SUPERFLUIDITY

A SUPER-FLUID, CLOUD-NATIVE, CONVERGED EDGE SYSTEM

Research and Innovation Action GA 671566

DELIVERABLE I5.2B:

MECHANISMS FOR NETWORK SERVICE DYNAMICS AND PERFORMANCE

Deliverable Type:	Report
Dissemination Level:	CO
Contractual Date of Delivery to the EU:	2017-03-31
Actual Date of Delivery to the EU:	2017-04-11
Workpackage Contributing to the Deliverable:	WP5, T5.2

Editor(s):	John Thomson (ONAPP)
Author(s):	ONAPP (John Thomson, Michail Flouris, Anastassios Nanos, Xenia Ragiadakos, Julian Chesterfield), BT (Louise Krug, Paul Veitch, Phil Eardley), INTEL (Tomasz Kanceki, Edel Curley), NEC (Filipe Manco, Felipe Huici), ALB (Carlos Parada, Francisco Fontes), CNIT (Claudio Pisa, Stefano Salsano), REDHAT (Luis Tomas Bolivar)
Internal Reviewer(s)	Juan Manuel Sanchez (TELCARIA), Christos Tselios (CITRIX), Luis Tomas Bolivar (REDHAT)

Abstract: The task from which this Internal Deliverable is produced is Task 5.2. In Task 5.2, the aim is to design and implement next generation, superfluid capabilities for virtualized network services running on the SUPERFLUIDITY platform at speeds of 10Gbps and above. In order to reconfigure in the order of 10s of milliseconds, quick instantiation of services needs to be supported. To this effect we investigate how unique services can be supported in a rapid and scalable manner. Improvements are made throughout the stack to improve the performance at each layer and also to combine the improvements to allow for the ambitious goals to be realised. This Internal Deliverable reports on the status of the various piece-meal improvements that are being carried out at different layers. Their integration into the full stack will be reported in the final Deliverable associated with Task 5.2, namely D5.2.

Keyword List: Virtualisation. Performance. Network. Service aggregation. Scalability. Millisecond instantiation time. Configuration. End-to-end service support.

INDEX

List of Figures

List of Tables

Glossary

(TO BE COMPLETED IN THE FINAL VERSION OF THE DELIVERABLE)

SUPERFLUIDITY DICTIONNARY	
TERM	DEFINITION
API	Application Programming Interface
App	Application
eNB	E-UTRAN Node B
EPC	Enhanced Packet Core
ICMP	Internet Control Message Protocol
GTP	GPRS Tunnelling Protocol
MEC	Multi-access Edge Computing
MEC App	MEC Application running in the edge of the network
MANO	Management and Orchestration
NAT	Network Address Translation
NFVI	Network functions virtualization Infrastructure
SFC	Service Function Chaining
SGW	Serving Gateway
TEID	Tunnel Endpoint Identifier

TOF	Traffic Offloading Function
UE	User Equipment

Table 1: SUPERFLUIDITY Dictionary.

1 Introduction

This Interim Deliverable records 9 months of effort of Task 5.2 “Network Services Dynamics, Performance and Scalability”, between M13 and M21 and has been edited by ONAPP. The final deliverable is due to be completed 9 months from this Interim Deliverable and will be prepared by NEC. As such contributions are incomplete at this stage and the emphasis for someone reading this document is to understand the methodology and aims for the Task and to understand how it fits in with the scope of the rest of SUPERFLUIDITY. Effort has been taken to avoid repetition of content from I5.2 that was produced for M12 but as it reports the continuation of effort in the task some repetition has been necessary. The final version of the deliverable will capture the content from I5.2, I5.2b and the new content created between now and then.

The document is structured as follows;

In this Section, Section 1, we detail the structure of the report, detail the motivation, capture the quality assessment performed and provide an Executive Summary.

We then start by detailing investigation work, carried out by BT, into various container management solutions in Section 2.

Next, we cover the management and performance of Virtual Interface Manager (VIM) platforms in Section 3 that has been carried out by CNIT.

We then cover the work describing Multi-access Edge Computing (MEC) in Section 4.

Effort from NEC into the ‘Superfluid platform’ is recorded in Section 5, which is the basis for a forthcoming paper submission to SOSP.

Open vSwitch can be used to provide the networking layer and as such the performance builds upon this as a basis. This effort has been carried out by REDHAT and is recorded in Section 6.

Work from BT Adastral Park has been recorded into the investigation of Intel Cache Monitoring Technology and Cache Allocation Technology (CMT/CAT) as tools to mitigate and protect workloads against resource-hogging effects of NFV Noisy Neighbours. This effort is recorded in Section 7 The main content of that section has been submitted and accepted for presentation at the forthcoming IEEE NetSoft ’17 conference in Bologna, Italy.

Analysis of the MicroVisor, distributed hypervisor platform is carried out in Section 8. This work has been carried out by ONAPP.

Finally, in Section 9 we draw conclusions on this interim deliverable and detail how these technologies that have been detailed are expected to come together in the SUPERFLUIDITY realisation, carried out in WP7.

1.1 Deliverable Rationale

The emphasis on Task 5.2 is to create the capabilities for virtualised network services running on the SUPERFLUIDITY platform. This work therefore focuses on the platform improvements and optimisations needed to enable network functions (see use cases in WP2, specifically D2.2) and Re-usable Function Blocks (RFBs) as described in the SUPERFLUIDITY architecture (see I3.1 for the current version at time of delivery and D3.1, which is due at the same time as D5.2).

The focus therefore is to develop support for extremely light-weight virtualisation techniques that allow network functions to be run independently and isolated from each other but that can also be chained together to form more complex services, as described in the Service Function Chaining (SFC) analysis in D/I3.1. The motivation for light-weight virtualisation techniques as opposed to legacy virtualisation methods is that it allows for greater consolidation (more services can be located at the optimal location), greater scale (more services for a given set of resources), easier deployment (short boot up times) and better manageability (orchestration of a large number of workloads across a large set of devices).

Given the larger bandwidth and capacity envisaged for the increasing number of devices in 5G-PPP the core and backhaul network will need to support faster links and respond rapidly and efficiently to service requests. In order to dynamically handle network packets and user-load, it is essential to move away from pre-allocated static VMs to rapidly deploying service responders.

In recent years, light-weight containers and specialised Unikernel systems have been developed to bridge the gap between the increase in demand and the capabilities of legacy virtualisation systems. There is invariably a trade-off as the systems are getting smaller and more specialised for certain functions. This has been the case for the divide between hardware and software for many years, with functions that are specialised and requiring high performance being designed and implemented in hardware but at the cost of being hard to modify and requiring a long deployment time. The same choices are becoming apparent for software virtualisation techniques, with the trade-off between design, build, configuration and deployment time being compared to the performance.

The vision of this task is to tie in with the KPI mapping in WP4 and orchestration elements in WP6 to allow for the assessment of workloads running in a platform and to then determine whether a given workload is better suited in a specialised light-weight or generic VM and calculate the cost of reconfiguration. The platform support and analysis will allow a comparison of the performance for different types of workloads in different execution/virtualisation environments.

When the performance of light-weight containers and specialized Unikernels increases, in an attempt to reduce the instantiation and boot time up to the order of tens of milliseconds or even less, the performance of Virtual Infrastructure Managers (VIMs), that control them may become a critical bottleneck. Therefore, this deliverable also considers the VIM performance, by defining models for their characterization and by identifying solutions for their improvements.

1.2 Quality Review

Review Team member responsible of the deliverable: Juan Manuel Sanchez (TELCARIA), Christos Tselios (CITRIX), Luis Tomas Bolivar (REDHAT)

VERSION CONTROL TABLE			
VERSION N.	PURPOSE/CHANGES	AUTHOR	DATE

1.3 Executive summary

1.3.1 Deliverable description

This internal deliverable will carry an initial survey and measurements of available virtualization technologies (e.g., type-1 hypervisors such as Xen, type-2 hypervisors like KVM or lightweight virtualization mechanisms like containers) to estimate and document the trade-offs between them and evaluate their suitability towards achieving the primary platform goals of quick instantiation, service migration and massive consolidation, together with any auxiliary functions. This deliverable will further evaluate such mechanisms on different types of commodity-hardware, from full-fledged x86 servers down to microservers. Finally, the deliverable will consider the performances of Virtual Infrastructure Managers (VIMs) that control the virtualization technologies.

Task description – T5.2

Description: This task will design and implement next generation, superfluid capabilities for virtualized network services running on the SUPERFLUIDITY platform at speeds of 10Gbps and above. More specifically, we will provide mechanisms for near-instantaneous reconfiguration (e.g., in milliseconds, such that changing or even moving the processing between servers is transparent to end points), quick instantiation (e.g., we could target the ability to instantiate network processing on a per-flow basis, or on a as-needed basis, as new flows arrive) and dynamic scalability at tiny timescales. To achieve these goals, the work will look into specialized virtual machines and OSes, but also into lightweight virtualization techniques such as containers (e.g., LXC, Docker or FreeBSD Jails). Further, the task will target massive consolidation of network services, perhaps thousands or

even higher number of VMs on a single, inexpensive commodity server, with the aim of (1) severely reducing the platform operator's operating and electricity costs and (2) improving the scalability, and thus the fluidity, of the overall architecture.

To complement the migration of processing we also need to coordinate the migration of network traffic. To this end we will rely on Openflow/MPLS for routing inside operator networks, and will also integrate VM migration and cellular offloading mechanisms with endpoint mobility, such as Multipath TCP.

In the RAN part, we will study and implement dataflow engine with dynamic allocation and mapping capabilities for Cloud-RAN applications, enabling simultaneous operation of multiple virtual baseband stacks developed in WP4 on same SUPERFLUIDITY platform. The engine will cope with modularized, scalable and heterogeneous hardware platform by integrating scheduling algorithms and methods of T5.1 in order to allow dynamic mapping of abstracted task graphs to particular hardware arrangement and configuration. The engine will make use of standard parallel programming interfaces (e.g. POSIX) for the purpose of seamless portability and dynamic instantiation within SUPERFLUIDITY platform. The main motivation is to focus on engine architectures allowing dynamic adaptation and configuration to particular workload characteristics which is inevitable for Cloud-RAN applications.

1.3.2 Summary of results

These will be produced for the final iteration of the Deliverable that is due in December 2017 (M30).

2 NFV container management analysis

In this section, we describe our findings on how we could operate and manage Network Functions inside containers. There are many different management options for containerized applications, but we demonstrate that NFV applications have some unique features that render normal management solutions currently incomplete. Following a mixture of documentation review, simple implementations and discussions with the wider container community we identify certain solutions that could be developed appropriately and also provide a more detailed review of these approaches.

2.1 General container management systems

2.1.1 Future NFV architectures

In order to assess the management tools, we need to consider their operational environment. The easiest situation to envisage is that containers are simply used as a replacement for a VM NF. These virtual functions are located in large, managed data centres and need to support the networking of medium to large enterprises. These containers would therefore exist on a per-customer basis and we may wish to ensure the quality of experience as we do today with VMs, through over-provisioning and resource pinning. In this case, the fast spin up times of containers are likely to be restricted to handling of failures. The hardware is likely to be fairly homogeneous.

Going forward, we may look at providing increased flexibility for these customers giving them the ability to scale out some of the functionality, perhaps using a micro-services architecture. To date, what micro-services really mean for NFV is still an open issue. Nevertheless, we can understand that micro-services will lead to greater dynamics, requiring the need to flex not just the containers, but the container networking.

Getting more complex, we may wish to own the containers and the platform or we may be providing a container platform for other tenant container providers. A simple architecture for this just provides a VM per tenant allowing them to use containers; but going forward we may like to consider a container-native solution as it could allow for much greater efficient resource utilisation.

At the same time, we need to be able to provision the containers at specific, remote customer locations. The hardware here is likely to be more heterogeneous. There is a greater chance that the customer will be configuring and using the hardware separately from anything the operator is doing.

Finally, in a “fog” scenario, the required functionality would be instantiated at the “best” location based on the required customer experience and resource utilisation. Resources could be available within the network edge as well as in remote data centres or inside customer premises. The functionality may also need to follow the customer movement between different locations, introducing the need for efficient handover mechanisms.

The specific issues with containers compared with VMs are that containers tend to be smaller, more dynamic and ephemeral than VMs, with the ability to trade performance and isolation against resource usage. In addition, containers tend to be configured at build or run time, unlike VM NFs which may be configured as a hardware box once launched. NFV applications also have many differences to conventional container applications. NFV applications include security functions such as firewalls and intrusion detection, performance enhancing and measuring functions as well as basic network functions such as routing. Unlike common container applications:

- NFV applications typically require multiple network connections, often with predefined static network addresses. Multicast network support may also be needed. Integration with SDN will become increasingly important
- NFV applications are frequently stateful
- Geographic function placement and potentially limited network connectivity are much more significant. In addition to providing NFs in the data centre, we must consider both the customer premise and the future fog (where the container location is chosen dynamically based on end to end network performance and utilisation).

2.1.2 Summary of management tools investigated

The table below summarises our findings of a number of different container management solutions when applied to NFV applications. In the next section, we look in more detail into Kubernetes and Ansible.

Table 2: Strengths, weaknesses and comments regarding various VIM management systems

TOOL	STRENGTHS	WEAKNESSES	RECOMMENDATIONS
Swarm	Start Docker over a selection of hosts	Very limited functionality and configurability. Docker only	Discounted –small scale and propriety
Kubernetes	Schedule/deploy a group of related containers.	Very basic connectivity. Best for homogeneous infrastructure. Docker only	Detailed investigation as the community is open to new ideas
Openshift	GUI to manage containers, providing load balancing, auto-scaling, recovery, resilience, upgrades with roll-back	Relies on Kubernetes,	Reliant on changes to Kubernetes – re-examine once Kubernetes networking is uplifted
Openstack	A dominant complete and mature VM solution looking to integrate with Kubernetes	Heavyweight, complex, data centre focussed. Containers not equivalent to VM	Recommended for further investigation due to its compatibility with

		– relies on Kubernetes	VM NFs
Mesos/ marathon	2-level scheduler, deployment, load balancing, scaling, recovery & resilience	"Lacks coherent network story". Multiple IPs per container not possible.	Discounted as no plans to change networking
Ansible // Tower	Flexible yet simple and lightweight scripting and inventory management toolset designed for multi-tenant.	No scheduling mechanism, limited automatic monitoring	Detailed investigation. Has proved useful in demos. Community open to new ideas
Rancher	GUI to manage containers, on top of schedulers such as Kubernetes, adds image catalogues, authentication tools and common services (load balancer)	Targeting enterprise market-place. Docker only. Inventory management could be awkward with many systems	Useful for small scale demos
OSM	Strong architecture	immature – not launched in time for investigation	Recommended for further investigation at a later date

2.2 Key Technologies

Here we delve deeper into the most important tools, presenting conclusions from deployment experiments.

2.2.1 Ansible

2.2.1.1 Overview

Ansible is an open-source command-line solution to help manage systems and application deployment across multiple systems (system configuration). Redhat's Ansible Tower is a


commercially available product[1] that provides a GUI to Ansible and adds some features such as the ability to schedule events, attribute changes to particular users and provide event notifications.

2.2.1.2 Understanding Ansible

The core of Ansible is the playbooks which describe the desired system state, and the inventory of systems that you are managing. When a playbook is run, Ansible compares the required state to the defined state and modifies the system as required. Where state managing modules are not available, simple commands can be executed, although then care will be needed to manage all outcomes. Note that whilst the basis is state checking, it is essentially a combination of declarative and imperative behaviour.

The inventory can be a very simple list of systems or dynamically created e.g. from a set of cloud resources or a database. Variables can be associated with the systems or groups of systems. Within the Inventory, hosts can exist in groups (eg "Web-Servers" and "Docker-Hosts"). Systems can be in multiple groups; and groups can be created from other groups. Playbooks are then run against systems in the inventory – for example we can run a patching playbook against all "Web-Servers" that are not "Redhat" servers. Tower allows us to set up templates to facilitate the running of playbooks – for example prompt for specific variables or specify specific systems. It also allows the scheduling and execution of playbooks.

We showed how we could use Ansible to provide containerised advanced fault analytics on edge CPE and also to build and deploy a Docker Quagga router, such as might be useful within a data centre deployment. Here, the fact that Ansible can be used for many more things than just containers was very beneficial, as we could connect the routers to OVS switches within the same playbook making it easy to track container connectivity.

Ansible requires no agent on the managed device, but does need SSH[2] and a way of managing keys. To control Docker (and probably to run other modules) some python code is also needed.

2.2.1.3 Ansible Conclusions

The main advantages of Ansible are its low complexity and its good documentation. It is lightweight, only requiring SSH access to the hosts. It allows us to integrate network and application management within a single playbook. It has a large number of predefined modules, but also allows us to execute basic commands on any system. It allows us to define actions to take on playbook errors, and can be used to deploy a staged upgrade (rolling upgrade) across systems to avoid

[1] £46 - £70 per year per managed host, standard support level
[2] A few other methods of connecting to hosts are possible


service outage. Tower could be used to ensure that container images and system patches were kept up to date. Ansible needs to be combined with other tools to provide a complete management solution.

It does not have a scheduler. Whilst it is easy to fix the geographical location through use of the inventory, it is less easy to choose to deploy on a server with lots of free resource for example. If we want a deployment of 1 to N instances, Ansible simply places the instances on the first matching systems in the inventory list. Whilst customised rules could be built - dynamically updating the inventory with system load information for example, for a pure container workload, instead a separate Kubernetes (or other) scheduler could be run, through which Ansible could deploy containers.

Tools will also be needed to help provide the operational features such as rapid failure detection and response to load. Whilst Ansible can deploy systems with a load balancer, it does not manage the service performance, for example scaling systems automatically in response to high load (although an underlying scheduler could provide this). With Tower, it is possible to schedule checks that a system is still in the required state, but this is may be a heavyweight approach if looking at identifying faults in sub second timescales over millions of systems. Another approach is to constantly pull/push metrics and analyse these for unusual behaviour using telemetry applications such as SNAP.

For NFV, one key question is how to reconfigure a service chain with minimal service downtime. When working at large scale, the different systems may be in different initial states (e.g. power failure or a local manual reconfiguration). Further experimentation is needed to understand how to manage this.

Overall Ansible and Tower are very useful tools that will likely be used in applications beyond just containers. They allow for the deployment of SDN and containers, as well as providing integration points with scheduling and monitoring solutions.

2.2.2 Kubernetes

2.2.2.1 Overview of Kubernetes

Kubernetes is an open source system for the deployment, scaling and management of containerised applications. It consists of a master node, with additional software installed on each Docker host to make a Kubernetes cluster. Although it is not yet suitable for general NFV applications, it is in scope because of its popularity within the container world and the willingness of the community to develop the system.

Some key concepts of Kubernetes are:


1. Pods. A group of one or more related containers than share a network namespace and are best co-located. A Pod may be a group of micro-services that provide a service. Multiple identical pods may be deployed depending upon the expected load.
2. Service. A construct that can load balance across pods. The service address does not change if a pod changes host. This service endpoint may be a load balancer.
3. Networking plug-ins: There are a number of different systems that can be used to provide the networking between the Kubernetes elements (i.e. between the containers). Each networking option operates in different ways in terms of the use of NAT, tunnels and IP addresses. Note that the method used to reach the containers externally is independent of this internal cluster networking mechanism.

2.2.2.2 Installation of Kubernetes

The technology is immature, rapidly changing and documentation weak. For example, it took several attempts to successfully deploy a Kubernetes cluster. The first successful build was on Centos 7 bare-metal using custom build with the flannel network option. This lets us run Kubernetes, deploy containers from our own repository, and use our own defined network address pool. Other successful builds, all on Centos-7 have since been made using the alpha release Kubeadm system. Kubeadm[3] is a utility that helps an admin to set up a Kubernetes cluster by setting up role based access control (RBAC).

2.2.2.3 Understanding Kubernetes

The interface is command line driven and scrappy (the new web interface is immature). The --help option leads to a stream of information making it unsuitable for beginners. There are many ways that things silently fail. For instance, “kubectl run” may originally appear to have run successfully, however when the system is later queried it is possible to discover that there was a typo in the image name which means that the container has not actually been deployed.

Kubernetes’ main role is as a tool that lets us schedule containers across multiple hosts. Kubernetes automatically schedules pods to the less loaded system based on CPU and memory load. This may not always be the best choice – for example if we wish to dynamically flex the number of available hosts to manage our power consumption, then running containers on the fewest number of systems is preferable.

The scheduler is policy configurable, for example by giving a node a label of `disk_type=ssd`, we can then specify that the Pod must run only on nodes with `disk_type=ssd`. The Kubernetes

[3] https://kubernetes.io/docs/getting-started-guides/kubeadm/


team are developing this use of labels, for example to allow "***prefer*** **disk_type=ssd**", or "***not*** **co-located with specifically** ***labelled Pods***". There is no (obvious) inventory file or database that can be used to separately manage the labels associated with hosts. Labels could be used to give geographic meaning to a node –for example *location=phone number*. However, based on the experience of using the platform, Kubernetes is unlikely to work well over a very distributed network, with constant chatter between the hosts and master.

In addition to scheduling, Kubernetes helps with replication, checking the status of containers and hosts and updating images by gradually taking replicas out of use and re-starting with the new image. We can scale an application by (manually) changing the number of replicas. If a machine dies, Kubernetes will restart the containers on a different host, and ensures that the network implications of this change are hidden. This recovery process is slow. The working assumption of Kubernetes is that a high availability service will have many replicas, so that in case of isolated failures the service remains alive, just slightly under-resourced.

