.3 activities.

From a load balancing capabilities standpoint, the key difference we identified is the availability of a far broader range of:

- Load Balancing Algorithms: [24]
- Persistence Types: [25]



From an experimentation standpoint, we have successfully deployed and validated the above integration against the versions of OpenStack that were released during the lifetime of the Superfluidity project so far, namely Liberty, Mitaka, Newton, Ocata and Pike.

As part of this activity, we have leveraged enhancements in both the NetScaler LBaaS driver (OpenStack Neutron/LBaaS project: [26], [27]), as well as NetScaler MAS and ADC (commercial software). The VPX (virtual) edition of NetScaler ADC was used for that purpose, specifically the 12.0 release, and the respective NetScaler MAS release. The above outcomes were integrated with the Superfluidity platform as part of WP7 activities.

Finally, for perfect alignment with the support for the execution of Ansible playbooks by the NFVO (ManagelQ, see A3.4), we have implemented two extensive Ansible modules:

- `netScaler_server`: automates and manages the configuration of NetScaler servers (http://docs.ansible.com/ansible/latest/netScaler_server_module.html)
- `netScaler_service`: automates creation and manages the configuration of NetScaler services (http://docs.ansible.com/ansible/latest/netScaler_service_module.html)

The implementation of the above Ansible modules for NetScaler was contributed as open source (<https://github.com/citrix/netScaler-ansible-modules>) and relevant developer documentation was published (<https://developer-docs.citrix.com/projects/netScaler-ansible-modules/en/latest/>).

A3.6 Service Function Chaining

As identified in the requirements analysis (Deliverable D2.1), many complex use cases consist of compositions of in-network services, which require Service Function Chaining (SFC) to materialize. The components of the IETF SFC architecture, namely the Service Functions (SF), SF Forwarder, Network Overlay, SFC Proxy and SFC Classifier are analysed further in Deliverable D2.2.

Based on our monitoring, the open source projects above have been making more progress since:

- OpenStack SFC support (<https://docs.openstack.org/ocata/networking-guide/config-sfc.html>) is maturing, adding new capabilities (<https://docs.openstack.org/networking-sfc/latest/>): New SFC Driver for OVN (https://docs.openstack.org/networking-sfc/latest/contributor/sfc_ovn_driver.html, <http://openvswitch.org/support/dist-docs/ovn-architecture.7.html>), support for Symmetric Port Chains, Service Function Tap for Port Chains, etc.
- The OPNFV OVS project introduced “NSH for VXLAN” support in the OPNFV Colorado release. However, this required a special release of OVS (which included Yi Yang’s OVS NSH patches: https://github.com/yyang13/ovs_nsh_patches). Actually, it seems this is still the case in the OPNFV latest (Euphrates) release.



- The OPNFV Colorado, Danube and Euphrates versions support the SFC [os-odl_l2-sfc-noha](http://docs.opnfv.org/en/stable-os-odl_l2-sfc-noha), [os-odl_l2-sfc-ha](http://docs.opnfv.org/en/stable-os-odl_l2-sfc-ha) scenarios, i.e. OpenStack + OVS + OpenDaylight + SFC, at least for the Apex and Fuel installers. The version of OpenDaylight OPNFV Euphrates is dependent on is Nitrogen SR1 (SR2 is the latest). The latest documentation of the OPNFV SFC submodule: <http://docs.opnfv.org/en/stable-euphrates/submodules/sfc/docs/development/design/index.html>
- As foreseen in section 3.1.1, the VNFM has to be involved as well. The scenarios implemented by OPNFV SFC utilize OpenStack Tacker for that purpose (see <http://docs.opnfv.org/en/stable-euphrates/submodules/sfc/docs/development/design/architecture.html#vnf-manager>). More information on how this is implemented: <https://specs.openstack.org/openstack/tacker-specs/specs/newton/tacker-networking-sfc.html>
- Lastly, when using NSH with VXLAN tunnels, it is important that the VXLAN tunnel is terminated in the SF VM. This allows the SF to see the NSH header, allowing it to decrement the NSI and also to use NSH metadata. When using VXLAN with OpenStack, the tunnels are not terminated in the SF VM, but in the OVS bridge. In the OPNFV Danube release, this required a workaround (<http://docs.opnfv.org/en/stable-danube/submodules/sfc/docs/development/design/architecture.html#ovs-nsh-patch-workaround>), but this is no longer the case in the Euphrates release (<http://docs.opnfv.org/en/stable-euphrates/submodules/sfc/docs/development/design/architecture.html>).

As clear from the above, the current situation with SFC is unnecessarily complex, due to unfulfilled upstream project dependencies that require special patches and workarounds. In anticipation of improvements, we invested in implementing NSH in NetScaler. This is in NetScaler 11.1 build 47.14 (June 2016), or later releases: https://docs.citrix.com/content/dam/docs/en-us/netScaler/11-1/release-notes/NS_11_1_53_11.html (#593459).

As a result, a NetScaler ADC (virtual) appliance can now play the role of the Service Function in the SFC architecture. The NetScaler instance receives packets with Network Service headers and, upon performing the service, modifies the NSH bits in the response packet to indicate that the service has been performed. In that role, the appliance supports symmetric service chaining with specific features, for example, INAT, TCP and UDP load balancing services, and routing. Restrictions of the initial release: Only VXLAN-GPE is supported as the tunneling protocol. IPv6 is not yet supported.



A3.6.1 Service Function Chaining with IPv6 Segment Routing (SRv6)

The SFC based on NSH still suffers from limited maturity of the implementations, as outlined in the previous section. Hopefully, this will improve in the future. Anyway, there is also an intrinsic shortcoming of the traditional SFC/NSH approach, because it typically requires stateful operations in the SFC proxies and SFC classifiers along the path of the Service Chains. Handling thousands of Service Chains with this stateful approach could lead to scalability concerns. In addition to these potential scalability issues, the need of setting up the required state information in the intermediate nodes is not optimal considering the requirement to setup and configure Service Chains in very short time.

For the above reasons, we have also considered a “disruptive” solution based on IPv6 Segment Routing (SRv6). The SRv6 technology works by carrying a *segment list* (a list of IPv6 addresses) in the IPv6 header. SRv6 can be used as underlay or overlay technology and it extends the capability of the networking layer, allowing to put “network programs” in the packet headers [28]. SRv6 can become a key enabler for the Future Internet Architecture, where a high level of flexibility is a required. SRv6 has been recently introduced in Linux Networking (with 4.10 kernel) and its standardization is progressing in IETF [29] [34], where it has been recently proposed for the user plane of mobile networks [30]. Consequently, the 3GPP has recently started a study item that will consider SRv6 as a candidate for user plane in 5G [31].

In the context of Superfluidity, we have explored this technology because it offers the possibility to configure a Service Chain only operating at the edge of the network, with no need to interact with the nodes in the middle of the chain. We have contributed to an Internet draft on using Segment Routing for Service Function Chaining [33] and we have participated to the realization of a proof of concept using the features of the Linux OS [32]. It is evident that, although being a very promising technology, there is still work to be done integrate it in production scenarios using the reference NFV technologies (e.g. OpenStack at the VIM level and the various NFV Orchestrators). For this reason, the said SFC/SRv6 proof of concept [32] has not been integrated in the overall Superfluidity demonstrator.

A3.7 OpenShift-Ansible

To provide the flexibility required by Superfluidity as well as future 5G deployments, we have worked on a set of ansible playbooks that allows to easily consume all the above features, as well as to better integrate with current cloud environments. This set of playbooks (<https://github.com/openshift/openshift-ansible>) allows to install OpenShift on top of different cloud infrastructures: OpenStack, Amazon, Azure, and Google Compute Engine.

Our efforts within the Superfluidity project has focused on the OpenStack provider as this is the one we suggested for our architecture:

<https://github.com/openshift/openshift-ansible/tree/master/playbooks/openstack>

Our main contributions are related to the implementation of kuryr roles inside the openshift-ansible playbooks so that kuryr components can be deployed on the OpenShift installation on OpenStack,



using the above mentioned features (ports-pool, containerized and cni-split). These new roles enable the installation of OpenShift in an existing OpenStack deployment with kuryr configured by just executing one playbook.

A3.7.1 ManagelQ integration

It must be noted that by combining openshift-ansible playbooks with the support at ManagelQ to execute ansible playbooks, we are now able to also easily deploy OpenShift clusters in existing OpenStack deployments (on top of VMs) with kuryr support from ManagelQ.

More work has been done at the integration between openshift-ansible and ManagelQ for an even more simple user experience. Thanks to this integration a tenant can simply ask for an OpenShift deployment from ManagelQ UI (e.g., ask for a deployment with 1 master node, 1 infra node and 5 worker nodes) and it will be automatically deployed by executing the openshift-ansible playbook with the configured parameters on the selected OpenStack provider, i.e., on top of OpenStack VMs , giving the user a functionality similar to have: OpenShift as a Service.

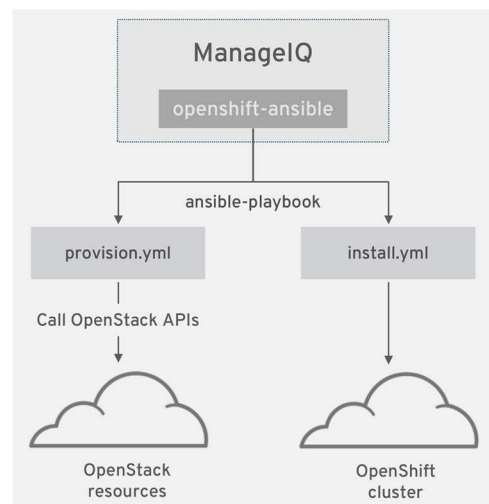


Figure 50: OpenShift-Ansible integration

A3.7.2 Baremetal Support

Finally, in addition to the previous support, we have also extended the openshift-ansible playbooks to also support provisioning baremetal nodes. This adds the flexibility of having an OpenShift/Kubernetes installation that runs both on top of OpenStack VMs as well as on baremetal nodes. This is specially relevant for NVF deployments where some components may require to run (containerized) on baremetal nodes for increase performance, but still being on OpenStack neutron networks so that they can talk to other VMs or nested containers transparently.



As presented in next figure, these modifications involve creating different type of application nodes (where the pods run) where they can be either Virtual Machines created on OpenStack by using the Nova component, or physical servers (a.k.a. baremetal node) created on OpenStack by using the Ironi component.

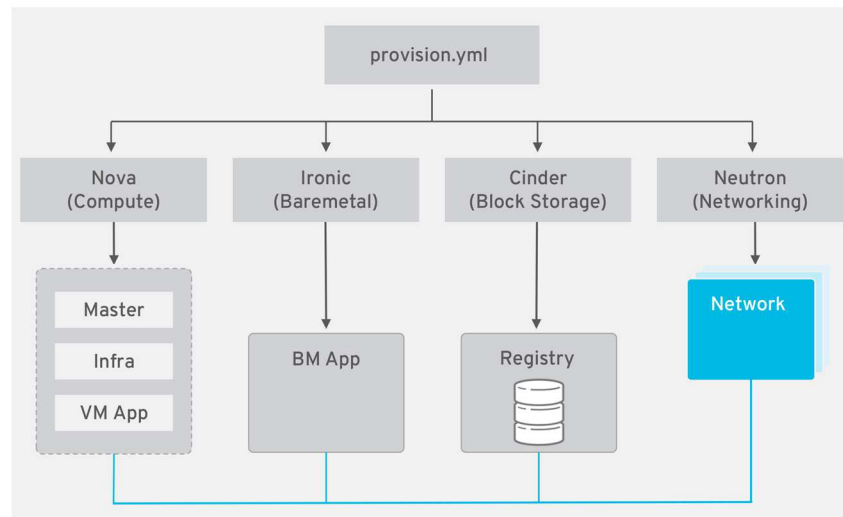


Figure 51: Provision Interactions Overview

To fully enable this work, changes were not only needed at the openshift-ansible side, but also at Kuryr. We are working on adding support for multi-pools:

- <https://bugs.launchpad.net/kuryr-kubernetes/+bug/1747406>
- <https://review.openstack.org/#/c/528345>

The main reason behind this proposal is the existence of different type of nodes that will need to use different type of neutron drivers (i.e., baremetal and nested-vlan pod vif drivers). This forces us to make the kuryr-controller able to handle different type of pod vif drivers, and in turn having different pool drivers to also speed up the creation of pods regardless of the pod vif used. To do that, the multi-pool support enables different pool drivers (and pod vifs) at the same time, and it reads the driver that it needs to use depending on the node where the pod is being deployed.

In addition to this modification, we extended the ManageIQ support for openshift-ansible playbooks to also ask for the number of VMs and baremetal that the user would like to have for its OpenShift deployment. A Proof-of-Concept of these modifications was presented at DevConf 2018 in Brno as a keynote:



- <https://www.youtube.com/watch?v=BloeOXBsETo>

A3.8 RDCL 3D

The vision of the Superfluidity project is to orchestrate functions dynamically over and across heterogeneous environments. It is not possible to consider a single language and tool to cover the diversity of the heterogeneous environments. In the Superfluidity architecture, the different languages for the description, composition and orchestration of services and service components are referred to as RDCL (RFB Description and Composition Languages). The project has designed and developed a tool called RDCL 3D (Design, Deploy and Direct) to assist in the editing of the descriptors of services / service components and in the interaction with different types of orchestration tools. The RDCL 3D tool is not focused on a specific data model / description language but it is meant as a framework that can be specialized for any data model / description language.

RDCL 3D offers a web GUI that allows visualizing and editing the descriptors of components and network services both textually and graphically. A visualized network service designer can create new descriptors or upload existing ones for visualization, editing conversion or validation. The created descriptors can be stored online, shared with other users or downloaded in textual format to be used with other tools. In particular, these descriptors can be used for the deployment and operational management of NFV services and components.

Figure presents a screenshot of the RDCL 3D web GUI, where users can perform intuitive drag-and-drop operations on the graph representation of RDCLs. The source code of RDCL 3D [10] is released under the Apache 2.0 Open Source license, to facilitate its uptake by research and industrial communities [11]. In addition, a live deployment of the system can be found online at: <http://rdcl-demo.netgroup.uniroma2.it>, where users can login using a guest account and explore all RDCL 3D capabilities. It is also possible to request an account that allows saving and retrieving projects and related descriptor files across different sessions.

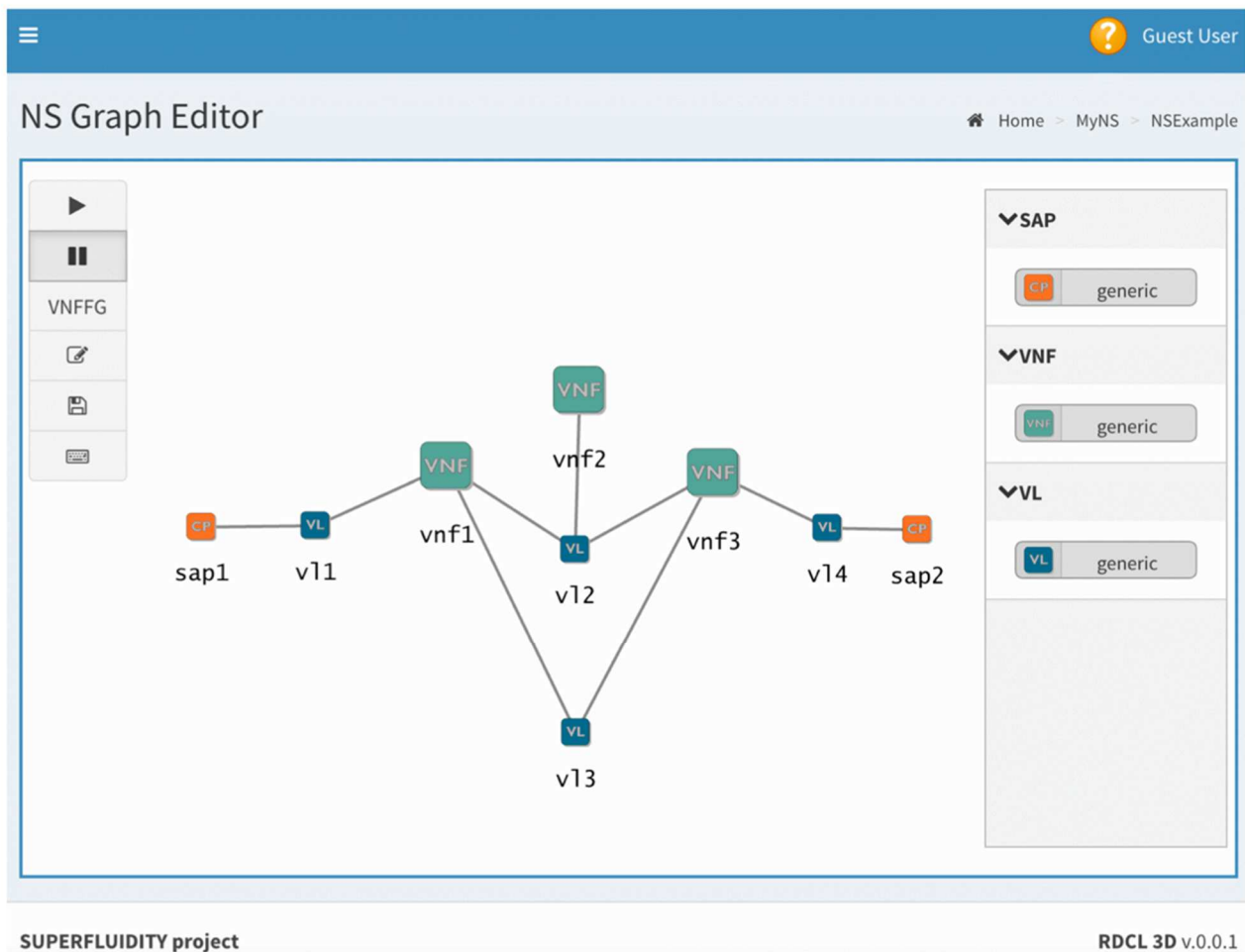


Figure 52: Network Service design using the RDCL 3D GUI

Currently the RDCL 3D framework supports the models defined by the latest ETSI NFV ISG specifications [12] [13], the TOSCA Simple Profile for NFV [14], the TOSCA Simple Profile in YAML [15], the network elements described in the Click configuration language [16] and the Open Source Mano information model [17]. However, RDCL 3D is designed for extensibility in order to support new models or combine existing ones.

With reference to the ETSI MANO architecture, RDCL 3D can play different roles, which correspond to different usage scenarios:

1. RDCL 3D can be used as a standalone tool to edit the NS and VNF descriptors, as shown in <1> in Figure 53. This approach is currently implemented in the demo for the ETSI and TOSCA models. The produced descriptor files can be manually retrieved and provided to an Orchestrator (NFVO).
2. RDCL 3D can support the direct interaction with programmatic APIs of external Orchestrators (<2> in Figure 53) by including the logic for such interaction in an Agent module. In this case,



the Agent module receives instructions from the RDCL 3D web GUI, hands the descriptor files to the external Orchestrator, and provides feedback to the user on the GUI. Note that this scenario can be extended when there is the need to combine different orchestration platforms with different description languages, so that RDCL 3D can become a *meta-orchestrator*.

3. Another usage scenario that we have considered is to use the platform to build Orchestrator prototypes, so that RDCL 3D plays the role of the NFVO (<3> in Figure 53). This is especially useful when one needs to explore new functionalities and it is easier to have a small stand-alone proof-of-concept implementation rather than integrating the new functionality into a fully-fledged NFVO.
4. A different usage of the proposed network is to be integrated as a library within Orchestrators that do not yet support a GUI as shown in <4> in Figure 53.

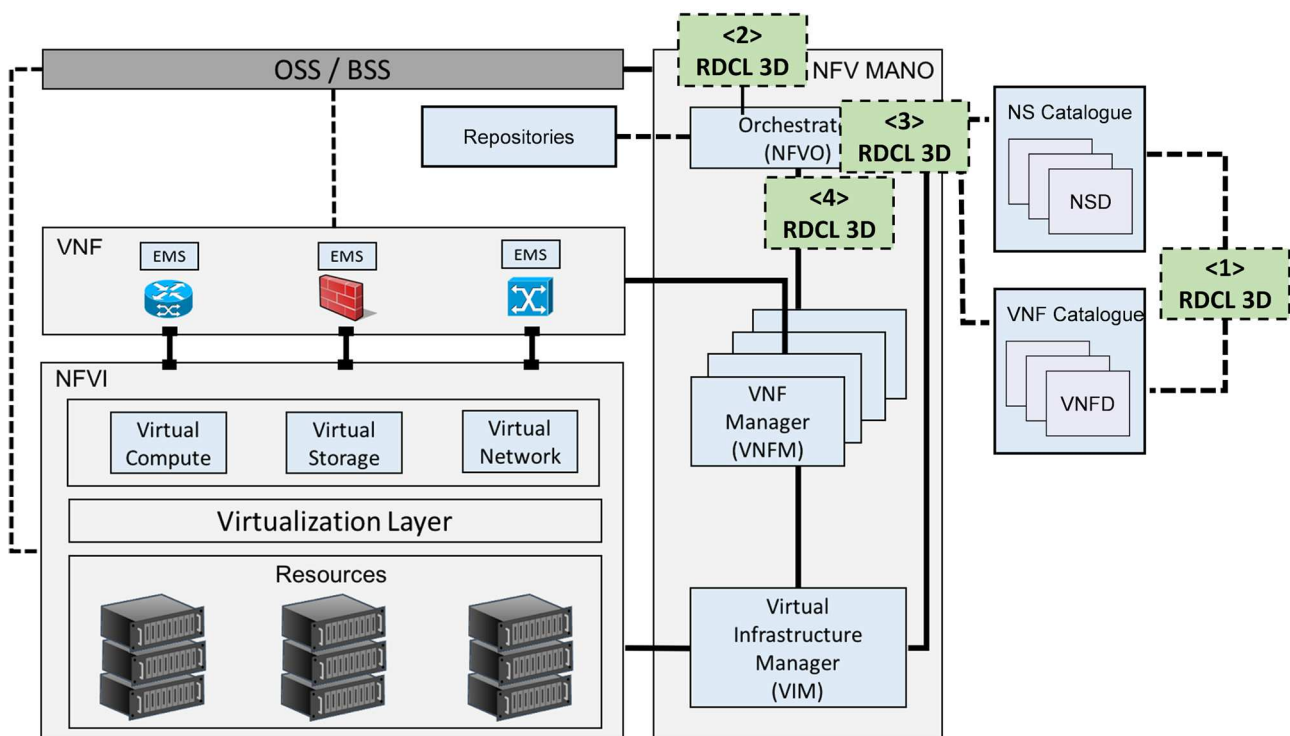


Figure 53: Positioning RDCL 3D in the ETSI MANO architecture

A3.8.1 Integration between RDCL 3D and ManageIQ

The integration between RDCL 3D and ManageIQ is implemented through the following steps:



- The user/network designer employs the RDCL 3D GUI to design or load a network service using the Superfluidity/ETSI language
- The network service is translated by an RDCL 3D agent into HEAT or Kubernetes templates, which are included in an Ansible playbook. The Ansible playbook contains the network service templates along with the commands (for OpenStack or OpenShift) to deploy them
- The RDCL 3D agent commits the Ansible playbook in a local git repository and pushes the changes to a remote git repository to which ManageIQ has access
- ManageIQ pulls the Ansible playbook from the git repository and executes the Ansible playbook.

A3.8.2 Software Architecture

RDCL 3D is a web application with backend and frontend components, as depicted in Figure The backend component running on a web server is based on the Python Django framework. It includes the persistence layer (database). The frontend component running in the web browser is developed in JavaScript and exploits the D3.js library. The platform is designed with a modular approach both in the backend and in the frontend, so that it can be easily extended to support new project types. Each project type can be seen as a “plugin” for the RDCL 3D framework, composed by a backend plugin (in Python) and a frontend plugin (in JavaScript).

Each user (e.g. a service designer or network administrator) can instantiate projects of the supported project types. The descriptor files for the projects are stored in the persistence layer in the backend. Predefined descriptor files are available for each project type (i.e. they represent examples of services or existing components that can be reused). A Data model for a project is created in the backend by parsing and validating the descriptor files (<1> in Figure 54). This process is specific for the project type, therefore it is performed by the specific plugin for the project type. The instance of the Data model contains all the information of a project instance i.e. all the information contained in the project descriptor files. The Data model is send to the frontend (<2> in Figure 54), where is it processed and filtered to produce the different graphical views on the browser GUI (<3> in Figure 54). The project descriptor files are also sent to the frontend. The operations on the GUI (e.g. adding or removing nodes and links, editing of the local descriptor files) are reflected on the local version of the Data model and sent to the backend when it is needed to update the information stored in the persistence layer (e.g. the descriptor files). When interacting with orchestrators which provide their own catalogue service (e.g. Open Source MANO), the backend is responsible for retrieving and then uploading descriptors to the orchestrators.

The backend can also optionally deploy the designed network services through an *RDCL 3D Agent*. An RDCL 3D Agent can deploy the designed network services by either interacting directly with the APIs



of an orchestrator or a VIM or by pushing the network service descriptors to a git repository to which the Orchestrator/VIM has access. Moreover, to allow the interaction with specific VIMs, the RDCL 3D Agent can also perform online translation between descriptor formats. The RDCL 3D Agent exposes REST APIs, which are invoked by the Deployment Handler plugin. Our RDCL 3D Agent implementations are based on node.js, but any technology which allows to expose HTTP REST APIs can be employed.

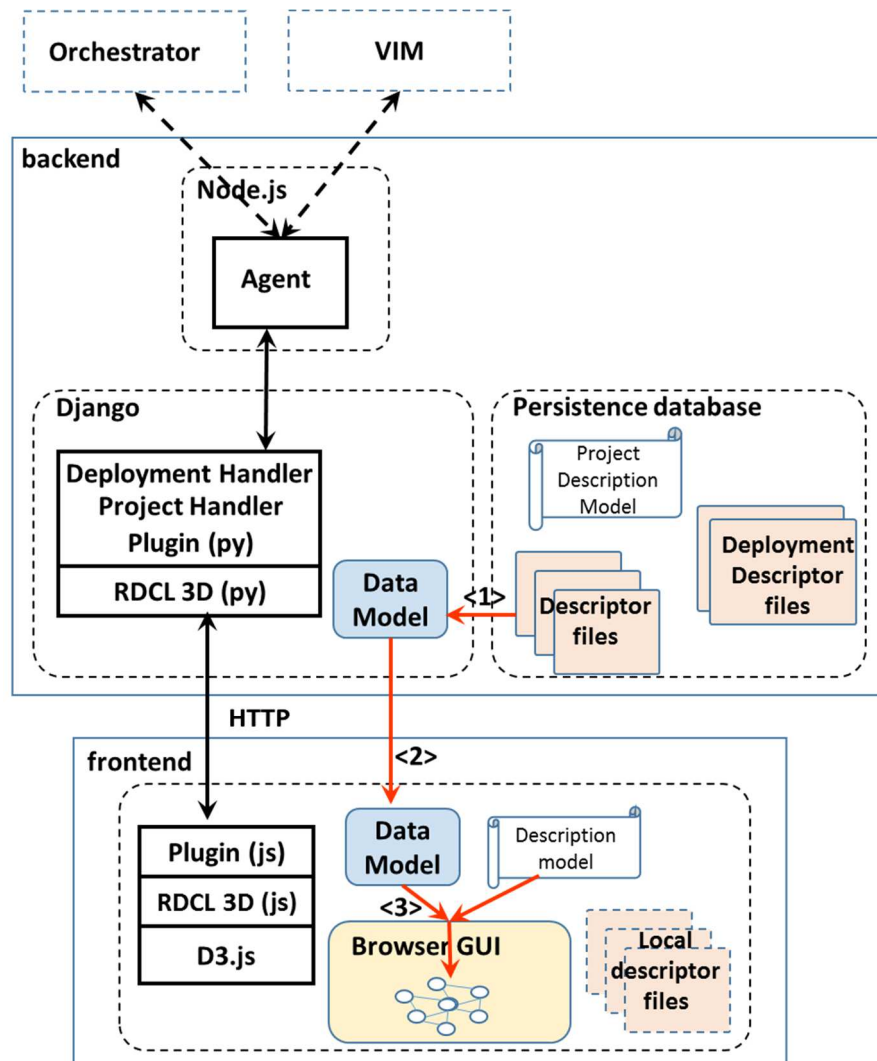


Figure 54: RDCL 3D Software Architecture

The operations performed by the frontend, like visualizing a view of the graph, adding/removing nodes and links, are dependent on the project type, so that they should be handled by the JavaScript plugins. We are able to minimize the code that needs to be developed in a plugin to support a project type by introducing a *description model* for a project type. The description model includes the types of nodes and links that are supported, their relationships, the constraints in their composition,



describes what are the different views of the projects and which nodes and links belongs to which view. The description model is expressed as a YAML file. By parsing it, the JavaScript frontend is able to perform most of the operations without the need of specific code for the project type. In order to handle the operations specific to the project type, the description model includes the possibility to associate operations to functions that are defined in the plugin. The definition of the structure of the description model and some examples are included in the *docs* folder in [5].

To simplify the integration of new project types, the framework includes a script that creates the skeletons of the Python and JavaScript plugins (respectively for the backend and the frontend) and of the description model. Starting from these skeletons, a developer adding the support of a new project will:

1. include the parser to translate the descriptor files into the Data model representation in the Python plugin (backend);
2. customize the description model, capturing all the relevant properties of the project type and identifying the operations that need to be processes in a specific way for the project type by the JavaScript plugin;
3. develop the project type specific processing operation in the JavaScript plugin (frontend);
4. optionally, develop a deployment handler and an RDCL 3D Agent to allow the direct deployment of network services.

Summarizing, RDCL 3D is a web framework for visualizing and editing services and components in NFV scenarios. RDCL 3D is not focused on a specific data model / description language. It is designed to facilitate the support and the integration of any model and language: i) it has a modular architecture in which a new project type can be added as a plugin; ii) a description model allows describing the structural properties of the project type, minimizing the need to develop code; iii) a script is used to generate the skeletons of the plugin and of the description model, to reduce the development effort.

A3.9 Optimization of Service Deployment Templates using a Service Characterisation Framework

A key focus area in Task 6.1 is the control framework which addresses resource allocation for virtualised network functions and services. Resource allocation is important in the context of “virtualisation” of network services as it plays a very important role in both service assurance and Total Cost of Ownership (TCO). It is important to allocate the right quantity and type of resources for service execution in order to support a required service level, which is usually measured against Service Level Objectives (SLOs). It is also equally important to prevent overprovisioning of resources



as this leads to higher TCO, due to unused resources and, in some cases, can result in service performance degradation.

In Cloud Computing environments, allocating resources to enable execution of a service is performed by an orchestrator (such as OpenStack Heat, ETSI Open Source Mano (OSM), etc.). In order to fulfil user requirements, the orchestrator takes as input a service descriptor, which specifies types and quantity of the resources for each service component (for instance, the number of vCPUs, the number and type of vNICs, the preference for enabling enhanced platform features, etc.). The requested resources are then allocated at deployment time through a virtual infrastructure manager (VIM).

The presence of enhanced platform features and their specific configuration options, such as core pinning, Linux kernel hugepages, SR-IOV enabled Network Card Interfaces (NICs), and so forth, can be exploited to further improve service performance, if required by the user. However this strictly depends on the availability of those resources within the infrastructure landscape and their relevance to a given service. The current approach for resource allocation is based on pre-defined deployment descriptors which are defined by the service vendors: the orchestrator takes them as an input and requires the instantiation of such a pre-defined configuration to deploy the service on the specific hardware platform at deployment time. This service descriptor is defined by the vendor on the basis of the specific infrastructure available during on-boarding tests and does not take into account all potential infrastructural differences (related to hardware and software configurations) within the service provider's production environment.

A3.9.1 Service On-Boarding Characterisation

The main focus of the service characterisation activities is to support automated characterisation of the relationship between service performance and resource allocation cost on a Network Function Virtualised Infrastructure (NFVI). Characterisation is carried out prior to initial placement of a new service into production, resulting in performance optimisation of the services based on the available hardware ingredients. For this reason, this approach is part of the pre-deployment characterisation of a service. In the context of Task 6.1, the design and implementation of an automation framework is envisioned in order to automate some aspects related to the generation of placement insights for an orchestrator. The primary goal therefore is to automatically define a set of rules that can be interpreted by an orchestrator in order to make intelligent decisions on the quantity and type of resources to be allocated to a service by a VIM.

Automation is key to determining the best composition of quantity and types of resources to be allocated to a service according to its required KPIs and SLOs and considering changes in operational conditions such as variations in user load. Making automated and performant deployments decisions



enables support for performance requirements in a scalable manner, which results in increased efficiency in the management of features exposed by the platform and the infrastructure resources. The results of the characterisation procedure in this context are optimised service descriptors which contain the values for each configuration parameter of interest corresponding to the most efficient resource allocation option in order to satisfy the customer specified SLOs. This supports the **[KpiTemplate-01]** requirement: “The system must be able to dynamically define a workload deployment template to ensure that resource allocations can support required SLA’s and SLO’s”, as outlined earlier in section A3.1.1

A3.9.2 Characterisation Lifecycle

There are three phases necessary in order to characterise a service and optimise its deployment template:

- **Experiment Execution:** automatic deployment of the service and execution of stress tests.
- **Data Collection:** collection of metrics from workload generators and telemetry agents to collect metrics related to the service’s behaviour under representative operation scenarios.
- **Data Analysis:** extraction of information from the data collected and generation of orchestration insights.

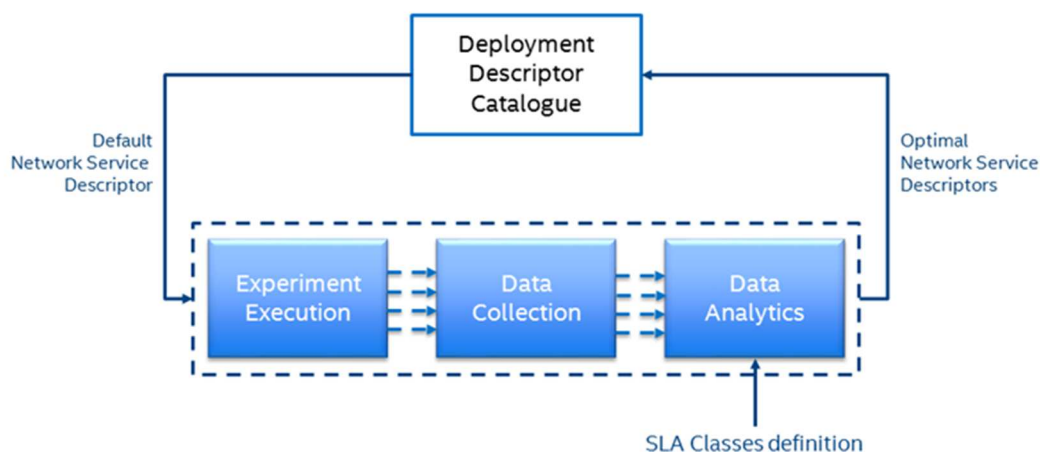


Figure 55: Characterisation Phases

Characterisation of a service starts with the Service Descriptor taken from a given descriptor catalogue. Examples of catalogues include OpenStack’s Murano project and OSM’s Network Service Descriptor (NSD) and Virtual Network Function Descriptor (VNFD) catalogues.

The following sections outline the details of each phase shown in Figure and highlight some important requirements.



A3.9.2.1 Experiment Execution

The framework automates the execution of experiments analysing different configurations in order to determine the relationship between service performance and the deployment configuration parameters. There are a multitude of possible parameters of interest for a given configuration selection. Parameters of more potential significant in the context of Superfluidity are:

- The *flavour* of the virtual components, as per the number of cores and the amount of RAM to be used by each component of a service;
- The *vNIC type*, which could be based on either virtual switch technology (such as Open vSwitch) or pass-through (such as SR-IOV);
- The *core pinning policy*, which may or may not allow core isolation on a given compute node;
- The *memory page size*, which could allow usage of either small or large memory pages to improve the efficiency of memory management.

In order to identify the specific configuration to be used, the characterisation methodology has to stress test the deployment of the service changing specified configuration values and measuring the impact that the changes have on performance.

For the configurations of interests, a single experiment can be defined as per the following sequence of actions:

1. Deployment of the service according to the predefined configurations options;
2. Deployment of any noisy neighbours or concurrent workloads, (if required);
3. Execution of the stress test (in the form of a packet/traffic generation tool);
4. Termination of the service.

Actions 1, 2 and 4 can be implemented by exploiting standard orchestrator functionalities. Therefore, to automatically execute experiments, integration with an orchestrator API is necessary. Action 3 requires the execution of a stress test, such as the application of packet traffic via packet generator, and therefore requires the integration of the framework with appropriate open source or commercial packet generators.

If the configuration space of interest is reasonably small, a brute force approach is plausible, where all possible permutations are repeatedly tested and measured to generate statistically validate data sets. For instance, analysing four parameters and assuming each of them can assume two possible values, if the service is composed of one single component, the total number of possible permutations is 16 and assuming the experiments will be repeated three times in order to validate statistical consistency, this will result in 48 unique experiments, which can be executed in reasonable



time period by an automated engine provided the total deployment time per unique configuration is in the order of minutes.

A3.9.2.2 Data Collection

The collection of data is a very important aspect of the characterisation process. There are two types of data that are of relevance to workload/service characterisation, namely:

- **Stress test metrics:** metrics collected by the packet generator, such as throughput, latency, jitter, etc. It is likely that a subset of these metrics will be considered as SLOs, and represent the target for analysis (for instance, the user could require a throughput higher than a given value, or a latency lower than a given value).
- **Telemetry metrics:** metrics that reflect the effect of the service execution on specific infrastructure ingredients. They are usually represented by hardware counters or ingredient specific measurements (such as CPU, NIC and disk utilisation, etc.).

The main focus of this characterisation activity is on the first category (stress test metrics), since they are part of the SLOs that will represent the constraints of the problem.

This requires integration of the characterisation methodology with the packet generator used for the experiment execution in order to trigger and control the execution of the stress tests as stated before and to collect the results of the test and format/clean the data in a manner suitable for the data analysis pipeline to process them.

A3.9.2.3 Data Analysis

The main scope of the data analysis is focused on the optimisation of the deployment template in order to determine the rules that will be used to eventually generate the configuration.

Those configurations can be seen as a formal representation of the trade-off between the user requirements, expressed in the form of SLOs (and a probability to satisfy them over time) versus the service provider requirements, expressed in the form of a cost.

The constraints taken into account are represented by the SLOs defined in the user requirements. They need to be treated as thresholds (for instance the desired throughput has to be higher than 5 Gbps) and the value of those SLOs have to be within the desired range with a given probability (for instance the throughput is higher than 5 Gbps for 99% of the execution time).

For each configuration option, it is necessary to calculate the contribution to the total cost. For instance, selecting SR-IOV instead of an OvS network connection will increase the cost due to the fact that the availability of SR-IOV channels is finite within a given NFVI. The cost does not necessarily need to be expressed in an actual monetary terms, but it needs to be proportional, such that the



comparison between the different configuration options is meaningful (for instance, it could be based on availability of resources).

Selecting the best combination of configuration options requires the analysis of the data collected during previous phases in order to associate the configuration options to the impact that they have on performance. From this perspective, it is important to identify which are the specific configuration parameters that provide a key contribution to the performance of the service under test as well as the impact on cost. Application of an appropriate data analytic technique or pipeline of techniques is necessary to identify the configuration that satisfies the SLO constraints while providing the lowest cost.

A3.9.3 Exploration of the Deployment Configuration Space for a Virtualised Media Processing Function

A3.9.3.1 Experimental Setup

The testbed used for investigating deployment configurations comprised of a dual socket E5 2620 v3 (15M Cache, 2.40 GHz) based server running an OpenStack Kilo environment as shown in Figure 56. The Hammer workload generator was deployed on a dedicated single socket E5620 (12M Cache, 2.40 GHz) based server and connected to an OpenStack controller node via a 10 Gbps switch. A Heat template was defined to deploy the virtualised media processing function (Unified Origin). The template was manually updated with required configuration settings prior to each deployment. At the end of each of experiment, a python script was used to parse the Hammer results output file and to insert the extracted data into an InfluxDB database. The open source data visualisation platform Grafana was used for inspection of the experimental results.

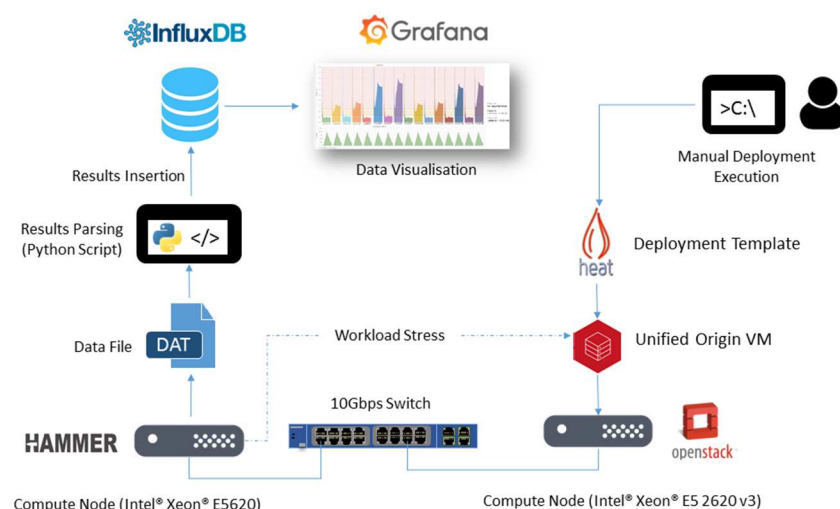




Figure 56: Experimental Configuration

A3.9.3.2 Experimental Results

The definition of a Heat template to be used for the deployment of a virtualised media processing function (Unified Origin) was successfully implemented, as well as integration with Citrix Hammer, enabling the automated deployment in conjunction with stress test execution in an OpenStack cloud environment. An experimental campaign was carried out to identify the performance ranges and the impact that the configuration parameters have on Unified Origin.

The results obtained from 16 different configurations are shown in next Table and were collected using the testbed configuration as shown in Figure.

	vCPUs	RAM	vNIC Type	Memory Page size
Configuration 1	4	4	SR-IOV	Large
Configuration 2	4	4	OvS	Large
Configuration 3	4	4	SR-IOV	Small
Configuration 4	4	4	OvS	Small
Configuration 5	2	4	SR-IOV	Large
Configuration 6	2	4	OvS	Large
Configuration 7	2	4	SR-IOV	Small
Configuration 8	2	4	OvS	Small
Configuration 9	4	2	SR-IOV	Large
Configuration 10	4	2	OvS	Large
Configuration 11	4	2	SR-IOV	Small
Configuration 12	4	2	OvS	Small



Configuration 13	2	2	SR-IOV	Large
Configuration 14	2	2	OvS	Large
Configuration 15	2	2	SR-IOV	Small
Configuration 16	2	2	OvS	Small

Table 7: Configuration settings

The results obtained are shown in Figure 57 and Figure 58, which are snapshots of the data in InfluxDB.

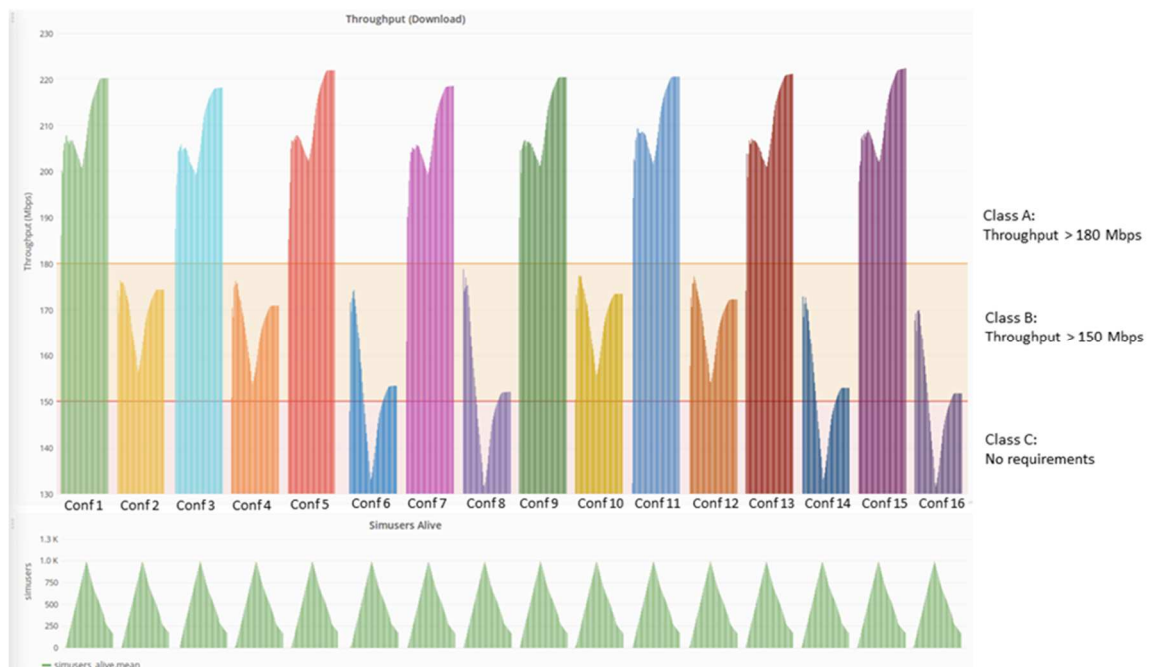


Figure 57: Throughput results for different configurations.

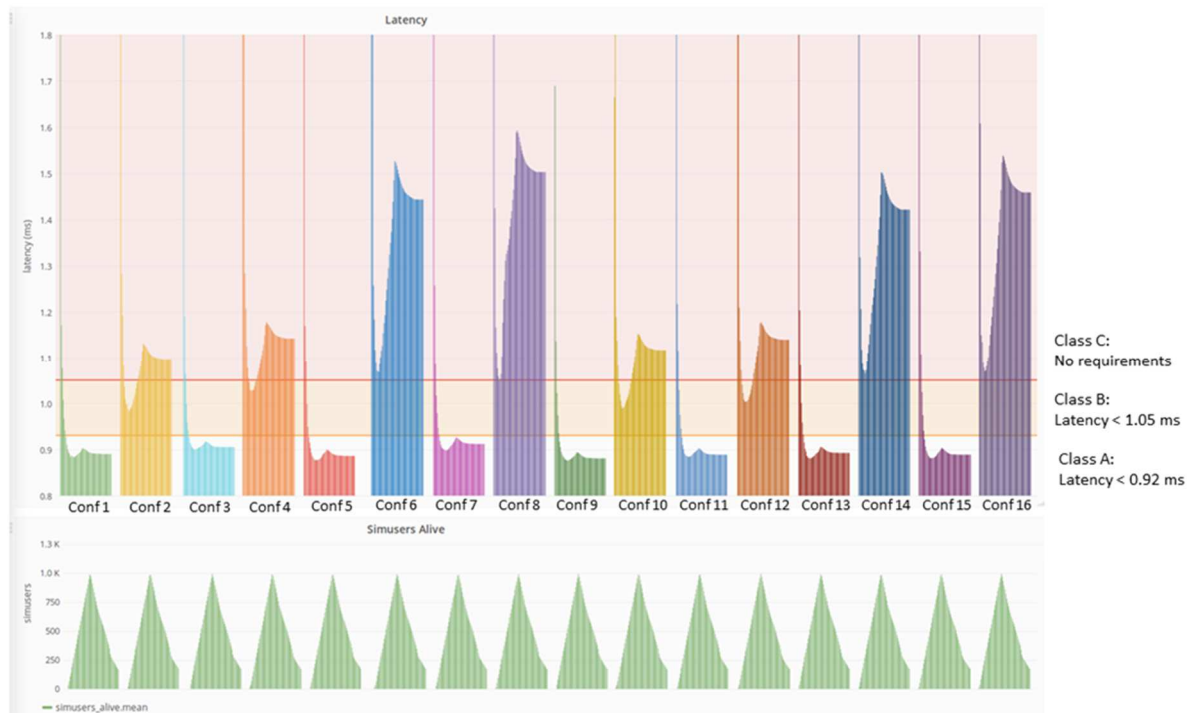


Figure 58: Latency results for different configurations

These results show that varying the combination of the four parameters it is possible to classify the configurations in to 3 unique performance classes:

- Class A: Throughput > 180 Mbps and Latency < 0.92 ms
- Class B: Throughput > 150 Mbps and Latency < 1.05 ms
- Class C: Best effort (no specific requirements) for both throughput and latency

In Figure 57 and Figure 58, the simuser profile used for the experiment is shown. The simuser ramp scales from 0 to 1000 users in the first phase of the experiments and then ramps back down to 0 in the second phase of the experiment. The results also show the impact of increasing the number of users on the KPIs, throughput increases and the latency decreases when the number of user increases and vice versa.

An initial analysis of the results in relation to the configuration parameters highlights how the usage of SR-IOV has positive impact on both throughput and latency and this specific configuration on its own seems to be guarantee of good level of performance, placing the service performance within SLA Class A ranges. At the same time, this represents the most expensive solution from an infrastructure perspective, which might lead the service provider to select an option corresponding to a lower cost in the case the user requirements fall into SLA Classes B.



The results obtained for configurations 6, 8, 14 and 16 highlight that when only 2 vCPUs and OVS based configurations are used both latency and throughput performance is degraded significantly, placing the service performance within SLA Class C.

These results show that the resource allocation policy has a huge impact on the performance of the service. Therefore it is important to identify which hardware features influence a service in order to deterministically achieve target performance levels and to avoid the allocation of resources which have no meaningful impact on performance, therefore avoiding wasteful allocation of resources and reducing service execution costs.

Collection of characterisation results requires human involvement in the execution loop, which introduces a time penalty (for experimental execution and results generation). The automated methodology is designed to minimise human involvement in the execution loop. The results obtained using the manual approach act as a reference or ground truth to sanity check the results obtained by the automated methodology described in the following section.

A3.9.4 Generalised and Automated Generation of Optimised Service Deployment Templates

As previously demonstrated, achieving the performance level defined in Classes A, B or C clearly depends on the resources allocated to the service. The set of resources assigned to a service for its execution is defined as the deployment configuration of the service. The deployment configuration of a service can therefore be associated to a cost, which relates to the quantity and type of resources in the configuration. Every deployment configuration can also be associated with a given performance level, expressed using SLOs, such as throughput and latency.

As outlined previously selecting the configuration to use in order to achieve a performance representing class A, B or C can be done by humans, through manual inspection and deciding the best set of resources to be used. In order to optimise the utilisation of infrastructure resources and to reduce the cost of service execution, while achieving a desired level of performance, both the cost and performance of every configuration need to be considered. The number of possible configurations is usually high, which makes investigating all possible configurations very time consuming and impractical operationally. The problem is further exacerbated by the large number of virtual services that need to be characterised by Service Providers for inclusion in their service catalogues. The solution to this problem requires two main elements: a high degree of automation and a generalizable implementation.

In previous sections, the characterisation process was described together with a number of requirements. Among them, a number of similarities have been identified with the framework developed during the course of Task 4.1's automated KPI mapping activities, which has been used for KPI Mapping of Unified Origin. As a consequence, the framework previously developed has been



reengineered and extended to accommodate the new requirements identified in the context of Task 6.1, i.e. automated optimisation of service deployment templates.

The proposed methodology finds a generalizable and automated solution by generating a model (using a machine learning approach) which expresses the allocation of resource in relation to specific levels of performance, from which rules are extracted and used to add intelligence to the decision process, determining which resources to use for the execution of a given service.

A3.9.4.1 Performance Measurements Based on SLOs

When using SLOs to measure performance of a service, it is important to decide how to represent the performance classes (such as Class A, B and C defined in section x.xx) from a statistical perspective (choosing from measuring the average, the median and variance, the minimum/maximum, etc. of a SLO over time) since this choice clearly impacts the meaning and the reliability of the results. To solve this problem in the context of the methodology defined in this section, the throughput measurement methodology as specified in the RFC 2544 (<https://www.ietf.org/rfc/rfc2544.txt>) was utilised. This applies to the measurement of the performance of a Device under Test (DUT) acting as middle-box in a network. The throughput measurement as defined by the RFC 2544 requires the middle-box to maintain a constant level of throughput for the full duration of the test trial: if the throughput falls below the required threshold during the trial, the current level of throughput is determined not to be supported by the middle-box and it is necessary to use a lower value of throughput and to repeat the test until the DUT supports the current throughput level for the full duration of the trial. The highest throughput which satisfies this condition is taken as the maximum supported throughput for the DUT. Since the methodology defined in this section needs to be applied to generic services and not only to middle-boxes, it requires the definition of measurement criteria which can generalise the process. From this perspective, there are different aspects which need to be considered, some of which are service-specific:

1. Definition of a specific traffic/user profile for the service to model the behaviour of real life traffic/users;
2. Ability to run a required configuration profile against a virtual service instance deployed on a cloud infrastructure using a traffic/packet generator with fully automated control;
3. Implementation of a system to measure the performance level (in terms of SLOs) which expresses a performance measurement that is appropriately contextualised for the service under consideration.

For this reason, some effort was spent to implement new features in the Service Characterization Framework such that it could be extended by adding Use Cases. A use case is intended as a flow of actions designed for a specific service in order to support: the collection of user input for the definition of traffic profiles which might change across different use cases; the configuration and



control of a packet/traffic generation technology to stress test the service; the collection of performance measurements specific for the service under test.

The user is also required to specify the SLOs of interest and if they are required to be minimised or maximised. For example, the user may want throughput to be maximised and latency to be minimised. This information is used by the methodology to decide if the data points in the test trial can be considered part of a given Class (for instance, in Class A the throughput threshold will be considered as the one where the throughput is equal or **greater than** 180 Mbps, whereas for the latency threshold it will be equal or **lower than** 0.92 ms). Following this approach, the minimum value are taken into account for SLOs that need to be maximised (i.e. throughput), whereas the maximum value will be taken into account for SLOs that need to be minimised (i.e. latency).

A3.9.4.2 Automated Methodology for Deployment Rule Extraction

Automating the identification of patterns in the relationship between resource utilisation and performance is necessary to solve the problem in a scalable manner. However, the implementation of an intelligent methodology that can reason over the appropriate decision is non-trivial. To solve this problem, a fully automated pipeline was defined as part of Task 6.1 activities which is composed of data collection, data manipulation and machine learning steps. This ultimately has been integrated into the Service Characterisation Framework and allows the collection of data relating to the behaviour of a service. It also allows the identification of patterns across different configurations of the same service and to determine a set of rules to combine into a service-specific policy in a fully automated manner, without any prior -knowledge of the service.

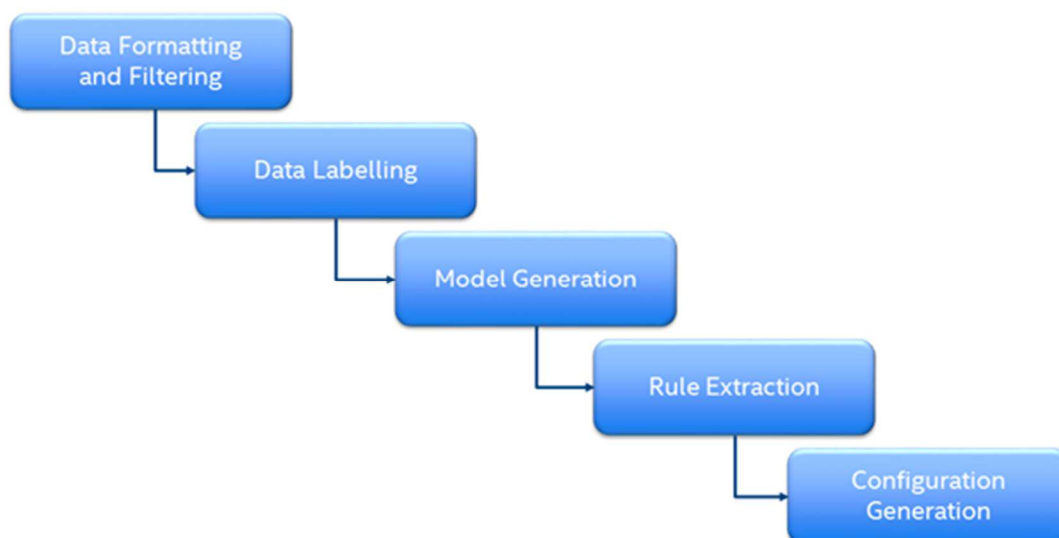


Figure 59: Template Optimisation Methodology Pipeline



Once the data related to the experiments is collected, the following steps are performed by the framework:

1. **Data formatting and filtering:** Data for all the experiments is merged in a single data frame, taking all the available samples. The number of samples depends on the duration of the experiments (which is defined by the user when defining the Hammer profile and ramp) and the sampling rate used by Hammer (the default value is 1 sample every 3 seconds). After the data merge, all the data points are analysed to identify outliers which are removed from the data set.
2. **Data Labelling:** The values of the user defined SLOs are analysed and the class of each data point is determined accordingly. Every data point is labelled with the name of its corresponding class, as per the user defined SLA classes (for instance, Class A, B and C).
3. **Model Generation:** The classification problem was solved using the C4.5 machine learning algorithm [37]. This approach classifies each data point creating a decision tree model that takes into account all available test trials with the same deployment configuration. For this application decision trees provide a stable classification approach particularly when dealing with limited amounts of data and when anomalies are present in the data.
4. **Rule Extraction:** Once the model has been generated in the form of a decision tree, the rules used by the decision tree are extracted and processed, in order to discover the most important configuration parameters to facilitate the classification. Processing the rules allows the system to discover the key configuration parameters first, which are then coupled with the cost information. This in turn allows the selection of the deployment configuration that provides the lowest cost among the available options.
5. **Configuration Generation:** Once the configurations are listed and connected to the cost information, the configuration with the lowest cost for each user defined class is selected and included into the results, which are converted into a JSON structure and exposed to an Orchestrator.

A3.9.5 Service Characterisation Framework Implementation

The Service Characterisation Framework designed and developed in the scope of Task 4.1 in WP4 was extended during the Task 6.1 activities with new features in order to accommodate the requirements of methodology developed. For instance, the capability of implementing new Use Cases to support the generalisation of stress testing services. This allows the Service Provider to easily integrate various packet generators (such as IXIA, Spirent, Agilent, etc.) by developing plugins for the Service Characterisation Framework and integrating the plugin into the automated experiment execution



module. In order to implement this feature the “Stress tools” box in the architecture was re-engineered as well as its interface with the core of the framework supporting a plugin system. The Hammer Use case was re-engineered to support this architecture change. Also the capability to collect input from the user was enhanced in order to define different profiles as necessary to stress KPIs differently for the service. In the latest version, the use cases take as its input a portion of a YAML file which defines the use case parameters. For instance, Hammer can be configured using the following YAML syntax:

```
profile_name: mixed-profile # Name of the profile to use
step_deltas: [20] # Number of Users to add per step
step_periods: [10] # Number of seconds of each step
step_repetitions: [100] # Number of steps
```

These new features allows the user to define the arbitrary behaviour of the packet generator, as requested by a specific test and represents the basis of automated experimental execution. The current high-level architecture of the framework is shown in Figure 60.