Many network functions are stateful. Whilst stateful applications are possible, this is not a primary assumption for Kubernetes. The basic instructions for stateful applications assume no replicas.

2.2.2.4 Kubernetes Networking

Kubernetes relies on additional tools to implement the required networking. The network model is still considered to be weak[4], but this is an area that is being actively developed. Recent communications suggest that many of the issues will be resolved over the next six months.

It starts with the assumption that all containers and hosts in a cluster will be able to communicate with each other without the need for any NAT, although tunnelling is common. This can be achieved by allocating a routable IP address per pod (e.g. using private IPv4 addresses)[5]. The network plugins may also support policy control to e.g. prevent some pods talking to other pods. The status of IPv6 Kubernetes is unclear[6], but solid IPv6 might make a clean network solution possible.

These addresses change if the pod moves host, and multiple pods delivering the same service will have different IP addresses. To ensure that a service remains ***reachable,*** Kubernetes assumes that there will be a single address –the service_cluster_ip - that will always be associated with the service. Any item in the cluster can access the service using this IP address. The implementation of the service cluster IP address (i.e. mapping from service_cluster_ip address to the possible Pod

[4] http://www.devoperandi.com/load-balancing-in-kubernetes/
[5] Running Flannel with the "Backend" in the flannel.yaml file set to "host-gw" and checking the routing tables of all devices on the same subnet can avoid tunnels if all hosts are on a common physical subnet.
[6] https://github.com/kubernetes/kubernetes/blob/master/docs/design/networking.md

addresses) is often described in relation to a cloud load balancer, such that this address is an address/port pair from the load balancer which will balance traffic between the available service endpoints (i.e. between the different IP addresses of the replicated pods). With our private cloud, we use a "NodePort" address[7]. This maps the same specific port on every host to pods implementing the service using iptables and the kube-proxy which implements (to the best of our understanding) a TCP-split proxy service. The service can then be reached via any host IP address and the service specific port. If the host IP addresses are public, this gives external reachability to the service.

Tests show that NAT and tunnelling can be avoided, but we noticed packet duplications and retransmissions. It appears that this is a known problem[8]. Again, this is simply evidence of immaturity of the solution currently.

2.2.2.5 Kubernetes Conclusions

Overall we could say that Kubernetes is fragile (many component parts), complex (many parts with poor documentation) and immature. However it is one of the more mature and useful container management and scheduler systems currently available.

Today, Kubernetes would be useful in situations where we might have a homogeneous set of co-located hardware resources on which we want to run reliable services that only require single public interfaces with rapid scale out. Examples are therefore found in control plane services, for example within mobile networks.

The Kubernetes community is pro-active, and we have begun interaction with them to drive changes. We understand that they expect to support i.e. multiple public network interfaces, IPv6, different addressing models and multicast in the near term. This would greatly improve Kubernetes for NFV applications.

3 Performance evaluation and tuning of Virtual Infrastructure Managers (VIMs)

The proposed general model of the VM instantiation process is shown in Figure 1. We decompose the operations among the VIM core components, the VIM local components and the Compute resource/hypervisor. The VIM core components are responsible for receiving the VNF instantiation requests (i.e. the initial request in Figure 1) and for choosing the resources to use, i.e. the

[7] It is not clear if we use the kube-proxy split TCP connection if there is a kuberentes load balancer available
[8] https://github.com/kubernetes/kubernetes/issues/27489

scheduling. This decision is translated into a set of requests which are sent to the VIM local components. These are located near the resources and are responsible for enforcing the decisions of the VIM core components by mapping the received requests to the corresponding hypervisor technology API calls. These APIs are typically wrapped by a driver, which is responsible for instructing the Compute resource to instantiate and boot the requested VNFs.

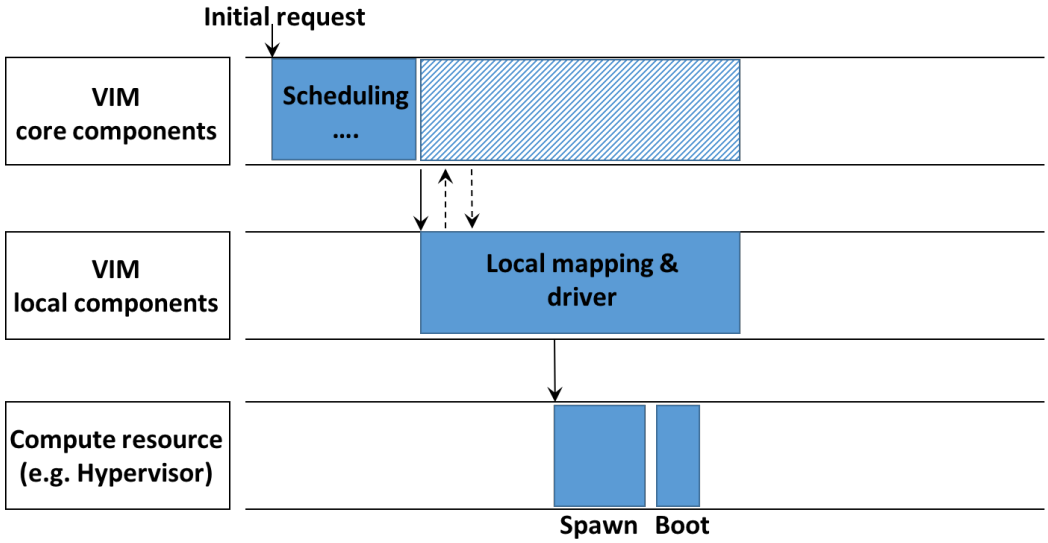

Figure 1: VIM instantiation general model

Figure 2 shows how the Nova, Nomad and OpenVIM components can be mapped on the proposed reference model. In the next subsections, we provide a detailed analysis of the operations of the three VIMs. As for the Compute resource/hypervisor, in this work we focus on Xen [1], an open-source hypervisor commonly used in production clouds, as the resource manager. Xen can be configured to use different toolstacks (tools to manage guest creation, destruction and configuration). A toolstack can interact with the hypervisor directly or using a toolstack API. The toolstack API provided by Xen is called libxl and is adopted by the majority of Xen compatible toolstacks.

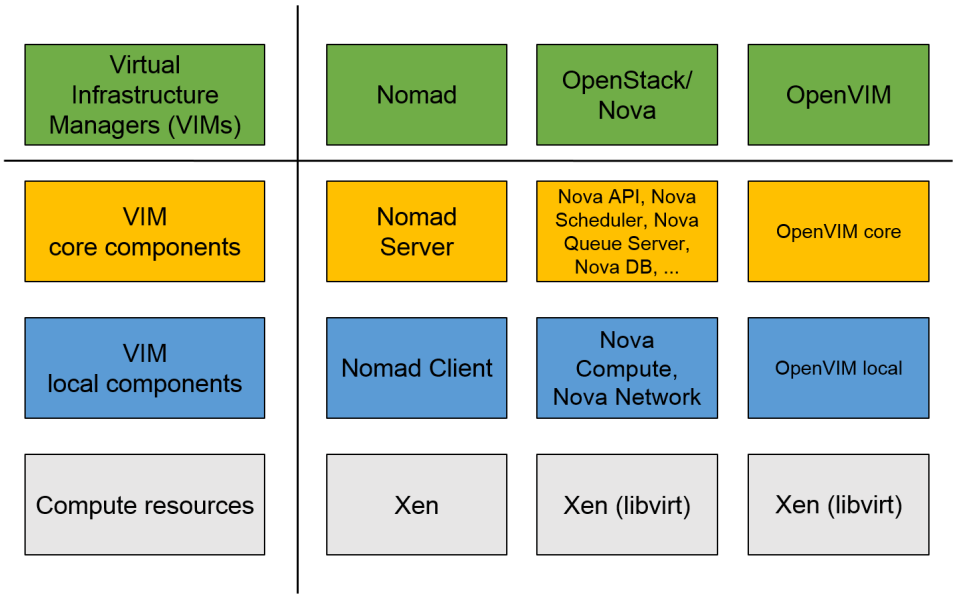

Figure 2: Mapping the reference model onto the considered orchestrators

3.1 Modelling OpenStack Nova

Figure 3 shows the scheduling and instantiation process for OpenStack Nova. The requests are submitted to the Nova API using the HTTP protocol (REST API). The Nova API component manages the Initial requests and stores them in the Queue Server. At this point, an authentication phase towards Keystone is required. The next step is the retrieval of the image from Glance, which is required for the creation of virtual resources. At the completion of this step, the Nova Scheduler is involved: this component performs scheduling tasks by taking the requests from the Queue Server, deciding the Compute nodes where the guests should be deployed and sending back its decision to the Nova API (passing through the Queue Server). The components described so far are mapped to the VIM core components, in our model. After receiving the scheduling decision from the Nova Scheduler, the Nova API contacts the Nova Compute node. This component manages the interaction with the specific hypervisors using the proper toolstack and can be mapped (along with Nova Network) to the VIM local components. The VM instantiation phase can be divided in two sub-steps: Network creation and Spawning. Once this task is finished, Nova Compute sends all the necessary information to Libvirt, which manages the spawning process instructing the Xen hypervisor to boot the virtual machine. When the completion of the boot process is confirmed, Nova Compute sends a notification and the Nova API confirms the availability of the new VM. At this point, the machine is ready and started. The above description, reflected in Figure 3, is a simplified view of the actual process: for sake of clarity many details have been omitted. For example, the messages exchanged between the components traverse the messaging system (Nova

Queue Server, which is not shown), and at each step the system state is serialized in the Nova DB (not shown).

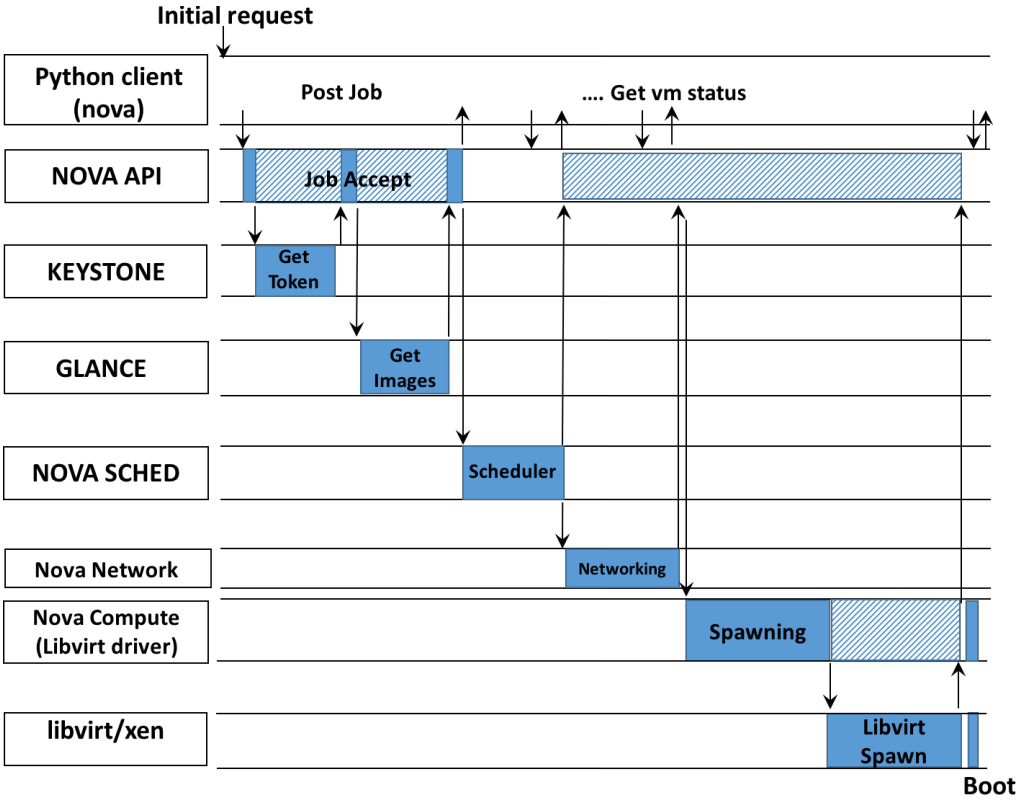

Figure 3: VIM instantiation model for OpenStack Nova

3.2 Modelling Nomad

The scheduling and instantiation process for Nomad is shown in Figure 4. According to our model, the Nomad Server is mapped into the VIM core components. It receives the requests for the instantiation of VMs (jobs) through the REST API. Once the job has been accepted and validated, the Server takes the scheduling decision and selects in which Nomad Client node to run the VM. The Server contacts the Client sending an array of job IDs. As a response, the Client provides a subset of IDs which are the ones that will likely be executed in this transaction. The Server acknowledges the IDs and the Client executes these jobs. The Nomad Client is mapped to the VIM local components and interacts with compute resources/hypervisors. We used XL, the default Xen toolstack, to interface with the local Xen hypervisor. XL is built using libxl and provides a command line interface for guest creation and management. The Client executes these jobs loading the Nomad Xen driver, which takes care of the job execution interacting with the XL toolstack. The instantiation process takes place and once completed the Client notifies the Server about its conclusion. Meanwhile the boot process of the VM starts and continues asynchronously with respect to the Nomad Client.

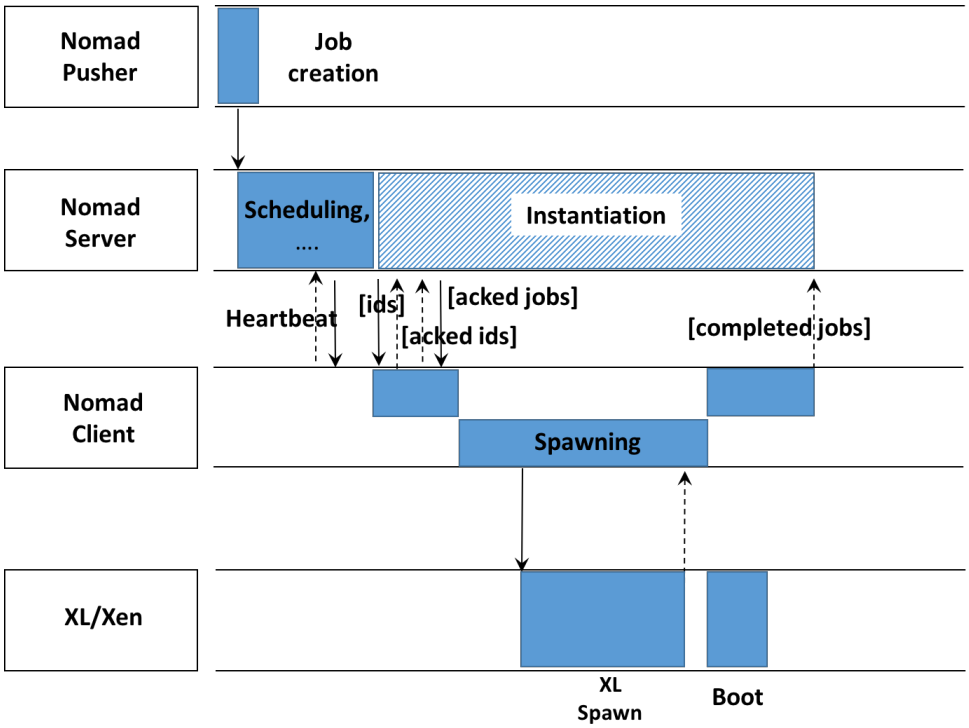

Figure 4: VIM instantiation model for Nomad

3.3 Modelling OpenVIM

In Figure 5, we depict the instantiation process for OpenVIM. The openvim CLI client submits an instantiation job request to the OpenVIM core. Then the flavors and the image information related to the job request are retrieved by the OpenVIM core from the OpenVIM Database. Next, the information on the current status of resources is retrieved by the OpenVIM core from the OpenVIM Database and a decision is taken by the scheduler. The scheduling decision information, which includes the compute node chosen for the instantiation, is updated in the OpenVIM Database and an HTTP 200 OK status message is sent back to the openvim client. The scheduling information in the OpenVIM database is then read by a thread on the OpenVIM core associated to the chosen compute node. The thread starts the spawning process by generating an XML description of the instance and submitting it to the OpenVIM local's libvirt daemon running on the chosen compute node. The libvirt daemon creates the instance based on the received XML. Then the associated thread on the OpenVIM core calls again the libvirt daemon to instruct it to start the previously-created instance. Finally, the libvirt daemon boots the instance through the Xen Hypervisor and the information on the started instance is updated in the OpenVIM database.

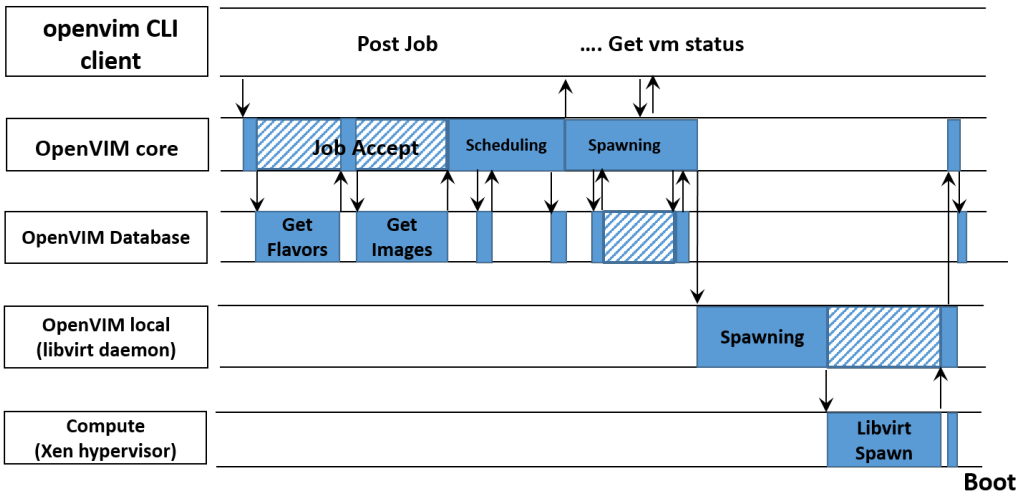

Figure 5: VIM Instantiation model for OpenVIM

3.4 Experimental results

In order to evaluate the VIM performances in the VM scheduling and instantiation phase, we combined different sources of information. We analysed the message exchanges in order to obtain a coarse information about the beginning and the end of the different phases of the VM instantiation process. The analysis of messages is a convenient approach as it does not require to understand and modify the source code. We have developed a VIM Message Analyser tool with a Python script and the Scapy [2] library. The VIM Message Analyzer (available at [3]) is capable of analyzing Nova and Nomad message exchanges. For a more detailed breakdown of the timings for specific components or phases, we inserted timestamp logging instructions inside the code of the Nomad Client, Nova Compute nodes as well as the OpenVIM core. We have generated the workload for OpenStack using Rally [4], a well known benchmarking tool. For the generation of the Nomad workload, instead, we have developed the Nomad Pusher tool. It is a utility written in the GO language, which can be employed to programmatically submit jobs to the Nomad Server. For OpenVIM we have simply scripted the OpenVIM command line client operations.

We executed experiments to evaluate the performance of the considered VIMs. We present here two main results: i) the total time needed to instantiate a ClickOS [5] VM (representing a Micro-VNF); ii) the timing breakdown of the Spawning process in Nova and of the Driver execution in Nomad. The first result is based on the VIM Message Analyzer we have developed. The timing breakdown is obtained with the approach of inserting timestamp loggers in the source code of the VIMs. All results have been obtained by executing a test of 100 replicated runs, in unloaded conditions. Error bars in the figures denote the 95% confidence intervals of the results. In each run, we requested the VIM to instantiate a new ClickOS VM. The VM is deleted before the start of the next run. Our experimental set-up is composed by two hosts with an Intel Xeon 3.40GHz quad-core


CPU and 16GB of RAM. One host (hereafter referred to as HostC) is used for the VIM core components, the other host (HostL) for the VIM Local components and the Compute resource. We are using Debian 8.3 operating systems with Xen-enabled v3.16.7 Linux kernels. Both hosts are equipped with two network interfaces at 10 Gb/s: one interface is used as the management interface and the other one for the direct interconnection of the host (data plane network). In order to emulate a third host running OpenStack Rally and Nomad Pusher, we created a separated network namespace using the iproute2 suite in the HostC, then we interconnected this namespace to the data plane network.

3.4.1 OpenStack Nova Experimental results

For what concerns OpenStack we run Keystone, Glance, Nova orchestrator, Horizon, and the other components in HostC, while we run Nova Compute and Nova Network in HostL.

With reference to Figure 3, we report in Figure 6 the measurements of the instantiation process in OpenStack, separated for each component. The topmost horizontal bar (Stock) refers to the OpenStack version that only includes the modifications to boot the ClickOS VMs. The experiment reports a total time exceeding two seconds which is not adequate for highly dynamic NFV scenarios. Analysing the timing of the execution of the single components, we note that most of the time is spent during the spawning phase while the other components account for around 0.5 seconds. In Figure 7 (topmost horizontal bar), we show the details of the spawning phase which is split in three phases: 1) Create image, 2) Generate XML and 3) Libvirt spawn. The first two are executed by the Nova Libvirt driver and the last one is executed by the underlying Libvirt layer. The Nova Libvirt driver also executes some network configuration steps which are not shown, as they happen in parallel with the Create image phase, which always terminates after the network configuration has finished. By analyzing the timing of the stock version, we can see that the Create image is the slowest step with a duration of about 1 second. This step includes operations like the creation of log files and the creation of the folders to store Glance images. The original OS image (the one retrieved from Glance) is re-sized in order to meet user requirements (the so called flavors in OpenStack's jargon). Moreover, if it is required, the swap memory or the ephemeral storage are created and finally some configuration parameters are injected into the image (e.g. SSH key-pair, network interface configuration).

The Generate XML and Libvirt spawn steps introduce a total delay of 0.4 seconds, but optimizing these steps is non-trivial as all the performed operations cannot be skipped or downsized. Indeed, during the Generate XML step, the configuration options of the guest domain are retrieved and then used to build the guest domain description (the XML file given in input to Libvirt). For instance, options like the number of CPUs and CPU pinning are inserted in the XML file. Once this step is over,

the libxl API is invoked in order to boot the VM. When the API returns, the VM is considered spawned, terminating the instantiation process. In this test we are not considering the whole boot time of the VM, as this is independent from the VIM operations, and thus out of the scope of this work. The considered spawning time measures only the time needed to create the Xen guest domain.

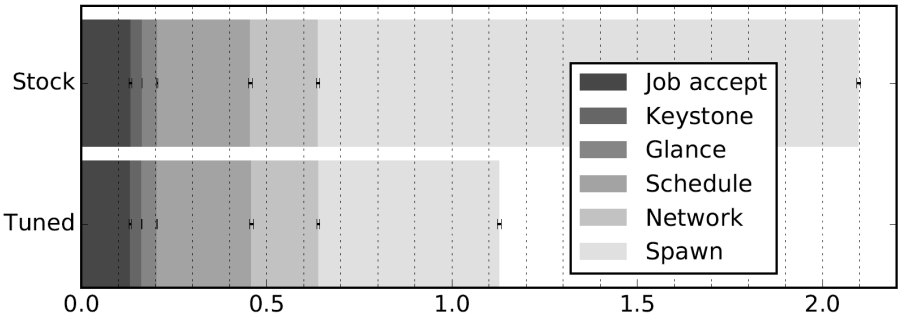

Figure 6: ClickOS (micro-NFV) instantiation time breakdown on OpenStack

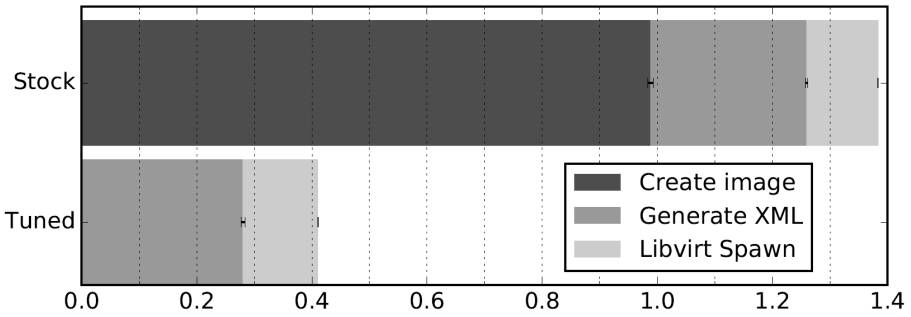

Figure 7: ClickOS spawning time breakdown on Nova-compute

3.4.2 Nomad Experimental results

According to the Nomad architecture, the minimal deployment includes two nodes. Therefore, we have deployed a Nomad Server, which performs the scheduling tasks, in HostC and a Nomad Client, which is responsible for the instantiation of virtual machines, in HostL. The Nomad Pusher runs in HostC but in a different network namespace. We identified two major steps in the breakdown of the instantiation process: Scheduling and Instantiation. The topmost horizontal bar in Figure 8 reports the results of the performance evaluation for the stock Nomad. The total instantiation time is much lower than the one obtained for OpenStack. This result is not surprising: Nomad is a minimalistic VIM providing only what is strictly needed to schedule the execution of virtual resources and to instantiate them. Looking at the details, the Scheduling process is very light-weight, with a total run time of about 50 ms. The biggest component in the instantiation time is the spawning process which is executed by the XenDriver. Diving in the driver operations, we identified 4 major steps: Download artifact, Init Environment, Spawn, and Clean, as reported in Figure 9. In the first step, Nomad tries to download the artifacts specified by the job. For a Xen job, the Client is required to download the configuration file describing the guest and the image to load. This part

adds a delay of about 40 ms and can be optimized or entirely skipped. Init Environment and Clean introduce a low delay (around 20 ms) and are interrelated: the former initializes data structures, creates log files and folders for the Command executor, the latter cleans up the data structures once the command execution is finished. The XL spawn is the step which takes longer but by studying the source code we found no room to implement further optimizations: indeed the total spawning measured time is around 100 ms. Considering a light overhead introduced by the Command executor the time is very similar to the what we obtain by running directly the XL toolstack. The overall duration of the spawning phase is 160 ms, lower that the 280 ms for the instantiation phase reported in Figure 8. This is due to the notification mechanism from the client towards the server. It uses a lazy approach for communicating the end of the scheduled jobs: when the message is ready to be sent, the client waits for a timer expiration to attempt to aggregate more notifications in a single message. This means that the instantiation time with Nomad is actually shorter than the one shown in Figure 8.

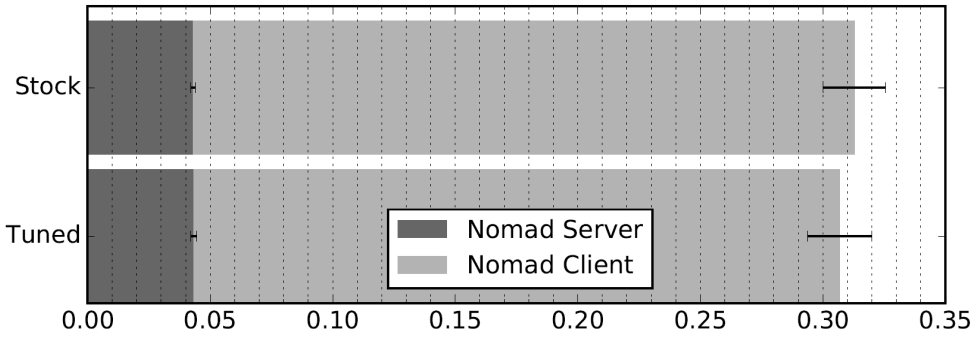

Figure 8: ClickOS: (micro-NFV) instantiation time breakdown on Nomad

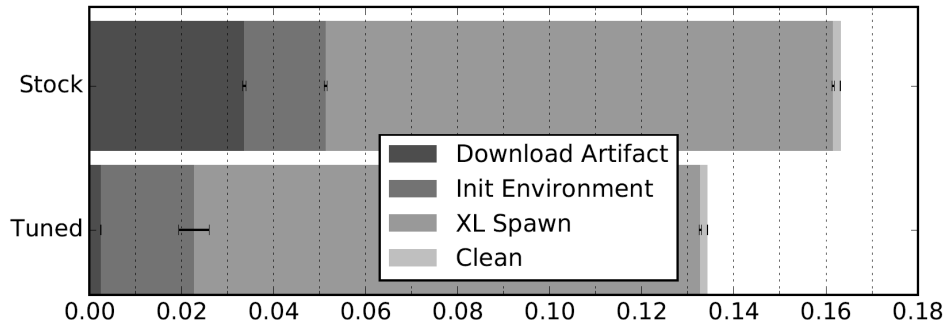

Figure 9: ClickOS spawning time breakdown on Nomad

3.4.3 OpenVIM Experimental Results (Work in Progress)

In this section we present some experimental results for the performance evaluation of OpenVIM. The information is derived from the analysis of the OpenVIM, libvirt and Xen logs and from the insertion of timestamp logging instructions. The openvim command line client and the OpenVIM core run on HostC, while on HostL we have installed Xen, the libvirt daemon and the OpenSSH server, configured to allow the remote issuing of commands by OpenVIM core.

Figure 10 reports our current results. We call OpenVIM "Stock" the version of OpenVIM patched only to support the instantiation of ClickOS Unikernel VMs, while we call "Tuned" our OpenVIM deployment with a set of Unikernel-oriented optimizations. With reference to the model depicted in Figure 5, the REST phase corresponds to the "Job Accept" phase on the OpenVIM core, while the Scheduling and Spawning phases correspond to the phases with the same names on both figures. The results show that the time used in the REST and Scheduling phases remain substantially unchanged after our optimizations. For what concerns the spawning phase, this takes the greatest amount of time in the "Stock" version of OpenVIM, but we are able to reduce it substantially in the "Tuned" version by removing the code which is not strictly needed for the instantiation of Unikernel VMs. The performance tuning of OpenVIM is still work in progress: we are investigating if we can further reduce the instantiation times, especially in the Scheduling phase, to approach the ones obtained with Nomad. We also plan to provide a breakdown of the spawning time for OpenVIM, as we have done for the other considered orchestrators.

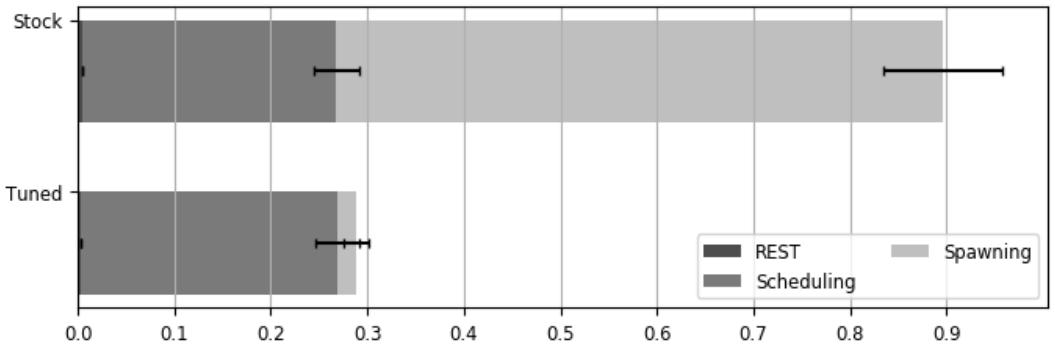

Figure 10: ClickOS (micro-NFV) instantiation time breakdown on OpenVIM

4 Multi-access Edge Computing (MEC)

4.1 Introduction to MEC

Multi-access Edge Computing (MEC), which formerly stood for Mobile Edge Computing, offers application developers and content providers cloud-computing capabilities and an IT service environment at the edge of the mobile network. This environment is characterized by ultra-low latency and high bandwidth, as well as real-time access to services that can be leveraged by applications. Examples of these services are, radio network information or location.

MEC provides a new ecosystem and value chain. Operators can open their networks edges, e.g. Radio Access Network (RAN), to authorized third-parties (e.g. app providers), allowing them to rapidly deploy innovative applications and services towards the subscribers, enterprises and vertical segments.

4.2 Overall Architecture of MEC

Originally, MEC was the natural development in the evolution of mobile base stations and the convergence of IT and telecommunications networking. Multi-access Edge Computing will leverage new vertical business segments and services for consumers and enterprise customers.

MEC Use cases include, among others:

- Video analytics
- Location services
- Internet-of-Things (IoT)
- Augmented reality
- Optimized local content distribution and
- Data caching

It uniquely allows software applications to tap into local content and real-time information about local-access network conditions. By deploying various services and caching content at the network edge, core networks are alleviated of further congestion and can efficiently serve local purposes.

New MEC industry standards (coming from ETSI ISG MEC) and deployment of MEC platforms will act as enablers for new revenue streams to operators, vendors and third-parties. Differentiation will be enabled through the unique applications deployed in the Edge Cloud. Figure 11, below, shows the MECC architecture as defined by the ETSI MEC (MEC-Arch-003].

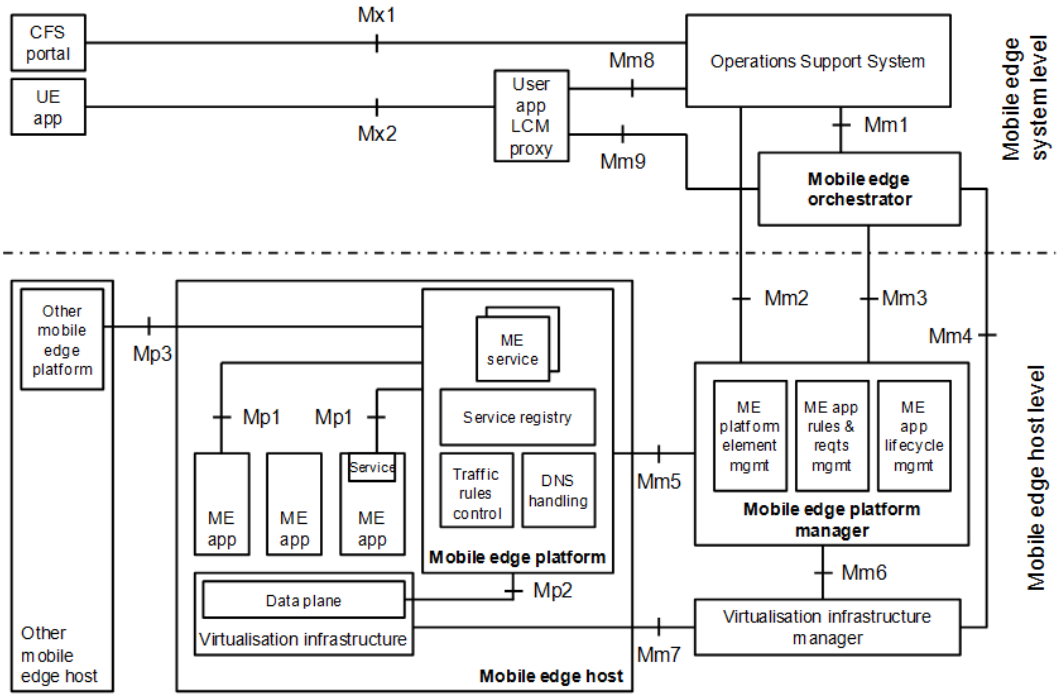

Figure 11: ETSI MEC: architecture reference model [ETSI-MEC].

Figure 12, below, depicts the implementation view of the MEC architecture blocks for implementation.

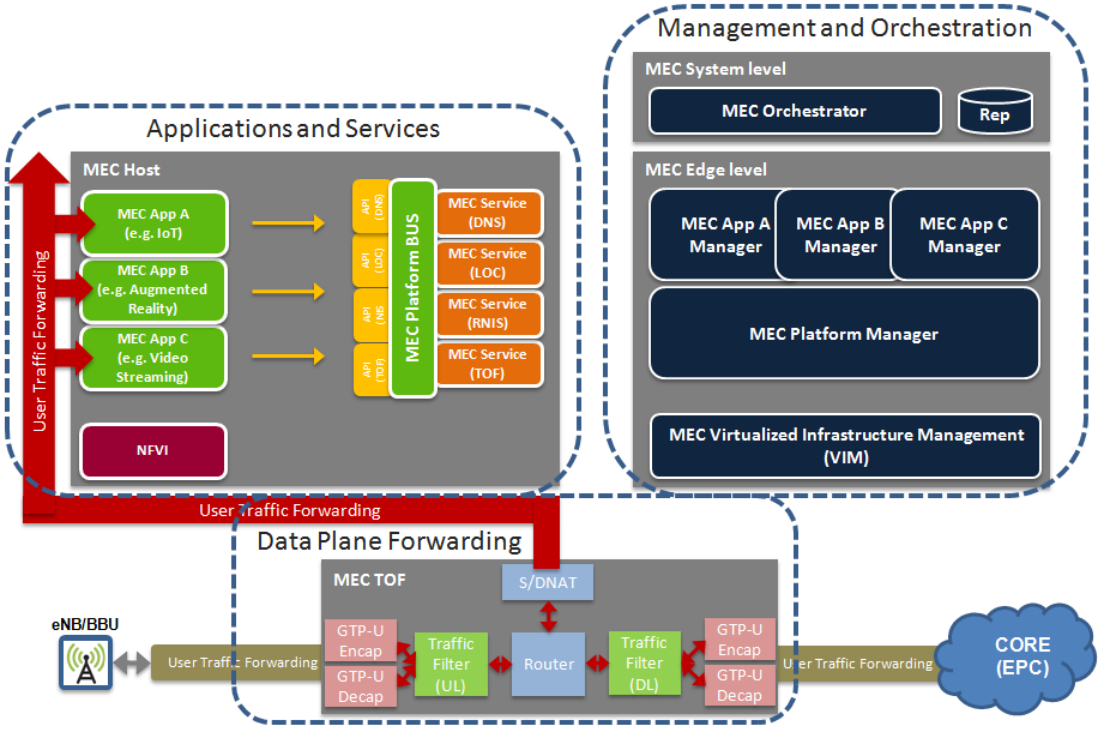

Figure 12: ETSI MEC: main architectural blocks.

In this picture, it can be identified 3 main building blocks: TOF, ME Host and ME Management and Orchestration (MANO).

4.2.1 Multi-edge Traffic Offloading Function (ME TOF)

The high level architecture of the ME TOF [9]block is depicted in Figure 13, below.

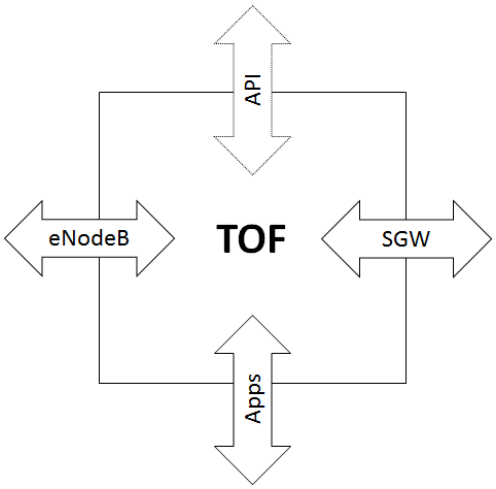

Figure 13: ME Traffic Offloading Function Block (ME TOF)

The TOF has three data-plane interfaces (eNodeB, SGW and Apps) and a single control interface (API). The data-plane carries user-data to/from the respective elements it connects. The eNodeB is the radio access, the SGW is a data plane EPC element, and Apps are running at the edge cloud and require access to the data plane.

The data-plane interfaces should be seen as network adapters. Although there are three interfaces in the data-plane, the two that connect to the eNodeB and SGW can (with some caveats, as described later in this document) be merged into a single network adapter that connects to the S1 interface.

The API interface is a REST service and is used to provision the TOF. This interface can bound to a particular network adapter (management), but can also use any other interface at the host. This allows for greater flexibility on how one chooses to perform the networking to reach the API, however this also means there is the need to externally harden the access to the API during the host's configuration (there are major security implications involved, do not overlook this step).

[9] §5.1.1.3 - https://portal.etsi.org/portals/0/tbpages/mec/docs/mobile-edge_computing_-_introductory_technical_white_paper_v1%2018-09-14.pdf

4.2.2 ME Host

The high level architecture of the ME Host block is depicted in Figure 14.

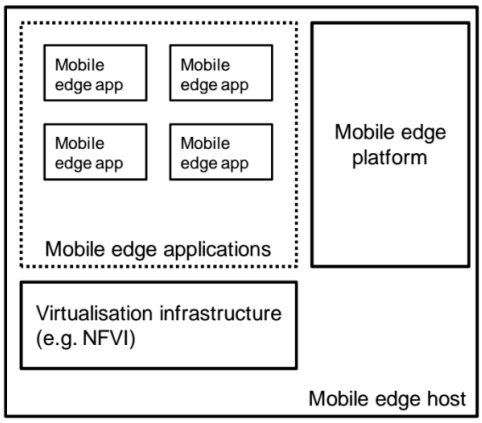

Figure 14: ME Host Block

The ME Host is composed by three main components: virtualization infrastructure (cloud), mobile edge platform and mobile edge applications.

The virtualization infrastructure is an NFVI-like infrastructure, which provides compute, storage, and network resources, for the purpose of running mobile edge applications (ME Apps).

The mobile edge platform is the collection of essential functionality required to enable mobile edge applications to consume and provide mobile edge services. The mobile edge platform includes also basic/standard services.

Mobile edge applications are instantiated on the virtualization infrastructure of the mobile edge host based on configuration or requests validated by the mobile edge management (ME MANO).

4.3 ME Traffic Offloading Function

The *Traffic Offloading Function* (TOF) will intercept traffic in the S1 interface, between the RAN (Radio Access Network – eNB) and the EPC (*Enhanced Packet Core*'s – SGW). Upon the services are provisioned, it will either divert the traffic to a ME App or simply forward it to the EPC as usual. In order to do so, the TOF must have connectivity to the S1 interface through a configured network adapter(s).

The Edge (Apps) interface and the connection to the network must use different network adapters, ideally with a dedicated adapter to the eNodeB and another for the SGW. If this adapter is shared, there is no possible to auto-detect which is the list of eNodeBs and SGWs (therefore their IPs must be pre-provisioned in configuration files).

4.3.1 TOF Capabilities Summary

The TOF key capabilities are as follows:

- Encapsulation and de-encapsulation of GTP-U users traffic (in Kernel Space).
- Auto-discovery of the TEIDs (Tunnel Endpoint IDentifier) from the GTP-U traffic.
- Learn the TEID and establish the SGW/eNodeB tunnels even when the UE traffic never reaches the EPC (use of the *Pinger* function for this purpose).
- Forwarding of the traffic to the ME App (iptables built-in NAT function).
- Support for multi SGWs/eNodeBs with any TEID.
- Work as regular switch for non-GTP-U traffic, in case of separated eNodeBs/SGWs interfaces
- Auto-purge stale SGW/eNodeB tunnels after some configured idle time.