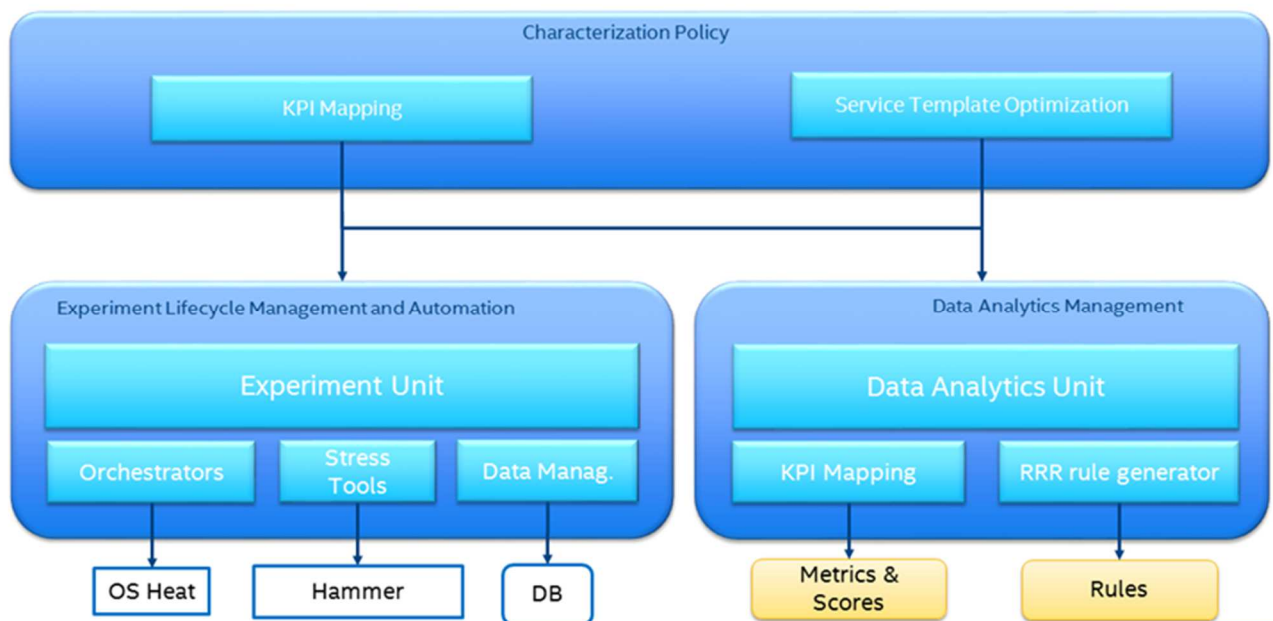


Figure 60: Service Characterisation Framework high-level architecture.

Both the KPI mapping and the Service Template Optimisation policies can be executed by the framework, which provides a common interface to all policies supported by the Experiment and the Data Analytics Units. This allows execution of both the KPI Mapping and Service Template Optimisation policies for Virtualised Video Processing Function (Unified Origin).

The **Experiment Unit** provides automatic management of the experiment lifecycle as described before: supports the deployment of a service, the deployment of any desired noisy neighbour e.g. CPU, network I/O etc., the execution of a stress test and termination of services. In order to



implement this capability, a heat template to deploy Unified Origin was defined that enables the automated deployment of the service. The Experiment Unit manages the generation of all the user required permutations and their execution lifecycle.

To execute the deployment of the services the Experiment Unit is integrated with OpenStack Heat and ETSI OSM. Integration with OpenStack Heat supports the deployment of a service based on a Heat template, whereas with OSM the user is required to specify the name of a service currently available within the NSD catalogue.

To run a stress tests appropriately for Unified Origin, the framework was integrated with Citrix's packet generator Hammer. The framework creates a Hammer profile for a simulated users (simuser) profile in accordance to a specific user definition (e.g. request type, number of users, linear to stepwise user ramp etc.), which is used to stress test different deployment instances of Unified Origin. The user can specify to the framework which kind of simuser profile to use, defining the name and all the necessary details of the user ramp dynamically. Additionally, integration allows at the end of an experiment the collection the metrics stored by Hammer. Those metrics are collected by the framework by automatically establishing an SSH session to the Hammer node and copying the data file (CSV format) for extraction and storage in an InfluxDB database in a time series format. The **Data Analytics Unit** provides automatic analysis of the experiment data. It loads the experiments data according to the specific type of processing required. The data analytics module which was initially developed support KPI Mapping in Task 4.1 (see D4.1) was extended to support Service Template Optimisation.

A3.9.6 Automated Methodology Results

In order to characterise the workload the Service Characterisation Framework was used to run 3 different iterations for each of the 16 configurations listed above (for a total of 48 experiments). This is supported by a YAML configuration file used by the framework, which is composed of different sections as discussed below.

Service definition Section:

```
service_id: ./heat_templates/unified_origin.yaml
deployment_parameters:
  vm_ram: [2048, 4092]
  vm_vcpus: [2, 4]
  vnic_type: [normal, direct]
  memory_page_size: [small, large]
  cpu_policy: dedicated
```

Experiment section:



```
instance_name: unified_origin
use_case:
name: hammer
parameters:
profile_name: mixed-profile
step_deltas: [20]
step_periods: [2]
step_repetitions: [100]
nic: eth5
vlan_id_1: 400
vlan_id_2: 401
hammer_ip_address: 10.1.0.26
```

Characterization Methodology section:

```
iterations: 3
action:
type: template_optimization
parameters:
sla_classes:
class_A:
latency:
- '{}' < 0.92'
curl_bytes_download:
- '{}' > 180'
class_B:
latency:
- '{}' > 0.92'
- '{}' < 1.05'
curl_bytes_download:
- '{}' < 180'
- '{}' > 150'
costs:
vm_ram:
2048: 1
4092: 2
vm_vcpus:
2: 1
4: 2
```



```
vnic_type:  
normal: 1  
direct: 10  
memory_page_size:  
small: 1  
large: 5
```

The configuration file firstly defines the service ID (file name of the Heat template (YAML) for service deployment) and the parameters required for the template, including the features of interest to be used during the deployments, (e.g. network connection type, memory page size, etc.). Secondly, the configuration file specifies the characteristics of the stress to be applied using Hammer during each experimental deployment (e.g. step size of simulated user ramp, number of steps, duration of steps, hammer profile for HTTP_GET requests, etc.). Thirdly it provides information about the characterisation methodology to be used, in this case “template optimisation” and the parameters for the characterisation to be used, like the number of iterations of experiments to be executed, the classes of performance to be explored, (expressed on the basis of the KPIs’ values) and the costs corresponding to each configuration value.

Collecting the results of the 48 experiments, the Service Characterisation Framework applied the template optimisation methodology and returned the results in the following JSON object:

```
{  
  Class_A:  
    vm_ram: 2048,  
    vm_vcpus: 2,  
    vnic_type: direct,  
    memory_page_size: small  
    cpu_policy: dedicated  
  },  
  {  
    Class_B:  
      vm_ram: 2048,  
      vm_vcpus: 4,  
      vnic_type: normal,  
      memory_page_size: small  
      cpu_policy: dedicated  
    }  
  {  
    None:
```



```
vm_ram: 2048,  
vm_vcpus: 2,  
vnic_type: normal,  
memory_page_size: small  
cpu_policy: dedicated  
}
```

The class “None” is related to the best effort behaviour, (which corresponds to Class C in previous data set). The methodology provides full execution of the pipeline, from experiment execution to the extraction of the policy into JSON format. The full execution cycle took approximately 10 hours to complete without any human intervention, after the definition of the YAML configuration file for the framework. Examination of the results and comparing them with the explorative data set, the template optimisation methodology was able to successfully extract the patterns from the tested configurations and interpret them correctly, generating the rules for the optimised template. These rules corresponded to the minimum impact on cost, according to the user input. The rules were outputted as a JSON file representing the 3 optimal set of configuration values to use for the 3 deployment templates (one for each class).

For Class A the importance of having SR-IOV (which is indicated with `vnic_type = direct`) was identified, whereas all the other variables become useless once SR-IOV is enabled, and therefore are set to the lower cost values.

For Class B, in the same manner, the vCPUs were identified as the key configuration parameter in order to achieve the required performance level. For Class C the minimum cost configuration was selected by the methodology.

A3.9.7 Superfluidity System Integration

As described in the previous sections, the focus of this work has been the development of an automated methodology which can define optimised deployment templates to deliver specific levels of performance in compliance with SLOs. This work provides input and is exploited by the following scenes which are being implemented as part of the overall Superfluidity demonstrator in WP7.

- Scene 2a - Offline Workload Characterisation
- Scene 2b - Streaming service deployment
- Scene 3a - Core Service Automatic Scaling

The functionality additions to the Service Characterisation Framework are being used in the workload characterisation activities. The optimised deployment templates generated by the framework are



used in scenes 2b and 3a to deploy the virtualised video processing component (Unified Origin) in an optimised manner.

A3.10 MicroVisor Orchestration

One of the most challenging requirements captured in Section 2 is [AppSched-03], that states that an application must be provisioned in $<10\text{ms}$. For a VIM such as OpenStack, this poses a number of challenges as this Python-based framework was designed for ‘traditional’ VMs that usually comprise a Linux or Windows-based guest OS. These VMs are heavy-weight and need miniaturization before they could start in the order of seconds. Containers and other light-weight virtualization techniques as those currently investigated in Superfluidity can start up much faster. When looking to approach fast provisioning and orchestration tools it is important to profile all aspects of the virtualization workflow.

In order to support $<10\text{ms}$ provisioning times it is important to consider the design of the orchestration platform and to remove overheads. The MicroVisor, Hypervisor platform that OnApp are bringing to Superfluidity is purpose-built, light-weight, distributed and focused on maximising the performance of virtual workloads running on distributed resources. As such, there have been improvements carried out to the MicroVisor orchestration framework that can be used to help decide on decisions for the rest of the Superfluidity orchestration tools. The MicroVisor UI is based on OpenStack and has had various improvements to be able to manage the expected workloads of Superfluidity.

A3.10.1 UI design for managing a large collection of resources

Virtual workloads that are going to load in $<10\text{ms}$ are potentially going to be far more numerous than standard visualization approaches currently account for. Horizon, which is the OpenStack Dashboard can handle the scale of Virtual Machines that currently are used by large enterprises, but will likely have some scalability issues when faced with orders of magnitudes more VMs than are currently used. A rethink of the UI is therefore needed for it to display the information available to administrators and end-users in a useful manner.

Computer assisted workload placement will therefore move from being just an optimization effort, to being a tool to help manage the workloads at the scales that are expected. A mockup diagram showing a possible visualization of the physical to virtual workloads is shown in *Figure 61*. This Figure captures the physical, network overlay and virtual resources and how they relate to each other. The work is ongoing to determine which visualization mechanisms users and administrators will find useful.

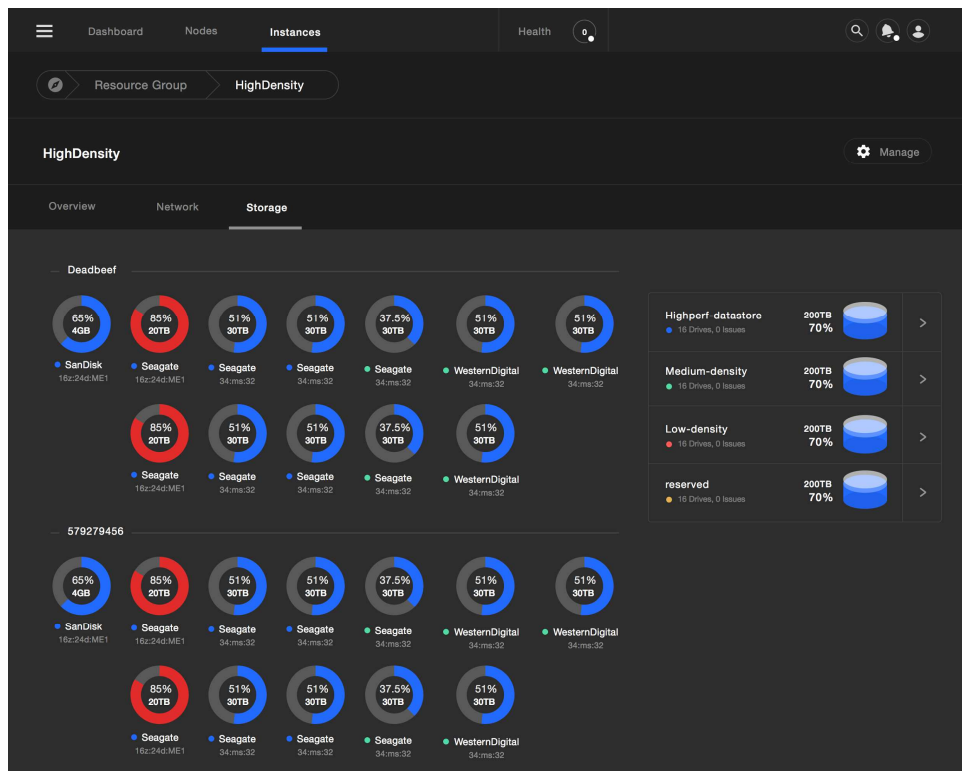


Figure 61: Visualization UI

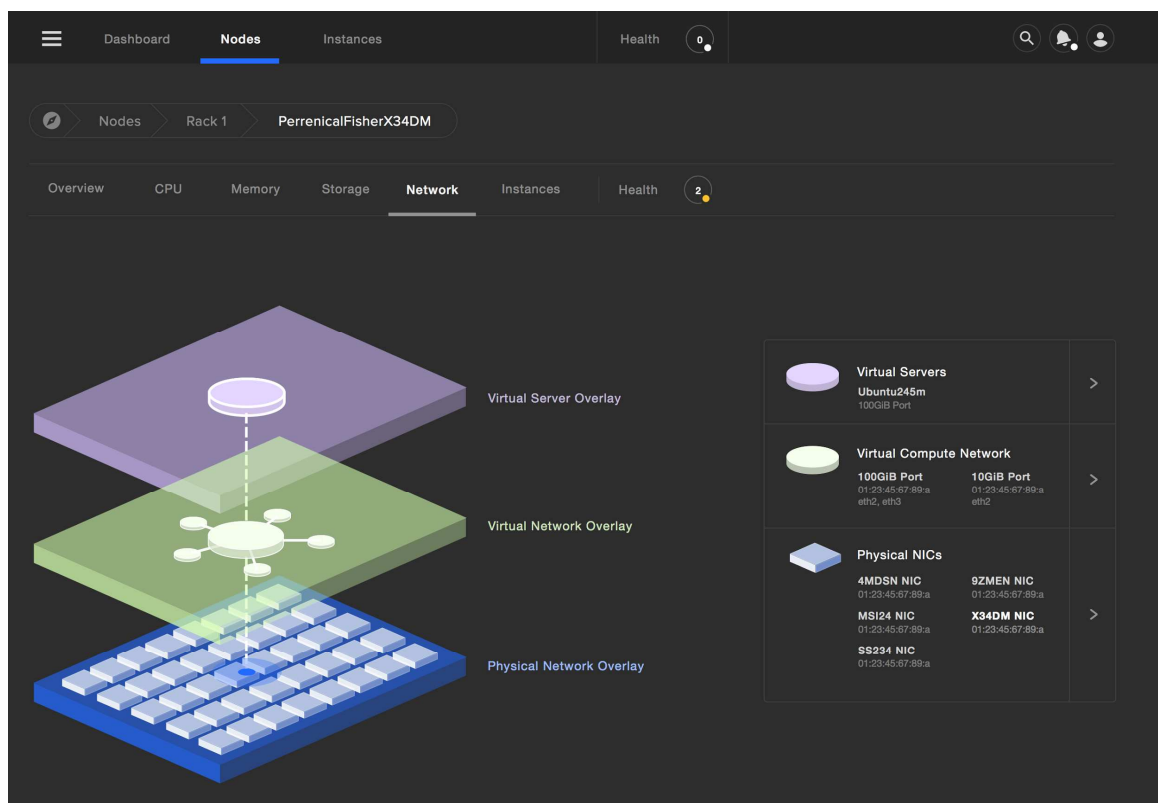


Figure 62: Mock-up diagram showing a UI that relates virtual to physical resources



In *Figure 62* a visualization mock-up of the administration panel is shown. In this visualisation, the physical racks have a number of rack servers that are numbered and can be probed for more information. Each rack then has a number of compute nodes that can be contained within a single physical server. The CPU load of each compute unit is then visualised, with standard traffic light colouring used to indicate low-utilisation (green), through to heavily loaded compute units (red). This gives an administrator a powerful tool to quickly identify if there are any servers that are struggling and to indicate issues that potentially need to be resolved either through computer assisted orchestration, or manual intervention.

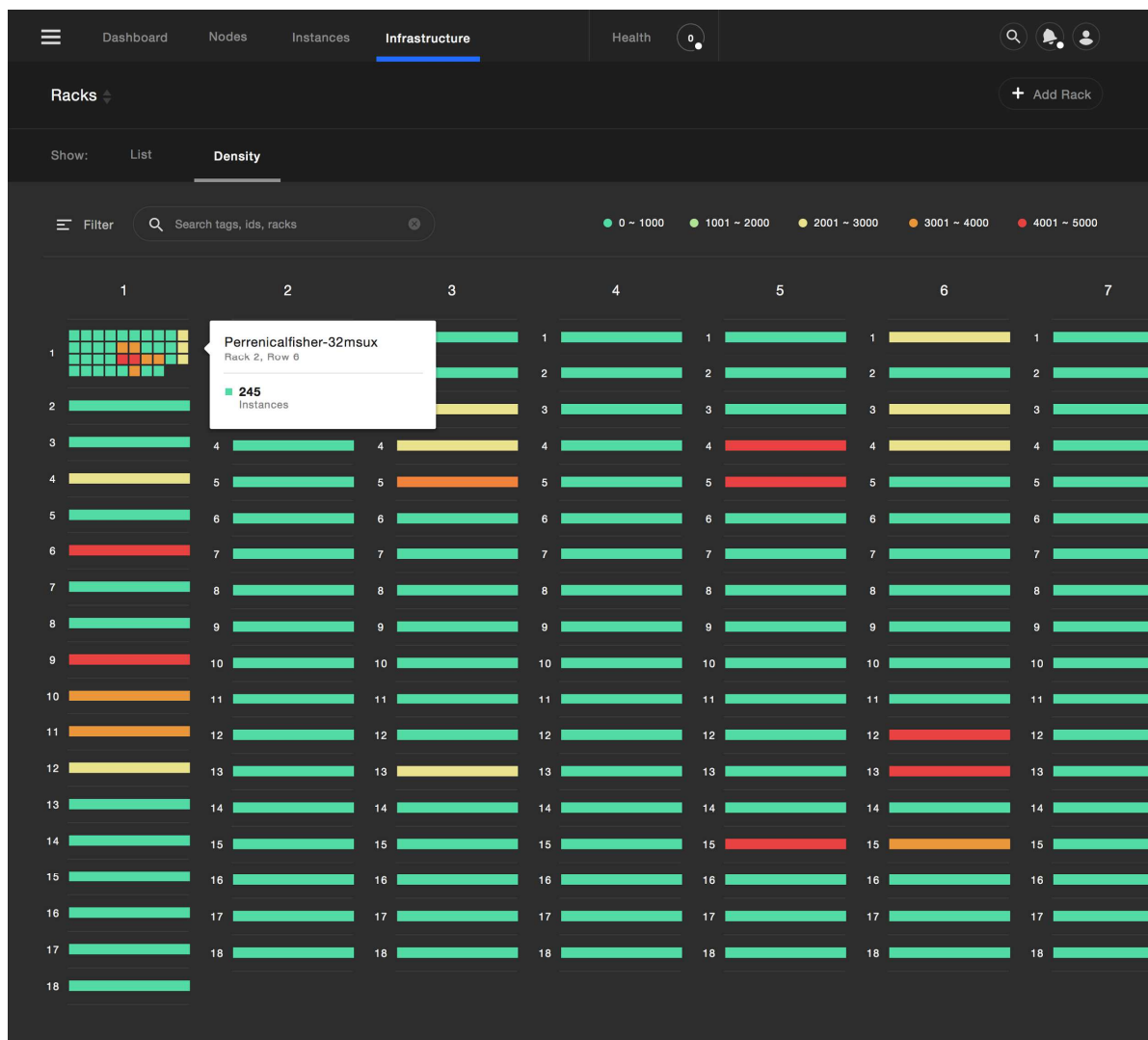


Figure 63: Mock-up diagram showing the rack utilization

Aside from the physical to virtual mapping and CPU load that have been shown in the previous Figures, it is also important to show the utilization of the storage resources. A mock-up Figure showing the utilization of the storage can be seen in Figure 63. Disks that are close to being full are shown in red with the less utilized disks being coloured in blue. All of the storage resources are



associated with particular racks and are separated accordingly. Also shown in the diagram is the notion of tiered storage performance levels. Given that certain virtualized workloads may have different I/O requirements it is important for the system to indicate different performance levels. This information can be captured in the data models in T4.1 and then analysed by the algorithms and heuristics in T5.1 to decide on where to place the workloads.

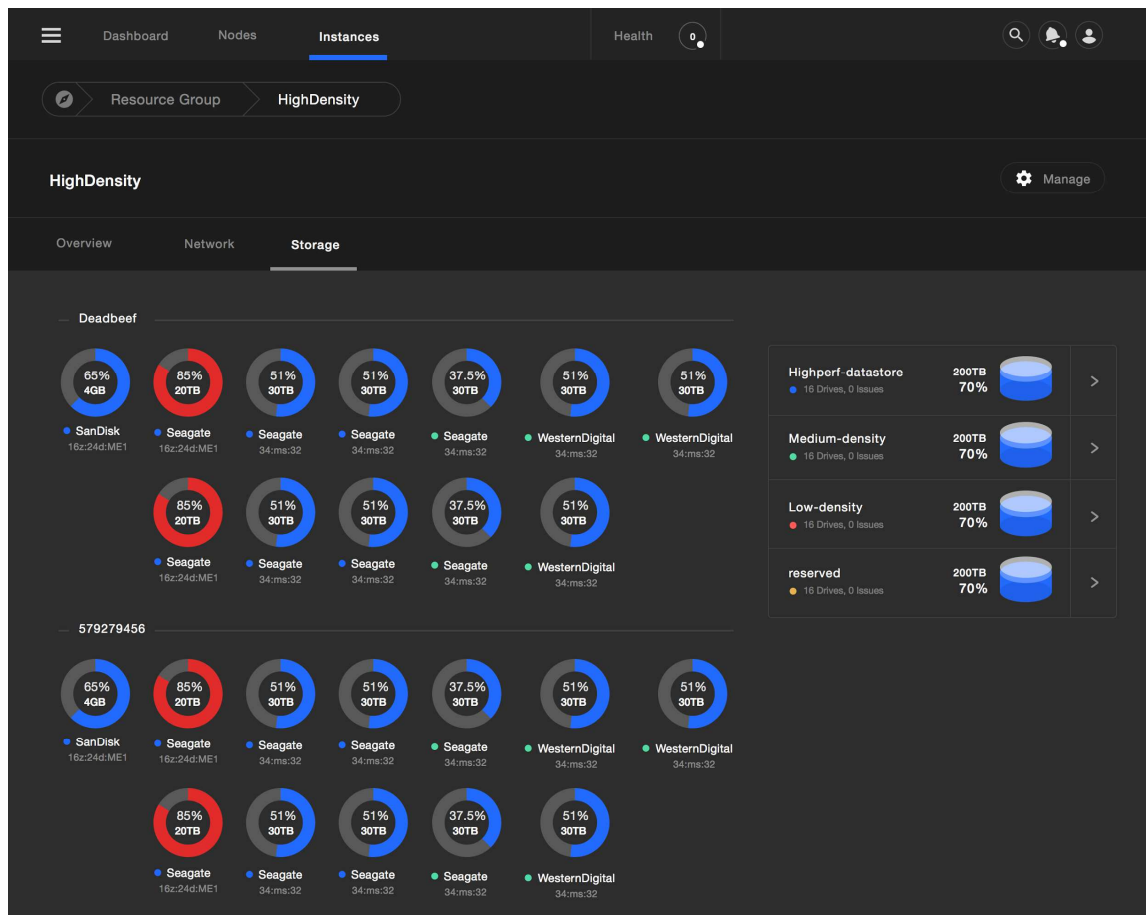


Figure 64: Mock-up diagram showing the storage utilization in the management UI

Given that SDN networking will also allow reconfiguration of a network, it is important for both the management platform and the orchestration system to be able to capture and possibly modify the network topology. This will allow maximization of the performance for a given set of workloads and configurations decided by the administrator. In Figure 64, a mock-up of the network mapping UI is shown. This can be used to visualize the current network topology and also could be used to capture modifications required of the network that could be then mapped to the network routers and hypervisors through tools such as OpenDaylight or others.

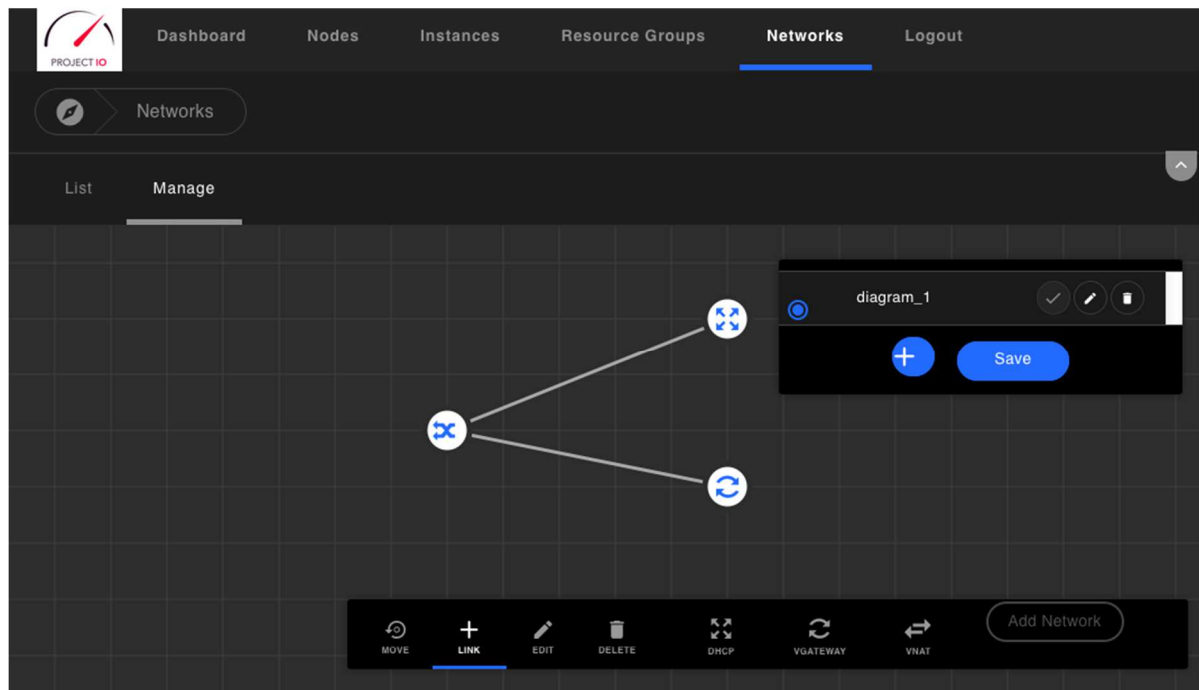


Figure 65: Mock-up showing the network planner UI



ANNEX B: Access-Agnostic SLA-Based Network Service Deployment – Task 6.2



B1 NEMO enhancements – implementation details

In order to integrate NFV expressions into NEMO a set of new features has been implemented.

Firstly, there is the need to reference the VNFD file as Universal Resource Identifier (URI). For this purpose, a new parameter has been defined in the NodeModel so that the lexical analyser will recognize a new token corresponding to the VNFD and the parser will add it as part of the NodeModel's parameters.

Secondly, NEMO needs to be aware of the virtual network interfaces defined in the VFND. In order for this to happen, a new network object has been defined, named ConnectionPoint. This new object has different functions: it can record the name of one network interface or it can be just the point of connection for a new NodeModel. Moreover, a new interesting feature has been integrated: we can think about VNFDs with N interfaces that could be set for the ConnectionPoint and we can choose which of these interfaces could be attached to the ConnectionPoint when it is instantiated. This is already possible by using the NodeModel's properties.

Thirdly, NEMO model provided a connection model to express the link between two network nodes. However, this feature has been extended and it now provides the link between either network nodes or connectionPoints. Because of this, it is possible to create the link between two simpler VNFDs.

The implementation of these basic features has enabled the creation of recursive VNFs. This means that it is possible to reuse NodeModels recursively to create complex NodeModel. Each NodeModels will check whether a node type matches the basic definitions (host, l2-group, l3-group, ext-group, chain-group, firewall and loadbalancer) or needs to be instantiated because it has a template definition (so achieving the recursiveness).

Moreover, it is possible to delete every Object created by NEMO. So that if a mistake occurs while writing the intent, it is possible to delete it and rewrite it again.

Finally, the processing of the NodeModels is needed in order to generate a complex YAML file (based on the baseline provided by OSM VNFDs) as outcome.



B2 Support for heterogeneous and nested execution environments

This section proposes extensions to the ETSI NFV ISG specification [4] to support nested VDUs using heterogeneous technologies.

7.1.6.2.2 [Vdu Information Element] Attributes

Clause 7.1.6.2.2 is modified as follows.

The following rows are added to Table 7.1.6.2.2-1.

Attribute	Qualifier	Cardinality	Content	Description
vduNestedDesc	O	0..1	Identifier (Reference to a VduNestedDesc)	This is a reference to the actual descriptor deployed in the VDU (e.g. a Click configuration). The reference can be relative to the root of the VNF Package or can be a URL.
vduNestedDescType	O	0..1	Enum	Identifies the type of descriptor file referred in the vduNestedDesc field (Click configuration, kubernetes template, etc.) and consequently the type of the Execution Environment running in the VDU. This field must be present if vduNestedDesc is present.
vduParent	O	0..1	Identifier (Reference to a Vduld)	This is a reference to the parent VDU which contains this VDU, thus this field is needed only for Nested VDUs. The referred Vduld must be defined in the current VNFD. Setting this field to the special value "None" specifies that this VDU is set to be deployed on bare metal.
vduParentMandatory	O	0..1	Boolean	This field specifies if the parent VDU must be present or if this VDU can be deployed also without its parent VDU. This



				field can be present only if the <i>vduParent</i> field is present and specified. The absence of this field is equivalent to setting its value to “false”.
--	--	--	--	--

Table 8: VDU attributes

We have extended the VDU information element contained in the VNF Descriptors to reference different types of Execution Environments that can be instantiated within a VDU and described by means of some descriptor. So far we considered kubernetes VDUs and Click router configurations VDUs. In particular, we have introduced four new attributes to the VDU information element: namely *vduNestedDescType*, *vduNestedDesc*, *vduParent* and *vduParentMandatory*.

- The *vduNestedDescType* attribute defines the type of RFB Execution Environment that is running in the VDU (in our case *kubernetes* or *click*).
- The *vduNestedDesc* attribute is an identifier. It provides a reference to the actual descriptor that is deployed in the REE running in the VDU (in our case a Click configuration file or a Kubernetes template). The proposed *vduNestedDesc* attribute uses the same approach of the *swImage* attribute, which provides a reference to the actual software image that is deployed in a “regular” VDU.
- The *vduParent* attribute is also an identifier. In case of a nested VDU, it references the parent VDU. An example of nested VDU could be a kubernetes pod (i.e. a group of containers) *K* inside a Virtual Machine *V*. In this case the VDU associated to *K* would have its *vduParent* attribute set to the *vduld* of the VDU associated to *V*. The VDU identifier must belong to the current VNFD, as the *vduld* is unique only in a VNFD scope (see clause 7.1.6.2.2).
- The *vduParentMandatory* attribute applies also to nested VDUs only and specifies if the VDU can be deployed also without its parent VDU. Referring to the above example, it specifies whether the pod *K* can be deployed also on bare metal (*vduParentMandatory* set to false) or if *K* must be deployed inside *V* (*vduParentMandatory* set to true).

7.1.6.4.2 [VduCpd] Attributes

Clause 7.1.6.4.2 is modified as follows.

The following rows are added to Table 7.1.6.4.2-1.



Attribute	Qualifier	Cardinality	Content	Description
InternalIfRef	O	0..1	String	Identifies the network interface of the VDU which corresponds to the VduCpd. This attribute allows to bind the VduCpd to a specific network interface of a multi-interface VDU.

Table 9: Additional VDU attributes

We have also introduced a new attribute, namely `internalIfRef`, to the `VduCpd` information element. The `VduCpd` information element is referenced by the VDU information element through the `intCpd` attribute. We add the attribute `internalIfRef` to the `VduCpd` element to create a correspondence between a `VduCpd` element and the network interface of a multi-interface VDU. For example, ClickOS instances internally name their interfaces as numbers starting at “0”. A ClickOS-based firewall with two interfaces would thus have an interface named “0” and an interface named “1”. The firewall could expect (in its Click configuration file) traffic from an external network A on port “0” and traffic from an internal network B on port “1”. The VDU corresponding to this ClickOS-based firewall would thus have two internal VDU connection points, one leading (through other connection points and virtual links) to the network A and one leading to network B. In this case the `VduCpd` element that would lead to network A would have its `internalIfRef` attribute set to “0”, while the `VduCpd` element that would lead to network B would have its `internalIfRef` attribute set to “1”.

B2.1 Notes on Kubernetes Nesting

When specifying Kubernetes VDUs, the `vduNestedDescType` attribute is set to the value “kubernetes” while the `vduNestedDesc` attribute is set to the identifier of a Kubernetes template.

Kubernetes templates can describe a pod, which in general includes several containers (sharing the same IP address). Thus in case the `vduNestedDescType` is set to “kubernetes”, the VDU represents a pod (not a single container).

Application containers such as Docker do not expose to users the concept of network interface. Thus in the NFV scenario we should consider a pod as single interface VNF/VDU. This means that for kubernetes VDUs we do not need to specify the `InternalIfRef` `VduCpd` attribute.

Kubernetes VDUs can use the `vduParent` attribute as specified above and as exemplified below.



Example1: pod to be deployed on VM

In this example we have a kubernetes pod to be deployed inside a VM/kubernetes worker node. We assume that the VDU associated to the VM has the *vduld* == “*vmid*” and that the pod is described by the kubernetes template *k8stemplate*. Then the values of the attributes of the VDU associated to the pod would be:

- *vduNestedDescType*: kubernetes
- *vduNestedDesc*: *k8stemplate*
- *vduParam*: *vmid*
- *vduParamMandatory*: true

Example 2: pod to be deployed on bare metal

In this example we have a kubernetes pod to be deployed directly on bare metal. We assume that the pod is described by the kubernetes template *k8stemplate*. In this case the values of the VDU associated to the pod would be:

- *vduNestedDescType*: kubernetes
- *vduNestedDesc*: *k8stemplate*
- *vduParam*: none
- *vduParamMandatory*: true

Example 3: pod should be deployed on VM, but can be deployed also on bare metal

In this example we have a kubernetes pod to be deployed on a VM/kubernetes worker node, but, if needed (e.g. due to resource unavailability), the container can be deployed directly on bare metal. We assume that the VDU associated to the VM has the *vduld* == “*vmid*” and that the pod is described by the kubernetes template *k8stemplate*. Then the values of the VDU associated to the pod would be:

- *vduNestedDescType*: kubernetes
- *vduNestedDesc*: *k8stemplate*
- *vduParam*: *vmid*
- *vduParamMandatory*: false



To actually deploy a kubernetes pod in its parent VM/worker node, the *nodeSelector* field of *PodSpec* can be used:

<https://kubernetes.io/docs/concepts/configuration/assign-pod-node/>

The appropriate *nodeSelector* value should be added during the translation/deployment phase to the kubernetes template referenced by the *vduNestedDesc* attribute.



B3 Core data-center placement

See attached: PAPER – 1: Optimizing NFV Chain Deployment Through Minimizing the Cost of Virtual Switching



B4 Mobile Edge Computing

B4.1 Placement

Placement of MEC Applications, meaning defining at which ME Hosts they will be deployed and run, must be determined by their requirements, especially by the maximum allowed delay to the UEs to be served. Considering that MEC addresses the need to have services provided as close as possible to users, it is expected that MEC Applications will be deployed at the closest ME Host to UEs to serve. However, the existing delay budget and resources availability may decide differently. Additionally, availability of certain ME Services, like RNIS (radio interface status) or LOC (location), will also constrain the MEC Applications placement.

In addition, depending on Applications' provided services and resources availability, Applications may not be immediately moved to all the ME Hosts, as part of the on-boarding procedure. Business and licensing conditions may also determine when and where to deploy and run MEC Applications.

ME Hosts will serve a number of eNBs. This number may vary from one ME Host per eNB, possibly running at the eNB, to several eNB being served by a single ME Host. Considering C-RAN (Cloud-RAN) deployments, it makes all sense to consider the deployment of ME Hosts, sharing the same cloud infrastructure with RAN centralized components.

ME Orchestration (MEO) must have a topological view of the entire MEC System, in order to determine where to deploy ME Applications to serve specific geographical areas. Thus, a mapping between eNB and ME Hosts is required. This information was referred before as stored in the MEC Hosts Inventory repository.

In addition, ME Application descriptors must include parameters to help MEO to decide where to deploy each ME Application. This must be complemented by Operators' rules and established business relationships, and translated into MEO understandable policies.

MEC Applications' placement will be decided based on Applications and Operators' requirements and constraints, and not UEs, with the exception of instantiations requested by entities running at UEs (e.g. Applications counterparts running at UEs).

Thus, in general, the following placement factors may apply, determining how many MEC Application instances are required and where to deploy those instances:

- **Delay**
MEC Application Descriptors shall indicate desirable and maximum admissible delay towards UEs to be served.
- **Geographical scope**
Some ME Applications may provide localized services, like Augmented Reality for a specific building or street. The geographical scope will, most likely, be provided in such a way (e.g.



reference to a monument or specifying a geographical area) that will need to be mapped to eNBs, and from that to ME Hosts, by the MEO.

- **Required resources**

Besides specifying computational and networking resources to run, MEC Applications may require specific hardware to run, like video processing.

- **Available resources**

Considering that all on-boarded ME Applications cannot be deployed and run at all ME Hosts, their instantiation may depend on the availability and establishment of priorities to access ME Hosts' resources.

- **Required MEC Services**

MEC Applications may need to access local MEC Services to run, for instance, with RNIS or LOC.

- **Licensing and agreements with service provider**

Business agreements with ME Application providers may also determine where and how many instances to deploy. The limitations may be in any of the parties (Operator only allowed or allowing to simultaneously run a certain number of Application instances).

- **UEs' location**

This is a specific scenario that will apply to Applications' instantiation requests generated by the UEs (via the "User App LCM Proxy", e.g. for Application computation off-loading). Besides any of the abovementioned factors, this specific one will be determinant in the placement of the requested MEC Application instance.

MEO shall handle the identification and evaluation of the parameters that apply. This will happen after MEC Applications on-boarding and whenever the MEC System topology change, for instance, by the addition or removal of MEC Hosts or the reorganization of the mapping between eNB and ME Hosts. It will also happen whenever a UE or the OSS requests the instantiation of some MEC Application. No algorithm or parameters evaluation process is proposed for the moment.

B4.2 Service Migration & Mobility

B4.2.1 Mobility in MEC scope

In MEC context, service migration need, or relocation, may happen as a result of UEs' mobility, and applies to MEC Applications. While moving, most likely UEs will attach to different eNB. Depending on the network topology, the same or different MEC Hosts may serve those eNB. While moving between eNB served by the same MEC Host, no relocation shall happen, as UE's mobility will not be noticed by the MEC System. For the other situations, the frequency and the type of relocation to be executed depend on the Application type and mode of operation:



- **Generic Applications** (not tied to any UE in particular)
 - **Stateless applications:** no need for any relocation action
 - Application instance is already working at the destination ME Hosts: Service is provided at the edge
 - Application instance is not working at the destination ME Hosts: Service is not provided at the edge
 - **Stateful applications:** state may be is need
 - Scenario detailed below
- **UE specific Applications**
 - Service needs to follow the UE, independently from being stateless or stateful, in order to keep proximity, according to the specified requirements

(Generic) MEC Applications will be deployed at certain ME Hosts, as described above. They will have traffic rules associated to them, instructing the ME Host Data Plane on how to identify and handle traffic of interest, to be delivered to each MEC Application. These rules will be defined, in general, based in destination FQDN or IP/Ports. It means that, in general, Applications will be deployed and make their services available in anticipation and independently of the UE which will request them. Thus, MEC Applications' provided services are accessible at the specific ME Hosts on which the MEC System decided to deploy those MEC Applications. Therefore, UE's mobility shall not trigger, in general, MEC Application relocations, except for the ones instantiated under UE's request. However, application state created and associated to the UEs may need to be relocated.

MEC Applications mobility is mentioned in current ETSI MEC ISG documents and its discussion triggered the creation of an Work Item (WI), to be developed till end of MEC Phase 1 (end of 2016). For instance, [ETSI GS MEC 002, Mobile-Edge Computing (MEC); Technical Requirements] states that: *"Other mobile edge applications, notably in the category 'consumer-oriented services', are specifically related to the user activity. Either the whole application is specific to the user, or at least it needs to maintain some application-specific user-related information that needs to be provided to the instance of that application running on another mobile edge host.*

As a consequence of UE mobility, the mobile edge system needs to support the following:

- *continuity of the service,*
 - *mobility of applications (VM), and*
 - *mobility of application-specific user-related information.*
- "*

The same document identifies three mobility requirements:

1. *"[Mobility-01] The mobile edge system shall be able to maintain connectivity between a UE and an application instance when the UE performs a handover to another cell associated with the same mobile edge host."*



2. “[Mobility-02] The mobile edge system shall be able to maintain connectivity between a UE and an application instance when the UE performs a handover to another cell not associated with the same mobile edge host.”
3. “[Mobility-03] The mobile edge platform may use available radio network information to optimize the mobility procedures required to support service continuity.”

As a summary, the MEC System shall guarantee service continuity, by application or “application-specific user-related information” mobility, or maintaining connectivity to the ME Host where the required MEC Application is running. This last option will have as a limitation the maximum delay allowed by the Application.

Whenever relocation is needed, the following types may apply.

B4.2.2 MEC relocation types

Considering the stated requirements, the following application relocations types can be envisioned:

- **Application state relocation**

Application state associated to served UEs is transferred between different MEC Applications instances, located at the old and new MEC Hosts, involved in the UE’s mobility.

Even if it is possible to have a MEC System assistance (aspect not yet defined at ETSI ISG MEC), MEC Applications will most likely manage themselves the state transfer via communication between the involved instances.

- **Application instance live relocation**

Applications running at a ME Host, as VDU, and providing services to specific UEs, are live migrated from old to the new ME Hosts (between different NFVI), serving the eNBs where the UEs are now connected. Mechanisms for a complete VDU relocation, shall resort to virtualized infrastructure capabilities.

MEC Management and Orchestration needs to participate in the process, coordinating it.

- **Application instance emulated relocation**

Application instances relocation can be emulated by the creation of a new instance and deletion of old instance, at originating and target ME Hosts

If there is no need for Application migration or state transfer but the provided services are needed at the new MEC Host, relocation can be emulated by starting a new Application instance at the new ME Host, while stopping the previous instance at the originating ME Host. For the UE, this operation shall be transparent and the service works in a continuous way.

MEC Management and Orchestration needs to deal with this process.

- **No relocation, keeping service anchor**



In order to keep service continuity and due to required Application state maintenance specificities, the solution may consist in keeping the provided service anchored at the original MEC Application instance, running at the first ME Host the service started being provided.

This implies associated traffic rerouting between ME Hosts.

- **No relocation** at all

At target ME Host, either the MEC Application is not running (e.g. because of the defined Application geographical scope) or an already existing instance is able to provide the same service, without need for any communication with previous instance (e.g. stateless MEC Application). There is no state or application relocation.

B4.2.3 UE's mobility detection

In all relocation scenarios, and without considering any interaction with EPC control plane (e.g. S1-MME), UE's mobility is only detected and the new serving MEC Hosts is only known once the UE attaches to the new eNB and its traffic is identified. Even if the UE is aware of eNB change, it is not aware of the possible MEC Host change. This way, any needed relocations can only be triggered after UE's arrival at the new MEC Host.

UE's mobility detection can be done at two levels:

1. **By the ME Hosts**, by observing traffic send to/from a new IP/TEID (Data Plane level)
As a result, the detecting ME Host may proactively query neighbouring ME Hosts about source IP (which ME Host was previously handling that IP).
The previous ME Host handling the UE, may query/notify all running Applications or only the ones that stated the existence of state associated to that IP, about relocation needs.
2. **By the MEC Applications** themselves, when applicable traffic rules deliver traffic to them
As a result, and if required (stateful Applications), will require the hosting ME Host to provide its mobility services.

In addition, UE's mobility may be detected by the UE itself. If eNB change is notified to local Application counterpart, this one may notify the MEC System, via the "LCM Proxy", about the possible need for relocation actions.

B4.2.4 Relocation need detection

In parallel to that, the need for relocation (instance or state) must be identified. This can be known, and also considering previous scenarios:

1. Previously by the ME Host as part of the MEC Application description and communicated to the ME Host at instantiation time



2. At Application execution time (state is created and exists, associated to certain UEs) and communicated to the ME Hosts via an appropriate API:
 - a. Inform the MEC System, what IP addresses need to be monitored regarding mobility aspects (proactive)
 - b. Whenever a new IP is detected by an Application instance, it may ask the hosting ME Host to trigger the required relocation actions (reactive)
3. Unknown by the ME Host

B4.2.5 Proposed processes

Both aspects, UE's mobility detection and need for relocation, must be considered together. One approach consists in delegating to MEC Applications the identification of UE's mobility and the identification of relocation needs. Based on that, the following proposals are made.

A. Proposed process for Generic MEC Applications:

1. UE handovers to a new eNB, served by a new MEC Host
2. Configured traffic rules will extract and deliver UE's traffic to applicable MEC Applications
3. Upon detecting traffic from a new IP, stateful MEC Applications will query the hosting ME Host if Application's state exists in another ME Host, expressing relocation actions
4. ME Host queries neighbouring ME Hosts about state related to that IP and MEC Application (IP, AppID)
5. If existing, previous ME Host, notifies, based on AppID, local Application instance about relocation needs
6. As a consequence, that MEC Application instance requests the ME Host to:
 - a. Establish a tunnel to the new MEC Host for traffic redirection or
 - b. Provide new MEC Application IP address, for managing state relocation
7. Depending on the previous:
 - a. Traffic rules at the new ME Hosts are changed accordingly
 - b. Application instances exchange state and service for that UE continues at the new Application instance

A similar behaviour may be achieved but with Applications communicating with the corresponding Manager, running at the respective ME Platform Manager. The Application Manager may work as a central point for all instances, coordinating with the MEC System entities, relocation needs and actions. No details for this option are provided.

B. Proposed process for UE's specific MEC Applications:

1. UE handovers to a new eNB, served by a new MEC Host
2. ME Host Data Plane detects a new IP/TEID



3. ME Host queries neighbouring ME Hosts about specific MEC Applications running for that UE related to that IP
4. If existing, previous ME Host, notifies, the local Application instance about UE's mobility
5. As a consequence that MEC Application instance requests the ME Host to:
 - a. Establish a tunnel to the new MEC Host for traffic redirection or
 - b. Proceed with Application instance relocation to the new ME Host

B4.2.6 Mobility API

Besides the proposed relocation mechanisms, other solutions are possible. The existence of an API for Applications to communicate with ME Hosts is needed and additional options may be provided, giving place to the definition of other solutions. It is therefore proposed the following API features:

1. Apps to notify ME Host about UEs (IP addresses) to be monitored
2. Apps to request ME Host about UE's originating ME Hosts
3. Apps to request ME Host about originating App IP address
4. Apps to request ME Host about another hosting ME Host IP address
5. Apps to request ME Host to store information at local persistent storage
6. Apps to request ME Host to move/copy stored information to another ME Host
7. Apps to request ME Hosts LCM operations (Stop, Create)
8. ME Hosts to notify Apps about UE's mobility (IP address)
9. ME Host to notify an App about the arrival of stored information (AppID)
10. ME Host to query other ME Hosts about UE handling



B5 Optimal scaling and load balancing based on MDP models

B5.1 Introduction

We address the problem in which the enterprise has to find dynamic policy of VM deployment, displacement and scheduling of incoming VNF tasks, as a function of the set of costs and the number of VNF instances currently being served in the already active VMs.

An enterprise, which decides to virtualise any of their network services needs first to deploy ("build") the corresponding service, in order to be able to run VNF instances.

The deployment process involves having leased a VM and loading on it the corresponding software. Once deployed, the enterprise has to pay per time of usage for the leased VMs, regardless of the load.

The process of deployment can take time, and an additional deployment cost. Hence, having idle resources is undesired. On the other hand, the delay involved with the deployment or lack of space for the new tasks can cause some VNF instances to be rejected from running, thus inflicting a profit loss to the enterprise. Note that in some cases, the displacement ("destroy") operation, i.e., the process of releasing VMs can also incur a cost.

An additional basic demand is to facilitate scaling. For simplicity, consider an enterprise which needs to run networking tasks of only one type. Hence, we assume all VMs are similar and able to run similar VNF tasks (e.g., flows a firewall handles). The total number of VMs allocated is dynamically increased (via a scale out operation) or decreased (scale in operation) according to the demand from this network service. In scale out/in operations, we increase/decrease (i.e., deploy/displace) the number of VMs that are allocated, respectively. Note that we do not consider the scale up and down operation, which are less common in NFV use cases. (In scale up/down operation we can, for example, add/remove CPU cores to a given VM).

Clearly, increasing the number of VMs would disperse the total load and improve the performance of each VM, yet would imply having leased more costly resources. Accordingly, our scaling decision should minimize the number of leased VMs, while keeping with the application required SLA.

Hence, the decision whether to scale out or in remains an important problem that needs to be addressed, especially in dynamic scenarios.

In conjunction with the scaling decisions, we are also facing a load balancing challenge, steering the traffic flows to the different VMs and balancing the load between them. In this study, we tackle both the scaling decision and the load balancing strategy as a single problem.

Going back to the simple single-VNF enterprise scenario, the load balancing will merely amount to having equally loaded VMs. Hence the trade-off in this case means the average load on a VM against



the number of deployed VMs. Yet, having in mind deployment time and cost, this trade-off still represents a significant challenge as the optimal policy derivation is not straightforward.

To this end, we model the problem by queuing system with a dynamic (but limited) number of queues; each queue stands for a VM running VNF instances. We assume that VNF tasks arrive with constant average rate. Upon each arrival event, the Decision Maker (DM) decides to which VM the arriving task should be scheduled, or whether it should be rejected. The VM deployment decisions are made at arrival times, as long as the limit of active VMs is not reached.

At departure from a queue (i.e., running of VNF task ends) which has been left empty, DM decides whether to keep that queue alive or to destroy it.

In order to reflect the load at busy VMs we introduce a delay cost which (not necessarily linearly) increases with the load.

The maximal number of running task on a single VM is limited by borderline number which indicates a performance fall beyond the SLA demands and, hence, should never be exceeded.

Deployment and displacement operations of VM consume additional fixed costs. Once a VM was deployed, the enterprise is charged with fixed per-unit of time reservation and maintenance cost. We term this keep-alive or the holding cost.

The DM aims to find a policy which will maximize the total income in the long run.

While the set of costs described above implies no trivial policy could exist, some of the parameters have contradicting impacts which may dictate certain properties of the policy.

For example, in the case where the delay cost function sharply increases with a load, the DM will attempt to deploy as many VMs as possible.

On the other hand, in the case where keep-alive cost is comparatively high, the DM may prioritize minimization of the active VMs number.

A distinctive impact has a VM deployment time, which we separately explore.

In this work, we treat the dynamic VM deployment and VNF tasks scheduling problem by introducing a control model based on MDP formulation. The solution of the MDP provides the optimal policy.

Once applied, the policy potentially reveals the average number of active VMs and average number of tasks in the task queue of each VM.

This allows the enterprise to assess the demands and to plan ahead.

Moreover, the solution to the MDP is expressed by value function which indicates what are the costs and revenues the enterprise will receive in the long run, for a given scenario along with its set of parameters.

Complete details of this work are available at: [18]. In the following we present some of the numerical results of our work.



B5.1.1 Numerical results

In the following, we present simulation results which both show additional threshold properties of optimal policies, and provide a comprehensive study of the value functions and the corresponding policies. We explored a system with five identical queues.

Figure 66 demonstrates the impact of the keep-alive queue cost. The simulation was run with negligible delay cost, rejecting fine equal to 10, $\lambda = 4$ (VNF tasks arrival rate) and $\mu_i = 1, \forall i$ (servers total processing rate). Buffer limit was 4 tasks. One sees that there is a small region of keep-alive queue cost, where the number of active queues declines. The trade-off in this simulation is the accumulatively paid fine against the accumulatively paid keep-alive queue cost. As long as the queues are identical, there is no preference which queue should be kept idle. That is why the number of active queues declines to zero within a very small interval of keep-alive queue cost. Hence, once it is more affordable to pay the fines by rejecting all incoming tasks rather than maintaining the VMs (i.e., the queues) the queues stay idle at all times.

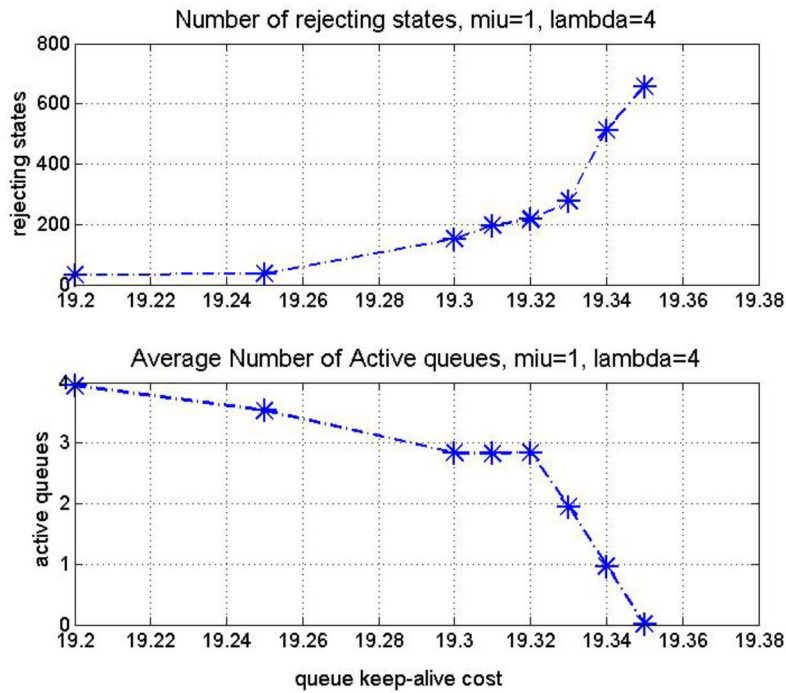


Figure 66: Impact of the allocated VNF cost

Figure 67 shows simulation of the delay cost h , while the keep-alive cost was fixed equal to 1, the buffer size of each queue was 6. The arrival intensity was 4.75 and $\mu_i = 1, \forall i$. Rejecting fine was equal to 10. The delay constants were $\eta_1 = 1, \eta_2 = 1.8, \eta_3 = 2.5, \eta_4 = 3.5, \eta_5 = 4.5, \eta_6 = 5.5$. By the cost model, one expects that the tasks will be balanced in the queues. Observe that in this simulation, the keep-alive cost was low enough to allow that. Indeed, all queues were active while



the total average number of tasks declined with the delay cost. Observe that in this simulation, the interval of the varying cost was significantly larger. This stems primarily from the fact that η_1 is small. Note that the lower graph in both simulations shows the number of rejecting states, out of the states-space.

Additional thresholds can be seen in "build" and "destroy" actions. For example, if The building cost β and/or the destroy cost ψ are high if compared to keep-alive cost, the optimal policy acts to leave all queues active, even if empty.

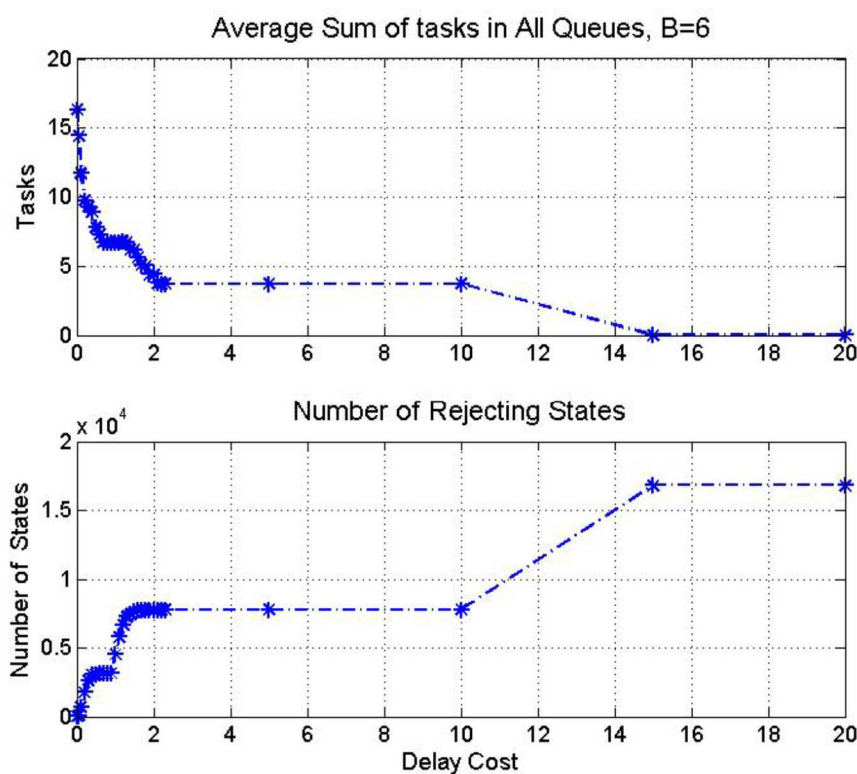


Figure 67: Impact of delay cost

To conclude, the set of system parameters will determine if the optimal policy will prioritize balanced and mostly active queues or a small number of active and comparatively busy queues. The values of β and ψ will determine if keeping alive the empty queues is economically reasonable.



B5.2 Machine learning techniques for the LCM for containerized workload

B5.2.1 Introduction

To allow multi-tenancy of containerized applications, typically, each tenant/application is allocated VMs as the underlay infrastructure. Then, the containerized application scales out and in within that underlay. Accordingly, we are now facing two scaling processes, that is, (i) of the underlay VMs, and (ii) of the containerized applications.

For the first process, namely the VM Scaling, we have our MDP model (see section B5.2) that can be extended to tackle containerized workload. Such extension is pending investigation in the third year. In this work, we focus on the second process of the scaling of the containers. Here we devise through machine learning techniques, a mechanism that automatically scales the containerized application based on infrastructure metrics. We further demonstrate that our mechanism dramatically outperforms Kubernetes. Specifically, we demonstrated that a containerized video streaming application suffers from high delays (due to lack of resources) while being managed by Kubernetes. On the other hand, our mechanism manages to scale the application in a timely manner while providing the appropriate resources to obtain the required application QoS (i.e., packet latency).

B5.2.2 Kubernetes scaling mechanism

Typically, the LCM operation of containerized applications is handled by the Containers Orchestration Engine (COE). Since Kubernetes is the most dominant COE of today, in this work we adopted it as our COE of choice. Indeed, Kubernetes manages the scaling operation. However, in this work, we demonstrated that Kubernetes suffers from severe drawbacks. Specifically, Kubernetes v1.57 triggers scaling only based on CPU utilization or based on application metrics (via APIs). Indeed, this was handled in Kubernetes v1.6, where other infrastructure metrics may trigger scaling. Yet a function that considers all infrastructure metrics is not defined and it is not clear how to combine those metrics to trigger scaling operation.

Accordingly, in the project's second year we tackle this issue by applying machine learning techniques to control the scaling decisions.

B5.2.3 Machine learning based scaling

Our goal is to devise a machine learning mechanism that receives as an input samples of the resource utilizations by each container and predicts application performance.

As a learning sequence, we assume that our system receives indications on the application performance. Based on this learning metric, we aim at devising a mechanism that learns the optimal



scaling policy based on infrastructure metrics that minimize the consumed resources (minimize the number of deployed containers) while maintaining the application's required performance.

B5.2.4 Offline implementation – video streaming application

Aiming at devising the optimal online machine learning scheme, we first analyse and solve this problem in an offline scenario.

To that end, we consider a containerized video streaming server. Specifically, we containerized this server especially for this work (available at Docker Hub under ruvenmil/mycont2). Next, we ran this server under different deployment configurations, i.e., with multiple number of clients as well as with multiple numbers of deployed pods (Kubernet's minimal deployment unit). For each configuration, we recorded the packet delay. Note that for video streaming the packet delay constitutes a representative indication for the video quality. Our setup topology is given in Figure 68.

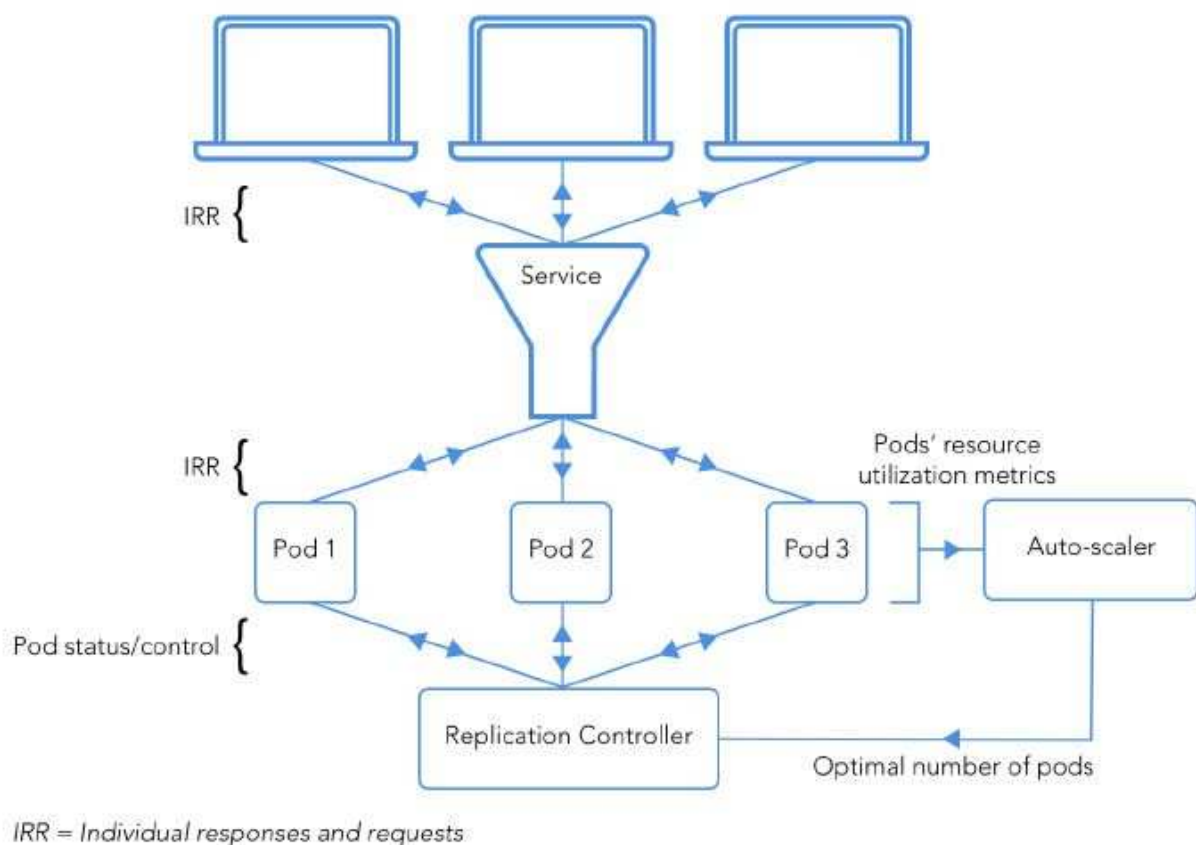


Figure 68: setup topology

In addition to the packet delays, each record also includes the infrastructure utilization, namely, CPU, memory and bandwidth consumption.



With all data at hand, we employed several prediction mechanisms, including: random forest and gradient boosting, and obtained a predictor for the video quality based on the infrastructure resource consumption. Next, we triggered the scaling operation based on this predictive function. Utilizing that function obtained a much better video quality and initiated scaling just on time.



B5.2.5 Implementation results

In the following we compare our auto-scale algorithm with the default scheme of kubernetes.

Packet error measurement:

We can see in Figure 69 that the suggested algorithm has better performance and less error burst (depicted in dark blue).

Kubernetes:

suggested algorithm

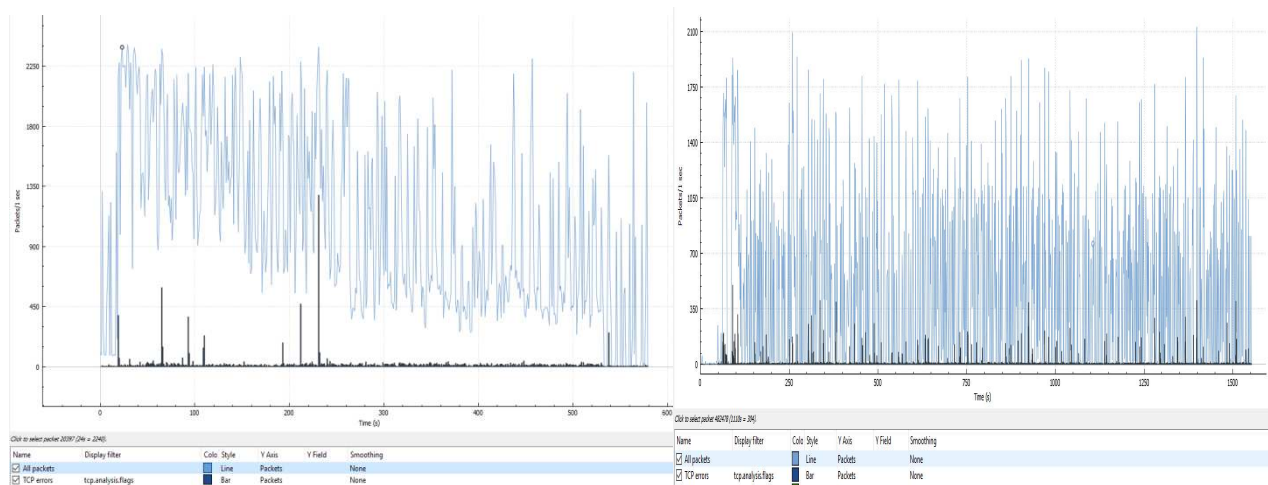


Figure 69: packet errors

Throughput measurement:

Figure 70 depicts the throughput measurement results. One can see that the achieved throughput is higher with our suggested algorithm compared to with the of the shelf Kubernetes mechanism.

Kubernetes:

suggested algorithm

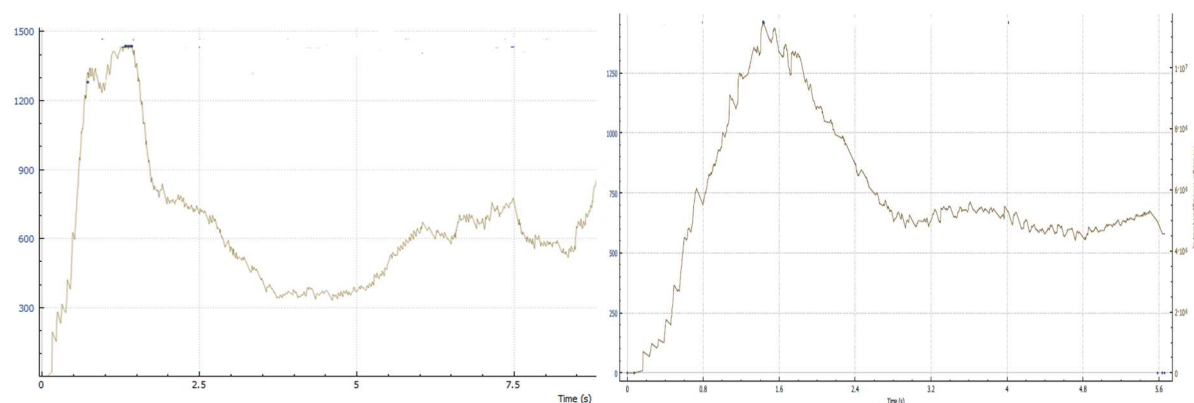


Figure 70: Throughput measurements



B6 Load Balancing as a Service

B6.1 Load Balancing principles

Efficient load balancing is an area of paramount importance for identifying possible flaws and limitations of the SUPERFLUIDITY architectural components, most of which are currently under heavy development, undergoing continuous modifications to improve performance.

The actual definition of load balancing, as the distribution of network or application traffic across a cluster of servers, consequently leading to improved responsiveness and increased service availability, is provided in Deliverable D5.3. In addition, Deliverable D2.1 identified the load balancing-dependent use cases, where specific performance must be established to fulfil the pre-defined scalability and high availability requirements, while the specific requirements of Load Balancer as a functional block were further analyzed in Deliverable D2.2. Last but not least, an additional analysis of certain load balancing approaches can also be found in **Annex A, section A3.5** of this deliverable. With load balancing references being omnipresent in the majority of technical deliverables, one may identify the significance of this functionality for the SUPERFLUIDITY platform as a whole.

B6.2 HA Proxy and LBaaS in OpenStack

The project's virtual infrastructure manager (VIM) of choice, OpenStack, offers certain open-source options for implementing the infrastructure load balancing capability, originally through HAProxy, a single-threaded, event-driven, non-blocking engine combining a very fast I/O layer with a priority-based scheduler, currently integrated from an upstream open-source project. OpenStack fully supports HAProxy deployment in its controller nodes where each instance of the software configures its front end to accept connections only to the virtual IP (VIP) address, while the backend is a list of all available IP addresses that may need load balancing of their ingress traffic. However, the most effective load balancing service of OpenStack, currently integrated inside Neutron is no other than Load Balancing as a Service (LBaaS), which allows both proprietary and open-source load balancing technologies to handle the excessive traffic load. As stated in [18], LBaaS builds on top of HAProxy by leveraging agents that control HAProxy configuration and manage the HAProxy daemon, is therefore somehow considered as an enabling module that introduces additional functionality to the core load balancing mechanism of OpenStack. OpenStack is also introducing carrier-grade load balancing backend mechanism through Octavia [19].

B6.3 Citrix Netscaler ADC and MAS

The increased demands of contemporary networking environments in terms of traffic, necessitate the evaluation of the proposed SUPERFLUIDITY architecture paired with commercial, carrier-grade



load balancing options. Citrix NetScaler ADC [20] is an application delivery controller that provides flexible delivery services for traditional, containerized and microservice applications from any cloud or private datacenter, and was evaluated as part of SUPERFLUIDITY. As covered in **Annex A, section A3.5** of this deliverable, NetScaler ADC can be integrated into the NFV architecture, working in parallel with all existing MANO entities, such as VIM and the SDN Controller.

NetScaler ADC is operated through a dedicated Element Manager, namely the NetScaler Management and Analytics System (MAS) [21] NetScaler MAS integrates using standard APIs with OpenStack, translating necessary messages to the RESTful APIs supported by NetScaler ADC. It facilitates administrators to monitor, automate and manage network services for scale-out application architectures with ease, and provides application-level integration with external orchestration systems.

Integrating MAS with OpenStack

The overall evaluation process conducted as part of SUPERFLUIDITY, involved a direct comparison of the capabilities of open-source load balancing options already integrated in vanilla versions of OpenStack (HAProxy) against NetScaler ADC. This process also required deploying and validating NetScaler ADC and MAS in a dedicated NFV Lab, thus verifying that certain enhancements introduced in these products as well as the NetScaler LBaaS driver [26] [27] are working properly.

As stated in previous paragraphs, for being able to use NetScaler ADC instead of the integrated load-balancing solutions of OpenStack, certain modifications are needed in Neutron LBaaS module. In particular, Citrix engineers developed an LBaaS plugin which can be deployed in Neutron and implements all the LBaaS driver CRUD APIs for operating on OpenStack VIPs, Pools, Pool Members and Health Monitor entities. The integration consists of a driver class configured in the Neutron config file (neutron.conf), and the accompanying unit tests, while its abstract functionality is shown in Figure 71.

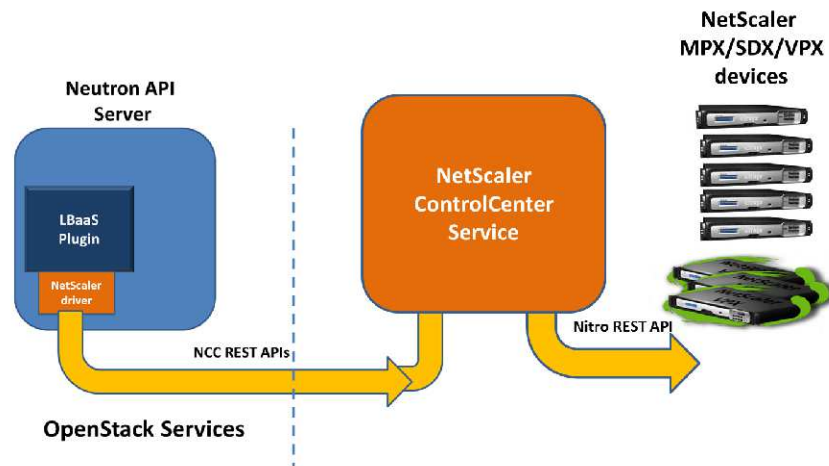


Figure 71: NetScaler LBaaS Integration

The NetScaler LBaaS integration consists of a driver class that implements the Neutron LBaaS driver which calls the NetScaler Control Center (NCC) service using NCC REST APIs. NCC is a separate service that runs outside of OpenStack infrastructure, and is deployed as a "virtual appliance" on supported hypervisor platforms (KVM/ESX/XenServer) for ease of setup. Since Release 11.1, NCC is integrated in NetScaler MAS, which is now in charge with all NetScaler resource and tenancy management tasks, as well as NetScaler device configuration, allowing the driver component in OpenStack to remain lightweight, simpler to maintain and easier to evolve going forward as the Neutron service evolves. During SUPERFLUIDITY validation process, a full-scale NetScaler MAS node was deployed and integrated with the existing OpenStack Platform, as follows.

OpenStack Neutron LBaaS plugin includes a NetScaler driver that enables OpenStack to communicate with the NetScaler MAS. OpenStack uses this driver to forward any load balancing configuration done through LBaaS APIs, to the NetScaler MAS, which creates the load balancer configuration on the desired NetScaler instances. OpenStack also uses the driver to call NetScaler MAS at regular intervals to retrieve the status of different entities (such as VIPs and Pools) of all load balancing configurations from the NetScaler ADCs. NetScaler driver software for OpenStack platform is bundled along with the NetScaler MAS. To download and install the drivers, it is necessary to first install NetScaler MAS and launch the application.

B6.4 NetScaler MAS Installation

For consistency reasons NetScaler MAS was installed on a Linux KVM server, after verifying that all hardware virtualization extensions and *virsh*, a command line tool for managing virtual machines, were available. As described in [35], after obtaining the necessary image files the installation process includes, navigating to the folder where the compressed file is saved, using the tar command to untar



the NetScaler MAS image, verify that a domain disk image (.qcow2 file format), a domain XML file and the necessary checksum file were present, editing the XML file for specifying the necessary networking attributes, using *virsh* to define the VM attributes through the recently added XML file and initiating NetScaler MAS by entering the following command:

```
virsh start [<DomainName> | <DomainUUID>]
```

Certain configuration steps for the NetScaler MAS were also needed, after the previous process is concluded and the service became available. In particular it was necessary to:

- Login to the NetScaler MAS node
- Type shell > networkconfig to configure the management IP address
- Complete the initial network configuration by adding information regarding Netmask and GW IP
- Execute the deployment script by typing the following command in the shell prompt # deployment_type.py
- Restart the server and login to the GUI

B6.5 NetScaler Driver Software Installation

It is possible to install NetScaler driver on OpenStack via NetScaler MAS GUI. After logging in, click Downloads and download the latest NetScaler bundle .tar file to a temporary directory of the OpenStack Controller. The bundle includes LBaaS V2 drivers for Openstack Liberty/Mitaka/Newton releases along with the Heat plug-in. Extract the files from the NetScaler driver tar, navigate in to the OpenStack <Release Name> folder and execute the following command to install the driver and specify the NetScaler MAS IP address:

```
./install.sh --ip=<NetScaler_MAS_IP> --password=<password> --protocol=<protocol> --neutron-lbaas-path <neutron-lbaas-directory-path>
```

B6.6 Registering OpenStack with NetScaler MAS

OpenStack information needs to be registered on the NetScaler MAS. The process includes specifying the OpenStack controller IP address and cloud administrative user credentials, and also the OpenStack NetScaler driver user credentials. It is also possible to later specify the same login credentials in the NetScaler_driver section of the Neutron configuration file (neutron.conf) so that NetScaler driver in OpenStack can connect to NetScaler MAS during LB configurations.

After OpenStack and NetScaler MAS are registered with each other, both can talk to each other. Also, OpenStack users can use their existing credentials in OpenStack to log on to the NetScaler MAS user interface to check how their LB configurations are performing in NetScaler [35].



B6.7 Adding OpenStack Tenants in NetScaler MAS

Provided that OpenStack and NetScaler MAS are now interconnected through the previous registration process, it is possible to create a Tenant in OpenStack using the NetScaler MAS. Simply

- Navigate to **Orchestration > Cloud Orchestration > OpenStack > OpenStack Tenants**, and then click **Add**.
- In **Add OpenStack Tenants** page, click **+Add**, and then select the OpenStack tenant.
- Click **OK**.

B6.8 Provisioning NetScaler VPX instance in OpenStack

Download the required NetScaler instance image from the Citrix download page, and upload it on Glance, the OpenStack Imaging service. Having an image available on Glance allows you to configure a NetScaler instance on-demand when assigning the instance to the tenant.

To auto-provision the NetScaler VPX devices on OpenStack

1. In NetScaler MAS, navigate to **Orchestration > Cloud Orchestration > OpenStack**.
2. Click **Deployment Settings**.
3. Set the necessary parameters:
 - o Management Network
 - o Profile Name
 - o Licences
 - o NetScaler VPX image in Glance
 - o Proxy settings

The overall process of NetScaler MAS integration with OpenStack Workflow is shown in Figure 72.

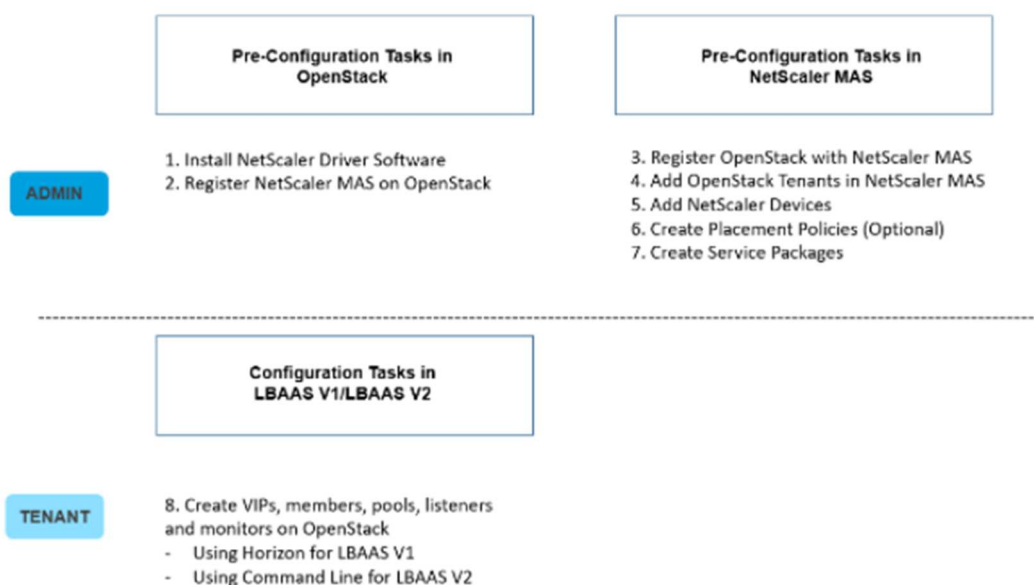


Figure 72: NetScaler MAS - OpenStack Integration Workflow [36]



The above outcomes will be integrated with the SUPERFLUIDITY platform as part of WP7 activities.



ANNEX C: Automated Security Verification Framework – Task 6.3

C1 Debugging P4 programs with Vera

See attached: PAPER – 2: Debugging P4 programs with Vera



C2 Equivalence and its applications to network verification

See attached: PAPER – 3: Equivalence and its applications to network verification



References

- [1] “NeMo: An Application’s Interface to Intent Based Networks” <http://nemo-project.net/>
- [2] “High-level VNF Descriptors using NEMO” <https://datatracker.ietf.org/doc/draft-aranda-nfvrg-recursive-vnf/>
- [3] ETSI NFV ISG, “Network Functions Virtualisation (NFV); VNF Packaging Specification”, ETSI GS NFV-IFA 011 V2.1.1 (2016-10) pdf link
- [4] ETSI NFV ISG, “Network Functions Virtualisation (NFV); Network Service Templates Specification”, ETSI GS NFV-IFA 014 V2.1.1 (2016-10) pdf link
- [5] L. Chiaraviglio, F. D’Andreagiovanni, G. Sidoretti, N. Blefari-Melazzi, S. Salsano, “Optimal Design of 5G Superfluid Networks: Problem Formulation and Solutions”, 21st Conference on Innovation in Clouds, Internet and Networks (ICIN 2018), February 20-22, 2018, Paris, France
- [6] L. Chiaraviglio, L. Amorosi, S. Cartolano, N. Blefari-Melazzi, P. Dell’Olmo, M. Shojafar, S. Salsano, “Optimal Superfluid Management of 5G Networks”, 3rd IEEE Conference on Network Softwarization, NetSoft 2017, 3-7 July 2017, Bologna, Italy
- [7] M. M. Tajiki, S. Salsano, M. Shojafar, L. Chiaraviglio, B. Akbari, “Energy-efficient Path Allocation Heuristic for Service Function Chaining”, 21st Conference on Innovation in Clouds, Internet and Networks (ICIN 2018), February 20-22, 2018, Paris, France
- [8] V. Frascolla, J. Englisch, L. Chiaraviglio, S. Salsano, S. Barberis, V. Palestini, A. De Domenico, E. Calvanese Strinati, K. Takinami, K. Yunoki, K. Sakaguchi, T. Haustein, “Millimeter-waves, MEC, and network softwarization as enablers of new 5G business opportunities”, 1st Workshop on Economics and Adoption of Millimeter Wave Technology in Future Networks at IEEE WCNC 2018, 15-18 April 2018, Barcellona, Spain
- [9] L. Chiaraviglio, N. Blefari-Melazzi, C.F. Chiasserini, B. Iatco, F. Malandrino, S. Salsano, “An Economic Analysis of 5G Superfluid Networks”, 18th IEEE International Conference on High Performance Switching and Routing (IEEE HPSR), 18-21 June 2017, Campinas, Brazil
- [10] RDCL 3D Home Page, <https://github.com/superfluidity/RDCL3D>
- [11] Salsano S, et al “RDCL 3D, a Model Agnostic Web Framework for the Design of Superfluid NFV Services and Components”, 3rd IEEE International Workshop on Orchestration for Software Defined Infrastructures, O4SDI at IEEE NFV-SDN conference, Berlin, 6-8 November 2017 (arXiv preprint: <https://arxiv.org/pdf/1702.08242>)
- [12] ETSI GR NFV-IFA 015, “NFV MANO Release 2; Report on NFV Information Model” V2.1.1 (2017-01)
- [13] ETSI GS NFV-IFA 011: “NFV MANO, VNF Packaging Specification”. V2.1.1 (2016-10)
- [14] “TOSCA Simple Profile for Network Functions Virtualization (NFV) Version 1.0”, Committee Specification Draft 03, OASIS, 2016



- [15] “TOSCA Simple Profile in YAML Version 1.0”, OASIS, 2016
- [16] J E. Kohler, et al., “The Click modular router,” ACM Transactions on Computer Systems (TOCS), vol. 18, no. 3, 2000
- [17] OSM Information Model, https://osm.etsi.org/wikipub/index.php/OSM_Information_Model
- [18] M. Shifrin, E. Biton, and O. Gurewitz. Optimal control of VNF deployment and scheduling. In Science of Electrical Engineering (ICSEE), IEEE International Conference on the, 2016.
- [19] OpenStack Octavia [Online]. Available: <https://wiki.openstack.org/wiki/Octavia> [Accessed 28/06/2017]
- [20] Citrix NetScaler ADC [Online]. Available: <https://www.citrix.com/products/netscaler-adc/>
- [21] Citrix NetScaler MAS [Online]. Available: <https://www.citrix.com/products/netscaler-management-and-analytics-system/>
- [22] Citrix Docs, “NITRO API”, <https://docs.citrix.com/en-us/netscaler/12/nitro-api.html>
- [23] Citrix Docs, “Integrating NetScaler MAS with OpenStack Platform”, <http://docs.citrix.com/en-us/netscaler-mas/12/integrating-netscaler-mas-with-openstack-platform.html>
- [24] Citrix Docs, “Load balancing algorithms”, <https://docs.citrix.com/en-us/netscaler/12/load-balancing/load-balancing-customizing-algorithms.html>
- [25] Citrix Docs, “About Persistence”, <https://docs.citrix.com/en-us/netscaler/12/load-balancing/load-balancing-persistence/persistence.html>
- [26] OpenStack Neutron/LBaaS Project [Online]. Available: <https://wiki.openstack.org/wiki/Neutron/LBaaS/NetScaler>
- [27] OpenStack LBaaS Netscaler Driver Repository [Online]. Available: https://github.com/openstack/neutron-lbaas/tree/master/neutron_lbaas/drivers/netscaler
- [28] “SRv6: Network as a Computer and Deployment use-cases” https://pc.nanog.org/static/published/meetings/NANOG71/1445/20171005_Dawra_Segment_Routing_ipv6_v1.pdf
- [29] <http://www.segment-routing.net/ietf/>
- [30] “Segment Routing IPv6 for Mobile User-Plane”, draft-ietf-dmm-srv6-mobile-uplane-00
- [31] “Study on User-plane Protocol in 5GC”, 3GPP #: 29.892
- [32] A. AbdelSalam, F. Clad, C. Filsfils, S. Salsano, G. Siracusano, L. Veltri, “Implementation of Virtual Network Function Chaining through Segment Routing in a Linux-based NFV Infrastructure”, 3rd IEEE Conference on Network Softwarization, NetSoft 2017, 3-7 July 2017, Bologna, Italy
- [33] F. Clad, C. Filsfils, P. Camarillo, D. Bernier, B. Decraene, B. Peirens, C. Yadlapalli, X. Xu, S. Salsano, A. AbdelSalam, G. Dawra, “Segment Routing for Service Chaining”, Internet Draft, draft-clad-spring-segment-routing-service-chaining, Work in progress, October 2017



- [34] C. Filsfils, et al, "SRv6 Network Programming", Internet Draft, draft-filsfils-spring-srv6-network-programming, Work in progress, October 2017
- [35] Citrix Docs, "Installing NetScaler MAS on KVM" [Online]. Available: <http://docs.citrix.com/en-us/netscaler-mas/12/deploy-netscaler-mas/install-mas-on-kvm.html>
- [36] Citrix Docs, "Integrating NetScaler MAS and OpenStack - Preconfiguration" [Online]. Available: <http://docs.citrix.com/en-us/netscaler-mas/12/integrating-netscaler-mas-with-openstack-platform/preconfiguration-tasks-mas-openstack.html>
- [37] "C4.5 algorithm", https://en.wikipedia.org/wiki/C4.5_algorithm



PAPER – 1: Optimizing NFV Chain Deployment Through Minimizing the Cost of Virtual Switching

Optimizing NFV Chain Deployment Through Minimizing the Cost of Virtual Switching*

Marcelo Caggiani Luizelli[†]
UNIPAMPA and UFRGS, Brazil
marceloluizelli@unipampa.edu.br

Danny Raz[†]
Technion, Israel
danny@cs.technion.ac.il

Yaniv Sa'ar
Nokia, Bell Labs
yaniv.saar@nokia-bell-labs.com

Abstract—*Network Function Virtualization (NFV)* is a novel paradigm that enables flexible and scalable implementation of network services on cloud infrastructure. A key factor in the success of NFV is the ability to dynamically allocate physical resources according to the demand. This is particularly important when dealing with the data plane since additional resources are required in order to support the virtual switching of the packets between the *Virtual Network Functions (VNFs)*. The exact amount of these resources depends on the way service chains are deployed and the amount of network traffic being handled.

Thus, orchestrating service chains that require high traffic throughput is a very complex task and most existing solutions either concentrate on handcrafted tuning of the servers to achieve the needed performance level, or present theoretical placement functions that assume that the switching cost is part of the input. In this work, we bridge this gap by presenting a deployment algorithm for service chains that optimizes performance by minimizing the actual cost of virtual switching. The results are based on extensive measurements of the actual switching cost and the performance of service chains in a realistic NFV environment. Our evaluation indicates that this new algorithm significantly reduces virtual switching resource utilization when compared to the de-facto standard placement in *OpenStack/Nova* – allowing a much higher acceptance ratio of network services.

I. INTRODUCTION

Despite the ever-growing popularity of *Network Function Virtualization (NFV)*, we are still far away from having large scale fully operational NFV networks. One of the main obstacles on this path is the performance of the network functions in the virtual environment. The hardware middleboxes that are in use by network operators today are specifically designed to provide the needed high performance (and high reliability), but getting the same level of performance from commercially off-the-shelf hardware is much more challenging. Hardware accelerators (such as DPDK and SR-IOV) were developed specifically for this purpose, yet the deployment of high performance service chains in a virtual environment still remains a complex handcrafted process (e.g. as indicated by the reports in [1] and [2]).

* This paper has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 671566 ("Superfluidity"). This paper reflects only the author's views and the European Commission is not responsible for any use that may be made of the information it contains.

[†] Work done while in Nokia, Bell Labs.

As a result, service chain placement in such scenarios is mostly static and operators lose one of the main attractive features of NFV – the ability to dynamically allocate resources according to the current need. Such a dynamic mechanism would allow a much more efficient utilization of resources, since the same physical resource can be used by different *Virtual Network Functions (VNFs)* when needed. Thus, achieving both high performance and agility, by being able to dynamically change the resource allocation of service chains, remains a great challenge.

Identifying near optimal deployment mechanisms for NFV service chains has recently received significant attention from both academia and industry ([3]–[15]). However, to the best of our knowledge, existing studies do not take into account the cost of resources required to steer the traffic within a chain, which is non-negligible for deployment of packet intensive chains such as in the domain of NFV. Therefore, typical models (e.g., in NFV orchestrators) might either lead to infeasible solutions (e.g., in terms of CPU requirements) or suffer high penalties on the expected performance.

One noticeable exception is the very recent paper [16] that focuses on evaluating and modeling the virtual switching cost in NFV-based infrastructure. Virtual switching is an essential building block that enables flexible communication between VNFs but it also comes with an extra cost in terms of computing resources that are allocated specifically to software switching in order to steer the traffic through running services (in addition to computing resources required by the VNFs). This cost depends primarily on the way the VNFs are internally chained, packet processing requirements, and accelerating technologies (such as DPDK [1]).

Figure 1 illustrates a possible deployment of four service chains on three identical physical servers (*A*, *B* and *C*). As one can see, service chain φ^1 is composed of three VNFs – $\varphi^1 = \langle \varphi_1^1, \varphi_2^1, \varphi_3^1 \rangle$, φ^2 is composed of four VNFs, φ^3 is composed of five, and φ^4 is composed of two VNFs. In the depicted deployment (shown in Figure 1), servers *A* and *C* have the same number of deployed VNFs and thus may have the same computing resource requirement for processing. However, determining the amount of processing resources (CPU)

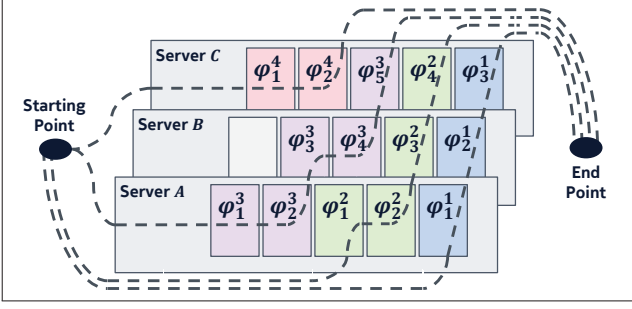


Fig. 1: Example of possible deployment of four service chains on top of three physical NFV servers

needed for switching inside these server is far from being straightforward. In some cases, the deployment can be infeasible due to lack of sufficient computing resources for the switching task. The amount of computing resources needed for the internal switching depends on the structure of the chaining in the server, and the amount of traffic associated with each chain.

Existing work that measure and evaluate the performance of the internal switching (e.g. [2], [16]–[18]) mainly focus on two types of deployment strategies: a *distribute* strategy where each VNF in the chain resides on a different server (for example, service chain φ^1 in Figure 1), and a *gather* strategy where the entire chain is grouped on the same server (for example, service chain φ^4 in Figure 1). In [2] these two deployment strategies are respectively referred to as “north/south” and “east-/west” deployments. The work in [16] analyzes these two placement strategies, and develops a cost function that predicts the amount of computing resources needed for the internal switching. Interestingly, OpenStack/Nova ([19]) global policies also follows these two deployment strategies. Namely, after a basic filtering step that clears resources according to local-server constraints (e.g., CPU cores, memory, affinity, etc.), it implements two types of global decisions that boil down to: *load balancing* of a certain metric (i.e., spread VNFs across servers), and; *energy saving* of a certain metric (i.e., stack VNFs up to the limit).

In this paper, we consider arbitrary deployments (like the one described in Figure 1) and develop a general service chain deployment strategy that considers both the actual performance of the service chains as well as the needed internal switching resource. This is done by decomposing service chains into sub-chains and deploying each such sub-chain on a (possibly different) physical server, in a way that minimizes the total switching overhead cost. Of course, there are exponentially many ways to map the sub-chains to servers. We introduce a novel algorithm based on an extension of the well-known reduction from the weighted matching to the min-cost flow problem, and show that it gives a near optimal

solution, with much more efficient run time comparing to the exhaustive search.

We evaluate the performance of our algorithm against the fully distribute or fully gather solutions, which are very similar to the placement of the *de-facto* standard mechanism commonly utilized on cloud schedulers (e.g., OpenStack/Nova with load balancing or energy conserving weights) and show that our algorithm significantly outperforms these heuristics (up to a factor of 4 in some cases) with respect to operational cost and the ability to support additional network functions.

The main contributions of this paper are:

- (i) **NFV deployment cost model.** We develop a general switching cost model that predicts the switching related CPU cost for arbitrary deployments and evaluate its accuracy over a real NFV environment.
- (ii) **Optimal deployment mechanism.** We develop an efficient online placement algorithm that uses this new cost model to minimize the switching cost of service chain requests. We evaluate the expected performance of this novel algorithm and show that it can significantly increase utilization by allowing more network functions to run on the same NFV infrastructure in a more efficient way.

The rest of the paper is organized as follows. In Section II we define our model and problem. In Section III we develop the cost function for the virtual switching. In Section IV we describe the proposed algorithm. In Section V we evaluated the performance and the quality of our algorithm, and in Section VI we discuss related works. Finally, in Section VII we conclude our work and provide future directions.

II. MODEL AND PROBLEM DEFINITION

The recommended best practice for virtualization intensive environment requires each server to allocate two disjoint sets of CPU core, one set for the hypervisor to provision resources and another set for the VNF to operate ([17]). Given a server S we denote by S^h the number of CPU cores that are allocated and reserved solely for the hypervisor to operate, by S^v the number of CPU cores reserved for the VNFs to operate and by S^{h+v} the overall number of cores in the server.

We define *service chain* φ to be an ordered set of VNFs $\varphi = \langle \varphi_1, \varphi_2, \dots, \varphi_n \rangle$, and denote by $|\varphi| = n$ its length. We denote by $|\varphi|^p$ (and $|\varphi|^s$) the number of packets per second (and average packet size, respectively) that service chain φ is required to process. For a VNF $\varphi_i \in \varphi$ we denote by φ_i^c the CPU required for its operation. Throughout this paper, unless explicitly saying otherwise, we assume that we are given a set of k servers $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$ and an ordered sequence of m service chains $\Phi = \langle \varphi^1, \varphi^2, \dots, \varphi^m \rangle^1$.

¹We always use subscript φ_i to refer to the i -th VNF in service chain φ , and upperscript φ^j to refer to the j -th service chain in Φ .

Given a service chain $\varphi = \langle \varphi_1, \varphi_2, \dots, \varphi_n \rangle$ we define a *sub-chain* of VNFs to be a sub-sequence $\varphi_{s \rightarrow e} = \langle \varphi_s, \dots, \varphi_e \rangle$ where $1 \leq s \leq e \leq n = |\varphi|$. We denote by \Rightarrow the operator that *concatenates* VNFs. An ordered set of r sub-chains $\langle \varphi_{s_1 \rightarrow e_1}, \dots, \varphi_{s_r \rightarrow e_r} \rangle$ is a *decomposition* of φ if the concatenation of all sub-chains recomposes φ , i.e. $\varphi = \langle \varphi_{s_1 \rightarrow e_1} \Rightarrow \dots \Rightarrow \varphi_{s_r \rightarrow e_r} \rangle$. We say that a set of VNFs is an *affinity group* (and *anti-affinity group*) if all VNFs are mapped to the same server (and respectively mapped to different servers).

We define $\mathcal{P} : \Phi \rightarrow \mathcal{S}^k$ to be a *placement function* that for every service chain $\varphi \in \Phi$, maps every VNF $\varphi_i \in \varphi$ to a server $S_j \in \mathcal{S}$. We denote two particularly interesting placement functions:

- (i) \mathcal{P}_g – we call *gather* placement, where every service chain φ is an affinity group, namely:

$$\forall \varphi \in \Phi \forall \varphi_i, \varphi_j \in \varphi : \mathcal{P}_g(\varphi_i) = \mathcal{P}_g(\varphi_j)$$

- (ii) \mathcal{P}_d – we call *distribute* placement, where every service chain φ is an anti-affinity group, namely:

$$\forall \varphi \in \Phi \forall \varphi_i, \varphi_j \in \varphi : i \neq j \rightarrow \mathcal{P}_d(\varphi_i) \neq \mathcal{P}_d(\varphi_j)$$

Recall that OpenStack/Nova ([19]) implements a limited set of global decisions policies: (i) load balancing – spreads VNFs across servers which corresponds to the distribute placement, and (ii) energy saving – stacks VNFs up to a limit which corresponds to the gather placement.

Figure 2 illustrates a few possible deployment strategies. Figure 2(a) depicts deployment of VNFs that follow gather placement function \mathcal{P}_g . Figure 2(b) depicts deployment of VNFs that follow distribute placement function \mathcal{P}_d . Figure 2(c) depicts deployment of VNFs that follow arbitrary mixed placement function \mathcal{P} . Intuitively, our goal is to develop a placement function \mathcal{P} that decomposes each service chain into arbitrary decomposition of the service chain, while minimizing the operational cost of the network traffic switching. The decomposition groups together sub-chain into an affinity group, while exactly one (arbitrary) VNF from each sub-chain is in the same anti-affinity group.

For a given placement function \mathcal{P} that deploys all service chains in $\Phi = \langle \varphi^1, \varphi^2, \dots, \varphi^m \rangle$ on the set of servers $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$, we define two cpu-cost functions:

- (i) $\mathcal{C}^v : \mathcal{P} \rightarrow (\mathbb{R}^+)^k$ is the *guest cpu-cost function* that assigns per server the CPU required to operate the VNFs allocated on the server.
- (ii) $\mathcal{C}^h : \mathcal{P} \rightarrow (\mathbb{R}^+)^k$ is the *hypervisor cpu-cost function* that assigns per server the CPU required to operate the network traffic switching on the server.

For simplicity, we use \mathcal{C}_i^v (and \mathcal{C}_i^h) to refer to the i 'th value associated with server S_i , and \mathcal{C}^v (and \mathcal{C}^h) to denote the sum of all k servers, i.e., $\mathcal{C}^v = \sum_{i=1}^k \mathcal{C}_i^v$ (and respectively $\mathcal{C}^h = \sum_{i=1}^k \mathcal{C}_i^h$). We say that a placement function \mathcal{P} is *feasible* with respect to the set of servers \mathcal{S} , if for every

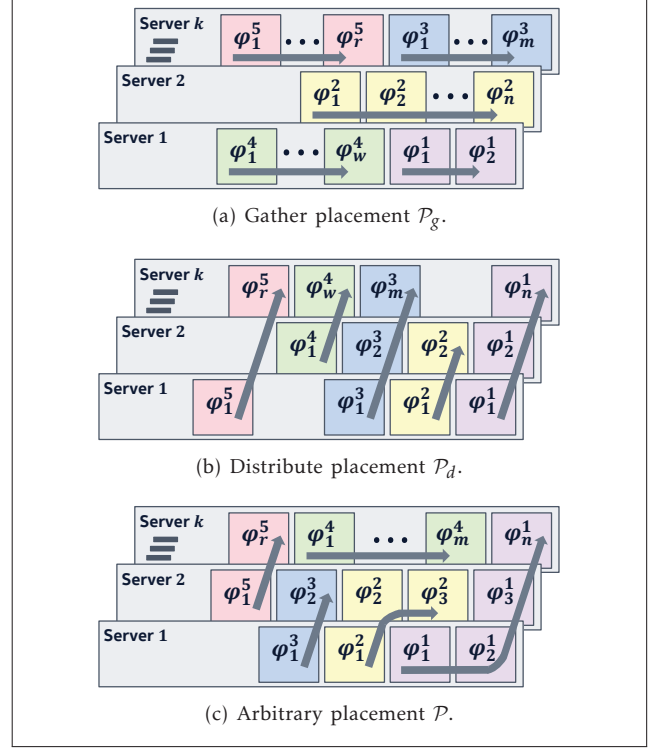


Fig. 2: Given a set of 5 service chains Φ and a set of k servers \mathcal{S} , illustrating different deployment strategies

server, the sum of the hypervisor cpu-cost function and the guest cpu-cost function do not exceed the number of CPU cores that are available on this server, i.e.,

$$\forall S_j \in \mathcal{S} : \mathcal{C}_j^h(\mathcal{P}) + \mathcal{C}_j^v(\mathcal{P}) \leq S_j^{h+v} \quad (1)$$

Note that given a service chain φ , the distribute placement function requires to have at least $|\varphi|$ servers in order for the deployment to be feasible. On the other hand the gather placement function requires to have at least one server that can host all VNFs in φ in order for the deployment to be feasible. Thus, there could be situations where neither of these placements is feasible while a feasible placement exists.

We can now formally define the algorithmic problem of interest. We are given a set of k servers \mathcal{S} , and an online sequence of m service chains Φ . For each service chain in the sequence we need to find a feasible placement to all the VNFs in the chain such that the total cpu-cost increase is minimized. Finding a feasible placement for a given service chain is a zero-one acceptance problem since we are not allowed to place only part of the service chain, i.e., we either have to place the entire chain or nothing. Minimizing the total cpu-cost releases additional resources that can potentially be used to allocated additional network functions, and improves the acceptance rate of service chains in the sequence. Our evaluation of acceptance rate (see Section V) shows that

the online cpu-cost minimization algorithm outperforms OpenStack/Nova scheduler, and brings us to a near optimal offline minimization.

III. THE OPERATIONAL COST OF SWITCHING

In this section we define a model that captures the operational cost of virtual switching for a given server. Namely, given a placement function \mathcal{P} that entails a set of sub-chains (originating from possibly different service chains) to deploy on server S_j , our goal is to develop a function that predicts the CPU cost of server S_j .

A. Virtual Switching

In virtualization-intense environments, virtual switching is an essential functionality that provides isolation, scalability, and mainly flexibility. However, the functionality provided by software switching also introduces a non-negligible operational cost making it much harder to guarantee a reasonable level of network performance, which is a key requirement for the success of the NFV paradigm. Assessing and understanding this operational cost and particularly the cost associated with virtual switching is a crucial step towards driving cost-efficient service deployments.

Several recent publications addressed the performance of intensive traffic applications in NFV chaining settings (e.g., [2], [16]–[18]). In most industry related work the goal is to define the setting that provides the best performance on a specific hardware, and not on the switching cost of a given service chain under a certain setting. As we mentioned, [16] is different as it does try to evaluate the switching cost but it does so only for the special cases of gather and distribute.

Yet the results of these recent studies indicate that the operational cost of deploying service chains depends on the installed OvS (either kernel OvS or DPDK-OvS), the required amount of traffic to process, the length of the service chain, and the placement strategy. Moreover, it appears that there is no single strategy that is always superior, with respect to the operational cost, and that the best strategy depends on the system parameters and the characterization of the deployed service chains.

B. The Cost of Virtual Switching

Let $\{\varphi_{s_1 \rightarrow e_1}^1, \dots, \varphi_{s_r \rightarrow e_r}^r\}$ be a set of r sub-chains, each is part of a decomposition from possibly different service chain. Each sub-chain $\varphi_{s_w \rightarrow e_w}^w$ (for $1 \leq w \leq r$) might carry different traffic requirements, that are defined by service chain $\varphi^w \in \Phi$. Figure 3 illustrates such a deployment on server S_j . The total guest cpu-cost $\mathcal{C}_j^v(\mathcal{P})$ is just the sum of the required CPU for each VNF, but calculating the hypervisor switching cpu-cost $\mathcal{C}_j^h(\mathcal{P})$ is much more involved.

For a single sub-chain the switching cost is exactly the gather cost of a chain deployed on server S_j and can be obtained directly from function \mathcal{F} defined in [16].

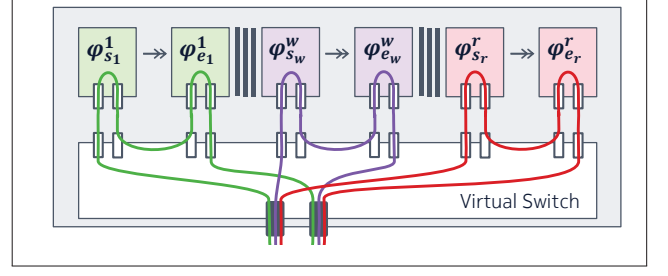


Fig. 3: Server S_j deployed with r sub-chains from possibly different service chains $\{\varphi_{s_1 \rightarrow e_1}^1, \dots, \varphi_{s_w \rightarrow e_w}^w, \dots, \varphi_{s_r \rightarrow e_r}^r\}$

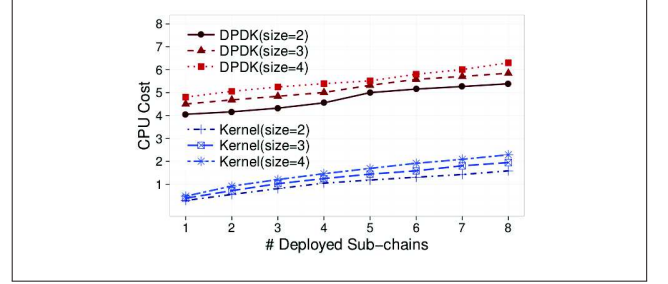


Fig. 4: The cpu-cost of concurrent deployment as a function of r , for several selected sub-chain sizes

However, when we deploy more than one sub-chain, the overall cost is not the sum of the separate deployments. Figure 4 depicts this CPU switching cost as a function of the number of concurrent deployments (for various sub-chain lengths and different switching configurations). One can see that the cost of deploying two sub-chains is much smaller than twice the cost of deploying one sub-chain (this is true for each of the sub-chains shown in this figure). Thus in concurrent chain deployments the switching cost is amortized and the total cost is smaller than the sum of two separate costs.

To quantify this we define $\|\varphi_{s \rightarrow e}\|^r$ to be the *concurrent deployment* of r copies of subchain $\varphi_{s \rightarrow e}$, and $\Delta(\varphi_{s \rightarrow e}, r)$ to be the *switching cost delta* between r times deploying a single sub-chain $\varphi_{s \rightarrow e}$ and the concurrent deployment of r copies of $\varphi_{s \rightarrow e}$, i.e.,

$$\Delta(\varphi_{s \rightarrow e}, r) = (r \cdot \mathcal{F}(\varphi_{s \rightarrow e}) - \mathcal{F}(\|\varphi_{s \rightarrow e}\|^r)) \quad (2)$$

We performed extensive evaluations separately for environment installed with kernel OvS and DPDK-OvS and for different sub-chain lengths. Part of the results are shown in Figure 4. In order to be compliant with previous work, we used in this evaluation an environment setup that is similar to the one described in [16].

Given a set of r sub-chains $\{\varphi_{s_1 \rightarrow e_1}^1, \dots, \varphi_{s_r \rightarrow e_r}^r\}$ that are deployed on server S_j (as illustrated in Figure 3), we can

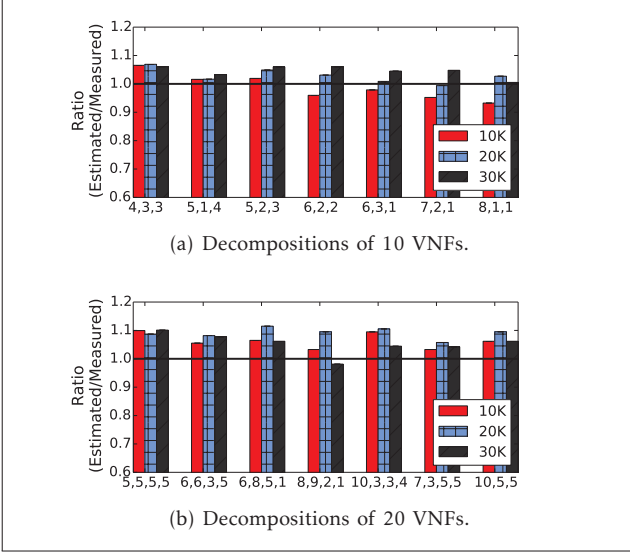


Fig. 5: The ratio between the computed values of \mathcal{C}_j^h compared to measurements of the corresponding executions

now compute the cpu-cost as follows:

$$\mathcal{C}_j^h = \sum_{w=1}^r \mathcal{F}(\varphi_{s_w \rightarrow e_w}^w) - (r-1) \cdot \left(\sum_{w=1}^r \Delta(\varphi_{s_w \rightarrow e_w}^w, r) \right) / r \quad (3)$$

Where $\Delta(\varphi_{s_w \rightarrow e_w}^w, r)$ is computed as shown in Equation 2, with the traffic requirements defined for $\varphi_{s_w \rightarrow e_w}^w$. The left-hand side of Equation 3 is the sum of the cpu-cost associated with the gather deployment of each sub-chain, and the right-hand side reflects the saving due to the concurrent deployment of r chains which is $(r-1)$ times the average switching cost delta. Note that for $r=1$ we do not have any savings and the cost is just the gather cost of deploying a single sub-chain.

C. Empirical Validation

To validate our formulation, we evaluate function \mathcal{C}_j^h for various sub-chain deployment configurations. Figure 5 presents the ratio between the cpu-cost estimated by function \mathcal{C}_j^h compared to measurements of the corresponding executions, for several traffic requirements. Figure 5(a) depicts the ratio for 10 VNFs, which are grouped into 3 different sub-chains, and Figure 5(b) depicts the ratio for 20 VNFs, which are grouped into 4 different sub-chains. As shown, our estimated cpu-cost is not very far from the measured executions (on average, less than 5%).

IV. OPTIMIZED OPERATIONAL COST PLACEMENT

In this section we introduce our placement algorithm for Operational (switching) Cost Minimization (OCM). Given a sequence of service chains Φ and a set of physical servers \mathcal{S} , our algorithm entails a strategy to deploy the service chains onto the set of physical servers. Per

each service chain in the sequence, we find a partition of the chain into sub-chains and an allocation of a server to each sub-chain in a way that minimizes the total switching cost. This on-line handling does not guarantee global minimum switching cost but as we show in this section it does reduce the switching CPU cost and allow deploying significantly more services.

A. Service Chain Partitioning

Given a service chain φ , our goal is to assign a server for each VNF $\varphi_i \in \varphi$, such that the overall placement is feasible and has the minimal network switching cost. OpenStack/Nova implements a naive approach which goes over all VNFs in the service chain, for each VNF filter the feasible servers, and then follows one of its global policies to assign a server. There are two problems with this approach. The first problem is that the placement is done per VNF and thus after placing several VNFs, it might realize that there is no feasible server for the next VNF and revoke the placement of the entire service chain. The second drawback is the complexity of this algorithm (even when succeeding to find a feasible allocation). When the number of servers is $K = |\mathcal{S}|$ and the maximum number of VNFs in service chain is bounded by N , the runtime complexity of this approach is $O(K^N)$. Since in practice the number of servers can reach hundreds or more, this approach easily becomes impractical.

To address this scalability issue we take a different approach, which iterates over all set partitions of the service chain into subsets of VNFs, relying on the fact that the size of service chains is practically bounded by a small constant. For each partition in the set, we use the switching cost model, described in the previous section, to determine the cost of \mathcal{C}_j^h for each server. Then we find an optimal placement of this partition by finding the maximal matching between sub-chains and servers. To analyze the complexity of this approach, we need to bound the number of different partitions, which entails the required number of iterations.

The number of all possible partitioning of a set is known to be the *Bell number* of N (denoted by $B(N)$), and a recent study ([20]) established an upper bound of $(\frac{0.792N}{\ln(N+1)})^N$ for it. Thus, in order to find an optimal placement, the *optimal* OCM algorithm needs to iterate over $O(N^N)$ different partitions of the chain. Note, however, that this complete enumeration of set partitioning also includes placements that allow the traffic to enter and leave a server more than once (e.g., partitioning service chain $\varphi = \langle \varphi_1, \varphi_2, \varphi_3 \rangle$ into subsets $\{\varphi_1, \varphi_3\}$ and $\{\varphi_2\}$). Since in practice this is not a reasonable placement, we consider a heuristically relaxed version of the problem that allows much fewer iteration (however, does not guarantee to find the optimum).

When assuming that in a reasonable deployment every service chain goes through a server at most once, finding

all set partitioning (without repetitions) corresponds to enumerating all possible decompositions of φ . Thus we need to solve the combinatorial problem known as *integer composition*, which lists all possible ordered lists of positive integers n_1, \dots, n_r , such that their sum is equal to N (i.e. $N = \sum_{i=1}^r n_i$). Integer composition has been widely studied in the literature, including fast algorithms for its solution (e.g., [21], [22]), and the number of all possible integer compositions of N is exactly 2^{N-1} . Namely in order to find an optimal placement that goes through a server at most once, the OCM algorithm requires to iterate over $O(2^N)$ different partitions.

B. Using Matching to Find Optimal Cost Placement

For each partition, we want to find an optimal placement that maps each sub-chain in it to a server in \mathcal{S} . Note that we can also restrict the assignment such that each server is assigned with at most one sub-chain. This is true since assigning two consecutive sub-chains is equal to assigning a bigger sub-chain and this is covered by a different partition. We construct a bipartite graph where the nodes on one side are the sub-chains in the partition and the nodes on the other side are the servers in \mathcal{S} . The cost of an edge that connects sub-chain to a server is the extra C_j^h cost according to the switching cost model if an assignment of this sub-chain to this server is feasible, and ∞ otherwise (equivalent to removing this edge).

It is not too difficult to verify that a minimal cost placement of this set partitioning corresponds exactly to a minimum-weight perfect matching in the bipartite graph. Solving the minimum-weight perfect matching in the bipartite graph problem can be reduced to the problem of finding a minimum cost flow in a graph ([23]). The complexity is polynomial, and for a given service chain φ of length $N = |\varphi|$ and K servers, can be solved in time $O((N + K) \cdot N^2 \cdot K^2)$.

C. Operational Cost Minimization Algorithm

We are now ready to present our placement algorithm. The *Operational Cost Minimization* (OCM) algorithm receives as an input a set of servers $\mathcal{S} = \{S_1, S_2, \dots, S_K\}$ and a sequence of service chains $\Phi = \langle \varphi^1, \varphi^2, \dots, \varphi^m \rangle$. For each service chains $\varphi \in \Phi$, the OCM invokes the optimal service chain placement step shown in Algorithm 1 that is made of three building blocks: (i) list all set partitions of the VNFs in φ , or all decompositions in the relaxed version of the algorithm (Subsection IV-A); (ii) Given a set partition build the objective function, namely a cost function that predicts the operational cost of network traffic switching per each server (Subsection III-B); (iii) Given an objective function build a reduction to minimum-weight matching in bipartite graphs between partitions and servers, where the weights are given by the objective function (Subsection IV-B). We denote by \mathcal{P}_o (and \mathcal{P}_r) the *optimal* placement function (and the *relaxed* placement function) that implements the optimal OCM

Algorithm 1 optimal service chain placement step

Input: $\mathcal{S} \leftarrow \{S_1, S_2, \dots, S_K\}$: set of servers
 $\varphi \leftarrow \langle \varphi_1 \rightarrow \varphi_2 \dots \rightarrow \varphi_n \rangle$: service chain
1: $\text{min-cost} \leftarrow \infty$: minimum cost found so far
2: $\text{deploy-map} \leftarrow \text{NIL}$: maps all VNFs to servers in \mathcal{S}
3: $\mathcal{A} \leftarrow$ all possible set partitioning (or decompositions)
4: **for** every set partition $a \in \mathcal{A}$ **do**
5: **for** every partition $p \in a$, and server $S_j \in \mathcal{S}$ **do**
6: $C_j^h, C_j^v \leftarrow$ compute the cost of deploying p on S_j
7: $G \leftarrow$ reduce to minimum cost flow in a graph
8: **if** $\text{min}(G) \leq \text{min-cost}$ **then**
9: $\text{min-cost} \leftarrow \text{min}(G)$
10: $\text{deploy-map} \leftarrow$ extract solution from G
11: **return** deploy-map

algorithm (and respectively the relaxed version of the OCM algorithm).

Recall that in practice the number of servers K can reach hundreds or more, and since the runtime complexity of OpenStack/Nove approach is $O(K^N)$, it easily becomes impractical for chains that contain more than 2-3 VNFs. The OCM algorithm iterates over set partitioning, and for each set partition reduces the problem to minimum-weight matching in bipartite graphs, that can be solved in runtime of $O((N + K) \cdot N^2 \cdot K^2)$. The number of different set partitioning that the optimal OCM algorithm needs to iterate is bounded by $O(N^N)$, which entails a runtime complexity of $O(N^N \cdot (N + K) \cdot N^2 \cdot K^2)$ that can roughly scale to chains of size 5-6. On the other hand, the number of different decompositions that the relaxed version of the OCM algorithm needs to iterate is $O(2^N)$, which entails a runtime complexity of $O(2^N \cdot (N + K) \cdot N^2 \cdot K^2)$. Since typical chain length is roughly between 2 and 10 VNFs, the number of different decompositions is practically a constant. In Subsection V-A we evaluate the running times of all approaches, which reaffirms our formal analysis.

V. EVALUATION

To assess the expected performance of the OCM algorithm as well as the quality of the results (i.e., the ability to increase utilization) compared to commonly used placement strategies, we implemented placements \mathcal{P}_o and \mathcal{P}_r , and performed an extensive set of simulation based evaluations.