4.3.2 TOF Internal Modules

This section describes the Internal Modules of MEC TOF component. The detailed architecture is presented in Figure 15.

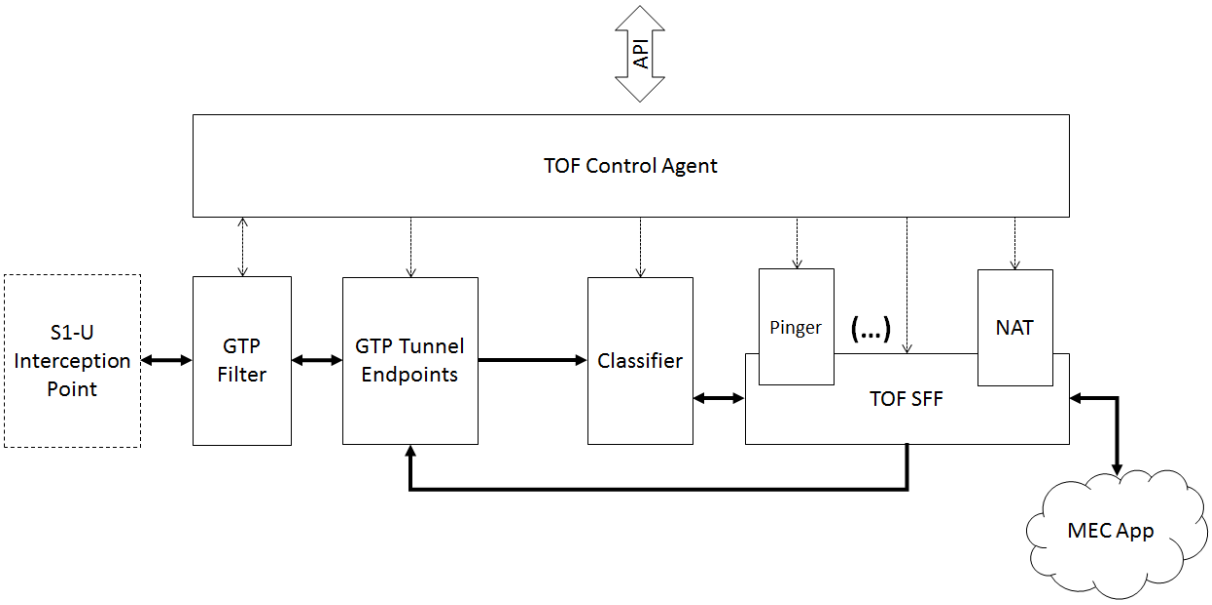

Figure 15: Traffic Offloading Function (TOF) Architecture

The TOF component comprises two large groups of components: the control-plane (the TOF Control Agent) and the data-plane (everything else). Within the data-plane, there is an additional distinction between components, the ones that deal with GTP-encapsulated flows and the others that deal with already de-encapsulated UE-flows (clean user traffic).

4.3.2.1 GTP Encapsulated Data-plane

Within this type of modules, all flows are encapsulated within GTP-U. For those, the traditional flow identification/control (5-tuple or OpenFlow) considers the mobile network endpoints (eNodeB/SGW).

S1-U Interception Point

The *S1-U Interception Point* isn't actually a component of the TOF, but rather a representation of the network adapter(s) that deliver the user's traffic from the network to the TOF.

4.3.2.1.1 GTP Filter

The *GTP Filter* determines whether or not the GTP-U traffic must be delivered to the control-plane (happens when a new tunnel is spotted), which in turn will perform all the required *GTP Tunnel Endpoints* configuration.

A new tunnel is spotted anytime an unknown TEID arrives for a given set of IP endpoints, meaning that there is a new user session. The way this detection is performed follows a reverse logic, in which, by default, the packet goes to the control-plane unless a matching rule of TEID + endpoints matches that packet.

Because flows will only pertain to the outer tunnel endpoints, one must deeply inspect the packets to retrieve the TEID. Therefore the GTP Filter needs to process traffic packet by packet.

All filtering process is done with iptables/netfilter rules. Packets which need to be processed in the control-plane are passed to the control application using NFQUEUES, being each queue dedicated to a single purpose (for instance, all packets that come from an eNodeB go into queue 0, while all packets that come from a SGW go into queue 1).

4.3.2.1.2 GTP Tunnel Endpoints

The *GTP Tunnel Endpoints* performs the GTP-U encapsulation/decapsulation of traffic that comes from / goes to the mobile network, as configured by the TOF Agent/Controller when receiving GTP-U packets from the GTP Filter.

A patched OpenvSwitch (OVS) with support for GTP-U (in kernel space) is used, along with NATing capabilities (*netfilter* for statefull translations and *tc* for stateless translations) to change the destination/source addresses as needed so that the OVS acts as the endpoint of existing S1 traffic (rules are automatically inserted by the Control Agent).

4.3.2.2 UE Decapsulated Data-plane

Since the UE traffic is already decapsulated, it is possible to control these flows using OpenFlow.

4.3.2.2.1 UE Decapsulated Data-plane Classifier

The *Classifier* determines which path that packets from a particular flow must traverse, just as a SFC (Service Function Chaining) in the IETF chaining model. Because our SFC is built of port-chains, the way the Classifier reports its classification is by simply outputting to different ports, which are reserved for these criteria.

Port-chains are created using OpenFlow rules, as well as flows/packets are classified with OpenFlow.

4.3.2.2.2 UE Decapsulated Data-plane Service Function Forwarder (SFF)

4.3.2.2.3 The *TOF SFF* is in charge of forwarding packets according to the classification criteria set by the *Classifier*. In this case, this means the SFF will ensure the traffic coming from a given Classifier's output port will go to the next port of that chain and so on.

Given the current feature-set (VNFs) and the use-cases in which these are employed, the port-chaining does not need any other criteria than just which port that packet/flow came from. However, future scenarios may demand a reengineering which either replicates the Classifier's rules across the port-chain, or add a way to this classification, such as a TOS mark, VLAN or the full-blown NSH (Network Services Header).

4.3.2.2.4 UE Decapsulated Data-plane Functions (VNFs)

In this section are the functions that, although are used to perform actions within the TOF, they could (conceptually) be reused outside the TOF. There are two major features being showcased here, (1) the TOF can act as a SFF, (2) the TOF could also reuse functions which are already built/are closed boxes/perhaps not even virtual.

Pinger

The *Pinger* is a simple function which permanently generates ICMP traffic towards the outside of the mobile network.

It is used to act on behalf of the UE so that a return tunnel from the SGW can always be established, regardless whether the UE ever sends packets to the SGW (i.e. even if nothing other than the MEC

app is used). The ICMP traffic causes the return of downstream traffic which allows the TEID downstream discovery.

OpenFlow is used to rewrite the source address (to match the UE's), with the addition of two virtual network adapters which are added to the port-chain (to fit in our SFF/SFC model).

NAT

NAT (Network Address Translation) performs the replacement of a destination's IP (provisioned through the API) by the respective ME App IP, so that traffic is routed to that App.

4.3.3 Internal Interfaces

The internal interfaces between the referred components are plain shell commands, which run inside the respective element's network namespace.

The GTP Filter and the NAT components are both configured using the *iptables* command. The GTP Tunnels Endpoints, Classifier and the SFF are configured using Open vSwitch's (OVS) OpenFlow commands. Lastly, the Pinger is not directly configured (the function is always generating ICMP requests to the target). However, OpenFlow is also used to rewrite the source address (to the one of the UE) and then connect the Pinger's output to the proper GTP Tunnel.

4.3.3.1 Internal Interfaces, flow diagrams

A detailed description of the interaction between the internal components is described in this section in the form of flow diagrams.

TEID detection and tunnel(s) setup

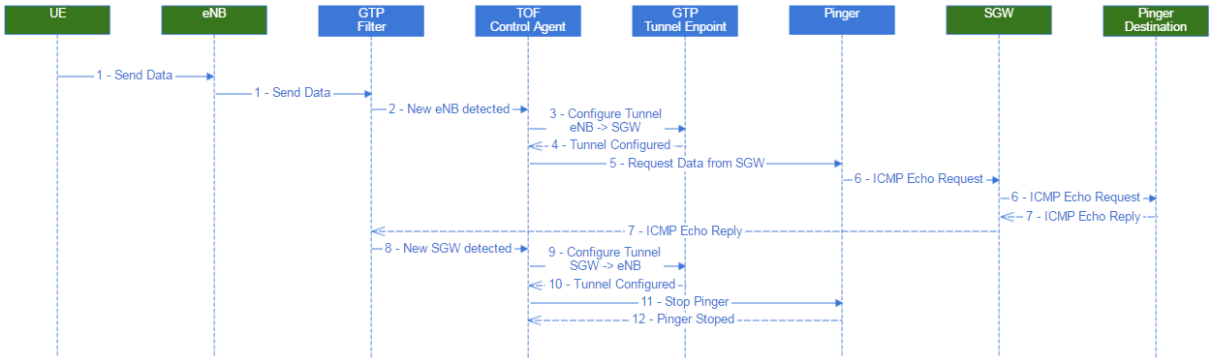

Figure 16: TEID Detection and Tunnel Setup Flow Diagram

Figure 16 depicts the TEID Detection and Tunnel Setup flow diagram. The steps are as follows:

1. The UE generates traffic to a Public Data Network (PDN) (e.g. Internet), which is forwarded to eNB. This traffic is caught at the S1 interface (between eNB and SGW) at the TOF, more precisely by the GTP Filter.
2. If the TEID from this eNB is from a new session, the TOF Control Agent is contacted.
3. The TOF Control Agent configures the GTP Tunnel between the eNB and the SGW, with the TEID present in GTP-U Header of packet.
4. The TOF Control Agent waits for the end of Tunnel Configuration.
5. To configure the tunnel from the SGW to the eNB, the TOF Control Agent configures the Pinger to generate traffic which will force a data packet from SGW.
6. The Pinger generates an ICMP Echo Request to a destination in thePDN (this traffic uses the GTP tunnel from TOF to SGW, configured in the step 3).
7. The Pinger replies and the response is caught by GTP Filter.
8. If the TEID from this SGW is a new session, the TOF Control Agent is contacted.
9. The TOF Control Agent configures the GTP Tunnel between SGW and eNB, with the TEID present in GTP-U Header of the packet.
10. The TOF Control Agent waits for the end of Tunnel Configuration.
11. The TOF Control Agent configures the Pinger to stop to generate ICMP traffic.
12. The TOF Control Agent waits for the end of Pinger configuration. The process is concluded.

Application registration

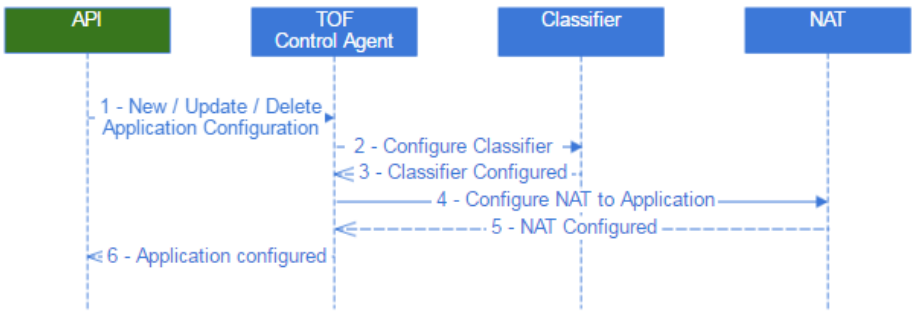

Figure 17: Application Registration Flow Diagram

Figure 17 depicts the Application Registration flow diagram. The steps are as follows:

1. Via API, it's possible to register a new application, update or delete the configurations.
2. The TOF Control Agent validates the request and, in case of correct use, configures the Classifier.
3. The TOF Control Agent waits for the end of configuration.
4. The TOF Control Agent configures the NAT to the MEC Application.
5. The TOF Control Agent waits for the end of configuration.

6. The TOF Control Agent replies to the API with success or the error code if unsuccessful.

4.3.4 TOF APIs

A detailed specification of the APIs is described in this section in the form of SWAGGER[10] format.

4.3.4.1 TOF Service API

This API configures the service application in TOF, and is referred in the previous section.

/Service

GET /service

Description

List all configured services

Responses

Code	Description	Schema

200	Services configured	[
		service {
		name: string *
		⇄ serviceIP: string *
		applicationIP:string *
		}
		]

POST /service

Description

Configure a new service

Parameters

Name	Located in	Description	Required	Schema
service	body	Service to	Yes	⇄newService {

[10] http://www.swagger.io

Name	Located in	Description	Required	Schema
		configure		name: string * serviceIP: string * applicationIP:string * }

Responses

Code	Description	Schema

200	Service configured	service {
		name: string *
		⇄ serviceIP: string *
		applicationIP:string *
		}

400	Bad Request	
409	Conflict	

/service/{name}

PUT /service/{name}

Description

Update a service

Parameters

Name	Located in	Description	Required	Schema
name	path	Name of service to update	Yes	⇄ string

service	body	Service to update	Yes	newService {
				name: string *
				⇄ serviceIP: string *
				applicationIP: string *
				}

Responses

Code	Description	Schema

200	Service updated	service {
		name: string *
		⇄ serviceIP: string *
		applicationIP: string *
		}

Code	Description	Schema
400	Bad Request	
404	Not found	
409	Conflict	

DELETE /service/{name}

Description

Deletes a service based on the name or service IP supplied

Parameters

Name	Located in	Description	Required	Schema
name	path	Name or service IP of service to delete	Yes	⇄ string

Responses

Code	Description
200	Service deleted
404	Not found

4.4 ME Host

The Mobile Edge Host is an edge-level entity which comprises the Mobile Edge Platform (MEP) and a Virtualization Infrastructure. The Virtualization Infrastructure provides compute, storage, and network resources for the Mobile Edge Applications (ME Apps). The MEP is responsible to provide standard services like DNS, network information or location to be consumed by ME Apps, and to manage the access (authentication and authorization) of those applications to the services. ME Apps can also produce services to be consumed by other ME Apps.

4.4.1 ME Host Internal Modules

This section describes the Internal Modules of the ME Host. The detailed architecture is presented in Figure 18 (according to ETSI ISG MEC Architecture document).

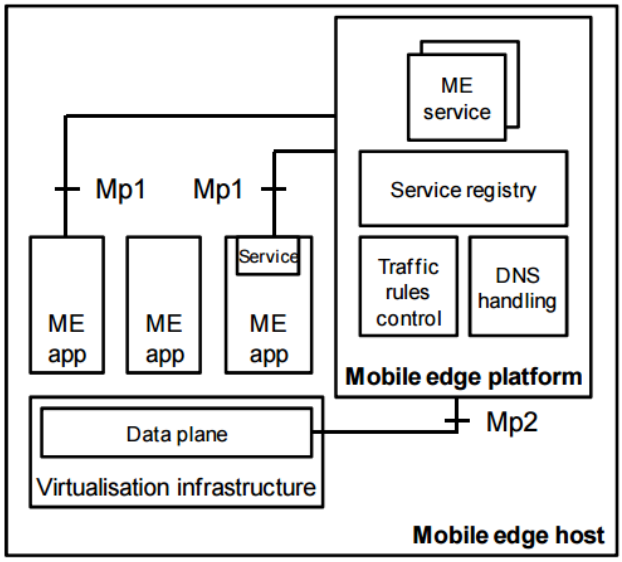

Figure 18: MEC Host Architecture

The ME Host is made of three large groups of components, Virtualization Infrastructure, Mobile Edge Platform and Mobile Edge Applications.

4.4.1.1 ME Host Virtualisation Infrastructure

The Virtualization Infrastructure of the ME Host uses the OpenStack as the NFVI/VIM, to support all the requirements of ME Host. Therefore, the ME Host acquires the specifications and capabilities of this NFVI/VIM.

4.4.1.1.1 ME Host Virtualisation Infrastructure Capabilities Summary

This Virtualization Infrastructure is capable of:

- Deploying/dispose Mobile Edge Applications.
- Manage the resources through Graphical User Interface (GUI).
- Routing between the ME Apps and the data plane.
- Isolation between ME Applications resources.
- Isolation between traffic towards different MEC applications.
- Access to Services.
- Multi-tenancy.

4.4.1.1.2 ME Host Virtualisation Infrastructure Internal Connectivity

The approach used to connect the applications to the TOF is presented in Figure 19.

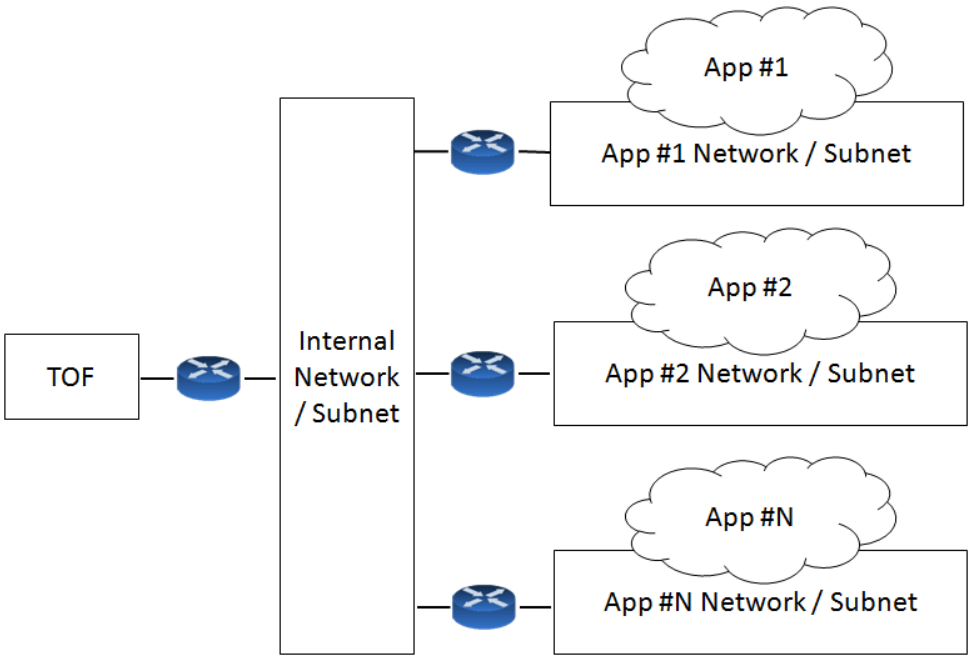

Figure 19: Connectivity between the TOF and ME Apps

In this approach, the ME Apps run in different tenants, as a way to isolate the resources. The same way, the network that connects ME Apps to the data-plane (via TOF) is also isolated. This traffic is isolated trough routers and an internal network marked as a public network, which possibility gives a floating IP from the internal network to the App's port. In this manner, the ME App is accessible by the TOF using this floating IP.

The approach used to connect ME Apps to the Service API is presented in Figure 20.

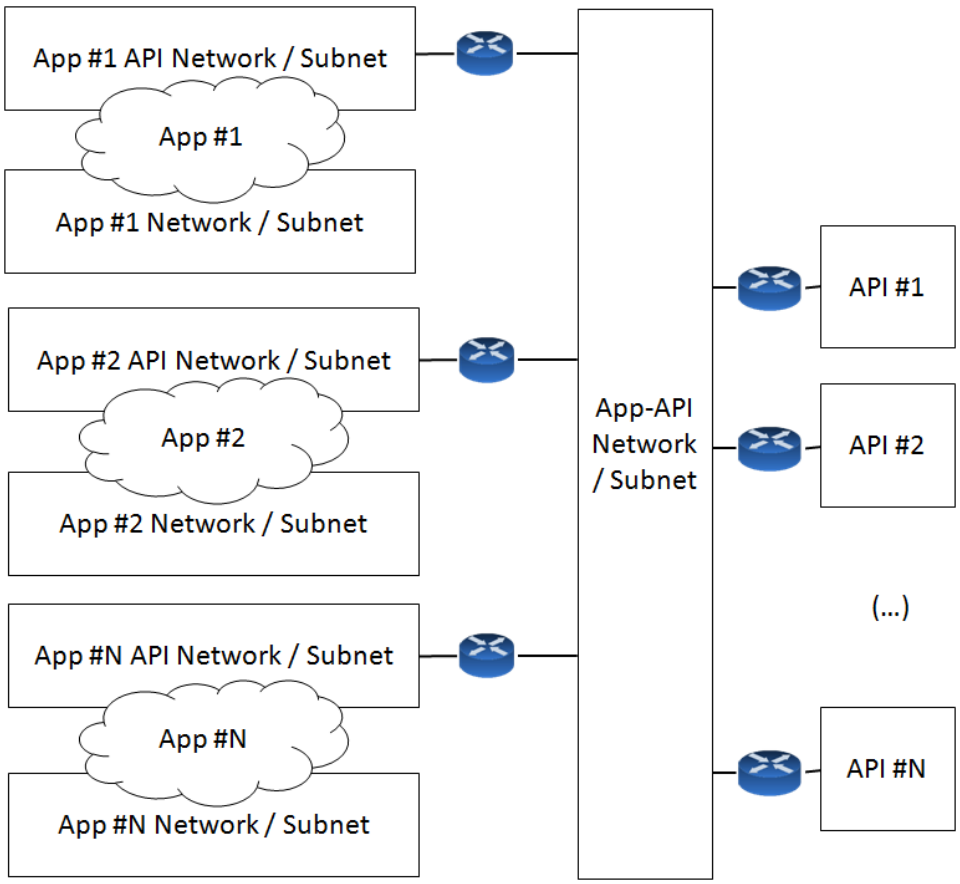

Figure 20: Connectivity between ME Apps and Services APIs

For this approach, the strategy used was the same as the one used to connect applications to the TOF, using a network marked as a public, and giving a floating IP to the applications to communicate with the MEC platform and with authorized APIs.

ME Apps are this way connected through 2 networks, one towards the TOF and another towards the ME Platform.

4.4.1.2 Mobile Edge Platform

The Mobile Edge Platform is capable of:

- Offering an environment where the Mobile Edge Applications can discover, advertise, consume and offer Mobile Edge services.
- Receiving traffic rules from the management layer, and instructing the data plane accordingly.
- Receiving DNS records from the management layer and instructing DNS server accordingly.
- Receive any service provisioning data and instructing this service accordingly.
- Hosting mobile edge services, in case services are virtualized.

4.4.1.3 Mobile Edge Applications

Mobile edge applications are running as virtual machines (VM) on top of the virtualization infrastructure provided by the Mobile Edge Host, and can access to the Mobile Edge Platform to consume and provide Mobile Edge Services. So far, only VM infrastructure is supported but other can be supported in the future.

To interact with Mobile Edge Platform, the ME Applications need to implement the interface Mp1, according to the specifications defined by the ETSI ISG MEC.

(Please note that the sections in yellow are incomplete and will be completed for the final deliverable).

4.4.2 Mobile Host, Internal Interfaces

Interfaces between internal modules

Regarding to the Virtualization Infrastructure, the NFVI are configured with your specific commands/APIs (the NFVI/VIM used was OpenStack).

4.4.3 Mobile Host Flow Diagrams

How it works, flows

4.4.4 Mobile Host Management APIs

Management aspects

Regarding to the Virtualization Infrastructure, the APIs are the specific APIs from the NFVI/VIM used (the NFVI used was OpenStack).

4.5 ME MANO

4.5.1 ME MANO Internal Modules

Internal modules that compose this block

4.5.2 ME MANO Internal Interfaces

Interfaces between internal modules

4.5.3 ME MANO Flow Diagrams

How it works, flows

4.5.4 ME MANO Management APIs etc

Management aspects


5 The Superfluid platform

5.1 Introduction to the Superfluid platform

Lightweight virtualization technologies such as Docker [6] and LXC [7] are gaining enormous traction not only in the research field but also in terms of real-world deployment. Google, for instance, is reported to run all of its services in containers [6], and Container as a Service (CaaS) products are available from a number of major players including Azure's Container Service [9], Amazon's EC2 Container Service and Lambda offerings [7] [8], and Google's Container Engine service [10].

Beyond these services, lightweight virtualization is crucial to a wide range of use cases, including just-in-time instantiation of services [12] (e.g., filters against DDoS attacks, TCP acceleration proxies, content caches, etc.) and NFV [14] [15], all the while providing significant cost reduction through consolidation and power minimization [15].

The reasons for containers to have taken the virtualization market by storm are clear. In contrast to heavy-weight, hypervisor-based technologies such as VMWare, KVM or Xen, they provide extremely fast instantiation times, small per-instance memory footprints, and high density on a single host, among other features.

However, no technology is perfect, and containers are no exception. Security, for one, has been and continues to be a thorn on their side. First, their large trusted computing base (TCB), at least compared to type-1 hypervisors, has resulted in a large number exploits [11] [18]. Second, a container that causes a kernel panic will bring down the entire host. Further, any container that can monopolize or exhaust system resources (e.g., memory, file descriptors, user IDs, forkbombs, etc.) will cause a DoS attack on all other containers on that host [15] [11]. Over the years, a significant amount of effort has resulted in the introduction of mechanisms such as user namespaces and Seccomp that harden or eliminate a large number of these attack vectors. However, a simple misconfiguration can still lead to an insecure system [11].

Beyond security, another downside of containers is that their sharing of the same kernel rules out the possibility to specialize the kernel and its network stack to provide better functionality and performance to specific applications [16]. Finally, containers do not currently support live migration, although support for it is under development [10] [8].

At least for multi-tenant deployments, this leaves us with a difficult choice between (1) containers and the security issues surrounding them and (2) the burden coming from heavyweight, VM-based platforms. Clearly, we cannot easily, overnight, fix all of the security issues related to containers, nor prevent new ones from arising (This is also true of hypervisors, but their smaller TCB and more mature code base should result in fewer exploits). Could we perhaps re-architect existing hypervisor technologies to provide lightweight virtualization on top of them? The explicit goal would be to achieve numbers in the same ball-park as containers: instantiation in milliseconds,


instance memory footprints of a few MBs or less, and the ability to concurrently run one thousand or more instances on a single host.

In this section, we introduce LightVM, a *lightweight* virtualization system based on a type-1 hypervisor. LightVM retains the strong isolation virtual machines are well-known for while providing the performance characteristics that make containers such an attractive proposition. In particular, we make the following contributions:

- An overhaul of Xen's architecture, completely removing its back-end registry (a.k.a. the XenStore) which represents not only a performance bottleneck but also a single point of failure. We call this noxs (no XenStore), and its implementation results in significant improvements for boot and migration times, among other metrics.
- A revamp of Xen's toolstack, including a number of optimizations and the introduction of a *split* toolstack that separates functionality that can be run periodically, offline, from that which must be carried out when a command (e.g., VM creation) is issued.

In this document, we will describe the general requirements and the design and implementation of LightVM. The final version of this deliverable will then include full evaluation results as well as a demonstration of LightVM's applicability through the implementation of three use cases which might include fast, fine-granularity filters against large, DDoS attacks; a high-density TLS termination proxy; and a lightweight compute service akin to Amazon Lambda or Google's Cloud Functions but based on a Python unikernel.

5.2 Requirements and Initial Investigation

The goal is to be able to provide lightweight virtualization on top of hypervisor technology. More specifically, as requirements, we are interested in a number of characteristics typical of containers:

- **Fast Instantiation**: Containers are well known for their small startup times, frequently in the range of hundreds of milliseconds or less. In contrast, virtual machines are infamous for boot times in the range of seconds or even minutes.
- **High Instance Density**: It is common to speak of running hundreds or even up to a thousand containers on a single host, with people even pushing this boundary up to 10,000 containers [15]. This is much higher than what VMs can typically achieve, more in the range of tens or hundreds at most, and normally requiring fairly powerful servers.
- **Pause/unpause**: Along with short instantiation times, containers can be paused and unpaused quickly. This can be used to achieve even higher density by pausing idle instances, and more generally to make better use of CPU resources. Amazon Lambda, for instance, "freezes" and "thaws" containers.
- **Low Memory Footprint**: Because they share the host/kernel, containers typically require much less memory than virtual machines (a few *MB*s or tens of *MB*s versus hundreds of *MB*s or even *GB*s for VMs).

Before we can even begin to devise a solution that would meet these requirements, we need to understand both the magnitude of the problem and what the issues are. To do so, we must select one of the major, existing hypervisor technologies and conduct an initial investigation as to what it is capable of out of the box, and where the bottlenecks and problems lie.

We settle on Xen [9], since the fact that it is a type-1 hypervisor and so has a small trusted computing base and its code is fairly mature, result in strong isolation (the ARM version of the hypervisor, for instance, consists of just 11.4K LoC [24], and disaggregation [11] can be used to keep the size of critical dom0 code low). To better understand the following investigation, we start with a short introduction on Xen.

5.3 Short Xen Primer

The Xen hypervisor is a type-1 hypervisor in charge of managing basic resources such as CPUs and memory (see Figure 21). When it finishes booting, it automatically creates a special virtual machine called `dom0`. Dom0 typically runs Linux and hosts the *toolstack*, which includes the `xl` command and the `libxl` and `libxc` libraries needed to carry out commands such as VM creation, migration and shutdown.

In addition, `dom0` hosts the XenStore, a proc-like central registry that keeps track of management information such as which VMs are running in the system and device information, along with the `libxs` library containing code to interact with it. The XenStore provides *watches* that can be associated with particular directories of the store and that will trigger callbacks whenever those directories are read or written to.

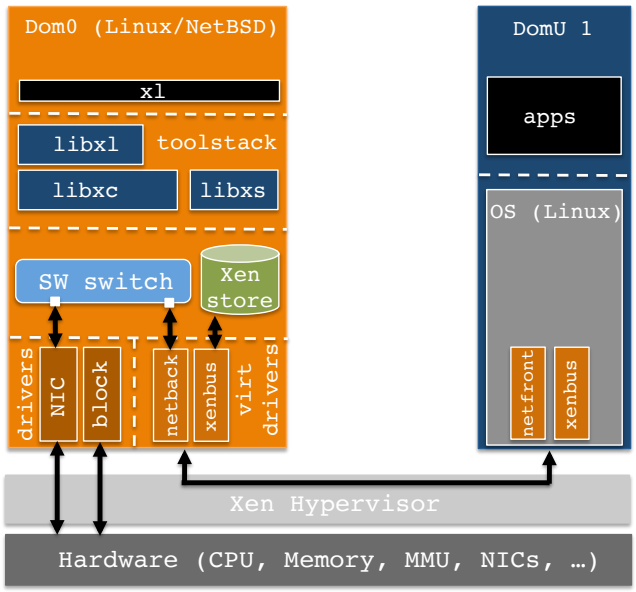

Figure 21: The Xen architecture including toolstack, the XenStore, software switch and split virtual drivers between the driver domain (dom0) and the guests

Typically, **dom0** also hosts a software switch (Open vSwitch is the default) to mux/demux packets between NICs and the VMs, as well as the (Linux) drivers for the physical devices (Strictly speaking, this functionality can be put in a separate VM called a driver domain, but in most deployments **dom0** acts as a driver domain). For communication between **dom0** and the other guests, Xen implements a *split-driver* model: a virtual back-end driver running in **dom0** (e.g., the netback driver in the case of networking) communicates over shared memory with a front-end driver running in the guests (the **netfront** driver). So-called *event channels*, essentially software interrupts, are used to notify drivers about the availability of data.

5.4 Design and Implementation

In this section, we introduce LightVM, a re-design of some of the basic Xen mechanisms in order to provide lightweight virtualization over that platform.

5.4.1 LightVM Architecture

The three main obstacles to implementing lightweight virtualization on Xen: (1) the XenStore, since many operations require multiple interactions with it, a problem that worsens with an increasing number of VMs; (2) the toolstack, where many functions run whenever a command is executed even though they do not need to; and (3) to a lesser extent the hotplug script in charge of adding virtual devices to the software switch (we will present evaluation results confirming this in the final version of this deliverable).

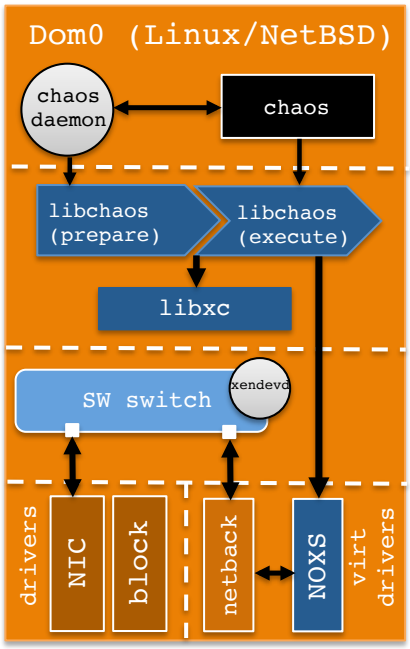

Figure 22: LightVM overall architecture showing noxs, the split toolstack (chaos) and accompanying daemon, and xendevd in charge of quickly adding virtual interfaces to the software switch

To address these, we re-design significant portions of the Xen ecosystem and call the resulting architecture LightVM (see Figure 22). Briefly, we remove the XenStore and replace it with a lean driver called noxs, provide a split toolstack that separates functionality between a preparation and an execution phase, use a much leaner **`chaos/libchaos`** library than the standard **`xl/libxl`** and include a small daemon called **`xendevd`** that quickly adds virtual interfaces to the software switch. Next, we cover each of these in detail.

5.4.2 Eliminating the XenStore (NOXS)

The XenStore is crucial to the way Xen functions, with many **`xl`** commands making heavy use of it. As way of illustration, Figure 23 shows the process when creating a VM and its (virtual) network device. First, the toolstack writes an entry to the network back-end's directory, essentially announcing the existence of a new VM in need of a network device. Previous to that, the back-end placed a watch on that directory; the toolstack writing to this directory triggers the back-end to assign an event channel and other information (e.g., grant references, a mechanism for sharing memory between guests) and to write it back to the XenStore (step 2 in the figure). Finally, when the VM boots up it contacts the XenStore to retrieve the information previously written by the network back-end (step 3).

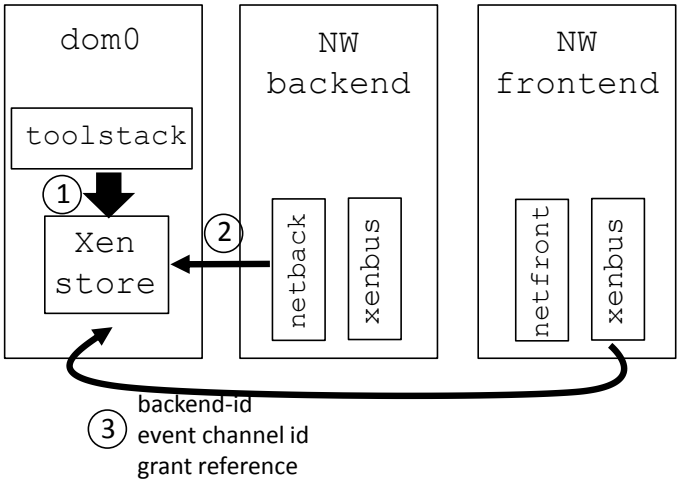

Figure 23: Standard VM creation process in Xen

The above is a simplification: in actuality, the VM creation process alone can require interaction with over 30 XenStore entries, a problem that is exacerbated with increasing number of VMs and devices. Worse, the Xen- Store represents a single point of failure.

Is it possible to forego the use of the XenStore for operations such as creation, pause/unpause and migration? As it turns out, most of the necessary information about a VM is already kept by the hypervisor (e.g., the VM's id, but not the name, which is kept in the XenStore but is functionally not strictly needed). This insight here is that the hypervisor already acts as a sort of centralized store, so we can extend its functionality to implement our noxs (no XenStore) mechanism.

Specifically, we begin by replacing `libxl` and the corresponding `xl` command with a streamlined, thin library and command called `libchaos` and chaos, respectively (please refer back to Figure 22); these no longer make use of the XenStore and its accompanying `libxs` library. Further, the toolstack keeps track of which back-ends are available (e.g., network, block) and is responsible for assigning any information needed for communication between them and the guests (e.g., event channels and grant references).

In addition, we modify Xen's hypervisor to create a new, special device memory page for each new VM that we use to keep track of a VM's information about any devices, such as block and networking that it may have. We also include a hypercall to write to and read from this memory page, and make sure that the page is shared read-only with guests (but read/write with `dom0`).

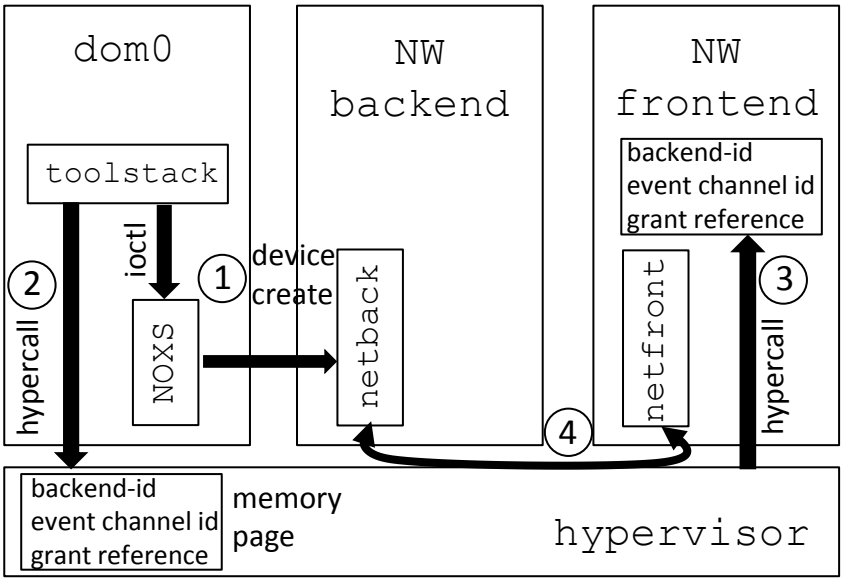

Figure 24: VM creation process using the noxs implementation

When a chaos create command is issued, the toolstack first requests the creation of devices from the back-end(s) through an `ioctl` handled by the noxs Linux kernel module (see step 1 in Figure 24); note that currently this mechanism only works if the back-ends run in `dom0`, but the architecture allows for back-ends to run on a different virtual machine. The back-end then returns the details about the communication channel for the front-end. Second, the toolstack calls the new hypercall asking the hypervisor to add these details to the device page (step 2 in the figure).

When the VM boots, instead of contacting the XenStore, it will ask the hypervisor for the address of the device page and will map the page into its address space using hypercalls (step 3 in the figure); this requires modifications to the guest's operating system, which we have done for Linux and MiniOS. The guest will then use the information in the page to initiate communication with the back-end(s) by mapping the grant and bind to the event channel (step 4). At this stage, the front and back-ends setup the device by exchanging information such as its state and its mac address (for networking); this information was previously kept in the XenStore and is now stored in a device

control page pointed to by the grant reference. Finally, front and back-ends notify each other of events through the event channel, which replaces the use of XenStore watches.

To support migration without a XenStore we create a new pseudo-device to handle power-related operations. We implement this device, called `powerctl`, following Xen's split driver model, with a back-end driver (power- back) and a front-end (powerfront) one. These two devices share a device page through which communication happens.

With this in place, migration begins by `chaos` opening a TCP connection to a migration daemon running on the remote host and by sending the guest's configuration so that the daemon pre-creates the domain and creates the devices. Next, to suspend the guest, chaos issues an `ioctl` to the back-end, which will set a field in the shared page to denote that the shutdown reason is 'suspend'. The front-end will receive the request to shutdown, upon which the guest will save its internal state and unbind noxs-related event channels and device pages. Once the guest is suspended, we rely on `libxc` code to send the guest data to the remote host.

5.4.3 Split Toolstack

In the previous section we showed that a significant portion of the overheads related to VM creation and other operations comes from the toolstack itself. Upon closer investigation, it turns out that a significant portion of the code that executes when, for instance, a VM create command is issued, does not actually need to run at VM creation time. This is because this code is *common to all VMs*, and so can be pre-executed and thus off-loaded from the creation process.

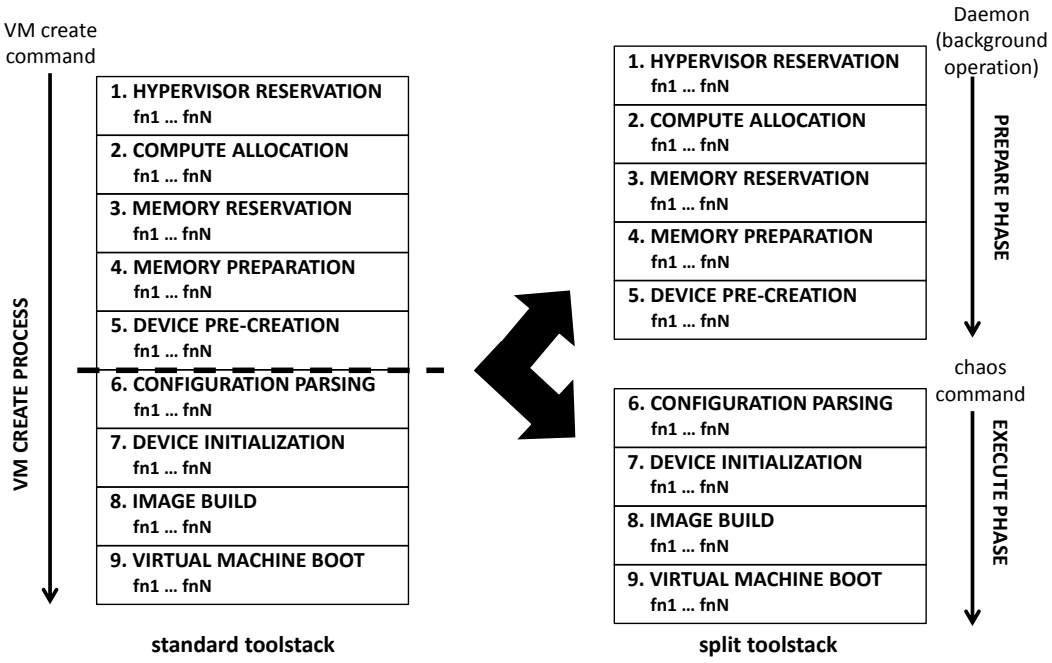

Figure 25: Toolstack split between functionality belonging to the prepare phase, carried out periodically by the chaos deamon, and an execute phase, directly called by chaos when a command is issued

To take advantage of this, we replace the standard Xen toolstack with the `libchaos` library and split it into two phases. The *prepare* phase (see Figure 25) is responsible for functionality common to all VMs such as having the hypervisor generate an ID and other management in- formation and allocation CPU resources to the VM. We *offload* this functionality to the chaos daemon, which generates a number of VM *shells* and places them in a pool. The daemon ensures that there is always a certain number of (configurable) shells available in the system.