We consider a typical NFV-node (i.e., a small data-center) that is composed of 100 high end servers. Each server is an high-end HP ProLiant DL380p Gen8 server with: two Intel Xeon E5-2697v2 processors, where each processor is made of 12 physical cores (24 cores in total) running at 2.7 Ghz; two NUMA nodes (Non-Uniform Memory Access), each has 192 GBytes RAM (total of 384 GBytes), and; an Intel 82599ES 10 Gbit/s network device with two network interfaces (physical ports). This

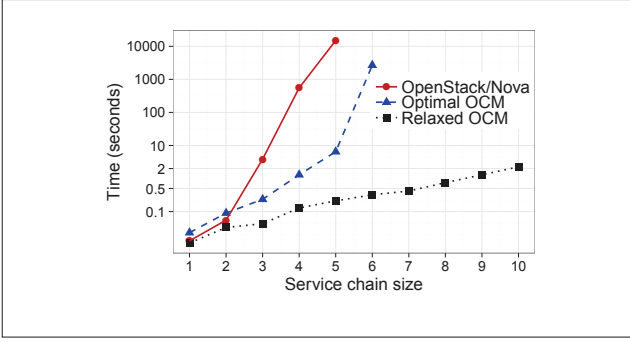


Fig. 6: Runtime analysis of different placements

is the type of servers that were used in the evaluation of the cost function \mathcal{C}_j^h . We measure the operational cost of switching using standard Open vSwitch (kernel OvS) as well as on DPDK accelerated Open vSwitch (DPDK-OvS).

For the purpose of this evaluation, we developed a generator of service chains requests (i.e., Φ). Service chain requests are handled one by one as described in our optimal service chain placement step (Subsection IV-C). The number of VNFs in a service chain is chosen uniformly at random in the range [2-10] (which is roughly the expected size of service chains in real deployments). Since our goal is to assess the operational cost of network switching, which is not directly affected by the amount of CPU needed by the VMs that implement the VNF, we assign to each VNF a single CPU. We vary the amount of packets processed in each service chain in the range of 10K to 1.5M packets per second. Unless otherwise specified, each experiment was performed 30 times, and considered 700 different service chain requests.

A. Evaluating the Runtime

Figure 6 depicts the runtime (in log scale) of placing a single service chains request, comparing our placements, \mathcal{P}_o and \mathcal{P}_r with OpenStack/Nova for different service chain sizes.

The evaluations reaffirm our formal analysis (see Subsection IV-C). Placement \mathcal{P}_r results indeed show that the runtime aligns with the analyzed complexity which adds a factor of $O(2^N)$ that as expected can easily scale to service chains of length ~ 10 and more (executing for few seconds). The same analysis is also valid for placement \mathcal{P}_o which adds a factor of $O(N^N)$ that as expected can scale to chains of length ~ 6 (up to a minute of execution). Finally, the results of OpenStack/Nova quickly surpasses 10K seconds for very few VNFs in the chain, which again aligns with our complexity analysis of $O(K^N)$ (i.e., the runtime is sensitive to the number of servers K).

When scaling our experiment to 1K servers, the relaxed OCM is still able to find solutions in a reasonable

time (in the order of few seconds in the worst case). It is important to emphasize that we significantly improved the runtime while delivering quality-wise solutions as shown in the following.

B. Evaluating the Quality of the Results

We compare the quality of the results of placement function \mathcal{P}_r to the two commonly used strategies as discussed in Section II: (i) distribute strategy (load balancing policy in OpenStack/Nova, north/south in [2]) (ii) gather strategy (energy saving policy in OpenStack/Nova, east/west in [2]). We assume hard requirements (instead of soft constraints) and, therefore, requests are rejected whenever it is not possible to meet such requirements.

NFV operational cost. Figure 7(a) and Figure 8(a) depict the average additional CPU required to deploy a VNF, ranging over the traffic requirements of the service chain (in packet per seconds). As we observe, the additional number of cores needed for the virtual switching functionality is not negligible, and the cost tends to be higher as the traffic requirement increases. Note that for low network traffic requirements the operational cost of the distribute strategy is substantially higher in DPDK-OvS. This is since in DPDK-OvS there is a subset of poll-mode threads in the hypervisor, which consume resources regardless to the network traffic (doing busy-waiting). When comparing the amount of the additional resources required, placement function \mathcal{P}_r requires less resources to achieve the same network performance guarantee. The improvement factor in the common case ranges between 20% and 40%, and in extreme cases can reach to as high as 400%.

Service chain acceptance ratio. Figure 7(b) and Figure 8(b) depict the average acceptance ratio (namely, the ratio between the number of deployed chains and the number of the requested chains), ranging over different network traffic requirements. Placement function \mathcal{P}_r is able to deploy a higher number of requests in most case, except for very few cases where its results are similar to the gather strategy \mathcal{P}_g . Since the distribute strategy \mathcal{P}_d requires higher switching resources, the acceptance ratio is substantially affected in those cases (see Figure 8(b), where the acceptance is $\approx 25\%$). However, note that for higher network traffic requirements, the acceptance of the gather strategy degrades to be worst than the the distribute strategy. In turn, placement function \mathcal{P}_r better takes advantage of the available resources, since it provides flexible solutions that better fit available physical resources and, therefore, ensures a higher acceptance rate. The improvement of placement function \mathcal{P}_r over standard deployment policies is much more noticeable if we look at requests that carry high network traffic requirements.

Physical infrastructure resource utilization. Figure 7(c) and Figure 8(c) illustrate the average unused re-

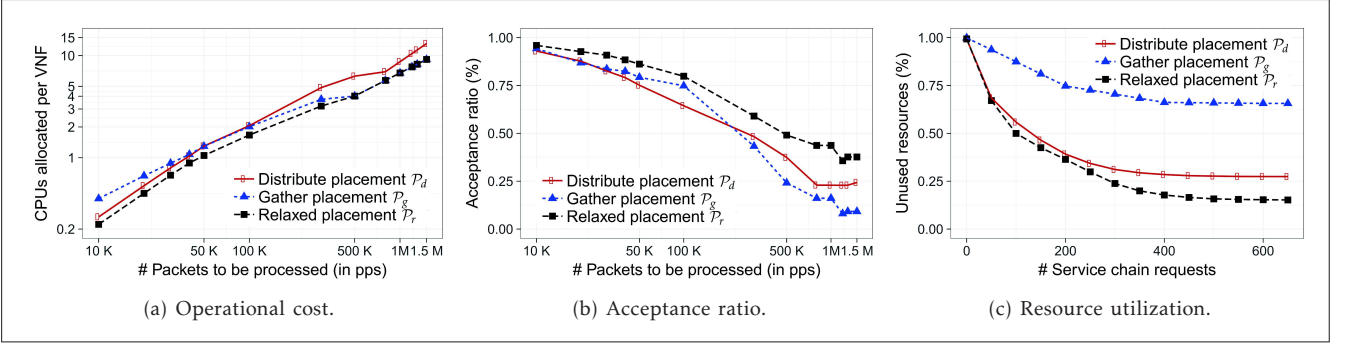


Fig. 7: Analysis of service chains deployment on NFV servers with kernel OvS

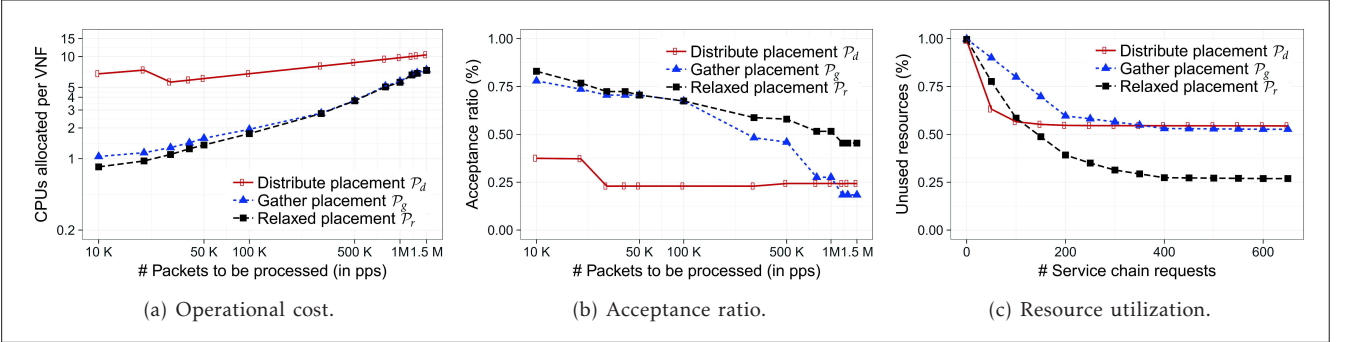


Fig. 8: Analysis of service chains deployment on NFV servers with DPDK-OvS

source over time (i.e., the lower the average values the better). Observe that placement function P_r can utilize more resources in the physical infrastructure (up to 18% of unused resources) with comparatively lower operational cost. The average of unused resources may be lower depending on the workload (for longer service chains, placement function P_r can better take advantage of chain decomposition). In DPDK-OvS, the average percentage of unused resources is slightly higher in comparison to kernel OvS. This is since DPDK-OvS requires a fixed amount of resources to operate (the threads required by the poll-mode drivers) which reflects high operational costs and less flexibility on placing service chains.

Overall one can see that reducing the switching CPU cost locally at each deployment of a new request, proves itself to be a very good strategy as it indeed improves the acceptance rate and allows a much better utilization of the resources.

VI. RELATED WORK

Various studies considered the problems of virtual network function placement ([4], [5], [14]) and chaining ([3], [6]–[9], [13], [15]). However, none of these works take into account the cost of resources required to steer the traffic within a chain, which is non-negligible for deployment of packet intensive chains such as in the

domain of NFV. Therefore, these solutions might either lead to infeasible solutions or suffer high penalties on the expected performance.

In [5] the authors addressed VNF placement, and used an optimization model along with approximation algorithms to solve the problem. They focus on where to deploy (VNFs) and how to assign traffic flows. Their work established a theoretical background for NFV placement, building on two classical optimization problems, namely facility location and generalized assignment. In [4] the authors focused on cost-oriented placement of elastic demands. They provide a dynamic mechanism to place, migrate and/or reassign network traffic mainly to cope with traffic fluctuations. However, their objective function do not take into account the internal switching cost of the server.

The authors of [3], [6], [15] and [7] studied joint optimization problems for placement and chaining of VNFs. In [3] they focused on formally specifying service chains and on analyzing their ILP model under different objective functions. In turn, [6] and [15] provide a general ILP model which takes into account end-to-end delay and resources constraints. The work in [7] provides a similar model that focus on reducing the general operational expenditures of datacenters over time (e.g., energy consumption of deployed services). [10] explored the relation between resource consumption on physical

servers and links. [11] and [12] were the first to introduce approximation algorithms to the joint NFV optimization problem. The work in [11] presented the first polynomial time service chain approximation for request admission control. Their solution is based on classical rounding techniques. Finally, [12] provides a deterministic approximation algorithm based on submodular functions covering incremental deployment.

Most of these work focused on arbitrary cost functions (e.g., reducing the amount of deployed VNFs). However, the real operational cost depends on many factors including the server settings and the way VNFs are deployed on physical servers (intra- and inter-server). Several recent studies (mainly industry-driven) address this issue of “configuration for optimal performance” [2], [16]–[18], but in these works this is done by a careful manual process. For the best of our knowledge this is the first paper that addresses this gap and presents an automated algorithmic approach for this important problem.

VII. CONCLUSION AND FUTURE WORK

In this work we introduced the Operational Cost Minimization (OCM) placement algorithm, a performance-oriented deployment mechanism that minimizes internal switching CPU overhead and improves network utilization. This algorithm uses a novel cost model that captures the operational cost of the internal virtual switching for a given server. We provided empirical evidence, using a real NFV-based environment, indicating that our cost model is accurate comparing to actual deployment measurements (lower than 5%). Using this cost model, we introduced an efficient online placement algorithm that minimizes the switching cost of service chain requests. We show that OCM significantly reduces the operational costs and increases utilization, when compared to commonly used deployment strategies (up to a factor of 4 in the extreme case and 20% – 40% in typical cases).

We plan to integrate OCM in OpenStack/Nova and measure its performance in real NFV-based deployments. We also intend to explore extensions of our algorithm/model to support non-linear service chains and to cope with others network requirements (e.g., network monitoring). Finally, our work opens opportunities to design new pricing models for service chaining by NFV providers.

REFERENCES

- [1] “Intel dpdk,” <http://dpdk.org/>, accessed: 07-20-2017.
- [2] P. Kutch and B. Johnson, “Sr-iov for nfv solutions practical considerations and thoughts,” <http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/sr-iov-nfv-tech-brief.pdf>, Intel Networking Division (ND), Feb 2017.
- [3] S. Mehraghdam, M. Keller, and H. Karl, “Specifying and placing chains of virtual network functions,” in *Cloud Networking (CloudNet)*, 2014 IEEE 3rd International Conference on, Oct 2014, pp. 7–13.
- [4] M. Ghaznavi, A. Khan, N. Shahriar, K. Alsubhi, R. Ahmed, and R. Boutaba, “Elastic virtual network function placement,” in *Cloud Networking (CloudNet)*, 2015 IEEE 4th International Conference on, Oct 2015, pp. 255–260.
- [5] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, “Near optimal placement of virtual network functions,” in *2015 IEEE Conference on Computer Communications (INFOCOM)*, April 2015, pp. 1346–1354.
- [6] M. C. Luizelli, L. R. Bays, L. S. Buriol, M. P. Barcellos, and L. P. Gaspary, “Piecing together the nfv provisioning puzzle: Efficient placement and chaining of virtual network functions,” in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, May 2015, pp. 98–106.
- [7] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba, “On orchestrating virtual network functions,” in *Proceedings of the 2015 11th International Conference on Network and Service Management (CNSM)*, ser. CNSM ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 50–56.
- [8] W. Rankothge, J. Ma, F. Le, A. Russo, and J. Lobo, “Towards making network function virtualization a cloud computing service,” in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, May 2015, pp. 89–97.
- [9] M. Bouet, J. Leguay, and V. Conan, “Cost-based placement of vdpi functions in nfv infrastructures,” in *Network Softwarization (NetSoft)*, 2015 1st IEEE Conference on, April 2015, pp. 1–9.
- [10] T. Kuo, B. Liou, J. Lin, and M. Tsai, “Deploying chains of virtual network functions: On the relation between link and server usage,” in *IEEE International Conference on Computer Communications (INFOCOM 2016)*, San Francisco, USA, April 2016.
- [11] M. Rost and S. Schmid, “Service chain and virtual network embeddings: Approximations using randomized rounding,” *CoRR*, vol. abs/1604.02180, 2016. [Online]. Available: <http://arxiv.org/abs/1604.02180>
- [12] T. Lukovszki, M. Rost, and S. Schmid, “It’s a match!: Near-optimal and incremental middlebox deployment,” *SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 1, pp. 30–36, Jan. 2016.
- [13] J. J. Kuo, S. Shen, H. Kang, D. Yang, M. Tsai, and W. Chen, “Service chain embedding with maximum flow in software defined network and application to the next generation cellular network architecture,” in *IEEE International Conference on Computer Communications (INFOCOM 2017)*, Atlanta, USA, April 2017.
- [14] M. Wenrui, O. Sandoval, J. Beltran, D. Pan, and N. Pissinou, “Traffic aware placement of interdependent nfv middleboxes,” in *IEEE International Conference on Computer Communications (INFOCOM 2017)*, Atlanta, USA, April 2017.
- [15] M. C. Luizelli, W. L. da Costa Cordeiro, L. S. Buriol, and L. P. Gaspary, “A fix-and-optimize approach for efficient and large scale virtual network function placement and chaining,” *Computer Communications*, vol. 102, pp. 67 – 77, 2017.
- [16] M. C. Luizelli, D. Raz, Y. Sa’ar, and J. Yallouz, “The actual cost of software switching for nfv chaining,” in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, May 2017, pp. 335–343.
- [17] Intel, “Intel open network platform release 2.1: Performance test report,” Internet Engineering Task Force, Mar. 2016, Available: <https://01.org/packet-processing/intel@-onp>.
- [18] B. O. Maryam Tahhan and A. Morton, “Benchmarking virtual switches in opnfv,” <https://tools.ietf.org/html/draft-ietf-bmvg-vswitch-opnfv-04>, Work in progress, accessed: 07-20-2017.
- [19] “Openstack,” <http://www.openstack.org/>, accessed: 07-20-2017.
- [20] D. Berend and T. Tassa, “Improved bounds on bell numbers and on moments of sums of random variables,” *Probability and Mathematical Statistics*, vol. 30, no. 2, pp. 185–205, 2010.
- [21] M. Merca, “Fast algorithm for generating ascending compositions,” *Journal of Mathematical Modeling and Algorithms*, vol. 11, no. 1, pp. 89–104, 2012.
- [22] J. Kelleher and B. O’Sullivan, “Generating all partitions: A comparison of two encodings,” *CoRR*, vol. abs/0909.2331, 2009. [Online]. Available: <http://arxiv.org/abs/0909.2331>
- [23] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network flows - theory, algorithms and applications*. Prentice Hall, 1993.



PAPER – 2: Debugging P4 programs with Vera

Debugging P4 programs with Vera

Paper #44

ABSTRACT

We present Vera, a tool that exhaustively verifies P4 programs using symbolic execution. Vera automatically uncovers a number of common bugs including parsing/deparsing errors, invalid memory accesses, loops and tunneling errors, among others. Vera can also be used to verify user-specified properties in a novel language we call NetCTL.

To enable scalable, exhaustive verification Vera automatically generates all valid header layouts, it uses symbolic table entries to simulate a variety of table rule snapshots and uses a novel data-structure for match-action processing optimized for verification. These techniques allow Vera to scale very well: it only takes between 5s-15s to track the execution of a purely symbolic packet in the largest P4 program currently available (6KLOC) and can compute SEFL model updates according to table insertions and deletions in milliseconds.

We have used Vera to analyze all P4 programs we could find including the P4 tutorials, P4 programs in the research literature and the switch code from <https://p4.org>. Vera has found several bugs in each of them in seconds.

1 INTRODUCTION

Programmable network dataplanes such as those enabled by P4 [2] promise to help our networks meet ever-increasing application demands. On the downside, unverified changes to network functionality can introduce bugs that may cause great damage. Recently, faulty routers in two airline networks have grounded airplanes for days (for both Delta and Southwest Airlines), showing just how disruptive the effects of incorrect network behavior can be. Given the momentum behind programmable networks, we expect such faults and many others will cripple programmable networks.

In this paper, we argue that dataplane programs should be verified before deployment to enable safe operation. We present Vera, a verification tool that enables debugging of P4 programs both before deployment and at runtime. At its core, Vera translates P4 to SEFL, a network language designed for verification, and relies on symbolic execution with Symnet [28] to analyze the behavior of the resulting program. Vera incorporates a set of novel techniques that together enable scalable and easy-to-use P4 verification.

Vera exhaustively verifies a P4 program: it uses the parser of the P4 program to generate all parsable packet layouts (e.g. header combinations), and makes all header fields symbolic (i.e. they can take any value). It then tracks the way these packets are processed by the program, following all branches

to completion. We manually prove that Vera correctly translates from P4 to SEFL by defining the operational semantics for both P4 and SEFL and by proving that P4 instruction(s) and their SEFL translation are equivalent.

Vera automatically checks for common problems in P4 programs including loops, parsing/deparsing errors, tunneling bugs, overflows and underflows, among others. Since verification is exhaustive, if Vera does not find such problems it guarantees the P4 program is bug-free. Additionally, Vera can check user-provided functional properties and thus prove the box conforms to a user-specification. We rely on NetCTL, a novel specification language, to allow user-specified policies.

Upon deployment, P4 programs are incomplete (they lack table entries). Vera can analyze such P4 programs by using symbolic table entries instead of requiring actual table entries. This uncovers all the possible processing behaviors of a P4 program. We use the results to ensure that there exist valid paths through the box (i.e. that yield correct outputs); if there aren't any then the specification cannot be implemented using the current P4 program, regardless of the actual table contents. Also, faulty paths provide the necessary constraints for table entries that uncover bugs in the P4 program: we report these to the user, which can either fix them or ensure that offending table rules are never installed.

At runtime, controllers will insert rules in the P4 program, and these must be checked against the policy of the network (which involves network-wide verification). Here, the time available for verification is constrained. Vera introduces a novel data structure that concurrently optimizes both update time and verification time.

We have used Vera to analyze all public the P4 programs we could find including the P4 tutorials, a load balancer (Beamer [25]), a packet-trimming switch (NDP [12]), P4xos, an implementation of Paxos, and the complete datacenter switch implementation provided by p4.org. In seconds, Vera has found bugs in each of these programs, with little or no specification effort on our side.

2 BACKGROUND AND MOTIVATION

An example P4 program is shown in Figure 1 and has a few main parts: the parser, the ingress pipeline, the egress pipeline and the deparser. The parser transforms the packet from bits into headers according to a parser specification provided by the programmer (see Fig. 2). The parser specification also dictates how packets are *deparsed* from separate headers into a bit representation on output. After input parsing, a ingress control function decides how the packet will

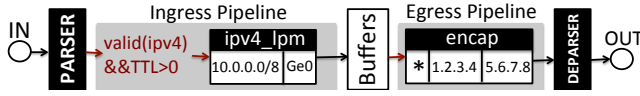


Figure 1: ENCAP: P4 that encapsulates IP packets.

be processed; the control instructions in our example are shown in red. Then, the packet is queued for egress processing. Upon dequeuing, it is processed by an egress control function and then it leaves the switch.

Both ingress and egress control functions direct the packet through any number of match-action tables. The control functions conditionally steer the packet to various tables based on header values, metadata or table match outcomes; packet contents cannot be changed in the control function. In our example, there is one table used on input, `ipv4_lpm`, and one table on output called `encap`. In the ingress control function, packets with valid IPv4 headers and strictly positive TTL values are sent to the `ipv4_lpm` table; on egress, all packets are matched against the `encap` table.

Match-action tables are where the bulk of packet processing takes place. The P4 program defines which fields will be matched in any given table; for instance, the `ipv4_lpm` table matches the destination address field in the IPv4 header. The actual table rules are provided at runtime by a controller or statically, when the P4 is deployed; in our example, there is a single rule for prefix `10.0.0.0/8`.

When a packet visits a table, it is matched against the existing table rules and upon a match the action associated with the rule is executed. Actions can change packet contents (modify fields, add or remove headers) or metadata, drop or clone the packet. For instance, when a packet destined to `10.0.1.1` visits the `ipv4_lpm` table, it will match the `10/8` rule and execute the associated action. This action is not shown in the example, but it sets the `egress_spec` metadata to that of interface “`Geo`”, among other changes.

After table matching, packet processing resumes in the control function where the outcome of the match—which action was executed, if any—can be used to decide how the packet will be processed next.

After ingress processing, if the packet is not dropped, it is placed in one of the queues for the egress interface based on the `egress_spec` metadata (this and other implicit edges are shown as black arrows in Fig.1). It then continues to egress processing, as dictated by the egress control function. In Fig. 1 the packet will visit the `encap` table where all packets execute the `ipip_encap` action (code in Fig. 3) that adds an inner IP header, copies the outer header to the inner header, and then modifies the outer header with addresses given as parameters. On egress, other processing includes changing the ethernet addresses and computing checksums.

Debugging P4 programs is hard. Despite its apparent sim-

```

parser start {
  extract(eth);
  return select(eth.type){
    0x800 : parse_ipv4;
    default: ingress;
  }
}
parser parse_ipv4 {
  extract(ipv4);
  return ingress;
}

action ipip_encap(srcIP,dstIP) {
  add_header(inner);
  //copy outer to inner header
  copy_header(inner, ipv4);
  //set outer header addr. and proto
  modify_field(ipv4.src, srcIP);
  modify_field(ipv4.dst, dstIP);
  modify_field(ipv4.proto, 0x5E);
}

```

Figure 2: Parser.

Figure 3: Encap action.

plicity, programming P4 is often counter-intuitive and unexpectedly difficult, and debugging P4 programs is particularly challenging. In traditional programming, hardware traps interrupt the program when a critical error such as unmapped memory access or division by zero is attempted, and debuggers can quickly track the source of such errors. Unfortunately, this is not the case with P4 programmable switches: when present, errors are handled silently at runtime: the packet triggering the offending behavior is either dropped or modified in unspecified ways; in both cases, tracking the location of the bug is very difficult, even when using the software P4 switch (the behavioral mode switch). Loops are also very difficult to catch: a single looping packet will slightly decrease the throughput; only when the pipeline fills with looping packets the throughput will collapse and the effects will be easily visible externally.

We have discovered a variety of bugs in the P4 programs we have access to. The most common mistakes seem to be parser bugs, invalid header accesses and encapsulation bugs, but we have also seen out-of-bounds register accesses and loops caused by recirculated packets. Most of these bugs are facilitated by traits of the P4 language that we discuss next.

An overarching problem of P4 is that it is not memory-safe. When an uninitialized header field is read, the switch behavior is unspecified; the packet could be dropped, or a quasi-random value returned. Since the switch does not throw any runtime exceptions, catching and fixing such bugs, especially if they are triggered rarely, is a nightmare.

Another fundamental problem is that P4 programs are underspecified. First, functionality is split between the P4 program and the match-action rules inserted at runtime. At compile time only the program is available and, sometimes, a few static rules that will be inserted in the P4 tables at startup. The runtime rules are unknown as they will be generated and inserted by the control plane, a separate piece of software. The programmer thus has to understand the behavior of a seemingly underspecified program, which is far from easy.

Secondly, a lot of P4 processing is performed implicitly, without being explicitly requested by the programmer. In our example, packets without a valid IPv4 header, or with zero TTL will arrive at the buffers without the `egress_spec` metadata value set, and will be implicitly dropped. Such implicit drop behavior seems convenient, but it can result in

the program dropping useful packets and is difficult to debug. To avoid such errors, the programmer should explicitly drop unwanted packets instead, but reasoning about all such packets is difficult at compile time.

Thirdly, dropped packets in the ingress pipeline continue match-action processing (because of the difficulty of removing packets from the pipeline), and they may match entries in downstream tables. Such behavior may lead to errors where packets dropped by one table (e.g. `acl`) are later revived unintentionally by another table (e.g. `lpm`).

Another peculiarity that is unique to P4 is that parsing must account for all header layouts the P4 program can accept on input *and* emit on output, despite the fact that the code is written in terms of ingress parsing (see Fig. 2). A common mistake, also present in our example, is when the header layout of outgoing packets is not captured in the parser spec: in our case, we do not parse the inner IP header. At deployment, this bug will make our program output packets that only contain the outer (encapsulation) header. Finding the root cause of the bug is quite difficult, but fixing it is easy enough. For this, the `parse_ipv4` code in Fig.2 could be replaced by:

```
parser parse_ipv4 {
  extract(ipv4);
  select (ipv4.protocol){
    0x5E: parse_inner_ipv4;
    default: ingress;}
  parser parse_inner_ipv4 {
    extract(inner_ipv4);
    return ingress;}
```

The new code correctly *deparses* encapsulated packets, but has an unintended consequence: incoming IP-in-IP packets will have their inner header overwritten: in the encapsulation action, when a new inner header is added and its fields modified, they will overwrite the existing inner header. We can fix this bug by checking that the inner header is invalid before encapsulation, or by using a header array instead.

`encap` shows how easy it is to make mistakes even in very simple P4 programs. Our analysis has found bugs in all the P4 tutorials (found at <https://p4.org/code/>), despite their simplicity and reduced size. As runtime debugging of P4 programs is tricky, developing even simple, correct P4 programs is a challenging task.

Verification approaches. If P4 is to meet its goal of enabling programmable networks that replace today's reliable, ossified ones, we must ensure P4 programs are easy to debug and fix. An ambitious goal is to *catch all bugs at compile time*.

In verification, there is a fundamental trade-off between specification effort and verification complexity. Iterative design and specification approaches such as Cocoon [27] require massive input from the programmer/verifier but are feasible computationally and guarantee that the generated code matches the specification. As network processing changes quickly, such approaches are both unlikely to keep up and

will be too expensive to use in most networks.

At the other end of the spectrum, we can use testing and simply inject all the packets that the program's parser will accept and then check if their processing leads to problematic behavior. Such testing requires almost zero specification and it covers all possible behaviors, but will scale poorly.

Symbolic execution is an excellent middle-ground: it can scalably explore the processing of all possible packets and does not require programmer input for verification. For traditional programs, symbolic execution offers much better code coverage compared to testing [3], but it rarely explores all possible paths, and thus it is not exhaustive—it does not guarantee absence of bugs. Network dataplanes are however much simpler than standard C code, e.g. they do not include loops. Exhaustive dataplane symbolic execution is feasible for moderate sized enterprise networks [28, 8].

3 VERA: SYMBOLIC EXECUTION FOR P4

We present Vera, a verification tool that uses symbolic execution to test the behavior of P4 programs for all possible packets. To our knowledge, Vera is the first automated verification tool for P4. To run Vera, the user passes as arguments the name of the P4 source file together with a set of commands that insert table rules at program startup. Vera first translates the P4 program together with the provided table rules to the SEFL language, obtaining an equivalent program (see §3.2 for details). The resulting SEFL program is a collection of virtual ports, each with associated SEFL instructions.

A sketch of the SEFL version of the `encap` program is given in Figure 4, where edges denote how packets may flow through the program. Snippets of SEFL code generated by Vera for `encap` are given in Figures 6 and 7.

Vera examines the P4 parser state machine and generates one symbolic packet for each header layout that can be accepted by the program; in the `encap` example, it will generate two possible packets: one containing an ethernet header and another one containing an ethernet header followed by an IP header. The symbolic packets have all their fields set to symbolic values, meaning they take any value in their domain. Vera injects these packets into the input port of the program and uses Symnet [28] for symbolic execution.

In Symnet, one execution path represents one symbolic packet and its associated metadata, possibly with constraints for the header or metadata values (we use packet and path interchangeably). As long as it has one or more active packets, Symnet selects one of them and executes the next SEFL instruction for that packet. All packets are processed until completion. Completed packets can either be failed or successful. Failed packets were either dropped or have triggered a bug and were terminated by Symnet. In successful packets all fields have feasible constraints or concrete values, and

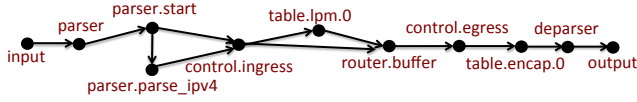


Figure 4: Ports generated by Vera for Encap P4.

there are no more instructions to execute—for instance, when they reach the output port.

When symbolic execution finishes, the output is a list of packets. For each packet we know if it is failed or successful, the ports it has visited, the instructions it has run, and the state of each header field and metadata: concrete values or constraints for the symbolic header fields.

In Fig. 5 we show the output of Vera when injecting an ethernet/IP packet in the encap program. The initial packet is shown as a white box. Path constraints are shown as annotations on the different edges. All final states are shown in red and represent failed packets. Vera finds four different packets; each of these takes a different path from the input to the output states in Fig. 5. Three of these paths result in implicit packet dropping in the buffering mechanism: when `eth.type` is set to a value different from `0x800`, when `ipv4.TTL` is zero or less, and when the destination address does not match `10/8`. A single packet makes it to the deparser; its constraints are `eth.type=0x800`, `ipv4.TTL>0` and `ipv4.dst` matches `10/8`. Despite this, the packet fails in the deparsing stage because the inner IPv4 does not exist in the parser specification.

In the rest of this section we describe in more detail why translating P4 to SEFL is the right choice (§3.1), how the translation is performed (§3.2), how we can deal with missing table entries (§3.3) and how symbolic execution can besped up by smartly generating the match-action table code (§3.4).

3.1 Why Symnet/SEFL?

In principle, any symbolic execution engine can be used with similar results. We chose SEFL/Symnet because it has been optimized for network dataplanes, showing superior performance to llvm/Klee [28]. Secondly, Symnet offers a memory model very similar to that of P4 (packet headers and metadata) and, in addition, it offers memory safety—unallocated or misaligned header accesses are automatically caught, as are header overlaps. These traits considerably simplify bug catching during symbolic execution, allowing us to focus most of our effort a correct translation from P4 to SEFL. Finally, there already exists a wide range of compilers that take FIB snapshots, Click modular router configurations and output SEFL, meaning we can integrate our P4 models in larger legacy networks and perform network-wide dataplane verification without any added cost.

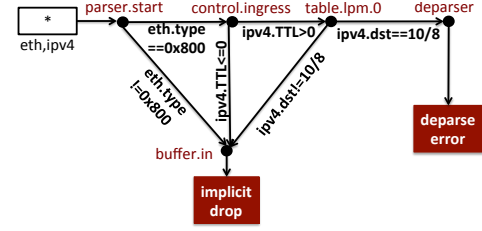


Figure 5: Symbolic execution of encap finds four failed paths for ethernet/ip packets.

3.2 Translating P4 to SEFL

The expressive power of P4 is very similar to that of SEFL. This is perhaps not surprising as both languages have been designed to capture data plane processing, albeit for different end goals: SEFL aims to enable cheap verification while P4 aims to be easily deployable in hardware. Neither SEFL nor P4 have loop instructions.

SEFL allows creating any number of named ports, which have associated SEFL instructions. The packet gets directed to these ports explicitly, by using the forward instruction, or implicitly, via directional links where each link connects two ports. Our parser uses ports to preserve the layout of the P4 program in the SEFL program it outputs.

SEFL has two types of variables: packet headers and metadata. Metadata are key/value pairs where the keys are strings and values can take any type. Packet headers are allocated in a linear address space, and all fields have absolute starting addresses. SEFL offers memory safety: any access to unallocated metadata or packet header fields terminate the current execution path. Additionally, packet header accesses must always be aligned to be allowed, and field allocation ensures neighboring fields do not overlap. P4 also has two types of variables: metadata, which we naturally map to SEFL metadata, and packet header fields.

When a packet enters a SEFL-P4 program, it is encoded as a series of successive header fields (as it is in practice). Our translation for the P4 parser code generates one port for each path in the parse tree. The packet is directed from input to the `parser.start` port. The current parse location in the header is remembered as a SEFL tag which is a pointer to a location in the packet. When the packet enters the P4 box, we create the current tag as follows: `CreateTag(current, START)`; `START` is a tag maintained by SEFL that points to the beginning of the packet.

A known problem with any verification approach is that checksums cannot be verified when header fields are symbolic; however this verification is a crucial part of the parsing process. Instead, we validate headers by checking that header fields are allocated at the right locations before we extract a P4 header, as suggested by Symnet [28]. In particular, to implement the `extract` call, we generate one check for each

```

parser.parser_start:
exists(current,48);
exists(current+48,48);
exists(current+96,16);
eth.src = @current;
eth.dst = @(current+48);
eth.type = @(current+96);
current += 112;
valid.eth = 1;
If (eth.type==0x800)
    Forward(extract_ipv4);
Else
    Forward(control_ingress);

parser.parse_ipv4:
exists(current,4);
...
exists(current+128,32);
ipv4.version = @current;
...
ipv4.dst = @(current+128);
current += 160;
valid.ipv4 = 1;
Forward(control_ingress);

```

Figure 6: Generated code for the P4 parser.

header field to be extracted (see Fig. 6). For this, we use the `exists` instruction in SEFL that checks for an allocated variable of given size at a certain position in the header.

If the packet header does not match the expected layout, one of the `exists` functions will fail, and the error will be logged. If all the header fields exist, the parser implementation in SEFL then proceeds to create a SEFL metadata value for each parsed header field; the name of the metadata is `header_instance.field_name`. When parsing is conditional, constraints are added as shown in the example for the `eth.type` field. After parsing has finished, all parsed headers are available as SEFL metadata. In addition, the SEFL code initializes P4 metadata and other information from the parser (e.g. which headers are valid), and forwards the packet to the `control_ingress` port that contains the SEFL code for the ingress control function. Translating control flow instructions is straightforward, as there is a one-to-one mapping between SEFL and P4 instructions here. We show the code for the ingress pipeline of `encap` in Fig.7.

To translate match-action table processing, Vera generates a new port for each `apply` call and associates with it the SEFL code implementing match-action processing (e.g. `table.lpm.0` in Fig.7). The port name is guaranteed to be unique as it is formed by concatenating the table name and a per table sequence number. The parser also creates a new port in the control function (`control_ingress.1` in our example), where processing will resume after table processing. To execute an `apply` call, we insert in the control function a forward to the respective table invocation port. Table processing will finish with a forward to the control function. Vera directly translates all P4 primitive actions to SEFL.

Registers are arrays of predefined *size*, and translating them is tricky because SEFL does not have array support for metadata. Vera creates one metadata for each array entry; the name of each metadata includes its location in the array (e.g. the variable `a[0]` holds the first value in the array, `a[1]` the second, and so forth). To access a register value, we insert a series of `if/else` instructions that successively test the value of the index at runtime against all possible locations, and then access the correct array location. While inefficient,

```

control.ingress:
If (valid.ipv4==1&&ipv4.ttl>0)
    Forward(table.lpm.0);
Else
    Forward(buffer.in);

control.ingress_1:
Forward(buffer.in);

table.lpm.0:
If (ipv4.dst match 10/8){
    lpm.0.Hit = 1;
    lpm.0.action.lpm = 1;
    ipv4.ttl--;
    meta.egress_spec = 1;
} Else lpm.0.Hit = 0;
Forward(control_ingress.1);

```

Figure 7: Generated code for the ingress pipeline.

this solution does not increase the number of explored paths when the index is concrete.

The various clone actions are implemented using the `fork` instruction which creates a new execution path that is a copy of the current path. On the cloned path the `instance_type` metadata is set to signal that this is a cloned packet and the packet is then redirected according to the specific clone instruction either to buffering or parser input. Packet redirection actions such as `resubmit` or `recirculate` are implemented by forward to the parser input port.

The drop action, when applied on ingress, is implemented by setting the `egress_spec` to a predefined value (511). The packet is only dropped when it reaches the buffering mechanism, as per the P4 spec. In the egress pipeline, drop is implemented using the `fail` instruction that terminates the current path and prints an error message.

Whenever a header instance must be added or removed, our SEFL code first checks that the header is valid (for removal) or invalid (for addition), throwing an exception otherwise. Creating the header instance is easy: we generate SEFL code to allocate SEFL metadata for each field; the name will be `instance.field`, and is guaranteed to be unique by the P4 compiler. Copying headers is similarly trivial to translate.

Dealing with header arrays, however, is a bit trickier because SEFL does not have support for arrays. To circumvent this problem we have implemented a working but inefficient solution that relies on the fact that header arrays have a predefined size and additions/removals are always done at locations known at compile time. Below is the implementation for `add_header(ip[0])`¹ for an array of maximum two headers: the code simply treats all possibilities, in parallel. The first path will succeed if there is no header in the header array yet, otherwise it will silently fail. The second path will only succeed when the first header is valid but the second one is not, and the third will throw an exception when both headers are allocated and adding a third is not possible.

```

Fork {
  Path1: If (!Exists(ip[0])) { ip[0].valid = 1; }
  Path2: If (Exists(ip[0]) && !Exists(ip[1])) {
    ip[1].valid = 1; copy_header(ip[1], ip[0]);
    forall f in fields(ip[0]):
      ip[0].f = 0;
  }
  Path3: If (Exists(ip[0]) && Exists(ip[1])) {
    Fail("Attempting to add header in a full array")
  }
}

```

¹the implementations for `remove_header`, `push` and `pop` are similar.

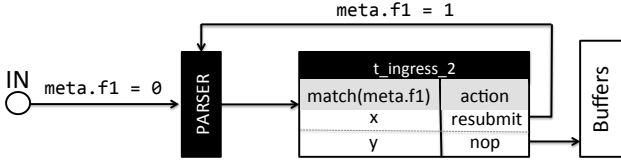


Figure 8: Using symbolic table entries to analyze the Resubmit P4 tutorial.

While inefficient, this solution works well enough for the examples we tested, mostly because header array sizes used in practice are fairly small. A more elegant solution requires array support in SEFL—we leave this to future work.

Our implementation of the buffering mechanism simply checks for an appropriate egress spec before deciding whether to forward the packet to the egress pipeline or to drop it.

The truncate action takes length as parameter and works at deparsing, only emitting the specified number of bytes. SEFL will throw an exception when the packet boundary falls in the middle of a valid header field.

The final step in the pipeline, deparsing, is the inverse process of parsing. First, the SEFL program searches for a path through the topologically sorted parse tree which matches valid header fields. If such a path is not found, a deparsing exception is raised; otherwise, the P4 spec guarantees that a single path is possible, and the code simply copies the metadata to the packet layout, before releasing the packet to the specified egress interface. The deparser also raises an error when it finds an extra valid header that does not match the selected parse tree path. This catches a common class of deparser bugs, such as those discovered in Beamer (see §5).

3.3 Symbolic table entries

Vera provides exhaustive program analysis for any given table rule snapshot, but such snapshots do not cover all possible rules that may be inserted; worse, at compile time the only available rules are static rules that help test common-case functionality. Unfortunately, a P4 program that is deemed correct for any single snapshot of its match-action tables, may not be correct for other table entries.

To enable exploring a larger space of possible table snapshots, Vera allows the programmer to insert symbolic entries in match-action rules for both field matches and action parameters. A symbolic rule uses the same format as its concrete counterpart with the difference that both the match entries and action parameters can take symbolic values.

Consider the example in Fig. 8 where we show a part of the ingress pipeline of the Resubmit P4 example application. The match criterion for the table `t_ingress_2` is the metadata field `meta.f1`. The corresponding action can either do nothing (sending the packet to buffering) or resubmit the packet to the parser, also modifying `meta.f1` to 1.

To better understand the behavior of this program, the programmer has added two symbolic rules in the table, one for each possible action, and `x` and `y` are the respective symbolic match values. When exploring this P4 program, Vera will treat `x` and `y` as any other symbolic variable, collecting constraints and checking their feasibility. The two correct paths correspond to the packet directly hitting the nop action ($x \neq 0, y == 0$), or being resubmitted and hitting the nop action ($x == 0, y \neq 1$). Vera also finds two faults, both corresponding to cases where the packet is implicitly dropped because it doesn't match any rule; the constraints for the two faulty paths are $\{x \neq 0, y \neq 0\}$ and $\{x == 0, y \neq 1\}$ respectively.

Using this information, the developer of the controller program has valuable insights regarding which rules can break the intended functionality for the `t_ingress_2` table, and is assured that its code does not loop for *any* values of `x` and `y` for the existing rules. Note that this does not mean the program can never loop, regardless of its table entries. If we add one more symbolic rule with field `z` and action `resubmit`, Vera finds a loop where $x == 0, z == 1$ and $y \neq 0, 1$.

Symbolic table entries allow the programmer to explore a wide range of dataplane behaviors and to reason about the actions the controller must take without verifying the controller itself (such verification is much more difficult because the controller is a general-purpose program).

In our example, we have manually inserted table entries. How should we insert such rules to explore all possible dataplane behaviors for generic P4 programs?

To answer this question, first consider a program that does not *recirculate packets* and contains a single table T_1 that has n possible actions a_1, a_2, \dots, a_n . We can prove that it suffices to add n symbolic entries, one corresponding to each action, to explore all possible behavior (see Appendix). If the program has another table T_2 , we can apply the same rule: one symbolic entry per action will explore all possible behaviors. In fact, as long as we keep adding different tables and there is no recirculation, adding a symbolic entry per action per table is guaranteed to explore all possible behaviors. To achieve the same goal with recirculation, we need to add one additional symbolic rule for each recirculation.

Symbolic table entries can produce all the possible behaviors of an arbitrary P4 pipeline, without knowledge about the controller behavior. The resulting constraints on the symbolic rules can pinpoint what concrete table entries will reproduce the behavior captured by the Vera. Details and proofs can be found in the Appendix.

3.4 Fast verification of match-action tables

The match code for match-action processing is easy to translate into a series of If/Else SEFL instructions. The resulting code will have at least as many If instructions as table

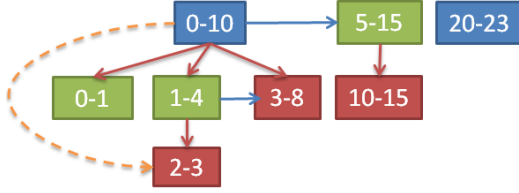


Figure 9: Match condition forest

entries, and this will make symbolic execution explore a separate execution path for each table entry. With enough table entries, this approach will render symbolic execution infeasible to use.

To ensure symbolic execution actually scales to large P4 programs, we need to optimize the match code while preserving its functionality. There are two general directions of optimization: a) reducing the number of paths explored by symbolic execution and b) reducing the number of constraints that need to be checked by the solver on each path.

The first part is fairly easy: we generate exactly one path for each distinct action invocation in the table rules. For an ACL table that has two actions (nop and drop), we will generate the following code:

```
Fork {
  Path1(nop): Constrain(...);
  Path2(drop): Constrain(...); drop();
}
```

For routing, the forward action has a parameter that specifies the output interface. In this case, our code will group all forward actions that have the same parameter into a single path, generating in effect one path for each output interface.

The second part of code generation is to ensure the path constraints are correct. This step is not trivial as the code must not only include the constraints for the associated rules, but also negated constraints for higher priority rules. As an example, the default route should forward a packet only when no other forwarding rule matches; the constraints in this case must include the negation of all other forwarding rules which have higher priority. From a table-wide perspective, if we have many overlaps, the worst case number of constraints is quadratic in the number of entries. With FIB sizes in the order of 100K, this is a show-stopper.

Our solution builds upon existing work [18, 4] and is applicable to a wide range of matching strategies including longest-prefix match, range, etc. At the core of our solution lies a data structure consisting of a forest of trees. To ease presentation, we show an example that matches a single field in Fig.9. In the figure, one node represents the match condition for one table rule and the colors represent rule priority (red>green>blue). In this data structure any pair of nodes falls in one of the four situations below:

- (1) The nodes are completely **independent** if their corresponding conditions do not overlap.
- (2) **Parent-of** - a node is the parent of another if the value domain corresponding to the child is strictly a subset of

the one for the parent (red links). The parent must have lower priority than the child.

- (3) **Ancestor-of**(dotted line) - when two nodes are connected in the same tree by several 'parent-of' links.
- (4) **Neighbor-of**(blue) - nodes that have overlapping conditions, but neither condition is fully contained by the other; and neither node has an ancestor linked to the other node via a 'neighbor-of' link. As with 'parent-of' links, the source has priority lower than the destination.

To add one node to our data structure, we start at the top of the forest, checking which nodes overlap with the new one (we implement this efficiently using interval trees). If there is no overlap, we add the new node as a standalone one. Otherwise, the new node will become either a parent, a child or a neighboring node. If it becomes a parent or neighbor node, the node is inserted at the current level and the appropriate links are created. If it is a child, then the algorithm continues recursively in the subtree rooted at its newly found ancestor. Complexity is logarithmic in the number of nodes.

Given this forest, it is trivial to construct the minimal constraint required to match any given node: add the node's condition and the negated constraints of all its children and neighbors. We have proven that our algorithm generates the theoretical minimum number of constraints. Intuitively, this is because we only add the negated constraints of the direct children and neighbors, and not those of ancestors. In figure 9 red nodes have the highest match priority, then green and lastly blue ones. At first, node [0-10] should add negated constraints for all its sub nodes, plus all the nodes in the tree rooted at [5-15]. Looking closer, negated constraints for [10-15] and [2-3] are redundant since the constraints corresponding to their 'parent' nodes mitigate the overlap.

We implemented the forest construction algorithm in its most general form, that can handle range, longest-prefix and wild card matching and analyze its scalability in §5.

3.5 Translator correctness

Guaranteeing correctness of code translation is a major precondition for correct software. In our work correctness is based on the formalization of P4 and SEFL semantics and the proof of semantic preservation by the translation process. We choose to use the "big-step" operational semantics, for both P4 and SEFL statements. The semantics defines a relation of the form $\langle S, s \rangle \rightarrow s'$ where the pre-state s and post-state s' represent the states before and after execution of the program statement S . The definition of \rightarrow is given by the associated semantic rules. The proof of semantic preservation relies on the semantic equivalence of the statements. We consider two statements to be semantically equivalent if for all states s and s' $\langle S_1, s \rangle \rightarrow s'$ if and only if $\langle S_2, s \rangle \rightarrow s'$. We developed the "big-step" operational semantics for P4

and SEFL and showed the semantic preservation by proving operational correspondence between the two semantics. A specification of the semantics of the relevant statements of P4 and SEFL together with the proof of the semantic preservation can be found in the Appendix.

4 DEBUGGING P4

By default, Vera inserts checks that capture a wide range of bugs in P4 programs and flags such bugs to the user as failed paths. For each failed path, Vera also generates a concrete packet that matches the path constraints which can be used to test the bug in P4 switches, be they software or hardware. Below are the types of errors that Vera catches automatically:

- **Implicit drops** are flagged when a packet reaches the buffering mechanism without having an `egress_spec` set. Vera catches this by adding an assertion that the `egress_spec` must be non-zero when it reaches the `buffer.in_port`.
- **Table rules that match dropped packets** are flagged as errors by adding an assertion that `egress_spec` \neq 511 in the preamble of all actions.
- **Invalid memory accesses** are frequent P4 mistakes, when users do not test the validity of a header before using its fields. Vera relies on Symnet's memory safety guarantees to capture these errors; when accessing an unallocated field, Symnet will fail the current path.
- **Header errors** Malformed headers are captured during parsing by using the `exists SEFL` instruction. Adding an existing header or removing an inexistent one are also caught automatically as deparsing errors.
- **Scoping and unallowed writes** Certain metadata values are read-only in P4, yet the P4 compiler allows the program to write them (e.g. the `egress_port` metadata). Further, static registers can only be read from one table according to the spec, yet the compiler allows such reads. Vera catches such errors during the translation process and reports them to the user.
- **Out-of-bounds array accesses** are caught automatically by Vera by adding, before each array access, an out-of-bounds check for the index. At runtime, the solver will check if the constraint is satisfiable and if it is the user will get a failed path providing an example packet that triggers a possible out-of-bounds access.
- **Field overflows/underflows** are the only arithmetic exceptions possible in P4 (because division is not supported) and Vera catches them by adding a check before each addition/subtraction operation.

Loops are also caught automatically. The loop detector runs by default on the parser input port and on the egress input port which are the two places where packets can be redirected backwards in the P4 pipeline. Whenever a packet enters visits one of these ports, Vera remembers the entire

memory state (i.e. the values and constraints or all the metadata and header fields). When a packet revisits the same port, its memory state is compared to *all* the previous saved memory states. Two memory states are different iff at least one symbol has a different value in the two states. Note that we compare not only concrete values, but also symbolic ones: if a metadata is bound to the same symbolic value in both states, it is deemed to be equal. Whenever Vera discovers two memory states that are equal, it fails the current path with the "loop detected" message.

We note that even if a P4 program does not have any of the bugs above, it may not do the job it is supposed to. For instance, a router that explicitly drops all packets it receives is bug free but it also doesn't do anything useful. It is therefore important to also be able to reason about the **correctness** of the P4 program. The correctness properties each box has to conform to depend on its intended use, and differ among different network boxes. In the next section we describe a specification language that enables specifying a wide-range of correctness properties for network boxes and explain how Vera automatically verifies whether these properties hold.

4.1 Correctness verification with NetCTL

For any given P4 program, Vera will explore a large number of paths, many of which are successful. In our evaluation, we typically see hundreds such paths. Examining them manually to decide whether the behavior is correct is time consuming and error-prone. We wish to specify desirable properties and have Vera check them automatically.

The specification must combine *packet constraints at specific ports* of the P4 switch (or *state properties*) with *constraints over the possible paths* which the packets may take between ports (or *path properties*). We can already express state properties via SEFL instructions. For instance, the property '*destination IP is always x at port out*' can be verified by placing the SEFL instruction `Dest-IP := x at port out` and observing the successful paths from port out.

In order to express path properties, we have considered a wide range of SDN policy languages, e.g. the Kinetic[19] family, FatTire[26], NetPlumber[16], as well as approaches relying on logic programming (e.g. Shenker's FML[14]). We have found that all such languages are limited in their ability to express compositional constraints.

We have thus turned to Computation Tree Logic (CTL) [6]. In CTL, temporal operators such as **F** (i.e. sometime in the future) and **G** (i.e. always in the future) are combined with path quantifiers: \exists (on some path) and \forall (on all paths). For instance, the policy: $\forall F_{\text{destTCP}} == 80$ evaluated at some port *P* of a box, expresses that on all possible packet paths from *P*, `destTCP` will eventually become 80.

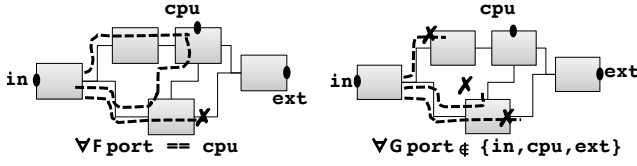


Figure 10: NetCTL example

The syntax of NetCTL is given below:

$$\varphi ::= SEFL \mid \neg\varphi \mid \varphi \wedge \varphi \mid XY\varphi$$

where $X \in \{\exists, \forall\}$, $Y \in \{\mathbf{F}, \mathbf{G}\}$.

Unlike Merlin, FatTree or NetPlumber, NetCTL is compositional: starting from simple properties, we can construct more complex ones. For instance, we can express that “*when-ever the IP destination of a packet becomes a public address, port P is reachable*” via the formula: $\forall \mathbf{G}(\text{ip} \neq 192.168.0.0/16 \rightarrow \exists \mathbf{F} \text{port} == \text{Internet})$. CTL can express many other properties including TCP connectivity and invariance across tunnels.

Checking NetCTL with Vera. In principle, checking NetCTL formulae can easily be implemented *after* exhaustive symbolic execution by checking the outputs. However, this approach is also inefficient because in many cases we can check a property without exploring all possible paths.

That is why Vera integrates NetCTL verification with symbolic execution. In our implementation, NetCTL verification is performed as added checks on the packets after each SEFL instruction is executed; the overhead of these checks is very small in practice. After every check we can decide to prioritize a certain path or stop execution altogether. Because of this, in most cases, Vera checks NetCTL properties faster than exhaustive symbolic execution (see §5).

In Fig 10, we briefly illustrate NetCTL verification. The figure describes two symbolic execution traces performed on the same topology — a simplistic illustration of the P4 NAT model, described in more detail in the subsequent sections. Boxes represent SEFL code blocks and solid lines — links between boxes. Dashed lines describe the paths explored by our verifier. The formula $\varphi_1 = \forall \mathbf{F}(\text{port} == \text{cpu})$ (left) expresses that all paths eventually reach port cpu. In order to evaluate it, our checker performs symbolic execution starting at port in. The checker will explore each encountered path until $\text{port} == \text{cpu}$ is satisfied, the path ends, or it becomes unsatisfiable. Suppose the checker explores three paths, as shown in the figure. Since the formula $\mathbf{F}(\text{port} == \text{cpu})$ is true on the first two paths only, φ_1 is false.

The formula $\varphi_2 = \forall \mathbf{G}(\text{port} \notin \{\text{in}, \text{cpu}, \text{ext}\})$ (right) expresses that all packets are dropped by the NAT. To verify it, our checker will determine if $\text{port} \notin \{\text{in}, \text{cpu}, \text{ext}\}$ is true on *each* execution path, and after each SEFL code-block. In our example, this is indeed the case, thus the policy is true.

5 EVALUATION

In our evaluation we seek to understand the coverage Vera provides and its scalability to large P4 programs (LOC) as well as large match-action tables. Tests were run on a server with a quad-core i5 processor and 8GB of RAM.

We discuss the bugs we found in a series of available P4 programs in §5.1. We then use NetCTL to express the correctness properties of a NAT and verify whether the simple NAT tutorial has these properties in §5.2. Finally, we examine how verification time scales with the number of rules in match-action tables in §5.3.

5.1 Bugs caught

We have used Vera to examine all P4 programs we could find publicly, but we do not claim our evaluation is exhaustive in any way. We ran tests in two ways: first, with command files supplied by the authors and using symbolic entries.

A summary of results is shown in Table 11 where the programs without a citation are from the official P4 codebase. We list the size of each P4 program, the time it takes Vera to exhaustively verify it, and the bugs we have found. Vera has found multiple bugs in each program we have examined. When using concrete table entries, the verification time ranges from 1s for all possible packets for toy programs (P4 tutorial) to around 15 seconds for `switch.p4` for one symbolic packet. With symbolic table entries, the runtime for all programs except the switch is under 10s; for the switch runtime increases to 30 mins to check one symbolic packet.

The severity of the bugs we found differs: some bugs are critical and will impact heavily the operation of the program (for instance out-of-bounds accesses or deparsing errors) while others are more benign. For instance, implicit drops are present in many programs, but their severity is not as high so they can be considered “warnings”. One type of bug that we did not find in any program was arithmetic exceptions (underflows or overflows). This is because few fields are obtained using arithmetic operations, and when they do the code is correct (TTL appears most often).

We now discuss some of the more interesting bugs in more detail. The `copy-to-cpu` tutorial is meant to show how a packet can be forwarded to the controller; a 16 bit `cpu_header` is added to the packet and sent to the CPU. The program has a parser bug in the following piece of code:

```
return select(current(0, 64)) {
  0 : parse_cpu_header;
  default: parse_ethernet;
}
```

Vera found two bugs in this code. When the input packet contains a single ethernet header, if the first 64 bits are set to zero this the parser will try to extract the `cpu_header` which will fail. However, even when the packet contains a

Program	Size (LOC)	Verification time (sec)	Implicit drop	Parsing	Deparsing	Header ops.	Invalid access	Underflow / overflow	Loop	Processing dropped packets
copy-to-cpu	70	0.1		•		•				
resubmit	70	0.4							•	
encap	130	0.45	•		•	•				
simple router	145	0.55	•							
simple NAT	290	1.25	•			•				
simple router + ACL	200	0.8	•							•
Axon	100	14		•			•			
Switch	6000	5-15/sym.pkt.				•	•			
Beamer mux[25]	340	1.4	•		•		•			
NDP switch[12]	210	0.8					•			
P4xos[7]	650	13.4	•				•			

Figure 11: Bugs found by Vera in P4 programs available publicly.

cpu_header the code is wrong: the header is only 16bits, so the check will also include the ethernet source address. When the latter is not zero (most often), the parser will assume this packet is pure ethernet and try to extract the ethernet header, which will fail. In other tutorials which parse the cpu_header, the cpu header definition includes a 64bit preamble which must be zero; updating the header definition would also fix the copy-to-cpu example.

Beamer [25] is a load balancer: it takes packets, encapsulates them with an IP-IP header, and sends them to a backend server. In Beamer, Vera has found a typical deparsing bug. The exact encapsulation depends on the TCP destination port in packets: if the port is less than 1024, the output packet layout is *eth,ip,ipopt,ip*, and this is deparsed correctly. If the port is larger than 1024, the output packet does not include the *ipopt* header and this leads to a deparsing error because this packet is not correctly parsed by Beamer. When we shared our findings with the Beamer authors, they mentioned that their prototype also had a deparsing error on the first branch which they caught after some effort, but they had missed this bug that was not tested by their unit tests.

P4xos [7] is a P4 implementation of the Paxos protocol. Vera has found an out-of-bounds static register access in action read_round in the following instruction:

```
register_read(local_meta.round, rounds_register, paxos.inst);
```

The problem is that the *inst* header field can take any value, leading to faulty accesses (a *todo* in the code acks this issue).

Switch is the largest P4 program available today; it implements the full stack of protocols needed to operate a data-center top-of-rack switch and is thus a good benchmark for our verification tool. switch.p4 verification takes between 5 to 15 seconds *per packet type*, when all packets fields are made symbolic. As there are 60.000 possible header layouts given by the parser, total verification would take 170 hours (a week) on a single machine, but this can be easily parallelized.

We have analyzed, however, only tens of packet headers because after each run we need to manually check the outputs (two-three hundred paths, typically), sift through the failed paths, and decide which ones are novel bugs and which represent bugs we already know about. This process

is very time consuming. Exhaustive verification is therefore feasible computationally, but we need to design more tools to automatically interpret outputs; this is our future work.