The *execute* phase then begins when a VM creation command is issued. First, `chaos` contacts the daemon and asks for one of these shells to be removed from the pool. It then completes this shell with VM-specific operations such as parsing its configuration file, building initialization its devices and booting the VM. 6

5.5 Related Work

A number of OS-level virtualization technologies exist and are widely deployed, including Docker, LXC, FreeBSD jails and Linux-VServer, among others [6] [7] [16] [26]. In terms of high density, the work in [15] shows how to run 10,000 Docker containers on a single server. Zhang et al. implement network functions using Docker containers and can boot up to 80K of them [29]. In our work, we make the case for lightweight virtualization on top of a hypervisor, providing strong isolation while retaining the attractive properties commonly found in containers.

In terms of hypervisors such as Xen, KVM and VMWare [24] [29] [31], a number of works have looked into optimizing those platforms to reduce their overheads in terms of boot times and other metrics. For example, Intel Clear Containers [29] takes a similar approach to ours, optimizing KVM to enable lightweight virtualization on that platform. Their reliance on KVM results in a larger trusted computing base and their boot times and VM memory footprint are worse than ours (roughly 150 ms and 18-20 MB per-container overhead, respectively). Along those lines, ukvm [32] implements a specialized unikernel monitor on top of KVM and uses MirageOS unikernels to achieve 10 ms boot times (the main metric the work focuses on). Jitsu [12] optimizes parts of Xen to implement just-in-time instantiation of network services by accelerating connection start-up times. In our work, we aim to provide container-like dynamics, simultaneously providing (1) small boot, pause/unpause and migration times (sometimes an order of magnitude smaller than previous work), (2) high density and (3) low per-VM memory footprints.

Beyond containers and virtual machines, other works have proposed the use of minimalistic kernels or hypervisors to provide lightweight virtualization. Exokernel [14] is a minimalistic operating system kernel that provides applications with the ability to directly manage physical resources. NOVA [30] is a microhypervisor consisting of a thin virtualization layer and thus aimed at reducing the attack surface (NOVA's TCB is about 36K LoC, compared to for instance 11.4K for Xen's ARM port). The Denali isolation kernel is able to boot 10K VMs but does support legacy OSes and has limited device support [34]. The work in [27] proposes the implementation of cloudlets to quickly

offload services from mobile devices to virtual machines running in a cluster or data center, although the paper reports VM boot times in the 60-90 seconds range.

Finally, unikernels [24], virtual machines based on minimalistic operating systems, have seen a significant amount of interest in the research domain; Mirage [24], ClickOS [5], Erlang on Xen [15] and OSv [22] are a few examples of these. Our work does not focus on unikernels, but rather leverages them to be able to separate the effects coming from the virtual machine from those of the underlying virtualization platform. For example, they allow us to reach high density numbers without having to resort to overly expensive servers.

5.6 Future Work

We have introduced a re-vamping of the Xen hypervisor architecture that allows us to derive "superfluid" characteristics from virtual machines that are usually typical only of containers, all the while retaining the strong isolation guarantees that hypervisors are famous for.

In the final version of this deliverable, we will include a full evaluation of our prototypical implementation, including use cases to show the applicability of our design.

6 Open vSwitch (OVS) firewall

6.1 Open vSwitch Firewall Driver

Open vSwitch (OVS) is the most popular network back-end for OpenStack deployments and it is widely accepted as the de facto standard OpenFlow implementation. Its main goal is to be a programmable switch, implementing both traditional switching functionality as well as programmability through OpenFlow. It is commonly used in network virtualization and most of its functionality is implemented/evaluated in user-space and a set of flows are programmed into the kernel with matches and actions.

OVS is good for stateless, flow-based networking. For instance to compose a network, including switching, routing and building network processing pipelines. However, it leaves out services that might be inserted into that network, such as firewalls. However, OpenStack security groups (SG) give you a way to define packets filtering policy that is implemented by the cloud infrastructure. Unfortunately, OVS could not interact directly with iptables to implement security groups. To overcome this problem, OVS integration into OpenStack (ML2 + OVS) make use of iptables to implement security groups, however a linux bridge between each instance (VM) and the OVS integration bridge is required, i.e., the VM needs to be connected to a tap device that is put on a linux bridge, and then connect that linux bridge to the OVS bridge using a veth pair (see Figure 26). Then, the linux bridge contains the iptables rules pertaining to the instance.

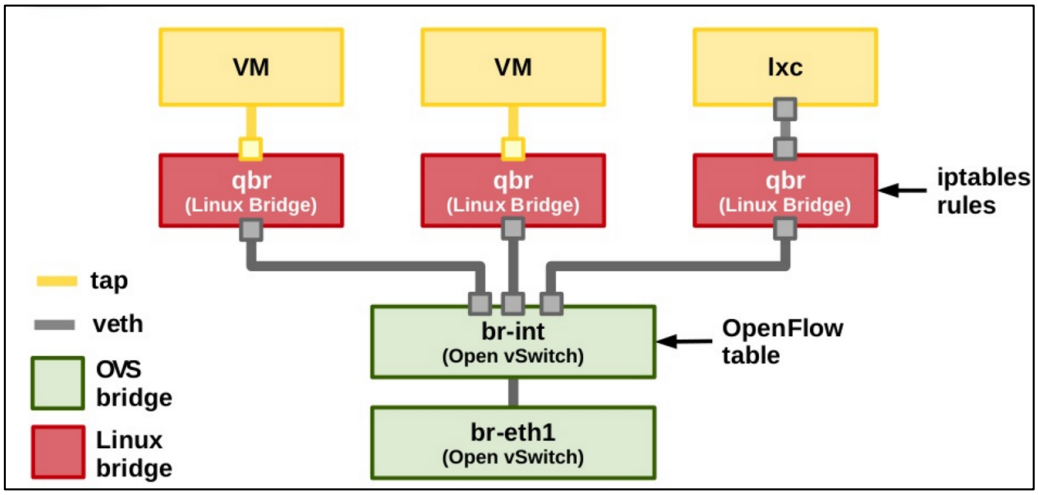

Figure 26: Open vSwitch integration with Linux Network stack

Even though it is great that this works, the extra layers are not ideal and create both scalability and performance problems that could be of great impact in the 5G/Edge cloud scenario, therefore being a barrier for OpenStack adoption for NFV purposes. In order to get rid of all of the extra layers

between VMs (and/or containers) and OVS, and yet being able to apply security groups, there is a need of building stateful firewall services in OVS directly. Consequently, the solution was to modify the OVS agent to include an optional firewall driver that natively implements security groups as flows in OVS rather than linux bridge and iptables. There were two ways of implementing a firewall in OVS, but none ideal:

- Match on TCP flags by enforce policy on SYN, allowing ACK|RST. Although this solution is fast, it allows non-established flows through with ACK or RST set (only TCP).
- Use "learn" action to setup new flow in reverse direction. This solution seems more correct than the previous one, but it forces every new flow to OVS user-space, reducing flow setup by order of magnitude.

OVS development initially had a narrow focus on supporting novel features necessary for advanced applications such as network virtualization. However, due to its wider use it became clear it was not enough with OVS being highly programmable and general, but it also needed to be blazingly fast. Consequently none of the previous solutions was good enough and a new one was needed.

The solution proposed is the integration of connection tracking (conntrack module) from linux kernel to enable stateful tracking of flows, i.e., the OVS can call into the kernel connection tracker. To understand how this is achieved, it is needed to understand the OVS architecture – see Figure 27. The forwarding plane consist of two parts: a slow-path user-space daemon called ovs-vswitchd and a fast-path kernel module. Most of the complexity such as forwarding decisions and networking protocol processing are handled at user-space; while the kernel module is in charge of tunnel termination and caching traffic handling.

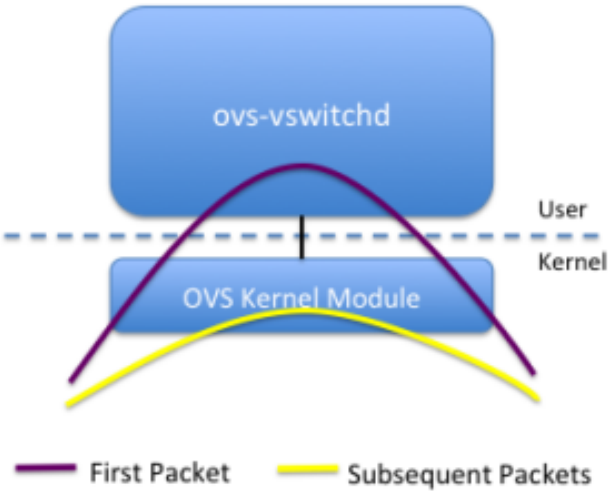

Figure 27: OVS architecture


The conntrack functionality can be used to build stateful services in OVS as a stateless flow can select a class of traffic that needs to be tracked, passing matching packets off to the conntracker for further evaluation. Thanks to its use, when a packet is received by the kernel module, its cache of flows is consulted. If a relevant entry is found, then the associated actions (e.g., modify headers or forward the packet) are executed on the packet. By contrast, if there is no entry, the packet is passed to ovs-vswitchd (i.e., user-space) to decide the packet's fate. Ovs-vswitchd then executes the OpenFlow pipeline on the packet to compute actions, passes it back to the fast-path for forwarding, and install a flow cache entry so similar packets will not need to take these expensive steps. Therefore, subsequent packets are processed entirely in the kernel, with the consequent processing speed up.

Without going into tedious details about the flows, here is an idea of what it lets you do in your packet processing pipeline:

1. In one stage, you can match all IP traffic and send it through the connection tracker.
2. In the next stage you now have the connection tracker's state associated with this packet, so:
    a. For packets representing a new connection, a custom policy can be used to decide if connection should be accepted or not. If it is accepted, you can tell the connection tracker to remember this connection
    b. When packets are associated with existing connections they can be allowed through, and the opposite for return traffic.
    c. It is known if a packet is invalid because it is not the right type of packet for a new connection and does not match may existing known connection.

6.1.1 OVS Features

The OVS firewall driver has the same API as the current iptables firewall driver, keeping the state of security group and ports inside of the firewall. Class SGPortMap was created to keep state consistent and maps from ports to security groups and vice-versa. Every port and security group is represented by its own object encapsulating the necessary information. Note: Open vSwitch firewall driver uses register 5 for marking flow related to port and register 6 which defines network and is used for conntrack zones, i.e., to support overlapping CIDR when belonging to different tenants.

6.1.1.1 Firewall API calls

There are two main calls performed by the firewall driver in order to either create or update a port with security groups - prepare_port_filter and update_port_filter. Both methods rely on the security group objects that are already defined in the driver and work similarly to their iptables counterparts:

- prepare_port_filter must be called only once during port creation, and it defines the initial rules for the port.
- When the port is updated, all filtering rules are removed, and new rules are generated based on the available information about security groups in the driver.

Security group rules can be defined in the firewall driver by calling update_security_group_rules, which rewrites all the rules for a given security group. If a remote security group is changed, then update_security_group_members is called to determine the set of IP addresses that should be allowed for this remote security group. Calling this method will not have any effect on existing instance ports. In other words, if the port is using security groups and its rules are changed by calling one of the above methods, then no new rules are generated for this port. update_port_filter must be called for the changes to take effect.

All the machinery above is controlled by security group RPC methods, meaning that the firewall driver does not have any logic of which port should be updated based on the provided changes, it only accomplishes actions when called from the controller.

6.1.1.2 OpenFlow rules

At first, every connection is split into ingress and egress processes based on the input or output port respectively. Each port contains the initial hardcoded flows for ARP, DHCP and established connections, which are accepted by default. To detect established connections, a flow must be marked by conntrack first with an action=ct() rule. An accepted flow means that ingress packets for the connection are directly sent to the port, and egress packets are left to be normally switched by the integration bridge.

Connections that are not matched by the above rules are sent to either the ingress or egress filtering table, depending on its direction. The reason that forces rules based on security group rules to reside in separate tables, is to allow their easy detection during removal. For more information, please visit: http://docs.openstack.org/developer/neutron/devref/openvswitch_firewall.html

6.1.2 OVS Usage

The native OVS firewall implementation requires the kernel and user space support for conntrack and consequently a minimum versions of the Linux kernel and Open vSwitch:


- Kernel version 4.3 or newer includes conntrack support.
- Kernel version 3.3, but less than 4.3, does not include conntrack support but OVS modules can be built
- Open vSwitch version 2.5 or newer

On the other hand, to enable the OVS firewall driver on the nodes running the OVS agent, it is needed to edit the openvswitch_agent.ini file and enable the firewall driver like this:

```
[securitygroup]
firewall_driver = openvswitch
```

7 Cache Allocation Technology

A major challenge with Network Functions Virtualisation (NFV) is ensuring protection of Virtual Network Function (VNF) resources against "Noisy Neighbor" effects. This is essentially the result of shared resources being consumed *in extremis* within a multi-tenant setup, meaning one VNF's resources are restricted by that of another VNF. One of the major shared resource bottlenecks is the central processor's Last Level Cache (LLC). This deliverable details a testbed which enables "Cache Allocation Technology" (CAT), so as to deterministically prioritize LLC resources between competing workloads. A number of CAT "Class of Service" (CoS) paradigms are explained for a range of service chain scenarios, involving virtual Firewall and virtual Router VNFs alongside a Noisy Neighbor VNF. Significant performance benefits are confirmed, bringing the performance of target VNFs in the presence of a LLC-hungry Noisy Neighbor, into alignment with the baseline scenario of a "Noise-Free" neighboring VNF.

7.1.1 Summary of results

Cache Allocation Technology (CAT) can be used to make NFV performance more *predictable & deterministic*, by mitigating Noisy Neighbor effects on shared processor resources such as the Last level Cache (LLC). For specific tests undertaken, the most notable impact on performance was observed when target VNFs were deliberately starved of LLC resources by allocating minimal LLC bandwidth (i.e. a single CAT CoS way). This was referred to as the "Uneven", or "11-1" model, whereby 11 of the cache's 12 available CoS ways were assigned to the Noisy Neighbor VNF, while the remaining single CoS way was used by all other workloads. The Even CAT CoS settings mitigate against performance impacts, and bring the results into close approximation to the "Stress-Free" Neighbor scenario where the LLC resources are not being aggressively hogged. As an example, the "Even" CAT CoS model, reduced average latency between 47% & 92% compared to Uneven CoS, while the maximum latency was reduced between 49% & 98%.

7.2 CAT Background

7.2.1 Noisy Neighbors and Last Level Cache (LLC)

A crucial aspect of NFV performance management is ensuring protection of VNF resources against "Noisy Neighbor" effects. This is the result of shared resources being consumed *in extremis* within a multi-tenant setup, meaning one VNF's resources are restricted by that of another VNF, in such a way as to negatively impact performance. Within a physical Central Processing Unit (CPU) socket architecture (Figure 28), a number of shared resource pools exist including interconnect/IO, Last level Cache (LLC) and Shared Memory Bandwidth (SMB). So although a VNF can have CPUs

"pinned" for performance assurance, they will still be subject to resource-sharing within the LLC, and if a memory-hungry neighboring VNF is instantiated on the same host OS, this constitutes a "Noisy Neighbor" problem.

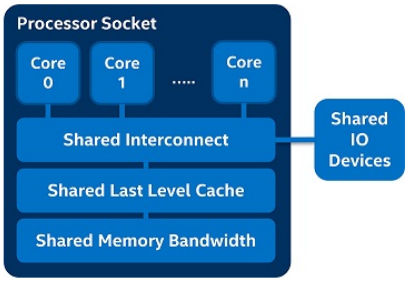

Figure 28: Shared Processor Resources

Since 2015, Intel launched a range of core processors enabled to support features such as Cache Monitoring Technology (CMT), Cache Allocation Technology (CAT) and Memory Bandwidth Monitoring (MBM). The technologies provide pro-active monitoring of shared resource consumption, and active management of the available resources in order to achieve platform-level Quality-of-Service for different VNFs in a multi-tenant environment. This paper provides a detailed practical analysis of the impact of CAT on a realistic multi-tenant NFV scenario, using commercially-available virtual Router and virtual Firewall VNFs as the "systems under test". This builds and expands on previous studies such as [6], which although able to demonstrate the performance benefits of CAT, were based on completely synthetic workloads for both the "Target" and "Noisy Neighbor" VNFs. Section II provides an overview of the testbed, including the methodology for implementing active CAT. Section III presents the detailed test results for a range of scenarios, while Section IV presents the conclusions and key directions for further work.

7.3 Testbed Overview

The testbed is based on Linux KVM (Kernel-based Virtual Machine) using Open vSwitch plus DPDK, and the x86 server uses Intel "Xeon D" processors, enabled to support CMT and CAT software. Table 3 lists key hardware/software elements.

Table 3:Testbed Hardware and Software Components

Testbed Component	Version/Description
X86 Server Hardware	Intel Xeon D-1537 (16 * logical vCPU processors, 16G RAM, 256GB SSD Storage)
Hypervisor Base OS & Kernel	Centos 7 3.10.0-327.22.2.e17.x86_64
Hypervisor Open vSwitch (OVS)	2.6.0
DPDK (Data Plane Development Kit)	16.07
QEMU	2.5.1.1

VNF 1- vRouter	Brocade 5600 virtual router: 4 vCPU, 4G RAM
VNF2- vFirewall	Fortinet Fortigate VM64-KVM: 1 vCPU, 1G RAM
VNF3- Noisy Neighbor VM	Fedora 22: 2 vCPU, 2G RAM Stress Processes: stress-ng-0.03.20, memtester version 4.3.0
Test Equipment	Spirent Avalanche 4.40, running "RFC2544" throughput test wizard, fixed +"iMIX" profiles

The testing involved two distinct service chain scenarios as shown in Figure 29. The first comprised of the so-called "Target" VNF, which could either be the virtual Firewall or virtual Router, alongside a "Noisy Neighbor" VNF in a multi-tenant, mixed VNF scenario. The target VNFs are in the data path and considered revenue-generating and high value. The second scenario involved two serially-connected virtual Routers alongside the Noisy Neighbor VNF. In both scenarios, the Noisy Neighbor is not in the data path, but has access to the Last Level Cache (LLC) and other processor resources shared by all VNFs. This is an ideal test case, as the impacts of Noisy Neighbor on the "Target" VNFs can be observed, while maintaining a distinct separation of the data path used by the traffic generator. The Noisy Neighbor makes use of multiple synthetic resource-hogging stress processes installed and run from the Linux OS, and as shown in Table 1, include "*stress-ng*" and "*memtester*" applications.

The target VNFs use "vhost" drivers connecting to the virtual switches (labelled as Lan-Bridge and Wan-bridge in the diagram), which themselves are connected to physical Gigabit Ethernet interfaces set-up using DPDK "Poll-Mode Drivers" (PMDs). As already stated, the Noisy Neighbor VNF is not in the data path and thus uses standard "virtio" drivers connecting to the virtual switch labelled "vm-bridge".

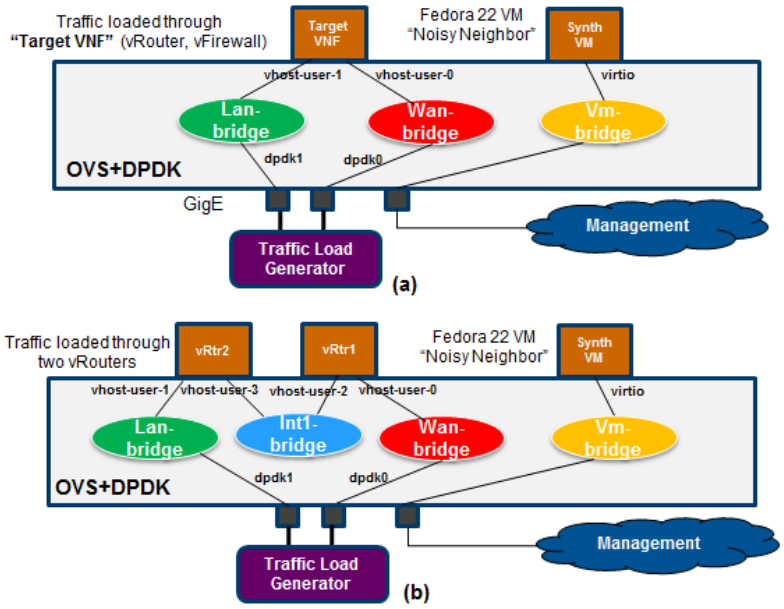

Figure 29: (a) Target VNF + Noisy Neighbor (b) Dual VNFs + Noisy Neighbor

7.4 Methodology

Cache Allocation Technology is run in the host hypervisor, and involves the use of a "Class of Service" (CoS) construct acting as a resource control tag into which a CPU thread can be grouped. Other, more granular mappings at application/VM/container level are also possible, but this paper covers the CPU thread case. Software-programmable control is provided over the amount of Last-Level Cache (LLC) space that can be consumed by a given CPU core thread, hence prioritization can be applied for Virtual Network Functions (VNFs) running on the host hypervisor.

Definition of capacity bitmasks determines how much of the LLC space is available, and the degree of overlap and isolation. Typically, a capacity bit to cache way mapping is 1:1 but one-to-many options are also possible. For the Intel® Xeon® D-1537 processor, the LLC has 12 ways, and the CoS capacity bitmask has 12 bits, each comprising 1MB of cache storage. The term "CoS ways" is also used in this paper to describe the capacity bitmask. Table 4shows which CPUs are pinned for specific VNFs and processes; it is worth noting that in the Linux system setup, isolation of CPU cores 1-15 ensures core 0 is reserved for Linux processes.

Table 4: VNF and System Process CPU Pinning

Function/Process	Pinned CPUID(s)- 16 vCPU processor
Noisy Neighbor VNF	6,7
Target VNF- Single Firewall	5
Target VNF- Single Router	1,2,3,5
Target VNF- Second Router	9,10,11,13
OVS-PMD- (Open vSwitch Poll Mode Driver)	4, 12
OVS-db (Open vSwitch Database)	1

Table 5 shows the range of CoS definitions and corresponding CPUID assignments used for the performance evaluation described in detail in the following section. In the Uneven CoS model, 11 of the 12 CoS Ways are assigned to the Noisy Neighbor, while only a single CoS way is available for the remaining CPUIDs, which includes that of the target VNF (e.g. vRouter or vFirewall). This model encourages an extremely aggressive "hogging" of LLC resources by the Noisy Neighbor to demonstrate what impact on performance the cache starving can have on the "Target VNFs"; this scenario could indeed be representative of a worst-case scenario whereby a *malicious* Noisy Neighbor generates intentional Denial-of-Service behaviour to shared cache resources. The Even CoS models allow a fairer distribution of LLC bandwidth (i.e. an equal split of CoS ways) between competing Guest VNFs on the host platform, as well as other key system processes such as OVS-PMD and OVS-db.

Table 5: Initial Class of Service Definitions Using Capacity Bitmasks

CAT CoS Model	Binary Bitmasks	Hex	Cache capacity	CPU Assignments
Uneven ("11-1")	**1111 1111 1110** 0000 0000 000**1**	0xffe 0x1	11 MB 1MB	6,7 (Noisy N'bor) 0-5,8-15 (Everything Else)
Even ("4-4-4", Single virtual Firewall)	**1111** 0000 0000 0000 **1111** 0000 0000 0000 **1111**	0xf00 0xf0 0xf	4 MB 4 MB 4 MB	6,7 (Noisy N'bor) 5 (virtual Firewall) 0-4,8-15 (Everything Else)
Even ("4-4-4", Single virtual Router)	**1111** 0000 0000 0000 **1111** 0000 0000 0000 **1111**	0xf00 0xf0 0xf	4 MB 4 MB 4 MB	6,7 (Noisy N'bor) 1,2,3,5 (virtual Router) 0,4,8-15 (Everything Else)
Even ("3-3-3-3", Dual virtual Routers)	**111**0 0000 0000 000**1 11**00 0000 0000 00**11 1**000 0000 0000 0**111**	0xe00 0x1c0 0x38 0x7	3 MB 3 MB 3 MB 3 MB	6,7 (Noisy N'bor) 1,2,3,5 (1st virtual Router) 9,10,11,13 (2nd virtual Router) 0,4,8,12,14,15 (Everything Else)

7.5 Results

7.6 Virtual Firewall Tests

Figure 30 shows the maximum throughput (Mbit/s) for the virtual Firewall alongside the Noisy Neighbor VNF for a selected range of fixed frame sizes, and three setup variations:

- No stress processes running in Neighboring VNF.
- Stress processes running in Neighboring VNF and Uneven "11-1" CoS model of Table 3.
- Stress processes running in the Neighboring VNF and Even "4-4-4" CoS model of Table 3.

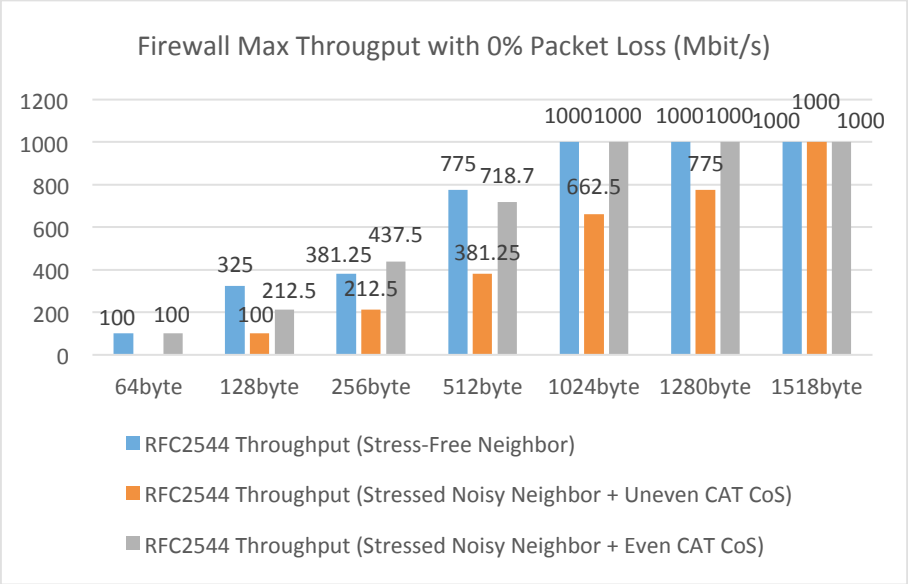

Figure 30: vFirewall Maximum "RFC2544" Throughput

Figure 31 and Figure 32 show the corresponding average and maximum latency for a subset of fixed frame sizes, with measurements taken after 60 seconds, and the traffic load used for each case was the corresponding baseline throughput realized with a *stress-free* neighboring VNF: 64byte @ 100Mbits, 256byte @ 381Mbit/s, 1024byte & 1518byte @ 1000Mbit/s.

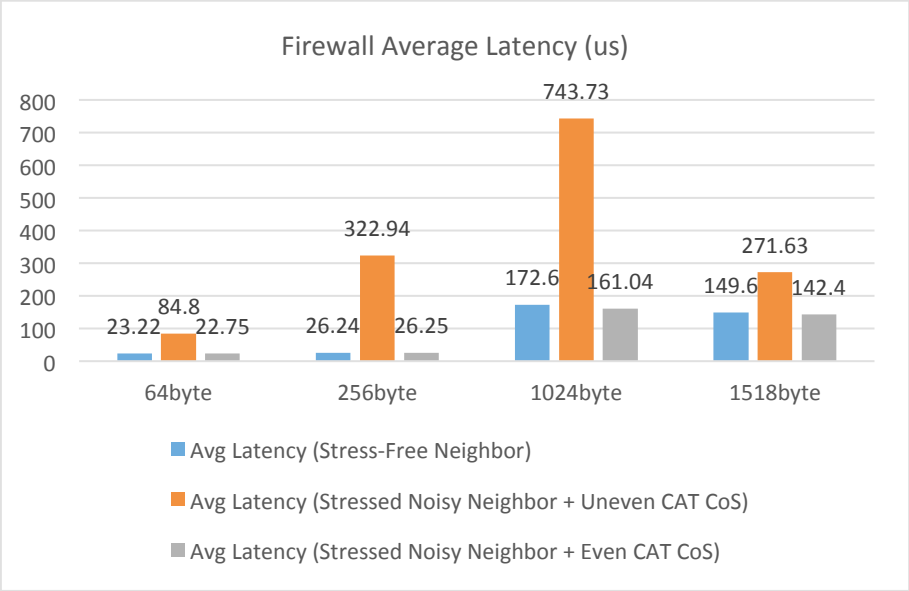

Figure 31: vFirewall Average Latency

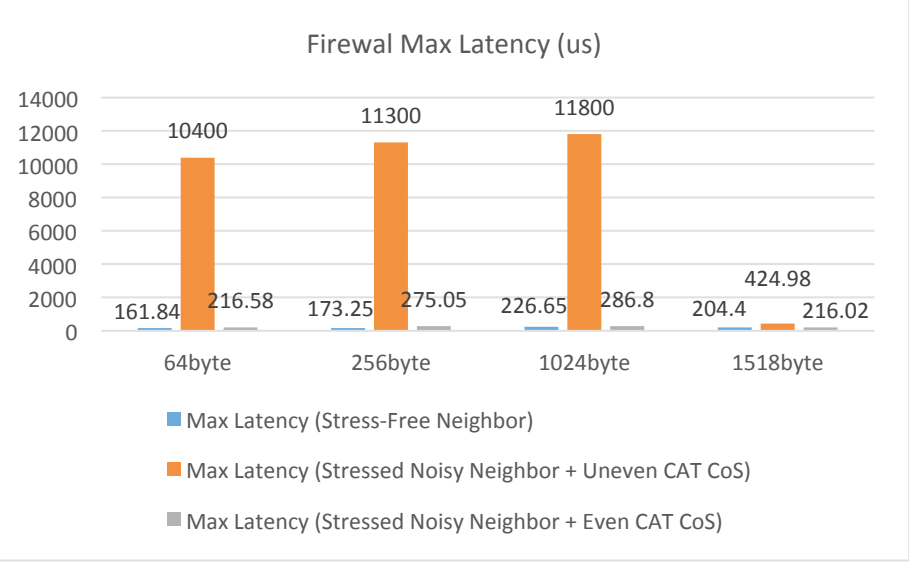

Figure 32: vFirewall Maximum Latency

The graphs show that Firewall throughput & latency are significantly impacted by the presence of a Noisy Neighbor + Uneven CAT CoS settings. The Even CAT CoS settings mitigate against performance impacts, and bring the results into close approximation to the "Stress-Free" Neighbor scenario where the LLC resources are not being aggressively hogged. The average latency is below 170us, while the maximum is below 300us achieving more deterministic performance. In

percentage terms, the "Even" CAT CoS model, reduces average latency between 47% & 92% compared to Uneven CoS, while maximum latency is reduced between 49% & 98%. It is also insightful to observe the LLC occupancy of the Noisy Neighbor and virtual Firewall VNFs during the load tests. Figure 33 shows the measured LLC in KB using the embedded Cache Monitoring Technology (CMT)- for the Uneven and Even CoS cases, respectively.

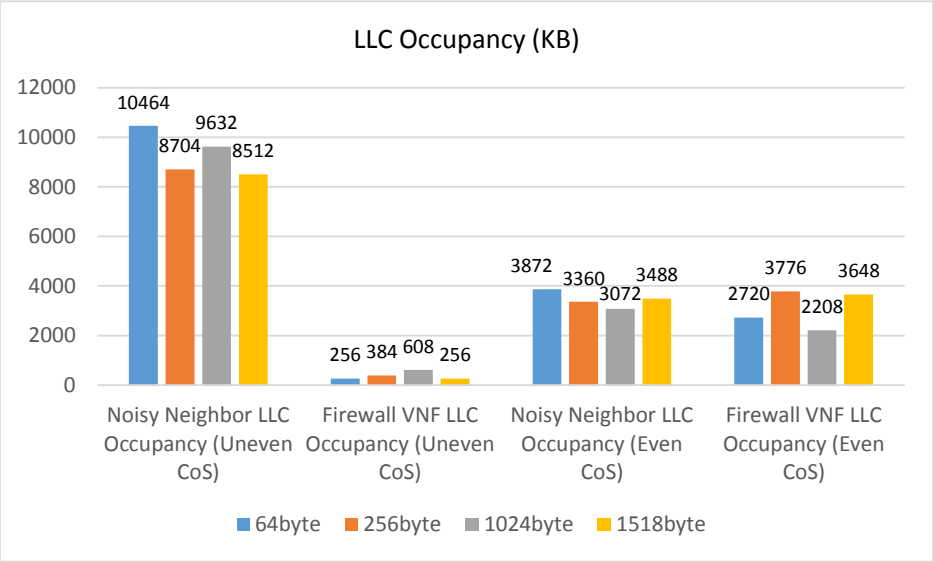

Figure 33: Comparative LLC Occupancy for Uneven and Even CoS Paradigms

These experiments confirm the occupancy levels of the LLC is heavily dominated by the Noisy Neighbor in the Uneven "11-1" case, (e.g. ~10.5MB utilisation for 64byte packets), but is much more balanced between the Noisy Neighbor and Target VNF for the Even CoS "4-4-4" case.

7.7 Virtual Router Tests

A similar set of tests were executed with the virtual Router substituting the virtual Firewall with the same three setup variations as described earlier. The tests achieved broadly comparable results with those for the virtual Firewall, hence for brevity are summarized in Table 6.

Table 6: Average and Maximum Latency SIngle Virtual Router

Fixed Frame Size	Average Latency us (Noisy Neighbor, Uneven CoS)	Average Latency us (Noisy Neighbor, Even CoS)	Max Latency us (Noisy Neighbor, Uneven CoS)	Max Latency us (Noisy Neighbor, Even CoS)
64byte	7096	29.59 (-**99%**)	8542	115.08 (-**98%**)
256byte	119.14	38.83 (-**67%**)	207.77	88.84 (-**57%**)
1024byte	140.11	86.59 (-**38%**)	206.22	102.34 (-**50%**)
1518byte	143.08	91.62 (-**36%**)	164.71	109.11 (-**33%**)

With Even CoS, the average latency is below 100us, while the maximum is below 120us achieving more deterministic performance. In percentage terms, the "Even" CAT CoS model, reduces average latency between 36% & 99% compared to Uneven CoS, while the max latency is reduced between 33% & 98%. The same virtual Router was also used in a "Three VNF" set-up whereby two virtual Routers were service chained together, plus a Noisy Neighbor (i.e. the testbed setup of Figure 29(b)) the Even CoS model in this case is the "3-3-3-3" model of Table 5. Table 7 presents both the Average and Maximum Latency Figures.

Table 7: Average and Maximum Latency Single Virtual Router

Fixed Frame Size	Average Latency us (Noisy Neighbor, Uneven CoS)	Average Latency us (Noisy Neighbor, Even CoS)	Max Latency us (Noisy Neighbor, Uneven CoS)	Max Latency us (Noisy Neighbor, Even CoS)
64byte	13438.95	41.64 (-**99%**)	16335.98	163.77 (-**99%**)
256byte	14948.75	76.75 (-**99%**)	17506.31	171.47 (-**99%**)
1024byte	186.53	134.1 (-**28%**)	261.24	160.79 (-**38%**)
1518byte	328.81	138.8 (-**57%**)	1321.77	163.59 (-**87%**)

For the dual virtual Routers with Even CoS, average latency is sub 140 us, while the maximum is sub 180 us. In percentage terms, the "Even" CAT CoS model, reduces average latency between 28% & 99% versus Uneven CoS, while maximum latency is reduced between 38% & 99%.

7.8 Variable CoS Model Results

The availability of multiple "CoS ways" to manage LLC resources opens up the possibility of a large and flexible number of set-up permutations in terms of CPU thread allocations, whether isolated or overlapping assignments are made, and so on. The earlier results were based on distinct paradigms to show the performance impact of a Noisy Neighbor on target VNFs while using distinctly "Uneven" and "Even" CAT CoS models. It is insightful to run additional tests using *intermediate* CAT CoS settings as shown in Table 8. Compared to the "11-1" model, the "7-4-1" model reduces the LLC bandwidth allocation for the Noisy Neighbor (from 11 to 7 CoS ways), whilst dedicating resources for key system processes, hence assigns 4 CoS ways for "Everything Else" including host OS, OVS-db, and OVS-PMD. Just a single CoS way is used for the target VNF, which, in these tests is the virtual Firewall. The "6-4-2" model meanwhile, doubles the LLC bandwidth for the target VNF (1 to 2 CoS ways), while reducing the Noisy Neighbor CoS ways from 7 to 6.

Table 8: Additional (intermediate) Class of Service Definitions

CAT CoS Model	Binary Bitmasks	Hex	Cache capacity	CPU Assignments
"7-4-1"	**1111 111**0 0000 0000 000**1 111**0 0000 0000 000**1**	0xfe0 0x1e 0x1	7 MB 4 MB 1 MB	6,7 (Noisy Neighbor) 0-4,8-15 (Everything Else) 5 (virtual Firewall)
"6-4-2"	**1111 11**00 0000 0000 00**11 11**00 0000 0000 00**11**	0xfc0 0x3c 0x3	6 MB 4 MB 2 MB	6,7 (Noisy Neighbor) 0-4,8-15 (Everything Else) 5 (virtual Firewall)

The average and maximum latency for both fixed 256 byte frames and "iMIX"- generating a *realistic* split of frame sizes - are shown in Figure 34 and Figure 35: the results are plotted for the various CAT CoS paradigms, and clearly illustrate how different permutations of CoS way allocations yield different outcomes, with some dependency on traffic profiles. The "7-4-1" profile reduces average latency compared to "11-1" (i.e. Uneven CoS), but is more notable for fixed 256byte frames, than with iMIX traffic. For maximum latency, the "7-4-1" profile has no notable impact compared to "11-1". The "6-4-2" profile visibly reduces both average and maximum latency compared to "11-1" (i.e. Uneven CoS) for both traffic cases and indeed has comparable values to the "4-4-4" paradigm, suggesting even a single extra CoS way can notably improve target VNF performance. A compelling observation therefore is confirmation of a *minimum* required number of CoS ways for the target VNF (i.e. two), to achieve comparable performance with the completely "Even CoS" model.

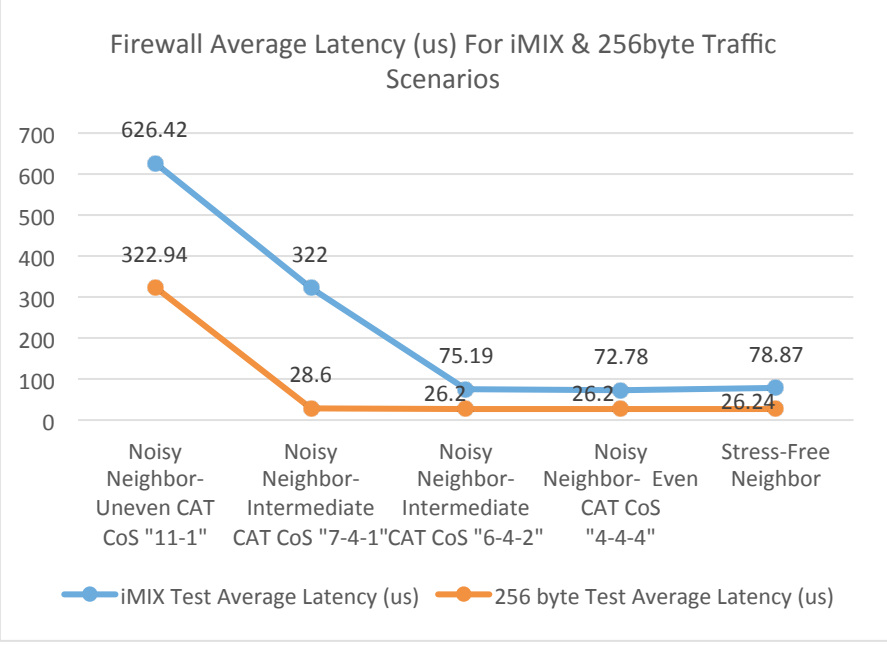

Figure 34: vFirewall Average Latency

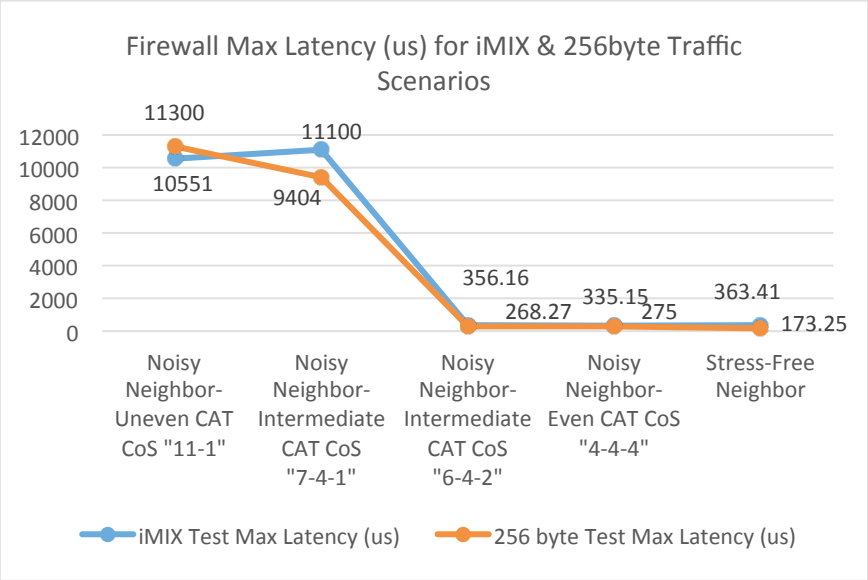

Figure 35: vFirewall Maximum Latency

8 Performance analysis of using the MicroVisor for an NFV platform

The following section describes the performance analysis of ONAPP's MicroVisor platform for running NFV type operations. In most of the comparisons, we compare the performance of stock Linux, Xen, a Virtual Machine (VM) running on Xen and a VM running on the MicroVisor (MV). For more detail about how the platforms have been set up please see Appendix I in Section 11. These results are interim and are expected to be more fully populated for the release of the final deliverable.

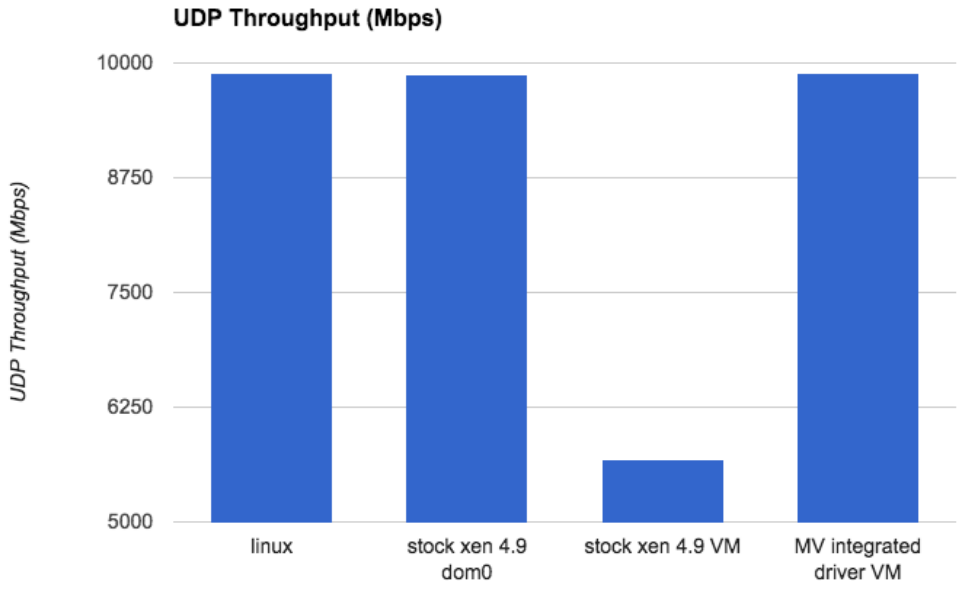

Figure 36: UDP throughput compared for stock Linux, Xen, Stock Xen VM and MV VM

In Figure 36, we plot the UDP throughput measured with the netperf benchmark between a bare-metal node running vanilla linux v.4.11 and a node running:

a) Vanilla linux v4.11
b) Stock Xen 4.9
c) MV

The goal of this experiment is to identify the overhead related to virtualising network I/O paths. In the first bar we set our baseline: two linux hosts are able to transfer UDP data at a rate of 9.8Gbps, which is almost the available line rate. This small overhead is considered to be protocol related.

The third bar plots the throughput measurement between a vanilla linux host and a stock Xen guest VM (bridged setup). Clearly there is considerable overhead related to a number of factors:

a) Driver domain intervention. To account for the driver domain overheads, we analyse this in more detail. The second bar in this graph plots the driver domain UDP throughput. The virtualisation overhead involved in this case is PCI passthrough (interrupt mapping to event channel translation) as well as memory access overheads regarding Para-virtualized page table walking and updates.

b) Scheduling effects. The Xen scheduler plays a significant role in I/O throughput with regards to scheduling a specific guest to a physical core in time for it to complete I/O. This is not network specific. When a hardware interrupt reaches the hypervisor, the driver domain is notified using the event channel mechanism and the incoming packet walks up the network stack of dom0's linux kernel. It crosses the network bridge and reaches the netback driver which ends up issuing a hypercall to notify the guest of an incoming frame. The guest is notified and it wakes up on a different core, completing the network transaction. So there needs to be really good collaboration of all the intermediate components (scheduling driver domain's vCPUs and guest's vCPUs to specific cores) in order to achieve maximum throughput. This is clearly not the case in the stock Xen approach.

The fourth bar presents the UDP throughput achieved by a guest running on a MV without a driver domain (integrated driver). Clearly, the MV integrated driver guest outperforms the stock Xen guest in terms of UDP throughput by almost 70% (9.8 Gbps vs. 5.6Gbps), achieving near-line rate and identical throughput to the baremetal linux case as well as the stock Xen dom0 case.

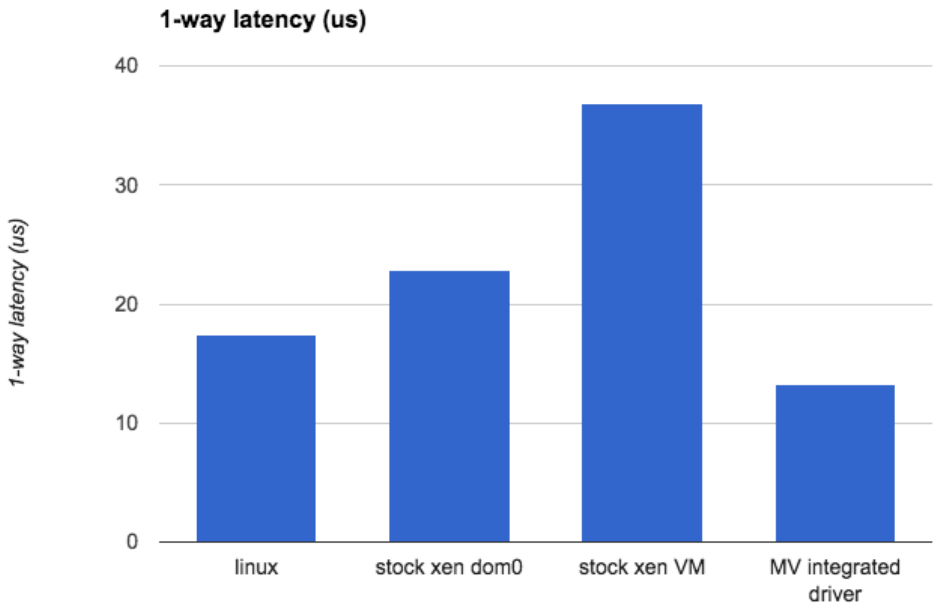

Figure 37: 1-way network latency for Linux, Xen dom0, Xen VM and MV integrated driver

Furthermore, we measure the number of transactions/packets per second that the network interfaces can transmit and receive in the same testbed. These numbers translate into the 1-way latency that the system is able to achieve. We plot these in Figure 37.

Once again, we use the linux case as a baseline -- transferring one byte from a linux host to another one using the UDP protocol takes approximately 17 us. Performing the same benchmark in a guest VM running on stock Xen we see that it takes 37 us, almost 2x more. To validate the hypothesis that the source of overhead is network bridging, pci passthrough and Xen scheduling, we perform the same experiment as above, measuring the 1-way latency for dom0. This case is clearly better (22 us), but still over the baseline (30% more).

To showcase the advantage of using the MV integrated driver case, we measure the time that an ethernet frame with a 1 byte payload needs to leave the MV (hypervisor context) and reach an opposite node (MV, hypervisor context). We find that the time needed is approximately 13 us. This provides a clear advantage over all the other cases, even baremetal linux. It is expected that the UDP protocol overhead regarding latency is not more than 1-2%.

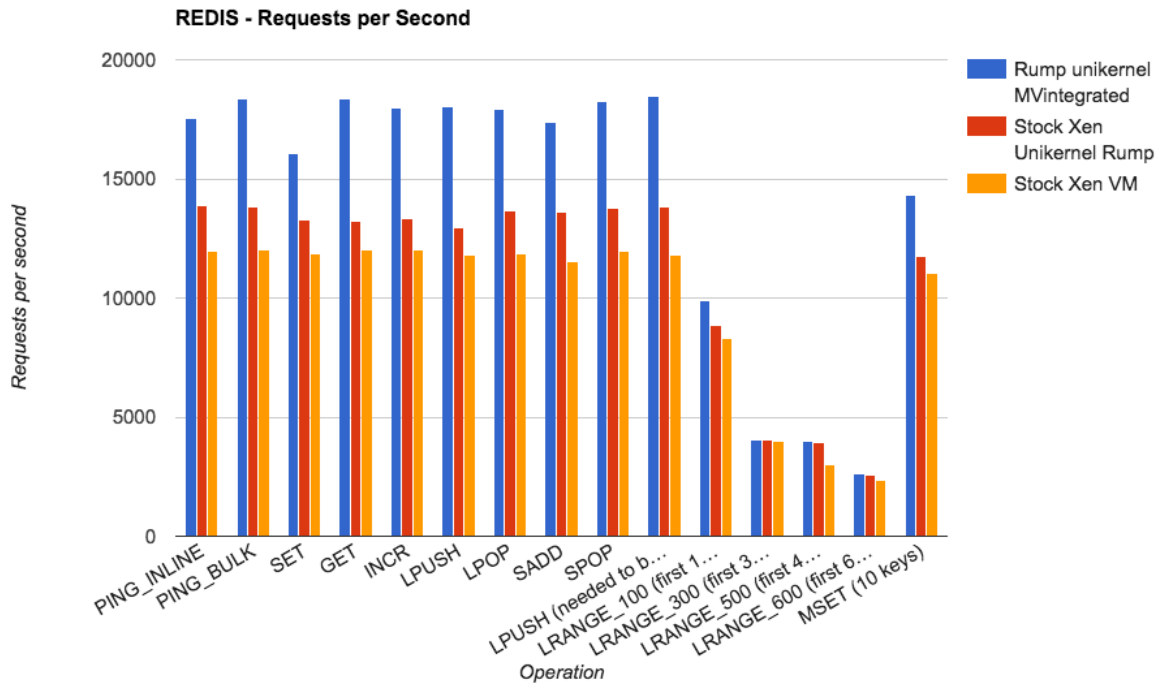

Figure 38: Redis DB - requests per second handled by RUMP Unikernel running on MV, Xen and standard Xen

To provide a real-life use-case scenario we deploy a popular, in-memory data structure store, REDIS and capture its performance from a remote client. We run the standard redis-benchmark tool, provided by the redis software stack.

Network bandwidth and latency usually have a direct impact on REDIS performance.

Figure 38 plots the number of requests per second a specific REDIS instance can handle when deployed in a stock Xen guest VM and as a rump unikernel in a MV setup and in a stock Xen setup.

We can derive two important issues from Figure 38: first, the unikernel instance of the REDIS application provides some benefit in terms of request handling; second, the improvement in network latency and throughput plays a significant role as seen in the MV case.

Specifically, we highlight the difference of the two key sets of measurements of Figure 38 and plot the percentage improvement of the rump unikernel in the MV and Xen normalized with baremetal linux execution. These results are plotted in Figure 39.

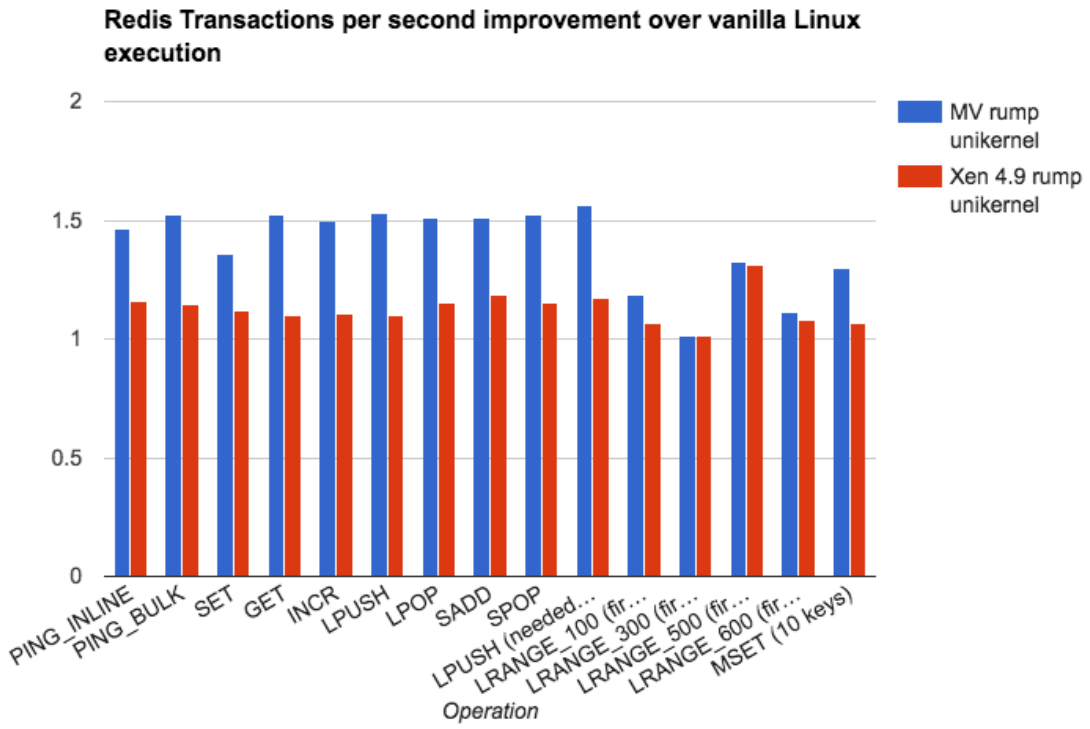

Figure 39: Redis DB performance, normalised against stock Linux for MV and Xen running RUMP unikernels

We see that for most of the tests, the rump unikernel ran in the MV is almost 50% better than baremetal linux execution. Compared to the stock Xen case, the benefit is almost 30-40%. For only one set of the tests, execution is exactly the same as in linux and stock Xen. In most of the cases,


the improvement in network latency and throughput directly affects the performance. Given that the cpu/memory virtualization overhead is negligible in this kind of benchmark, and that the unikernels are pinned to specific physical CPUs, the performance benefit from running a unikernel instance of this kind of workload is clear.

9 Conclusion

In this interim deliverable, we have provided an update on various technologies that have been worked on in the course of Task 5.2 since M13 until M21. Different virtualisation techniques are being applied at different levels of the stack in order to increase the performance. Although the work currently is piecemeal there will be a strong push to try and get these technologies working together to provide cumulative performance benefits.

We have seen how performance improvements to the underlying hypervisor platform can push improved performance up the stack in the MicroVisor section. We have also seen how improving network virtualisation can help improve the manageability and interoperability of the platform in the OVS work. By reducing the overhead of the VIM we have seen that improvements again can be made as reported in the efforts regarding Chaos VM manager.

We also note the architectural elements, key to SUPERFLUIDITY including the utilisation of re-usable function blocks that are composed by means of RDCL. Each of these components and systems provides improvements on the state of the art on their own. The next, challenging phase will be on how to integrate these technologies together.

10 Bibliography

[1] (2004) Xen Project. [Online]. http://www.xenproject.org

[2] Scapy project. [Online]. http://www.secdev.org/projects/scapy/

[3] VIM tuning and evaluation tools. [Online]. https://github.com/netgroup/vim-tuning-and-eval-tools

[4] Openstack rally. [Online]. https://wiki.openstack.org/wiki/Rally

[5] Joao Martins et al., "ClickOS and the art of network function virtualization," in *11th USENIX Conference on Networked Systems Design and Implementation*, 2014, pp. 459-473.

[6] Docker. The Docker Containerization Platform. [Online]. https://www.docker.com/

[7] LinuxContainers. LinuxContainers.org. [Online]. https://linuxcontainers.org

[8] J. CLARK. (2014, May) theregister.co.uk. [Online]. http://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/

[9] Microsoft. Azure Container Service. [Online]. https://azure.microsoft.com/en-us/services/container-service/

[10] Amazon. Amazon EC2 Container Service. [Online]. https://aws.amazon.com/ecs

[11] AWS. (2017, March) AWS Lambda - Serverless Compute. [Online]. https://aws.amazon.com/lambda

[12] Google. The Google Cloud Platform Container Engine. [Online]. https://cloud.google.com/container-engine

[13] A. MADHAVAPEDDY et al., "Jitsu: Just-In-Time Summoning of Unikernels," *USENIX Symposium on Networked Systems Design and Implementation*, no. 12, pp. 559–573, 2015.

[14] J. SHERRY et al., "Making Middleboxes Someone Else's Problem: Network Processing As a Cloud Service.," in *Conference on Computer Communication*, New York, 2012, pp. 13–24.

[15] W. ZHANG, J. HWANG, S. RAJAGOPALAN, K. RAMAKRISH-NAN, and T WOOD, "Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization," in *Conference on Emerging Networking EXperi-ments and Technologies*, vol. 12, 2016, pp. 3–17.

[16] A. VERMA, G. DASGUPTA, T. K. NAYAK, P. DE, and R. KOTHARI, "Server Workload Analysis for Power Minimization Using Consolidation," in *Annual Technical Conference*, Berkeley, 2009, pp. 28–28.

[17] E. KOVACS. securityweek.com. [Online]. http://www.securityweek.com/docker-fixes-

vulnerabilities-shares-plans-making-platform-safer

[18] A. GRATTAFIORI. Understanding and Hardening Linux Containers. [Online]. https://www.nccgroup.trust/us/our-research/understanding-and-hardening-linux-containers/

[19] A. MOURAT. 5 security concerns when using Docker. [Online]. https://www.oreilly.com/ideas/five-security-concerns-when-using-docker

[20] J. HERTZ. Containers. Abusing Privileged and Unprivileged Linux. [Online]. https://www.nccgroup.trust/uk/our-research/abusing-privileged-and-unprivileged-linux-containers/

[21] I. MARINOS, R. N. WATSON, and M. HANDLEY, "Network Stack Specialization for Performance.," in *Conference on Computer Communication SIGCOMM*, New York, 2014, pp. 175–186.

[22] P. EMELYANOV. Slideshare.net. [Online]. http://www.slideshare.net/Docker/live-migrating-a-container-pros-cons-and-gotchas

[23] T ANDERSEN. (2015, May) Live Migration in LXD. [Online]. https://insights.ubuntu.com/2015/05/06/live-migration-in-lxd/

[24] IBM. (2015, November) Docker at insane scale on IBM Power Systems. [Online]. https://www.ibm.com/blogs/bluemix/2015/11/docker-insane-scale-on-ibm-power-systems

[25] P. BARHAM et al., "Xen and the Art of Virtualization," *SIGOPS Oper. Syst.*, no. 37, pp. 164–177, October 2003.

[26] S. STABELLINI. slideshare.net. [Online]. http://www.slideshare.net/xen_com_mgr/alsf13-stabellini

[27] P. COLP et al., "Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor.," in *Symposium on Operating Systems Principles*, vol. 23, New York, 2011, pp. 189–202.

[28] P. HENNING KAMP and R. N. M. WATSON, "Jails: Confining the omnipotent root. ," in *SANE Conference*, vol. 2, 2000.

[29] Soltesz Stephen, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors.," *Operating Systems Review*, vol. 41, no. 3, pp. 275-287, March 2007.

[30] VMWare. VMWARE. vSphere ESXi Bare-Metal Hypervisor. [Online]. http://www.vmware.com/products/esxi-and-esx.html

[31] A. KIVITY, Y. KAMAY, D. LAOR, U. LUBLIN, and A. LIGUORI, "KVM: the Linux Virtual Machine Monitor," in *Ottawa Linux Symposium*, Ottawa, 2007.

[32] A. VAN DE VEN, An introduction to Clear Containers.

[33] D. WILLIAMS and R. KOLLER, "Unikernel Monitors: Extending Minimalism Outside of the Box.," in *Hot Topics in Cloud Computing*, vol. 8, Denver, 2016.

[34] D. R. ENGLER, M. F. KAASHOEK, and JR., J. O'TOOLE, "Exokernel: An Operating System Architecture for Application- level Resource Management.," in *Symposium on Operating Systems Principles*, vol. 15, New York, 1995, pp. 251–266.

[35] U. STEINBERG and B. KAUER, "NOVA: A Microhypervisor-based Secure Virtualization Architecture.," in *5th European Conference on Computer Systems EuroSys*, New York, 2010, pp. 209–222.

[36] A. WHITAKER, M. SHAW, and S. D. GRIBBLE, "Scale and Performance in the Denali Isolation Kernel. ," *SIGOPS Oper. Syst*, vol. 36, pp. 195–209, December 2002.

[37] M. SATYANARAYANAN, P. BAHL, R. CACERES, and N. DAVIES, "The Case for VM-Based Cloudlets in Mobile Computing," *IEEE Pervasive Computing*, no. 8, pp. 14–23, October 2009.

[38] Anil Madhavapeddy and David, J. Scott, "Unikernels: Rise of the virtual library operating system.," *Queue* , vol. 11, no. 11, p. 30, 2013.

[39] ErlangonXen. (2012, July) ErlangonXen. [Online]. http://erlangonxen.org/

[40] A., LAOR, D., COSTA, G., ENBERG, P., HAR'EL, N., MARTI, D., AND ZOLOTAROV, V.K IVITY, "OSv-Optimizing the Operating System for Virtual Machines," in *USENIX Annual Technical Conference*, Philadelphia, 2014, pp. 61–72.

[41] Intel Whitepaper. (2016, February) Increasing Platform Determinism With Platform Quality of Service for the Data Plane Development Kit. [Online]. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/increasing-platform-determinism-pqos-dpdk-paper.pdf

11 Appendix I

Details of how the ONAPP benchmarking was configured are captured here.

Benchmark setup

Benchmark sets

a) Memcached (v1.4.36)
   Memaslap (libmemcached v1.0.18)
   Client: memaslap -s host:port -t 60s -T 1 -c 20 -X 64
   Server options: -u root -m 8192
b) Redis (v3.2.8)
   client: redis-benchmark -h host -p port -q -c 1
   Server options: maxmemory $[4096 * 1048576]
c) Netperf (2.7.0)
   netperf -t UDP_RR, TCP_RR, UDP_STREAM, TCP_STREAM
   Server options: default

1) Baremetal
   In this test, we setup two nodes with baremetal linux (v4.11) and run all benchmarks from the above set between the two nodes and localhost
2) Docker
   In this test, we setup one docker host with the respective containers (memcached, redis and netperf) and execute the tests between the baremetal node and the docker container. There are cases where for completeness we ran the tests from the dockerhost to the container
3) Stock Xen VM
   We setup a Xen host with default values and run the tests from/to the baremetal node.
4) Stock Xen unikernel
   Instead of using a fully blown VM, we use a unikernel built with the rumpkernel framework.
5) MV unikernel
   We setup one node as a MV and the other as the baremetal node.
6) MV integrated driver unikernel
   Same as Stock Xen unikernel.

The docker containers used are:

- paultiplady/netperf (netperf, iperf, NPtcp)
- Redis:3.0.7-alpine (redis)
- Memcached:1.4.35-alpine (memcached)