Overall, the switch code is much cleaner than all the other examples we have looked at, reflecting the fact that this is production quality software. Below we discuss three of the more interesting bugs we found in the switch. The first bug is in the remove_vlan_double_tagged action which is triggered in the vlan_decap table:

```
remove_header(vlan_tag[0]);
remove_header(vlan_tag[1]);
```

The code above first removes the header at position 0 in the array. This makes all headers at higher indices shift their position to the left; in other words, *vlan_tag[1]* becomes *vlan_tag[0]*, and the second remove instruction fails silently. After the action we are left with one active header, instead of having both removed. Depending on packet processing downstream, this bug may result in outgoing packets having a vlan tag when they shouldn't have.

Another bug is accessing an invalid field in table *l3_rewrite* where both the ipv4 and ipv6 source addresses are matched, and they can't be valid simultaneously.

A third bug appears when the data-plane is configured to allow layer 3 VXLAN encapsulation/decapsulation. The switch correctly behaves when input an Ethernet/IP/TCP - i.e. it VXLAN encapsulates the frame. Now, assume a VXLAN encapsulated frame is input from the non-tunnel interface of the switch. The expectation is that frames be further encapsulated within a new VXLAN header, while keeping the existing one intact. However, the actual implementation of the reference switch has a single VXLAN header and the switch attempts to add a header which is already valid. Vera quickly discovers the offending operation, while providing important insights into the error location within the P4 program - i.e. match-action table trace, offending table name and match conditions.

5.2 Correctness verification of simple NAT

Vigor is a provably correct NAT implementation in C [29] that required an intense verification effort from networking

Policy	NetCTL formula and input port	Verification time and explored paths
(1) A NAT without entries drops all packets	$p : \forall G(\text{port} \notin \{\text{in}, \text{ex}, \text{cpu}\})$ where $p \in \{\text{in}, \text{cpu}\}$	708 paths in 2234 ms
(2) If only miss entries exist, in-packets reach the controller and ext-packets are dropped	in: $\forall F(\text{port} == \text{cpu})$ ex: $\forall G(\text{port} \notin \{\text{in}, \text{ex}, \text{cpu}\})$	354 paths in 1034 ms 285 paths in 1157ms
(3) With hit entries, matching in- and cpu-packets reach ext	$p : \forall F(\text{port} == \text{ex})$ where $p \in \{\text{in}, \text{cpu}\}$	232 paths in 816 ms
(4) A response to a in-packet reaches in	in: $\exists F(\text{port} == \text{in})$	113 paths in 1413 ms
(5) The NAT performs a correct IP mapping	in: $\forall G(\text{port} == \text{in}) \rightarrow (\text{ini_dstIP} == \text{srcIP})$	304 paths in 1913 ms

Figure 12: Verification of a P4 NAT

researchers. We want to check if the simple NAT implementation from P4 offers similar correctness guarantees. The first step is to express the properties a NAT should follow. All our verification rules specify (i) a port where the symbolic packet will be injected; (ii) init code: the header layout and other instrumentation to be performed the initial *symbolic* packet and (iii) the appropriate policy in the NetCTL language. The NAT defines an interior (in) and exterior port (ex), as well as a port for packets sent to the controller (cpu). We also use a meta-variable port which stores the current port during symbolic execution. The NAT table has *hit* and *miss* actions for the in/ext ports which handle the situation when mappings exist or don't exist for the current packet.

We list in Fig. 12 a subset of the policies we have specified to describe correct NAT behavior, in the form ' $\text{port} : \varphi$ ', where port is the input port, and φ is a NetCTL formula. We omit describing the init code in most cases, as it is less important for understanding our methodology.

The first policy requires that the NAT drops all packets if there are no table entries. The policy states: *on all execution paths, at no point does the packet reach any of the NAT output ports*. Vera confirms that this policy holds in 2.2s.

We have checked policies (2-5) with and without symbolic table entries, but present only results corresponding to symbolic entries because they have greater coverage and give insights into the correct behavior of the controller.

The second policy expresses that all packets from the input ports that do not match hit rules will reach the controller. To check this policy we add symbolic entries for the *miss* actions and Vera confirms the policy is true in around 1s.

The third policy verifies that the NAT translates packets before sending them to the output interface. To verify it, we insert symbolic entries for the hit actions and call Vera. Vera finds an example where this policy is violated: after a hit action is executed, the destination IP field is unconstrained and it can reach both in and ext during the routing phase. Concretely, this means that the NAT will also translate packets destined for the LAN, which is not intended.

In order to check policies (4-5), we have built a simple TCP responder in P4 which flips the IP and TCP source/destination

addresses. Policy (4) checks that the NAT enables for bidirectional connectivity. We start with symbolic entries for the hit action and Vera verifies that bidirectional connectivity exists, and the successful path shows how the table rules must look like: a *hit-int-to-ext* rule must exist which matches the packet's 5-tuple and has the *is-ext-if* fields set to false, and a *hit-ext-to-int* rule must exist which matches the translated packet's reverse 5-tuple, with the *is-ext-if* field set to true. Further, this rule also restricts to possible action parameters for the *hit-int-to-ext* rule: the *srcIP* must be the IP of the router's external interface.

Policy (5) further verifies that the translation works correctly. In order to verify it, in the init code we create a new variable *ini_dstIP* which stores the initial destination IP header field. Thus (5) expresses that whenever the packet reaches in, the current source IP field must be equal to the initial destination IP. The actual policy we verified checks the entire 5-tuple, not just the destination IP.

Taken together, our verification gives a clear controller spec. When a new connection arrives from the LAN, it will be sent to the controller (cf. policy 2) which *must* insert two hit rules in the NAT table (cf. policy 4) to enable translation; the contents of the rules are completely specified (except the source port). Finally, policy (3) specifies that, if we want the initial packet to be translated too, the controller *must* inject it *after* it inserts the *hit-int-to-ext* rule. Using these rules to develop a correct controller is our future work.

Vera partially explores the NAT model for policies 4-5. Verification stops as soon as a successful path is found, and this significantly reduces the number of execution paths (100-200). In contrast, unconstrained symbolic execution explores more than 2000 paths— twenty-times more—in 3.6s.

5.3 Scalability of match-action processing

All our examples so far were run with a few concrete or symbolic table entries. Here we want to examine the performance of our match-action algorithm at scale. We used the Stanford dataset [15] containing router FIBs of 180K entries and compared the Naive If/Else implementation with Vera and a SEFL model hand-optimized for routing from prior work [28]. We show the results in Table 1, highlighting the model generation time and the symbolic execution

Algorithm	Action	1K	10K	100K	180K
Naive	model create	2ms	5ms		
	sybex	34s	1 hour		
Hand-crafted[28]	model create	23ms	1.5s	218s	242s
	sybex	220ms	2.7s	12s	16s
Vera	model create	36ms	1.5s	83s	178s
	sybex	230ms	2.7s	12s	16s

Table 1: Match-action performance for IP forwarding.

time of the resulting P4 program. The naive model chokes at just 10K entries in the FIB, while Vera and the hand crafted model give good performance even for large routing tables containing 180K entries. Compared to [28], Vera is slightly faster in building the model, and is faster during model updates: it takes just 2.5ms to update a model with 100K entries, whereas the hand-crafted model takes around 100ms.

To test more complex match-action processing, we implemented and populated an ACL table that matches on TCP port ranges and the IP destination address (lpm) and has two actions: allow and drop. There is no prior work to compare against, so we are mainly interested in headline times. Also, we wanted to test the effect of grouping rules with similar actions into the same path. The results are shown in Table 2; the build time is the same for both versions of Vera. The results shows how reducing the number of paths dramatically reduces verification time (factor of 50x for 100K entries).

6 RELATED WORK

Network verification research mostly focuses on understanding whether network-wide properties such as reachability and isolation hold. Dataplane analysis tools such as HSA[15], NOD[22], Veriflow [18], Anteater[23] and Symnet[28] require a snapshot of the network dataplane, including the processing done by each box, links between boxes and forwarding rules, and test whether the desired end-to-end properties hold. Control plane verification aims to answer the same network-wide questions but without requiring the dataplane snapshot: tools like Arc[11], Batfish[9], Minesweeper[1] or CrystalNet [21] can predict how a control plane change will affect the dataplane, flagging property violations when they occur. Vera is complementary to this work: it generates SEFL models of P4 programs and these can be used in network-wide dataplane analysis with Symnet [28].

Verifying if the implementation of a networking element is correct is another area of research. Dobrescu et al. [8] use symbolic execution with S2e [5] to analyze the C++ implementation of Click modular router elements [20]. Vigor [29] is a formally verified NAT implementation written in C. Vera is complementary to these works: it can guarantee safety of P4 code, and in conjunction with NetCTL it can also prove the correctness of an implementation according to some specification (see our NAT example).

Algorithm	0.1K	1K	10K	100K
Build	6ms	36ms	1.6s	73s
No grouping	0.8s	9s	80s	651s
Group	0.05s	0.16s	1.5s	12s

Table 2: Performance for ACL processing.

Vera is not the first verification effort geared at P4 programs. The most mature existing work is that of Lopez et al. [24]: they translate P4 to the NOD language [22] and then perform end to end dataplane reachability tests, as well as testing a form of header bugs called “well-formedness”. NOD does not offer support for dynamic encapsulation, so P4NOD cannot capture many of the header errors that Vera can. Lopez et al. only examine two small programs, and do not find problems in the programs themselves, focusing on end to end verification instead.

Work in progress presented by Foster[10] proposes that programmers annotate P4 programs with Hoare logic clauses (pre and post conditions) to enable static verification. Their approach targets catching many of the bugs that Vera catches automatically, but it requires human specification which is cumbersome. Another work-in-progress by Rosu [17] has created an executable formal semantics for P4 that can be used to translate P4 into a format that can be verified with a number of tools including symbolic execution. This approach is similar in spirit to ours; the advantage of Vera is that it relies on SEFL which is designed specifically for network data planes, and thus enables more efficient verification.

Symbolic execution research has a long history, aiming to automatically find bugs in general-purpose programs. Tools like Klee[3] and S2e[5] are mature and can be readily used. Unfortunately, for most traditional programs symbolic execution does not explore all paths and can offer no guarantees that programs are bug-free. In contrast, Vera shows that using symbolic execution to exhaustively analyze P4 programs is feasible and can capture many bugs.

7 CONCLUSIONS

P4 promises to enable truly flexible networks that can adapt to application needs, but P4 programming is not as easy as it may seem at first sight due to language features stemming from its close relationship to switch hardware.

In this paper we have implemented Vera, a tool that uses symbolic execution to exhaustively verify even the largest P4 programs available today. To achieve such performance, Vera relies on a number of innovations including symbolic match-action entries, automatic policy verification and a novel match-action data structure. Together, these have helped Vera to catch many interesting bugs in all programs we have analyzed, with modest runtimes. Moreover, Vera can help derive controller specifications; we plan to enforce these specifications at runtime in future work to enforce correctness even when the controller is not formally verified.

REFERENCES

- [1] Ryan Beckett et al. "A General Approach to Network Configuration Verification". In: *SIGCOMM*. 2017.
- [2] Pat Bosshart et al. "P4: Programming Protocol-independent Packet Processors". In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014).
- [3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs". In: *Proc. OSDI'08*.
- [4] Marco Canini et al. "A NICE Way to Test Openflow Applications". In: *Proc. NSDI'12*.
- [5] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: A Platform for In-vivo Multi-path Analysis of Software Systems". In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: ACM, 2011, pp. 265–278. ISBN: 978-1-4503-0266-1. DOI: 10.1145/1950365.1950396. URL: <http://doi.acm.org/10.1145/1950365.1950396>.
- [6] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0-262-03270-8.
- [7] Huynh Tu Dang et al. "Paxos Made Switch-y". In: *SIGCOMM Comput. Commun. Rev.* 46.2 (May 2016).
- [8] Mihai Dobrescu and Katerina Argyraki. "Software Dataplane Verification". In: *Proc. NSDI'14*. NSDI'14.
- [9] Ari Fogel et al. "A General Approach to Network Configuration Analysis". In: *NSDI*. 2015.
- [10] Nate Foster. *A Program Logic for Automated P4 Verification*. 2017.
- [11] Aaron Gember-Jacobson et al. "Fast Control Plane Analysis Using an Abstract Representation". In: *SIGCOMM*. 2016.
- [12] Mark Handley et al. "Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17.
- [13] Flemming Nielson Hanne Riis Nielson. "Semantics with Applications: An Appetizer". In: *Springer Verlag, London, 2007*.
- [14] Timothy L. Hinrichs et al. "Practical Declarative Network Management". In: *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*. WREN '09. Barcelona, Spain: ACM, 2009, pp. 1–10. ISBN: 978-1-60558-443-0. DOI: 10.1145/1592681.1592683. URL: <http://doi.acm.org/10.1145/1592681.1592683>.
- [15] Peyman Kazemian, George Varghese, and Nick McKeown. "Header Space Analysis: Static Checking for Networks". In: *Proc. NSDI'12*.
- [16] Peyman Kazemian et al. "Real Time Network Policy Checking Using Header Space Analysis". In: *Proc. NSDI'13*.
- [17] Ali Kheradmand and Grigore Rosu. *Executable Formal Semantics of P4 and Applications*. 2017.
- [18] Ahmed Khurshid et al. "VeriFlow: Verifying Network-wide Invariants in Real Time". In: *Proc. NSDI'13*.
- [19] Hyojoon Kim et al. "Kinetic: Verifiable Dynamic Network Control". In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. NSDI'15. Oakland, CA: USENIX Association, 2015, pp. 59–72. ISBN: 978-1-931971-218. URL: <http://dl.acm.org/citation.cfm?id=2789770.2789775>.
- [20] Eddie Kohler et al. "The click modular router". In: *ACM Trans. Comput. Syst.* 18.3 (Aug. 2000), pp. 263–297. ISSN: 0734-2071. DOI: 10.1145/354871.354874. URL: <http://doi.acm.org/10.1145/354871.354874>.
- [21] Hongqiang Harry Liu et al. "CrystalNet: Faithfully Emulating Large Production Networks". In: *Proc. of the 26th Symposium on Operating Systems Principles (SOSP)*.
- [22] Nuno P. Lopes et al. "Checking Beliefs in Dynamic Networks". In: *Proc. NSDI'15*.
- [23] Haohui Mai et al. "Debugging the data plane with anteater". In: *Sigcomm*. 2011.
- [24] Nick McKeown et al. *Automatically verifying reachability and well-formedness in P4 Networks*. Tech. rep. Sept. 2016. URL: <https://www.microsoft.com/en-us/research/publication/automatically-verifying-reachability-well-formedness-p4-networks/>.
- [25] Vladimir Olteanu et al. "Stateless Datacenter Load-balancing with Beamer". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018. URL: <https://www.usenix.org/conference/nsdi18/presentation/olteanu>.
- [26] Mark Reitblatt et al. "FatTire: Declarative Fault Tolerance for Software-defined Networks". In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN '13. Hong Kong, China: ACM, 2013, pp. 109–114. ISBN: 978-1-4503-2178-5. DOI: 10.1145/2491185.2491187. URL: <http://doi.acm.org/10.1145/2491185.2491187>.
- [27] Leonid Ryzhyk et al. "Correct by Construction Networks Using Stepwise Refinement". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 683–698. ISBN: 978-1-931971-37-9. URL:

<https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/ryzhik>.

- [28] Radu Stoenescu et al. "SymNet: Scalable symbolic execution for modern networks". In: *SIGCOMM*. 2016. DOI: 10.1145/2934872.2934881. URL: <http://doi.acm.org/10.1145/2934872.2934881>.
- [29] Arseniy Zaostrovnykh et al. "A Formally Verified NAT". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: ACM, 2017, pp. 141–154. ISBN: 978-1-4503-4653-5. DOI: 10.1145/3098822.3098833. URL: <http://doi.acm.org/10.1145/3098822.3098833>.

APPENDIX

The following section presents the "big-step" operational semantics for some relevant statements of P4 and SEFL as well as the proof of semantics preservation through the translation process. The semantics defines a relation of the form $\langle S, s \rangle \rightarrow s'$ where the pre-state s and post-state s' represent the states before and after execution of the program statement S . The state includes header variables and metadata. The definition of \rightarrow is given by the semantics rules, written as inferences where a horizontal line separates premises (above) from the conclusion (below). The meaning of statements is summarized as a function $\mathcal{S} : Stm \rightarrow (State \rightarrow State)$, specified as \mathcal{S}_P for P4, and \mathcal{S}_S for SEFL.

The rules for most of the statements either rewrite the state by updating a header field or invoke a control state. In order to deal with the transformations in terms of variables we represent the states with two mappings: a variable environment that associates a location with a variable and a store that associates a value with a location. The variable environment env_V is an element of $Env_v = Var \rightarrow Loc$, where Loc is the set of locations. A store sto is an element of $Store = Loc \cup \{next\} \rightarrow \mathbb{N}$, where $next$ is used to hold the next free location. A function $new : Loc \rightarrow Loc$ is used to produce the next location. Under these assumptions a state is defined as $s = sto \circ env_V$ [13].

SEFL

The SEFL statements can create header fields and metadata (*Allocate*, *Deallocate*), assign the result of an evaluated expression to a variable (*Assign*), manipulate tags to simplify access to header fields (*CreateTag*, *DestroyTag*). The control statements are *Fork* and *If*. SEFL includes two statements that constrain the execution of the current path, *Fail* and *Constrain*. For a full specification of the SEFL language, please see [28].

In order to specify the semantics of SEFL statements we need to introduce the set $DV(D_V)$ of variables allocated in D_V , where D_V is a meta-variable ranging over the syntactic

category of variables, given by the following syntax:

$$\begin{aligned} D_V &::= Allocate(v); D_V \mid Deallocate(v); D_V \mid \varepsilon \\ DV(Allocate(v); D_V) &= \{v\} \cup DV(D_V) \\ DV(Deallocate(v); D_V) &= DV(D_V) \setminus \{v\} \\ DV(\varepsilon) &= \Phi \end{aligned}$$

We use \rightarrow_D to specify the transition relation for variables: $\langle D_V, s \rangle \rightarrow_D s'$. The set of tags $DT(D_T)$ has a similar specification:

$$\begin{aligned} DT(CreateTag(t, e); D_T) &= \{t\} \cup DT(D_T) \\ DT(DestroyTag(t); D_T) &= DT(D_T) \setminus \{t\} \\ DT(\varepsilon) &= \Phi \end{aligned}$$

The statements *Allocate* and *Deallocate* update the variables environment and store using the transition relation \rightarrow_D .

$$\begin{aligned} [\text{alloc}] \quad & \frac{\langle D_V, env_V[v \mapsto l], sto[l \mapsto \varepsilon][next \mapsto new\ l] \rangle \rightarrow_D (env'_V, sto')}{\langle Allocate(v); D_V, env_V, sto \rangle \rightarrow_D (env'_V, sto')} \\ & \text{where } l = sto\ next \\ [\text{dealloc}] \quad & \frac{\langle D_V, env_V[v \mapsto l], sto[l \mapsto \varepsilon] \rangle \rightarrow_D (env'_V, sto')}{\langle Deallocate(v); D_V, env_V, sto \rangle \rightarrow_D (env'_V, sto')} \end{aligned}$$

The semantics of the next statements is related to the variables environment, therefore we use the notation $env_V \vdash \langle S, s \rangle \rightarrow s'$ in order to emphasize the presence of the environment.

The statement *Assign*(v, a), assigns the result of the evaluation of the expression a to the variable v . \mathcal{A} , defined as a total function, denotes the meaning of arithmetic expressions, $\mathcal{A} : Aexp \rightarrow (State \rightarrow \mathbb{Z})$.

$$[\text{assign}] \quad \frac{env_V \vdash \langle Assign(v, a), sto \rangle \rightarrow sto[l \mapsto a]}{\text{where } l = env_V v \text{ and } a = \mathcal{A}[a](sto \circ env_V)}$$

The next semantic rules describe the *sequential composition* and *If* statements. The values of boolean expressions are true values. Their meaning is defined by the total function $\mathcal{B} : Bexp \rightarrow (State \rightarrow T)$.

$$\begin{aligned} [\text{seq}] \quad & \frac{env_V \vdash \langle S_1, sto \rangle \rightarrow sto' \quad env_V \vdash \langle S_2, sto' \rangle \rightarrow sto''}{env_V \vdash \langle S_1; S_2, sto \rangle \rightarrow sto''} \\ [\text{if}] \quad & \frac{env_V \vdash \langle S_1, sto \rangle \rightarrow sto'}{env_V \vdash \langle \text{if } b \text{ then } S_1 \text{ else } S_2, sto \rangle \rightarrow sto' \quad \text{if } \mathcal{B}[b](sto \circ env_V) = t} \end{aligned}$$

$$[\text{if}] \quad \frac{env_V \vdash \langle S_2, sto \rangle \rightarrow sto'}{env_V \vdash \langle \text{if } b \text{ then } S_1 \text{ else } S_2, sto \rangle \rightarrow sto' \quad \text{if } \mathcal{B}[b](sto \circ env_V) = f}$$

The *Fork* instruction can be expressed as an *If* with the condition on both branches satisfiable.

The *Constrain* statement ensures that the variable satisfies the specified constrain. The execution path fails if it doesn't.

$$[\text{constr}^t] \quad \frac{env_V \vdash \langle \text{Constrain}(v, c), sto \rangle \rightarrow (env_V, sto)}{\text{if } \mathcal{B}[c](sto \circ env_V) = t}$$

$$[\text{constr}^f] \quad \frac{env_V \vdash \langle \text{Constrain}(v, c), sto \rangle}{\text{if } \mathcal{B}[c](sto \circ env_V) = f}$$

$$[\text{fail}] \quad env_V \vdash \langle \text{Fail}(msg), sto \rangle$$

P4

The P4 statements follow the behavior already introduced. They rewrite the state by updating a field or invoke a control state in the parser, table or action component. For simplicity in this specification we omitted the semantic rules for the parser phase.

The *modify_field* statement sets the field using a passed parameter or a field in the metadata. In our case the environment keeps the variables (including local) as well as the metadata and their values, therefore the semantics evaluates the expression in the environment.

$$[\text{mod_field}] \quad \frac{env_V, env_A \vdash \langle \text{modify_field}(x, a), sto \rangle \rightarrow sto[l \mapsto v]}{\text{where } l = env_V x \text{ and } v = \mathcal{A}[a](sto \circ env_V)}$$

In order to specify the semantics of actions we introduce the action environment env_A , which is an element of $Env_A = Aname \hookrightarrow Stm \times Env_V \times Env_A$. The action environment allows us to associate action names with their body as well as the action environment and variable environment at the point of the declaration. The transitions will have the form $env_V, env_A \vdash \langle S, sto \rangle \rightarrow sto'$.

$$[\text{act}] \quad \frac{env'_V, env'_A \vdash \langle S, sto \rangle \rightarrow sto'}{env_V, env_A \vdash \langle \text{action } a, sto \rangle \rightarrow sto' \quad \text{where } env_A a = (S, env'_V, env'_A)}$$

Actions that modify fields can be combined in parallel or sequentially. The semantic rules follow:

$$[\text{seq}] \quad \frac{env_V, env_A \vdash \langle S_1, sto \rangle \rightarrow sto' \quad env_V, env_A \vdash \langle S_2, sto \rangle \rightarrow sto''}{env_V, env_A \vdash \langle S_1; S_2, sto \rangle \rightarrow sto''}$$

$$[\text{par}] \quad \frac{\langle S_1, s \rangle \rightarrow s', \langle S_2, s' \rangle \rightarrow s''}{\langle S_1 \text{ par } S_2, s \rangle \rightarrow s''} \quad \frac{\langle S_2, s \rangle \rightarrow s', \langle S_1, s' \rangle \rightarrow s''}{\langle S_1 \text{ par } S_2, s \rangle \rightarrow s''}$$

The *apply(table)* statement applies the actions in the initial state followed by the control statements in the modified state.

$$[\text{apply}] \quad \frac{env_V, env_A \vdash \langle S, sto \rangle \rightarrow sto' \quad env'_V, env'_A \vdash \langle S', sto' \rangle \rightarrow sto''}{env_V, env_A \vdash \langle \text{apply}(table), sto \rangle \rightarrow sto' \quad \text{where } table = (r, a) \text{ and } env_A a = (S, env'_V, env'_A) \text{ and } S' \in \{\text{apply}, \text{noop}\}}$$

$$[\text{if}^t] \quad \frac{env_V, env_A \vdash \langle S_1, sto \rangle \rightarrow sto'}{env_V, env_A \vdash \langle \text{if } b \text{ then } S_1 \text{ else } S_2, sto \rangle \rightarrow sto' \quad \text{if } \mathcal{B}[b](sto \circ env_V) = t}$$

$$[\text{if}^f] \quad \frac{env_V, env_A \vdash \langle S_2, sto \rangle \rightarrow sto'}{env_V, env_A \vdash \langle \text{if } b \text{ then } S_1 \text{ else } S_2, sto \rangle \rightarrow sto' \quad \text{if } \mathcal{B}[b](sto \circ env_V) = f}$$

Correctness of the translation

We introduced the semantics of the SEFL and P4 languages in order to prove that the translation process ensures the semantics preservation. The correctness of the translation amounts to show that if we translate a P4 statement into SEFL code and execute that code, then we obtain the same result as specified by the operational semantics of P4.

The translation of P4 statements into SEFL code is given by the function $\mathcal{TS} : Stm_P \rightarrow Stm_S$.

We give the translation rules for non trivial P4 statements:

$$\begin{aligned} \mathcal{TS}[\text{modify_field}(x, a)] &= \text{Assign}(x, a) \\ \mathcal{TS}[\text{apply}(table \equiv (S, S'))] &= \mathcal{TS}[S]; \mathcal{TS}[S'] \\ \mathcal{TS}[S_1; S_2] &= \mathcal{TS}[S_1]; \mathcal{TS}[S_2] \end{aligned}$$

Theorem

For every statement S of P4, $\mathcal{S}_P[S] = \mathcal{S}_S[S]$.

The theorem relates the behavior of statements in P4 and SEFL under the "big-step" operational semantics. It expresses the property that if the execution of statement S from some state terminates in one of the semantics it also terminates in the other and the resulting states will be equal. In order to prove the theorem we must first prove the following Lemma.

Lemma

For every statement S of P4 and states s and s' , if $\langle S, s \rangle \rightarrow s'$ then $\langle \mathcal{TS}[S], s \rangle \rightarrow s'$.

match-action table	
match(ip.dst)	action
8.8.8.8	modify_field(ip.src, 1.1.1.1)
9.9.9.9	drop

Figure 13: Concrete match-action table

match-action table	
match(ip.dst)	action
8.8.8.8	modify_field(ip.src, *)
9.9.9.9	drop

Figure 14: Symbolic action parameter substitution

Proof: the proof is by induction on the shape of the derivation tree for $\langle S, s \rangle \rightarrow s'$.

The case [mod_field]: we assume that

$$\langle \text{modify_field}(x, a), s \rangle \rightarrow s', \text{ where } s' = s[x \mapsto \mathcal{A}[a]s].$$

From the translation rules we have

$$\mathcal{TS}[\text{modify_field}(x, a)] = \text{Assign}(x, a)$$

According to the *Assign* semantic rule

$$\langle \text{Assign}(x, a), s \rangle \rightarrow s' \text{ where } s' = s[x \mapsto \mathcal{A}[a]s].$$

Therefore the resulting states are equal.

The case [apply]: we assume that

$\langle \text{apply}(\text{table}), s \rangle \rightarrow s''$ holds, where $\text{table} = (r, a)$, because $\langle S; S', s \rangle \rightarrow s''$ holds, where $\text{env}_A[a \mapsto S]$ and $S' \in \{\text{apply}, \text{noop}\}$.

From the translation rules we have that $\mathcal{TS}[S; S'] = \mathcal{TS}[S]; \mathcal{TS}[S']$.

The case [seq]: we assume that

$$\langle S_1; S_2, s \rangle \rightarrow s'' \text{ holds because}$$

$$\langle S_1, s \rangle \rightarrow s' \text{ and } \langle S_2, s' \rangle \rightarrow s''.$$

From the translation rules we have $\mathcal{TS}[S_1; S_2] = \mathcal{TS}[S_1]; \mathcal{TS}[S_2]$.

By applying the induction hypothesis on the premises, we have $\langle \mathcal{TS}[S_1], s \rangle \rightarrow s'$ and $\langle \mathcal{TS}[S_2], s' \rangle \rightarrow s''$.

This proves the Lemma.

SYMBOLIC TABLE MODELING

In the following section we will gradually introduce the intuition behind the modeling of symbolic tables.

In figure 13 we present a P4 match-action table. It allows two actions: rewrite the IPv4 source field to a value specified as an action parameter, or drop the packet. The match strategy for this table is exact matching on the IPv4 destination field. In the example there are two entries: i. one matching the address '8.8.8.8' in which case it rewrites the IPv4 source field to '1.1.1.1' ii. another matching the address '9.9.9.9' in which case the traffic is dropped.

Based on this concrete example we state the following:

- (1) *A single symbolic table entry per action can model the behavior of the whole set of concrete entries that employ the same action.*

In our example (figure 14), if we swap the concrete value of the parameter for 'modify_field' action for a symbolic one and then perform symbolic execution

match-action table	
match(ip.dst)	action
*	modify_field(ip.src, *)
9.9.9.9	drop

Figure 15: Fully symbolic table entry

match-action table	
match(ip.dst)	action
*	modify_field(ip.src, *)
*	modify_field(ip.src, *)

Figure 16: Duplicated symbolic table entry

match-action table	
match(ip.dst)	action
*	modify_field(ip.src, *)
*	drop

Figure 17: Fully symbolic table

we will get back a set of symbolic execution paths. Each resulting execution path will state a different branching condition on the rewritten source address field, covering the entire space of possible values.

Furthermore (figure 15), we can make the match value symbolic instead of '8.8.8.8'. This will now match any packet. While this may seem useless, as the constraint 'IPv4.dst == *' will always be satisfiable, looking closer, it has a subtle, yet very important side-effect: from this point until the field gets rewritten, any subsequent constraint imposed on the 'IPv4.dst' field will be transitively imposed on the symbolic match value from the symbolic table as well. In the end, this will partition the space of the possible match values according to the symbolic paths discovered.

- (2) *There is no need to use multiple symbolic entries for the same action.*

To see why it suffices to use one symbolic entry per type of action, let us consider the case in which we added another extra one, with different symbolic entries for the same type of action. In figure 16 the first entry will behave the same as before. In the case of the second one, the action effects will be indistinguishable from the ones belonging to the first entry as they both rewrite the source address to an unconstrained symbolic value. When it comes to the symbolic match value, there will be an extra constraint that the two symbolic values must be different (to avoid an overlap), which brings no information, since this is already part of the P4 spec itself. Thus we conclude there is no information gain from deploying multiple symbolic entries per type of action.

On the other hand, a second symbolic entry corresponding to the 'drop' action will suffice to cover all the possible behavior for this table (figure 17).

- (3) *Any action can act as the default but there is no need to model this explicitly by further insertions of symbolic table entries.*

The rationale behind this can be regarded as a corollary of the previous observation. This default action will collect the negated constraints of all previous symbolic match values - which states nothing more than what the definition of the default action in the P4 spec already does.

Known modeling limitations:

- (1) Only parameters of immediate value type can be substituted by symbolic values. We mention that no production-grade P4 program we used for evaluation broke this assumption.
- (2) In the case of overlapping match conditions different entries must state different priorities. Our model

does not cover such cases which means in practice, a symbolic path explored by the analysis might not be executed in reality, being shadowed by another one. In other words, multiple execution paths carry overlapping match conditions, the one executed in the concrete case is the one with the highest priority and this behavior should be considered as an additional post-symbolic execution step.

- (3) Even if a concrete table entry is validated according to Vera, it might get rejected by the switch at runtime because it either overlaps another entry with the same priority, or the table is full. Such corner-cases should also be considered outside Vera.



PAPER – 3: Equivalence and its applications to network verification

Equivalence and its applications to network verification

Paper #91

ABSTRACT

Network verification promises to find rare bugs in networks, but using it requires that administrators (completely) characterize the expected behavior of the network in formal languages such as Datalog or CTL. The difficulty of achieving this task hampers deployment of verification widely.

We propose to use equivalence between different network dataplanes as an implicit, simpler way to specify the required correctness properties. While equivalence is a well-known undecidable problem for general-purpose programs, we show that equivalence is decidable and can be verified efficiently when applied to network dataplane processing.

We present EQ, an algorithm that checks the equivalence of two network dataplanes implemented in the SEFL language by using symbolic execution [31]. We implement EQ in the `sdiff` tool and use it to catch a variety of bugs in Openstack Neutron, P4 programs and network dataplane updates. Our evaluation highlights that equivalence is an easy way to find bugs, scales well to relatively large programs and discovers subtle issues otherwise difficult to find.

1. INTRODUCTION

Misconfigured or faulty networks ground airplanes, stranding thousands of passengers and render online services inaccessible for hours on end, leading to disgruntled users and massive losses in revenue. Network verification promises to fix such rare yet devastating problems by ensuring that networks always follow their operator’s stated policy (a correctness specification), which might include forwarding and isolation specifications. Verification proposals can uncover faulty dataplane configurations [15, 21, 31, 22, 17], can simulate the effect of control plane changes (such as configuration changes) before they are applied [7, 9, 2] or inject these changes into an emulated clone of the live network to examine their effects [20].

Despite this work, verification is rarely used in practice. There are at least two key problems that affect the wide applicability of verification techniques: first, expressing policies is a difficult task since they are sometimes expressed in abstract formal languages. Secondly, ensuring the specified set of policies captures all intended behavior is even more difficult.

We observe that in many cases network correctness properties can be specified implicitly by equivalence to other networks. One example is in public clouds where tenants specify a simple, abstract network configuration (e.g. two VMs connected via a L2 network), and this is translated by the cloud management software into an

actual configuration for switches and routers that must offer equivalent functionality to the two VMs. Another example is a network administrator that knows his network behaves correctly¹—and he simply wants the network to behave in the same way in the future. Finally, a P4 [3] programmer might restructure its program to meet the target switch constraints, and wants to ensure the new program is equivalent to the old one.

Checking equivalence could therefore be very useful for easy-to-use, no-specification verification; unfortunately, checking the equivalence of arbitrary programs is a well-known undecidable problem. Network dataplanes, however, perform restricted functionality that can be verified exhaustively using symbolic execution [31, 6]. This implies that we can use the results of symbolic execution—an exhaustive exploration of how a symbolic packet can be processed by a given network—to decide if two networks offer the same functionality from the point of view of an external observer.

We present EQ, the first algorithm that automatically checks the equivalence of two network dataplanes. We have proven EQ’s correctness and have implemented it in the `sdiff` tool that uses the Symnet symbolic execution engine to test the equivalence of two network dataplanes expressed in the SEFL language [31].

We have used `sdiff` to find bugs in Neutron, OpenStack’s cloud management software networking driver, by checking the equivalence of tenant configurations and low level implementation of those configurations. We have found 6 implementation bugs in Neutron, 1 of which was unknown, and a number of configuration bugs. `sdiff` can also check that P4 program optimizations preserve correctness, different dataplane models of the same network functionality are equivalent and network dataplane updates preserve correctness in tens of seconds, even for large FIBs. `sdiff` achieves all these tasks in seconds/minutes, and easily scales to deployments including tens of Openstack VMs and routers with 10K entries in their FIBs.

2. BACKGROUND AND MOTIVATION

Almost all network verification tools work by constructing a snapshot of the network dataplane, creating a verifiable model and then automatically verifying if certain desired properties hold [22, 17, 21, 15, 31, 7, 2, 9]. Example properties include basic reachability and isolation, but can also describe changes to packet headers, invariants, paths, bidirectional reachability, state-

¹After taking pains to actively test it.

ful processing, and so forth. The accuracy of network verification depends on whether the desired properties completely and correctly specify the processing to be performed by the network.

These properties are in fact a *network specification* and they must simultaneously be complete and accurate to ensure correctness. Completeness is particularly difficult to achieve since the administrator must reason about all possible packets that can appear at any node in the network. Accuracy is difficult too: specs described in formal languages such as Datalog (as suggested by NOD [21] or FML [11]), Computation Tree Logic (NetCheck [26]), or iterative specification [29] are powerful because they capture a wide range of properties, but are (very) difficult to use by network administrators. Simpler APIs such as the “plumbing” graph of NetPlumber [16] or `ipfw`-like rules of IN-NET [30], on the other hand, are easy to use but only capture basic properties such as reachability and isolation. There is a fundamental tradeoff between ease of development and the accuracy and completeness of the specification.

In this paper we propose **equivalence** as a simpler alternative to explicit specification of network properties required in all existing works. In particular, the network administrator or programmer will be able to supply two models of networks that they think should behave equivalently, and our goal is to check whether they actually do, giving counter examples if they don’t. Specification by equivalence is both accurate² and complete because it describes the processing of all possible packets, and is directly applicable in many scenarios. We discuss some of these in our evaluation (§6).

Unfortunately, checking equivalence is undecidable for general programs because it can be reduced to the halting problem [8]. However, equivalence *is* decidable for programs that always terminate and have bounded inputs: to check it, enumerate all possible inputs, run both programs and check that the outputs match.

Network dataplanes are such programs: they have bounded inputs (packets) and rarely loop (P4[3] does not include traditional loops). Loops can appear, especially network-wide, but they can be caught automatically [22, 21, 31]. We will show that it is possible to quickly and automatically decide if two programs describing network dataplanes are equivalent. To make our approach scalable, we rely on exhaustive symbolic execution instead of testing all possible inputs.

3. APPROACH

Consider the example in Figure 1 where we show two code fragments that model a router with three FIB entries; this example is taken and slightly adapted from Symnet [31]. The code is written in the SEFL language,

²as accurate as the network models it is based on

an imperative language that has been designed to allow cheap verification of network dataplanes [31].

The first program is simple to understand as it relies on a sequence of **If/Else** clauses that forward the packet to the correct output port; there is one **If** per FIB entry. The second program is optimized for symbolic execution: it does not use **If** instructions at all, first **Forking** the packet (i.e. creating clones of it) and then using the **Constrain** instruction for each clone to restrict the packets that may leave on each port. **Constrain** drops all packets that do not match the constraint and has no effect on packets that do.

The two programs are meant to be equivalent, which means they must process packets in exactly the same way. Informally, this means that injecting *any* packet into equivalent input ports of the two programs (e.g. `in`) will result in both dropping the packet, or both emitting the same packet(s) on equivalent output ports (see §4 for a formal definition).

Our goal is to automatically and scalably decide if two programs expressed in SEFL are equivalent. Exhaustive testing for all inputs is one way to achieve this goal, but it is not feasible to use in practice for networks: network headers size are 64B or more, meaning that there are 2^{512} possibilities. Instead, we use symbolic execution to simulate how a program may process any possible input without actually running the program that many times. We now give a brief overview of how symbolic execution works; for a detailed description, see, for example [4].

Symbolic execution allows the programmer to declare certain variables as having symbolic values—for instance, the inputs—and then executes the program using these symbolic variables. In SEFL, input packets can be made symbolic and the Symnet symbolic execution engine can be used to track how these packets are handled. In figure 1 we show the symbolic execution of the two programs when we have a symbolic packet at input where the IP time-to-live field (TTL) and the IP destination address `dst` can take any value allowed in their range.

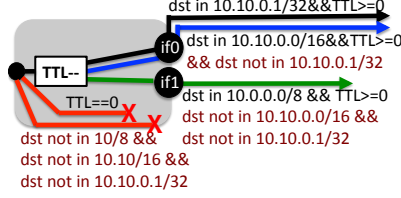
When a branch condition depends on a symbolic variable, the symbolic execution engine uses a constraint solver (Z3[5] for Symnet) to check if the condition is satisfiable: if it is, the constraint is recorded in the path and the execution continues on the “then” branch (path). At the same time, the engine checks whether the negated constraint holds, and if it does it also continues execution on the “else” branch, recording the negated constraint that must hold. Both paths are now explored until they finish, independently. For instance, in the basic router program, the first **If** branch results in a path where the TTL is at least 1, then decrements the TTL and forwards the packet to the appropriate interface(s). The else path where the TTL is zero is also explored, but it stops immediately because the packet is dropped.

In symbolic execution, operations that have concrete

```

If (TTL>0) TTL--;
Else Fail;
If (dst in 10.10.0.1/32)
  Forward("if0");
Else If (dst in 10.10.0.0/16)
  Forward("if0");
Else If (dst in 10.0.0.0/8)
  Forward("if1");
Else Fail;

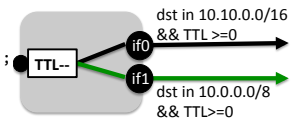
```



```

TTL--; Constrain(TTL >= 0);
Fork (
  Path1 {
    Constrain(dst in 10.10.0.0/16);
    Forward("if0");
  }
  Path2 {
    Constrain(dst in 10.0.0.0/8);
    Forward("if1");
  }
)

```



Basic router coded with If/Else: (l)code (r)symbex

Optimization with Fork/Constrain: (l)code (r)symbex

Figure 1: Two SEFL programs modeling a router with three entries in its FIB. Are they equivalent?

operands are executed as they would in the original program, while operations with symbolic inputs are not executed per se, they are just “remembered”: when decrementing the TTL field, Symnet simply remembers that the current value can be obtained from the initial value minus 1, and so the path constraint becomes $TTL \geq 0$ after TTL is decremented.

The output of Symnet is a set of paths, where each path represents a symbolic packet. For each path Symnet lists the instructions it has executed, the ports it has visited and, for each header field the associated constraints or the concrete value. Paths can either be successful or failed. Failed paths are paths where packets are explicitly dropped. Successful paths have only satisfiable constraints and no more instructions to execute.

In the basic router program, symbolic execution outputs two failed paths (shown in red) and three successful paths (Fig.1). Two successful paths (black and blue) leave the router on interface `if0`, and one on `if1` (green). The path constraints are written next to each path, and are derived by Symnet from the `If` conditions (shown in black) or negated `If` conditions appearing above (shown in red). By taking the union of constraints of the black and blue paths we know exactly which packets can leave `if0`, and by examining the constraints on the green path we know which packets will leave on `if1`. In the optimized program, there are only successful paths: one on `if0` and one on `if1`.

To decide if the two programs are equivalent, we need an algorithm that compares the sets of paths exiting on port `if0/1` in both programs and checks if they are equivalent. We discuss possible approaches next.

4. NETWORK PROGRAM EQUIVALENCE

Before we discuss how equivalence can be checked, we give a formal definition of equivalence. This, in turn, requires constructs that characterize how a network dataplane (program or model) processes a packet.

Let *Prog* denote the set of programs - defined as mappings between *Ports* (function names) and *SEFL* instructions. Let *Packet* denote the set of all admissible input memory states³. Injecting a packet *p* into a program *prog* at port *port₀* will result in a set of out-

³We call it *Packet* because usually the input variables for a network program are actually packet headers.

put packet and port pairs $O(prog, p, port_0)$ defined as follows.

DEFINITION 4.1. Let $O : Prog \times Packet \times Ports \rightarrow 2^{Packet \times Ports}$ defined as:

$$O(prog, p, port_0) = \{(\sigma_1, port_1), (\sigma_2, port_2), \dots, (\sigma_n, port_n)\}$$

be the set of packet and output port pairs resulting from the execution of program *prog* given input packet *p* and input port *port₀*.

Define **network equivalence** as follows:

DEFINITION 4.2. Let *p* ∈ *Packet* an input packet and $P_1 \in Prog$, $Ports_1$ a set of program ports in P_1 and $P_2 \in Prog$, $Ports_2$ a set of program ports in P_2 . Let \mathcal{R} a bijective mapping between $Ports_1$ and $Ports_2$.

We call programs P_1 and P_2 **equivalent** with respect to a subset *Q* of packets $Q \subseteq Packet$, input ports $port_1 \in Ports_1$ and $port_2 \in Ports_2$ with $\mathcal{R}(port_1) = port_2$ and output relation $\omega \subseteq Packet \times Packet$ iff $\forall p \in Q, \exists \chi$ bijection between $O(P_1, p, port_1)$ and $O(P_2, p, port_2)$ such that $\chi(\sigma_{1i}, pc_{1i}) = (\sigma_{2j}, pc_{2j}) \iff$

$$\mathcal{R}(port_{1i}) = port_{2j} \quad (1)$$

$$(\sigma_{1i}, \sigma_{2j}) \in \omega \quad (2)$$

DEFINITION 4.3. We call P_1 and P_2 **equivalent** with respect to $Q \subseteq Packet$ and $\omega \subseteq Packet \times Packet$ iff $\forall port_1 \in Ports_1$ and $port_2 \in Ports_2$, P_1 and P_2 are equivalent with respect to $Q(port_1, port_2)$ and ω .

Intuitively, the above definition goes to say that given the same input packet, the number of output packets from both programs coincide and there must be a one-to-one correspondence between packets emitted by both programs. Also, packets in correspondence must satisfy the output packet condition ω , which typically requires that the values of the header fields in the two packets are equal.

Checking equivalence. Symbolic execution outputs can be used to check that two programs are equivalent according to the definition above, but the algorithm to do so is far from trivial. We first discuss informally three basic approaches which we call input, output and functional equivalence, and explain why they all need to be implemented simultaneously to ensure correctness.

The simplest way to check equivalence is to compare which packets can exit any given port by examining the feasible values for each header field—we call this co-domain or **output equivalence**. Say ports p_1 and p_2 belong to two different programs, but they should be equivalent. The algorithm to check for output equivalence is simple: compute the reunion of possible values for each header field for each of these ports, and then check whether the resulting sets of packets are identical (this can easily be achieved using set operations). In more detail, let the allowed packets for port p_1 be S_{p_1} and S_{p_2} for port p_2 . Then, check whether the two sets are exactly the same: $(S_{p_1} \setminus S_{p_2}) \cup (S_{p_2} \setminus S_{p_1}) = \emptyset$.

In the example in Figure 1, let's first consider the two **if1** ports: compared to the basic model, the optimized model wrongly allows more packets to pass (packets in 10.10.0.0/16), so the two paths are not equivalent, and thus the models are not equivalent for ports **if1**. If, however, we consider the two **if0** ports, we find that $S_{p_1} = S_{p_2} = \{TTL \in [0, 255]; dst \in 10.10.0.0/16\}$, which means that the two paths are equivalent.

The careful reader will have noticed that the two routers differ in how they treat packets when TTL is 0. The basic router will drop the packet straightaway. However, the optimized router will decrement the TTL regardless of its value, and when it is zero it will wrap around to 255 as TTL is unsigned — thus, the constraint $TTL \geq 0$ always holds. The two models are not equivalent, but checking just output equivalence is not enough to capture this problem.

The next step is to also check the constraints applied on the original (input) values of the header fields, before any modifications are made; when combined with output equivalence checking, we are now checking **input/output equivalence**. With input/output equivalence, we will find that the basic model only allows packets to pass when $TTL \geq 1$ while the optimized model allows packets when $TTL \geq 0$; the two ranges are not the same, so the two models are not equivalent.

Checking for input/output equivalence is necessary to find bugs, but on its own it is still not sufficient. To see why this is the case, consider two trivial models where one leaves the TTL field unchanged, while the other executes the instruction $TTL = 255 - TTL$. Both the input values (0-255) and the possible output values (0-255) of the two models are the same, yet they are obviously not equivalent. What we also need is **functional equivalence**: regardless of the initial value of the TTL, the two values of the TTL after executing the two programs should always be equal. In our example, functional equivalence is not true because the condition $TTL = 255 - TTL$ does not always hold.

Note that all these three checks are simultaneously needed to ensure equivalence: removing a single check leads to wrong results. We have already shown that in-

put/output equivalence is not sufficient and functional equivalence is needed. We now discuss why any other combination of two checks is insufficient either.

First, assume we have functional equivalence and output equivalence; are the two models necessarily equivalent? In model 1 from Fig. 1, $TTL = 0$, while in model 2 we have `Constrain(TTL>100);TTL=0`. The output is always 0, and for any allowed packet we have both functional and output equivalence. Yet, the first model allows all packets through, while the second only allows those with $TTL > 100$; the models are not equivalent.

Input and functional equivalence are not also insufficient. Consider a simple model that allows packets to pass unchanged, and another that duplicates the packet by executing a **fork** instruction. These two models are equivalent from an input and functional point of view, however they are not equivalent on output: the first emits a single packet while the second emits two.

4.1 Equivalence with EQ

Now that we have given the intuition of how equivalence should be checked, we present our equivalence algorithm and prove its correctness. Before we describe our algorithm, we present a series of definitions that help us explain how it works.

Let M be a SEFL program. A symbolic packet p represents a set of concrete packets that have the same header structure - i.e. the same set of header fields where each field has a starting address and size and where each header satisfies a set of *constraints*. For every header field f , let $Expr(p, f)$ denote the concrete value or symbolic expression to which the field is bound in packet p . Given a symbolic packet p and an input port i to start exploration, symbolic execution will produce a set of symbolic packet and output port pairs:

$$Symbox(M, i, p) = \{(o_1, \pi_1), (o_2, \pi_2), \dots, (o_n, \pi_n)\}$$

Each of the output packets can be either successful or failed, and will impose constraints on the values of input packet p . A successful packet on a given port in symbolic execution corresponds to one or more packets emitted on that port in actual execution. A failed packet, on the other hand, is a packet that does not leave the program in standard execution (see Def. 4.2).

Given output packet $(o_l, \pi_l) \in Symbox(M, i, p)$, define the corresponding input packet to be a subset of p such that symbolic execution of M with packet p outputs o_l as follows:

$$InputPacket(p, o_l) = p_l \Leftrightarrow Symbox(M, i, p_l) = \{(o_l, \pi_l)\}$$

The task of the equivalence algorithm is to take these symbolic outputs and decide whether they satisfy equivalence. We say that two symbolic packets satisfy functional equivalence $p_1 = p_2$ if, for every header h in p_1 situated at offset f and having size s , there exists a header h' in p_2 at the same offset and of the same size,

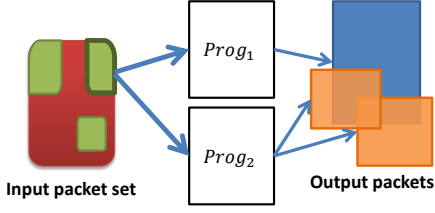


Figure 2: Ensuring input equivalence in EQ

such that $Expr(p_1, h) \neq Expr(p_2, h')$ is not satisfiable. In other words, we require that for every header of p_1 there exists an equivalent header in p_2 that takes exactly the same values.

To prove equivalence as defined in 4.2, we must simultaneously ensure input, output and functional equivalence. EQ, our algorithm is shown in Algorithm 1 and checks equivalence between M_1 and M_2 when injecting symbolic packet p_0 at input port i . The presentation assumes that both programs have the same number of ports, and that all equivalent ports are named the same in both programs.

EQ enforces input equivalence by performing symbolic execution on the first program (line 2), taking the constraints applied to the input packet for each output packet (line 4), and then using this constrained packet to run the second program (line 5). EQ assumes that M_1 outputs at most one packet for each possible input packet; we relax this assumption later.

The intuition behind EQ is depicted in Figure 2. The green region on the left denotes the same input packet that is injected in the two programs ($Prog_1$ and $Prog_2$). The blue region is the symbolic packet resulting from the first program, and the two orange packets are the symbolic packets that result from the second program. Equivalence is achieved whenever the blue region perfectly overlaps the reunion of the orange regions, and when the orange regions are disjoint.

Because the same packet is injected in M_1 and M_2 , we have input equivalence guaranteed. We now have to verify that functional equivalence and output equivalence hold. Lines 8-16 check for functional equivalence: first, if q is a failed packet and we have a successful one in M_2 (line 9) the two programs are not equivalent (this follows from the assumption that packets in M_1 are not overlapping). Secondly, corresponding failed packets are trivially equivalent so we do not check them further. Successful packets, though, must leave the same output port and have all header fields obey functional equivalence (the checks in line 11).

The last part of the algorithm checks for output equivalence. In S we collect only the success packets from the symbolic execution of M_2 . If q is a successful packet, output equivalence requires that the union of all input packets which yield success packets in S be equal to the set of input packets which yield q (line 22). Finally, we check that there are no overlaps in the packets in S by

Algorithm 1 EQ Equivalence algorithm

```

1: function EQUIVALENCE( $M_1, M_2, i, p_0$ ) ▷ Are
    $M_1$  and  $M_2$  equivalent for symbolic packet  $p_0$  injected
   on port  $i$ ?
2:    $Q_1 \leftarrow \text{SymbeX}(M_1, i, p_0)$ 
3:   for all  $(q, \pi_1) \in Q_1$  do
4:      $p_i \leftarrow \text{InputPacket}(p_0, q)$ 
5:      $Q_2 \leftarrow \text{SymbeX}(M_2, i, p_i)$ 
6:      $S \leftarrow \emptyset$ 
7:     for all  $(l, \pi_2) \in Q_2$  do
8:       if  $\text{Success}(l)$  then
9:         if  $\text{Fail}(q)$  then
10:          return false
11:        else if  $l \neq q$  or  $\pi_1 \neq \pi_2$  then
12:          return false
13:        else
14:           $S \leftarrow S \cup l$ 
15:        end if
16:      end if
17:    end for
18:    ▷ Check output equivalence
19:    if  $\text{Success}(q)$  then
20:       $p_2\text{SuccessPackets} \leftarrow \bigcup_{l \in S} \text{InputPacket}(p_0, l)$ 
21:      if  $p_2\text{SuccessPackets} \neq p_i$  then
22:        return false
23:      end if
24:      for all  $x, y \in S, x \neq y$  do
25:        if  $\text{InputPacket}(p_0, x) \cap \text{InputPacket}(p_0, y) \neq \emptyset$  then
26:          return false
27:        end if
28:      end for
29:    end if
30:  end for
31:  return true
32: end function

```

verifying pairwise intersection between their input domains (lines 25-29). If there is an overlap, there must be input packets which result in two output packets for M_2 . Since M_1 has exactly one output packet (check in line 26 together with our assumption), the two programs are not equivalent.

Correctness of EQ. We say that EQ is correct iff $EQ(M_1, M_2, i, p_0) = \text{true} \Leftrightarrow M_1$ and M_2 are equivalent with respect to input packets p_0 , input ports (i, i) and ω defined as $\{(p_1, p_2) \mid p_1, p_2 \text{ have the same header fields and } \forall f \text{ header field in } p_1, p_2 \text{ } Expr(p_1, f) = Expr(p_2, f)\}$ where the port correspondence \mathcal{R} is port equality.

ASSUMPTION 1. Program M_1 is such that $\forall p \in Q$ and $i \in \text{Ports}_1$, $|O(M_1, p, i)| = 1$

ASSUMPTION 2. Let p a symbolic packet and $M \in \text{Prog}$ with Ports program ports of M , then $\forall i \in \text{Ports}$ $p = \bigcup_{(o, \pi) \in \text{SymbeX}(M, i, p)} \text{InputPacket}(p_0, o)$,

THEOREM 4.4. The algorithm EQ is correct.

PROOF. We show that whenever $EQ(M_1, M_2, i, p_0)$ is true, we can construct a bijection χ mapping each

successful *located* packet produced by M_1 to one produced by M_2 , c.f. Def. 4.2. We map each element $(\sigma_{1j}, loc_{1j}) \in O(M_1, p, i)$, where p is a concrete packet from p_0 , to a unique pair $(\sigma_{2j}, loc_{2j}) \in O(M_2, p, i)$. Fix $loc_{2j} = loc_{1j}$. To choose σ_{2j} , we look at the symbolic packet $q \in Q_1$ such that $\sigma_{1j} \in q$, computed by our algorithm, at line 2. Since the algorithm returns true, we know $Success(q) = true$. Otherwise, χ is left undefined. Also, from the assumption 1, we know q is unique. We write $constraints(q)$ to refer to the set of constraints applied to the symbolic packet q , which produce the concrete packet σ_{1j} . Let $\{(l_1, \pi_1), \dots, (l_n, \pi_n)\} = Symbox(M_2, i, p_i)$ where $p_i = InputPacket(p_0, q)$ be the set of symbolic packets computed by the algorithm at line 5. Since EQ returns true, we know all symbolic packets l_i are equivalent with q (see line 11). Also, lines 24-26 of the algorithm guarantee that $constraints(l_i)$ produces a unique packet, which must be equal to σ_{1j} . We choose that packet to be σ_{2j} . \square

Complexity. The equivalence algorithm heavily relies on the constraint solver for symbolic execution and to check functional equivalence; these components account for most of the runtime of the algorithm. Unfortunately, finding tight bounds for symbolic execution complexity is not possible because the runtime of SMT solvers heavily depends on the constraints being checked, and cannot be bounded tightly enough for the results to be useful (exponential complexity in the worst case). Here we wish to understand how much more expensive equivalence is in comparison to normal symbolic execution. A rough approximation of symbolic execution complexity is the number of paths explored. Say M_1 will explore m_1 paths and M_2 will explore m_2 for input p_0 . EQ will explore $m_1 \cdot m_2$ paths in the worst case where, for each path from M_1 we have m_2 paths explored (line 5).

Relaxing Assumption 1. Our translators (§5) obey Assumption 1, but only unicast programs obey this in practice; multicast functionality will not be verified correctly. To remove assumption 1, we must account for the fact that M_1 can output more than one packet for any input packet. Our approach is to partition the input packet space labeling equivalence classes with the program paths they represent, and then apply EQ as before, but this time checks in line 11 will be replaced by finding a bijective mapping between two finite sets. In the worst case, the number of paths explored by the revised algorithm is $n! \cdot m!$ where n is the number of output packets from M_1 and m is the number of output packets from M_2 .

5. IMPLEMENTATION

We have implemented EQ in a tool called **sdiff**. **sdiff** takes as input two SEFL programs, together with two correspondence maps, one between their input ports

and the other with respect to their output ports. By default, **sdiff** injects a basic ethernet/IP/TCP packet into the provided input ports and runs EQ, checking for equivalence a number of header fields including IP and Ethernet source and destination, TTL, protocol number, TCP ports, etc. Note that both the symbolic packet and the fields to be checked are customizable. The user may choose which packets must be injected and which fields must be checked to cater for their own notion of equivalence, even though **sdiff** provides sensible default inputs for common use-cases.

sdiff's output is split in two parts: one where paths satisfy equivalence, and one where they don't. The latter output is particularly interesting: for every erroneous pair of output packets originating from the same input packet **sdiff** lists the reason why equivalence does not hold. For debugging purposes, **sdiff** also lists the instructions executed, ports visited and constraints applied in each of the two programs, while also providing a concrete input packet that can be used to check that equivalence does not hold.

We ran the tool on the example shown in Figure 1. It takes around 1s to check equivalence and output the results to file. The failed paths output file lists 12 symbolic packets that were treated differently by our two programs, highlighting two main reasons for the lack of equivalence. The first six packets catch the TTL decrement bug in the optimized model (the TTL underflows when it is zero): each symbolic packet includes example values for the header fields which will result in a failed state in the basic model and a success path in the optimized model (in all, TTL is 0). The other six packets highlight the second problem: there is an overlap between packets exiting on port `if0` in the basic model and `if1` in the optimized model; the example given here is a packet in 10.10/16. Using **sdiff**'s output, in this example it is easy to find the bug and then correct it.

5.1 Core implementation

Our implementation includes 2000LOC of Scala for the core algorithm, together with changes to the SEFL language and the Symnet symbolic execution tool. We have added a new instruction to SEFL called **Let** that aggregates all the path constraints in a given symbolic packet, and restores it to its original observable state. **Let** corresponds to the **InputPacket** instruction - line 4 in Algorithm 1.

With these instructions, **sdiff**'s implementation of EQ is straightforward. After obtaining the list of output packets from M_1 , we retrieve the path conditions using **Let**, and then inject it in the second program. We also implemented two functions called **Diff** and **Intersect**. **Diff** handles symbolic set difference provided the same input variables and is used to implement the conditions corresponding to line 24 in Algo-

rithm 1. The implementation translates two Symnet memory states into two Z3 parse trees, the former being a conjunction of constraints while the latter is a negated constraint conjunction and feeds them into the solver. The **Intersect** procedure operates similarly to **Diff**, but is implemented to produce the intersection of two input equivalence classes.

The assumption that execution through the first program yields pairwise disjoint sets of input packets greatly simplifies **sdiff**; this assumption holds true for all the programs we have checked. However, if false, this assumption could result in wrong results. That is why **sdiff** also checks that the union of all input constraints for all output packets are disjoint. If any of these checks fails, **sdiff** outputs packets for which it is “unsure” in a separate file that can be examined by the user. In a future version we plan to implement the version of EQ that does not depend on this assumption.

5.2 OpenStack Neutron Integration

sdiff requires two SEFL programs to check equivalence. To generate SEFL programs for our evaluation, we have both created new translators (see below) and reused existing ones, in particular translators from router FIBs to SEFL [32] and from P4 programs to SEFL [anonymous].

Checking Openstack Neutron is our most significant use of equivalence so far; it required quite a bit of coding to automatically integrate with Neutron and translate to SEFL. We describe this integration work here.

OpenStack is an open-source cloud management platform that enables IaaS clouds. Similar to other clouds (e.g. as EC2), OpenStack abstracts away the complexity of the provider’s infrastructure, providing the tenant with the ability to create, manage and connect virtual machines with ease. In particular, tenants are able to connect their VMs in rich network topologies that can include VLANs connected via routers and NATs, as well as offering security policy enforcement at VM level.

Neutron is the networking service of OpenStack and it is implemented as a distributed, pluggable middleware app which *translates* high level *tenant configuration* into lower-level pieces of configuration. We show an example Neutron deployment and tenant perspective in Figure 3. The tenant creates two virtual machines (called *Public* and *Private*) that are attached to separate VLANs. The VLANs are interconnected by the mgmt router and Internet connectivity can be achieved via *Router*. This configuration is translated by Neutron into configuration rules for the different OVS switches (*br-int*, *br-tun*, etc.) and in iptables rules (not shown). The figure clearly shows that the correspondence between what the tenant had intended vs. what was actually deployed is not one-to-one, thus making manual testing cumbersome.

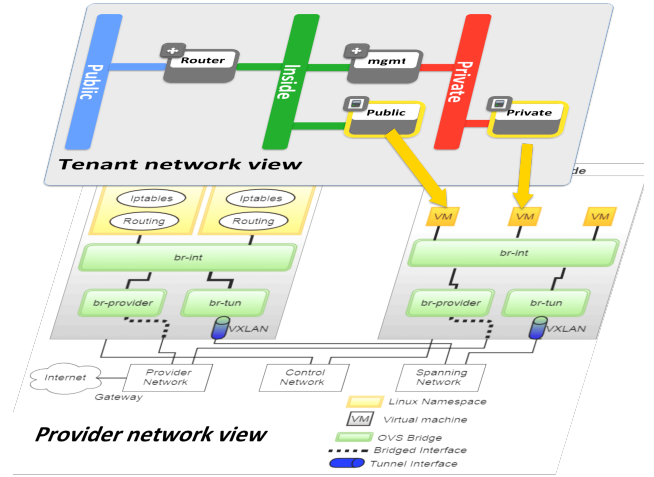


Figure 3: Neutron tenant vs provider

To verify whether Neutron works correctly, we automatically generate two SEFL models and then use **sdiff** to get all possibly conflicting behaviors between the two models as described in §3. The first model is derived from the tenant configuration. The second model is created from a snapshot of the actual Openstack deployment resulting after the VMs are instantiated: this snapshot includes OVS OpenFlow rules, iptables rules, etc. We provide details below.

Modelling tenant configurations. Our translator uses a tenant-level snapshot of the configuration (a dump of the Neutron database in practice) and then generates SEFL code that implements each user defined resource (e.g. router, switch, NAT).

The simplest abstraction is a tenant-level network which forwards packets according to a static ARP table which maps ethernet addresses to physical ports. L3 routers used by tenants may also perform source network address translation (NAT) to a *public* network in order to provide virtual machines with access to the Internet. It is also a router’s responsibility to provide floating IPs which allow traffic to flow from an outside network to a machine within the tenant’s deployment (i.e. a tunnel). Neutron also defines *security groups* which are rules that filter traffic at VM level and allow stateful operation too.

Translating this functionality to SEFL is straightforward. Each abstract object is translated separately, and they are connected using links according to the tenant configuration. We skip details for brevity.

Dataplane modelling. To model Neutron’s implementation of the tenant configuration in the cloud data-center, a snapshot of the current state of all machines is taken prior to equivalence analysis. We are interested in packet processing components residing within the system such as routing tables, software or hardware switches, firewalls, and so forth.

The most commonly used implementation of Neutron leverages `iptables` rules and OpenVSwitch (OVS) software switches [25] to implement the tenant configuration on the compute and network nodes. Both are strongly connected to Linux Kernel’s network stack to handle routing, connection tracking and stateful processing. In order to ensure cross-machine network communication and inter-tenant traffic isolation, tunneling (usually VXLAN) between compute hosts is employed.

Both *iptables* and *OVS bridges* use similar concepts such as tables and rules which match against packet header fields or per-packet metadata and apply one or more actions. To translate such matches we generate simple **If/Else** constructs; provided that all *matches* in a rule are satisfied, an action will be fired which will either alter the state of the packet (e.g. push a tunnel header) or alter the processing pipeline (e.g. drop the packet or forward to another processing block).

One of the most challenging components to model from a behavioral point of view is the connection tracking engine implemented within the Linux Kernel as part of the Netfilter framework. Same as *iptables*, the connection tracking engine uses the concept of hooks to perform a set of actions on an input packet.

Modelling stateful processing in SEFL is straightforward as long as the state depends only on the given flow (i.e. it is not global state). The Symnet paper shows how a NAT can be modelled by storing the NAT mapping as metadata within the packet; this metadata is then checked when the packet returns to the NAT for reverse translation [31].

We use a similar technique here. Conceptually, the connection tracking engine (or *conntrack*) defines a connection as a 5-tuple and tracks it independently. To model *conntrack* we use two sets of metadata variables called *forward expectations* and *backward expectations*. The former represent packets flowing in the same direction as the initial SYN packet, while the latter represent reply packets belonging to the same connection.

When state is created for a connection (a *conntrack* commit action), we store packet information as metadata as follows:

$$\begin{aligned} FW.\{IPSrc, IPDst, \dots\} &\leftarrow p.\{IPSrc, IPDst, \dots\} \\ BK.\{IPSrc, IPDst, \dots\} &\leftarrow p.\{IPDst, IPSrc, \dots\} \end{aligned}$$

Notice how the backward expectation corresponds to the packet we expect to receive from the remote end. Below is the *conntrack* code:

When we examine Neutron configurations we typically start symbolic execution without any connection information in any *conntrack* module in the topology. However, in more advanced examples, all connection tracking fields may be assigned symbolic input values, thus enabling us to capture corner-case behaviors, such as described in paragraph *Inconsistent connections in the tracker* in §6.1. This assumption is safe in most

```

procedure CONNTRACK(p)
  if  $p.\{IPSrc, IPDst, \dots\} = FW.\{IPSrc, IPDst, \dots\}$  then
     $ct\_mark, ct\_state \leftarrow tracker.\{CTMark, CTState\}$ 
     $\{IsForward, IsBackward\} \leftarrow \{1, 0\}$ 
  else if  $p.\{IPSrc, IPDst, \dots\} = BK.\{IPSrc, IPDst, \dots\}$ 
  then
     $ct\_mark, ct\_state \leftarrow tracker.\{CTMark, CTState\}$ 
     $\{IsForward, IsBackward\} \leftarrow \{0, 1\}$ 
  else
     $ct\_state \leftarrow NEW$ 
     $IsForward \leftarrow 1$ 
     $ct\_mark \leftarrow 0$ 
  end if
end procedure

```

cases, since the lifetime of a connection is much smaller than that of an *iptables* rule or an OVS flow and the chances of having new connections for every packet are high.

Verifying Neutron. To check the equivalence between Openstack dataplane and the tenant configuration, we start *sdiff* using the two generated models together with mappings between input and output ports in the two models. We discuss our findings in §6.

6. EVALUATION

We run our evaluation on a server with an i5-4590 quad-core processor @ 3.3GHZ and 8GB of RAM. Our main goal is to understand whether *sdiff* can catch interesting bugs or behaviors in practice, for realistic network dataplanes. We examine a range of applications including Openstack Neutron, P4 program equivalence, model equivalence and checking the correctness of FIB updates. Throughout our evaluation we examine *sdiff*’s scalability.

6.1 Verifying Openstack Neutron

In our test-bed, we installed a minimal Openstack deployment that included two compute nodes and one network node. In each experiment, we had one or multiple tenants deploy VMs and network configurations using Openstack, and then ran *sdiff* on another machine that had snapshots of the tenant configuration and a snapshot of the dataplane rules installed by Neutron; acquisition was performed with scripts that dumped and copied the relevant data.

The largest Neutron configuration we analyzed with *sdiff* had 16VMs belonging to six tenants, and each tenant had configured three VLANs and one router to connect them. We ran two different tests with this configuration: one where we had a bug, and one where the deployment was correct. In both cases, there are around 1600 Openflow rules in the dataplane snapshot.

In the buggy deployment, verification takes around 200s: parsing the dataplane and tenant configurations needs 10.5s, generating the associated SEFL code around 1s, and equivalence testing 189s finding 87 “faulty” packets. In comparison, symbolic execution of the tenant

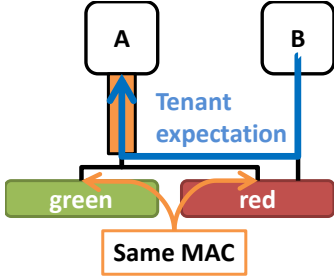


Figure 4: Identical trunked MACs

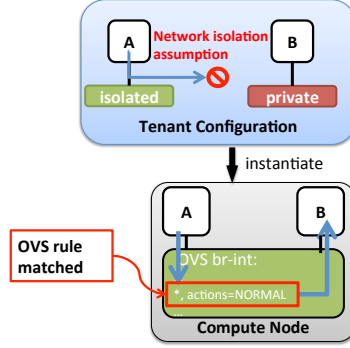


Figure 5: Network isolation

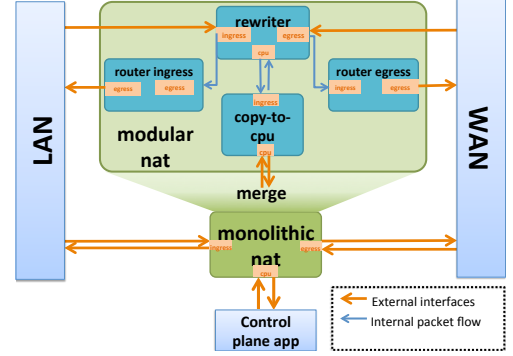


Figure 6: P4 modular vs. P4 monolithic

code took 9.3s and yielded 114/174 success/failed paths, and symbolic execution of the deployment code took 34s (186 success and 854 failed paths). In the correct deployment, verification takes 295s; the difference is given by the equivalence testing which now runs to completion because it does not find any inconsistencies (282s). Standalone symbolic execution of the tenant and deployment code took 8.3s and 41.7s respectively.

sdiff is not only able to detect issues in the implementation of Neutron on the underlying compute and network nodes, but it is also able to point out the dataplane rule which breaks the expected tenant-level behavior. We caught 5 **software bugs** introduced by Neutron’s middle-ware implementation, 2 **configuration bugs** introduced by a misconfiguration on machines in the deployment - and 1 **tenant-level misconfiguration**. In all the scenarios discussed in detail below, verification takes less than 10s because Openstack configurations are much simpler.

Identical MAC Addresses for trunking ports. The first bug appears in the simple configuration shown in Fig.4: the tenant defines a topology with two VMs connected via 2 networks (red and green). Machine A is connected via a trunk port to both the red and green networks while machine B is only connected to the green network. The tenant-level expectation for this topology is that all packets from B towards A reach their destination. However, if A uses the same MAC for both its trunked ports this prevents communication between A and B. This bug was reported on the OpenStack Neutron bug tracker ⁴. Equivalence testing of the tenant level topology vs the provider level implementation exhibited a number of failed states which indicated that all packets leaving machine B were being dropped in the forward path by an anti-spoofing rule in the *br-int* bridge connecting the two machines.

Allowed address pairs bug. An allowed address pair is an extra IPv4, MAC address pair specifically tailored to allow bridging at VM level. Thus, the expected tenant-level behavior is that traffic with destination ad-

resses (IPv4 and MAC) in the list of allowed address pairs for a particular VM be allowed on egress. ⁵

However, an implementation bug in the firewall module stops traffic from getting through. Thus, traffic issuing from VM A towards VM B is correctly forwarded to VM B, but the reverse traffic from VM B to VM A is dropped. The issue is correctly traced by **sdiff** which indicates failed (non-equivalent) states between the tenant and the provider perspectives. In the tenant view, A and B have bidirectional connectivity, whereas in deployment connectivity is broken. **sdiff** correctly captures the error in the reverse packet run and successfully identifies the offending rule.

No firewall enforcement on ICMP Type/Code. Security group support for ICMP Type/Code filtering was not implemented for older versions of Neutron ⁶. **sdiff** shows how ICMP traffic that is meant to be blocked by security rules is allowed in the dataplane.

Filtering with security groups. Security groups are collections of ACL rules that apply to all VMs part of that group. When specifying connectivity outside the group, tenants can use prefixes or remote security groups to specify the external source of traffic.

There was a bug in the implementation of filtering when remote security groups were used. In our setup, we had two groups called *green* and *blue* and a rule that all traffic from the *green* group should reach the *blue* group ⁷. We instantiated three VMs: A in the *blue* group, B in the *red* group, and C in both groups.

At runtime, C could not be reached by neither A or B, violating the tenant configuration. **sdiff** outputs multiple non-equivalent path in the two network topologies showing allowed traffic from A/B to C in the tenant configuration that is dropped in practice.

Inconsistent connections in the tracker. This bug appears when connectivity is repeatedly enabled and disabled for the same host-pair. We ran our test with

⁵<https://bugs.launchpad.net/neutron/+bug/1697593>

⁶<https://bugs.launchpad.net/neutron/+bug/1708358>

⁷<https://bugs.launchpad.net/neutron/+bug/1708092>

⁴<https://bugs.launchpad.net/neutron/+bug/1626010>

two VMs A and B, and the tenant allows traffic from A to B. In Neutron, all ACL rules are directional, and they are implemented using the connection tracker. In this case, A can initiate connections to B and B can respond, but B cannot initiate a connection to A.

After instantiation, A starts a connection to B which creates per-connection state in the conntrack module. Immediately afterwards, the tenant disallows traffic from A and B, which marks the connection in the conntracker as “dead” but does not delete the conntrack entry. When packets of the same connection reach conntrack, they will be dropped as expected. The problem appears when the tenant re-enables traffic from A to B: the flow entry mark is not cleared, and all subsequent packets are incorrectly dropped.⁸ In this case, `sdiff` exhaustively explores all possible connection states for a given input packet. The incorrect behavior is captured and reported by highlighting the offending rule and the preconditions which yield the undesired behavior.

A network isolation bug was discovered solely using `sdiff` and was reported as a Neutron bug⁹. In this setup, we have two machines running on the same host as in Figure 5, each connected to different, unconnected VLANs. The expected behavior at tenant level is that no traffic from B can ever reach A. Next, assume host A is part of a permissive security group whereby ingress HTTP traffic is allowed; further assume that B knows A’s MAC and IP addresses. Then, HTTP traffic from B will reach A, breaking the expected isolation. This bug exists even when host B belongs to a different tenant.

`sdiff` successfully detects the erroneous behavior, providing a packet from B that can reach A. To validate the bug, we reproduced the behavior in deployment. The result was that, indeed, HTTP traffic from B reached A, violating the isolation assumption.

We further note that this bug is difficult to catch with standard testing because ARP traffic was correctly blocked, and a simple `http-ping` from B to A would fail. Because `sdiff` uses symbolic packets, it finds a valid packet which will be received by A.

Old Linux Kernel. This is a configuration bug that we stumbled upon when deploying Openstack in our test-bed. The OpenVSwitch adaptor in Neutron needs Linux kernel support to access the Netfilter’s conntrack module; support exists since version 4.3.

In our deployment, we had a compute node with kernel version 4.2. We deployed two VMs (A and B) with a security group rule that allowed all traffic from A to reach B. This rule should have been inserted in the software bridge of the compute server by Neutron. However, since there is no kernel support for connection tracking (as required by the firewall module), the inser-

tion of the security groups fails, and all traffic between A and B will be dropped.

`sdiff` caught this behavior by reporting successful execution at tenant level while the same input packet was dropped in the deployed dataplane. Using `sdiff`’s output we could see that security group rules were not inserted in OVS, and later we realized the problem was due to the Linux kernel version.

Tunnel endpoint listening on *localhost*. This issue arises when Puppet, a provisioning tool, erroneously binds a tunnel endpoint on a compute node to the *localhost* address. The effect of this bug is that the VMs hosted on the affected compute node will not be able to communicate with VMs running on different compute nodes. This bug is detected by `sdiff` while the same deployment with correctly set tunnel endpoints is deemed equivalent. In this case, `sdiff` also points out to the mis-configuration that breaks equivalence.

Troubleshooting VM connectivity. A common configuration issue appears in tenant networks with many security groups. The tenant wishes to enable communication between two of its VMs but mistakenly adds them to different security groups (for instance because the two groups have similar names). By default, security groups are configured such that they allow ingress traffic from machines belonging to the same group, but not from other security groups. After deployment, the user notices that no traffic flows between its VMs.

Troubleshooting connectivity problems is difficult for tenants, as it requires manually checking the ports of a given VM, and the security groups which they belong to. With `sdiff`, the administrator is able to quickly assess that the tenant and provider perspectives are identical. Thus, there must be a misconfiguration at tenant level which does not meet the user’s expectation. The symbolic execution of the tenant level topology indicates that all failed outcomes are due to an ingress security group which is not matched at B’s level.

6.2 P4 equivalence

P4 [3] is a high level language that enables programming switch dataplanes that can also be efficiently implemented in hardware. Despite its apparent simplicity, coding P4 programs is tricky: unexpected behaviors may be accidentally introduced during the design or runtime phase. In this section we show how `sdiff` can be used to determine behavioral equivalence between different P4 programs with seemingly identical data-plane configuration and functionality.

Monolithic NAT vs modular NAT. One of the simplest P4 tutorials is a Network Address Translator provided. The NAT includes three distinct pieces of functionality: a *NAT rewriter*, which simply sets packet fields to given mappings, a *router* which per-

⁸<https://bugs.launchpad.net/neutron/+bug/1715789>

⁹Anonymized for blind review

```

ACL table:
table acl {
  reads { ipv4.srcAddr : exact; }
  actions { _drop; _nop; }
}
Entry:
table_add acl 10.0.0.3 _drop

```

```

Attempt 1:
apply(ipv4_lpm);
apply(acl);

Attempt 2:
apply(acl);
apply(ipv4_lpm);

```

Figure 7: Ways of adding an ACL to a P4 router.

forms longest prefix matching, assigns next hop address and selects the proper output port and a *CPU redirector*, which encapsulates a packet and sends it to a control-plane application if no NAT mapping is found. Even for a simple set of table rules, understanding the interactions between these different pieces is difficult.

To check whether the functionality of our P4 NAT works correctly, we wish to compare it to a modular design that runs different functionality in separate P4 programs which are connected. We still prefer the monolithic approach for deployment because it is cheaper to implement and performs better at runtime than our modular design that serializes and de-serializes packets between the interconnected boxes.

In Figure 6 we show the two implementations of the NAT. We used an existing P4 to SEFL translator to generate models for both NATs and used `sdiff` to check whether they are equivalent. In around 5s, `sdiff` shows that the implementation of the monolithic NAT is not equivalent to the modular implementation. In the monolithic implementation, it is possible to translate a packet intended for the LAN and then send it on the LAN interface. The same behavior is not possible with the modular NAT because the routing tables corresponding to LAN and WAN networks are split into 2 distinct routers - one for *ingress* and one for *egress*. This bug in the NAT that is otherwise quite difficult to catch.

Which is the correct order of table application?

In our next example, we take the simple router P4 tutorial and we enhance it to enable ACL processing. Assume that the P4 programmer initially instructs the ingress pipeline to first route packets and then apply ACL. In a subsequent run, the programmer decides to reverse the order in which the two tables are applied in order to avoid routing packets that would be dropped by the ACL; this approach seems more efficient. In our program, the ACL table has two actions: one that drops packets and that allows them through (see Fig. 7).

When we compare the two programs with `sdiff`, it is surprising to find they are different. The first version matches our expectation that unwanted packets (10.0.0.3) are indeed dropped. The second version surprisingly allows all packets through. This is because the packet is not actually dropped in the ACL table. The P4 spec states that a drop action within the ingress pipeline only marks the packet for rejection (setting the `egress_spec` metadata to 512) and continues execution from that point on. When the dropped packet hits the

Algorithm	Prefixes in FIB					
	100	500	1000	2000	5000	10000
<i>Basic router</i>	0.74s	0.6s	0.8s	0.9s	2.3s	6s
<i>Optimized router</i>	0.1s	0.2s	0.4s	0.8s	1.6s	3s
sdiff	1.68s	3.6s	6.9s	14.5s	57.9s	240s

Figure 8: Testing two symbolic routers for equivalence.

ipv4_lpm table, the default action sets the egress spec to that of a valid interface, reviving the packet.

6.3 Model Equivalence

Prior work has proposed various models for the same dataplane functionality that achieve different trade-offs in the runtime/verification space [6, 31].; for instance, a naive router implementation takes twenty times longer to analyze symbolically than an optimized one [31]. Optimizing routers for fast symbolic execution while ensuring correctness is not trivial, however, as our running example (Fig. 1) has shown.

We used translators implemented in prior work [32] that take router FIBs as inputs and output SEFL code for basic or optimized router models. We used FIB snapshots from the Stanford dataset [15], and randomly selected a subset to generate different size FIBs which were fed to the translators. In table 8 we show the time needed to perform symbolic execution of the basic and the optimized router models, and the time it takes `sdiff` to verify equivalence.

First, notice that the runtime of the optimized model is always smaller than the basic one, but the relative performance varies. `sdiff` did not find any differences between the two models. Equivalence verification time grows from around 1s for small FIBs to four minutes when we have 10K entries in the FIB. This is to be expected, since the worst case path count for equivalence checking is the product of the number of paths in the two models being checked.

Finally, we also inserted a bug in the translator, where in one corner case we generated a `>` constraint instead of a `<`. `sdiff` promptly highlighted the bug for models generated from a small (100 entry) FIB.

These results give confidence that the optimization works correctly; however, we stress that this is not a complete verification of the FIB to SEFL translators; we simply check that the outputs (the models) match for a few given inputs (the FIBs). To prove correctness, we would have to show that the outputs are the same regardless the input, but this is undecidable because the translators are general purpose programs.

6.4 FIB updates

Let’s consider another example where we have a verified network dataplane and we wish to check if an update preserves the way the dataplane works in terms of basic routing. As a first approximation, let’s consider

	Symbolic packet checked				
	0/0	10/8	10.30/16	10.30.11/24	10.30.11.91
sdiff	262s	117s	105s	22s	7s

Figure 9: Checking if FIB updates preserve equivalence.

the router model discussed in the previous section as a starting point for our dataplane, to which we make a single change in its 10K entry FIB: we modify the egress interface for one prefix from Vlan22 to GE/1.

Then, we check for equivalence between the dataplane with and without our change. We measure the time it takes to check that the update preserves equivalence while varying the range of packets for which we check equivalence. If we test a 0/0 symbolic packet, we end up checking equivalence for all packets: if we assume model generation is correct, it is excessive to check all possible packets—it suffices to only check the prefix that has changed. The time needed for equivalence verification is shown in Table 9: clearly, the more generic the prefix, the longer equivalence takes. For small changes, the equivalence checking time is just 7s, and it grows to 22s for a /24 prefix. While not exactly real-time, these times hint that online verification is possible if we parallelize symbolic execution (which should be feasible, since every path can be evaluated independently).

7. RELATED WORK

There exist a wide range of specification languages and verification tools for network dataplanes. NOD [21, 23] relies on models and queries written in a variant of Datalog, Anteater [22] translates networks and reachability queries to SAT formulae, while NetPlumber [16] takes as input a graph and network boxes modeled as bitwise transfer functions, and uses HSA [15] to check for compliance. Finally, NetCheck [26] takes specifications written in CTL and uses symbolic execution with Symnet to check them. All these tools have merits, yet one of their biggest problems is the difficulty of specifying what the network is meant to do. In many cases, the spec underspecifies the behavior, meaning that potential problems are missed.

Another line of work focuses on more rigorous specifications which are first proven correct and then translated to dataplane rules. Examples here include Kinetic [18] which takes Finite State Machine descriptions of network functionality, FatTire [28] that takes regular expressions specifying paths to be taken by packets, NetKAT [1] which takes a program written in a form of denotational semantics and finally Cocoon [29] which enables iterative design and specification for networks. All of these tools offer much stronger correctness properties, but this comes at the expense of usability.

Our work is complementary to the approach of specification followed by verification adopted in prior work. By implementing equivalence checking we allow the specification to be implicit, simplifying the task of the user.

In programming languages, equivalence testing is not a novel concept. DECKARD [13], CCFinder [14] and [19] P-Miner look for syntactically similar pieces of code that are equivalent. Our work aims to decide whether two network dataplane models process the packet in the same way, a much stronger definition of equivalence.

UC-KLEE [27] checks for equivalence in arbitrary functions by using symbolic execution with Klee, and is heuristic because symbolic execution may not finish on traditional programs. UC-KLEE’s equivalence algorithm focuses only on functional equivalence; therefore it can miss behaviors that are not equivalent. Finally, UC-KLEE’s goal of analyzing C programs is much tougher because it must deal with dynamic memory allocation. By using SEFL, a network-specific modeling language with restricted primitives, our tool circumvents this burden.

EQMINER [12] detects functionally equivalent code via random testing but does not offer guarantees that two programs are equivalent because it does not cover all possible test cases. Another work that aims to achieve the same goal with symbolic execution, targets functional equivalence for simple arithmetic functions, in code that has no branches [10]. Neither tool is exhaustive, so they do not offer correctness guarantees.

Our work targets network data-planes and offers an accurate definition of equivalence in this context, as well as an implementation that exhaustively checks interesting programs for equivalence. Such verification is currently out of reach for traditional programs, which may explain the relatively few prior works in this space.

8. CONCLUSIONS

Checking equivalence is a famous undecidable problem for general programs. However, network dataplanes have restricted functionality and we have shown that it is possible to check their equivalence. We have designed and proven the correctness of EQ, an algorithm that uses symbolic execution to scalably check the equivalence of network dataplanes. We have implemented EQ in **sdiff**, a tool which we will open-source soon.

Equivalence testing enables easy-to-use verification, and we have used it to find many Openstack Neutron and P4 program bugs, among others. While equivalence checking is more expensive than individual symbolic execution of the two programs, it runs in a few minutes for routers with 10K entries in their FIBs and in Openstack deployments with tens of virtual machines. Such scalability widens the applicability of **sdiff**.

The possible uses of **sdiff** are broad. One particularly interesting avenue of exploration is to check the equivalence between SEFL models and the actual dataplane (written in C); this is our future work.

References

- [1] Carolyn Jane Anderson et al. “NetKAT: Semantic Foundations for Networks”. In: *POPL’14*.
- [2] Ryan Beckett et al. “A General Approach to Network Configuration Verification”. In: *SIGCOMM*. 2017.
- [3] Pat Bosshart et al. “P4: Programming Protocol-independent Packet Processors”. In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014).
- [4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs”. In: *Proc. OSDI’08*.
- [5] Leonardo De Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Proc. TACAS’08*.
- [6] Mihai Dobrescu and Katerina Argyraki. “Software Dataplane Verification”. In: *Proc. NSDI’14*. NSDI’14.
- [7] Ari Fogel et al. “A General Approach to Network Configuration Analysis”. In: *NSDI*. 2015.
- [8] Rice H G. “Classes of recursively enumerable sets and their decision problems”. In: *Transactions of the American Mathematical Society* 74 (1953), pp. 358–366.
- [9] Aaron Gember-Jacobson et al. “Fast Control Plane Analysis Using an Abstract Representation”. In: *SIGCOMM*. 2016.
- [10] Kesha Hietala. *Detecting Behaviorally Equivalent Functions via Symbolic Execution*. 2016. URL: <http://hdl.handle.net/11299/181385>.
- [11] Timothy L. Hinrichs et al. “Practical Declarative Network Management”. In: *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*. WREN ’09. 2009.
- [12] Lingxiao Jiang and Zhendong Su. “Automatic Mining of Functionally Equivalent Code Fragments via Random Testing”. In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. ISSTA ’09. 2009.
- [13] Lingxiao Jiang et al. “DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones”. In: *Proceedings of the 29th International Conference on Software Engineering*. ICSE ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 96–105. ISBN: 0-7695-2828-7. DOI: 10.1109/ICSE.2007.30. URL: <http://dx.doi.org/10.1109/ICSE.2007.30>.
- [14] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. “CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code”. In: *IEEE Trans. Softw. Eng.* 28.7 (July 2002).
- [15] Peyman Kazemian, George Varghese, and Nick McKeown. “Header Space Analysis: Static Checking for Networks”. In: *Proc. NSDI’12*.
- [16] Peyman Kazemian et al. “Real Time Network Policy Checking Using Header Space Analysis”. In: *Proc. NSDI’13*.
- [17] Ahmed Khurshid et al. “VeriFlow: Verifying Network-wide Invariants in Real Time”. In: *Proc. NSDI’13*.
- [18] Hyojoon Kim et al. “Kinetic: Verifiable Dynamic Network Control”. In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015, pp. 59–72. ISBN: 978-1-931971-218. URL: <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/kim>.
- [19] Zhenmin Li et al. “CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code”. In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*. OSDI’04. 2004.
- [20] Hongqiang Harry Liu et al. “CrystalNet: Faithfully Emulating Large Production Networks”. In: *Proc. of the 26th Symposium on Operating Systems Principles (SOSP)*.
- [21] Nuno P. Lopes et al. “Checking Beliefs in Dynamic Networks”. In: *Proc. NSDI’15*.
- [22] Haohui Mai et al. “Debugging the data plane with anteater”. In: *Sigcomm*. 2011.
- [23] Nick McKeown et al. *Automatically verifying reachability and well-formedness in P4 Networks*. Tech. rep. Sept. 2016. URL: <https://www.microsoft.com/en-us/research/publication/automatically-verifying-reachability-well-formedness-p4-networks/>.
- [24] Rajdeep Mukherjee et al. “Equivalence Checking of a Floating-Point Unit Against a High-Level C Model”. In: *FM 2016: Formal Methods*. Ed. by John Fitzgerald et al. Cham: Springer International Publishing, 2016, pp. 551–558.
- [25] Ben Pfaff et al. “The Design and Implementation of Open vSwitch”. In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015, pp. 117–130. ISBN: 978-1-931971-218. URL: <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff>.
- [26] M. Popovici. “Verifying large-scale networks using NetCheck”. In: *2017 European Conference on Networks and Communications (EuCNC)*. June 2017, pp. 1–5. DOI: 10.1109/EuCNC.2017.7980647.

- [27] David A Ramos and Dawson R. Engler. “Practical, Low-effort Equivalence Verification of Real Code”. In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. CAV’11. 2011.
- [28] Mark Reitblatt et al. “FatTire: Declarative Fault Tolerance for Software-defined Networks”. In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN ’13. 2013.
- [29] Leonid Ryzhyk et al. “Correct by Construction Networks Using Stepwise Refinement”. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 683–698. ISBN: 978-1-931971-37-9. URL: <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/ryzhyk>.
- [30] Radu Stoenescu et al. “In-Net: In-network Processing for the Masses”. In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys ’15. Bordeaux, France: ACM, 2015, 23:1–23:15. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741961. URL: <http://doi.acm.org/10.1145/2741948.2741961>.
- [31] Radu Stoenescu et al. “SymNet: Scalable symbolic execution for modern networks”. In: *SIGCOMM*. 2016. DOI: 10.1145/2934872.2934881. URL: <http://doi.acm.org/10.1145/2934872.2934881>.
- [32] *Symnet Source Code Repository*. <https://github.com/nets-cs-pub-ro/Symnet/>.



ANNEX: Internal deliverable I6.3

Superfluidity H2020

I6.3: Modelling and Design for Symbolic Execution and Monitoring Tools

Costin Raiciu, Matei Popovici,
Radu Stoenescu - University Politehnica of Bucharest
Omer Gurewitz, Asaf Cohen - Ben Gurion University

July 1, 2016

1 Introduction

DOW Description: This internal deliverable will include the modelling and design for Task 6.3: the symbolic execution checking tool and the automatic monitoring and anomaly detection.

5G networks will accentuate the growing industry trend to shift away from static policies and hardware implementations to dynamic instantiation of software network functionality. Software networking functionality could be requested by the operator itself or by paying third-parties such as web content providers and mobile applications.

Superfluidity proposes the concept of a Reusable Functional Block as the API for 5G networking. Reusable Functional Blocks have a specification described in a higher level language that allows their users to compose them correctly, and can be implemented in multiple ways as long as they obey their specification: software (as a monolithic block), hardware (ASICs), or a composition of other RFBs. How can we secure the resulting 5G networks?

Enforcing network security has two major phases: the network operator specifies higher-level policies and then implements them using (low-level) networking functionality typically provided by third-party vendors. High-level policies could include access control lists (who can talk to whom), firewall rules, routing protocol configurations, and so forth. Networking functionality includes switches, routers, middle-boxes, etc. In traditional networks, high-level policies are fairly static and thus easy to manually check and deploy infrequently. Traditional networking hardware (switches, routers, simple firewalls) processes packets on custom-made ASICs that are thoroughly verified and seldom updated; such implementations give a fairly strong low-level security guarantee.

In 5G networks, network functions will be instantiated dynamically, and the network will run services configured not only by the operator, but also by third parties. Running third-party processing will be a major revenue source for operators and is

thus very attractive; however it can subject operator networks to many security risks that must be addressed. The great benefit of software network functions is that it can be easily upgraded and it avoids vendor lock-in. On the downside, diverse software with increasingly complex functionality and developed by many vendors is much more susceptible to low level exploits such as buffer overflows.

To enforce security in 5G networks SUPERFLUIDITY takes the three main directions:

1. **Describe operator policies in a high-level specification language.**
2. **Describe RFBs in a way that is amenable to static analysis.** Current informal specifications are not enough, and more precise descriptions are needed. Such descriptions could range from specifying the types of packets expected between boxes to a more complete specification in specialized languages such as SEFL [32] or NoD [17]. The language depends on the analysis we plan to perform: SEFL allows symbolic execution and is quite powerful; P4 [3] allows one to express expected header types using a finite-state-machine abstraction which makes it an ideal candidate if type safety is the only thing that is verified.
3. **Perform static analysis of RFB configurations to ensure policy is obeyed before deployment.** We will use static analysis tools to analyze network processing described as a graph of RFBs and check whether it matches the high level policies. Existing tools include Header Space Analysis [13], Network Optimized Datalog [17] or our own SymNet tool [32]. SymNet is the most powerful tool in terms of properties verified, and is easiest to use because SEFL, its associated modeling language, is imperative and simple.

SymNet can run reachability checks over network models by injecting symbolic packets and tracking their path through the network. The output is a list of paths explored in json format, and for each path the list of boxes visited, instructions executed and constrained applied to each header field (including exact values if that is the case). Using this output we can verify properties such as reachability, loop detection, header field changes, header visibility, and so forth. The operator policies can be translated into constraints on the output of SymNet, and verified (this is future work that needs to be done).
4. **Ensure that the implementation conforms to the specification at deployment time.** There is a gap between the abstract model of the network that we can statically analyze (steps above) and the actual implementation. This gap appears because it is impossible to verify large C implementations of networking code in useful time [9].

Even if we assume the implementations of networking functionality are correct, similar gap appears when multiple abstractions are applied in a network, for instance network virtualization. We give a thorough example in this document based on OpenStack Neutron, where the tenant API static analysis is fairly simple but the instantiation may be faulty because the OpenStack drivers—control plane software that instantiates the tenant networking configuration—are faulty.

To bridge this gap, we rely on multiple techniques, some of which are known and tested but have limited coverage, and some which are novel and are still subject of ongoing work. We list them here:

- **Active testing.** This is the obvious solution, and it can be guided from the results of static analysis as proposed by [39] and also implemented in SymNet. Intuitively, once the static analysis results are known, packets are generated for each path resulting from static analysis and injected in the real network; the outcome is then checked to see if it matches that predicted by static analysis. As all other active techniques, this is lightweight and useful (it helped us uncover many bugs in our SEFL models), but it is not sufficient on its own because it has poor coverage and cannot give any type of strong guarantees.
- **Monitoring and anomaly detection.** This is another runtime technique that applies machine learning to understand the standard behaviour of the network and detect attacks when the behaviour deviates from standard. A snapshot of Superfluidity work in this space is given in Section 2.
- **Static analysis at lower layers.** Symbolic execution can be run on the model, and on the lower level implementation of the model; if the resulting outputs are equivalent, then the implementation is correct. Defining equivalence is not easy, and different definitions capture different parts of the problem we want to solve—an initial exploration is provided in section 4.

In certain cases the lower level implementation is C code, which means we will need to resort to traditional symbolic execution (e.g. Klee [5]); for simple scenarios this will work, but it will not scale for complex pieces of networking. In other cases, we can also model the lower level implementation in SEFL: our OpenStack work presented in section 4, the results of running static analysis on the resulting data plane after a tenant’s configuration has been instantiated can be compared to the analysis of the abstract tenant view.

- **Automatic generation of C implementations from SEFL code.** SEFL code is memory-safe and easy to symbolically execute which allows us to prove it satisfies the high level properties of the operator. Is it possible to generate C code from these models? As SEFL is imperative, it is fairly easy to translate it to C code, and possibly manually audit the code blocks used in generation to ensure safety.

However, SEFL models that are optimized for symbolic execution have low branching factor and large constraint sets while fast networking code requires high branching factor and low constraint sets per branch (see examples from [32] and [35] for more insights into why this is the case). Is it possible to *transform* SEFL models optimized for symbolic execution into ones optimized for actual deployment? Our experience in modeling for SymNet shows a few manual techniques can be applied, but we have started to analyze automated ways to do so. A brief snapshot of this incip-

ient work that will be presented in the NetPL Sigcomm workshop can be found in 5.

2 Monitoring and detection

Network monitoring and intrusion detection are basic, indispensable tools for operators. In 5G networks their importance will only grow because of the increasing complexity and an never-ending stream of network configuration changes (i.e. NFV function instantiation and removal).

A crucial part of this work is accurate traffic classification: deciding what application is generating a given packet or set of packets.

2.1 Light-Weight Traffic Classification for the NFV platform

Network traffic classification is vital for many network management tasks such as traffic design, bandwidth allotment, accounting, security (e.g., filtering and anomaly detection), QoS and policy enforcement, etc. Obviously identifying applications is becoming harder and harder to attain as more and more applications such as video/audio streaming, social networking, and gaming are moving to the cloud, especially since many of these applications are using encrypted communication. All the more so, labeling applications in Network Functions Virtualization (NFV) platforms, where network services are running on independent hardware, while the resources are shared between many different network services, makes monitoring even more challenging, especially when enormous data volumes are traversing and processing capabilities are shared between vast number of services. Identifying applications based on passive observations of individual or streams of packets traversing the network, typically relies on two main costly procedures: *capturing and classification*.

The *capturing* procedure intercepts data packets traversing the network and stores them for further inspection. Typically, network administrators utilize monitoring tools, such as NetFlow, sFlow, IPFIX and various other packet level capturing tools to collect flow information and export it for further analysis. Nevertheless, utilizing such tools, which typically rely on substantial packet capturing, consumes precious resources, and hence hinders the scalability of the NFV. Specifically, data centers and NFV networks involve enormous data volumes, thus, the resources required for the monitoring are overwhelmed as the number of flows and link capacities grow. Moreover, utilizing complex lookup tables for finding rule matches limits the performance even further. To address this issue, statistical sampling strategies are used, which lower the CPU and storage requirements. The sampled data is gathered into flows according to predefined criteria (e.g., 5-tuple, QoS marks, VLAN (inner, outer or both), MPLS labels, etc.), on which classification is performed.

However, due to the non-uniform (typically heavy-tailed) distribution of IP flows for packets and bytes, utilizing uniform sampling techniques can result in inadequate traffic estimation. Hence, studies have suggested to utilize more sophisticated sampling techniques. Yet, these solutions require complex mechanisms and *many lookup table filtering rules* in order to obtain a representative sample set. Such mechanisms are

highly expensive and consume a lot of time and computing resources, thus, it is a great challenge to minimize the required filtering ruleset.

The *traffic classification* process labels traffic based on a predefined set of classes (e.g., applications). A commonly used classification approach is Deep Packet Inspection (DPI), which classifies traffic by inspecting the *payload* of each packet traversing the inspection point. DPI attains remarkably high accuracy for unencrypted traffic. Nonetheless, since DPI relies on the visibility of the payload to the classifier, its effectiveness diminishes with the usage of data encryption, which is the conventional wisdom, or when examining the content of the packets traversing the network is forbidden or limited due to privacy or complexity concerns. Furthermore, DPI requires knowledge of the latest syntax each application exploits in its payload, which imposes high storage and computational resources, further hindering the utilization of DPI. Consequently, utilizing Machine-Learning (ML) techniques for traffic classification, which overcome most of the aforementioned hindering factors (with an appropriate *feature set*) is an attractive solution, especially in the NVF environment.

Typically, ML involves three main steps. First, in the feature selection phase, flow statistical attributes, such as minimum, maximum or average packet size, flow duration, or inter-arrival time, by which previously unseen flows will be classified, are selected. Then, in the learning phase, the classifier is trained to map the attribute set to classes, according to the selected learning method. Finally, in the actual testing phase, unknown traffic can be classified, utilizing the rules learned. Note that every classification method may apply different priorities to different attributes, leading to different training results, hence, different performance in the testing phase.

In this study, we suggest a novel set of features, which relies on an extremely simple sampling strategy, namely, a single filtering rule to capture the required data and demands only sampling a minimal fraction of the traffic volume in Bytes (2-3%), yet results in very high classification accuracy. The suggested technique relies on the following key observation: since each application adheres to a proprietary standard message exchange, the data exchanged between two applications should follow a typical pattern which distinctively characterizes the application that generated it. Specifically, when examining the sequence of Application Protocol Data Units (APDU) exchanged between the entities running the application, one can identify an exclusive pattern which is typical to the associated application and is different from one application to the next.

For example, as illustrated in Figure 1, a typical client-server application will follow a typical sequential pattern of request and response APDUs, wherein each such client-server application is characterized by its unique control messages (application-level control messages), typical response APDUs (i.e., typical sizes and variances) and typical patterns of responses which are correlated with the application's proprietary requests.

Inspired by the aforementioned observations, one can utilize these distinct patterns within the network's premises to identify the generating application. Moreover, since the suggested system tracks the traffic generated at the Application layer, such application fingerprints are completely transparent to lower layers and especially transparent to side effects at these layers such as fragmentation, retransmissions, timing, congestion, packet loss, location, delays, etc. Figure 2 shows a sample of ten accumulated APDUs (a-APDU) exchanges for three different applications (SSH, RDP and eMule).

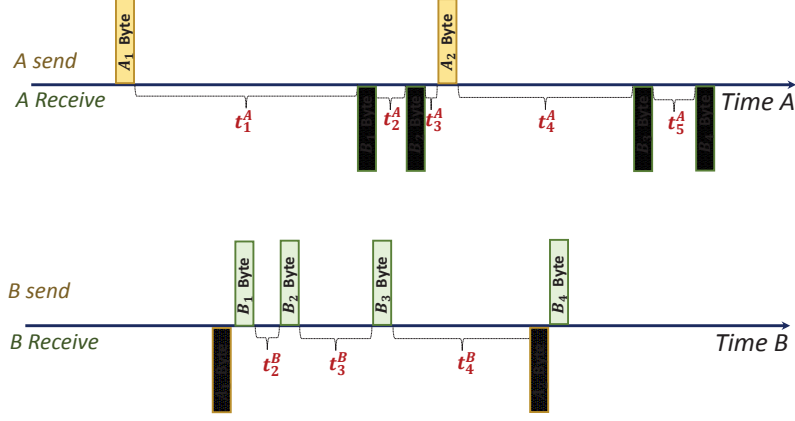


Figure 1: Illustrated example of data exchanged between two hosts

The graph depicts the interlaced number of Bytes sent by A (X-axis) vs. the number of Bytes sent by B (Y-axis) on a log-scale, based on captured TCP ACKs. For example, as can be seen in the figure, the difference between the first and second SSH entries indicates that in the first a-APDU A has sent around 1000 Bytes and B has sent around 500 Bytes; the difference between the second and third entries indicates that B has sent around 100 Bytes while A has sent no data (a-APDU (0,100)). Note that as stated earlier since each fingerprint entry only counts the accumulated number of bytes traversed throughout the network since the last change in direction, i.e., no time-stamps, number of exchanged packets or duplicates, our fingerprints are not sensitive to network noise (e.g., retransmissions, fragmentations, inter-arrival time, losses, etc.).

The challenge is hence, twofold. First, identifying the unique meta-data patterns (signatures) constructed by each application. Note that the application pattern recognition should be performed continuously in order to identify new applications and/or new legitimate patterns generated by already known applications. Furthermore, a specific application can generate several typical patterns, yet each pattern characterizes a specific application. Second, reconstructing the original APDU pattern, generated by the Application layer prior to the intervention of the lower layers, and especially, prior to the impact of network side effects. Note that this task can be done by acquiring all the log files or traces generated by each application, or by collecting and assembling all fragments generated by the two participants. However, since the classification is performed as a virtual network service, without relying on any collaboration from other virtual services or the hosts, and necessarily in real time, on an enormous number of applications traversing the network simultaneously, such solutions hinder the great potential of the NFV architecture.

We address both challenges by introducing a simple strategy that samples a minimal amount of traffic, yet attains a representative sample which enables characterization as well as reconstruction and classification of the unique APDU exchange between

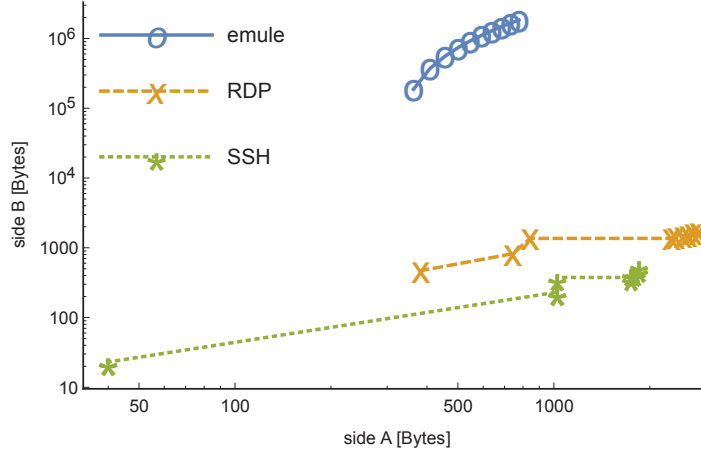


Figure 2: Ten a-APDU exchanges for three different applications: SSH, RDP and eMule. The axes depict the interlaced number of Bytes sent by A (X-axis) vs. the number of Bytes sent by B (Y-axis) in a flow, on a logarithmic scale.

each participating application entity. We devise a unique low-dimension attribute set, which allows the utilization of ML techniques to classify traffic while the flows are ongoing, with high accuracy for a large variety of applications. We show that our scheme requires very low sampling rate for TCP traffic and a slightly higher one for UDP traffic, yet provides very accurate traffic classification for both kinds of traffic.

Specifically, we extract TCP flow attributes only by sampling *zero-length packets*, i.e., packets that contain control bits, but do not contain any payload (e.g., SYN, ACK, etc.). Although zero-length packets are apparently frequent, we show experimentally that they comprise only 2-3% of the total traffic volume in bytes. Moreover, zero-length packets contain critical information on the connection state, yet, are easy to process, and more importantly, are resilient to the network parameters, such as congestion, fragmentation, delays, retransmissions, duplications and losses. Consequently, since each application behaves differently with respect to the amount of data it requires in order to work properly, it is likely to extract unique fingerprints that produce accurate classification for a large variety of applications. Leveraging the same approach, we extend our method to handle UDP flows as well; instead of inspecting the ACKs we inspect the UDP length field of each monitored flow.

In Figure 3 we depict the algorithm flowchart for TCP traffic. Specifically, upon a zero-length packet arrival of unseen flow, the collector creates a new a-APDU record and stores it in the database, where the flow's 4-tuple is used as a key. Each a-APDU record comprises the first and last ACK# and SEQ# which indicates on the a-APDU boundaries, and the direction of the APDU relative to the flow initiator, which is used for SEQ# and ACK# alignment. When a zero-length packet arrives, and its 4-tuple already exists in the collector's database, the collector first checks that the ACK# and SEQ# are relevant (e.g., this is not a retransmission, or out of order packet) then it

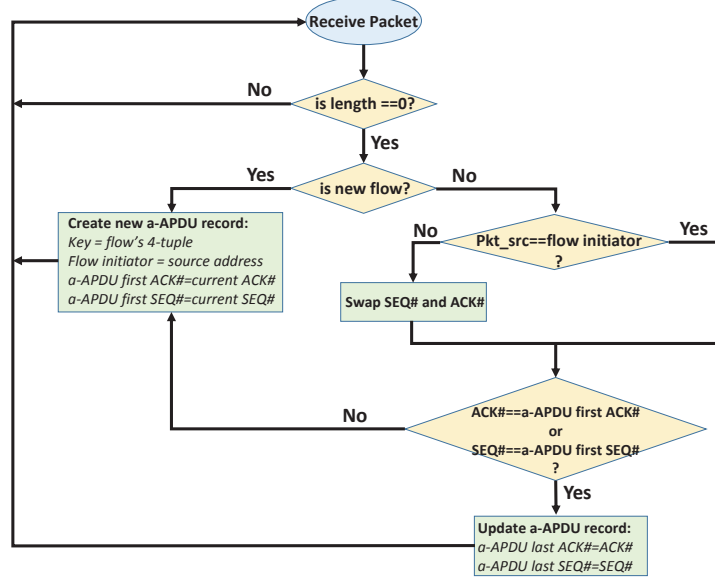


Figure 3: Algorithm flowchart.

checks if either the current packet SEQ# or ACK# is greater than the stored SEQ# or ACK#, respectively. Note that if only one of these fields grows and it is the same field as the last update for this particular flow, we need only to *update* the last ACK# and SEQ#, as no information passed from the other side during this period. Otherwise, the collector creates a new a-APDU record with first and last ACK# and SEQ# equal to the value of the packet's ACK# and SEQ#, respectively. Then, the collector adds the new a-APDU to the corresponding flow entry, and sends the updated a-APDU sequence for classification.

3 Detection of Anomalies in Virtual Network Functions

Cyber-attacks are a disturbing security threat in today's communication- and computer-based systems. They affect a wide range of domains, including electricity and water infrastructures, financial markets, medicine and healthcare, armies, enterprises and universities around the world.

Detecting and blocking such attacks, however, is becoming more and more challenging. While older computer viruses could be easily identified by locating known pieces of their code on the computer hard-drive or in email attachments, modern attacks are distributed [36], they utilize legitimate protocols and communication channels [33, 12, 8, 22, 11, 34, 4], and constantly change their code, servers, and attack strategies [31, 29].

As a result, the current literature includes numerous *anomaly detection techniques*,

which focus on detecting abnormal behaviour, rather than locating signatures of known attacks. Indeed, anomaly detection has been found useful in several cases, being able to detect *zero-day* attacks, previously unknown viruses and Trojans, and even unknown program behaviour on embedded devices [40].

Nevertheless, currently known detection techniques are still based on a few hindering assumptions, restricting our ability to cope with some of the current and future attacks. First, for successful anomaly detection, one must have a good model for the *normal data*. That is, when learning, for example, normal tenant behaviour, one has to correctly and robustly capture the essence of such behaviour, in order to identify deviations on the one hand, yet not to over-fit on the other, as this would result in high false alarm rates. Most of the known techniques today assume some statistical model for the data, and, in essence, estimate the parameters of such a model. A few examples include Markov [22] or ARMA models [6]. Simpler (only statistical-modeling wise) techniques base their detection on memoryless features, e.g., frequencies of events or proportions of packets of a given size, etc. For example, in [38], an Evil-Twin attack is successfully detected by identifying the average number of wireless hops. However, the key questions that arise are the following. What if, as detection systems designers, we *do not know* of any good statistical model for the normal data? Moreover, what if *there is no statistical model*? Are memoryless features, which disregard *the context* of the events in the system always sufficient? What if an anomalous behaviour can only be identified by the *order of the events*, and not necessarily by a single anomalous value of a certain feature?

A second important aspect is the detection complexity and the captures necessary for the anomaly detection system to perform properly. Managing a modern network, for example, requires employing numerous monitoring tools, and collecting huge amounts of data every second. Clearly, a deep inspection of all data is not feasible in most circumstances. What are, then, the main features of the data which can be *efficiently captured*, yet allow for good enough anomaly detection?

3.0.1 Anomaly Detection In SUPERFLUIDITY

To monitor system activity and alert in cases of mal-usage and malicious activity, SUPERFLUIDITY requires novel monitoring and detection tools. As mentioned above, such tools should be applicable without any prior knowledge on the normal behaviour of tenants in the system, and definitely without any assumption regarding the abnormal or malicious behaviour. Specifically, in SUPERFLUIDITY we will monitor derivatives of the tenants activities, such as memory requirements, processing time, traffic patterns and communication requirements, and use this data, without any prior model, to learn the normal activity of tenants and sub-systems. Note that this way monitoring can be done without sacrificing tenants privacy, and, in fact, will be possible even in cases of encrypted data or protected computation and storage. Then, using the learned data, activity in the system will be continuously compared to the normal structure. This way, the monitoring tools in SUPERFLUIDITY will be able to alert for abnormal behaviour. Moreover, changing trends in the system behaviour can be identified, allowing us to better prepare for times of higher demand, required re-allocation of resources and re-placements of services.

Therefore, in task 6.3, we developed a novel universal anomaly detection technique for VNFs, which does not require any a-priori information about neither the normal behaviour patterns nor the abnormal ones, yet efficiently learns the normal behaviour in order to generate a statistical model to which tested behaviour can be compared. The technique is based on the celebrated Lempel Ziv algorithm [41], the probability assignment induced by the prediction algorithm [10], and the learning technique we initiated in [30]. The technique inherits key useful aspects of the universal compression algorithm, that is, it performs optimally (in terms of estimating the model) even when there is no statistical model, and is extremely efficient to implement in practice. It offers a new look on the way to use data in the classification process, suggesting the *context* of the data sequence as the key characteristic used in the classification, rather than actual values. That is, the system does not rely on memoryless features of the data, such as specific times, sizes or other signatures. In contrast, it builds a *context tree* for the learned data. Then, when a new data sequence is tested, the order of values or events in it has the main impact on the classification performance.

3.1 Technical Preliminaries

Classification refers to the problem of labeling unknown (new) instances to the most appropriate class among a set of (known) predefined classes. When the underlying probability distributions for the classes $\{p_i(\cdot)\}_{i=1}^M$ are known, and we wish to decide which generated a given data sequence \mathbf{y} , a decision rule of the form

$$\hat{i} = \operatorname{argmax}_{1 \leq i \leq M} p_i(\mathbf{y})$$

is optimal in the sense of minimizing the probability of error [16]. In unary-class classification, however, information is available only on one type of instances, namely, there is only one class, $p(\cdot)$. The goal, then, may be to either identify instances belonging to this class, or, taking the opposite viewpoint, usually referred to as *anomaly detection*, identify instances which *do not belong to the class*.

Specifically, assume for now a *given* probability distribution (of a single class) $p(\cdot)$. From this point on, we refer to this class as *normal*. In anomaly detection, the goal is then to identify whether a new data instance \mathbf{y} belongs to the normal class, or, alternatively, is *anomalous*. Since, in most applications, the anomalous instances are threats one wishes to identify, we refer to a correct identification of an anomalous \mathbf{y} as *detection* and for an incorrect identification of normal data as *false alarm*. The optimal decision rule in terms of maximizing the detection probability given a fixed false alarm probability (in the Neyman-Pearson sense) is to compare $p(\mathbf{y})$ to a *threshold*, and decide that \mathbf{y} is normal if $p(\mathbf{y})$ is above the threshold and anomalous otherwise [20]. The threshold is determined according to the required false alarm probability.

In practice, the probability distributions governing the data (either multiple classes or a single one) are unknown, and there is only a limited amount of data to learn from. Furthermore, in most security-related applications, only few, if any at all, anomalous instances to learn from exist, yet more instances of normal behaviour are available. This asymmetry strengthens the need to take the anomaly detection approach in such circumstances, that is, build a behavioural model *only based on the normal instances*, and classify any instance deviating from that model as anomalous [7].

Thus, a reasonable approach is to estimate the probability distribution of normal data using the previously observed sequences and use the resulting estimate, $\hat{p}(\cdot)$, in the detection algorithm. Note, however, that the estimation problem differs significantly if a statistical model for the normal data is given, e.g., i.i.d. or Markovian of a certain order, in which case only a few parameters should be estimated; if, in a more complex scenario, the only knowledge is that the sequences are governed by *some* stationary and ergodic source; or, in the “worst” case, the data constitutes of *individual sequences*, that is, deterministic sequences with no pre-defined statistical model.

3.1.1 Universal Probability Assignment

The Lempel Ziv algorithm [41], LZ78, is a universal compression algorithm with a vanishing redundancy. Consequently, it can also be used as an optimal *universal prediction* algorithm [10], using the appropriate probability assignment. We briefly describe the compression method and the associated probability assignment algorithm.

The LZ78 algorithm is a dictionary-based compression method. For a given sequence of data symbols, a dictionary of phrases parsed from that sequence is constructed based on the incremental parsing process as follows. At the beginning the dictionary is empty. Then, during each step of the algorithm, the smallest prefix of consecutive data symbols not yet seen, i.e., which does not exist in the dictionary, is parsed and added to the dictionary. By that, each phrase is a unique phrase in the dictionary, that may extend a previously seen phrase by one symbol.

Given a sequence $s_1^n = (s_1 s_2 \dots s_n)$, a *parsed phrase*, P , is the smallest prefix of consecutive data symbols that has not been seen yet. This can also be considered as suffix concatenation of symbol s_i (from the sequence) with a previously seen phrase P' (from the dictionary), i.e., $P = (P' s_i)$. A *dictionary*, D , is a collection of all distinct phrases parsed from a given data sequences s_1^n , i.e., $D = \{P_1, P_2, \dots, P_i, \dots, P_n\}$. For example, the sequence *aabdbbbacbbda* is parsed as *a|ab|d|b|ba|c|bb|da|*.

A common representation of the dictionary is a rooted-tree, where each phrase in the dictionary is represented as a path from the root to an internal node in the tree according to the set of symbols the phrase consists of. In addition, leaf-nodes are added as suffix for each phrase in the tree. A statistical model can be defined for a given data sequences during the construction of a phrase-tree [10], as described next.

At the beginning, an initial tree is constructed including only a root node and k leaf-nodes as its children, where k is the size of the alphabet. Then, for each new phrase parsed from a sequence, the tree is traversed, starting from the root, following the set of symbols the phrase consists of, and ending at the appropriate leaf-node. Once a leaf-node is reached, the tree is extended at this point by adding all the symbols from the alphabet as immediate children nodes to that leaf, making it an internal node. In order to define a statistical model, each node in the tree, except for the root node, maintains a node traversal counter, where each leaf-node’s counter is set to 1 and each internal node’s counter is equal to the sum of its immediate children’s counters.

For a probability assignment, as all leaf-nodes’ counters are set to 1, they are assumed uniformly distributed with a probability $1/i$, where i is the total number of leaf-nodes. Each internal node’s probability is defined as the sum of its immediate children’s probabilities, which also equals the ratio between its counter and current i .

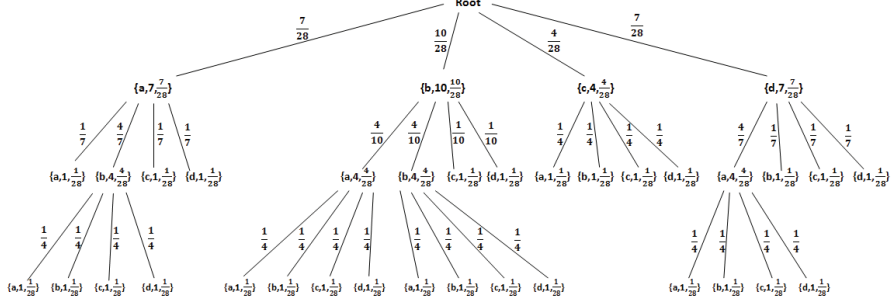


Figure 4: An LZ78 Statistical Model for sequence “aabdbbacbbda”.

For example, Figure 4 demonstrates the resulting statistical model for the sequence “aabdbbacbbda”. Each node in the tree is represented by the 3-tuple {symbol, counter, probability}. In addition, the probability of an edge is defined by dividing the nodes’ probabilities. Note that the probabilities of edges connected directly to the root are equal to the appropriate root-children’s counter divided by the total number of leaf-nodes, i , at each step of the algorithm. The probability of a phrase $P_i \in D$ is calculated by multiplying the probabilities of the edges along the path defined by the symbols of P_i . Moreover, note that for each phrase P_i , there exist a specific node in the tree whose probability represents the probability of that phrase. For instance, from the example shown in Figure 4, it can be seen that $P(ba) = \frac{10}{28} \times \frac{4}{10} = \frac{4}{28}$. Considering a sequence S , if during the traversal a leaf-node is reached before all the symbols of S are finished then the traversal return to the root and continue until all the symbols of that sequence are consumed [21]. For example, the probability of the sequence “bdca” given the same statistical model above, is defined as the following traversal probabilities multiplication: Root \rightarrow b \rightarrow d \rightarrow Root \rightarrow c \rightarrow a and is calculated as:

$$P(bdca|M_{aabdbbacbbda}) = \frac{10}{28} \times \frac{1}{10} \times \frac{4}{28} \times \frac{1}{4} = \frac{1}{784}.$$

This stems from the conditional probability $\hat{P}(s_{t+1}|s_1^t)$, where s_{t+1} is the next symbol after the (sub-)sequence s_1^t , which is calculated as the ratio between the counter of symbol s_{t+1} and the counter of symbol s_t . We consider s_1^t as *the context* of s_{t+1} at time $t + 1$.

3.2 Anomaly Detection Via Universal Probability Assignment

We now describe the building blocks of the anomaly detection system.

3.2.1 Preprocessing

A data sequence on which the anomaly detection algorithm operates is simply a sequence of values over some finite or infinite alphabet. Such a sequence may represent timing of events (e.g., times a certain service was requested), amounts of memory

required, or any other sequence of values. The strength of the algorithm is in its generality - the ability to adapt the algorithm to various data sequences, simply by changing the preprocessing stage. To keep this description in context, consider a certain service a tenant requests, e.g., networking, Input/Output or a usage of a certain piece of hardware. We will keep track of the *timings* of these requests.

The i th event, denoted by $e_{i,T}$, is defined by a tuple

$$e_{i,T} = (t_i, \dots),$$

where t_i is the time the event occurred for tenant T, followed by maybe some additional data. A *flow*, denoted by f_T , is series of events for tenant T, sorted by their time of occurrence, t_i . That is,

$$f_T = \{e_{1,T}, e_{2,T}, \dots, e_{n,T}\}.$$

For actual learning and testing, it is not required to use all features (fields) in the data. Good detection capabilities can be achieved even when focusing on a single feature. For example, timing data can be characterized by the *difference between two consecutive events* of the same flow, denoted by Time-Difference (TD) and defined by

$$TD_{i,T} = e_{i+1,T}(t_{i+1}) - e_{i,T}(t_i).$$

Consequently, a *single-feature data sequence* is a serialization of one of the features, e.g., with respect to Time-Difference, a sequence is defined as:

$$f_{T,TD} = \{e_{2,T}(t_2) - e_{1,T}(t_1), e_{3,T}(t_3) - e_{2,T}(t_2), \dots, e_{n,T}(t_n) - e_{n-1,T}(t_{n-1})\}.$$

It is important to mention that the above procedure may result in a sequence over a *large alphabet*. For example, times may be given with a very high precision. Such a high alphabet size may significantly increase the learning complexity. Hence, to reduce the range of values, quantization should be performed. For k quantization levels, a set of k centroids $\{c_1, c_2, c_3, \dots, c_k\}$, is used. The centroids are extracted from the available data during the training phase. Clearly, the number of centroids and the method for extracting them may affect the overall results.

Sequences of the above form, and $f_{T,TD}$ as an example (after quantization), *are the sequences we will use for both learning and, later on, detection*. While simple and one-dimensional, in the sense of tracking only a single feature, in this case, the time differences, these sequence are powerful as they capture *the context of the events for the tenant*. In other words, we will see that what matters the most is not necessarily a specific value of a feature, i.e., a single time difference being higher or lower, but, rather, the sequence of values and their relations. For example, when a tenant performs a certain computation, its memory access pattern might have a certain structure. This structure encompasses the relations between the time it takes to compute something and the time it takes to access memory. However, when a tenant is accessing memory only to identify cache misses, and learn about the memory usage of another tenant, its memory access pattern can be completely different.

Next, based on the sequences above, we describe the learning phase, where a model for the normal behaviour is build, and the testing phase, where *new sequences* are tested against this model in order to decide whether they are anomalous or not.

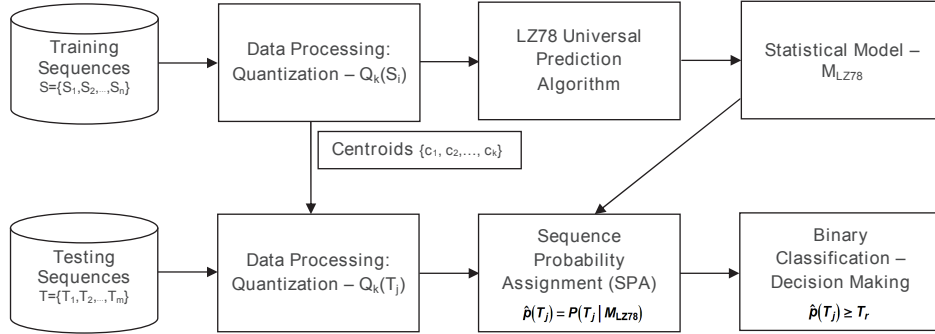


Figure 5: A Classification Model based on the LZ78 universal compression algorithm and its associated probability assignment.

3.3 Learning and Testing

The classification model is divided into a *learning phase* and a *testing phase*. In the learning phase, available data of normal behaviour is preprocessed (according to the concepts described in the previous section) and an LZ78 probability assignment tree is built, to serve as the statistical model for the normal data. In the testing phase, or the actual operation of the detection system, new, unclassified data arrives. This data goes through *the same preprocessing*, resulting in quantised sequences over the same alphabet. The sequences are then tested against the model. Figure 5 depicts the key building blocks.

Specifically, in the learning phase, an LZ78 statistical model is built according to the algorithm in Section 3.1.1, based on a given training set of discrete, quantized sequences over a finite alphabet $S = \{S_1, S_2, \dots, S_k\}$. Training is done only on normal, benign behaviour. Moreover, it is important to note that in practical cases, one might not have a long enough sequence from a single, normal flow. In such cases, normal sequences can be concatenated together to generate one long sequence from which the tree is built. The size of the alphabet has the following impact. On the one hand, small alphabet size results in low complexity and a robust model (without over fitting). On the other hand, it might group together different types of events, losing some of the important context in the data (e.g., treating minor differences as equal, hence losing subtle changes which might reflect distinct phenomena).

In the testing phase, first, each suspected testing sequence is separately quantized using *the same quantization method and the same set of centroids* $\{c_1, c_2, \dots, c_k\}$ which were extracted in the learning phase. This is a critical point in the testing phase, as trying to re-calculate optimal quantization for the tested sequences might result in most of them wrongly classified as anomalous. Then, the probability of each suspected sequence (testing sequence) T_j , from a given testing set $T = \{T_1, T_2, \dots, T_m\}$ is estimated based on the constructed statistical model and classified respective to a pre-defined threshold T_r . Namely, the probability of each testing sequence, $\hat{p}(T_j)$, is estimated using the sequential probability assignment technique described in Section 3.1.1 *given the LZ78 statistical model built in the learning phase*. Testing sequences for

which $\hat{p}(T_j)$ is greater than or equal to T_r , are classified as normal (as they “fit” the model) while a lower than threshold value is classified as anomalous.

It is important to note that for the testing phase the design of the classifier has a few degrees of freedom, based on the required trade-off between complexity, accuracy, and the availability of data. That is, while the optimal decision should be made based on a long enough sequence, in practice, one may combine a few decisions together, based on a few sequences which are known to belong to the same tenant. For example, when a tenant is suspected in malicious behaviour, one might test a few sequences of its data, even if belonging distant, possibly unrated time spans, or based on different features, and make a decision based on all results together.

3.4 Performance Evaluation

The performance of the classifier is measured by the false alarm and detection probabilities, also known as false positive rate (FPR) or Type 1 error and true positive rate (TPR), respectively, and is usually demonstrated using a Receiver Operating Characteristic (ROC) curve. The false alarm probability (or ratio) reflects the number of negative (in this case, normal) instances incorrectly classified as positive (anomalous) in proportion to the total number of negatives in the test, whereas hit detection ratio measures the proportion between the number of positive instances correctly classified and the total number of positives in the test. The ROC curve is generated with respect to a set of thresholds. Each threshold results in a point on the ROC curve, that is, it results in fixed false alarm and detection probabilities. Changing the threshold changes the trade-off between the two probabilities.

3.5 Adaptive Coding and Parallelization

The model described above includes a serial, single pass process of learning and modeling. That is, it builds a single LZ tree which serves as the model. One might wonder if the LZ tree can be updated or enhanced over time. For this, we refer the reader to a few adaptive-LZ techniques, which allow fast rebuilding of the tree in case the *normal data is non-stationary* and changes its form, e.g., [27]. Moreover, the current literature also includes a few algorithms for fast and parallel construction of the tree, e.g., [14].

3.6 Anomaly Detection as a Virtual Network Function

Till now, we have described the anomaly detection algorithm and how it can be used to detect anomalous tenant behaviour and alert in cases of mal-usage or changes and trends in normal behaviour. However, it is important to develop the learning and detection sub-systems of this thread *as virtual network functions* in their own right.

Specifically, consider the learning phase of this algorithm. In this phase, one is required to monitor tenant behaviour and to build a sequence of events for it. Clearly, this should be done either for one specific tenant, or for a group of tenants performing similar tasks. However, as these tenants might migrate, the monitoring tool needs to migrate as well. Moreover, one might choose not to follow a certain tenant, but locate the monitoring tool *at a place which is more convenient system-wise*, as long as the

new location can allow for the same learning process. New tools should be developed to decide how to migrate the monitoring tool, where to migrate to and how to consolidate the data it collects. The same holds for detection. An orchestrator needs to decide where to probe for mal-usage, how often to do it, and, maybe, do it distributively and using aggregated detection techniques.

4 Enabling security in OpenStack Neutron (based on Lanman 2016 paper)

The cloud is taking over the world of computing. Public clouds such as Amazon EC2, Microsoft Azure or Rackspace are widely used, and smaller clouds are being built pretty much everywhere. Network operators are building miniature clouds in their core networks (e.g. DT is deploying racks collocated with PoPs) while mobile operators are deploying processing close to the edge to enable mobile edge computing [19]. Deploying a cloud is no easy task. Major public clouds providers have each developed their own custom cloud management software, but the software is deployment-specific and not available publicly. New cloud players are very numerous and eyeing smaller deployments; having each of them develop cloud software makes no sense.

OpenStack is the leading community effort to build a production-quality, open source platform that enables building public and private clouds with ease. OpenStack has a lot of momentum, with major companies investing human and capital resources, and is reaching maturity. Hundreds of OpenStack clouds have already been deployed [2]. We focus on Neutron, the networking component of OpenStack, that allows users to specify their high-level networking configuration and deploys it. Neutron is notoriously unreliable, to the point where it has become known as the weakest link in OpenStack and bashed in popular media by company executives [26]. Neutron has certainly improved recently, but it is still far from perfect.

In this position paper we propose to use network static analysis, in particular symbolic execution [32], to improve Neutron. Rather than provide a definitive solution, we provide a high level approach to solving Neutron's woes. Our key idea is to use symbolic execution to analyze the properties of a) the tenant virtual network configuration before deployment and b) the actual network dataplane after deployment.

We have built a prototype to showcase our approach and check its validity, and performed a preliminary evaluation. Our initial results are promising: verification is fast (seconds) and can detect common problems with Neutron deployments.

4.1 OpenStack Networking with Neutron

Network virtualization in OpenStack is enabled by Neutron [23]. It offers an API to tenants allowing the creation of virtual networks that are decoupled from the underlying networking topology and the configuration chosen for deployment. The tenant network is then instantiated on the physical topology, and this mapping is influenced by the way the cloud provider has deployed OpenStack. Neutron layering is captured in Figure 6 and it aims to achieve the following goals:

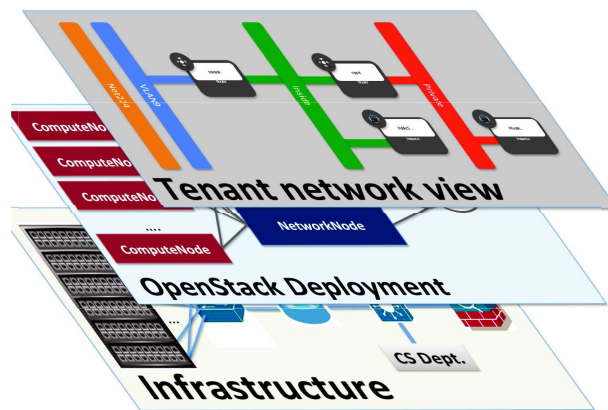


Figure 6: OpenStack Networking Layers.

- **Policy Compliance.** Neutron aims to allow tenants to configure networks that satisfy their high-level security policies, for instance separation of public and back-end traffic, reachability, etc.
- **Implementation Correctness.** The properties of the virtual network configured by the tenant should be matched by the actual deployment. For instance, Internet packets that can reach an instance in the virtual network should also reach the instance in the instantiated network.
- **Traffic Isolation.** A tenant's traffic should not reach other tenants unless Neutron is explicitly configured to do so.
- **Performance.** Neutron must allow cloud providers and tenants to effectively utilize fast interconnects including 40Gbps and 100Gbps Ethernet.

Achieving all these properties simultaneously is very tricky. Achieving tenant policy compliance appears simple: the tenant only has to correctly configure its virtual network (given the appropriate configurationAPI), however tenants are not networking professionals usually, so many configuration errors are possible. Furthermore, the short and inexpensive configure-deploy cycle facilitates the introduction of configuration bugs.

Implementation correctness and traffic isolation are achieved through coding best practices in the open-source community; however this implies that only very popular approaches will be heavily scrutinized, and many bugs will exist in less popular code. Achieving correctness and isolation in the context of proprietary drivers for third-party networking hardware is even more challenging. Even with code review, there is no guarantee that these properties are met in practice. Finally, achieving high networking performance implies using pass-through technologies such as SR-IOV for virtual machines, and relying on networking hardware to implement Neutron. SR-IOV traffic bypasses the local hypervisor stack (e.g. iptables and Openvswitch) and only basic security is provided, meaning that other tenant-specified functionality (such as fire-

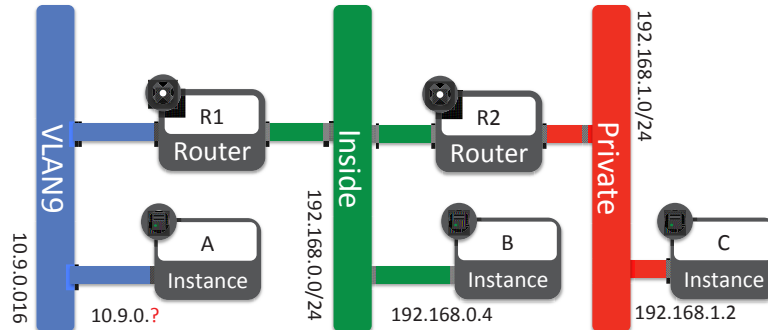


Figure 7: Example of a tenant network topology in the Horizon GUI.

walling) may not be provided. We now provide a more detailed view of OpenStack networking and highlight the difficulties when trying to meet the OpenStack goals.

Tenant network view. Tenants use an virtual network view to configure the way their instances are connected. To this end they can use layer 2 networks (flat or VLANs), routers, firewalls, VPNs and load balancers. In more detail, tenants can use the Horizon GUI or the Neutron API as follows:

- The simplest choice is to use a flat network where virtual machine instances are assigned IP addresses from a DHCP server run by the cloud provider. The DHCP server also provides a gateway for outgoing connections and performs network address translation. Incoming connections are dropped by default.
- Create VLAN(s) and associated subnet(s) where the connected instances are assigned, at configuration time, distinct IP addresses from the subnet's range. One instance can be connected (have interfaces in) multiple VLANs.
- Create routers that can interconnect different VLANs and provide Internet connectivity. All addresses assigned (either with subnets or DHCP) are private and thus not reachable from the Internet. Outgoing Internet connectivity can be provided by using NAT.
- Assign floating IPs if incoming connectivity is desired (e.g. for the tenant to be able to ssh into its instance). A floating IP is a public IP address that is associated to a private IP address of the tenant. Neutron perform address translation between the floating address and the private one, transparent to the VM.
- Specify firewall rules to be applied to specific ports.
- Further constructs include VPNs and traffic load balancing, and this list is likely to increase in the future.

In Figure 7 we show an example tenant networking configuration in the Horizon GUI of our university's OpenStack deployment. The tenant wants to deploy a web server connected to a database server. Its policy is that that its web server should be

publicly accessible on port 80, and that the database is only reachable from the web server, and not the Internet.

The tenant configures two VM instances: B will run the web server and is connected to the green VLAN, and C is the database server and is connected to the red VLAN. The two VLANs are connected via router R2, and the green VLAN is connected via router R1 to the Internet. R1 is setup as an Internet gateway. The blue VLAN (VLAN9) is created automatically by this particular OpenStack deployment and is where the Internet gateway resides. The tenant also starts A for testing purposes and attaches it to VLAN9.

Is this network configuration a correct implementation of this tenant's policy? An experienced OpenStack network administrator will, most likely, be able to debug this configuration quite easily and find that, for instance, there is no incoming connectivity to the web server since a floating IP has not been assigned. The average OpenStack tenant, however, will not be a networking specialist. Such a person needs a fairly large set of skills: they need to be able to create a VM image, they need to install and manage various servers (e.g. web and database), setup the tenant network and, finally, develop their application logic. In the pre-cloud era, multiple people were needed maintain such a website: e.g. a networking administrator, a web admin, a web developer. In the cloud era, it is possible (and expected) that a single person will fulfill all these roles, but they will not be networking experts.

Ensuring that a tenant policy is met by a network configuration requires more than manual debugging - we need tools that help the tenant quickly find the problems and fix them.

Cloud provider view of networking. When deploying Neutron, the cloud provider has to meet the above goals (correctness, isolation, performance) in the context of its local network policy, and with the added constraint of isolating tenant traffic from local traffic and treating tenant traffic as "outside" traffic when deciding access to local machines.

When the tenant starts the deployment of its configuration, virtual machine instances are placed on the available OpenStack Compute Nodes and the tenant networking configuration is instantiated; the instantiation depends on the way the cloud provider has deployed Neutron.

One of the most popular networking deployments is to use Openvswitch [25] on every Compute Node, use iptables for firewalling and have a Network Node running on one of the servers to implement NAT and routing. Traffic leaving from the Compute Nodes is encapsulated in VXLAN tunnels and carried to the Network Node, which also enables Internet connectivity. This configuration meets all the requirements, except the performance one.

There is however great flexibility in how a cloud provider can configure its network to work with Neutron, including:

- Using fault-tolerant Network Node implementations called VRRP.
- Deploying routing and firewalling on every Compute Node, in a configuration called DVR.

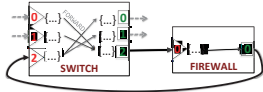


Figure 8: SymNet network models: elements and interconnections

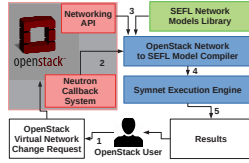


Figure 9: Verifying the tenant network

- Using hardware switch support and VLANs to ensure the tenant traffic isolation instead of VXLAN.
- Using merchant hardware to implement routing and firewalling (called firewall as a service).

A performance-oriented deployment would use SR-IOV for each tenant, bypassing the hypervisor stack, and apply VLAN encapsulation on the NIC. VLAN support in switches would then be used to carry traffic to a hardware firewall (e.g. a CISCO ASA box), where firewall rules belonging to the cloud provider and tenants would be applied. Routing between VLANs and NATting could also be implemented in hardware. In such a deployment, ensuring the isolation and correctness properties hold is trickier: tenant firewall rules could clash or overlap with provider rules, and the correct order in which they should be applied is not obvious. Installing a set of predefined rules for all tenants is feasible, however allowing per-tenant firewall policies is difficult to do.

This versatility of Neutron makes it adaptable for a wide range of uses and requirements. However, it also raises questions about the correctness of any non-trivial individual deployment, especially when less scrutinized third-party software or hardware are used.

4.2 Symbolic Network Execution

We propose to use static network analysis to improve Neutron. There are many static network analysis tools such as [37, 18, 13, 17, 24, 32]; they all require as input a model of network functionality including the processing performed in different boxes, such as switches and routing, as well as a snapshot of the network state, for instance router forwarding table snapshots or switch dynamic MAC tables. Then, the tools “simulate” what happens when certain packets are injected at different parts of the network: given a packet with specific header fields, static analysis tools tracks the path of the packet through the network and the evolution of its header fields.

The strength of static analysis stems from its ability to quickly test a wide range of possible packets (e.g. all possible headers destined to a server) without having to iteratively test all possible combinations of concrete header fields. How this is achieved

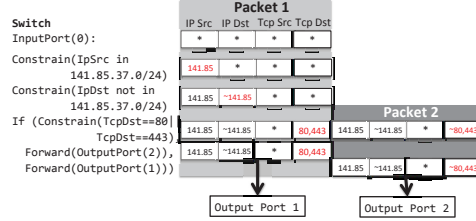


Figure 10: Symbolic execution example: injecting a symbolic packet in the switch model on input port 0. Two packets are generated with different constraints, one that exits on port 1 and one on port 2.

depends on the tool being used. The different tools offer different tradeoffs in terms of speed of analysis and properties checked, and an overview of all these tools can be found in [32]. A very good option is to use network symbolic execution, as enabled by the SymNet symbolic execution tool. We use SymNet in this paper and provide a brief overview below; please refer to [32] for more details.

In SymNet, network boxes are modeled as modular elements having an arbitrary number of input and output ports. Network links are modeled as directed edges between the output port of an element to an input port of another one. To describe the functionality of a box, each port has associated a set of instructions that are executed when a packet reaches that port. The set of instructions associated to each port is written in a language called SEFL and described in detail in [32]. SEFL is a simple imperative programming language and offers usual instructions e.g. assignment, if and basic expressions (addition, subtraction). SEFL is optimized to allow scalable network symbolic execution as follows:

- The `constrain` instruction adds restrictions on header fields, such as firewall rules.
- The `forward` instruction makes a packet go to a specified output port.
- The `fork(p1, p2, ...)` instruction sends a copy of the packet to each of the specified output ports (p1, p2, ...).
- There are no unbounded loops in SEFL.

We give an example in Figure 8 where we have two network elements: a three-port Openflow switch and a firewall connected to port 2 of the switch. Element input ports are numbered in red and shown as triangles; output ports are numbered in green, and shown as squares. Port 0 is connected to an inside network, and port 1 is connected to the Internet. The configuration aims to ensure that all packets are checked by the firewall, and performs ingress filtering for the ports connected to the two networks. Packets from the firewall port are sent to the local network or the Internet based on their destination address.

```
InputPort(0):
  Constrain(IpSrc in 141.85.37.0/24)
  Constrain(IpDst not in 141.85.37.0/24)
  If (Constrain(TcpDst==80 || TcpDst==443),
    Forward(OutputPort(2)),
    Forward(OutputPort(1)))
```

```
InputPort(1):
  Constrain(IpSrc not in 141.85.37.0/24)
  Constrain(IpDst in 141.85.37.0/24)
  If (Constrain(TcpSrc==80 || TcpSrc==443),
    Forward(OutputPort(2)),
    Forward(OutputPort(0)))
```

```
InputPort(2):
```

```

If (Constrain(IpDst in 141.85.37.0/24),
    Forward(OutputPort(0)),
    Forward(OutputPort(1)))

```

Below we provide the model for the firewall which only allows HTTP traffic from our local network and the associated return traffic. To keep per-flow state the model creates a metadata called `FirewallState` and sets in the packet. When the return traffic arrives, it will have the same variable set and will be allowed through.

```

InputPort(0):
If (Constrain(FirewallState==1),
    Forward(OutputPort(0)), //allow seen flows
    InstructionBlock(//outgoing HTTP traffic
        Constrain(IpSrc in 141.85.0.0/16),
        Constrain(IpDst not in 141.85.0.0/16),
        Constrain(TcpDst==80||TcpDst==443),
        Allocate(FirewallState),
        Assign(FirewallState,1),
        Forward(OutputPort(0))
    ))

```

To understand symbolic execution, we inject a packet on input port 0 of the switch and trace its evolution in Figure 10 (only switch processing on port 0 is captured in the figure). The packet has all header fields initialized to symbolic values:

1. The values of `IpSrc` and `IpDst` are constrained. Then, the `If` instruction results in two packets (or symbolic execution paths).
2. The “else” branch packet, packet 2, captures non-HTTP traffic and is forwarded to the switch output port 1; at this point symbolic execution of packet 2 stops, as output port 1 is not connected in our model.
3. On the `If` branch, the TCP destination port is constrained to be HTTP(S), and the packet is forwarded on output port 2 and onto the firewall on port 0.
4. The packet is processed at the firewall. There is no `FirewallState` in the packet so the else branch runs, and constraints are added for `TcpDst`. The state is allocated and assigned and the packet sent to output 0.
5. The packet enters the switch via input port 2, the else branch is run and the packet finally reaches output 1.

The output of symbolic execution is 1) a set of packets that have reached unconnected output ports, together with 2) a set of packets that have failed en-route, because some of the constraints applied to their header fields did not hold. The first category is more interesting for network verification. For each path, SymNet reports in a JSON-formatted output file all the instructions executed, all the ports visited, the values and/or constraints for all header fields.

In particular, if we examine the output from SymNet for the example above, we can conclude that outgoing reachability is permitted for all packets with correct IP

addresses. We also find that the packet headers are not modified by our configuration: all header fields are bound to the same symbolic variable at the beginning and end of the execution. To gain more insights in this configuration, we also inject a purely symbolic packet at input port 1, and also add a symbolic value for the FirewallState variable. The results show that all HTTP response traffic is dropped unless firewall state is set to 1, and all other traffic is allowed unmodified.

4.3 Analyzing OpenStack with SymNet

We provide a high level description of how SymNet can be used to improve Neutron to achieve all its goals, namely tenant policy compliance, implementation correctness, traffic isolation and performance. To this end, we will rely on SymNet for network verification on two layers: the abstract tenant network and the deployed network.

4.3.1 Checking the abstract tenant network.

To check policy compliance, the tenant can use SymNet to run reachability from all instances to all other instances and the Internet before the configuration is deployed. The SymNet output allows the tenant to quickly check whether the reachability matches their expected behavior.

We have implemented support for such testing in Neutron and our implementation is shown in Figure 9. Whenever a tenant uses Neutron to create or modify a virtual network topology (step 1) and prior to the effective deployment, we insert an additional verification step that uses SymNet to analyze the tenant configuration. The deployment process is stalled until the analysis process ends.

To receive information about the creation of new topologies or the modification of existing ones, SymNet registers to the Neutron callback system. Every time such a callback is executed (step 2), SymNet queries the OpenStack HTTP Networking API [1] for the Neutron network configuration currently in place. Next, we need a SEFL model of the tenant network. We have modeled the functionality offered by Neutron including routers, firewalls, NATs and created a library of SEFL models. For every network function used by the tenant, we obtain the corresponding SEFL element by configuring the generic SEFL library component with the parameters provided by the tenant (step 3). The element's input and output ports are then interconnected according to the virtual links provided by the tenant.

Finally, symbolic execution is performed by injecting symbolic packets at all the instances' network attachment points, as well at the Internet gateway. The result is detailed reachability for all VMs in the abstract tenant network. Currently, the result of the analysis is provided to the tenant in JSON format, which manually checks whether it obeys its policy. In the future, we plan to automatically check simple popular policies, such as:

- All-to-all VM communication for ICMP, UDP and TCP traffic.
- Outgoing ICMP and TCP reachability from all instances.
- Incoming SSH reachability (TCP on port 22) for all instances.

4.3.2 Checking the deployed network dataplane.

The second step of our verification happens *after the tenant's configuration is deployed*, and its goal is to ensure *isolation of tenant traffic* and *correctness of implementation*. To this end, we use two complementary approaches: guided testing and symbolic execution.

Guided testing[32, 39] is very simple: for every path resulting from the tenant network analysis, generate a matching packet and inject it in the actual network, observing the packets reachable at other instances or in the Internet. The big advantage for guided testing is that it can be run by the tenant, without cloud provider support and is independent of the deployed network. We have implemented a simple version of guided testing by using SymNet to generate test packets for the provided paths, generating packets using the Click modular router [15] and using `tcpdump` for reception.

Guided testing is not exhaustive: even if it reports success for the tested packets, there are no guarantees the deployment is indeed correct, or that tenant traffic is correctly isolated.

To get hard guarantees we resort to symbolic execution of the deployed network. Compared to the abstract tenant network, the setup needed to symbolically analyze the real network is much more complex: we need accurate SEFL models and snapshots of the dataplane state for all the boxes (hardware and software) deployed in the cloud-operator network that interact with tenant traffic. This includes network ports for *all* instances of all tenants, hypervisor functionality (software switching, tunnelling and local filtering), OpenStack network nodes, hardware switches and routers.

Creating a solution that is applicable to all networks is an extremely challenging task, given the heterogeneity of deployed infrastructure and the pervasive use of middleboxes [28]. Generating accurate SEFL models of middlebox functionality is not trivial and requires a lot of expert effort. There is currently no generally applicable recipe to all networks. Modeling real networks, however, is feasible. In prior work we have developed a model of our department's network [32]. This model relies on the following building blocks: a) a switch model that can be automatically created when given a snapshot of the dynamically-learned MAC table; b) a router model created from a snapshot of the forwarding table, obtained via standard commands on Cisco routers, and c) a CISCO firewall model (Application Security Appliance) that is created automatically given the configuration file. We are currently using this model as a basis to implement OpenStack deployment checking in our network.

Given an accurate model of a deployed network, we can use symbolic execution to ensure key properties for Neutron. We initiate reachability checks from the new tenant's instances and from the Internet and use the SymNet output to check isolation and correctness, as described next.

Checking isolation. If any VM from any other tenant is reachable from or reaches the instances of the new tenant, we report a violation of the isolation properties. Symbolic execution enables this analysis, but it is complex because its runtime depends grows linearly with the number of tenants/VMs. Optimizations are needed to ensure it scales to large clouds and may include only checking outgoing connectivity from the new tenant, or defining tenant equivalence classes and running reachability between equivalence classes.

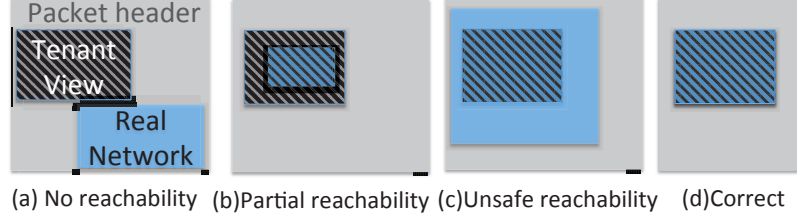


Figure 11: Checking for equivalence between the tenant abstract network configuration and the deployment

Checking correctness. We can compare the sets of paths resulting from the abstract tenant view and the deployment view to decide whether the deployment correctly implements the tenant network. In essence, we want to decide whether the two configurations are equivalent. Equivalence is undecidable for general programs, however we restrict our definition to reachability: we want to ensure the same packets are reachable in the two configurations.

Before we describe the algorithm, we give a few definitions. We assume all packets contain the same set of headers H_1, H_2 , etc., for which we care to verify equivalence; all other headers are ignored. Let $C_{H_i}(P_j)$ denote the set of constraints of header field H_i in packet P_j at some reference port in the network. Let $C(P_j) = C_{H_1}(P_j) \cap C_{H_2}(P_j) \cap \dots$ denote the conjunction of all constraints applied to all headers of packet P_j at the same port.

Output Equivalence Algorithm. *Input:* The set of symbolic packets $P_i, i = 1 \dots N$ and $Q_j, j = 1 \dots M$ obtained by checking reachability between ports a and b in the tenant network and real network, respectively.

Algorithm: To test for equivalence, first we compute the disjunction of constraints of all packets in the two networks at port b : $X = C(P_1) \cup C(P_2) \dots \cup C(P_N)$ and $Y = C(Q_1) \cup C(Q_2) \dots \cup C(Q_M)$. The two sets of paths are equivalent if and only if the expression $(X \cap \neg Y) \cup (\neg X \cap Y)$ is not satisfiable.

The intuition for this algorithm is given in Figure 11 where we show a packet with two header fields. The hashed areas corresponds to the reunion of constraints at node b resulting from symbolic execution of the tenant network and the actual network, respectively. The output equivalence algorithm aims to determine if the two areas overlap perfectly.

We have four cases: first, the tenant reachability does not overlap at all with the deployment reachability, resulting in a completely broken instantiation of the tenant network: the tenant has no reachability for its traffic, while unwanted traffic is allowed. In the next case we have partial reachability, but at least the configuration is safe: unwanted traffic is stopped. The third configuration allows full reachability, however also allows other packets through—this could indicate that the firewall rules have not been instantiated properly. Finally, the last case is when there is perfect overlap, and the two configurations are equivalent from an output port point of view.

Note that output equivalence is fairly weak in networks that modify header fields. There, additional constraints could be applied on the initial values of the header fields that influence reachability, yet they are not captured by output equivalence. In our

future work we plan to explore stricter notions of equivalence such as input-output equivalence.

4.4 Preliminary Evaluation

We ran reachability analysis for the tenant configuration in Figure 7 by injecting a symbolic packet at all the tenant instances and the gateway. It takes 5 to 7 seconds to run the reachability analysis, for which 3 to 5 seconds are spent in Neutron API calls and 2s to generate the SEFL code and run the reachability analysis.

Our first analysis showed no connectivity at all because the tenant configuration didn't have static IPs assigned and no DHCP server was enabled either. We changed the configuration by assigning static IPs to all the instances; the analysis found that all instances can communicate directly. Further, A and B have outgoing Internet connectivity, and that connections initiated from the Internet are not allowed. Finally, C has no Internet connectivity.

Next, we deployed the configuration and ran our guided testing implementation. Surprisingly, guided testing showed that instance C had Internet connectivity but it couldn't access instances A or B; this is the exact opposite of the tenant configuration, where C has no Internet connectivity but it can reach A and B. It turned out that this problem was transient: when we reran the guided testing scheme, the results were as expected. The culprit was identified to be the propagation latency between high level commands and driver implementation in Neutron, which we measured to be on the order of minutes in our deployment.

4.5 Conclusions

OpenStack Neutron is a complex piece of software, providing different views to different stake-holders and incorporating code from multiple parties. It has been anecdotally called the weakest link in OpenStack [26].

We argue this happens because debugging currently only relies on standard best practices for code development, without taking into account the particularities of Neutron. As a complement to existing approaches, we propose using static network analysis to improve OpenStack Neutron. We have shown how network symbolic execution can be used on two levels: to check the abstract tenant network and its deployment in the actual network. We have an initial implementation of these ideas that we have integrated with the Neutron API. Our preliminary evaluation has found interesting nuggets: a propagation delay bug in OpenStack (that was known) and can help tenants more easily deploy their networks.

This is a work in progress, and many refinements are needed to our prototype until it can be applied to a wide range of configurations automatically. On the algorithmic side, we intend to explore stronger version of equivalence. Finally, we intend to fully model our department's network (including Openvswitch, Neutron nodes, etc) and perform full symbolic analysis of the deployed network.

5 Symbolic execution - model equivalence & applications

There is a fundamental tension between the runtime speed and the symbolic execution speed of a program [35]. When analyzing a simple switch model, symbolic execution can take orders of magnitude less time when the code has been optimized for symbolic execution. However, executing that same code in practice will result in very poor performance [32]. Many researchers have observed this and produced optimized models that enable symbolic execution; unfortunately, there is a gap between the model and the actual code.

We take a principled approach to understand what transformations are correct and safe when optimizing models for symbolic execution. Intuitively, we want the optimized model to behave in the same way as the original code.

One approach for checking model equivalence is to look at the domain sets for each program variable. Suppose symbolic execution for model M yields n paths and on each path i , the variable v is bound to symbolic expression e_i . The *domain set* for variable v is thus $d_{M_1} = e_1 \vee e_2 \vee \dots \vee e_n$.

Two models M_1, M_2 are *output-equivalent* iff the domain sets for all variables are equivalent ($d_{M_1} \wedge \neg d_{M_2} = \text{false}$).

Output equivalence is limited in capturing input-output dependence. For instance, programs `if (x > 0) x = 1; else x = 0` and `if (x > 0) x = 0; else x = 1;` are output equivalent, but do not behave in the same way.

A more conservative alternative is to define equivalence in terms of execution paths: M_1, M_2 are equivalent if each path from M_1 is equivalent to some path from M_2 and vice-versa.

However, this approach leaves little room for symbolic execution optimizations which may target path reduction.

A tradeoff between the two is to consider *state-dependent equivalence*. Say P is a *state-predicate* (e.g. $P \equiv x \neq 0$). Two models are P -equivalent if they produce precisely the same output on any input that satisfies P . State predicates offer flexibility in deciding how strong requirements we place on model equivalence.

5.1 Implementation

In order to optimize models for symbolic execution, we trade off between: (i) the number of execution paths and (ii) the number of constraints per path, for each variable.

Optimization consists in a sequence of equivalence-preserving transformations similar to compiler code optimizations.

First, we perform general optimizations e.g. removing conditionals which result in only one successful path, removing dead code, merging consecutive variable constraints in a single one (thus having fewer invocations on the constraint solver). For some transformations, we may only preserve equivalence with respect to specific state predicates.

Second, we apply transformations aimed at optimizing for criteria (i) or (ii). For instance, in `if (x > 0) p1 else p2`, if we can determine that $x > 0$ holds (univer-

sally, or w.r.t. our given state predicate), we can transform the program to **assert** ($x > 0$); p_1 , thus reducing program branching.

References

- [1] OpenStack Networking API 2.0 Specification. <http://developer.openstack.org/api-ref-networking-v2.html>.
- [2] OpenStack users share how their deployments stack up. <http://superuser.openstack.org/articles/openstack-users-share-how-their-deployments-stack-up>.
- [3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.
- [4] P. Burghouwt, M. Spruit, and H. Sips. Towards detection of botnet communication through social media by monitoring user activity. In *Information Systems Security*, pages 131–143. Springer, 2011.
- [5] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI’08*.
- [6] M. Celenk, T. Conley, J. Willis, and J. Graham. Predictive network anomaly detection and visualization. *Information Forensics and Security, IEEE Transactions on*, 5(2):288–299, 2010.
- [7] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(3):15, 2009.
- [8] S. Chang and T. E. Daniels. P2p botnet detection using behavior clustering & statistical tests. In *Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence*, pages 23–30. ACM, 2009.
- [9] M. Dobrescu and K. Argyraki. Software dataplane verification. In *Proc. NSDI’14*, NSDI’14.
- [10] M. Feder, N. Merhav, and M. Gutman. Universal prediction of individual sequences. *Information Theory, IEEE Transactions on*, 38(4):1258–1270, 1992.
- [11] J. Francois, S. Wang, W. Bronzi, R. State, and T. Engel. Botcloud: detecting botnets using mapreduce. In *Information Forensics and Security (WIFS), 2011 IEEE International Workshop on*, pages 1–6. IEEE, 2011.
- [12] G. Gu, J. Zhang, and W. Lee. Botsniffer: Detecting botnet command and control channels in network traffic. 2008.
- [13] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proc. NSDI’12*.

- [14] S. T. Klein and Y. Wiseman. Parallel lempel ziv coding. *Discrete Applied Mathematics*, 146(2):180–191, 2005.
- [15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.
- [16] A. Lapidoth. *A foundation in digital communication*. Cambridge University Press, 2009.
- [17] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *Proc. NSDI’15*.
- [18] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *Sigcomm*, 2011.
- [19] Michael Till Beck and Martin Werner and Sebastian Feld and Ludwig Maximilian and homas Schimper. Mobile Edge Computing: A Taxonomy. In *AFIN 2014*.
- [20] J. Neyman and E. S. Pearson. *On the problem of the most efficient tests of statistical hypotheses*. Springer Series “Breakthroughs in Statistics”, 1992.
- [21] M. Nisenson, I. Yariv, R. El-Yaniv, and R. Meir. Towards behaviorimetric security systems: Learning to identify a typist. In *Knowledge Discovery in Databases: PKDD 2003*, pages 363–374. Springer, 2003.
- [22] S.-K. Noh, J.-H. Oh, J.-S. Lee, B.-N. Noh, and H.-C. Jeong. Detecting p2p bot-nets using a multi-phased flow model. In *Digital Society, 2009. ICDS’09. Third International Conference on*, pages 247–253. IEEE, 2009.
- [23] Openstack. Neutron Networking. <https://wiki.openstack.org/wiki/Neutron>.
- [24] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker. Verifying Isolation Properties in the Presence of Middleboxes. Tech Report arXiv:1409.7687v1.
- [25] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. G. s, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *NSDI*, pages 117–130, Oakland, CA, May 2015. USENIX Association.
- [26] T. Register. HP: OpenStack’s networking nightmare Neutron ‘was everyone’s fault’. http://www.theregister.co.uk/2014/05/13/openstack_neutron_explainer/.
- [27] G. Seroussi and A. Lempel. Lempel-ziv compression scheme with enhanced adaptation, Sept. 7 1993. US Patent 5,243,341.
- [28] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. y. Sekar. Making middleboxes someone else’s problem: Network processing as a cloud service. In *SIGCOMM*, 2012.

- [29] S. Shin, G. Gu, N. Reddy, and C. P. Lee. A large-scale empirical study of conficker. *Information Forensics and Security, IEEE Transactions on*, 7(2):676–690, 2012.
- [30] S. Siboni and A. Cohen. Botnet identification via universal anomaly detection. In *2014 IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 101–106. IEEE, 2014.
- [31] S. S. Silva, R. M. Silva, R. C. Pinto, and R. M. Salles. Botnets: A survey. *Computer Networks*, 57(2):378–403, 2013.
- [32] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Symnet: scalable symbolic execution for modern networks. In *Sigcomm*, 2016.
- [33] W. T. Strayer, D. Lapsely, R. Walsh, and C. Livadas. Botnet detection based on network behavior. In *Botnet Detection*, pages 1–24. Springer, 2008.
- [34] R. Villamarín-Salomón and J. C. Brustoloni. Identifying botnets using anomaly detection techniques applied to dns traffic. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 476–481. IEEE, 2008.
- [35] J. Wagner, V. Kuznetsov, and G. Candea. Overify: Optimizing programs for fast verification. In *Proc. HotOS’13*.
- [36] Y. Xiang, K. Li, and W. Zhou. Low-rate ddos attacks detection and traceback by using new information metrics. *Information Forensics and Security, IEEE Transactions on*, 6(2):426–437, 2011.
- [37] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of ip networks. In *Proceedings of Infocom*, 2005.
- [38] C. Yang, Y. Song, and G. Gu. Active user-side evil twin access point detection using statistical techniques. *Information Forensics and Security, IEEE Transactions on*, 7(5):1638–1651, 2012.
- [39] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proc. CoNEXT’12*.
- [40] X. Zhai, K. Appiah, S. Ehsan, G. Howells, H. Hu, D. Gu, and K. McDonald-Maier. A method for detecting abnormal program behavior on embedded devices. *Information Forensics and Security, IEEE Transactions on*, 10(8):1692–1704, Aug 2015.
- [41] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, 1978.



ANNEX: Internal deliverable I6.3b

1. Introduction

In the first year, Superfluidity has shown it is possible to perform exhaustive symbolic execution for network verification via the SEFL modelling language and the Symnet symbolic execution tool. Since then, work has continued to apply SEFL/Symnet verification to real-life problems and verify actual networks.

The present document is a snapshot of the work undertaken by Superfluidity in the area of security, correctness and verification, that has focused on extending our prior work in a few key directions:

- Designing NetCTL, a policy language that operators can use to express their requirements, and an accompanying verification tool that takes the policy as input and drives the symbolic execution (with Symnet) to check whether the policy holds (Section 2).
- Continuing the work of integrating Symnet in Openstack Neutron (Section 3).
- A novel data-structure that allows cost-optimal modeling of match-action tables and very small incremental costs for rule updates (Section 4).
- A reactive monitoring approach that complements our verification work (Section 5).

2. NetCTL: Policy language to drive symbolic execution

NetCTL is an adaptation of CTL - Computation Tree Logic, a language designed and successfully deployed for program verification.

In CTL, temporal operators such as **F** (i.e. sometime in the future) and **G** (i.e. always in the future) are combined with the path quantifiers **exists** (on some path) and **forall** (on all paths). NetCTL borrows these operators and adds SEFL code to describe state-based properties. For instance, the policy:

forall F destTCP = 80

evaluated at some node A of the network, expresses that on all possible packet paths from A, **destTCP** will eventually become 80. The policy combines the quantifier **forall** with the temporal operator **F**. The construction **destTCP = 80** is a SEFL code which describes a network property evaluated at a hop in the topology.

Unlike other, more expressive temporal languages such as CTL*, CTL requires that each path quantifier be directly preceded by a temporal operator. This syntactic restriction ensures CTL's desirable computational properties: model-checking is linear with respect to the size of the model and that of the formula. NetCTL benefits from the same computational properties - it allows verifying a network topology by doing a *single-pass* (i.e. a single symbolic execution).

More formally, the syntax of NetCTL is given below:

$\varphi ::= \text{SEFL} \mid \sim\varphi \mid \varphi \wedge \varphi \mid \text{XY } \varphi \mid \text{X } \varphi \text{ until } \varphi$

where $X \in \{\text{exists}, \text{forall}\}$, $Y \in \{F, G\}$. The syntax includes the standard boolean operators, the CTL operators already discussed, and the `until` operator : the formula

`p until q` expresses that `p` is true until `q` is true.

Unlike other policy languages, NetCTL is compositional: it allows expressing more complicated policies, starting from simpler ones. For instance, the following composed policy:

```
forall G (ip != 192.168.0.0/16 → exists F port == Internet)
```

evaluated in a network node `A`, expresses that on all paths starting from `A`, at each hop, if the IP destination of a packet becomes public, then there exists a path which reaches Internet. We have found that many interesting network policies can be naturally expressed in Netcheck. We illustrate a few such examples:

Traffic isolation

An example of a traffic isolation policy is: *"all traffic destined for A must pass through an IDS"*. The NetCTL policy which expresses this behaviour is:

```
forall (port !=A until port == IDS)
```

The policy states that on all paths, the current port is either different from `A` (hence `A` has not yet been reached) or is equal to `IDS` (and henceforth no other restrictions apply). In other words, the policy expresses that a packet cannot reach `A` before it reaches the `IDS`.

TCP end-to-end connectivity

Suppose we have nodes `A` and `B` of the network. TCP connectivity "at equilibrium" (for simplicity, we ignore connection establishment or timeouts) between `A` and `B` means that any TCP packet from `A` destined to `B` will indeed reach `B`; also, a reply from `B` which will eventually reach back to `A`.

To verify this behaviour, we must add the following code snippet at the input port of `A`:

```
allocate(x); allocate(y);  
x = TCPSrc; y = TCPDst
```

which stores the source and destination TCP fields in variables `x` and `y`. We also add the following code at the output port of `B`:

```
allocate(tmp);  
tmp = TCPSrc; TCPSrc = TCPDst; TCPDst = tmp;  
deallocate(tmp); Forward (B);
```

which swaps the source and destination TCP fields, thus emulating a reply. Finally, the NetCTL policy which captures end-to-end connectivity is:

```
exists F (port == B ∧ forall G (port == A ∧ TCPSrc == y ∧ TCPEst == x ))
```

To verify it, we introduce a fully-symbolic packet at node A of the network. The policy expresses that:

- there exists a path from A on which the port becomes B (hence B is reachable)
- on all paths starting from A, if A is again reachable, then the source and destination TCP fields are flipped.

Tunnel invariance

Suppose we would like to check that, between nodes A and B of the network, a certain field `header_val` is unchanged. This behaviour would ensure that a tunnel between A and B works correctly. We first add the following code at A which introduces a fully symbolic value:

```
allocate(crt_val);  
crt_val = *
```

Our policy for tunnel invariance is:

```
forall G ( port == A; crt_val = header_val } →  
  
forall G (port == B → header_val == crt_val ))
```

We note that the code snippet `port == A; crt_val = header_val` performs a verification (the port must be A) as well as a *state-change*: `crt_val` becomes equal to `header_val`. Thus, the policy expresses that on all paths, at each hop, if the port becomes A:

- store the contents of `header_val` in `crt_val`
- on all subsequent paths, at each hop, if the port becomes B, then `header_val` must have the same constraints as `crt_val`, i.e. its original value.

Verifier implementation

Our NetCTL verifier extends Symnet and has been written in Scala. It performs policy verification in time $O(n*m)$ where n is the size of the model (the total number of instructions) and m is the size of the policy.

The implementation differs from the standard CTL model checking algorithm and exploits the fact that, conceptually, SEFL models have a tree-like structure.

When trying to satisfy an existential or universal path quantifier (i.e. a formula such as $\mathbf{x} \ \varphi$ where $\mathbf{x} \in \{\text{exists}, \text{forall}\}$) at a branching instruction, the verifier will selectively branch program execution. If the policy φ is false on a program path, then forall φ is false the branching point --- symbolic execution need not continue. Conversely, if φ is true on a path, then exists φ is true and the verifier will not explore the other paths.

When trying to satisfy a temporal operator (e.g. \mathbf{F} , \mathbf{G}), the verifier will behave in an analogous fashion. For instance, when verifying a policy of the form $\mathbf{F} \ \varphi$ (resp. $\mathbf{G} \ \varphi$), symbolic execution will stop with success (resp. failure) whenever a state where φ holds (resp. does not hold) is found.

The entire verification process corresponds to a (partial) depth-first traversal of the symbolic execution tree of a model, starting from an initial node.

Discussion

NetCTL is *at least as expressive as* existing network policy languages, to the extent of the authors' knowledge. We can easily embed any policy from e.g. Merlin, Procera, NetPlummer (HSA) into NetCTL. Moreover, by relying on SEFL, our language can verify network properties which: (i) are difficult to model by existing approaches, (ii) are not (easily) expressible in existing policy languages.

NetCTL is fully-automated, and relies on the SEFL & Symnet ecosystem to generate models from existing networks. NetCTL **guarantees** network correctness over such models.

The NetCTL-based verification procedure can offer strong correctness guarantees to the extent to which the SEFL network models are accurate. While SEFL is more expressive than most existing modelling languages (e.g. HSA), it cannot capture certain box transformations such as: (i) packet reordering, (ii) packet fragmentation/assembly, (iii) traffic shaping, (iv) selective/random dropping of packets.

Also, SEFL is not designed to capture certain implementation bugs from middleboxes, which may affect how packets are handled by the network.

Thus, NetCTL delivers a *best-effort* verification. If a policy is found to be false, a counter-example (i.e. the packet trace which violates the policy) is provided. The existence of a gap between the network model and the actual network infrastructure means that a true policy does not guarantee correct behaviour in the implementation, in all situations.

One alternative to addressing the model-implementation gap, is to *generate executable network processing* from SEFL models. Thus - model verification renders network *correctness by design*. We are currently exploring this direction, for networks with simple (layer 2/3) traffic processing.

3. Openstack Neutron verification with symbolic execution

In this section we discuss how we can verify the network configurations of OpenStack, the leading cloud management software. OpenStack is an open-source cloud platform software deployed as an Infrastructure-as-a-Service architecture. OpenStack abstracts away the complexity of a computing environment and provides the user with a set of services to create, manage and use compute stacks in rich network topologies transparently. The interaction of the user with OpenStack is ensured using a set of public REST APIs and a graphical user interface. Internal communication within the cloud data-center is ensured via a set of private APIs implemented as RPCs (remote procedure calls).

OpenStack exposes a set of services to the users. The most important one, the compute service (nova) provides the user with the capability of launching machines as needed to be run in the provider data center. OpenStack also exposes image, block storage, authentication and authorization, service discovery, quota management and remote access services.

A user may be assigned to one or more tenants. A tenant (or project) provides the abstractions to handle separation between multiple customers in the same cloud. For instance, a company may choose to acquire an OpenStack tenant from a cloud provider and assign a number of users to it as they see fit. Thus, the internal services provided for the company are hidden from those of a different company.

Using the compute service, the user may launch a number of machines. It is the task of OpenStack's scheduler to handle assignment of each virtual machine launched by a user to a given hypervisor host within the topology. The hypervisor hosts are known as compute nodes. It is the purpose of the networking service to provide connectivity between multiple machines within a given tenant and to the Internet.

From a historical perspective, previous releases of OpenStack used nova networking as a service for providing the basic network functionalities required by OpenStack. However, the customers' needs to create rich network topologies within their tenants has led to the development of Neutron (previously Quantum) service. It provides the main abstractions that allow networking availability to the user and offers the possibility to create logically separated networked environments within tenants.

Initially, the nova networking service was using a flat scheme, consisting of one or more shared networks (provider networks) using a common IP addressing scheme and some segmentation features meant to keep traffic between tenants separate. However, there was no possibility to create private networks within a given tenant, nor were other services available, such as VPN, firewall etc. In the following, the main features of Neutron are described.

The main services that Neutron offers to the user are: networks, subnets, routers, VPNs, load balancers, firewalls, ports, DHCP, security groups. The concepts underpinned by the

aforementioned services are well-known in classic networking infrastructures and provide well- defined functions to the user.

For the scope of the current project, an OpenStack deployment can be seen from two distinct perspectives: the tenant perspective and the deployment perspective. The tenant perspective refers to all abstractions that are provided via OpenStack public APIs to the end-user (e.g. routers, networks etc.). The deployment perspective refers to all components that make up the underlying implementation of OpenStack (e.g. agents, plugins, drivers etc.) and the configurations deployed on the machines that represent the OpenStack environment (e.g. OpenFlow tables, iptables tables etc.)

As described previously, it is important to underline that modeling the behavior of an OpenStack system from one perspective or the other may yield different results. The bottom line of this research endeavor is proving equivalence of the two symbolic executions.

Neutron Architecture

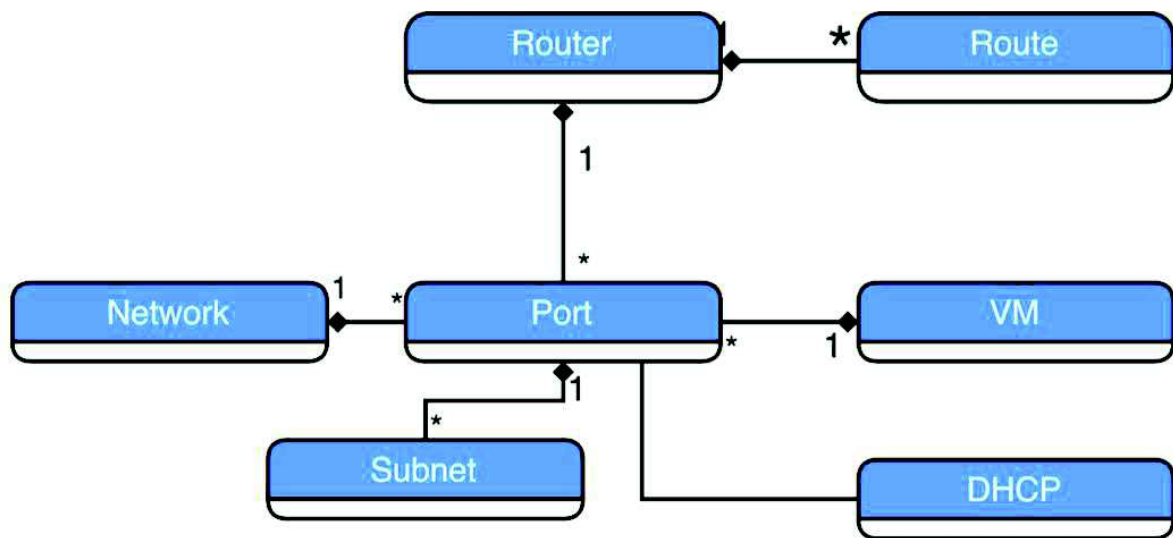
The OpenStack networking services of interest are: routing, floating IPs, self-service (per-tenant) networks, subnetworks. The metadata and DHCP services are not of interest for this matter and will not be analyzed further on.

As specified in the OpenStack Networking Guide, Neutron has a number of basic components:

- A Neutron Server - handles API service requests (e.g. at tenant level, allows for creating, retrieving, deleting and updating information on tenant level resources)
- Neutron Agents - provide Layer-2, Layer-3, DHCP and Metadata functions
- Neutron plug-ins - manage agents

A commonly used implementation for Neutron resides in the use of ML2 (Modular Layer 2) plug-in to handle agent management. Simply put, the ML2 provides a framework for deploying a number of drivers which handle connectivity and correct implementation for tenant-defined networks, as well as offering the possibility for tenant-based separation and segmentation.

Within the framework of ML2, the implementation analyzed is composed of multiple OVS (OpenvSwitch) bridges, a set of Linux ip namespaces with multiple interfaces and a set of iptables tables implemented within the aforementioned namespaces.



The object model of interest for the tenant perspective of OpenStack networking can be seen in the Figure above. As can be easily observed the central component in the diagram is the port. Each port connects a distinct piece of equipment, such as a router, a DHCP server or a VM and translates into a tap interface inside a Linux PC.

From the deployment perspective, an OpenStack cloud is composed of a number of machines that perform several functions with respect to their high level specifications.

- A machine in the system is called node.
- A node can be either a compute node or a network node.
- Each VM will run on a compute node.
- Each Neutron router will run on a network node.
- Each compute node will handle security groups per machine.
- Each node will have at least two OVS bridges: the br-tun and the br-int.
- Each node may have several bridges pertaining to a provider network.
- Each Neutron port maps to a tap network interface within the system.
- Each tap interface is connected on the br-int bridge.
- Tunneling is performed by br-tun.
- Each OVS bridge may behave like an OpenFlow bridge or like a normal L2 bridge.
- Each node distinguishes Neutron network traffic based on VLANs which are scoped to the node they are on.
- Tunneling assigns a system-wide unique tunnel id to a Neutron network.
- br-tun translates from tunnel id to scoped VLANs
- br-int and br-tun are directly connected.

The implementation considered for symbolic execution is using the ML-2 plug-in with L3 and OpenvSwitch L2 agents using VXLAN as means to provide spanning topology. Each compute node runs an OpenvSwitch layer 2 agent which performs all the switching and

segmentation logic. The L3 agent is implemented as a standalone network node which performs routing for each router resource declared in the tenant perspective of the topology.

In the following, structural requirements imposed upon some node in the system are presented. In general, assuming some packet flow, one can state that the packet-processing components that make up a node within an OpenStack deployment are:

- A node has one or more namespaces.
- A namespace has one or more interfaces.
- A node has one or more bridges.
- A bridge may connect one or more interfaces.
- A namespace has a routing table.
- A routing table is composed of one or more routes.
- A namespace has one or more iptables tables.
- An iptables table is composed of multiple iptable chains.
- An iptables chain has one or more rules.
- A rule has one or more matches.
- A rule has one action.
- A bridge can be an OVS Bridge or a Linux Bridge.
- An OVS Bridge contains multiple OVS interfaces.
- An OVS interface may be access port or trunk port.
- An access port has an associated VLAN tag.
- A trunk port may transport some or all VLAN tags.
- An OVS Bridge is configured via OpenFlow Configuration.
- An OpenFlow configuration has a set of OpenFlow ports.
- An OpenFlow port maps to an OVS interface.
- An OpenFlow configuration contains a set of OpenFlow tables. • An OpenFlow table is composed of multiple flows.
- A flow may contain multiple matches.
- A flow may contain multiple actions.

Notice that the above list aims at depicting a Linux node from a purely structural perspective. The behavior of blocks, the interaction between components, as well as the order in which packet-processing elements are executed are neglected for the moment and will be described more thoroughly later. The above structure was derived from iptables manuals and user guides, OVS documentation and the OpenFlow specification.

Implementation

In order to achieve integration with the distinct pieces of technology, the following steps were taken:

1. Acquiring configuration data from all sources
2. Interpreting acquired information into internal structures
3. Modeling the packet-processing entities in SEFL.

4. Integrating the functional blocks modeled in SEFL to generate a full packet-processing pipeline.

The body of work pertaining to all the aforementioned items was two-fold. Thus, from the tenant perspective, the steps depicted above involved acquiring the data exposed through Neutron APIs as resources, understanding their semantics and then performing the modeling and integration tasks as prescribed. To that end, Symnet acquires tenant-level data either through the Neutron REST APIs, given knowledge of some valid credentials and a publicly exposed communication endpoint, or by reading and parsing files exported through the command-line utilities provided by Neutron.

From the provider perspective, the configuration data acquisition is performed by dumping configuration files of all components on all nodes in the Neutron deployment. Thus, information regarding Linux IP namespaces, iptables tables, OVS database dumps, OpenFlow dumps for all switches defined in the OVS database, routing tables, IP interfaces and OpenFlow port mappings to actual interfaces are required in order to perform a full symbolic analysis of the system.

Furthermore, internal structures were defined as convenience objects for querying necessary information in the modeling process. The information acquired and enumerated above is translated into packet processing blocks (in SEFL).

Modeling approach

In the following, the approach taken towards modeling the packet-processing blocks of the *Neutron-Symnet* implementation is described. Thus, the most important packet processing blocks identified are: *Netfilter* hooks and *OpenFlow tables*. The first are mostly employed for verifying correctness of the *Neutron L3 agent*, whilst the latter is employed for verifying correctness of the L2 agent and security groups implementation.

Netfilter hooks, and especially *iptables* chains are used for routing and filtering traffic within the L3 agent - i.e. the component which implements Neutron Routers. Each iptables table is composed of a series of *rules*. An iptables *rule* is composed of a series of *matches* and a *target*. As an example of *SEFL modeling* of an *iptables* match, the following listing is presented to describe source IP matching:

```
def matchSourceIp(startIp : IPAddress, endIp : IPAddress) {  
    meta.matched = false;  
    if (packet.EtherType == EtherType.IPv4) {  
        if (packet.SourceIp <= endIp && packet.SourceIp >= startIp) {  
            meta.matched = true;  
        }  
    }  
}
```

Listing 1: Source IP match in iptables

In order to depict the modeling undertaken for an iptables target, a more challenging example is presented. Thus, the SNAT action is presented. Note that this particular action can only be fired for the first packet of a connection in the NAT POSTROUTING chain. As such, SNAT can be described as follows:

```
def fireSNATTarget(ip : IPAddress, portStart : Int, portEnd : Int) {
  meta.SNAT.DstOriginalAddress = packet.IPDst;
  meta.SNAT.SrcOriginalAddress = packet.IUSrc;
  if (packet.IPProtocol == IPProtocol.TCP) {
    meta.SNAT.SrcOriginalPort = packet.TCPSrc;
    meta.SNAT.DstOriginalPort = packet.TCPDst;
  } else if (packet.IPProtocol == IPProtocol.UDP) {
    meta.SNAT.SrcOriginalPort = packet.UDPSrc;
    meta.SNAT.DstOriginalPort = packet.UDPDst;
  } else if (packet.IPProtocol == IPProtocol.ICMP) {
    meta.SNAT.SrcOriginalPort = packet.ICMPId;
    meta.SNAT.DstOriginalPort = packet.ICMPId;
  }
  meta.SNAT.IsSNAT = true;
  packet.IPSrc = ip;
  if (portStart.isSpecified || portEnd.isSpecified) {
    if (packet.IPProtocol == IPProtocol.TCP) {
      packet.TCPSrc = Symbol();
      constrain packet.TCPSrc <= portEnd && packet.TCPSrc >=
portStart;
    } else if (packet.IPProtocol == IPProtocol.UDP) {
      packet.UDPSrc = Symbol();
      constrain packet.UDPSrc <= portEnd && packet.UDPSrc >=
portStart;
    }
  }
}
```

Listing 2. SNAT Target in SEFL

Connection Tracking

One of the most challenging components to model from a behavioral point of view is the connection tracking engine implemented within the Linux Kernel as part of the Netfilter framework.

Same as iptables, the connection tracking engine is using the concept of hooks to perform a set of actions on an input packet.

Conceptually, the connection tracking engine (or *conntrack*) denotes a connection as a 5-tuple composed of source IP address, destination IP address, IP Protocol and two distinct L4 addresses meant to distinguish between multiple connections. At the entry of a packet within conntrack, it is assigned to a new entry inside a locally managed table. A packet may be found as already being part of an existing connection or may be a new packet, which will in

turn be assigned a new entry inside the table. If a connection is found, then the packet may be in the forward direction or in the backward direction. If a packet is not part of a known connection, conntrack assigns two new entries inside its connection tracking table. The first sets forward expectations - i.e. how are forward packets mapped to this connection, while the second refers to backward packets - mapping return packets to the a connection in the table. The first packet from a connection dictates from the moment of its arrival, the expectations for forward and backward packets. Conntrack also provides to the user-space a set of states for a given connection. The states of a connection are either: NEW, ESTABLISHED, RELATED, INVALID. Apart from those, two virtual states called SNAT and DNAT are available, which represent the fact that the packet has undergone SNAT, DNAT or the reverse operations priorly in the netfilter chain. At kernel-level, the state machine is more complex, as it takes into account protocol specific information, such as TCP flags for packets and their tracing. The high-level logic for a connection is as follows:

- On packet arrived if no connection corresponds \Rightarrow Create new connection with forward expectation congruent to the packet's fields and backward expectations corresponding to the packet's reversed fields; Set connection state to NEW
- On packet arrived if connection corresponds and connection state is NEW and packet matches backward expectation \Rightarrow Set connection to state ESTABLISHED
- On packet arrived if connection corresponds and connection state is ESTABLISHED \Rightarrow Connection state remains the same

For simplicity, the modeling approach in the current paper does not take into account the RELATED state. In conntrack, a connection state of RELATED is used to represent a packet not part of a regular protocol ow, but rather an auxiliary part of it (e.g. ICMP packets for signaling errors in FTP connections).

In what timing is concerned, conntrack has two of points of activation: in the PREROUTING chain and in the POSTROUTING chain. Thus, in the PREROUTING chain, conntrack is called to identify the connection which the current packet belongs to. It is at this point where conntrack writes new connections to the connection table and sets internal variables to point to the corresponding entry in the table. In the POSTROUTING chain, the packet is committed to conntrack, establishing the backward expectations for the connection. Packet filtering is accomplished in hooks between the two steps.

NAT

In relation to the connection tracking module depicted above, NAT features (destination and source NAT) must be integrated with the connection tracking logic. Internally, a structure similar to that of the connection table is kept for all packets which undergo SNAT or DNAT.

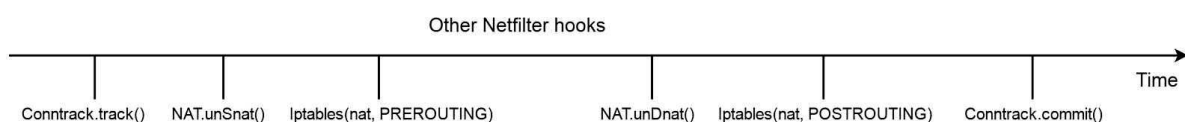
The point of insertion for NAT-related functions are the PREROUTING and POSTROUTING chains. In these chains, the following logic is applied: a packet passes through the iptables nat tables (either PREROUTING or POSTROUTING) if it is the first packet belonging to a connection.

For SNAT (source NAT), upon first packet of a connection arriving at POSTROUTING chain, the iptables nat table is traversed. Should any NAT action be executed, the packet is modified (i.e. its source network and transport layer source addresses are changed) and the original state of the packet is stored in the NAT table. Upon committing the packet to conntrack, the backward expectations for the packet will be those of the current packet (with all fields modified).

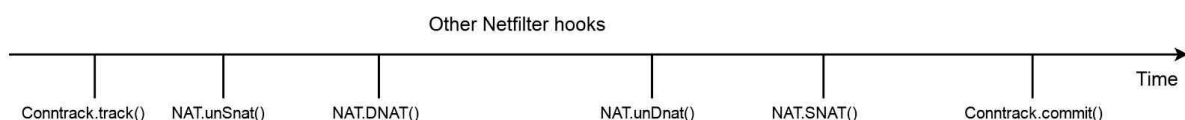
The reverse operation is applied in the PREROUTING chain after the conntrack track operation is executed and before any iptables PREROUTING traversals are performed. At this stage, the current packet is looked up in the NAT table and if a backward expectation is found, then it is transformed accordingly.

For a packet matching a forward expectation, the traversal of the iptables nat POSTROUTING chain is bypassed and the existing NAT mappings from the NAT table are applied.

For DNAT (destination NAT), a similar operation occurs, with the difference that the packet modification step occurs in the iptables PREROUTING chain, while the reverse step occurs in the POSTROUTING chain.



In the above figure, a set of functions which are applied to a newly arrived packet are depicted in the calling order defined within the netfilter framework. The representation does not take into account other iptables chains and the interactions they may exert upon the conntrack and NAT modules.



In the above figure, the same diagram is depicting the actions undertaken for a packet which already went through the connection tracking module and is not the first in the given connection. Notice that the calls to iptables nat table in chains PREROUTING and POSTROUTING are replaced with calls to the NAT modules corresponding to the insertion place.

All netfilter hooks (i.e. all iptables calls) performed between the PREROUTING and the POSTROUTING chains experience the packet with its internal source and destination addresses. If SNAT is regarded as a translation between an internal address to an external addresses, then all translation in a forward packet happens in the POSTROUTING chain. Meanwhile, all translations for a backward packet (i.e. un-SNAT) happen in the PREROUTING chain. The same goes for DNAT, with the difference that in the case of DNAT, the translation (DNAT) happens in the PREROUTING chain, since its semantics is

usually that of changing an external address to an internal one, while the reverse operation (i.e. un-DNAT) happens in the POSTROUTING chain.

OpenvSwitch Blocks

OVS (OpenvSwitch) bridges are important components of a Neutron deployment. The ML2 plug-in used in most installations relies on the OpenvSwitch driver to perform layer 2 bridging, tenant separation and filtering.

From a structural perspective, the OpenvSwitch bridge (OVS) can be seen as an object with a number of components:

- An OVS *bridge* is a *layer 2 software switch*.
- An OVS *bridge* has multiple *ports*.
- A *port* has multiple *interfaces*.
- A *port* can be an *access or trunking port*.
- A *trunking port* may specify a list of *allowed VLANs*.
- A *trunking port* with no *allowed VLANs* performs trunking for all VLANs.
- An *access port* must specify the *VLAN tag* it belongs to.
- A *port* can be directly connected with a *port* in another *bridge*.
- All traffic that enters a *directly connected port* is directly passed to the *port* it connects to.
- A *port* can be directly connected with a *remote port* using some encapsulation technology (e.g. vxlan, gre, geneve, ipsec).
- All traffic that enters a *port directly connected with a remote one* is directly passed to the *remote port*.

All OVS behavior is specified using OpenFlow flows. A *flow* belongs to an *flow tables*. For more details, see OpenFlow specification and *ovs-ofctl* command's *manpage*.

- An OpenFlow *switch* has multiple *flow tables*.
- An OpenFlow *switch* has multiple *ports*.
- Each *flow table* is composed of multiple *flows*.
- Each *flow* has a *priority* in the *flow table* it belongs to.
- Each *flow* has multiple *matches* and multiple *actions*.
- A *match* is a *key-value* pair which specifies a *field* and a *value* to match.
- Whenever a *packet* matches the conditions imposed by the *set of matches*, the *actions* are marked for execution.
- An *action* is a modification on either *packet fields*, *internal switch variables* or *processing flow*.
- *Actions* are cumulative; thus, actions are marked for execution as flows are being matched, but are executed whenever a flow traversal ends or an *apply-actions* action is encountered.
- All packets not matching any *flow* are applied the *default flow* (in the case under study, the packet is dropped).
- When a packet enters a *port*, the *input port* internal variable is set to that port and the packet is sent for processing into *table 0*.

In Neutron, the actions that need to be taken by the OVS layer are of *Apply-Actions* type (see OpenFlow Specification for details). This choice eases the process of modeling the OpenFlow pipeline and makes it, to some extent, similar to that of iptables.

A modelling sample for OpenFlow input port match is depicted below:

```
def matchInPort(inPort : Int) {  
    meta.matched = false;  
    if (meta.in_port == inPort) {  
        meta.matched = true;  
    }  
}
```

Listing 3. OpenFlow Input Port Match

The OpenFlow specification defines special port numbers for given applications (e.g. LOCAL, NORMAL, FLOOD etc.). For the scope of the current project the only special port encountered was the NORMAL port; in essence, it sends the packet out to OVS in order to perform normal layer 2 switching. At the moment of writing this paper, the current approach to modeling the Layer 2 OVS switch behavior is sub-optimal. The CAM table is not used to distinguish between destination ports and thus, the packet is simply forwarded on all ports except for the entry port. VLAN processing is taken into account for correct forwarding. Thus, the assumption that through an access port, only untagged packets can pass, while through a trunk port only 802.1Q-tagged packets can transit is asserted by the following code generation approach:

```
def l2Switching(bridge : OVSBridge, inputPort : Port) {  
    if (inputPort.isAccess) {  
        vlanEncapsulate(inputPort.vlanId);  
    }  
    for (vlan : bridge.knownVlans) {  
        if (packet.VLANId == vlan) {  
            for (port : bridge.ports) {  
                if (port.isAccess) {  
                    if (port.vlanId == vlan) {  
                        vlanDecapsulate();  
                        sendOut(bridge, port);  
                    }  
                } else if (port.isTrunk) {  
                    if (port.accepts(vlan)) {  
                        sendOut(bridge, port);  
                    }  
                }  
            }  
        }  
    }  
}
```

Listing 4. OVS Normal Switching

Integration

One of the most challenging aspects regarding handling complex software systems resides in making different pieces of technology work together. Throughout the course of the preceding subsections, a number of distinct blocks have been documented with focus on their packet-processing behavior. In the following paragraphs, a number of considerations regarding the way they interact are presented, as well as a general packet forwarding flow.

In Neutron, each tenant-level *port* maps directly to a network interface of type *tap*. The *tap interface* resides on a *compute node* and is connected to the OVS integration bridge. At this level, the security groups are enforced and switching behavior (such as VLAN tagging/un-tagging) is defined. Assume that a packet leaves out on some *tap interface* heading outbound. Then, the following actions apply:

- The packet enters the integration bridge which is an *OVS bridge*
- The packet enters the *OpenFlow* integration bridge through the *OpenFlow port* corresponding to the *tap interface*
- The *OpenFlow bridge* sends packet to table 0 for processing.
- The packet gets modified through the *OpenFlow pipeline*.
- If the packet goes through the *NORMAL* switching behavior, classical L2 switch behavior is implemented at OVS level.
- In the previous case, the switch will push a VLAN Id tag to the packet and flood it on all ports that belong to the given switch, except for the input port (i.e. the *tap interface*).
- If the packet exits through a *tap interface* with a VM attached, then the execution **ends (at VM)**.
- If the packet exits through the tunneling bridge, then, based on the VLAN ID, it will be assigned a tunnel id, 802.1Q encapsulated and sent out to the *Network node*.
- At *Network node* level, the packet is de-capsulated and sent to processing out on the tunneling bridge.
- At tunneling switch level in the network node, the ID of the tunnel the packet came from is translated into an internal VLAN ID.
- The packet is tagged with the internal VLAN ID and is forwarded to the integration bridge of the network node.
- At integration bridge level, the packet is broadcast to the OVS ports tagged with the VLAN tag equal to that of the packet.
- Upon receiving the packet on an access port at integration bridge level, the packet enters the Linux ip stack.
- Iptables filtering is performed as per *netfilter* hooks in the kernel
- Connection tracking is performed for the packet and, depending on its layer 3 destination address, it may be forwarded or received internally.
- If the packet is destined for an interface in the current namespace, the packet is delivered locally and the execution **ends (at namespace)**.
- Else, it is forwarded to either a bridged interface or an external interface.
- If the packet is forwarded to an *external interface*, then the execution **ends (at external interface)**.

The execution **ends** whenever the packet has reached either an **external interface**, a **namespace** or a **VM**. The enumerated outcomes are called *final ports*. In general, for describing two-way traffic between different components in the system, the *symbolic packets* arriving at any of the ports are mirrored (i.e. L2, L3, L4 destinations and sources are reversed) and resent in the system from that point.

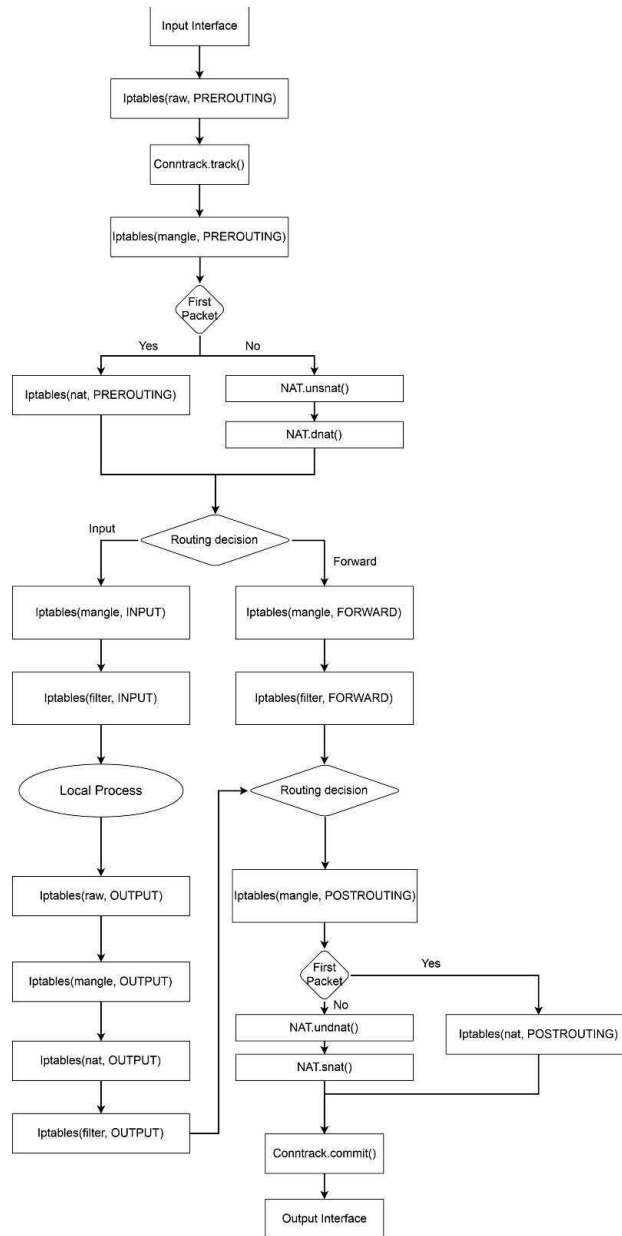
The steps depicted above present the normal forwarding (switching/routing) actions undertaken in a Linux-based Neutron deployment for a packet originated at VM level and heading outbound. Similar actions are required for other implementations.

In order to structure the packet processing flow, one can derive two basic modules which underly a Neutron deployment: a switching and a routing module. In the experiments under consideration, the switching component is comprised of multiple OVS switches, while the routing component is comprised of multiple Linux namespaces, handling IP routes and iptables rules augmented by connection tracking logic and NAT components.

The modules interact at: the interfaces at which packets enter, leave or transit the system and the connection tracking component (conntrack). Conntrack is used in because it integrates easily for creating stateful firewall processing. In Neutron, firewalling (i.e. security groups) may be obtained using either iptables or OVS drivers. For the environment under test, the OVS driver was chosen.

In principle, a packet-processing component (switching or routing) is composed a set of steps, by which a packet from an input interface is transformed, a set of component-specific variables are set and the packet is either sent out on another interface or dropped, as can be easily observed in the diagram below:

An interesting aspect regarding the interaction between components inside a Linux machine was the integration of the conntrack module specifications as per netfilter (in iptables) and those of the OVS conntrack module. To avoid limitations, the modeling process was tailored to handle both specifications.



Testing

In the following section, the results achieved testing the implementation depicted in Chapter 3 are presented. The tested scenarios, correctness of the outcomes and performance metrics are analyzed.

The experimental setup deployed in order to test the implemented features consists of two machines connected via 2 switches. The first machine, the *controller*, handles all OpenStack service layer, as well as all Neutron L3 functions (the L3 agent, DHCP agent and metadata agent). The L3 agent is of most interest for the current investigation since it is responsible for performing L3 routing between subnets as well as for providing internet access and floating IP functionality to the machines.

The second machine, *compute1* is a compute node as per OpenStack terminology and runs the virtual machines required at tenant level. From a networking point of view, *compute1* runs 3 OVS switches which enable machine interconnection and layer 2 forwarding between *compute1* and *controller*. Furthermore, the integration bridge on *compute1* enforces security group rules.

A snapshot of the experimental setup can be observed below. The topology has three distinct networks. They offer the following functionalities: the *Provider Network* is an administrator-defined network which ensures Internet access and floating IPs to the machines, the *Service Network* is responsible for driving the control plane communication between components that make up the OpenStack deployment, while the *Spanning Network* provides switching between machines using some segmentation protocol (e.g. VXLAN, GRE). For simplicity, the topology in Figure 3.4, was modified such that the *Spanning* and *Service* Networks are one and the same. For the OpenStack software components, it makes no difference if the two networks are not physically separated. In Figure 3.4, the full picture of the deployment is given.

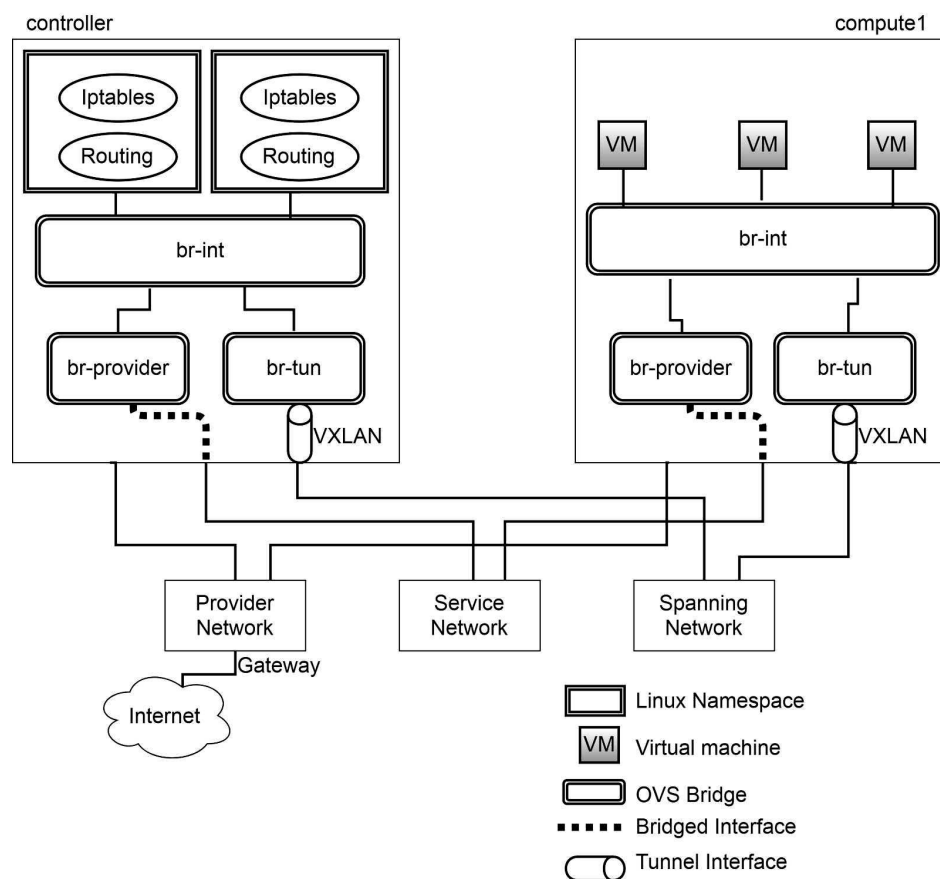


Figure: Physical experimental setup

Several tenant topologies were deployed. Two tenants were created, with two respective administrator users. For all the test cases, the two tenants will deploy a number of machines and administer networking components.

Test cases

In order to prove correctness of the modeling performed on the components and of their integration, three testing traffic patterns were considered:

1. North-south traffic via floating IPs. For this pattern, an external host attempts to reach an internal machine via floating IP. Packets flow in through the outbound interface at *controller* and are destined for a floating IP.
2. South-north traffic via SNAT. For this pattern, an internal host attempts to reach the Internet. Packets flow in through a VM interface at *br-int* in *compute1* and are destined for an external host at L3 level.
3. East-west traffic. For this pattern, an internal host attempts to reach another host on the same network. Packets flow in through a VM interface at the integration bridge in *compute1* and are destined for a host in its internally defined IP range at L3.

The traffic patterns considered are sufficient to demonstrate correctness of all modeled components in the topology. Furthermore, in subsection 3.4.2, a number of considerations are provided with respect to policy compliance of this deployment.

North-south traffic

The scenario under test uses a floating IP as packet destination and an external source IP address. In the following schema, the tenant settings **only** allow port 22 (SSH) TCP traffic inbound to the machine. The results obtained executing the *tenant* and *provider perspectives* were similar in that:

- From the *tenant perspective*, only one packet exited the port belonging to the floating IP machine, which was constrained to be an IP, TCP packet with destination port equal to 22 port.
- From the *provider perspective*, the only forward packets which arrived at the tap interface corresponding to the VM which was targeted were: IP, TCP, SSH packets and UDP packets with source port 67 and destination port 68. The latter belongs to the DHCP Discover protocol which is neglected at tenant-level.
- From the *tenant perspective*, return traffic was forwarded back to the initial destination going through the floating IP block and yielding a packet with destination TCP address constrained to 22.
- From the *provider perspective*, return traffic was also forwarded back to the initial destination and source TCP port was constrained to port 22.

Conclusion: In this experiment, end-to-end connectivity between an external host and an internal machine was proven with respect to the security policy defined to the machine (i.e. traffic filtering by TCP port).

South-north traffic

The testing scenario depicted in this paragraph shows compliance of the current system with Internet bound traffic from a local machine. The input packet was originated at a machine in the default security group with an external destination IP address. In the default security group, all outbound traffic from machines is allowed. The following outcomes were observed:

- All outbound traffic from a machine reached the external world from both *tenant* and *provider* perspectives.
- SNAT works correctly from both *tenant* and *provider* perspectives.
- All inbound traffic in reply to the initial packet was allowed back to the machine \Rightarrow reverse SNAT works from both *tenant* and *provider* perspectives

Conclusion: In this experiment, end-to-end connectivity between a machine and an outbound host was proven for all IP packets.

East-west traffic

The scenario presented throughout this paragraph shows that hosts on the same network have end-to-end connectivity, while at the same time shows tenant separation. The following outcomes were observed:

- Traffic issuing at some VM heading West reaches the DHCP Server and Router interfaces from both *tenant* and *provider* perspectives.
- Traffic between two machines in the same default security group is allowed.
- Traffic between two machines with the first in one security group and the second in another is not allowed (i.e. ingress packet policy is respected) from both *tenant* and *provider* perspectives.
- Traffic from a VM in one network is not reaching another tenant's networks in both *tenant* and *provider* perspectives.
- Reverse traffic is correctly sent back to the respective tenant from both *tenant* and *provider* perspectives.

Conclusion 1: In this experiment, end-to-end connectivity between machines running on the same network was proven for all IP packets.

Conclusion 2: In this experiment, in-tenant connectivity at Layer 3 was proven between different subnets - i.e. the L3 agent offers inter-subnet connectivity.

Conclusion 3: In this experiment, tenant separation was proven.

Results

In this section, results from multiple symbolic executions are analyzed with respect to their runtime performance. Since proving correctness of a given deployment must occur early in the installation phase and changes to the topology may occur frequently, the importance of quickly proving or disproving the properties of a given system is decisive. Thus, the results presented below take into account time as a metric for the performance of a symbolic analysis.

As per Symnet's semantics, all fork instructions yield 2 distinct and independent execution paths. With that in mind, it is obvious that *the state explosion* will occur for chained fork instructions. It is interesting to note that theoretically, the amount of paths which a symbolic packet transits is exponential in the amount of Fork instructions. However, some states will be quickly trimmed due to non-compliance to some constraints. The amount of concurrent states active at some moment is proportional to the amount of unconstrained symbolic values of the input packet. If the packet is made concrete from the beginning of the execution, the amount of concurrently existing states will drop dramatically and so will the time to execute the topology.

The experiments were performed on two distinct datasets containing information related to the state of the deployment. The *small* dataset is composed of a series of 121 configuration files, while the *large* dataset consists of a series of 134 files covering a larger number of user scenarios.

The scenario under test for the *small* dataset contains 2 virtual machines in different security groups attached to a self-service network and one router. The *deployment perspective* configuration set is composed: 151 *iptables rules* in all routing namespaces at the network node and 157 flows in all OVS bridges in the deployment.

For the *large* dataset, four more machines, a new network and a router interface were added. The *deployment perspective* configuration set is composed of 153 *iptables rules* in all routing namespaces at the network node and 275 flows in all OVS bridges in the deployment.

The most time-consuming component, as far as symbolic execution is concerned is the OpenFlow table of the integration bridge at node *compute1*. This conclusion was drawn by analyzing a set of profiling information regarding the execution time of multiple components in the system.

Tenant perspective

From the tenant perspective, all measurements which were taken indicate small processing times, even for the largest configuration sets. The greatest execution time observed in the system amounts to 5 seconds. The dependency of the execution time on the number and semantics of unconstrained packet variables is not as influential as for the *provider perspective*. In fact, the processing model for the *tenant perspective* implies explicitly cutting down illegal Ethernet and IP source and destination addresses (as per Security Policy and Address Spoofing specifications) immediately after packet insertion in the system. The number of issuing states is smaller in comparison to those in the *provider perspective*.

Provider perspective

The measurements acquired for each experiment reside in: time to parse the configuration files, forward execution time (i.e. the time for a packet to reach destination one-way) and backward execution time (i.e. the time for a reply packet to reach the initial source). To simplify the tables below, the time to parse the configuration files using ANTLR4 and a series of regular expression matchers takes on average 4.4s on the *large* configuration set and 0.5s on the *small* one.

The following tests are aimed to show the execution times of a East-west traffic pattern issuing from one VM and heading West to a known host. In this example, the known host represents a known Ethernet Destination. The statistics acquired for the current experiment are:

OpenFlow Flows at br-int compute1	215
Forward runtime	21.3s
Forward success states	6
Forward failed states	2

Backward runtime	28.2s
Backward success states	9
Backward failed states	37

Table 1: *Large* dataset: East-west traffic with Ethernet Destination bound

In the following example, a similar packet is issued at the same interface, with the only difference being that the Ethernet Destination is symbolic at input and the destination IP address is assumed constant at input. This change modifies significantly the time of execution, since in the first OVS OpenFlow table, there is no reference to the destination IP, but rather to the Ethernet Destination.

OpenFlow Flows at br-int compute1	215
Forward runtime	153.4s
Forward success states	27
Forward failed states	104
Backward runtime	222.6s
Backward success states	27
Backward failed states	68

Table 2: *Large* dataset: East-west traffic with Ethernet Destination unconstrained and known IP destination

As can be easily observed from comparing the results in tables 1 and 2, in the second case, the experiment runs roughly 10 times slower than in the former due to the exhaustive search it must perform inside the OpenFlow table at node *compute1*. Furthermore, to underline the weight of the number of flows in the first integration bridge, the following experiment was considered. Let a packet issuing some node be destined outbound to an external address. Two outcomes are discussed based on their respective dimension in terms of OpenFlow entries in the br-int table, as per table 3.3.

	Number of OpenFlow tables	
Metric	215	107
# OK States	27	15

# Failed States	113	97
Execution time (s)	67.5	39.3

Table 3: Time vs. number of OpenFlow flows for outbound traffic

From table 3, it can be easily observed that the dependency between time of execution and the number of OpenFlow Flows in the integration bridge at *compute1* is close to linear. However, in order to compare the penalties incurred by the increasing amount of configuration units in the system, a completely unconstrained input packet from some *port* in the system was sent. The results are compared for the two datasets:

	<i>Small</i> dataset	<i>Large</i> dataset
Forward time (s)	335.8	666.3
Backward time (s)	125	539

Table 4: Unbound packet analysis

The preliminary observation regarding the results in table 4 is that, even though execution time is large, its increase does not appear to be exponential in the number of configuration units, but more rather linear in this respect.

Deployment correctness

While testing and deploying the Symnet-Neutron integration, two important results were shown, stressing the importance of early verification within a cloud infrastructure. Because of a mis-configuration at *compute1* level, several OpenFlow flows were not successfully installed on the node. Thus, even though execution from the *tenant perspective* suggested that no packets were allowed from a machine to another, the execution from the *provider perspective* showed that traffic was allowed between the two. In reality, network diagnostics showed that the installed data plane configurations were not compliant to the tenant's intention.

Also, verification came to the rescue of the *tenant network administrator* at the moment where a machine was accidentally inserted in a different security group, while keeping the second in the default group. No connectivity between the two machines was available. A *tenant perspective* execution showed that the specified packet was being trimmed precisely at ingress security group level for the destination machine.

Notes on memory usage

The solution presented in this thesis aims to symbolically execute a complex network topology containing a lot of atomic instructions. As such, the expected memory consumption is large and growing exponentially with the number of *Fork* instructions encountered.

The observed behavior is that, indeed, the memory footprint of a Symnet execution is large and may incur hidden penalties, such as important garbage collection overheads (which

may, in turn, incur time penalties upon the execution engine). For the most unfavorable case, a memory footprint of roughly 8 GB was observed.

4. Optimizing match-action table verification

Due to recent advancements in switch design and architecture [0] that enable Match-Action Tables to be implemented in hardware, they are one of the driving forces for deploying Software Defined Networking(SDN) technologies. This paradigm of packet processing can be found in devices ranging from the traditional switches and routers - their forwarding information base (FIB) being represented in this way - to the more novel devices that allow custom packet forwarding(SDN - enabled) such as OpenFlow[*], P4[*], Open Packet Processor[*].

A MAT-instance consists of a number of table entries, each composed of a number of conditions and a number of actions. If a table contains a single entry, the set of conditions stated by this entry is checked against every packet that is received for processing. If the conditions hold, it is said that the packet matched the conditions and is next going to be processed according to the actions corresponding to this table entry. Otherwise, if the conditions did not hold, the packet is going to be processed according to a default set of actions, specific to every MAT-instance. In the case tables consisting of multiple entries, the packet is matched against every condition set(one per table entry). If the packet matched no entry, then the default action-set is applied. If only one entry matched, then processing happens in the same manner as with the single-entry table. If more than one entry were matched, then a tie-breaking strategy is required for ensuring processing by only one table entry. Some of the more common tie-breaking strategies are: setting a matching priority and then choosing the one with the highest priority or, in the case of routing FIBs, choosing the entry with the longest prefix(longest-prefix matching).

Modelling MATs for verification purposes

From a verification perspective, the task of deciding *what set* - or to put it slightly differently, *if a set of packets* gets processed by a given table entry is a fundamental primitive of any verification algorithm. The complexity of this task lies not in checking if a set of packets matches a set of conditions corresponding to a single table entry(which is bound in size, usually by a constant factor due to hardware limitations), but in determining what subset did not match any set of match conditions stated by table entry of higher priority. In other words, for a packet to be processed according to a certain table entry two conditions must hold: the packet must match the conditions of this table entry and it must not match any condition set stated by an entry of higher priority. As an example, for a packet to be forwarded by a router according to the default route ('0\0'), no other forwarding rule should be applicable.

In technical terms, for a table entry TE , consisting of a condition set S and an action set, to be applied S must hold for the packet P and, for every entry of higher priority TE_1, TE_2, \dots, TE_n defining the sets of conditions S_1, S_2, \dots, S_n , the negated sets of conditions $\overline{S_1}, \overline{S_2}, \dots, \overline{S_n}$ are constructed and their conjunction must also hold. Taking a look back at our

previous example of packet routing, applying the previous observation in the case of the default route would result in a number of negated conditions that is linear in the number of table entries. Furthermore, from a table-wide perspective, the total number of conditions that must be verified is quadratic in the number of entries (table size). This looks very discouraging given the commonality of FIBs with sizes in the hundreds of thousands.

To make matters worse, current MAT implementations (OpenFlow, P4, etc) support matching on multiple fields at once (in the same table entry). Even the way matching can be specified evolved, ranging from exact matching to more complex matching schemes, such as: bit masks, longest-prefix matching, ranges etc.

Another constraint is that any algorithm we might come up with must work well in the dynamic context in which data planes operate in production. Whenever rules are added or deleted, the computation of the model that considers this update should be performed at the same timescale.

Problem formulation

In this section we will give a concrete formulation of the problem.

As previously stated, a table entry TE is defined by two components a set of matching constraints S a packet must satisfy in order to be processed by the actions defined for that particular TE - which represent its second component. We will focus for the rest of this writing on just the first part of a TE .

The size of a constraint set S is dictated by the number of match conditions defined by the structure of the MAT. For instance if the MAT instance is a routing FIB, then the constraint set has only just one member - a longest-prefix match condition imposed on the IP destination field; another example would be a table that performs filtering based on the value of the destination port and transport protocol number - in which case the constraint set would be of size two, denoting two equality conditions imposed on destination port and protocol number. We will denote this by $S_n = \{C_{n1}, C_{n2}, \dots, C_{nm}\}$; in which C_{xy} denotes the y th condition belonging to the x th TE in the MAT; having the property that for each $j \in \{1, 2, \dots, m\}$ and any x , conditions C_{xj} affect the same field. This last property states that each TE should respect the MAT structure - the field being matched are consistent across TE s.

We define an overlap as the case in which for two TE s: TE_x and TE_y there is at least one pair of conditions C_{xj} and C_{yj} that can hold for the same packet.

For any such pair of TE s, the less prioritary one has to be augmented such that there will be no overlap. The trivial solution for this, building a new condition set $S_{x'} = \{C_{x1} \wedge \neg C_{y1}, C_{x2} \wedge \neg C_{y2}, \dots, C_{xm} \wedge \neg C_{ym}\}$, would clearly yield a valid $S_{x'}$ in the sense that there are no more overlapping conditions, clearly. The problem is one mentioned before that it would scale poorly in the number of extra conditions that are to be added - yielding a heavy toll on

the constraint solving procedure. Thus the goal is to build an algorithm that would yield a constraint set $S_{x\min}$ that is minimal in the number of extra conditions added to mitigate every overlap, but would work at time scales comparable to those at which MAT updates appear in practice.

We have built on the body of research already existing [1,2] on this subject by taking a step back and targeting a broader solution that would be applicable in a wide range of possible matching methodologies (lpm, range matching, wildcards etc) - that can easily incorporate specific optimizations (given a specific matching method) without requiring extensive modifications.

Our solution

We have started with the ADT represented by a forest of multiple-node trees. For every condition type, there will be one condition C_{xj} per table entry TE_x . For this set of conditions that share the same type (the same matching procedure applied to the same packet header) $\{C_{1j}, C_{2j} \dots C_{nj}\}$; n representing the size of the table (number of table entries) we will build one such forest. In this forest, every node will correspond to one condition C_{xj} .

In this data structure there can be four types of relations between the nodes:

- The nodes are completely independent if their corresponding conditions C_{xj} and C_{yj} do not overlap
- Parent-of - this representing a down-link in a tree between a parent node and a child node if the condition corresponding to the child node denotes a field space entirely contained by that denote by the condition of the parent (the parent's condition is more general and includes the child's condition)
- Ancestor-of - this representing a relation between two nodes that are connected in the same tree by several 'parent-of' links
- Neighbor-of - connecting two nodes that correspond to overlapping conditions, but neither condition is fully contained by the other; and no node already has an ancestor node linked to the other node via a 'neighbor-of' link. In other words, if there is a 'Neighbor-of' relation between two nodes, then there will be no other such link between descendant nodes of these two.

Having presented the definition of the ADT, for every condition C_{xj} in order to determine the simplest condition that would match condition C_{xj} but no other overlapping condition, one builds the set of conditions C as the union between the set of conditions corresponding to 'child-of' and 'neighbor-of' nodes and then negate every condition in this set. Note that, given the structure of the tree this operation has $\Theta(|C|)$ complexity.

The optimality of this algorithm in terms of the number of conditions added to mitigate the overlap comes from the fact that if a node is selected ('child-of' or 'neighbor-of') then there is no need to consider any of its descendants, since their conditions are fully covered by the one of the selected ancestor, thus rendering them redundant.

We implemented the forest construction algorithm in its most general form, that can handle range-matching, longest-prefix matching, wildcard matching etc. with no modification - given there is an implementation to test the relation between two nodes. Even without specific optimizations that can be made given the type of matching, we saw an improvement in the runtime of the model-construction algorithm as compared to the default implementation in Symnet, and also a massive reduction (2x) in the number of constraints being generated. We evaluated the implementation using FIBs taken from Stanford backbone router - having more than 180 000 entries.

Table size	Forest height	Average time to build model for one TE	Number of constraints
183 369	6	5ms	298 112
62 396	6	1.45ms	96 189
1 815	4	0.1ms	2 791

We can conclude that even for very large scale tables, the cost to construct the model corresponding to a new insert is still under 10ms, which makes it feasible to update the model at the pace at which MAT updates can occur.

[0]: Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. SIGCOMM Comput. Commun. Rev. 43, 4 (August 2013), 99-110. DOI: <http://dx.doi.org/10.1145/2534169.2486011>

[1] A NICE Way to Test OpenFlow Applications - Marco Canini, Daniele Venzano, Peter Perešini, and Dejan Kostić, EPFL; Jennifer Rexford, Princeton University; NSDI '12

[2]:VeriFlow: Verifying Network-Wide Invariants in Real Time - Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey, University of Illinois at Urbana-Champaign, NSDI '13

5. Anomaly detection

In a previous report, we proposed a novel universal anomaly detection algorithm, which was able to learn the normal behavior of systems and alert for abnormalities, without any prior knowledge on the system model, nor any knowledge on the characteristics of the attack. The suggested method utilized the Lempel-Ziv universal compression algorithm in order to

optimally give probability assignments for normal behavior (during learning), then estimate the likelihood of new data (during operation) and classify it accordingly.

In this report, we evaluate the algorithm on real-world data and network traces, showing how a universal, low complexity identification system can be built, with high detection rates and low false-alarm probabilities. We first apply the detection algorithms to the problems of malicious tools detection via system calls monitoring and data leakage identification. We then give some results for detecting anomalous HTTP traffic, e.g., Command and Control channels of Botnets.

Detection of Malicious Tools and Data Leakage

We apply the anomaly detection system suggested to system calls in order to detect malicious tools on a Windows machine, and to TCP traffic of a server in order to detect unwanted data leakage. In both experiments, the capability of the tool to detect abnormal behavior without prior knowledge is demonstrated.

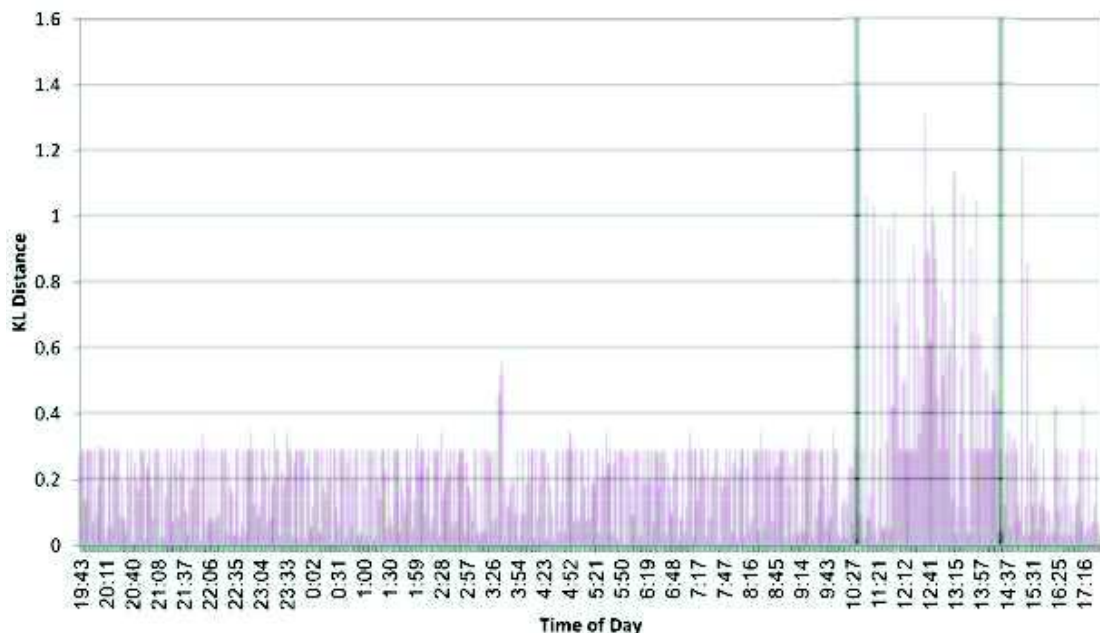
Monitoring the Context of System Calls for Anomalous Behavior

The sequence of systems calls used by a process can serve as an identifier for the process behavior and use of resources. Moreover, when a program is exploited or malicious tools are running, the sequence of system calls may differ significantly compared to normal behavior, incriminating the program or entire machine. The universal anomaly detection tool was used to learn the *context of normal system calls*, and alert for anomalous behavior. Specifically, the sequences of system calls created by a process (e.g., firefox.exe) were recorder, processed, and learned. Then, when viewing new data from the same process, the anomaly detection algorithm compared the processed new data to the learned model in order to decide whether the process is still benign, or was it maliciously exploited by some tool.

Due to the large amount of possible system calls, calls were grouped into 7 types, based on the nature of the call: *Device, Files, Memory, Process, Registry, Security and Synchronization*. That is, the quantization process did not include any minimization of distances or a requirement for uniform probabilities, but, rather, labeled the calls based on their known functionality. Recording and classification used NtTrace.

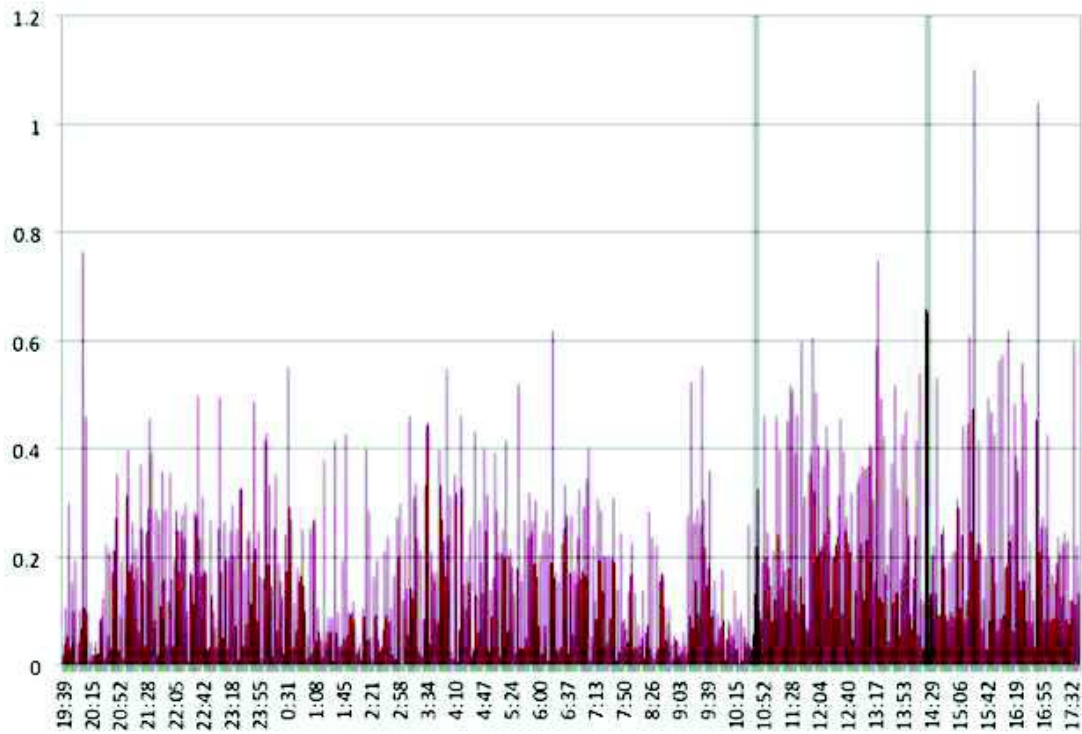
In the learning phase, system calls were recorder, quantized according to the types above and then a discrete sequence over the alphabet of size 7 was created. The sequence was used to build the (normal behavior) LZ tree, as described in the previous report, from which a histogram for the probabilities of tuples of length 20 was calculated. This histogram was *the only data saved from the learning phase*. The learning phase included 4 hours of data.

For testing, segments of 2 minutes were recorded. For each segment, a histogram was calculated, similar to the learning phase (calculating probabilities for tuples of length 20 over an alphabet of size 7). Decisions were made based on the Kullback-Leibler divergence between the normal histogram and the tested one.



The figure above plots the KL distances between the histogram during the learning phase, and the histograms extracted during the testing phase. *The process tested was firefox.exe, and the two vertical thick lines mark the time when the tool “Zeus” was active.* It is very clear that the context of the system calls changes dramatically when the tool is active, and that simple monitoring of the KL distances every few minutes is sufficient to detect a change in the system behavior. Note, however, that the specific tool used, Zeus, installs instances in several processes simultaneously, hence might not be as active within all processes.

The next figure below depicts the same setting, yet with winword.exe. It is clear Zeus is harder to identify within that process.



It is clear Zeus is harder to identify within that process. Nevertheless, from both figures, it is clear the anomaly detection algorithm suggested, when it monitors a few processes on the machine, can easily identify malicious behavior *with minimal delay and complexity*.

Identifying Data Leakage

In this part of the work, the universal anomaly detection algorithm was used in order to identify data leakage from a web server. Specifically, the setting was as follows. In the learning phase (a period of a few days), benign traffic on a web server was recorded using Wireshark. Similar to the previous examples, timing-based sequences were extracted, quantized and used in order to build an LZ tree. This LZ tree served as a model for normal data.

Then, using Ncat, a script was installed on the server. This script initiated downloads of large chunks of data from the server. Several periods, each 30 minutes long, of traffic which includes Ncat were recorded. For comparison, similar length periods of traffic without Ncat were recorded as well. An LZ tree was built for *each of the 30 minutes datasets*.

To identify data leakage, unlike the Botnets setting considered in Section V, in this case, we compared the *joint distributions of k-tuples* resulting from the LZ trees. That is, we used the distribution of k-tuples resulting from the LZ tree as an identifier for the data set, and calculated the distances between the distributions.

	Ncat1	Ncat2	Normal1	Normal2	Normal3	Normal4	Normal5	Normal6
MSE	0.962	1.262	0.044	0.153	0.143	0.43	0.142	0.017
KL	2.05	17.163	1.353	1.228	2.026	4.12	2.121	1.396

The table depicts the distances between the learned, normal data, and 8 testing periods, two which include data leakage using Ncat and 6 without. Two distance measures were used in this part of the work: Mean Square Error (MSE) and KL distance. Under MSE, the leakage sessions clearly stand out compared to normal data. Results under the KL distance are less clear, especially in the first Ncat session, which included more normal data than the second.

Finally, to further challenge the algorithm, and see whether data leakage will also stand out when the normal communication includes (peaceful) massive downloads, the normal communication was augmented with benign downloads of various sizes. The table below depicts the results (under the KL distance). It is clear that while Ncat stands out compared to normal traffic on the web server, it is almost indistinguishable when the normal *traffic learned includes downloads of large files*. This is expected, as Ncat uses a similar protocol, and the key differences in the timing are caused by file sizes. Hence, data leakage is clearly detected compared to normal surfing, yet, it is indistinguishable when the server, in peaceful times, serves large downloads.

	Normal	Normal + 1.3MB	Normal + 10MB	Normal + 200MB
Normal	0.906	0.843	0.583	0.72
Ncat	19.05	0.787	0.733	0.353

Identifying Anomalous Traffic

In this part of the experiment, we used the algorithm to identify anomalous traffic. Specifically, traffic between sets of hosts and clients was tested, and the goal was to identify anomalous traffic, e.g., HTTP traffic used as command and control of Botnets, etc. Each client, denoted by 'cid', may connect with several hosts (web-servers), denoted by 'hid'. On each transaction, data is sent both by the client and the host. This defines communication pairs CID HID. Each transaction is labeled as either legal, denoted by 'good', for normal data traffic generated by the client, or illegal, denoted by 'hostile', that is, Bot traffic. Labeling was done by the security company's experts based on well-known black-lists. Note that these labels are not used during the classification process. They are used only in the validation phase.

Each transaction is represented by a single record in the data set, which consists of the following fields: 'time', referring to the time the transaction took place; 'time-taken', is the total time the transaction took; 'cs-bytes' and 'sc-bytes' fields represent the total bytes sent by the client/server(host) to the server(host)/client during the transaction, respectively; 'mime-type' denotes the Internet content type of the transaction, such as: plain text, image, html page, application, etc.; 'cat' is the category of the transaction - 'good' or 'hostile'; and the 'hid' and 'cid' fields refer to the host-index (Internet site, web- server) and client-index respectively. Indices were given arbitrary to protect the identity of the hosts. However, some malicious sites are identified by their domain name, e.g., 'hotsearchworld.com' or 'blitzkrieg88.bl.funpic.de'.

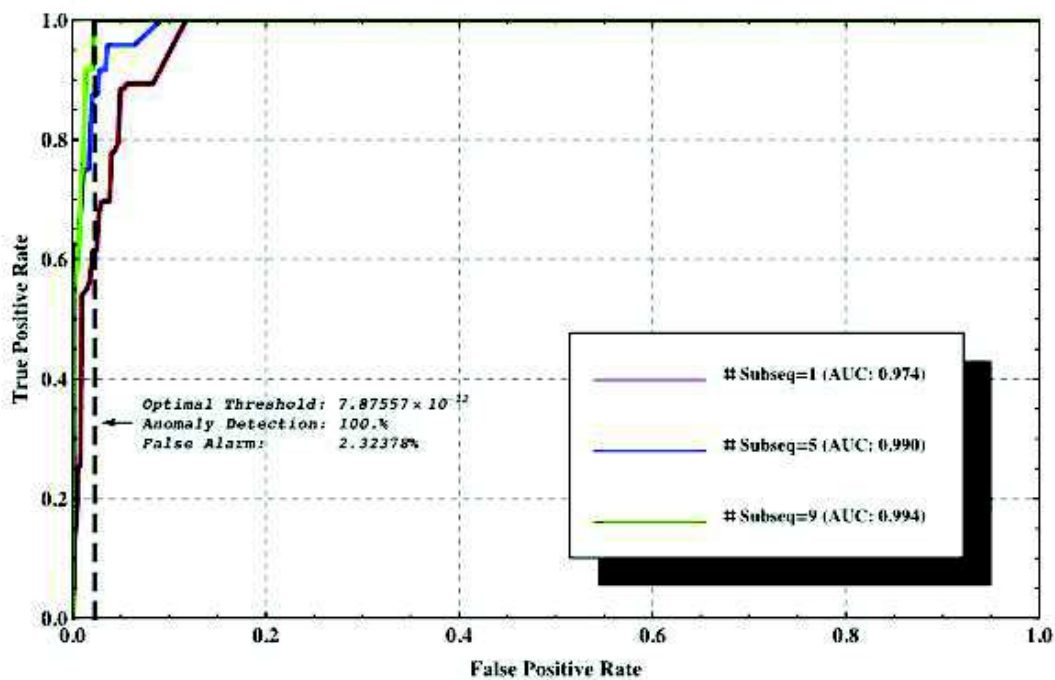
Processing of the data included serialization and feature extraction: first, the given data set is split into set of flows based on CID HID connections. A 'Flow' is a sequence of related transactions of the same communication pair CID HID sorted by time and with the same

label, either 'good' or 'hostile'. In total, there are 19164 flows labeled as 'good' and only 65 'hostile' flows (0.338%). This indicates the imbalance of the data set where most of the transactions are legal and only a small fraction is illegal. However, this is characteristic of real network traffic behaviour.

Next, selected features are extracted from each transaction, e.g., Time-Difference, Time-Taken, Server-Client-Bytes and Client-Server-Bytes. After quantization, the resulting sequences are the discrete time, finite alphabet sequence on which learning and testing was performed.

The best results, in terms of optimal threshold and ROC-AUC (Area Under Curve), were achieved using the Time-Difference (TD) representation of the data sequences along with the 'Uniform' quantization (several quantization algorithms were tested). To better understand why TD was superior, consider a *legitimate web surfer compared to a hostile connection using HTTP only as a C&C channel*. While the surfer must have a reasonable behaviour in the time domain, affected by the times required to read a page, the times required for the server to respond, etc., a C&C channel may behave differently, without, for example, a reasonable response time from the server as it only collects data from the bots, and the "GET" messages are used solely to *transmit* information. Due to space limitation, we do not include the results for the inferior features, and focus on the results under TD and uniform quantization.

Still, using TD, the optimal threshold for 100% detection results in 11.75% false alarms. However, *this is when only a single, short sequence is tested*. To further improve the above results, a majority vote for several sequences within the flow can be used. Each data segment is partitioned into several subsequences of length 10. The classification is done based on the majority of these subsequences' estimations, as either positive or negative, resulting in better classification performance as the number of subsequences is higher. For example, an AUC of 0.994 and false alarms rate of 2.32378% are achieved using a threshold of 7.87557×10^{-12} , as illustrated in the figure below.



A zoom on the relevant part of the figure is also available:

