FUTURE COMMUNICATION ARCHITECTURE FOR MOBILE CLOUD SERVICES
Acronym: MobileCloud Networking Project No: 318109
Integrated Project FP7-ICT-2011-8 Duration: 2012/11/01-2015/09/30

D3.4 Infrastructure Management Foundations – Final Report on component design and implementation

Type	Report
Deliverable No:	3.4
Workpackage:	WP3
Leading partner:	INTEL
Author(s):	Thijs Metsch, List of Authors overleaf.
Dissemination level:	PU
Status:	Final
Date:	30 June 2015
Version: 1.0	1.0

List of Authors

- Andy Edmonds (ZHAW)
- Alex Georgiev (CS)
- Andre Gomes (UBERN/ONE)
- Andy Edmonds (ZHAW)
- Atoosa Hatefi (Orange)
- Bruno Sendas (PTIN)
- Bruno Sousa (ONE)
- Carlos Parada (PTIN)
- Cláudio Marques (ONE)
- David Palma (ONE)
- Dominique Pichon (Orange)
- Eryk Schiller (UBERN)
- Faisal Mir (NEC)
- Florian Dudouet (ZHAW)
- Giada Landi (NXW)
- Giuseppe Carella (TUB)
- Islam Alyafawi (UBERN)
- Jakub Kocur (TUB)
- Julius Mueller (TUB)
- Lucio Ferreira (INOV)
- Lucio Studer Ferreira (INOV)
- Luigi Grossi (TI)
- Luis Cordeiro (ONE)
- Luis M. Correia (INOV)
- Luisa Caeiro (INOV)
- Luuk Hendriks (UTWENTE)
- Marius Corici (FhG)
- Michael Erne (ZHAW)
- Miguel Dias (PTIN)
- Monica Branco (INOV)

- Navid Nikaein (Eurecom)
- Paulo Tavares (PTIN)
- Peter Gray (CS)
- Piyush Harsh (ZHAW)
- Sina Khatibi (INOV)
- Thijs Metsch (INTEL)

Reviewers:

- Paolo Crosta (ITALTEL)
- Santiago Ruiz (STT)
- Andy Edmonds (ZHAW)

Versioning and contribution history

Version	Description	Contributors
0.1	Initial draft	INTEL
0.2	Version for peer review	TI PTIN NEC INTEL CS NXW ONE UTWENTE TUB INOV UBERN ZHAW Fraunhofer Orange, EURECOM
0.3	Changes based on peer review	INTEL
0.4	Version for GA review	TI PTIN NEC INTEL CS NXW ONE UTWENTE TUB INOV UBERN ZHAW Fraunhofer Orange, EURECOM
0.5	Changes based on GA review	INTEL
1.0	Final version ready for submission	TI PTIN NEC INTEL CS NXW ONE UTWENTE TUB INOV UBERN ZHAW Fraunhofer Orange, EURECOM

Executive Summary

This document presents the final results of the work-package 3 of the MCN project. It reports on the final architectural changes and key findings when implementing the infrastructure foundations. Even more key services have been evaluated for performance. It includes all the details on the infrastructure management foundations for a Mobile Cloud Framework.

The methodology to develop the foundation of a MobileCloud can be summarized as: the definition of and research on the components in need, the implementation of those components through prototypes (PoC) and finally the delivery of the components. This follows the same flow as the deliverables of this work-package. From the initial architecture and concepts report, two prototypical deliverables have been provided and this deliverable provides the final report on all components.

To summarize this deliverable, it will show the final findings on the major components making up the infrastructure foundation for a MobileCloud:

- Delivering enhanced networking foundations to support all services within the MCN project by leveraging state-of-the-art Software-Defined-Networking (SDN) capabilities. Next to this basic services to support other network functions (e.g. DNS, Re-direction service) have been implemented, tested and provided to the consortium.
- Methodologies for performance testing have been identified and applied to test the Performance tests of Cloud services. Major outcomes show that different technical implementation of Cloud services leads to different performance outcomes. Hence different technologies have been integrated into the foundations.
- When dealing with large-scale distributed systems gaining insights into on goings is key. Hence a full-stack cloud-native monitoring system has been integrated into the foundations. Next to this analytical concept can be applied to the data derived through this system.
- To bind all concepts together a CloudController provides a central component which enables existing (legacy) and future Mobile services to make use of the Cloud.
- Connectivity is a huge issue in distributed systems. Hence a Radio Access Network has been integrated into the foundation to provide this core feature, which enables higher layer services within and external to the MCN project.

Overall this work-package has implemented numerous services – which integrated to the CloudController – offers a rich set of features and through that form the foundations for a MobileCloud. The services have been implemented using an Agile development methodology. Documentations and test driven development ensured that a robust environment can be provided. To prevent the re-invention of the wheel many existing services have been throughout evaluated – and extended where needed. To prevent a lock-in to these technologies abstractions and standardised interfaces have been used.

Table of contents

Table of figures

Table of tables

1 Introduction

This is the final deliverable of work-package 3 within the MCN project. It shows the work achieved throughout the project for each of the tasks. Based on the initial concepts and architecture which is described in (D3.1 2013) two prototypes have been delivered: (D3.2 2014) and (D3.3 2014). As the technical work has hence been completed this deliverable presents the final findings and where necessary updates.

The work package is organized by task. However each task in this work package might have contributed to a set of services within the project. Each service has its own architecture and implementation. The overarching architecture to ensure all this services can play together is achieved through the overarching architecture described in (D2.2 2013) of the project and the main integration component developed in this work package: the CloudController.

Overall this deliverable presents the components and services needed to provide a foundation for the true MobileCloud. Spanning from basic service such as IaaS all the way to connectivity services such as RAN. The integration of all these services provides the basis for a future NVF compliant cloud enabled MobileCloud.

Outcomes of this work will be used by other work-packages, including WP6. Within that work-package there is a specific task for "Experimentation and Evaluation" (task 6.5).

1.1 How to read this document

This deliverable follows the same structure as (D3.1 2013). Hence it presents the outcomes of the work of each task in order (task 3.1 to task 3.5). Each of the tasks sections describes the outcomes of the tasks based on their services, implementations in the prototypes and their research work.

Following the task descriptions some more generic services of the category support are described in their own section.

It is strongly recommended to read (D3.1 2013) as it details the concept and architectures. In case of interest, the more technical details and pointers to reference code and documentation are presented in the deliverables (D3.2 2014) and (D3.3 2014).

1.2 Relationships between tasks

Tasks and work packages are highly integrated within the MCN project as shown in section 1.2 of (D3.1 2013). The executive summary has already shown how the work from the tasks in WP3 build upon each other. A more detailed view of task interdependence within the work package is shown in Table 1. It shows the kind of relationships between the tasks. For example Task 3.1 provides its features on Software Defined Networking towards Task 3.5.

Table 1 Task relationships

	Task 3.1	**Task 3.2**	**Task 3.3**	**Task 3.4**	**Task 3.5**
Task 3.1		Networking enhancements	Monitorable network	Network connectivity	SDN capabilities
Task 3.2	Networking enhancements		Performance measurements	Performance enhancements	Performance enhancements

Task 3.3	Monitoring capabilities	Performance measurements		Monitoring of the CC & Services	Monitoring capabilities
Task 3.4	*CloudController uses the features exposed*				E2E Orchestration
Task 3.5	*RANaaS consumes the features exposed*				

As mentioned earlier, each task in this document will detail their relationships in their respective sections.

In addition to task relationships within this work package, the relationships to other work packages and associated tasks are identified:

- WP2 – The specifications of all entities in this deliverable have been synchronized and have driven the overall architectural work done in Task 2.3.
- WP4 & WP5 – Task 3.1, 3.2 and 3.3 have been in particularly close contact with these work packages. Task 3.4 has been driving the overall architecture, and therefore in close contact with the Services using the CloudController. Task 3.5 is developing one of the Services using the foundations described here, and therefore is used to validate the architectural decisions. Task 3.5 is in tight relation with WP4, where together they are building the access and core mobile network architecture.
- WP6 – Implementations of the entities described within this document will be deployed on the testbeds to finally enable the scenarios described in past deliverables.
- WP7 – Some work carried out in this work package depends on standardization activities. They can be influenced by implementations realized in the next project phase. For now the standards used in this deliverable are listed and observed through work package 7. Next to this, technical outcomes were disseminated through the social media channels. It is also noted here that several people from this WP are contributing to the efforts of Standards Developing Organizations (SDOs).

2 Networking Foundations

The following sections describe the contributions from task 3.1, entitled "OpenFlow Extensions to OpenStack".

2.1 Introduction

Services are being shifted to the cloud, managed by Cloud Service Providers (CSP) that offer resources across the globe for serving geographically distributed end users. In the cloud context, resources are composed by virtual machines (VMs), storage and networks in the different data centres (DCs) owned by CSPs. The networking foundations to enable the communication between the distributed resources relies on Software Defined Networking (SDN) due to the separation of networking switching functions (minimize delay in packet processing) from the information intensive functions (corresponding to the classical networking rules).

The early reported problem statement, objectives or requirements have not changed since Deliverable (D3.1 2013) .

2.2 Network-as-a-Service

The following sections describe the Networking services which are based around the concepts of Software Defined Networking (SDN). They are roughly split by two concepts: Intra and Inter-Cloud network management.

2.2.1 Intra Cloud networking

Network-as-a-Service in MCN is used for the on-demand delivery of the virtual network infrastructures and services (e.g. DNS, load balancing), which enable the communication between the VMs composing the end-to-end MCN instances. These virtual infrastructures are logically isolated and delivered per-tenant, following the specification provided in the service description through the ITG. At the network data plane, they are mapped over intra-DC and, when required, inter-DC connections. MCN has defined an architecture (see section 2.2.2) where the Data Centre Network (DCN) is a programmable entity composed of OpenFlow L2-L3 devices which are configured through a centralized SDN controller implementing the logic of the network layer. The network programmability is a key feature to enable the orchestration of network services with end-to-end cloud services, in order to support some of the main principles of cloud resources, from self-service delivery to elasticity.

2.2.2 Architecture

The architecture to enable Network-as-a-Service has been specified in (D3.1 2013). The functional elements of the cloud service provider and network connectivity provider architectures are the same of the original specification. Nonetheless, the technologies enabling such components have been updated as detailed in Table 2.

Table 2 Network architecture components: final technologies

Architecture component	Candidate software tool or interface	Notes
Network Frontend	Neutron API	The APIs exposed by Neutron to provide networking services between devices managed by OpenStack compute service.

		The current version is v2.0 and allows managing three main types of entities (network, subnet, port) through the common CRUD operations.
	OCCI	The network part of the OCCI specification delivered by OGF can be used to request networking resources through a REST interface.
	OCNI	A cloud networking extension to OCCI developed under the European project SAIL (SAIL 2013).
Network Management System	Neutron	Neutron is the OpenStack component dedicated to the Networking-as-a-Service. It is considered as the reference choice for MCN software development and extensions.
OpenFlow Control Adaptor	OpenStack Neutron Plugin API and OpenFlow Plugin	The OpenFlow plugin must be chosen according to the specific OFC to be adopted. The ML2 plugin enables the interaction between Neutron and OpenDaylight.
OpenFlow Controller	Trema	The selection of the reference controller will take into account multiple criteria, like the supported OpenFlow version, the maturity of the code, and the possibility to easily implement the required extensions and functionalities. OpenDaylight has been used as reference platform for MCN developments. It supports both OpenFlow and OVSDB (for the configuration of the OpenVSwitch instances running in the servers) and is characterized by a flexible architecture easy to extend with new functions.
	Floodlight	
	Ryu	
	OpenDaylight	

2.2.3 Inter Cloud Service Provider approach

Next to the intra-cloud networking as described previously there is a need to connect two private networks through VPN in MCN as any MCN service could be deployed at multiple (geographically federated) cloud sites. For maintaining data and control isolation in a multi-tenant environment as cloud, VPN provides an ideal technology solution. In MCN project, the services that require VPN services can be categorized into two classes, where there is a need to create VPN tunnels between service instance components (SICs) which is handled by the service itself natively (example: IMS uses GRE tunnels between SICs to separate different traffic types), and the second class where the services needing VPN setup uses the support VPNaaS. In this second class also lies the scenario where the service instance itself assumes all SICs to be in the same LAN, but has to be split among multiple datacenters for supporting certain SLAs. It is this second class of services that VPNaaS in MCN project caters to.

The support service uses OpenStack's VPN creation and management supported by Neutron since the Havana release of OpenStack. The out-of-the box VPNaaS from Neutron suppots IPSec and SSL VPN tunnels between peer sites.

MCN's VPNaaS support service can easily support point-to-point and point-to-multi-point IPSec VPN connections thereby enabling arbitrary number of site-specific LANs to be inter-connected.

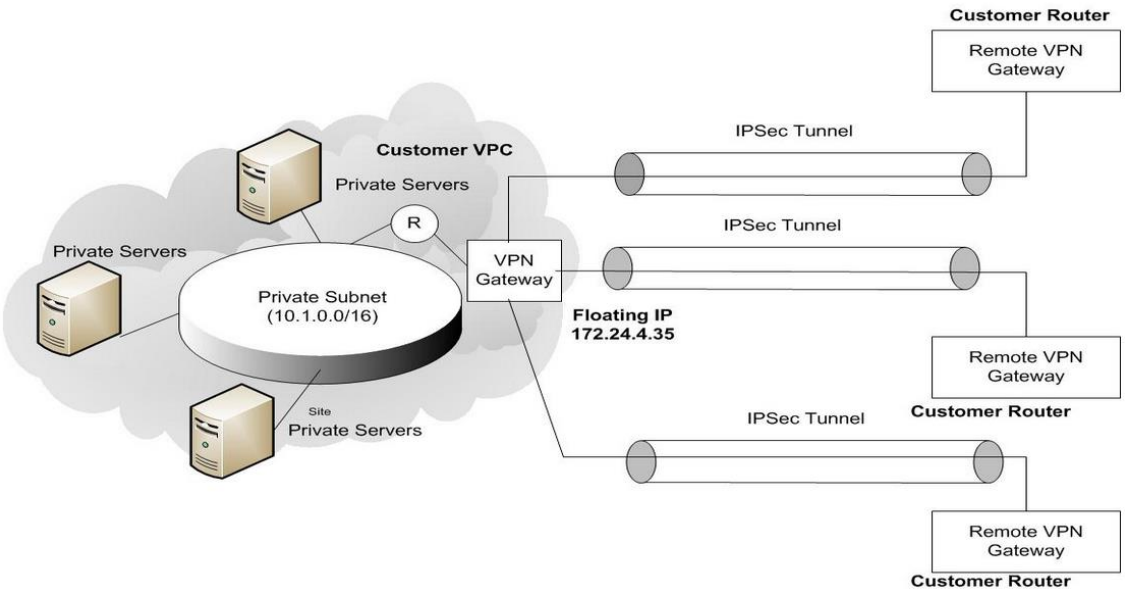


Figure 1 Multiple Connections (source: OpenStack VPNaaS blueprint)

Figure 1 shows a multiple IPSec VPN connections scenario, and below is the explanation of various elements from OpenStack's perspective:

- Private Subnet: Virtual private subnet that is created using Neutron APIs within the user's virtual network.
- VPN Gateway: Virtual router in the tenant virtual network that is connected to the external network in OpenStack network topology, in fact this refers to the public (floating) IP associated with the routers interface link connecting the tenant's network to external network in OpenStack.
- Remote VPN Gateway: is the floating IP of the remote tenant's router's interface connecting that network to the external network.

OpenStack's Heat (CC's deployment module) supports IPSec VPN creation through HOT. For setting up a IPSec VPN tunnels, one needs to correctly configure the IKE-Policy (Internet Key Exchange), IPSec-Policy (Internet Protocol Security), and then a VPN connection which is essentially configuring the OpenSwan VPN tunnel service to be attached to a particular subnet in the tenant's network and with a specific router in that virtual network.

Once these policies are set, then one more step remains, specifying the IPSec Site Connection, which tells the OpenSwan's IPSec driver to initiate the VPN tunnel setup between the two sites, aka the tho Peers. It is imperative that the IPSec and IKE Policies on both ends of the tunnel are exactly the same.

The OpenStack's VPNaaS Neutron add-on works well, and has been tested with OpenSource VPN management suite OpenSwan and is the default choice in MCN testbeds. Other than creation through Heat, IPSec VPNs can be directly created using Neutron CLI and REST APIs.

MCN VPNaaS which leverages the OpenStack's VPNaaS in Neutron was extensively tested for IPSec VPN tunnel creation between two different subnets inside different tenants' networks, one running at

ZHAW Bart testbed and another at ONESOURCE test-site in Portugal. The VMs in one subnet were able to ping and SSH into the VMs running in the remote subnets validating the test case.

2.3 Redirection service

The basic aim of the service is to *efficiently* and *correctly* forward traffic to appropriate resources by taking *dynamic* network view into account and exploiting network programmability features available based on SDN/OpenFlow principles. Prior work within MCN/WP3 includes two contributions (D3.2 2014) and (D3.3 2014) in this context. However, this contribution outlines the experiences and lessons learned within the context of the proposed service:

- The service primarily relies on the OpenFlow wire protocol that offers soft state about the installed forwarding rules (data plane) in the programmable switches. The state not only includes the forwarding behaviour but also encapsulates the network usage in terms of *OpenFlow based statistics* that are related to each flow entry. Further, such statistics are aggregated at different granularities to have more robust visibility about the *path state* between any two given points in the network.
- For facilitating any decision making at the higher layers (e.g. MCN proposed architecture), it is important to have *network based usage metrics* available to the higher layers. Such information could be exposed to higher layers (e.g. service specific or some general orchestrator etc.) through the North-bound Interface of the SDN/OpenFlow controller. For the MCN architecture, the "Monitoring as a Service" is a good candidate for keeping the network based metrics of interest.
- Further, the service proposes to have a "*SDN based Congestion Manager*" that exploits the OpenFlow based southbound interface of the programmable network. The Congestion Manager is located on the control channel between the controller and the OpenFlow based switch and exploits flow statistics for establishing the congestion state of the port or a switch. Such state could be combined with a certain traffic class, resource location etc. for selecting a better network path within or across data center domains. Given the path state within or across domain, global policies within or across domains could be applied based on QoS requirements, traffic type, resource type etc.
- The service is envisioned as a control plane application running within a logical SDN environment and may require other control plane applications like network topology, path computation etc. from the underlying network. Given the protocol split between the control and data plane within SDN/OpenFlow based networks, the *cloud dimension* adds additional scaling requirements that requires that the network scales with growing demands of the tenant traffic.

In short, the first contribution (D3.2 2014) primarily covers the conceptual work with both high and low level architectural descriptions for the Re-Direction Service as well as global policies for managing traffic across domains. However, the later contribution (D3.3 2014) specifically outlines the "*SDN based Congestion Manager*", along with related work, proof of concept implementation and few experimental results are also documented.

2.4 Extensions and considerations

Taking into account the requirements identified so far with respect to networking aspects, this section explains how OCCI and OpenStack can cope with it. Extensions to both are required.

2.4.1 OCCI Extensions

OpenStack neutron has an API which allows for controlling basic networking resources such as ports, networks and subnets. However calling these APIs directly from SO instances which are supposed to manage these resources would lead to a lock-in to a particular technology. To prevent this the existing standards implementation OCCI for OpenStack has been enhanced to also allow for control of these networking resources. This is especially helpful for future developments of services running on the infrastructure foundation

Now through a standard compliant cloud language the SDN resources of an OpenStack cloud can be controlled. This prevents lock-in and by using OCCI makes sure there is one consistent language to carry out most of the control & management calls in the MCN framework.

2.4.2 OpenStack Extension

OpenStack has been extended to deliver virtual network infrastructures with QoS features. These extensions have involved both Heat and Neutron and will be further detailed below, with the definition of new Neutron resources for QoS parameters and classifiers and their integration in the Heat templates as shown in (D3.2 2014). Since the network configuration is performed through the OpenDaylight controller, the Neutron ML2 plugin has been also modified to manage these new resources and its OpenDaylight driver implements the extended APIs with the Neutron service running on the controller side.

2.4.2.1 Traffic Steering

In D3.3 (D3.3 2014) the traffic steering extension implementation in OpenDaylight is described. This section refers to the traffic steering extension implementation in OpenStack.

The plugin structure is somewhat alike to ML2 and Group Policy plugins by including a steering manager, context objects, driver API, a dummy driver and an OpenDaylight driver. Like other Neutron plugins, administrator users will have to enable this plugin and the respective driver in Neutron configuration file. This extension has been confirmed to work in OpenStack Icehouse version.

The traffic steering extension in OpenStack consists of two concepts:

- Traffic Classification - a policy for matching packets, e.g. HTTP traffic, that is used for the identification of the appropriate actions to apply to the packets. It can be for example an explicit forwarding entry in a network device that forwards packets from one address, identified for example by an IP or MAC, into the Service Function Chain;
- Traffic Steering - ability to manipulate the route of traffic, i.e. delivering packets from one point to another, at the granularity of subscriber and traffic types. Neither the actual network topology nor the overlay transports are modified to accomplish this.

The manipulation of traffic redirection occurs at the port level. For example, all HTTP traffic coming from a VM interface is directed to another VM interface instead of going to the network gateway. With this in mind two new resources were added to Neutron:

- Steering Classifier - traffic classification which supports the following filters: protocol, source/destination MAC address, source/destination IP address and source/destination port range;
- Port-Chain - sequence of traffic redirections. This is done with a dictionary of lists of Neutron ports where the dictionary keys are the Neutron port UUIDS. Ingress traffic from these ports is steered to all ports in the dictionary value (a list of Neutron ports) according to a classification criterion.

The Traffic Steering API provides a flag that alerts in situations where there is more than one path from a source to the same destination or there is a path that can form a loop. Note that when redirecting traffic intended for the network gateway to another VM in OpenStack, it is necessary to manipulate the packets so that the VM can process the packet. More specifically, the destination MAC address has to be replaced with the MAC address of the VM interface. The rule to perform this action is automatically inserted in Open vSwitch by the OpenDaylight module.

The Neutron traffic steering plugin extends the Neutron database and adds two tables to store the steering classifiers and the port chains. Similar to other Neutron plugins, the traffic steering functionality is exposed in Neutron python client, Neutron command-line client and Neutron REST API. See Table 3 for a list of available operations.

Table 3 Traffic Steering available operations

Command line	URI	HTTP Verb	Description
steering-classifier-create	/classifiers	POST	Create a traffic steering classifier
steering-classifier-delete	/classifier/{id}	DELETE	Delete a given classifier
steering-classifier-list	/classifiers	GET	List traffic steering classifiers that belong to a given tenant
steering-classifier-show	/classifier/{id}	GET	Show information of a given classifier
steering-classifier-update	/classifier/{id}	PUT	Update a given classifier
port-chain-create	/port_chains	POST	Create a port chain
port-chain-delete	/port_chain/{id}	DELETE	Delete a port chain
port-chain-list	/port_chains	GET	List port chains that belong to a given tenant
port-chain-show	/port_chain/{id}	GET	Show information of a given port chain
port-chain-update	/port_chain/{id}	PUT	Update a port chain

2.5 Integration with other tasks

The following section details the relationship from this task to other tasks. For task 3.1 this is mainly towards task 3.3. Task 3.2 will provide inputs towards this task, and this will be detailed in later sections.

2.5.1 Monitoring of data center internal network and connectivity services

The OpenStack Ceilometer component is the entity in charge of collecting monitoring information about the overall Data Centre infrastructure and the services running on top of it. At the network layer, the OpenDaylight controller constitutes the centralized entity which retrieves and stores parameters about the network topology, status and performance. These parameters can be re-used at the upper infrastructure layers, for example to take decisions about VMs placement in OpenStack, but also at the service layers, for example to detect failures in the end-to-end service chains or breaches in the SLAs and to properly charge the overall service. This brings a requirement to collect network topology and monitoring information and distribute them in the MCN framework towards OpenStack and, more in general, among the different components which compose the MCN service instances. Following the common approach used in MCN architecture, the delivery of monitoring information across services is performed by the MaaS service (Task 3.3), which needs to be fed with the monitoring data generated at the SDN controller. Ceilometer acts as mediator for this exchange of monitoring information, implementing an OpenDaylight client which consumes the statistics and topology data from the SDN controller and forwards the most relevant to the correct MaaS instance, on a per-tenant basis. In particular, Ceilometer is extended with a REST client that uses the OpenDaylight REST APIs to request data about hosts and network nodes, topologies and OpenFlow statistics, for tables, ports and flows.

2.5.2 Quality assessment of OpenFlow-based measurements

The research with regards to the quality of OpenFlow measurements has been continued and expanded. In this research, we assess the accuracy and correctness of values that are available in OpenFlow switches. These values, e.g. the number of packets per flow, or the number of bytes per flow, provide valuable information enabling bandwidth estimation in all parts of a network. Inaccuracies however lead directly to wasting resources, or reduced quality of service, and therefore possible financial losses. Our first findings were already presented in the INDIS workshop at SuperComputing 2014 (Schmidt et al. 2014) and at *Research on Network* (SURFnet n.d.) meetings at the Dutch national research and educational network provider SURF, where new relations enabled expanding the research in two ways: firstly, we acquired access to new testbeds, enabling use to assess hardware of different vendors offering equipment for SDN solutions. Secondly, we designed new experiments to assess more aspects for correctness, including ICMP, IPv6 and ICMP6, as these are fundamental in both network functioning and network health.

The new testbeds are comprised of hardware from Brocade, Juniper and HP. Also new Pica8 hardware (which was used in the initial research) will be subject of the new experiments. As the initial research showed inaccuracies severely impairing the use of any OpenFlow-based measurements, several methods aiming at pinpointing the source of these inaccuracies were designed. This way, we intend to supply constructive to and useful feedback for vendors, improving and enabling production SDN networks to use OpenFlow-based measurements for link dimensioning, autonomous scaling and traffic engineering, and so forth.

Initial results of our expanded research show different types of problems with the different appliances, making heterogenous (in terms of vendors) networks a challenge per se. Our final results will be submitted to the special issue of the Journal on Selected Areas of Communications (JSAC) on "Measuring and Troubleshooting the Internet: Algorithms, Tools and Applications"(IEEE JSAC n.d.).

2.5.3 Integration with the Cloud Controller

In case that SO instances want to manipulate running virtual networking instances the OCCI interface on top of OpenStack neutron provides a great way of doing that. For the instantiations of virtual resources the CloudController plays a big role. Especially requirements on the virtual resources, such as QoS parameters, needed for SICs need to be defined in the ITG. Especially for the integration of ensured QoS parameters this integration has been show and defined in (D3.1 2013).

2.6 Conclusions

This task has helped establish the SDN enabled basic network functions needed in MCN. This reaches from the basic features used within the data center to across data center. We refer to them to Inter and Intra cloud network management. For both technologies have been evaluated and extended where needed. Especially the work on traffic steering and support for ensured QoS with the help of OpenDayLight are examples for this. Also the FMC concept and the Re-direction service described, build upon available technology but show with help of specific use cases the need to extending and enhancing them.

Next to the basic networking functions the tasks also provided support services which are network related. Especially the DNS service had to be implemented as the basic features provided by e.g. OpenStack did not sufficient the requirements. Details on the DNS service are reported upon in section 7.1.

Another key contribution was to provide a way to generically manage network resources through standardized interface. This has been achieved by using the OCCI standard implementation for OpenStack neutron.

3 Real-time Performance of Infrastructure Resource Management Frameworks

The following sections describe the contributions from task 3.2, entitled "Real-time Performance of Infrastructure Resource Management Frameworks".

3.1 Introduction

In the previous deliverable (D3.3 2014) we continued to focus on the performance benchmarking of MCN services and presented further workload specific optimisations. However, at that time of writing the deliverable only RANaaS had been thoroughly performance tested as a result of testing conducted UBurn and by Eurocom on Open Air Interface (see section 6.3). While we did provide a progress report on the other core services and a small selection of support services, performance benchmarking was mainly in a stage of planning and therefore very little performance analysis had been conducted. Consequently, the Year 2 review alluded to this, making it clear that some resources must be reallocated to increase the depth of the real-time performance testing and that a deeper analysis must be made. In general terms, it became clear that we must attempt to quantify the impact of virtualization and cloudification on real-time services. In the following subsections will outline how as a consortium we reacted by making top priority the deployment of services onto different virtualisation technologies to enable more in-depth performance analysis of services.

Firstly, services have been required to deploy on to different available testbeds (Intel, Bart, UBern, CloudSigma and Nano DC) to establish a final demonstrator, comprised of multiple clouds and multiple virtualization technologies, working together to provide one integrated end-to-end solution. This has been the subject of ongoing work coordinated by WP6 and due to be documented in D6.4. This has enabled service developers to perform in-depth performance analysis, with particular emphasis on scalability, self-adaption, dynamicity and reliability. This has been documented in detail in current deliverables such as (D4.5 2014), (D5.4, 2015) and in this deliverable. Task 3.2 is mainly concerned with the impact of virtualization and cloudification on the performance of MCN services. Further evaluations will be made and documented in D6.4.

3.2 Performance analysis of MCN services and additional recommendations

Performance testing of MCN services and their SICs has shown how different their requirements of computing infrastructure are. The following paragraphs summarise the performance testing results of these services.

3.2.1 Performance methodology in use

Recommendations documented in (D3.2 2014) with regard to performance benchmarking methodology has prompted service developers to utilize existing tools, scripts, profilers, performance monitoring tools, documentation and research papers in order to devise repeatable performance tests. When appropriate, these have been integrated into Jenkins jobs in order to profile the services when run on different testbeds. The results of this testing have been documented in the current run of deliverables such as (D4.5 2014), (D5.4, 2015) as well as this deliverable. Summaries have been provided in the following subsections, and have already prompted further workload-specific optimizations based on the performance requirements of each MCN Service. This will also preempt

the development of real-time performance dashboards with the ability to expose QoS breaches that impact on SLAs.

3.2.2 The impact of virtualization and cloudification on MCN services

IMSaaS has illustrated time and again that the main reason why the service fails is due to RAM exhaustion. The availability of additional CPU cores has no impact on whether or not the service fails if the allocated memory remains constant. As such, VM instances with a high amount of RAM in relation to CPU cores are suitable to this service. Tests on physical and on virtualised infrastructure reveals that allocating multiple CPUs to SCSCF has no impact on whether or not this SIC crashes. IMS and especially specific SICs within the service are far more memory sensitive than they are CPU resource hungry. This is the opposite of other services, for example SLAaaS. IMS has shown to process calls well within the carrier grade voice QoS tier (under 150ms) and as such the service can successfully be deployed or migrated to a geographically distant data center not necessarily close to other core services, such as RAN or EPC. SLAaaS has been shown to have an extremely low memory footprint and is more dependent on CPU processing capacity. This is the polar opposite of for example IMSaaS. Here, a low amount of RAM is best paired with more CPU cores.

The chart in Figure 2 illustrates how an increase of calls/s or other QoS defining KPIs map to service instances hence VMs.

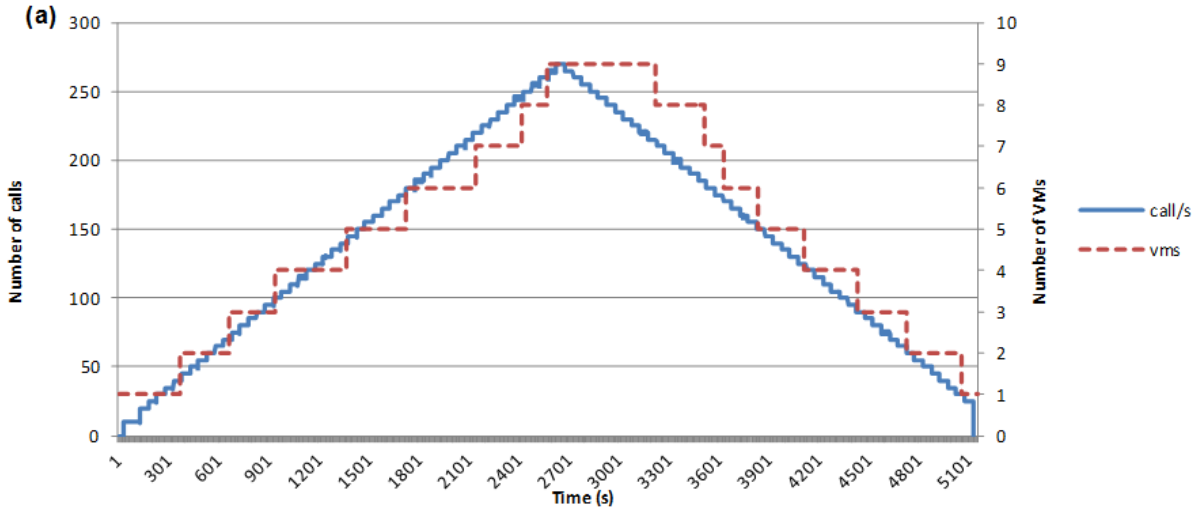


Figure 2 How an increase of calls/s or other QoS defining KPIs map to service instances / VMs

However, such a mapping can be subject to deviation depending on cloud and virtualisation technology specifics. If, for example, performance testing establishes a service such as IMS can successfully host 80 calls/s on a given processor architecture and frequency before starting to fail, this may not correlate directly to the same workload being deployed on another generation of processor. As such, each unit of computation should bear into consideration such computational capacity differences.

One such technology which aims to solve this issue is the 'Service Assurance Administrator, or Intel DCM: SAA'. As part of Intel DCM: SAA, Intel introduced a feature called the Service Compute Unit, which specifies the target performance for a particular application. Service Compute Units can be used to provide the right resources, or to redeploy workloads impacted by noisy neighbors to areas with less contention. Compute capacity varies between processor manufacturers and generations, and SAA

provides a reliable mapping between these differences. In the case of noisy neighbours, it can act to flag such service instances, which encroach on the SCU's processing capacity and thus provide the cloud stack the visibility and option to perform a service migration to a less congested host. SAA is implemented as a Linux KVM, or kernel-based virtual machine, and has a plug-in to OpenStack. Using SCU as a capacity management tool, rather than over-provisioning servers to anticipate VM bursting performance requirements can also increase server VM density. In the context of MCN, SAA could be used to provide consistent processing capacity to services with accurate mapping between for example known IMSaaS calls/second resource blocks and VM processing capacity, irrespective of the underlying physical infrastructure. Such a technology solution could thus greatly alleviate the ambiguity of requesting and receiving guaranteed cloud compute capacity across clouds and across cloud / infrastructure generations. DCM: SAA should be part of the SLAaaS framework, where supported by cloud stacks.

ICNaaS illustrates that virtualisation of the service has little impact on its performance. There is a virtualisation overhead, but it is not significant and the service can be considered for deployment on public or private cloud infrastructure. However, non-cached content does tax the underlying storage infrastructure significantly, to the point where this becomes a bottleneck. One solution to ensure high performance storage infrastructure is to use SSD block storage as the infrastructure foundations, ensuring as high an IO throughput as possible. When this is configured in a striped RAID configuration, performance increases further and is a must in a shared multi-tenant environment. CloudSigma currently uses RAID-Z SSD striping in one physical storage host, which goes a long way to improving I/O in a multi-tenant environment. This solution will imminently be superseded with triple redundancy persistence, which will further stripe the data across 3 or more physical storage hosts. This has been shown to improve throughput by a factor of 3 and significantly increases tolerance to simultaneous reads and writes to the drives. What these storage strategies currently achieve is far superior non-cached file I/O performance, where the bottleneck is not CPU performance, but rather file I/O, as is the case in non-cached ICNaaS.

The Follow-Me-Cloud concept has been extensively tested with multiple scenarios and multiple content cloning methodologies. With Knapsack, MeTHODICAL, DiA, TOPSIS being evaluated with different cache sizes (256 / 512 / 1024 / 20148 / 4096 MB), Knapsack and especially MeTHODICAL delivering consistently superior prediction of content demand at each layer of caching.

DSSaaS also shows very similar performance whether virtualised and running under KVM, or or bare metal. One worrying factor however which should be the subject of further optimisation is the provisioning time of new service instances. Scaling up typically takes in excess of 8 minutes, which although is an improvement over fixed infrastructure, is still an unacceptably long overhead for time-sensitive high-value content, for example the streaming of Pay-Per-View events. The process in bringing up a new service instance should be closer examined, the processes with the longest turnaround times identified and steps taken in minimising these lead times. Performance issues with running instances of cloudified DSSaaS are more than satisfactory. The only issues lie in the time taken to scale out new service instances.

This is in great contrast to the process of provisioning new instances of e.g. ICNaaS, which takes 85 seconds to instantiate a new service instance. The underlying cloud stack should also be examined to gauge whether or not a significant amount of time is spent waiting for the stack to provision a VM as, for example, CloudSigma's stack takes seconds to bring up a new VM.

3.2.3 Summary of infrastructure requirements

It has been shown that the virtualisation of MCN services can indeed be successfully completed and in most cases, each service when virtualised offers similar performance to that of a non-virtualised service. This is true for services which do not have extremely low-latency real-time performance requirements, such as RAN. Having reviewed the performance test results of all the tested MCN services, it can be noted that the performance difference between virtualised and non-virtualised service deployments is in most cases negligible, yet the virtualised instances offer significant benefits, such as on-demand scaling, service migration and increased service density on cloud stacks which are flexible enough to offer such service deployments. RANaaS has shown that it can be successfully deployed on bare-metal and virtualised using RT-Linux and container environments using current generation GPPs. It has also been shown that the processing times half with the introduction of each generation of CPU architecture, especially with each generation of CPU Multimedia instruction sets.

CloudSigma's stack offers the individual specification of compute resources on a per-VM and hence per-MCN service workload - CPU, RAM and storage. This solution can offer far higher VM provisioning density on an existing host. CloudSigma do as of late provide their proprietary KVM-based cloud stack as-a-service, offering these benefits to third party infrastructure owners. This KVM based cloud stack can be configured for oversubscription multipliers to be turned down or even disabled, ensuring performance equal to that of the performance tests conducted within (D6.4, 2014). Extremely latency sensitive services such as RAN are not suitable for public cloud with its variable and on occasion oversubscribed performance. It was however shown that OAI operates within acceptable limits when RT-kernels and hypervisors are deployed at every layer of the cloud stack. Optimal performance was however obtained when the service was deployed within containers, as this offers performance very close to that of bare metal.

CloudSigma has since (D3.3, 2014)'s publication upgraded its in-house developed SSD-based storage solution to a distributed SSD and HDD-based multiple-node storage solution, thus offering pooled throughput and IOPS performance. The solution makes use of RDMA RoCE Ethernet connectivity, offering significantly lower latency compared to standard Ethernet. Early internal tests show a performance improvement of a factor of 3 both in both throughput and IOPS performance, with the added benefit of storage high availability. This performance increase is superior to even that of SSD-only performance, which in turn far surpasses HDD-based cloud performance.

3.3 Integration with other tasks

The previous sections have already outlined the feedback loop between Task 3.2 and the MCN services being developed by other tasks. The following sub section will further detailed interactions.

3.3.1 Integration with Monitoring-as-a-Service

The following subsection outlines the interaction between Task 3.2 and Task 3.3 by providing inputs related to the performance benchmarking of the Moniring service in MCN. This evaluation is detailed in section A.2.

3.3.2 The impact of performance testing on service deployment

Having completed IaaS performance evaluation of services, it has become clear that different virtualisation technologies offer different typical performance levels and different clouds offer substantially different VM provisioning times. This should be considered when orchestrating MCN

service deployment and migration. For most services with non-realtime performance requirements, the IaaS load differences during workload execution are not significant to warrant further action being taken. However, it must be noted that in some cases service instance provisioning times can be excessive, for example 8 minutes for DSSaaS on OpenStack-based KVM clouds.

3.3.3 Selecting performance relevant features

These findings should be taken into consideration and preferably fed back to end-to-end service designers within the ITG editor (see section 5.2.4.3) using programmatic access to AaaS (see section 7.3). Typical static service performance values based on the conducted performance benchmarks should serve as a foundation for the generation of service profiles on specific and best fit-for-purpose infrastructure, quantifying service performance within service units and ideally, mapping existing hardware-based services to their cloud-based equivalents. These 'typical' service units should not however remain static and should constantly be updated using quantitative MaaS data, further refined by AaaS and bonded to contracts using SLAaaS to provide relevant and trustworthy tactical data and service performance guarantees to service deployers and EEUs. Cutting edge technology such as Intel's SAA will go a long way towards ensuring tolerable variance in IaaS performance.

Each service instance can specify in quantifiable units the SLOs they expect of the infrastructure used to host the service. The mechanism to facilitate the pairing of infrastructure and service SLAs is SLAaaS, using the service's OCCI-compliant JSON SLA template format.

Intel's DCM: SAA technology enables the categorisation and accurate quantification within predictable set boundaries of compute performance service levels, while Intel Trusted Execution Technology (TXT) ensures the technology stack is what it claims to be and what has been requested by the service. These two technologies when combined enable SLAaaS to more reliably pair infrastructure to services both in terms of required compute resources and verified infrastructure.

SLA templates can and must be written for each service, specifying the relevant services's QoS requirements of both the underlying infrastructure, and of dependent services. SLAaaS can then request resources matching or exceeding these metrics, subsequently enforcing their validity using hardware / stack technologies such as Intel DCM: SAA.

The infrastructure SLAs can either be manually provided by infrastructure providers or generated automatically using a combination of Intel SAA technology together with supporting software technologies, such as the SAA plugins for Openstack.

If service QoS is to be expected, this SLA template should then be referenced in the respective service's HEAT ITG, with SLAaaS verifying that the appropriate infrastructure matches these requirements prior to the stack being deployed, as well as policing the infrastructure and service performance at runtime.

3.4 Conclusions

Extensive performance testing has been conducted of MCN services on a variety of infrastructure foundations and virtualisation technologies. Both Intel and AMD compute performance testing has been conducted on KVM-based Type 2 virtualisation, on both open source-based academic as well as commercial, production grade proprietary cloud stacks, namely on ZHAW's OpenStack for the former and CloudSigma's own cloud being responsible for the latter. Container-based virtualisation performance testing has been conducted for RANaaS, which has been shown to offer performance

similar to that of bare-metal service instance deployment. RANaaS, due to its RT requirements best illustrates the large latency discrepancies between bare-metal / container virtualisation and that of KVM, especially when KVM virtualisation is performed on a public cloud without RT extensions being enabled. DSSaaS has identified the long service instance provisioning times typical of OpenStack clouds, something that can be vastly improved upon by using e.g. CloudSigma's KVM-based cloud or container-based virtualisation.

To date, performance testing has focused primarily on quantifying the performance characteristics of the different available IaaS offerings, with a particular interest in comparing virtualisation approaches and identifying performance optimisations of the underlying infrastructure to match the workload requirements. With IaaS benchmarking of services complete, the attention of service owners will shift to focus on data plane performance evaluation, which will be presented in (D6.5 2015).

4 Common Monitoring Management System

The following sections describe the contributions from task 3.3, entitled "Common Monitoring Management System".

4.1 Introduction

The following section describes the contributions of task 3.3 for the key findings and developments within the time range between M13 and M30. The concept of the "Common Monitoring Management System" specified until M12 by Task 3.3 has been realized from M13 on and made available (in various versions) to the consortium as Open Source Monitoring-as-a-Service/MaaS for early integration.

This task on monitoring focuses on the design, implementation and test of monitoring mechanisms, from the low-level resources to the high-level services, across the four different domains: radio access network, mobile core network, cloud data center and application domain.

It is of crucial importance in cloud environments to measure the complete infrastructure in order to directly react to changes in the system load. Therefore monitoring of heterogeneous environments which might vary in their topology during runtime and measurements of various metrics are the key requirements for virtualized environments. As the state-of-the-art section in D3.1 outlined, there is (still) no common monitoring tool available for supporting all required metrics which have been collected in the first phase by the MCN project. Today's available monitoring systems are missing: functionality for on-demand monitoring and lifecycle support. This task realized a solution, which addresses those missing features.

4.1.1 Problem Statement

This section maps the main objectives for task 3.3 (DoW 2012) against the achievements up to M30.

- **Obj.1:** Firstly, "monitoring should provide metrics for assuring the health and the performance of the MobileCloud infrastructure."

The implemented monitoring system supports the metrics required by the project consortium. Therefore we distinguish between a) standard cloud metrics for measuring software and hardware metrics (CPU, RAM, storage, network, etc.) and b) user-defined metrics individually designed for each MCN Service (e.g. EPC: number of switches on the data path, HSS: Number of active subscriber, RAN: Number of attached users per eNB). Both types of metrics are used by MCN Services to assure health and performance. A list of the (around 30 different) user-defined metrics can be found here [1].

- **Obj.2:** Secondly, "monitoring should support the creation of metric and context-based information (notifications/events) to be used to support adaptive life-cycle functionality […]", which was identified within the first year of the MCN project by using a questionnaire among the MCN Services.

[1] Available User-defined Metrics, https://wiki.mobile-cloud-networking.eu/wiki/Available_User-defined_Metrics

A support for the creation of metrics was solved by providing a comprehensive tutorial provided to the consortium available in the MCN git repository [2]. The tutorial explains with examples how to create user-defined metrics in MaaS and provides an early MaaS VM including example metrics for testing for the integration with other MCN Services.

In addition the support of asynchronous notifications of metrics was realized. Therefore a metric has to be selected and a thresholds criteria needs to be defined within MaaS. An asynchronous notification is returned, in case the threshold criteria has been met.

One example within MCN is the adaptive lifecycle functionality within EPCaaS: Scalability on the data plane is achieved by adding/removing switches from the data plane through logic managing the scalability within the service orchestrator, which subscribes to metrics on monitoring data for number of subscriber of each switch.

- **Obj.3:** "Essential and fundamental intelligent fault tolerance mechanisms […]" will be included in the design of the monitoring system. The support of several metrics by the monitoring system requires "domain specific logic" (e.g. domain logic required in task 3.5, task 4.3, task 5.1 or task 5.2).

Fault tolerance mechanisms of MaaS: An analysis of the critical factors for MaaS pointed out two aspects: service and data availability. Firstly the core service (CMMS) relies on the hardened Open Source toolkit Zabbix, which has been evolved over time to a mature, stable and well-accepted software product. Secondly the database of MaaS has been identified as the only possible performance bottleneck of MaaS. The MaaS internal can be switched from MYSQL to DBaaS, which uses redundant cloud storage and in turn enables high availability what in turn solves the performance challenge in the database. Finally, essential and fundamental intelligent fault tolerance mechanisms have been taken into the design process of CMMS/MaaS, but the design and implementation of a high available monitoring system was out of scope of the MCN project.

Fault tolerance mechanisms provided by MaaS have successfully been taken up by other MCN Services to increase the level of availability: In particular the usage of metrics to support fault tolerance and scalability of MCN Services has been realized. The scalability of the HSS is one example used within EPCaaS and IMSaaS.

- **Obj.4:** In turn, the monitoring system is able to extract monitoring data from different sources and exposes them over a unified interface towards external MCN components. Interfaces between the monitoring system and other MCN services components will be specified, defined and integrated.

CMMS and MaaS have been realized to support standardized, well defined and MCN internal agreed interfaces. Standard compliance has also been ensured through this approach while integrating one OpenStack monitoring system Ceilometer into MaaS (namely the Zabbix-Ceilometer-Proxy / ZCP).

General high-level, detailed functional as well as non-functional and service specific requirements [3] have been identified out of the MCN end-to-end use case scenario together with a questionnaire. A set of metrics have been derived out of those requirements. (Service specific requirements have been gathered in the beginning of the project through an interview questionnaire).

---

[2] MaaS create own userparameter, https://git.mobile-cloud-networking.eu/monitoring/MaaS/blob/master/doc/userparam.md

[3] Requirements for CMMS, https://wiki.mobile-cloud-networking.eu/wiki/Requirements_for_CMMS

4.2 Monitoring-as-a-Service

The following sections describe the full-stack Monitoring service in a bit more detail.

4.2.1 General concepts

This section outlines the CMMS in a general overview, architecture and reference points and the evolution of MaaS over the project lifetime and release cycles.

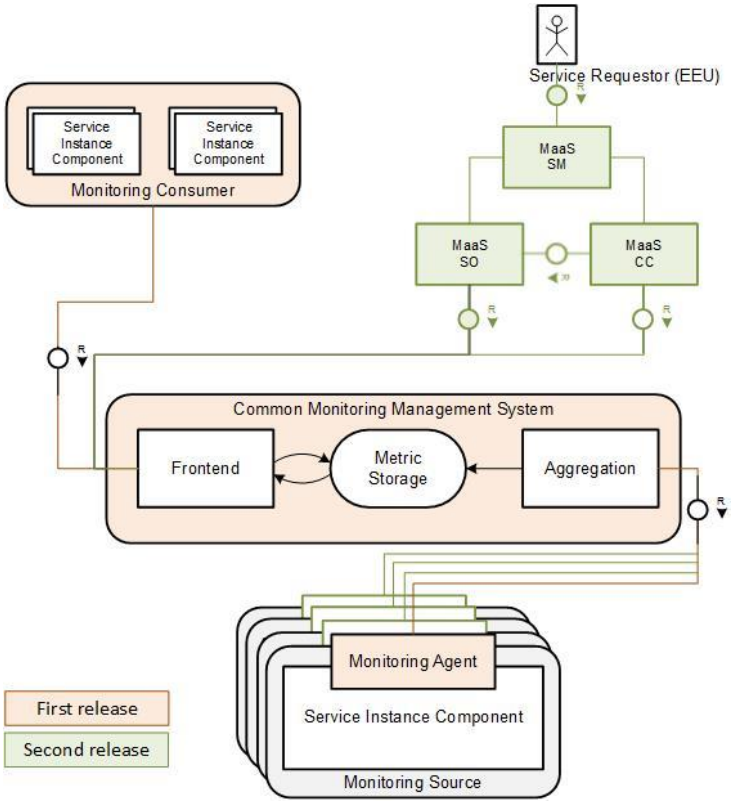


Figure 3 General concept of monitoring and relations

Figure 3 depicts the evolution of MaaS in two main release phases. The first release includes an early version, which already provided the main interfaces but was instantiated manually. Only basic metrics have been supported in the first version. The second phase included already more metrics, has been enhanced through SM, SO as well as CC/SDK features for MaaS and provided a larger set of functionalities towards the monitoring consumer.

4.2.2 Functional Elements

The following section holds the implementation details of the functional elements.

4.2.2.1 Non-Service Specific

This section describes the implemented functionalities of the common functional elements with reference to the CMMS. It should be noted that the following elements are not part of the CMMS service itself, but are using interfaces for communicating with the CMMS.

Monitoring Consumer: Monitoring Consumers are defined as the entities that use the Frontend interface to retrieve the metrics provided by the CMMS. We consider MCN Service that query MaaS

for monitoring data as Monitoring Consumer. Famous representatives are Analytics-as-a-Service, Mobility-Prediction-as-a-Service and Rating-Charging-Billing-as-a-Service.

Monitoring Agents: Monitoring Agents are defined as the entities that submit metrics to the full-stack MaaS. This subsection outlines some of the available [4] user-defined metrics that have been developed within this task and are MCN Service specific.

Userparameter[5]	Metric	Functionality
attached.subs.gw	Active overall subscribers of the Gateway	Measures number of all subscribers to all switches connected to the gateway
attached.switch.gw	Number of active switches of the Gateway	Measures number of all attached switches to the gateway
attached.subs.of.switch	Active subscribers to certain switch	Measures active subscribers per specific switch
attached.switch.discovery	Active switches of the Gateway	Adds and deletes scaling switches automatically

The MCN approach allows a dynamic instantiation of services in a virtualized environment. All these services as well as the infrastructure need to be monitored. New deployed and distributed services need to be connected with the centralized monitoring system in a smart and efficient way. As an example, the EPC service has to support autoscale under load and with predefined thresholds. Therefore Zabbix low-level discovery is used to quickly detect currently running switches and pair them with their attached subscribers. This can be used as an example to provide easy to configure and integrated monitoring for scaling devices, nodes and other MCN related resources.

4.2.2.2 Service Specific

In the MCN framework, MaaS communicates with several services that act as Monitoring Consumer. In the simplest interaction mode, these services retrieve monitoring information in polling mode, using the get methods of the Zabbix APIs (see section 4.2.3 for the details of the interfaces). The exchange of monitoring information follows the subscription – notification model, where the consumer services can be instantiated and, once running, is able to dynamically subscribe or unsubscribe to receive asynchronous data. In this case, MaaS (through its Zabbix component) triggers the generation of events based on the monitoring information collected through its agents and, as automated reaction, sends unsolicited messages towards the consumers with active subscriptions. However, each Monitoring Consumer service may implement its own interfaces and thus require the instantiation and activation of service-specific components at the MaaS level which implement the clients to enable this interaction.

For example, the RCBaaS is a Monitoring Consumer of monitoring data which are collected for accounting and billing purposes: RCBaaS requires information about anomalous events (e.g. service

[4] Available User-defined Metrics, https://wiki.mobile-cloud-networking.eu/wiki/Available_User-defined_Metrics
[5] Some of these metrics are potentially billable through the RCB service.

failures, consumptions exceeding a given threshold, etc) in order to correctly enforce different types of charging models. The RCBaaS interacts with the rest of the services in the MCN environment through a Rabbit Message Queue (Rabbit MQ) used to collect asynchronous messages generated from a variety of entities. For this reason, when RCBaaS subscribes to the MaaS for receiving monitoring data, MaaS instantiates a new sub-component to mediate the interaction towards RCBaaS. This component is a Rabbit MQ client which is triggered by Zabbix whenever the MaaS system detects at least one condition specified in the subscription request. The Rabbit MQ client receives as input the relevant monitoring values from Zabbix, translates them in the RCB format and delivers the messages to the RCB RabbitMQ.

4.2.3 Reference Points

This section identifies the reference points, which enable the interaction between the CMMS core and the other MCN platform components and services (see Table 4). Table 5 outlines a brief functional description for each functional block part of MaaS or interconnected with MaaS over interfaces. For each reference point, the list of the main primitives is provided in Table 5.

Table 4 CMMS Functional Block

Reference Point	Functional Definition	CMMS Component	External entity
Monitoring::CMMS.SM	The MCN EEU requests or disposes MaaS.	Frontend	MCN EEU
Monitoring::CMMS.SO	Provides means to: obtain the list of metrics available from the CMMS; obtain the running configuration from the CMMS (e.g.: obtain the endpoint of the Aggregation component); configure the CMMS.	Frontend	Maas CC
Monitoring::CMMS.CC	Instantiates the CMMS.SO and deploys the CMMS	Frontend	MaaS SO
Monitoring::CMMS.SI.Consumer	Provides the delivery of monitoring data to the monitoring consumer.	Frontend	Monitoring Consumer
Monitoring::CMMS.SI.Aggregator	Provides: means to register new types of metrics (e.g. configure data formats, delivery systems); the inbox to receive the metrics from the monitoring agents.	Aggregator	Monitoring Agent

Table 5 CMMS Reference Points - primitives

Reference Point	Primitive	Description	Parameters
Monitoring::CMMS.SM	POST	Instantiates a new MaaS object through deploying a	Input: MCN tenant, auth

		new SO which will get called with PUT and POST	token Output: MaaS location uri
	GET	Requests the defined SM attributes. Without location parameter the provided services of the SM will be returned.	Input: MaaS location uri, MCN tenant, auth token Output: MaaS instantiation meta information, referenceable IP
	DELETE	Disposes the deployed topology and the instantiated SO.	Input: MCN tenant, auth token, MaaS location uri
Monitoring::CMMS.SO	PUT	Initializes the SO.	MCN tenant, auth token, OCCI category
	POST	Deploys the defined topology on the configured OpenStack deployment as a new service instance. Can also be used to trigger provisioning or update the SO/service instance.	MCN tenant, auth token, OCCI category
	GET	Returns the state of SO and service instance.	MCN tenant, auth token
	DELETE	Disposes the deployed topology on the configured OpenStack deployment.	MCN tenant, auth token
Monitoring::CMMS.SDK/CC	get_maas	Either requests a new topology deploy from a running MaaS-SM or references an already deployed MaaS in the used tenant.	Output: MaaS object reference
	dispose_maas	Disposes of the deployed service instance and related SO	Input: MaaS object reference
	get_address	Retrieves the Zabbix API IP of the referenced MaaS object.	Output: API IP
	get_metric	Wrapper around the Zabbix	Input:

		API. Extracts the last measured value of the corresponding item.	Hostname, Metric name Output: last measured value
Monitoring::CMMS.SI.Consumer	Get	Logical primitive used by MaaS consumers to request monitoring data in polling mode. This primitive is implemented through a JSON-RPC message (with item.get method) in an HTTP request exchanged over Zabbix API. The associated HTTP response returns the desired values.	Input: Item ID, Filters (option) Output: List of item objects values
	Subscribe	Logical primitive used by MaaS consumers to register to the MaaS service in order to retrieve asynchronous notifications. This primitive is implemented through two JSON-RPC messages sent in HTTP requests over Zabbix APIs. The first message uses a trigger.create method and specifies the condition which generates the asynchronous reaction from the MaaS. The second message uses an action.create method and specifies the originating condition (through the trigger previously created) and the expected reaction.	Trigger.create Input: Expression which describes the triggering condition Output: Trigger ID Action.create Input: Conditions describing the triggering situation (i.e. a set of triggers); Operations to be executed (i.e. the scripts which needs to be launched to activate the suitable clients) Output: Action ID
	Notify	Logical primitive used by the MaaS to send an asynchronous message to a MaaS consumer which has subscribed to receive specific events. The actual message depends on the interfaces exposed by each MaaS consumer (e.g. Rabbit MQ messages for RCB-aaS etc.).	Output: List of item objects values

Moreover, it is shown how a standardized interface, in this case OCCI, can be used to support these reference points as described in the next section.

4.2.3.1 Using standardized interfaces

The interfaces specified and implemented within the task 3.3 on Monitoring have been mainly selected as standardized interfaces to allow stability among different versions as well as to provide backwards compatibility and conformance among interacting instances. The interfaces of MaaS provided in the first version have been designed aligned to the initial requirements and therefore are still used MaaS, but have been extended with functionality. The functionality provided gives easy to use access to request, manage, interact and dispose a MaaS instantiation. The latter is being provided by exposing the accessible IP address of the monitoring system and by providing a simplified method for MCN services to request monitoring-data through the monitoring service.

Monitoring makes use of MCN specific interfaces, which have been realized during the project. The deployment and disposing methods for MaaS are located in the utility module of the SDK, while the pure monitoring related function calls / methods reside in the monitoring module.

The most common MaaS API calls have been abstracted in the SDK and exposed towards the outside. For complex metric queries the current approach should still be to let the partner services talk directly with the well documented and feature rich Zabbix API. A decision has been taken to simplify the most common MaaS API calls, but not all possible calls.

The Zabbix APIs are based on JSON-RPC protocol and allow to retrieve data from Zabbix by specifying some filters or to subscribe to receive a certain type of monitoring information or event descriptions, under a given set of customizable conditions. Through the Zabbix API, different MCN Services can subscribe to receive information using their own interfaces, e.g. through the instantiation and dynamic execution of service-specific plugins at the MaaS level.

4.3 Monitoring as a supporting service

Within the scope of MCN, MaaS has been specified as a supporting service for other MCN Services. This section summarizes the lifecycle management of MaaS (Deployment, provisioning and disposal of MaaS) as well as the level of integration with other MCN Services, tasks and WPs.

4.3.1 Deployment, provisioning and disposal of MaaS

As the development of MaaS is aiming for a tight integration with other MCN Services, the deployment, provisioning and disposal methods have been implemented using SDK calls, mainly the deployment phase. While currently there is only one implementation for a single functionality, this will help to provide easy to maintain portability to different solutions.

The deployment phase implementation is using HEAT on top of OpenStack, which reflects the needs of our current testbed scenario and aimed production deployment. Topologies are expressed as Heat templates, which are being interpreted by HEAT and executed by OpenStacks services. Through using the SDK, the MaaS SO only needs to provide the deployer with the HOT or an updated one, and will only need to call the disposal method once to get to pre-deployment status. The common MCN procedures are followed this way. If other ways of virtualisation would be added at a later point in the project and a different deployer could be used, and only this method could easily be adopted.

The MaaS functionality has been integrated into the SDK with an easy to use and fairly straight forward way of deploying, disposing and accessing MaaS methods in an encapsulated way. The current implementation of MaaS is targeted to the described technologies (Zabbix) but can be extended by implementing the standardized interfaces for another monitoring solution.

4.4 Integration with other tasks

Table 6 outlines the progress on integrating MaaS with other MCN Services. Early phases of the MaaS integration can be found by EPCaaS, IMSaaS and RANaaS.

Table 6 MaaS integration by task

MCN Service (XaaS)	Cross-WP	Integration
EPC service	WP4	M14
IMS service	WP5	M14
RAN service	WP3	M14
RCB service	WP5	M25
DSS service	WP5	M21
DNS service	DNS	M18
AAA service	WP5	M29
SLA service	WP5	M30
MOB service	WP4	M29
ICN service	WP5	M23
CDN service	WP5	Not integrated

4.5 Conclusions

This section outlined the project results of the MCN service and full-stack monitoring solution MaaS within a timeframe of M13 until M30. The initial requirements on MaaS have been met with concept and implementation, which have been presented in (D3.2 2014) and (D3.3 2014).

The very early release of the first MaaS version to the consortium, made an early integration with other MCN Services possible and lead to a large coverage of almost all MCN services until M30.

As part of the future work, mainly extensions of MaaS are planned. This includes adding support for additional metrics which should support the evaluation work.

The two main challenges for MaaS during the project runtime have been the changing environment with multiple OpenStack versions as well as the evolving MCN Services, of which metrics have been extracted.

The stability of OpenStack and the difficult OpenStack upgrading process turned out to be challenging. In addition, the extensive review of available monitoring systems, their status analysis and potential usage in MCN turned out to be challenging and exhausting than expected initially.

This task on Monitoring and MaaS released the following Open Source projects in MCN GitHub repository: ZCP[6], MaaS[7] and SO, SM and CC/SDK[8] extensions.

[6] MaaS Zabbix Ceilometer Proxy - https://git.mobile-cloud-networking.eu/monitoring/MaaS
[7] MaaS VM on MCN OwnCloud - https://owncloud.mobile-cloud-networking.eu/owncloud/public.php?service=files&t=c4228caf3865de1ff589698ca4a803e3
[8] MaaS Extentions for SK/CC - https://git.mobile-cloud-networking.eu/cloudcontroller/mcn_cc

5 The CloudController

The following sections describe the contributions from task 3.4, entitled "CloudController, Algorithms, and Mechanisms for Virtual Infrastructures".

5.1 Introduction

The CloudController (CC) is one of the key components of Infrastructure Management Foundations. It is responsible for supporting the SOs end-to-end requirements and complementing the SOs service life-cycle management needs in the MobileCloud. All services in the MCN project have a relationship towards this entity.

Within the following sections the CC's internal services are described and final findings are reported upon.

The objectives, requirements and problem statement for the CloudController have not changed and remain as stated in (D3.1 2013).

5.1.1 On standards and pragmatism

Wherever possible and needed the OCCI (Nyren et al. 2011) has been used, extended and implemented for RESTful control interface. Next to this standardized approach de-facto standards such as AWS CloudFormation (AWS 2013) and Heat Orchestration Templates (HOT) (Alex Henneveld 2013) have been used.

5.2 CloudController architecture

The CC will support the SO in the end-to-end provisioning and life-cycle management needs of services in MobileCloud. To support the SO in fulfilling this role the CloudController is implemented in a modular way. Within the following sections final reports and decisions for each of the CC's internal components are shown.

5.2.1 Overall Architecture

The overall architecture of the CC has not changed since (D3.1 2013). For completeness we show the conceptual architecture as reported upon earlier in Figure 4.

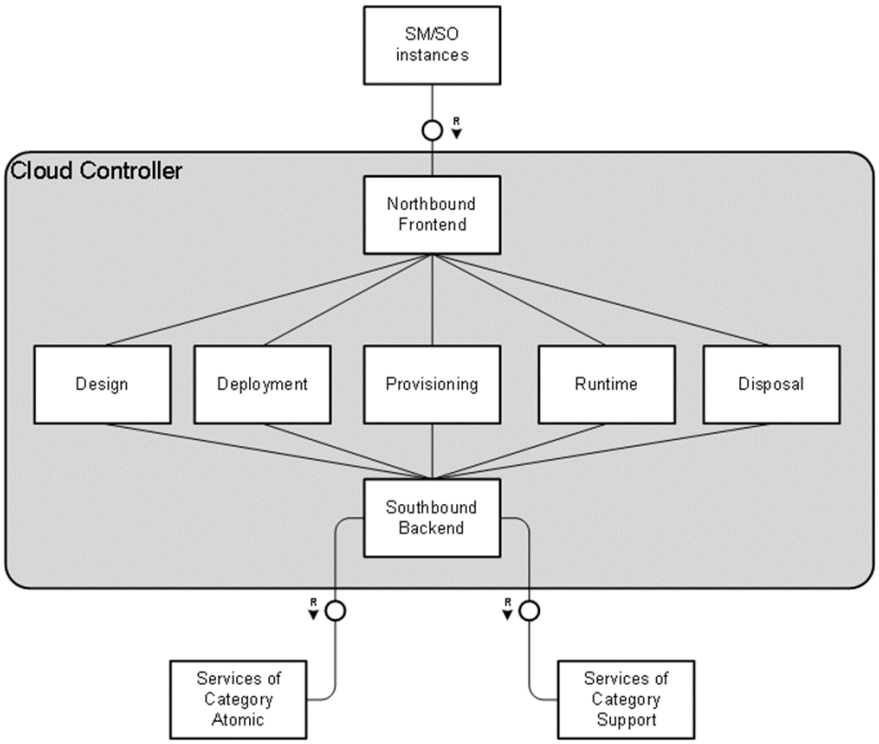


Figure 4 CloudController's conceptual overview

5.2.2 Sub-modules of the CloudController

Each of the following sections describes each of the modules of the overall CC logical architecture. Final technical details and complimentary design changes are reported upon. A specials focus is put on showing how concrete implementations detailed are abstracted. This will make sure that the general conceptual architecture holds when technical details changes and it will prevent the MCN services from being locked into a specific technology. This enables future reuse.

5.2.2.1 Northbound Frontend

As mentioned the main role of the Cloud Controlled is to make sure that SOs can manage their service instances. The entry-point into the systems is hence the northbound frontend of the CC. It enabled the deployment and control of the SO instances itself. Once a SO is instantiated it can make use of all the CC components to manage it's SICs.

The SO instances are hosted by the CC deployment module itself. For the current PoCs this part is realized by a PaaS solution called OpenShift. To make sure OpenShift can be replaced with other technologies, the specific interface had to be hidden. Hence the OCCI specification has been extended to support a PaaS model.

As reported in (D3.2 2014) the northbound API has been implemented as a OCCI compliant RESTful interface. This has deemed to be very powerful as all major control interfaces in the MCN project use the same language.

5.2.2.2 Design module

The design module is in charge of listing, registering and provide a query interface of all SM endpoints available. SM endpoints can pint to services of different categories such as MCN, Atomic, and Support. After some evaluation the OpenStack keystone project was chosen for this service.

To ensure abstraction, and prevent vendor lock-in, it has been abstracted using the Service Development Kit (SDK). Within the SDK there are methods to:

- Define and query SM endpoints on a high level, as well as
- A set of methods to directly look-up certain services. So instead of a lookup for a service of the type MaaS there has been a wrapper method which not only lookups the service, but also creates a new SI of the type MaaS.

The usage of Keystone has proven to be reliable and robust, especially in a federated environment. In that case all services to which the CC – and consequently the SO instances – want access to can be easily registered using the Regions. The abstraction through the SDK has deemed to be sufficient as it. In case Keystone would be replaces by another technology only another adapter needs to be implemented for the SDK. The code of the SO stays unchanged.

5.2.2.3 Deployment module

The deployment module of the CC is realized by two internal services:

- OpenShift for hosting SO instances, which is being abstracted using the Northbound frontend
- OpenStack Heat to manage IaaS based resources, which is abstracted using the de-factor standards AWS Cloudformation and HOT, which represent the ITG as information model, which is described in section 5.2.4.1.

The implementation of OpenStack Heat is abstracted using the Service Development Kit, while AWS CloudFormation and HOT based ITGs can be used in general. The SDK provides generalized adapters to interface with OpenStack heat. An adapter supporting AWS public cloud services could easily be implemented and while the ITGs and the SO implementation stay unchanged.

OpenStack Heat integrates very well with the basic atomic services of OpenStack, such as nova, neutron and cinder for compute, network and storage resources. However within the MCN project two major other stacks needed to be supported: CloudSigma's proprietary stack and Joyent's SDC.

5.2.2.3.1 CloudSigma

CloudSigma has built a proprietary stack from the ground up. While this does offer many performance benefits, it does present a challenge when it comes to facilitating compatibility with the ever-popular OpenStack cloud stack.

A key goal of the MCN project is to be able to have Telco services provisioned on demand, elastically and on geographically targeted cloud IaaS. Each end-to-end service is specified within a ITG, which defines how SICs are to be provisioned and interconnected.

The Orchestration engine for the ITG is OpenStack HEAT, this works well for OpenStack and to a great degree, Amazon clouds, but its compatibility is not ubiquitous with other cloud stacks, for example CloudStack, Eucalyptus or CloudSigma. To ensure compatibility with the CloudSigma stack,

CloudSigma has undertaken the development of a CloudSigma HEAT plugin, which enables HEAT to provision and decommission resources on the CloudSigma cloud as shown in Figure 5.

At this stage of the CloudSigma HEAT plugin development cycle, specific CloudSigma resource types have to be specified in order for HEAT to be able to provision them. It is not possible to directly map Amazon or OpenStack resource types to those of CloudSigma. This is mostly due to the high degree of VM instance configurability on the CloudSigma platform.

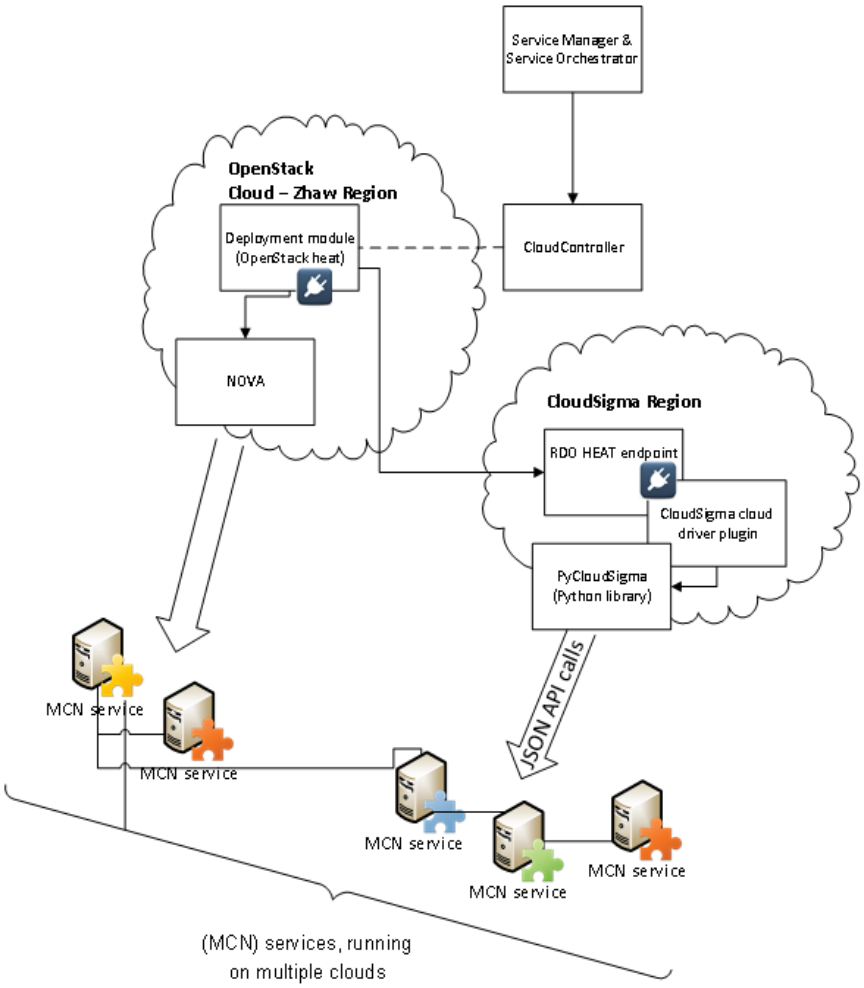


Figure 5 Resource placement of (MCN) services on multiple clouds

To meet the compute requirements of MCN, the most fundamental resource type is supported by CloudSigma - a compute instance. Considering the flexible and highly configurable nature of CloudSigma's VM instances, this involves a high level of parameterisation. Most of the parameters are optional and do supply default values if they are not specified within the HOT template.

To request a compute resource within the CloudSigma Cloud the ITG needs to define a resource of the following type:

```
"Type": "CloudSigma::Compute::Instance"
```

This demonstrates how easy it is to add ITGs to the SO bundle which offers support for CloudSigma's resources.

5.2.2.3.2 Joyent SDC

SmartDataCenter (SDC) is an alternative to OpenStack. It provides the same service, namely virtual machines, to customers. SDC is open-source and is mainly developed and used by Joyent, a public cloud provider. While OpenStack forces the user to evaluate which components and which plugins to use, SDC comes with a clear vision on how a cloud is built. It heavily focuses on reliability, resource fairness and guarantees. Thanks to the underlying hypervisor SmartOS, an OS that originated from the trusted enterprise OS Solaris, SLAs can be enforced on CPU, memory and even storage resources.

To ensure the deployment module of the CC can make use of SDC, a specific plugin is written. Hence the actual integration of SDC with OpenStack happens in a custom OpenStack Heat plugin. The0Hheat plugin "sdc_plugin.py" implements four resources:

- *SDC::Compute::KVM* - This resource represents a generic virtual machine, powered by KVM and additionally isolated with a SmartOS zone..
- *SDC::Compute::SmartMachine* - Almost identical to SDC::Compute::KVM, but this resource implements virtual machines as so called SmartMachines or Solaris Zones.
- *SDC::Network::Network* - This resource represents a network configured in SDC. The user has similar possibilities to OpenStack Neutron.
- *SDC::Network::SmartNetwork* - To enable the on-demand provisioning of networks the SmartNetwork resource was created.

Instead of talking to the admin APIs directly from the plugin, a library was created to handle all communication in a uniform matter. The python library "sdcadmin" abstracts the APIs into easy to manipulate objects. The flow of the resource requests is visualised in Figure 6.

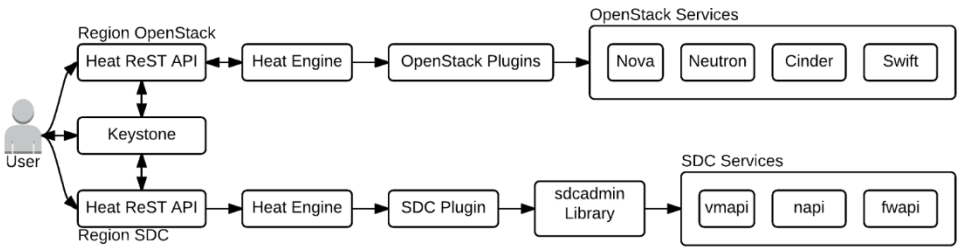


Figure 6 Integration of SDC into OpenStack heat

5.2.2.4 Provisioning module

In terms of functionality and requirements the Provisioning Module has been described in past deliverables. For example in deliverable, (D3.3 2014) we already show that Foreman was chosen to realize the Provisioning Module by providing some extra-functionality on top of Puppet. Also, within the same deliverable we provide a high-level description on how Puppet can be used for automating the configuration on external nodes.

When an external node with Puppet agent boots, the agent tries to reach the Puppet Master using the configured hostname, which must be statically set in the configuration files. After the Puppet Master has identified the node it will deliver to the agent the configuration template, which the agent will use

to configure the node. After the initial setup, the agent will periodically fetch the manifests (configuration templates) from the Puppet Master and will check if the node is in the desired state.

The Puppet Master has a configuration file (named 'site.pp') which associates manifests with hostnames. This configuration file has to be statically defined before starting the Puppet Master, which constitutes a drawback when using Puppet in a dynamic environment such as MCN. To circumvent this issue, Puppet uses a component named External Node Classifier (ENC) which is a mechanism to implement a dynamic node identification. ENC enables an interface in Puppet that can be used to ask an external element, in this case Foreman, to provide the manifest for a given node, see Figure 7. Besides the latter functionality, Foreman also provides others such as:

- Hostgroups – Foreman provides the ability to aggregate nodes in groups when sharing the same configuration. When necessary, a single update to the HostGroup definition will translate in the automatic re-configuration of all the nodes belonging to that group.
- Foreman Northbound Interface – Foreman provides a RESTful API exposing the necessary functionalities. This API greatly facilitates the job of integrating the Provisioning module in the SDK.
- Multiple functionalities for Cloud environments (e.g domains definition, audit tools …) that can be later used to extend the Provisioning module functionality.

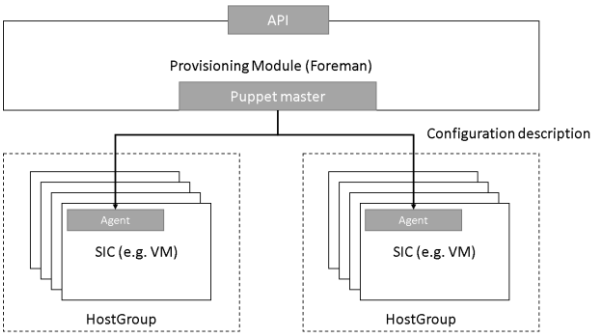


Figure 7 Provisioning Module deployment

By using this described approach the Provisioning module offers a rich set of features which support the SO in provisioning the SICs. Again to abstract the concrete implementation used here – Foreman and Puppet – the provisioning module has been abstracted using the SDK. Methods exist which allow the SO to provisioning SICs, without directly interfacing with Foreman.

5.2.2.5 Operation and Runtime Module

One of the goals of the SDK is to provide supporting tools for a service to easily manage itself during the runtime phase of the MCN lifecycle. In the architecture described in D2.5 this is represented by the runtime module. The most basic usage of the runtime module is for scaling of the service, based on current load or other constraints specific to the service. Runtime management also includes the health management of the service, such as deciding which actions are taken when one of the virtual instances of the service crashes.

To achieve our goal, we integrated Monasca, a well-known services for OpenStack, within the MCN SDK to gather proper metrics from running services and execute callback actions on a service orchestrator when service specific conditions are reached. This allows a service orchestrator to setup custom thresholds for the service instance it manages and receive alerts when they are reached.

When creating a service, the service developer can import the runtime module of the SDK, and exploit it first during the provisioning phase of the service to set service-specific alarms.

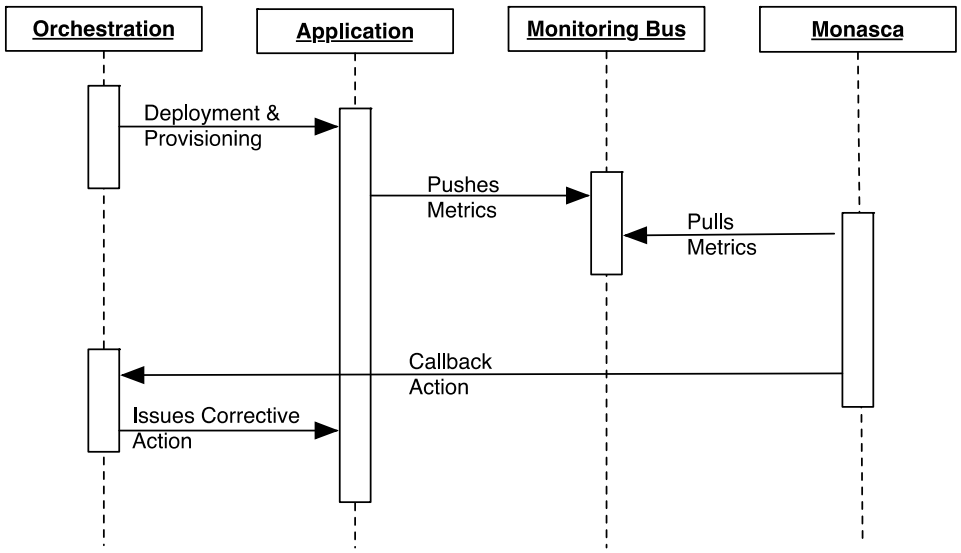


Figure 8 Monasca Integration with the MCN Orchestration

At the same time, it is necessary to configure the virtual machines comprising a service to send metrics to Monasca, this is done through a simple Monasca agent which task is to read service metrics and push them to a common Monasca-watched monitoring bus. This can be done using Heat with *SoftwareConfig* and *SoftwareDeployment* resources tasked to execute a simple configuration script to prepare the agent and configure its name, which is typically the ID of the current service instance.

In short, as shown in the sequence diagram in Figure 8, virtual machines from a service instance push metrics to a monitoring bus. Monasca, creates an alarm if the preconfigured conditions are reached. It then issues a callback action on the orchestrator with the name of the alarm as an OCCI attribute, so the orchestration knows which action to take.

The last step is for the service orchestrator to execute an action, which can for instance be an update action on the Heat template of the service, adding or removing virtual resources. Again the abstraction through the SDK and the usage of the OCCI interface to support a callback to the SO instance has proven to be very valuable and easy to use.

5.2.2.6 Disposal module

The disposal module is merely a logical module. The disposal of service and resource instance is realized through the deployment module. Once resources or services can be deleted the request for disposal is forward to those modules as they track the instances they manage.

5.2.2.7 Southbound module

The southbound module is in charge for communication with service instance of category support and atomics. To do so, the following principles are applied:

- To instantiate (virtual) resources (compute, network and storage entities) OpenStack Heat is used to talk to the appropriate services: OpenStack nova, neutron, and cinder as well as CloudSigma and Joyent's SDC plugins. The plugins hence realize the southbound API calls.

Once resource instances are available they can be directly or indirectly be interfaced by using either:

- their endpoints (IP address) for directly interfacing,
- or by using the SDK (deployment, provisioning, and runtime module) interfaces,
- or by standard control layers such as OCCI to e.g. manipulate network resources as described in section 2.4.1.

- To instantiate and manage SICs of the category support the design module and the SDK can be used. This gives enough abstraction to prevent a lock-in.

5.2.3 Service Dependencies

As reported upon in (D3.1 2013) the services the CC requires itself can be categorized in three types: support, atomic, and internal services. The list provided within (D3.1 2013), section 5.2.3 is still valid and holds and demonstrated in the various PoCs.

5.2.4 Interaction with Service Orchestrators and Service Managers

As the CC is a very central component to the MCN project the interactions need to be clearly defined. The following sections detail these aspects. In general the following flow can be used when developing a service for MCN:

- Based on the business needs define an information model which defines which other services are required by your services. Represent this information model in the STG.
- Customize the Service Manager (SM) which has been implemented by task 3.4. The STG is expressed using the interface of the SM.
- Define at least one ITG using the ITG editor described in section 5.2.4.3. Multiple ITGs can be defined for different scenarios (SLAs levels, multiple regions, etc.)
- Implement a specific SO which is able to control the life-cycle of the service. Embed the algorithms for orchestration and the ITGs from the previous step into the SO bundle. The algorithms should make use of the SDK – a sample SO has been implemented by task 3.4 for reference.

Once done the principals as defined in deliverable (D2.2 2013) apply and can be followed. The following sections give further details on the information model (STG and ITG) and the ITG editor.

5.2.4.1 CloudController information model

In MCN, service dependencies before service instantiation are described by the service template graph. Once the service is instantiated the STG is then realised by the service instance graph (SIG). It is the Service Orchestrator (SO) that is responsible for creating the SIG based on the STG. The conceptual aspects of the STG are detailed in (D2.5 2015), however here we detail how the STG is technically realised. The key contents of the STG, technically realised as a JSON file known as the service manifest, are as follows:

- service information: this information is at the root of the service manifest document. In this the following attributes are defined:

- **service_type**: this is the full OCCI service type identifier (known as the Category in the OCCI core model)
- **service_description**: is a simple textual description of the service
- **service_attributes**: is a hashmap of parameters that the service type advertises and whether they can be modified (mutable) or cannot be modified (immutable).
- **service_endpoint**: is the full URI to the HTTP endpoint where a service can be created, updated, deleted and retrieved.

- service dependencies: this is a list of all services that are required for operating and delivering the service offered to an EEU.
  - **depends_on**: Each element of the list is a hashmap that contains. The root of this hashmap is the service type identifier of the service dependency. This is the OCCI type (Category) identifier that is unique. An example of such a service type identifier is ```http://schemas.mobile-cloud-networking.eu/occi/sm#demo1```. Within the hashmap the expected inputs (a hashmap element named ```inputs```) to the service (for purposes of service creation) are present. These are typically formulated in the following fashion:

    <SERVICE_TYPE_IDENTIFIER>#<PARAMETER_NAME>

    PARAMETER_NAME can also be remapped to a different name and also an explicitly set static value can be set against it.

For example, if the service offered to the EEU is DSS and DSSaaS requires (as dependencies) MaaS and DNSaaS, where DNSaaS also requires MaaS then the complete definition required in the service manifest is:

```
   "depends_on": [
     { "http://schemas.mobile-cloud-networking.eu/occi/sm#maas": { "inputs": [] }
},
     { "http://schemas.mobile-cloud-networking.eu/occi/sm#rcb": { "inputs": [] } },
     { "http://schemas.mobile-cloud-networking.eu/occi/sm#dnsaas": {
         "inputs": [
           "http://schemas.mobile-cloud-
networking.eu/occi/sm#maas#mcn.endpoint.maas"
         ] }
     }
   ]
```

The ITG is not part of the STG, however it is technically related as both the STG realised as the service manifest and the ITG, currently realised as an OpenStack Heat template, are both packaged as part of the Service Bundle.

With both STG and ITG (service manifest and heat template(s)) available to the SO, it can when requested by the SM create a complete service instance, including all service dependencies. How the SO creates its needs based on the ITG has already been described in previous deliverables. What is additionally described here is how the external service dependencies, as specified by the service manifest, are orchestrated through composition.

The dependencies specified in the service manifest are executed upon by the SO's "Resolver" component. The Resolver component is something that every SO implementation has access to. It has

the same interface as service orchestrators have, adheres to the MCN lifecycle and so its operation is already familiar to those that are responsible for implementing and delivering services.

In version 0.3 we explicitly define the basic SO interface. SO implementations (e.g. those for RAN, EPC etc.) inherit from this interface, specified as a python class with no implementation (python does not support pure interfaces like Java), other than some basic initialisation in the constructor of the class. As part of this initialisation, the Resolver is created and its capabilities are available to subclasses (SO implementations).

In the context of a SO creating a service the following order of resource and service instance dependencies creation is followed:

1. Dependencies defined in both the ITG and the STG are deployed in parallel. This is possible as at this stage only the individual dependencies have to be created. This separation is given by the MCN lifecycle and proves to be very useful by maximising parallelisation of the deployment phase.
2. Once all individual dependencies of the ITG and STG have been deployed, the STG dependencies are provisioned. In this case, the required parameters of the respective service dependency are supplied by the resolver component. The result of this is a graph of STG dependencies that are fully configured and readied for use by the ITG dependencies. It is the separation of deployment and provisioning phases that is a large advantage in also allowing for simple and effective configuration (provisioning phase).
3. Now with all STG dependencies deployed and provisioned the ITG dependencies of the service instance can now be configured with the endpoints of the external STG dependencies. This information is supplied by the Resolver instance of the particular SO implementation. This information again is a hashmap of parameters required by the ITG defined instances.

The sequence diagram of this process was presented in (D2.5 2015).

5.2.4.2 Service Development Kit

As shown in section 5.2.2 the SDK is one of the key components of the CC which enable abstraction from underlying technology. This is needed to keep the SO implementation lightweight and prevent a lock-in to a certain technology. Future enhancements of the SOs and the SICs it manages are so easier to realize.

The SDK as implemented – and throughout tested & documented – within the project has provided the project with the following enhancements:

- **Abstraction** of the CCs internal services for the deployment, provisioning, and runtime module.
- **Integration** of support services such as CDN, RCB, Monitoring and DNS.
- **Ease-of-development** for developing SM and SOs by providing common functionalities. This includes the implementation of standardized interface of the SM and SO which are again based on the OCCI standard.

5.2.4.3 ITG editor

Since the early beginning of the MCN project the need for two different level of formalism to describe entities in for example the NFV context has been clearly recognized. On the one side there are the computing, storage and networking resources that need to be instantiated in the cloud infrastructure and properly configured in order to be capable to interwork one with the other. On the other side there are the services that, though deployed on an infrastructure, though virtualized, need to be fully abstract from the actual implementation in order to fully benefit of the MCN features. This distinction, that has become one of the distinctive points in the MCN architecture, has been maturing together with the deepening of the comprehension of how NFV requirements and objectives should be coped with.

The two formalisms adopted in the MCN project to cope with the two above described aspects are both graphs, the STG and ITG. Both of them map "nodes", representing objects with different meaning in the different contexts, with "edges" representing relationships among the objects. The power of the graph formalism can effectively describe those complex structures that are quite common in the definition of NFV realms.

As the graph concept is naturally and immediately associated with its graphic representation, the development of a graph editor with the aim of pushing the understanding of "plug 'n play" benefits of the NFV was a straightforward step. To this purpose a first abstract representation of services made of Service Components (nodes) interconnected through Interfaces (edges) was chosen and the (Service Template Graph Editor) StgEditor was early devised to make visible such approach.

While the development of the StgEditor led to a graphical tool capable of dynamically compose such graphs by dragging icons onto a canvas and by linking them with typed connections, and eventually generate Heat Orchestration Template (HOT) code, the ITG and STG concepts have become more precise. As the HOT formalism is used by Service Orchestrators to instruct the CC about the infrastructure resources needed by their service instances, it was decided that the formerly defined StgEditor should be rather renamed to ItgEditor to better position its role in the MCN overall picture.

5.2.4.3.1 ItgEditor multi-site features

The ItgEditor allows for a graphical depiction of a network of components (nodes) interconnected through interfaces (edges). When the graph is completed and all the components and interfaces have been properly customized, a command generates a HOT code template that can be deployed onto the cloud infrastructure by means of the Cloud Controller services. As there might likely be several geographically distributed cloud infrastructures interconnected to support the MCN services, the need to cope with multiple cloud platforms (VIM) and, prospectively, with multiple networking platforms (Network Service Manager – NSM), has arisen as a requirement for the ItgEditor.

To this purpose new concepts have been introduced in order to model the extended scenario. The first newly introduced concept is the Infrastructure. The infrastructure is needed to encompass all the available "platforms" where each platform provides either VIM or NSM.

Each platform is described by a number of common parameters including a name and a platform class. The platform class is needed in order to allow for different behaviors by the ItgEditor while generating the output code. In order to allow the editor operation of "dragging" a node onto a VIM, the latter is geometrically mapped onto a specific area on the background image displayed in the editor canvas. Other parameters are included and are specific for each platform e.g. for authentication purposes. Other parameters are eventually included in the definition to hold values that are "global" to the platform.

Figure 9 depicts the above described concepts: the red ovals, actually not displayed on the ItgEditor screen, ideally represent VIMs span over the geographical territory. The blue ovals, that in the figure represent the NSMs that, though actually not mapped on any geographical area, they are nevertheless mapped onto VIM couples.

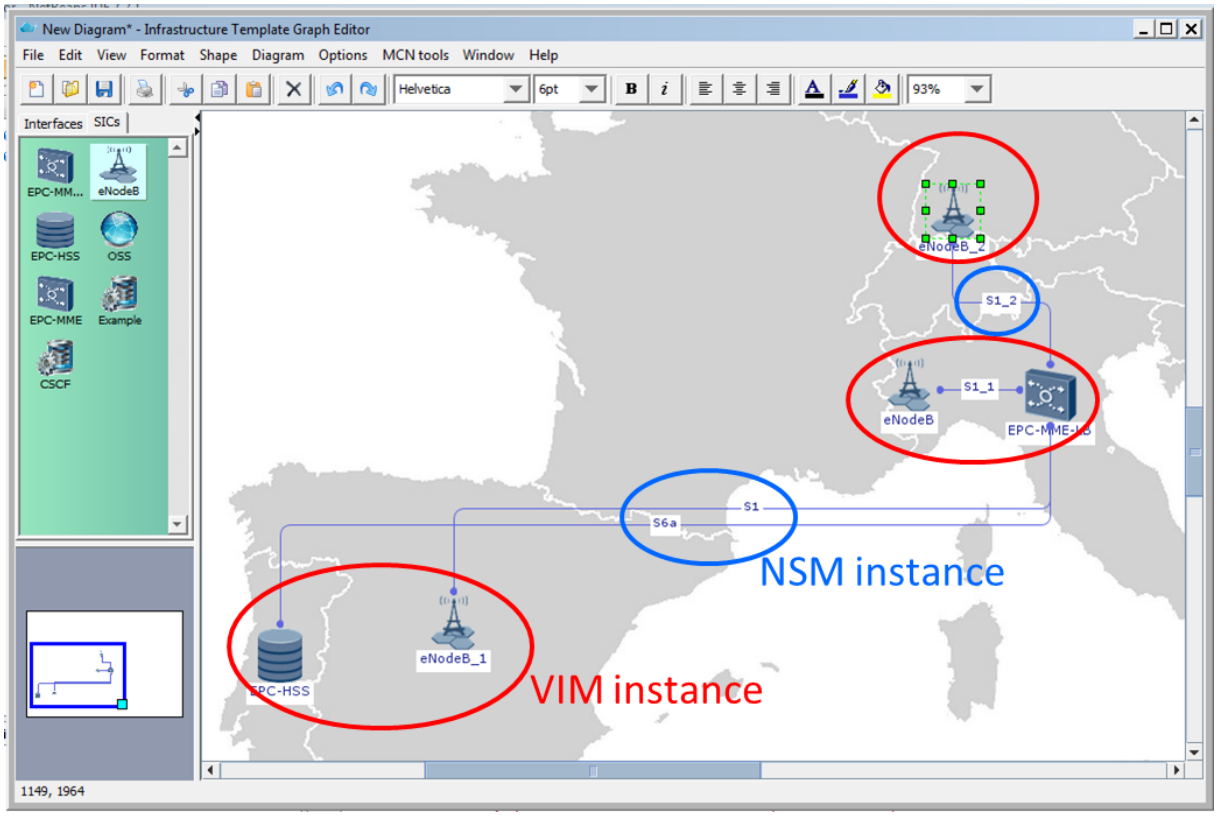


Figure 9 Federated infrastructure concept in the ITG editor

During the drawing process there might be the case where two VIMs overlap on the same geographical area. In this case it is possible to edit the node and choose on a drop down menu which is the VIM to be mapped. The same applies for edges: wherever more NSMs can be used to connect resources distributed over two VIMs, the edge can be edited and the chosen serving NSM selected in the drop-down menu.

5.2.4.3.2 The "build" process

Once the ITG has been drawn on the canvas, a command in the MCN tools menu instructs the ItgEditor to generate the code for the templates.

The build process starts analyzing all the nodes and edges in the graph. Each node is mapped to a sub-graph instance specific to a VIM instance. Each edge is mapped to a VIM instance if both nodes at its endpoints are mapped to the same VIM instance. If an edge connects two nodes mapped to two different VIM instances, it is mapped to the serving NSM. At the end of these distribution phase, all the edges mapped to some NSM are processed. The result of the build operation run on each NSM might generate both an output file containing metadata to be used in commending the NSM and some output information that is fed to the adjacent VIMs.

Then each sub-graph is processed and an output template file is generated per each of them. At the end of the process a manifest file listing all platforms, templates and configuration files is which are then embedded in the SO bundle.

5.3 Integration with other tasks

As the CC is a key component almost all tasks within the project interacted with task 3.4. Within the work-package task 3.1 has provided integrated the DNSaaS, provided OCCI interface for SDN and extended methods to manage QoS and traffic steering which can be used through the CC. Task 3.2 has done performance analysis and by integration of different cloud stacks such as OpenStack, Joyent SDC, and CloudSigma's cloud different performance levels can be achieved. Task 3.3 integrated the MaaS. Task 3.5 makes use of the CC functionalities.

Across work-packages task 3.4 has heavily influenced the architecture based on the requirement defined within WP2. EPCaaS of WP4 leverages the features of the CC and mainly the SDK. For WP5 the integration of CDN and RCB services through the SDK could be achieved. Also for WP5 the SLA service and the integration with the CC will be demonstrated through the integration work of WP6. Documentations and furthermore throughout tested software have been delivered to the other WPs.

5.4 Conclusions

Many of the basic concepts and initial architectures have been described in (D3.2 2014). Luckily many of those definitions only needed minor refinements over the lifetime of the projects. The implementation and technical details of the CC have been shown in the prototype deliverables (D3.2 2014) and (D3.3 2014). Hence this final report again gives a complete overview of the CC while leveraging from the refinements and technical details as presented in those two deliverables.

All components of the CC have been reported upon and the major lessons learned shown within the previous sections. The focus was to provide a robust framework which can be used within the project and for years to come. To enable this, the key focus has been on abstraction away concrete technologies and by providing standardized interface which are easy to reuse. The usage of OCCI for that "language" of control has proven to be very valuable, and together with the development of the SDK provided two major outcomes of the MCN project as a whole.

6 Radio Access Network-as-a-Service

The following sections describe the contributions from task 3.5, entitled "Wireless Cloud".

6.1 Introduction

The following sections describe the work conducted by Task 3.5 on the RAN as a Service (RANaaS). This task focuses on the design, implementation, and test of a set of elements for a system that can be used to manage Radio Access Network (RAN) for an organization that specializes in providing on demand RAN to customers. The work presented here describes all final architectural changes, evaluations, and developments from M12 to M30, with respect to those already detailed in (D3.2 2014) and (D3.3 2014).

6.2 RAN-as-a-Service Reference Architecture Model, Concepts and Information Flows

The following sections give more details on the architecture and implementation of the RANaaS.

6.2.1 Architecture Reference Model

While the concept of C-RAN has been clearly defined, more research is needed to find an optimal architecture that maximizes the benefits behind C-RAN, and based on which a true proof-of-concept could be built. From the perspective of the operator such an architecture has to meet the scalability, reliability/resiliency, cost-effective requirements. However, from the perspective of the software radio application, two main requirements have to be met: (1) strict hard deadline to maintain the frame and subframe timing, and (2) efficient/elastic computational resources (e.g. CPU, memory) to perform intensive digital signal processing for different transmission modes (beamforming, CoMP, etc).

Broadly, three main choices are possible to design a C-RAN, each of which provide a different cost-power-performance-flexibility trade-offs.

- **Full GPP:** where all the processing (L1/L2/L3) is performed on the host/guest systems. According to China Mobile, the power consumption of the OpenAirInterface full GPP LTE softmodem is around 70w per carrier.
- **Accelerated:** where only certain functions, such as FFT/IFFT, are offloaded to a dedicated hardware (e.g. FPGA, DSP), and the remaining functions operate on the host/guest. The power consumption is reduced to around 13~18w per carrier.
- **System-on-Chip:** where the entire L1 is performed on a SoC and the reminder of the protocol stack runs on the host/guest. The power consumption is reduced to around ~8w per carrier.

As shown in Figure 10, the hardware platform can either be full GPP or a hybrid. In the later case, all or part of the L1 functions might be offloaded to dedicated accelerators, which can be placed locally at the cloud infrastructure to meet the real-time deadline and provide a better power-performance trade-off or remotely at RRH to reduce the data rate of fronthaul. Different service compositions can be considered, ranging from all-in-one software radio application virtualization to per carrier, per layer or per function virtualization as mentioned earlier. The virtualization is performed either by a container engine or a hypervisor, under the control of a cloud OS, which is in charge of life-cycle management of a composite software radio application (orchestrator) and dynamic resource provisioning.

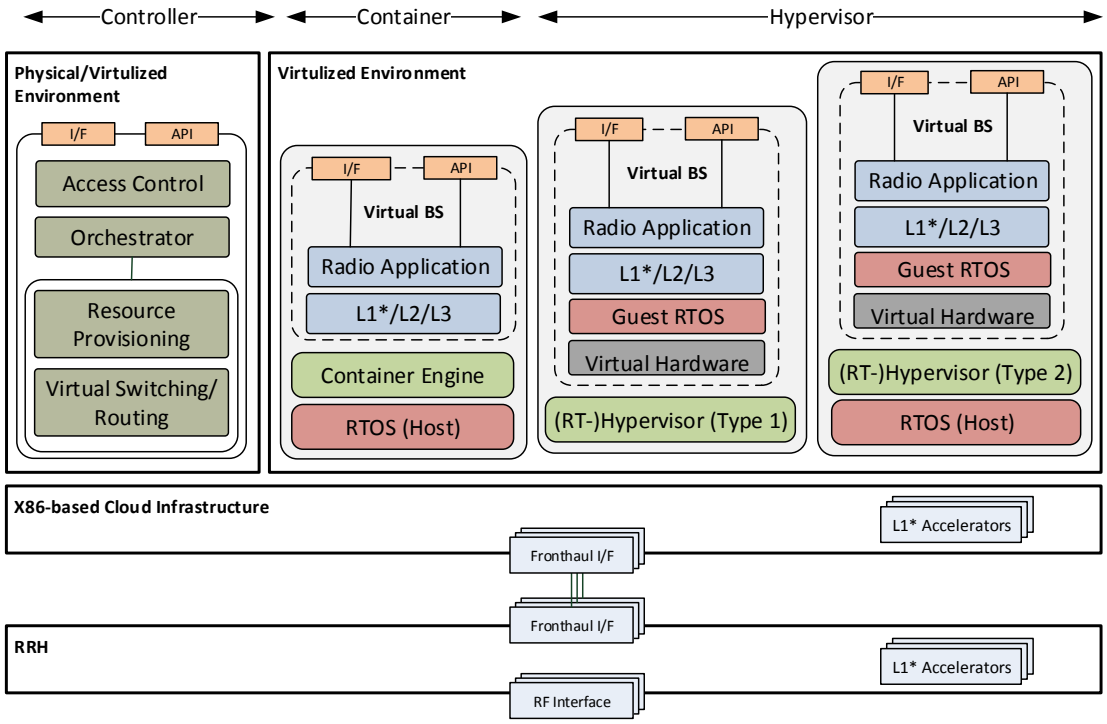


Figure 10 Candidate C-RAN architectures

6.2.2 RANaaS Lifecycle

RANaaS describes the service lifecycle of an on-demand, elastics, and pay as you go 3GPP RAN on the top of cloud infrastructure. Thus, lifecycle management is a key for successful adoption and deployment of C-RAN and related services (e.g. MVNO as a Service). It is a process of network design, deployment, resource provisioning, operation and runtime management, and disposal.

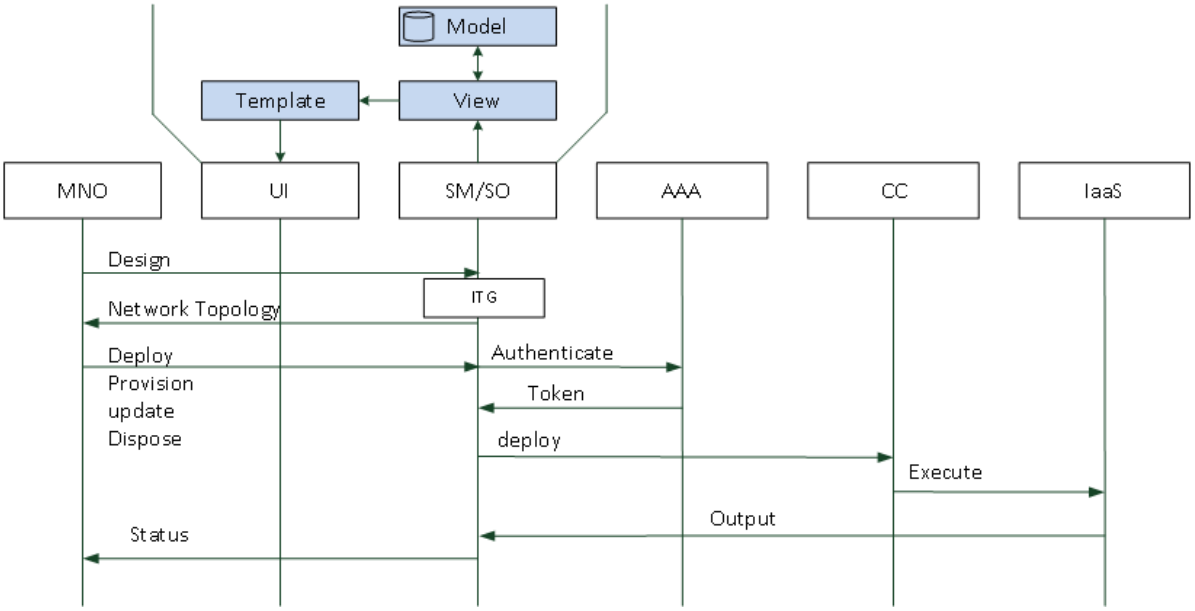


Figure 11 RANaaS lifecycle

6.3 RANaaS performance analysis

6.3.1 BBU Processing

Figure 12 illustrates the main RAN functions in both TX and RX spanning all the layers, which has to be evaluated to characterize the BBU processing time and assess the feasibility of a full GPP RAN. Since the main processing bottleneck resides in the physical layer, the scope of the analysis in this chapter is limited to the BBU functions. From the figure, it can be observed that the overall processing is the sum of cell- and user-specific processing. The former only depends on the channel bandwidth and thus imposes a constant base processing load on the system, whereas the latter depends on the MCS and resource blocks allocated to users as well as SNR and channel conditions. The figure also shows the interfaces where the functional split could happen to offload the processing either to an accelerator or to a RRH.

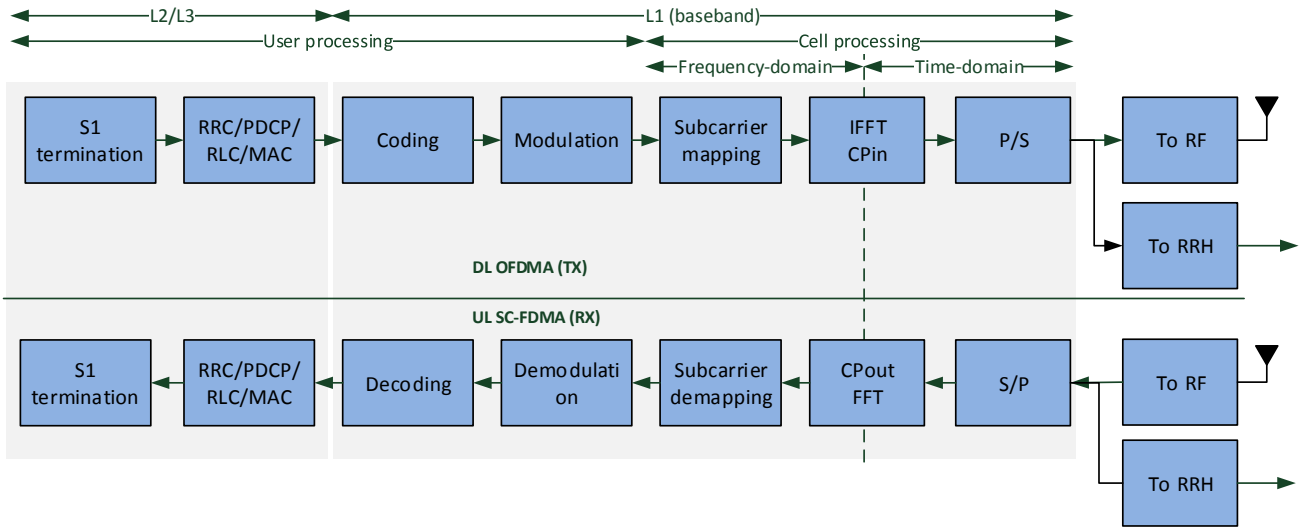


Figure 12 Functional Block diagram of LTE eNB for DL and UL

To meet the timing and protocol requirements, the BBU processing must finish before the deadlines. One of the most critical processing that requires deadline is imposed by the Hybrid Automatic Repeat Request protocol (HARQ) in that every received MAC PDU has to be acknowledged (ACK'ed) or non-acknowledged (NACK'ed) back to the transmitter within the deadline. In FDD LTE, the HARQ Round Trip Time (RTT) is 8 ms. Each MAC PDU sent at subframe N is acquired in subframe N+1, and must be processed in both RX and TX chains before subframe N+3 allowing ACK/NACK to be transmitted in subframe N+4. On the receiver side, the transmitted ACK or NACK will be acquired in subframe N+5, and must be processed before subframe N+7, allowing the transmitter to retransmit or clear the MAC PDU sent in subframe N. Figure 13 and Figure 14 show an example of timing deadlines required to process each subframe in downlink and uplink respectively.

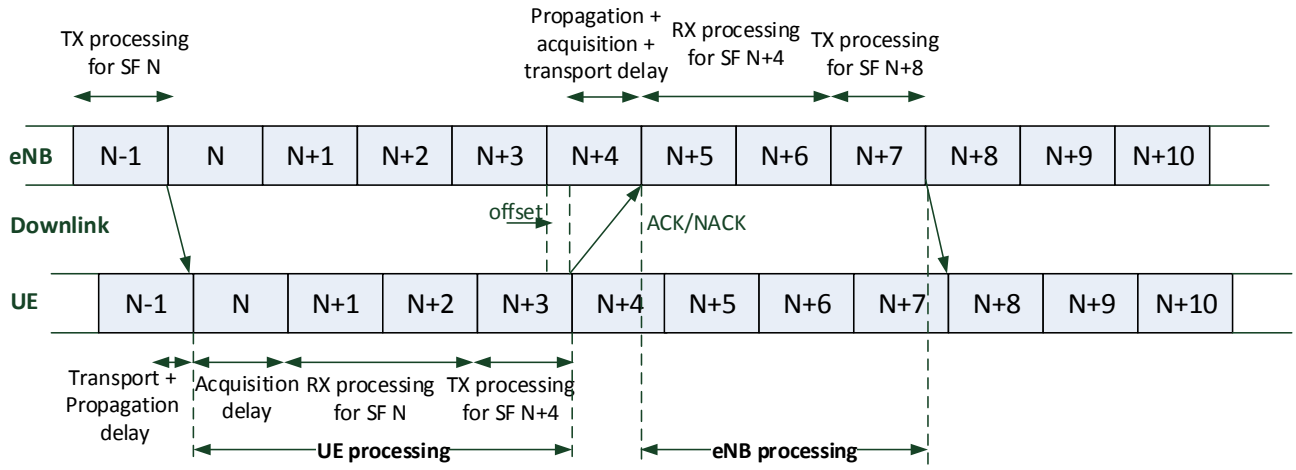


Figure 13 FDD LTE DL HARQ timing

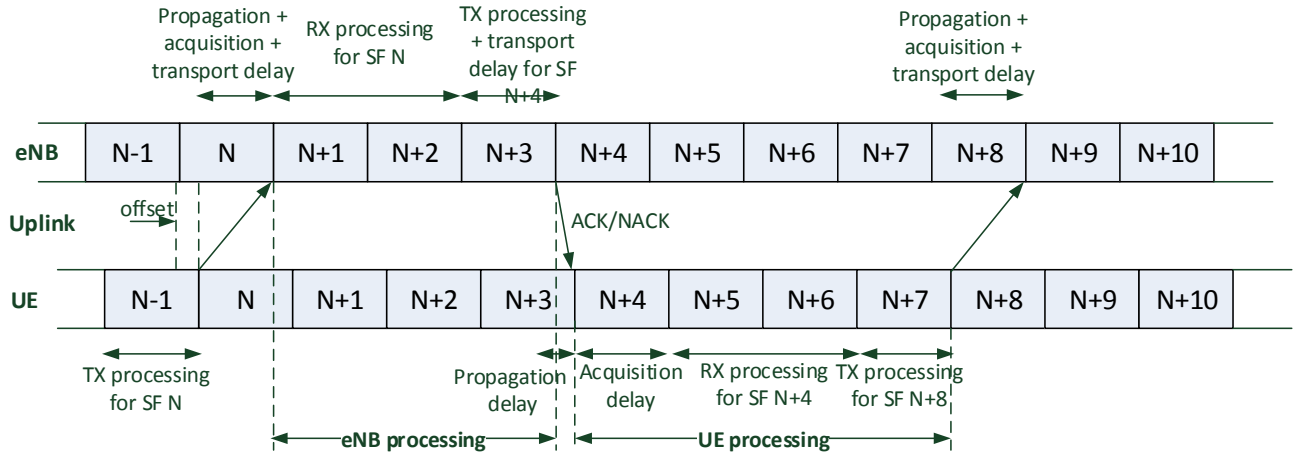


Figure 14 FDD LTE UL HARQ timing

It can be observed that the total processing time is 3ms, out of which 2ms is available for RX processing and 1ms for TX. Thus the available processing time for an eNB to perform the reception and transmission is upper-bounded as follows:

$$\boldsymbol{TRx + TTx \leq \frac{THARQ}{2} - (TPropagation + TAcquisition + TTransport + Toffset)}$$

where THARQ = 8, TPropagation + TAcquisition + TTransport $\leq$ 1ms, and Toffset = 0 in DL.

Depending on the implementation, the maximum tolerated transport latency depends on the eNodeB processing time and HARQ period. As mentioned earlier, NGMN adopted a 250 us for the maximum one-way fronthaul transport latency. Hence, the length of a BBU-RRH link is limited to around 15 km to avoid too high round-trip-delays (given that the speed of light in fiber is approximately 200 m/us). At maximum distance of 15 km, the remaining overall processing time will be between 2.3--2.6 ms.

6.3.2 Evaluation Setup

Four set of different experiments are performed. The first experiment analyses the impact of different x86 CPU architecture on BBU processing time, namely Intel Xeon E5-2690 v2 3Ghz (same architecture as IvyBridge), Intel SandyBridge i7-3930K at 3.20Ghz, and Intel Haswell i7-4770 3.40GHz. The second experiment shows how the BBU processing time scale with the CPU frequency.

The third experiment benchmarks the BBU processing time in different virtualization environments including LXC, Docker, and KVM against a physical machine (GPP). The last experiment measures the I/O performance of virtual Ethernet interface through the guest-to-host round-trip time (RTT).

All the experiments are performed using the OpenAirInterface DLSCH and ULSCH simulators designed to perform all the baseband functionalities of an eNB for downlink and uplink as in a real system. All the machines (hosts or guests) operate on Ubuntu 14.04 with the low latency (LL) Linux kernel version 3.17, x86-64 architecture and GCC 4.7.3. To have a fair comparison, only one core is used across all the experiments with the CPU frequency scaling deactivated except for the second experiment.

The benchmarking results are obtained as a function of allocated physical resource blocks (PRBs), modulation and coding scheme (MCS), and the minimum SNR for the allocated MCS for 75\% reliability across 4 rounds of HARQ. Note that the processing time of the turbo decoder depends on the number of iterations, which is channel-dependant. The choice of minimum SNR for a MCS represents the realistic behaviour, and may increase number of turbo iterations. Additionally, the experiments are performed using a single user with no mobility, 8-bit log-likelihood ratios turbo decoder, SISO mode with AWGN channel, and a full buffer traffic ranging from 0.6Mbps for MCS 0 to 64Mbps for MCS 28 in both directions. Note that if multiple users are scheduled within the same subframe in downlink or uplink, the total processing depends on the allocated PRB and MCS, which is lower than a single user case with all PRBs and highest MCS. Thus, the single user case represents the worst case scenario.

```
start = rdtsc();
bbu_function();
stop = rdtsc();
processing_time = (stop - start)/cpu_freq;
```

The processing time of each signal processing module is calculated using timestamps at the beginning and at the end of each BBU function. OAI uses the rdtsc instruction implemented on all x86 and x64 processors to get a very precise timestamps, which counts the number of CPU clocks since reset. Therefore the processing_time is proportional to the value returned by the following pseudo-code:

To allow a rigorous analysis, total and per function BBU processing time are measured as shown in Table 7. For statistical analysis, a large number of processing_time samples (10000) are collected for each BBU function to calculate the average, median, first quantile, third quantile, minimum and maximum processing time for all the subframes in uplink and downlink.

Table 7 OAI BBU processing time decomposition in Downlink and Uplink

RX Function	timing(us)	TX Function	timing(us)
OFDM Demodulation	109.695927	OFDM modulation	108.308182
ULSCH Demodulation	198.603526	DLSCH modulation	176.487999
ULSCH Decoding	624.602407	DLSCH scrambling	123.744984
⌊ *interleaving*	12.677955	DLSCH encoding	323.395231
⌊ *demultiplexing*	117.322641	⌊ trubo encoder	102.768645
⌊ *rate matching*	15.734278	⌊ rate matching	86.454730
⌊ *turbo decoder*	66.508104	⌊ interleaving	86.857803
⌊ *init*	11.947918		
⌊ *alpha*	3.305507		
⌊ *beta*	3.377222		
⌊ *gamma*	1.018105		
⌊ *ext*	2.479716		
⌊ *intl*	5.441128		
Total RX	931	Total TX	730

6.3.2.1 CPU Architecture Analysis

Figure 15 depicts the BBU processing budget in both directions for the considered Intel x86 CPU architecture. It can be observed that the processing load increases with the increase of PRB and MCS for all CPU architectures, and that it is mainly dominated by the uplink. Furthermore, the ratio and variation of downlink processing load to that of uplink also increases with the increase of PRB and MCS. Higher performance (lower processing time) is achieved by the Haswell architecture followed by SandyBridge and Xeon. This is primarily due to the respective clock frequency, but also due to a better vector processing and faster single threaded performance of Haswell architecture. For the Haswell architecture, the performance can be further increased by approximately a factor of two if AVX2 (256-bit SIMD compared to 128-bit SIMD) instructions are used to optimize the turbo decoding and FFT processing.

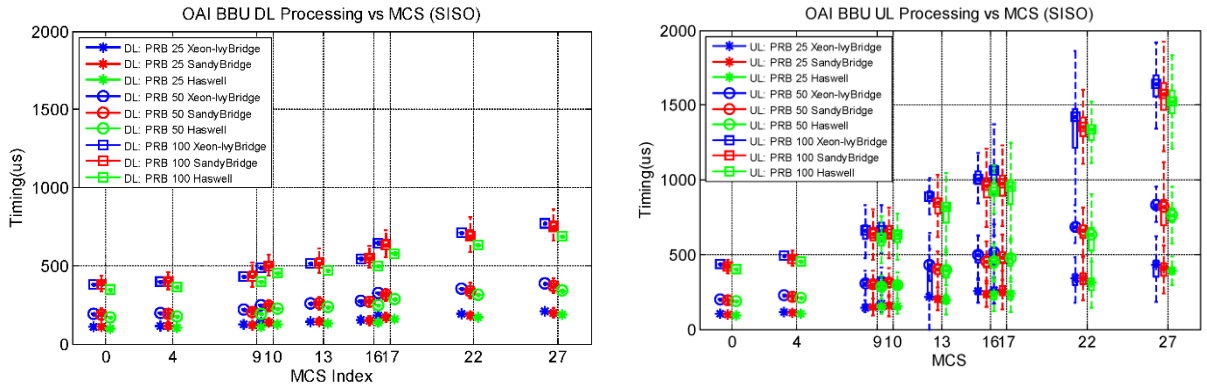


Figure 15 BBU processing budget in downlink (left) and uplink(right) for different CPU architecture

6.3.2.2 CPU Frequency Analysis

Figure 16 illustrates the total BBU processing time as a function of different CPU frequencies (1.5, 1.9,2.3,2.7,3.0, and 3.4 GHz) on the Haswell architecture. The most time consuming scenario is considered with 100 PRBs and downlink and uplink MCS of 27. In order to perform experiments with different CPU frequencies, Linux ACPI interface and cpufreq tool are used to limit the CPU clock. It can be observed that the BBU processing time scales down with the increasing CPU frequency. The figure also reflects that the minimum required frequency for 1 CPU core to meet the HARQ deadline is 2.7GHz.

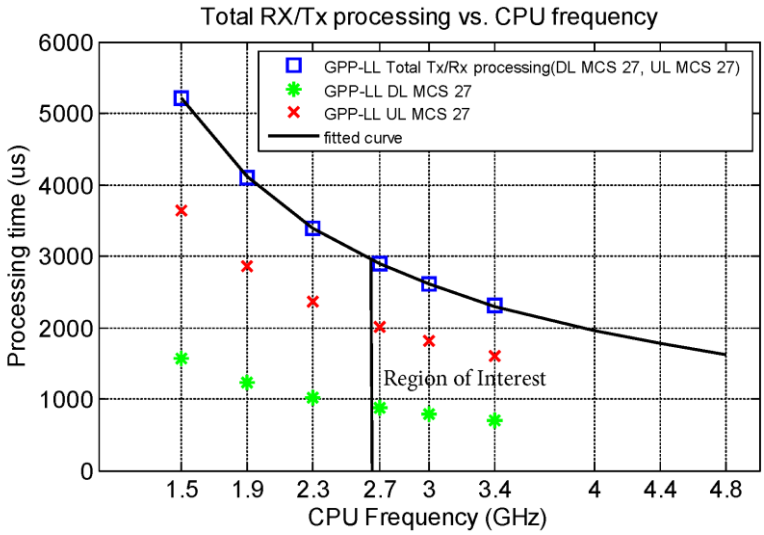


Figure 16 Total processing time as a function of CPU frequency

Based on the above figure, the total processing time per subframe, Tsubframe, can be modelled as a function of CPU frequency:

$$Tsubframe(x)\ [us]\ = \alpha / x$$

, where α = 7810 $\pm$ 15 for the MCS of 27 in both directions, and x is CPU frequency measured in GHz.

6.3.2.3 Virtualization Technique Analysis

Figure 17 compares the BBU processing budget of a GPP platform with different virtualized environments, namely Linux Containers (LXC), Docker, and KVM, on the SandyBridge architecture (3.2GHz). While on average the processing time is very close for all the considered virtualization environments, it can be observed that GPP and LXC have slightly lower processing time variations than that of DOCKER and KVM, especially when PRB and MCS increase.

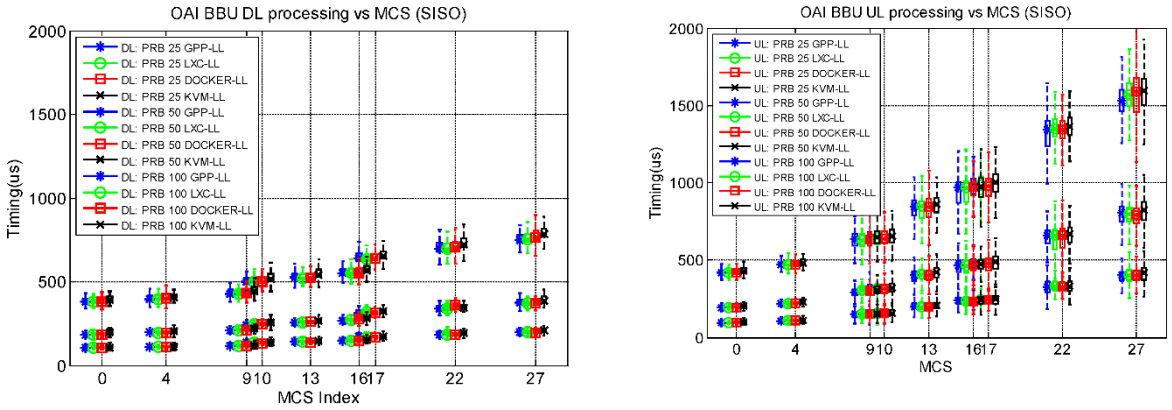


Figure 17 BBU processing budget in downlink (left) and uplink(right) for different virtualized environments

Figure 18 depicts the Complementary Cumulative Distribution Function (CCDF) of the overall processing time for downlink MCS 27 and uplink MCS 16 with 100 PRB. The CCDF plot for a given processing time value displays the fraction of subframes with execution times grater than that value. It can be seen that the execution time is stable for all the platforms in uplink and downlink. The processing time for the KVM (hypervisor-based) has a longer tail and mostly skewed to longer runs due to higher variations in the non-native execution environments (caused by the host and guest OS scheduler). Higher processing variability is observed on a public cloud with unpredictable behaviors, suggesting that cares have to be taken when targeting a shared cloud infrastructure.

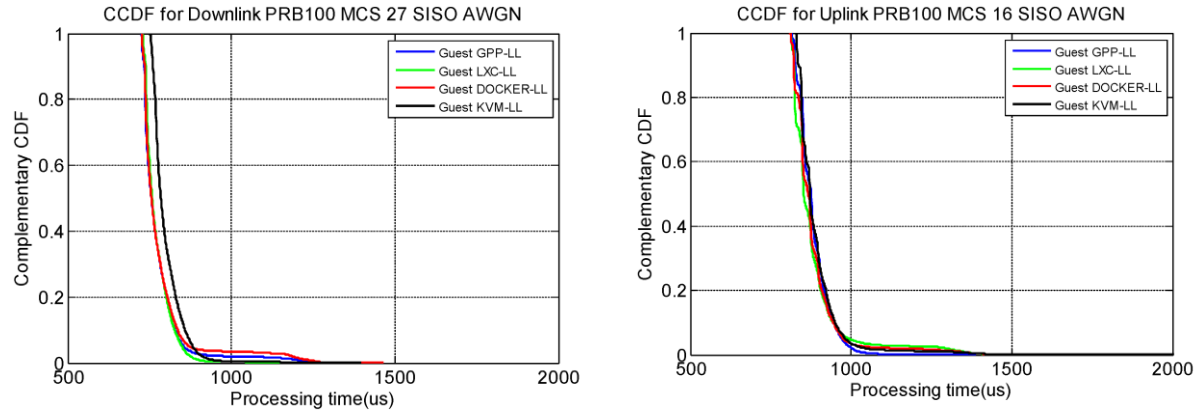


Figure 18 BBU processing time distribution for downlink MCS 27 and uplink MCS 16 with 100 PRB

6.3.2.4 I/O Performance Analysis

Generally, the one-way-delay of fronthaul depends on the physical medium, technology, and the deployment scenario. However in the cloud environment, the guest-to-host interface delay (usually Ethernet) has to be also considered to minimize the access to the RRH interface. To assess such a delay, bidirectional traffics are generated for different set of packet sizes (64, 768, 2048,4096,8092) and inter-departure time (1, 0.8, 0.4, 0.2) between the host and LXC, Docker, and KVM guests. It can be seen from Figure 19 that LXC and Docker are extremely efficient with 4-5 times lower round trip time. KVM has a high variations, and requires optimization to lower the interrupt response delay as well as host OS scheduling delay. The results validate the benefit of containerization for high performance networking.

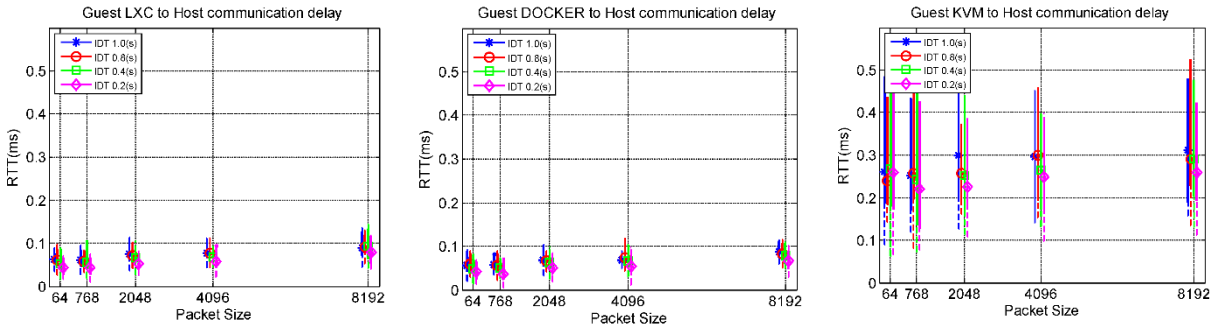


Figure 19 Round trip time between the host and LXC, Docker, and KVM guests

6.3.2.5 Discussion

By analysing the processing for a 1ms LTE sub-frame, the main conclusion that can be drawn for the considered reference setup (FDD, 20MHz, SISO, AWGN) is that with the CPU frequency of 3GHz, (on average) 1 processor core for the receiver processing assuming 16-QAM in uplink and approximately 1 core for the transmitter processing assuming 64-QAM in downlink are required to meet the HARQ deadlines. Thus a total of 2 cores are needed to handle the total processing of an eNB in 1ms (one subframe). With the AVX2 optimizations for this latest architecture, the computational efficiency is expected to double and thus a full software solution would fit with an average of 1x86 core per eNB.

When comparing the results for different virtualization environments, the main conclusion than can be drawn is that containers (LXC and Docker) offer near bar metal runtime (native) performance while preserving the benefits of virtual machines in terms of flexibility, fast runtime, and migration. Furthermore, they are built on modern kernel features such as cgroups, namespace, chroot, and sharing the host kernel and benefit from the host scheduler, which is a key to meet the real-time deadlines. This makes containers a cost-effective yet light-weight solution for RANaaS without compromising the performance.

In summary, the LXC/Docker proved to provide a bar metal performance as they exploit native Linux Features, and do not require a hypervisor. Furthermore, they are becoming more and more popular and well-integrated with OpenStack. However, they do not support multi-OS and require supports from the host OS.

6.3.3 RANaaS Prototype

Following the reference architecture models presented in Section 6.2.1 and the results obtained in section 6.3, a proof-of-concept prototype of RANaaS is built that which compromises of a specific web service which builds on top of the infrastructure foundations (c.f. Figure 20). The web service features a user interface (UI) for the web clients (e.g. MNO, MVNO), a service manager (SM) providing services for the web client, and a service orchestrator (SO) in charge of the RANaaS lifecycle. Through the usage of the Cloud Controller virtual resources are orchestrated and provided to the RANaaS prototype.

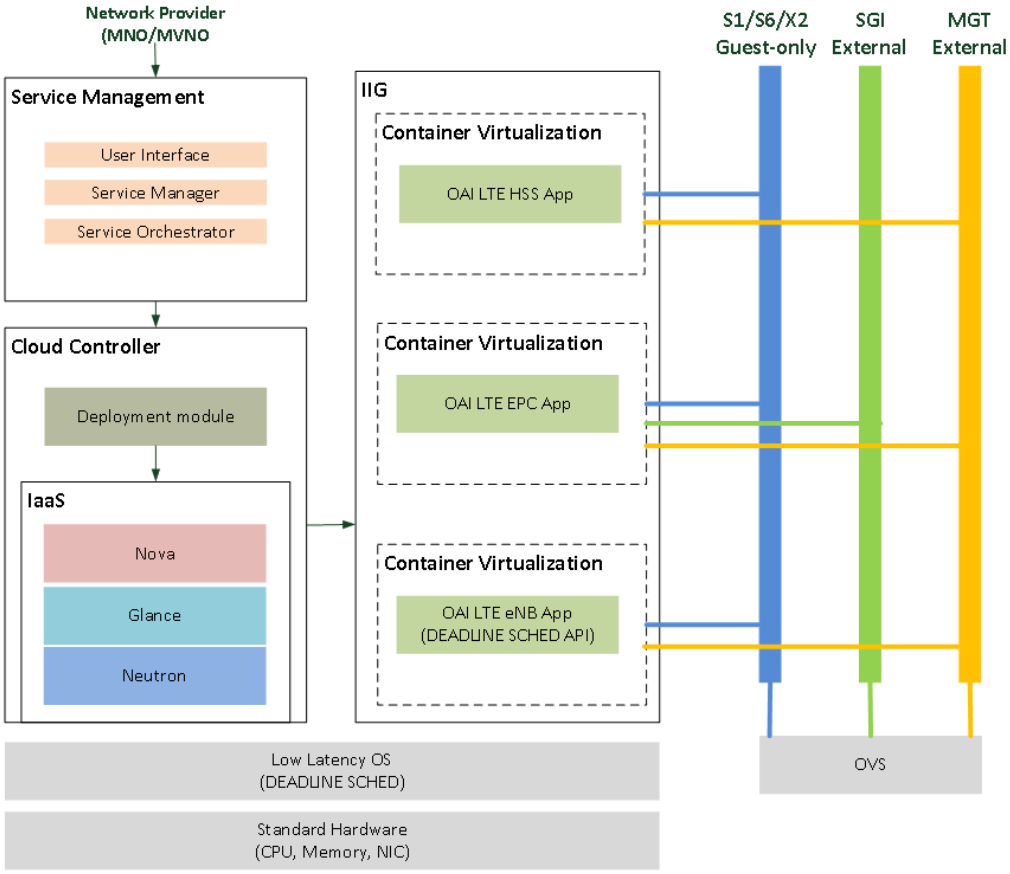


Figure 20 RANaaS prototype

6.3.3.1 Lessons Learnt

The lesson learnt from the above and other experiments can be summarized as follows:

- **Processing time deadline**
  - FDD LTE HARQ requires a round trip time (RTT) of 8ms that imposes an upper-bound for the sum of BBU processing time and the fronthaul transport latency. Failing to meet such a deadline has a serious impact on the user performance.
  - Virtualized execution environment of BBU pool must provide the required runtime.
- **Containers (LXC and Docker) vs Hypervisor (KVM)**
  - Containers are more adequate for GPP RAN as they offer near-bar metal performance and provide direct access to the RF hardware. KVM performance is also good, but requires pass through mechanisms to access the RF hardware to reduce the delay of the hardware virtualization layer.
  - In case of containers, RAN requires low latency kernel in the host.
  - In case of full virtualization (KVM), hypervisor has to support real time/low latency task (different techniques requires for type 1 and 2), and also the guest OS requires low latency kernel.
  - There is need for a dynamic resource provisioning/sharing, load balancing to deal with the cell load variations (scale out and in).

- **Hardware**
    - Probe the existing RF for their capabilities (FDD or TDD, frequencies, transmit power, etc) and select the one that is required for the target RAN configuration.
    - Support of RRH:
        - Fronthaul (BBU to RRH link): a full-duplex 10Gbps link is required. There are certain ETH configuration to be done here.
        - Either EXMIMO 2 (PCI-x if), and/or NI/ETTUS (USB3 if)
- **Flexible Configuration, build, run, and monitoring**
    - (Semi-)Automatic generation of the RAN configuration file through the UI or selection and editing of predefined configuration files. The same holds for the compilation (e.g. Rel 8 or Rel 10) and execution (e.g. enable disable hooks). Example of a config file can be found here: https://svn.eurecom.fr/openair4G/trunk/targets/PROJECTS/GENERIC-LTE-EPC/CONF/enb.band7.tm1.exmimo2.conf
    - Dynamic Monitoring of the status of RAN

6.4 On-going research activities

The following two sections detail the research topics addressed by task 3.5.

6.4.1 Management of virtual radio resources

The aim in this section is to study the performance of the proposed model under the different network traffic load. Hence, the number of subscribers are swept between 300 (i.e., low load) and 1500 (i.e., high load) per VNO. Figure 21 illustrates the distribution of the available virtual resources among the VNOs, in addition to the total network capacity, the total minimum guaranteed, and the contracted data rates. The contracted data rate for each VNO increases from 1912.5 Mbps (low load) to 8925 Mbps (high Load). The acceptable region for VNO GB and BG in the plot are shown by solid blue and light yellow colours. The total minimum guaranteed data rate, i.e., the summation of minimum guaranteed data rates of VNO GB and BG, in the low load is 20.58% of the network capacity (i.e., 1434.375 Mbps). Since the network capacity is considerably higher than the minimum guaranteed data rates, the best effort services are also served well. The allocation of 2446.23 Mbps to VNO BE is the evidence to this claim. It is worth noting that the share of VNO BE is 35.1% of the whole network capacity, which is 1.6 times higher than VNO GB. The reason behind this observation is the maximum guaranteed data rate of the guaranteed services. Although the assigned portion of available resources to VNO GB is not as big as the other two VNOs, it is served up to it maximum satisfaction. In contract, VNO BG that has a minimum guaranteed data rate but no maximum received 43.28% of the network capacity (3016.11 Mbps). The guarantee data rates grows up to 6693.75 Mbps (i.e., 96.05% of the whole available capacity) as the load increases. Obviously, the share of best effort services in this situation considerably decreases. The allocated capacity to VNO BE reduces to only 65.6 Mbps, which is 0.94% of the total available capacity and 97.32% of its initial value.

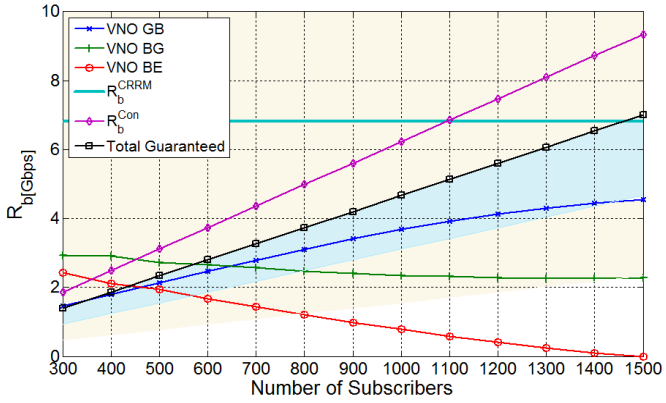


Figure 21 Variation of allocated data to each VNO.

Furthermore, Figure 22 illustrates the effect of demand variation on the allocation of data rates to the service classes of VNO GB. For each of the service classes, the solid colour presents the acceptable region for each service class. It can be seen that the streaming services are the services with the highest volume and they have the highest guaranteed data rate. The interactive, conversational, and background services are place respectively in the next order. It can be seem that in the low load situation the highest possible data rates are assigned. However, as the demands to the network capacity increases, the data rates move toward the lower boundary. The interactive service class is a very good example for this issue. While it has received the highest guaranteed data rate at first, the allocated capacity for the case with 1400 subscriber is altered into the minimum guaranteed. Considering the slope of allocated data rates in various services, the effect of serving weights and the service volume can be seen. Since the interactive class has lower serving weight comparing to conversational, it received almost the minimum acceptable data rate with 1100 subscribers. In the same situation, the conversational services are still provided by the highest acceptable data rate.

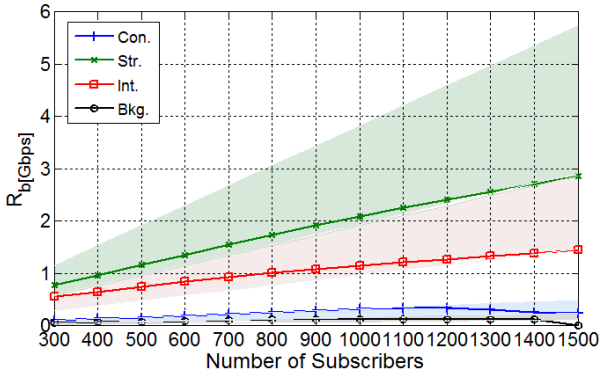


Figure 22 Allocated data rate to service classes of VNO GB.

In the next step, the effect of the channel quality on the management of virtual radio resources by considering the three approaches (i.e. OP, RL, and PE) is studied. Figure 23 presents the allocated data rates to VNO GB in conjunction with minimum and maximum guaranteed data rates. As long as these data rates are in acceptable region (i.e., shown by the solid colour), there is no violation to the SLAs and guaranteed data rates. It is obvious that in optimistic approach up to 600 subscribers, the maximum guaranteed data rate is offered to the VNO. Considering the other approaches, as the number of the subscribers increases the allocated data rate moves toward the minimum level of the guaranteed data rate. However, in the pessimistic approach as the number of subscribers passes 1100, violations to minimum guaranteed data rate can be observed. It means that the network capacity in this

approach lower than the total minimum guaranteed data rates. The model has to violate some of the minimum guaranteed data rates. This figure shows the effect of input SINR on resources usage efficiency and quality of offered service to the VNOs.

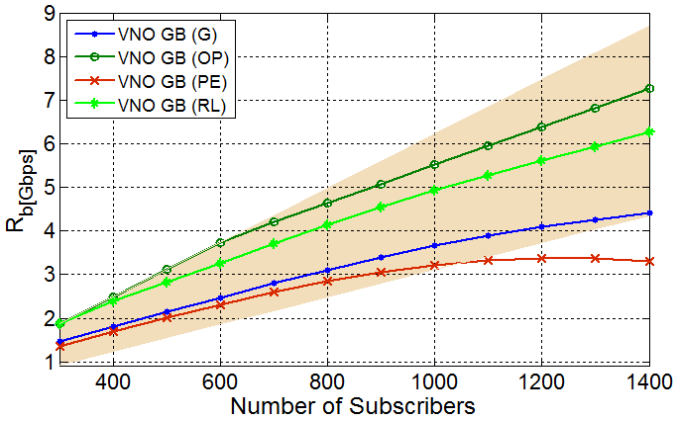
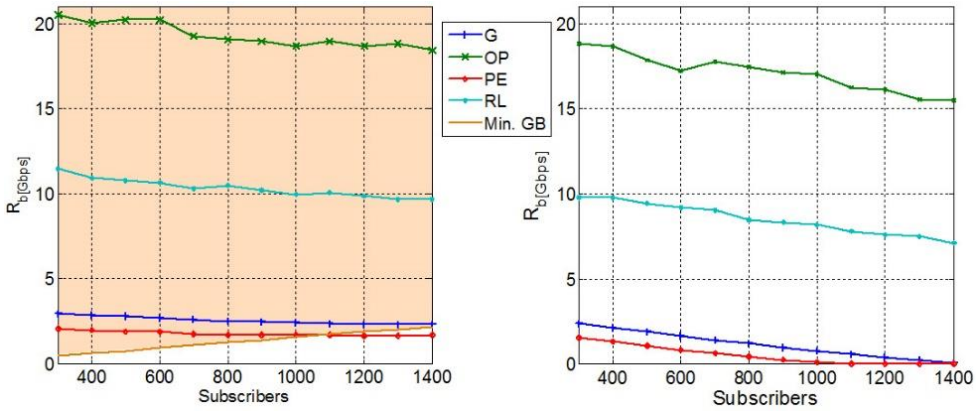


Figure 23 The allocated data rate to VNO GB in different approaches.

The allocate data rates to VNO BG and BE are also plotted in Figure 24 Just the same as VNO GB, it can be seen that high data rate is allocated to these VNOs in OP and RL approaches. In these cases, the high input SINR leads to the high network capacity and the model is not only able to serve the minimum guaranteed data rates, but it can serve acceptable data rates to the best effort (and best effort with minimum guaranteed) VNO. In the high load situations, the VNO BG and BE suffer more from resources shortage. The allocation of resources to VNO BE even stops when more than 1100 subscribers under PE approach is considered.



a) VNO BG **b) VNO BE**

Figure 24 The allocated data rate to VNO BG and VNO BE in different approaches.

Regarding to the distribution of allocated data rate to VNO GB, Figure 25 illustrates the variation of the capacity assigned to each service class of this VNO in different approaches. It can see that the conversational class, the class with the highest service weight, for the OP and RL approaches received the maximum guaranteed data rate. The allocated data rate for general and pessimistic approaches placed in the acceptable region. Although for high-density situations in PE case, the data rate decreases to minimum guaranteed data rate, services of this class never experience violation of guaranteed data rate. Streaming class, likewise, in this VNO is always served with the data rate higher than minimum guaranteed. Since the maximum guaranteed data rate for this class is too high and it has the second high serving weight, the assigned capacity did not reach to maximum. For interactive and

background classes, it is shown that they face violation of minimum guaranteed data rate in PE approach. The violation situation in background class is more severe, since the services did not received any data rate as the number of subscribers pass 1100 subscribers.

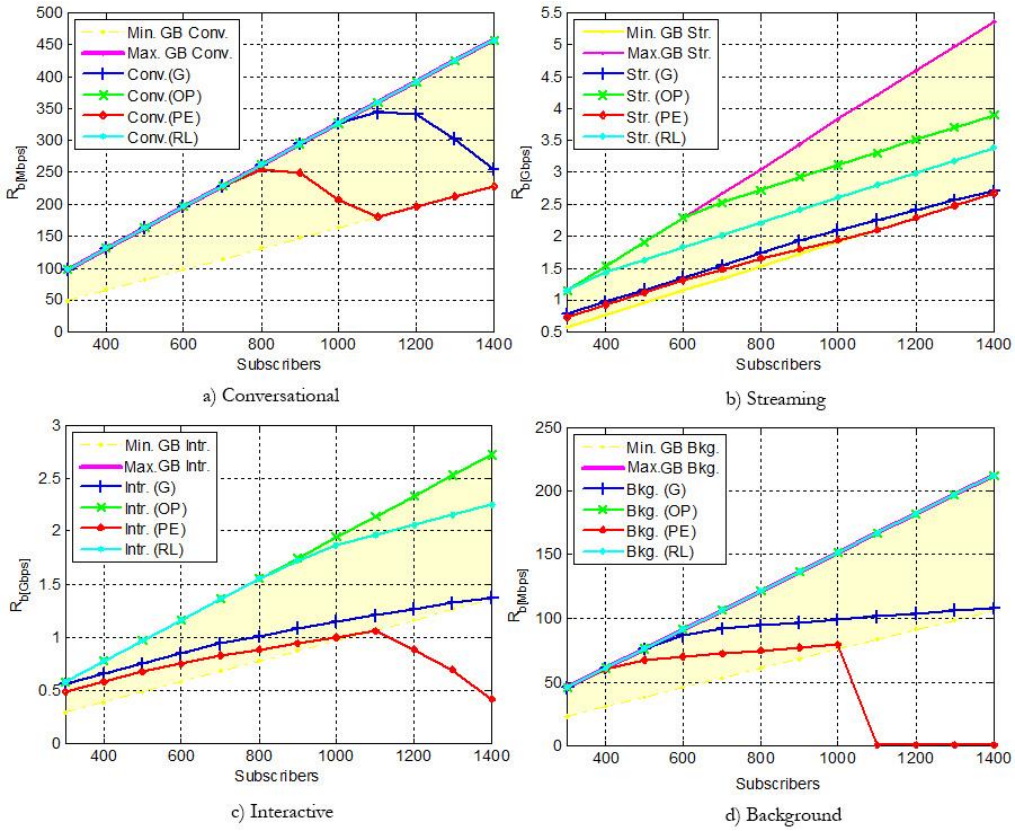
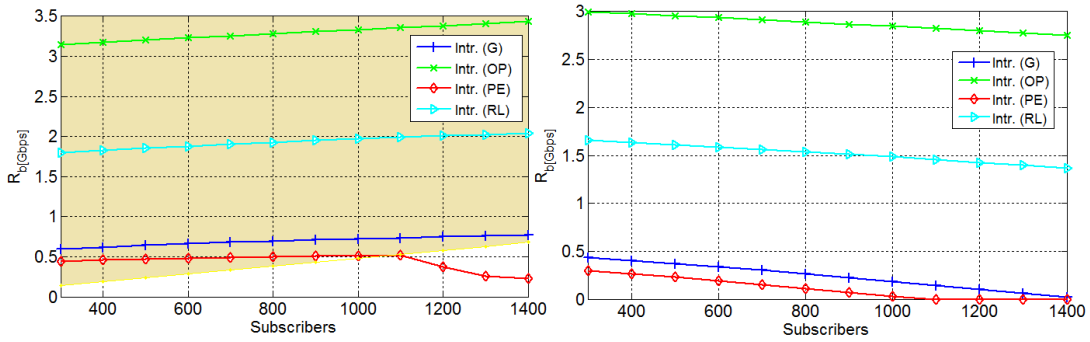


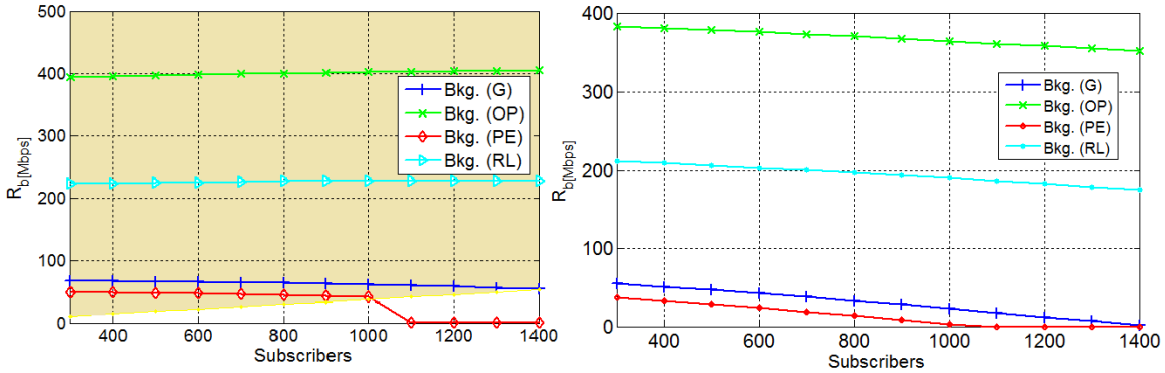
Figure 25 The allocated data rate to service classes of VNO GB.

Moreover, the allocated data rate to the interactive and background classes of VNO BG and BE is demonstrated respectively in Figure 26 and Figure 27. It can be seen that for the VNO BG the situation is very much similar to VNO GB. The main difference is the high boundary of allocated data rate. VNO BG does not have a maximum guaranteed data rate or high boundary for allocation of data rates. Consequently, when high network capacity is available, e.g., in OP situation, the services of this VNO are served by comparatively higher data rates comparing to VNO GB. As an example to this discussion, consider conversational class on both VNOs. For optimistic approach with 400 subscribers, VNO GB is granted with 100 Mbps where VNO BG receives 1.3 Gbps. On the other hand, in case of resource shortage, the VNO BG received lower data rates than the VNO GB. For instance, the share of interactive class of VNO BG when there are 1200 subscribers in PE approach is only 386.63 Mbps while the VNO GB is allocated by 907.77 Mbps.



a) VNO BG **b) VNO BE**

Figure 26 The allocated data rate to interactive class.



a) VNO BG **b) VNO BE**

Figure 27 The allocated data rate to background class.

6.4.2 Integration of VRRM in OAI

This implementation is to demonstrate the download operation of two groups of users, belonging to different Virtual Network Operators (VNOs), providing different type of services with pre-defined requirements as shown in Figure 28.

The main goal is to show that, if two VNOs want to provide service with different requirements/contracts, pushing a button launches an eNB via the RANaaS architecture, and UEs of each VNO do perform according to their SLAs (a gold service and a best-effort (BE)/for free service). This will be reported upon in future deliverables from WP6.

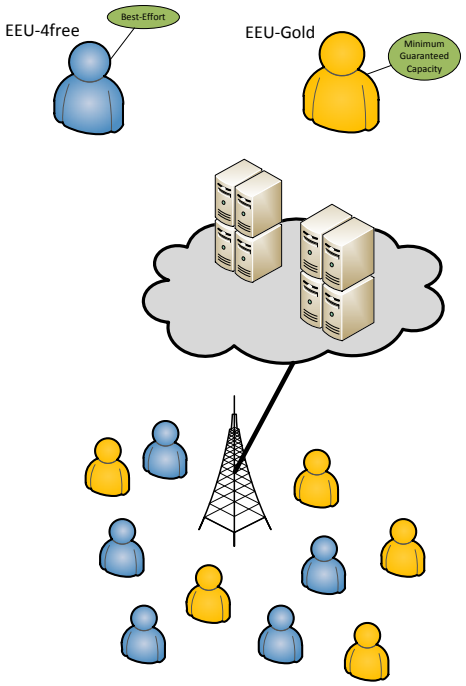


Figure 28 Scenario for VRRM demonstration.

6.4.2.1 Implementation

The concept of virtualisation of radio resources and the comprehensive management models are presented in (D3.1 2013) and (D4.3 2014). For implementation of them (i.e., the concept and model), it is assumed that a cloud host with Open Air Interface (OAI) server is available. OAI is a software-based LTE eNodeB developed in Linux used as infrastructure emulator. In addition to OAI servers, a windows-based Virtual Radio Resource Management (VRRM) server is also deployed on the same network domain. These servers are connected through an internal network managed by cloud provider. Hence, it can be assumed that these links can carry information and control signals among servers.

The primary goal is to integrate VRRM server with OAI for realisation of the aforementioned concept. The VRRM server based on information and statistics from the infrastructure emulator makes decisions, and issues policies. Figure 29 shows the chosen approach that is to define a bidirectional interface between OAI and VRRM server to:

- Transfer the scenarios and configurations from OAI to VRRM,
- Send calculated policies from VRRM to OAI,
- Retrieve real-time statics and information of VNOs form OAI for VRRM,
- Send updated policies from VRRM to OAI.

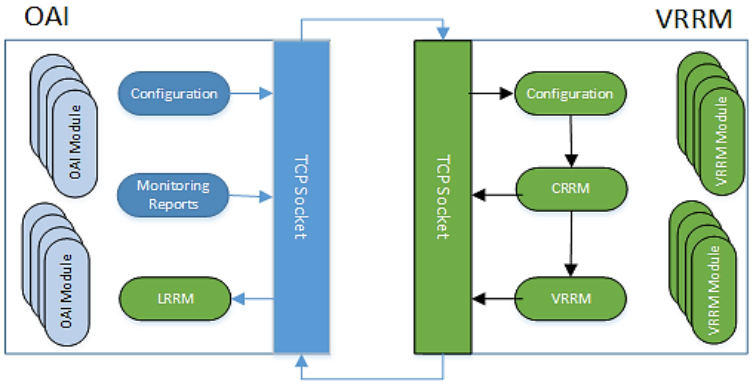


Figure 29 Simplified block diagram of VRRM and OAI interaction.

In addition, changes in OAI is required to support the virtualisation concept. These changes enable OAI to have multiple groups of subscribers, each one representing the subscribers of a VNO. With this approach, it is possible to impose different policies per VNO/group, offering different service quality to terminals according to the VNO/group they belong to, and collecting statistics of operation separately in order to adapt the scheduler to meet the diverse requirements. The following modification on OAI was defined for the implementation:

- Adding group-based statics to eNodeB;
- Introducing new set of long-time statics using the already implemented listing concept;
- Changing the codes to initialise, fill, and use the aforementioned statics;
- Changing the algorithm of MAC scheduler in order to support the groups policies;
- Adding support for bidirectional connection and the required protocols;
- Changing the codes to add the groups' information into the XML file.

More details on the implementation of this Scenario can be found in section A.2.

6.5 Integration with other tasks

Next to the usage of the services described earlier in this deliverable one of the key integration points is between this task and WP4.

The integration of OAI eNB with OpenEPC has started at the premises of both parties, namely Eurecom and Fraunhofer FOKUS / TUB. So far, the integration is successful and that the attach procedure is fully supported from a commercial UE connected to the OAI eNB and openEPC. In addition, basic interoperability testings have been performed by both parties validating the control and data planes compatibility with the respective 3GPP standards. In particular, we observed successful attach procedure and data transmission between COTS UE and openEPC. Nevertheless, few issues have been detected including (1) a mismatch of S1 SCTP stream identifier at eNB and EPC, (2) improper handling of unexpected UEcontext release triggered by eNB, (3) error is certain GTP/IP packets when the MTU is larger than 1512 bytes causing fragmentation at IP layer, and (4) OAI eNB S1 reconnection after losing EPC connection. Currently, the above issues are under investigation and they will be fixed by responsible partner. A detailed report on the integration will be given as a part of the WP6 deliverable.

6.6 Conclusions

This document shows all final architectural changes, evaluations, and developments carried out in Task 3.5 towards offering RANaaS, and describes the lessons learnt from the experiments.

A reference architecture model is presented describing different choices of hardware platform, service compositions, and virtualization techniques. Some evaluations have been conducted to analyze the RANaaS performance. It especially characterizes the BBU processing time under various experimental scenarios such as different x86 CPU architecture, different CPU frequency, and different virtualization environments. The I/O performance of virtual Ethernet interface is also measured. Following the reference architecture models and the performance results, a proof-of-concept prototype of RANaaS is built.

Finally, the concept of virtual RAN is studied. First, its performance is analyzed under the different network traffic load. Secondly, the way it is integrated within OAI is described.

7 Services of Category Support

The following sections detailed services which have been evaluated, tested or implemented by task 3.1 and 3.4 mainly to support the requirements which were raised towards WP3 by other work packages.

7.1 Domain Name System-as-a-Service

The Domain Name System (DNS) is an extensible, hierarchical distributed naming system for resources connected to the Internet or private networks. DNS-as-a-Service (DNSaaS) allows the creation of on-demand DNS servers capable of being configured through a well-defined API, as documented in (D3.1 2013), (D3.2 2014), and (D3.3 2014) deliverables.

The DNSaaS architecture has not been modified since the final report on (D3.3 2014). In fact, the DNSaaS solution relies on existing open source DNS components:

- PowerDNS that runs on DNS backend Servers.
- Designate is an OpenStack component used by the DNSaaS API to manage the domain information.

As presented in the final report of MCN components, (D3.3 2014), the functional elements of DNSaaS include:

- DNSaaS API that makes the interface with DNS Servers and DBaaS to manage the DNS information.
- DNS Servers that perform the process of DNS requests, relying on PowerDNS functionalities.
- DNS Forwarders that act as frontends to DNS clients by forwarding the queries to the DNS Servers.
- DBaaS that holds all the DNS information of all the records.

The interfaces of these functional elements have not changed.

The architecture of DNSaaS has been validated in MCN as an enabler for sophisticated, fully managed, auto-scalable cloud-based DNS services. The virtualization platform, Openstack based, proved in the results to be a good choice in terms of the flexibility offered to deploy the service and with good performance levels (see section A.1). The scaling algorithm proved to be effective regarding the adaption to the load introduced by diverse simultaneous clients.

DNSaaS has been employed in MCN as a support service. The integration with the MCN architecture included a method in the MCN CC SDK to enable other services to have an object to interface with the DNSaaS API.

7.2 Load Balancing Service

The LBaaS provided by OpenStack has been evaluated and tested. It has been deployed in the testbeds as needed to. However it is noted here that the LBaaS solution provided does not account for very specific protocol needs as seen in WP4 for EPCaaS. A specific solution for that will – if needed – be implemented by WP4.

7.3 Analytics Service

The implementation details of the analytics service has been detailed in (D3.3 2014). Since then the analytics service has shown great value – as being a general purpose tool when applied to performance analysis of services.

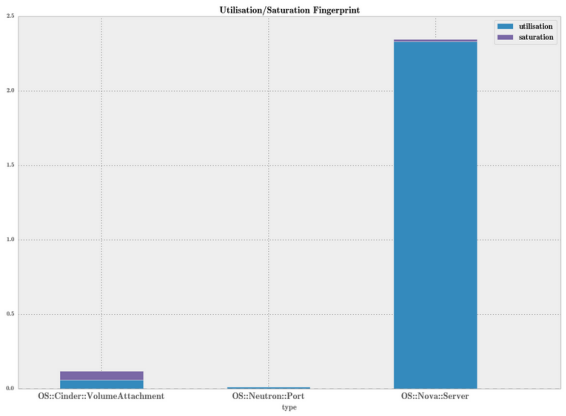
As the ITG defines a set of required resources for a SI – the IIG describes the actual virtual instances and how they are entangled. As defined earlier these can be represented as a graph. For the analytics service we have define methods which can be used in the workbooks to derive these IIGs from the deployment module for any deployed service.

Once the IIG is known it can be used as the high level block diagram as defined in the USE methodology – which was evaluated in (D3.1 2013). Each node in the IIG graph can be contextualized with usage information such as there Utilisation and Saturation. This can be done thanks to the fact that each node in the IIG has a certain type. And for each node (and hence the resource instance) the MaaS can be queried to retrieve this value. By deriving Utilisation and Saturation values for each node in the graph of the IIG we can get a high level understanding the service in place and its needs on the Infrastructure.

This method of fingerprinting and hence classifying any kind of workload has deemed very useful for performance evaluations of service. Based on the USE methodology as we defined it in (D3.1 2013) for task 3.2, and the IIG it is very easy to determine if SI has more need for compute, network or storage resources and based on that make decision for better placement and optimisations. The following screenshot in Figure 30 shows the analytics service in action with an example fingerprints for an IIG which consumes mostly CPU cycles and is light on network and storage traffic.

mcn_storage

Fingerprint was created for the mcn_storage stack and stored in the fingerprint database.



name	type	utilisation	saturation
server_1	OS::Nova::Server	2.367100	0.02
server_2	OS::Nova::Server	2.298500	0.01
server_port	OS::Neutron::Port	0.010192	0.00
server_port_2	OS::Neutron::Port	0.013376	0.00
volume_1_attach	OS::Cinder::VolumeAttachment	0.000000	0.00
volume_2_attach	OS::Cinder::VolumeAttachment	0.120000	0.12

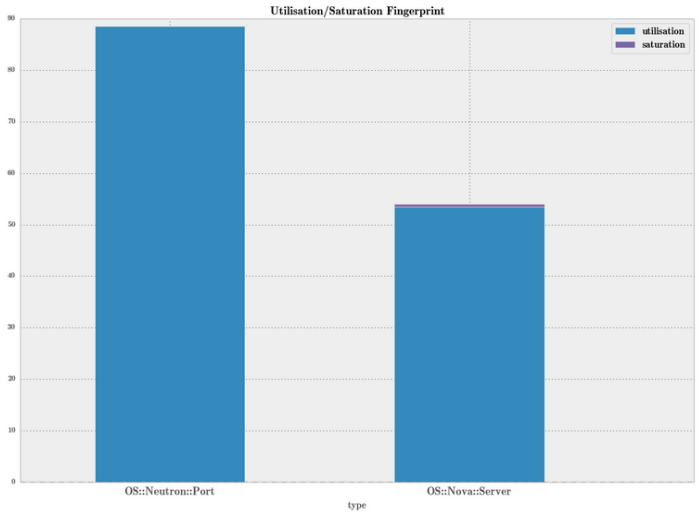
Figure 30 Automated fingerprint in the Analytics Service

The graphs show the aggregated Utilisation and Saturation values for certain types within an IIG. The table shows the details per resource instance.

Fingerprints now indicate whether a service is more compute, storage or network bound. Figure 31, for example, shows a fingerprint of a service which is more network oriented.

mcn_network

Fingerprint was created for the mcn_network stack and stored in the fingerprint database.



name	type	utilisation	saturation
server_1	OS::Nova::Server	55.200100	0.55
server_2	OS::Nova::Server	51.749100	0.42
server_port	OS::Neutron::Port	88.363100	0.00
server_port_2	OS::Neutron::Port	88.758044	0.00

Figure 31 Fingerprint of a network orientated service

Fingerprints can be generated for any given point in time and stored in a database. By applying machine learning algorithms, patterns can be abstracted over time. And then the outputs of those algorithms (in form of models) can be offered to trigger actuations. The analytics service offers functionalities to trigger actuations and talk back to the deployment module of the CC.

7.4 Database Service

OpenStack Trove (Trove 2014) has been evaluated as DBaaS and deemed to satisfy the requirements of the other services. The service has been deployed in the testbed of the MCN project consortium members.

8 Summary and Outlook

As this is the final deliverable of the work package each task reported on its final findings and results. All service implemented within the work package and the corresponding tasks have been reported upon again to give a complete overview of all the achievements. Individual details on implementations and prototypes have earlier been reported upon in deliverables (D3.2 2014) and (D3.3 2014).

The authors are extremely proud to show that the architecture and concepts as they have been defined in (D3.1 2013) have proven to be robust and only – if any – required minor adjustments. This is a very important concept for the infrastructure management foundations of a project such as MCN as other work packages build upon it.

Combining low-level service such as networking functions, performance and instrumentation related topics and offer them out via a central component like the CloudController has deemed to be very powerful and fostered the concepts of close collaboration within the work package. Offering connectivity service – like the RANaaS – made it possible within the work package to proof the basic features but also show that in the future of the Mobile Cloud connectivity is a foundational concept and service in need.

As foundations might change over time, due to changing technology and concepts, a lot of work has gone into providing a robust foundation which is not dependent on a particular technology. This was done by providing basic – yet standardized – interface such as OCCI. This obviously also holds for interface for integration such as the S1 interface which enables the integration of EPCaaS and RANaaS.

The way of wrapping existing cloud technologies/services, integrating service of any kind, and allow for the ease-of-development with the Service Development Kit can certainly be seen as one of the major outcomes of the project. Other than just a high level concept and architectures it provides a very robust – and proven in code – foundation for building services within the project and for the future to come.

Implementations of all the service presented in this deliverable will be carried over to the work of work package 6 where further integrations and evaluations will be conducted.

9 Terminology

AAA – Authentication, Authorization, and Accounting

AaaS – Analytics-as-a-Service

aaS – -as-a-Service

ACA – Admission Control Algorithm

API – Application Programming Interface

ARP – Address Resolution Protocol

ASP – Application Service Provider

AWS – Amazon Web Services

BBU – Baseband Unit

BS – Base Station

BSS – Business Support System

CAPEX – Capital Expediture

CC – CloudController

CDN – Content Delivery Network

CDNaaS – CDN-as-a-Service

CLI – Command Line Interface

CMMS – Common Monitoring Management System

COTS – Commercial Off The Shelf

CPRI – Common Public Radio Interface

CPU – Central Processing Unit

C-RAN – Cloud-RAN

CRUD – Create Retrieve Update Delete

CSP – Cloud Service Provider

DB – Database

DBaaS – Database-as-a-Service

DBMS – DB Management System

DC – data centre

DHCP – Dynamic Host Configuration Protocol

DMM – Distributed Mobility Management

DNS – Domain Name System

DNSaaS – DNS-as-a-Service

DoW – Description of Work

DRAM – Dynamic RAM

DSS – Digital Signage System

DSSaaS – DSS-as-a-Service

E2E – end-to-end

EEU – Enterprise End User

eND – eNodeB

EPC – Evolved Packet Core

FLOPS – Floating Point Operations

FMA – Fault Management

FMC – Fundamental Modelling Concept

FPS – Frames Per Second

FTP – File Transfer Protocol

GSM – Global System for Mobile communications

GTP – GPRS Tunnelling Protocol

GUI – Graphical User Interface

GW – Gateway

HD – Hard Disk

HDD – Hard Disk Drive

HTTP – Hypertext Transfer Protocol

HTTPS – Hypertext Transfer Protocol Secure

I/O – Input-Output

IaaS – Infrastructure-as-a-Service

ID – Identifier

IETF – Internet Engineering Task Force

IEUE – Individual End User Equipment

IMDB – Infrastructure Management DB

IMS – IP Multimedia Subsystem

IMSaaS – IMS-as-a-Service

IOPS – Input/Output Operations Per Second

IP – Internet Protocol

IPMP – Multi Path IP

IPv4 – IP version 4

IPv6 – IP version 6

IRT – Infrastructure Response Time

iSCSI – Internet Small Computer System Interface

ITG – Infrastructure Template Graph

KPI – Key Performance Indicator

KVM – Kernel-based Virtual Machine

LAN – Local Area Network

LBaaS – Load Balancer-as-a-Service

LTE – Long Term Evolution

MaaS – Monitoring-as-a-Service

MAC – Medium Access Control

MCNP – Mobile Core Network Provider

MEF – Metro Ethernet Forum

MIPS – Million Instructions Per Second

MVNO – Mobile Virtual Network Operator

NaaS – Network-as-a-Service

NAPTR – Name Authority Pointer

NCP – Network Connectivity Provider

NFV – Network Function Virtualization

NIC – Network Interface Card

NUMA – Non-Uniform Memory Access

OCCI – Open Cloud Computing Interface

OFC – OpenFlow Controller

OFS – OpenFlow Switches

OGF – Open Grid Forum

OID – Object ID

ONF – Open Networking Foundation

OPEX – Operational Expenditure

OS – Operating System

OSS – Operation Support System

P2P – Peer-to-Peer

PaaS – Platform-as-a-Service

PDCP – Packet Data Convergence Protocol

PHY – Physical Layer

QoE – Quality of Experience

QoS – Quality of Service

RAM – Random Access Memory

RAN – Radio Access Network

RANaaS – RAN-as-a-Service

RANP – RANaaS Provider

RCB – Rating Charging Billing

RI – Resource Instance

RLC – Radio Link Control

RRC – Radio Resource Control

RRH – Remote Radiohead

SAN – Storage Area Network

SD – Service Develop

SDK – Service Development Kit.

SDN – Software Defined Networking

SDO – Standard Developing Organization

SI – Service Instance

SIC – Service Instance Component

SIMD – Single instruction, multiple data

SLA – Service Level Agreement

SM – Service Manager

SMF – Service Management Facility

SNMP – Simple Network Management Protocol

SO – Service Orchestrator.

SOB – Service Orchestrator Bundle

SoIP – Storage over IP

SP – Service Provider

SQL – Structured Query Language

SR – Service Requester

SRV – Service record

SSD – Solid State Drive

SSH – Secure Shell

SSP – Support System Provider

STG – Service Template Graph

TCP – Transmission Control Protocol

TE – Terminal Equipment

TOE – TCP/IP offload engine

TTL – Time To Live

UE – User Equipment

UMTS – Universal Mobile Telecommunication System

URL – Universal Resource Locator

VIP – Virtual IP

VLAN – Virtual LAN

VM – Virtual Machine

vNIC – virtual NIC

VNO – Virtual Network Operator

VPN – Virtual Private Network

VPNaaS – VPN-as-a-Service

VRRP – Virtual Router Redundancy Protocol

WAN – Wide Area Network

WLAN – Wireless LAN

WP – Work-Package

A Appendix

A.1 Performance Evaluation of DNSaaS

This section describes the evaluation of the DNSaaS with the goal to validate the DNSaaS architecture and the virtualization solutions for this service (e.g. based on containers or virtual machines). With this in mind, two types of evaluation were performed:

- **Virtualization performance** -- assesses the performance of a DNS backend in different virtualization solutions, which include containers, bare metal and virtual machines on different versions of Openstack.
- **Performance scaling** -- assesses the performance of the scaling solutions introduced in the DNSaaS reference architecture, namely to assess the gain within a different number of DNS backends.

The following subsections detail the evaluation methodology and the results achieved with the evaluation.

A.1.1 Virtualization performance

The virtualization performance includes performance assessment in different infrastructures with the same characteristics in terms of CPU and memory. All the testes have been performed on a server with two (virtual) CPUs and 8Gb of RAM. The diverse virtualization approaches used in the testes are detailed in the Table 8.

Identification of infrastructure / Testbed	Type of Virtualization / Hardware	Version of virtualization
Bare Metal	N/A	N/A
Docker	Docker container	
Bart (ZHAW)	Openstack	Juno
Bern (University of Bern)	Openstack	IceHouce

Table 8 Virtualization approaches

The tests included load of DNS queries determined by the number of simultaneous clients and DNS queries. Table 9 summarizes the load introduced within the diverse number of clients. The clients use DNSperf (Nominum 2015) to perform the DNS requests in the server configured in different virtualization solutions. The DNS records are configured in the server prior to the tests and all the DNS caching mechanisms in the server are deactivated.

Test ID	Number of Clients	Query load introduced by clients
1	1	50k
2	10	500k
3	25	1250k

4	35	1750k
5	50	2500k

Table 9 DNS query load and number of clients

A.1.2 Performance scaling

The performance scaling included the same load as the virtualization performance testing detailed in Table 9. In this test the number of DNS Servers is variable according to the load in the DNS forwarder and the scenario under evaluation. Two complementing scenarios were included in the evaluation, a centralised – depicted in Figure 32 and a distributed – depicted in Figure 33. The centralised approach relies on a central database, while the distributed includes a replication of the database information in the diverse DNS Servers. During the evaluation process, all the databases have been configured using MySQL, validated in previous works as the database engine for DNSaaS (Bruno Sousa et al. 2014). All the DNS data was entirely loaded into memory due to its reduced size, even with millions of records and domains. Such kind of setup was mainly configured to avoid I/O issues reported in database engines, when retrieving data from disk storage. Note that the Database as a Service (DBaaS) could be employed. Nonetheless, the goal in this evaluation is to assess the performance of the diverse components of the DNSaaS architecture.

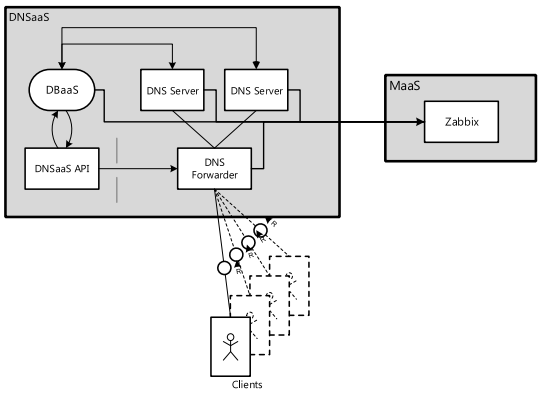


Figure 32 Centralised evaluation scenario

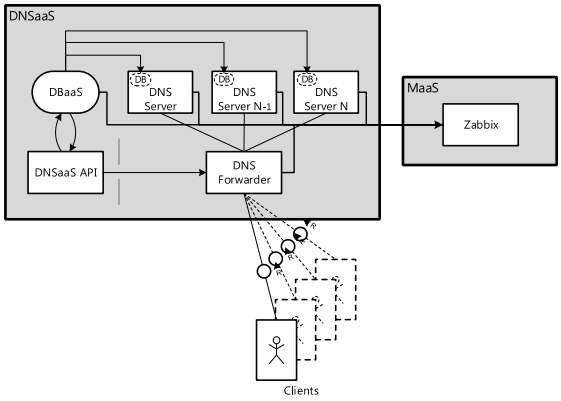


Figure 33 Distributed evaluation scenario

In order to accurately retrieve all the necessary metrics without imposing additional overhead on the DNSaaS components, a monitoring solution was included, which in a cloud-based paradigm could be provided by the Monitoring as a Service (MaaS). The collected information was kept in a separate machine, avoiding as much as possible, to create disturbances on the overall DNSaaS, and keeping compatibility with best practices in enterprise networks regarding monitoring of services. Regarding the scaling and elasticity goals for DNSaaS, such monitoring approach also plays an important role in the decision and prediction process of any scaling operations, collecting also infrastructure-related metrics, such as CPU Load and the available memory.

The scaling decisions were performed with an optimization algorithm that aims to reduce the number of Service Level Object (SLO) violations.

A.1.3 Performance Results

This section presents the performance results of the evaluation performed in the virtualization and performance scaling evaluations.

A.1.3.1 Virtualization Performance

The results are presented in terms of the supported queries per second (qps) and their respective latency considering the simultaneous clients. Figure 34 illustrates the query throughput in terms of queries per second. As expected the bare metal server supports a better throughput, around 12000 qps within 50 simultaneous clients. The worst virtualization environment corresponds to docker, which only is able to support 6500qps within the same number of clients. Such kind of performance gap has also been achieved in similar works considering UDP traffic (Morabito et al. 2015). The networking port mapping option was used in docker to map the DNS port (e.g 53) to a port inside the container (e.g 5000), and has verified in the results, this introduces a considerable delay, affecting the performance of the service. Both opestack-based approaches are better than docker container approach. The difference between the two versions of Openstack corresponds to the deployment options of Openstack, namely hardware. For instance the IceHouse version was deployed in a Dell PowerEdge R520 server.

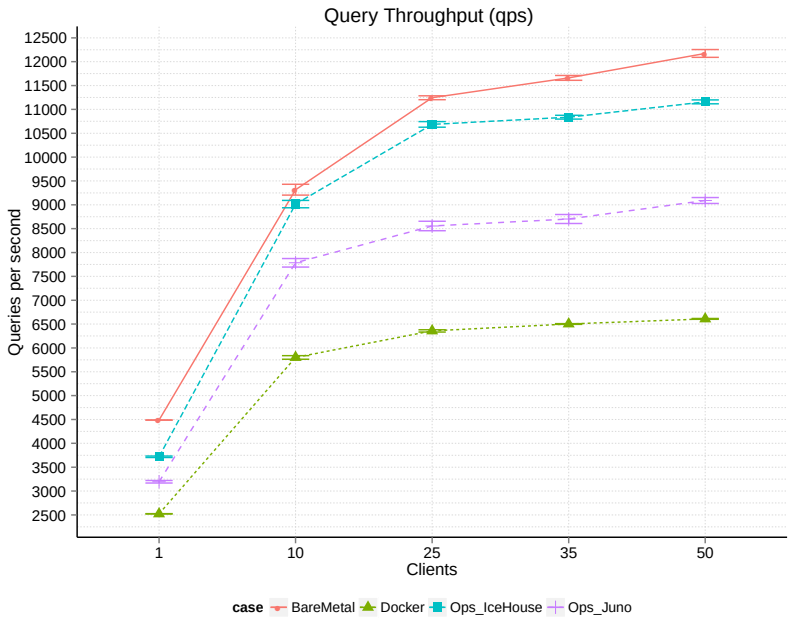


Figure 34 Query throughput (queries per second)

Figure 35 depicts the results of the query latency for the diverse clients. The achieved results are inline with the query throughput, having docker container with the worst performance and the base metal as the most performant. Indeed the performance gain almost doubles between the bare metal and the docker as the number of clients increases.

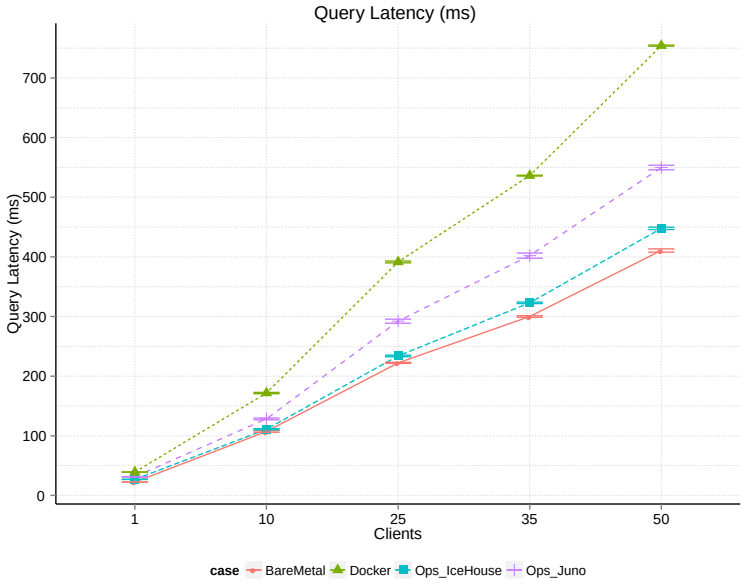


Figure 35 Query latency (in ms)

One of the advantages docker corresponds to the activation of the service, this means as soon as the software is configured in the container (database, PowerDNS server and software dependencies), the deployment of the DNSaaS is performed almost instantly. Within the openstack solutions the DNSaaS deployment and provisioning laid in times around 30s.

A.1.3.2 Performance scaling

The scaling performance is analysed in different perspective, the PowerDNS servers/Forwarder perspective and the client perspective. The former includes an analysis of the query latency, timeouts and error ratios observed in the recursor and DNS Servers. The latter includes the same metrics used in the virtualization performance testing, namely the query throughput and the query latency.

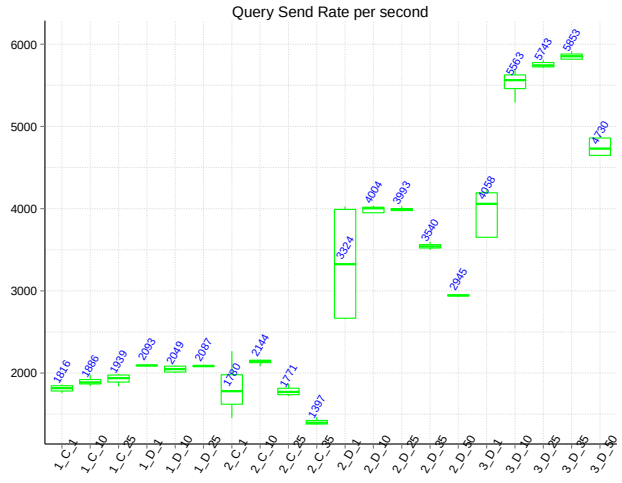


Figure 36 Query Send Rate (per second)

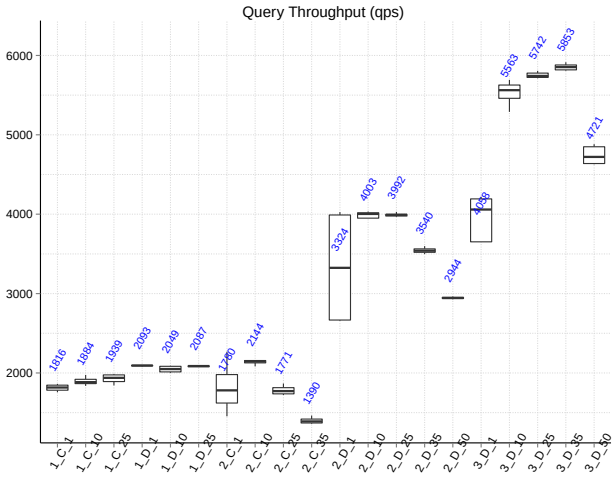


Figure 37 Query Throughput (per second)

Figure 36 depicts the query send rate and Figure 37 illustrates the query throughput in terms of queries per second. The query send rate is determined based on the equation depicted bellow and considers the total queries sent within the respective run time. Note that the employed tool in the tests, (i.e.

DNSperf) allows a sequential sending of queries, and all the configurations to maximize the send rate were performed.

$$query\ send\ rate = \frac{Total\ queries\ sent}{run\ time}$$

The DNSaaS architecture, as expected is able to support a better performance with three servers, indeed the best throughput is achieved with a load of 1.75M requests. With 2.5M the DNS clients experience timeouts and DNS requests losses. The performance improvement with three servers is also present with the latency processing, as depicted in Figure 38.

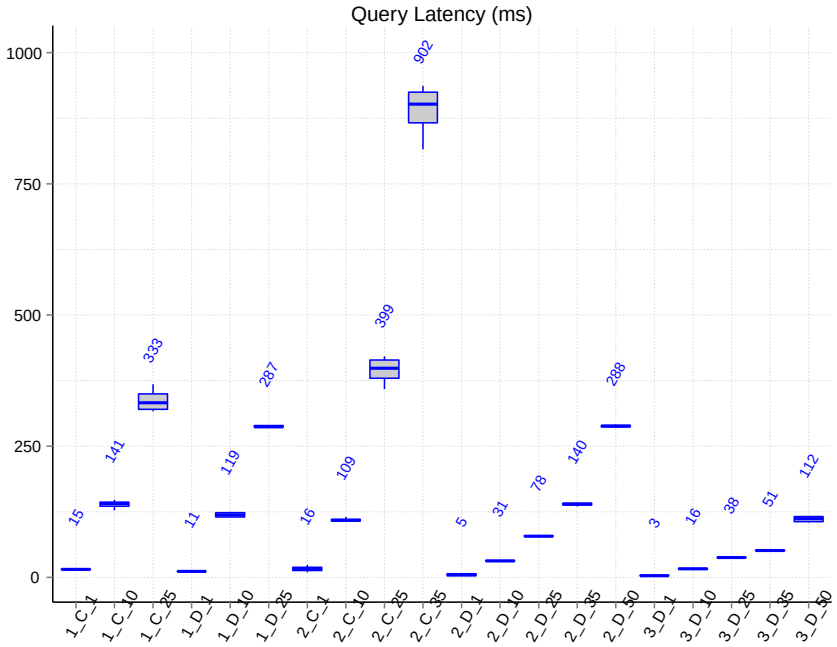


Figure 38 Query latency (ms)

It can be observed that the centralized scenario does not provide the best performance. The concurrency introduced by the diverse servers introduces a considerable impact in the database. Indeed, the three servers allow a performance gain in terms of query throughput and query latency.

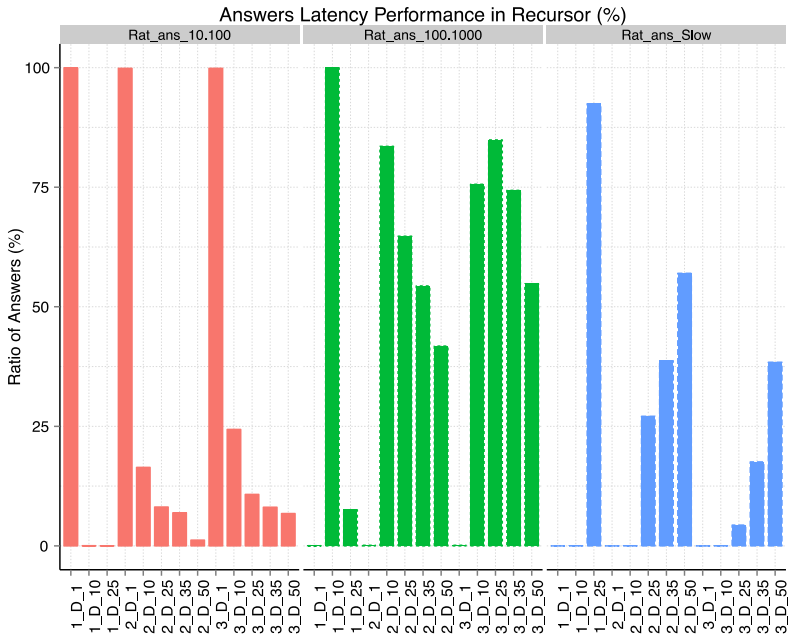


Figure 39 Ratio of answers latency performance in percentage.

Figure 39 depicts the ratio of answer regarding the latency performance. The centralized tests as well as the ratio of answer bellow 10ms are omitted. Rat_ans_10.100 corresponds to the ratio of answers provided in the 10 and 100ms range, Rat_ans_100.1000 to the answers between 100 and 1000ms, Rat_ans_Slow to the answers between 1000ms and 2000ms. The answers exceeding 2000ms are marked as timeouts. For a low volume of DNS queries, performed by single client (i.e., 50K), all the answers are processed in the range be- tween 10 and 100ms. Nonetheless as the number of clients increases, a majority of the replies takes longer than 100ms. In fact, referring back to the latency registered on the client-side for 1.25M DNS queries (25 clients), with a single DNS server, the time taken to complete the answers are above 1000ms.

Once again proving the correct performance of the proposed architecture, when scaling by adding another DNS server to the DNSaaS service, it is clear that the ratio of slow queries (i.e., Rat ans slow) is inferior when compared to tests performed against the same number of clients but using only one or two DNS servers. This becomes more noticeable when analysing the ratio of Timeouts and ServFails, which increase with the volume of DNS queries, as presented in Figure 40 but that become diluted when increasing the number of available DNS servers.

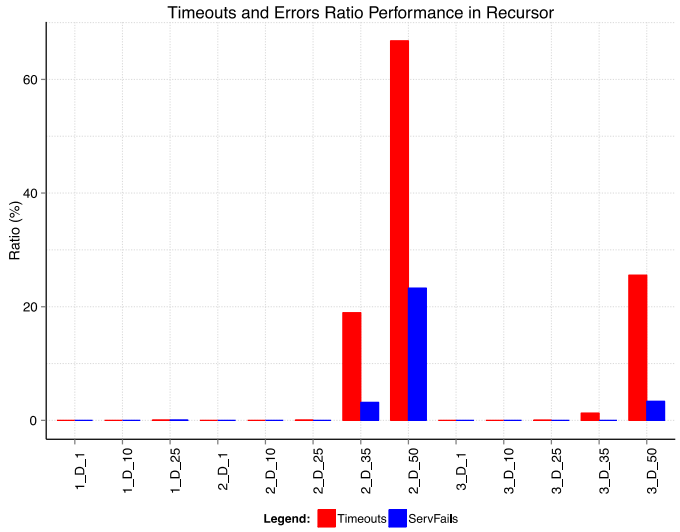


Figure 40 Ratio of timeouts and Servfails in percentage.

A.2 Performance benchmarking MaaS

In (D3.3, 2014) we extrapolated the expected service load based on large scale Zabbix deployment at CloudSigma, which uses Zabbix to monitor the load and stability of its production-grade cloud infrastructure across multiple locations. We outlined that each Zabbix compute and storage node template at CloudSigma includes an average of 100 items, together with approximately 25-30 triggers. What this means in practice is that 100 different metrics are being monitored, with 25-30 potential triggers defined to take action when certain conditions are breached on these metrics. The networking components within the CloudSigma cloud average at 300 items and 40 triggers per networking element, be it a switch or a router. These typically report to a separate Zabbix server to the compute and storage nodes. A summary of these parameters can be seen in the table and graph below:

Table # - Title

	WDC	SJC	ZRH
Hosts	22	39	53
Networking Nodes	7	36	16
Total infrastructure	29	75	69
Avg. total items monitored	4300	14700	10100
Avg. total triggers monitored	898	2394	2134

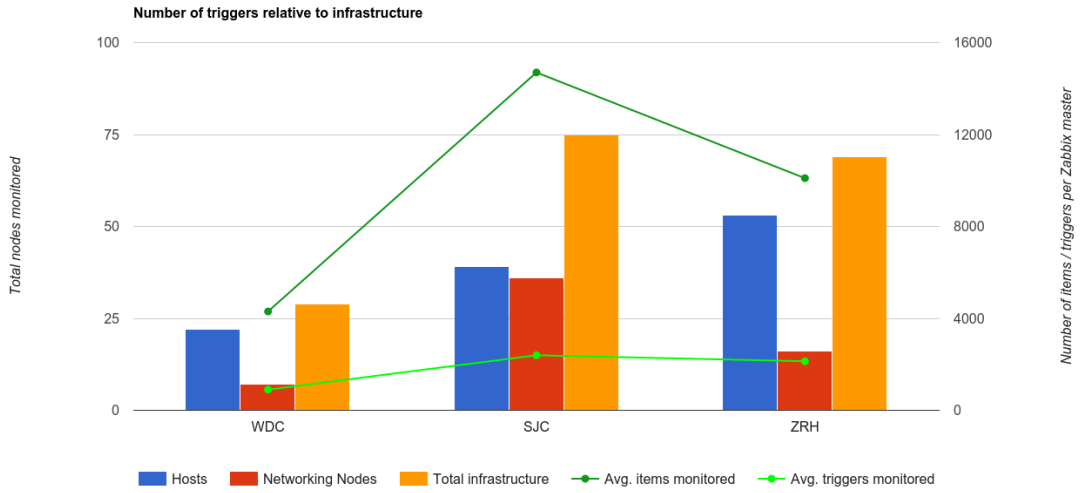


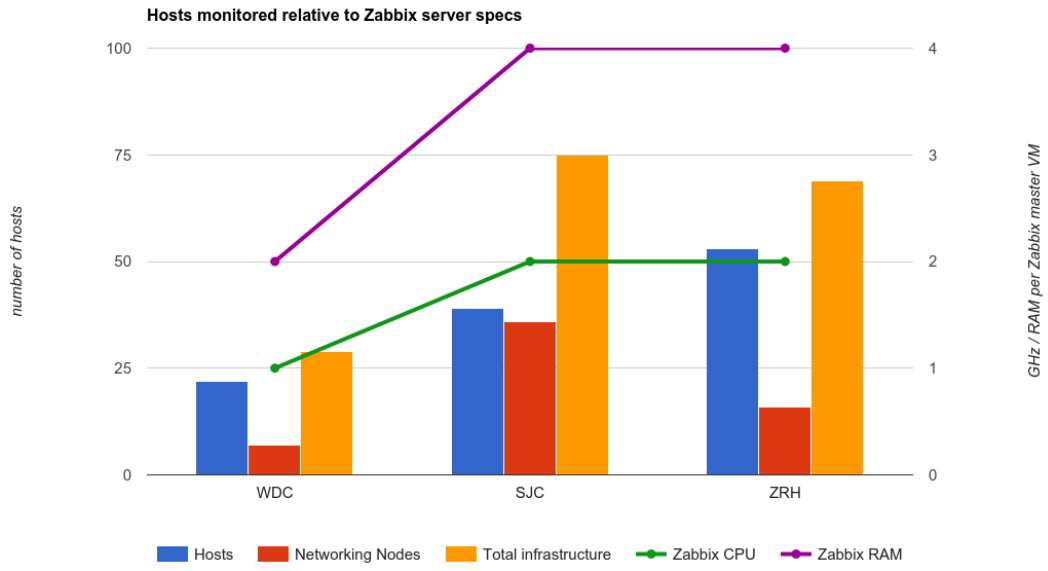
Figure # - Number of triggers relative to infrastructure

CloudSigma's Zabbix host is itself deployed within a VM outside of its cloud infrastructure, as part of the CloudSigma cloud stack. Each cloud location has such a Zabbix server. As infrastructure monitoring and rapid response is considered a critical part of the daily operation of the CloudSigma cloud, the Zabbix servers responsible for aggregating all host and networking infrastructure information are provisioned in relation to the size of the cloud in each location. More specifically, the relationship is as follows;

Table # - Title

	WDC	SJC	ZRH
Hosts	22	39	53
Networking Nodes	7	36	16
Zabbix CPU	1	2	2
Zabbix RAM	2	4	4

OBJ



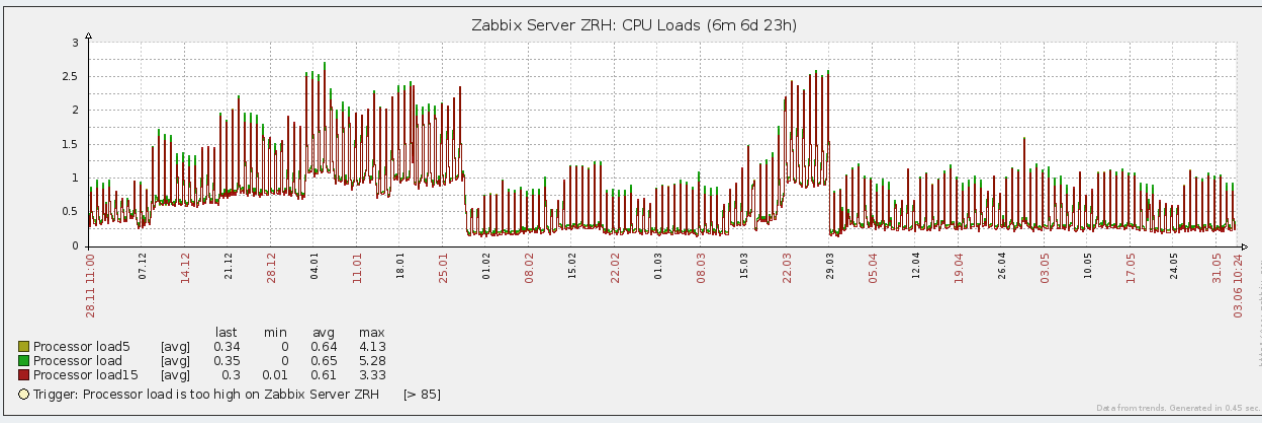
OBJ

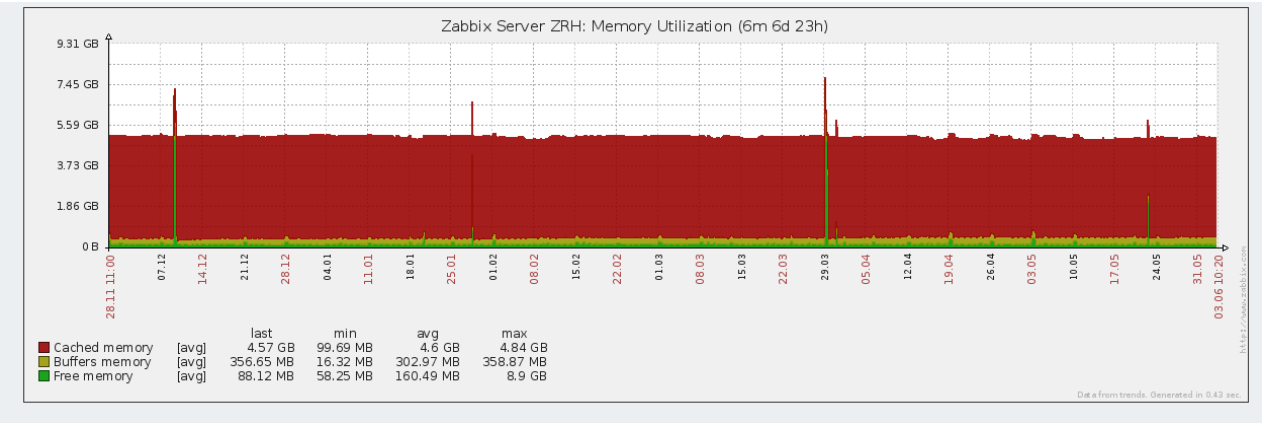
Figure # - Hosts monitored relative to Zabbix server specs

This graph illustrates the number of hosts monitored per location and its relativity to the Zabbix host specs. Please note that this graph does not represent the entire CloudSigma infrastructure, but rather a snapshot of the currently deployed public-facing production hosts. Additional hosts exist for testing and staging purposes, as well as idle hosts. These can be further allocated to the public-facing pool of hosts should cloud demand increase and one core tool to inform the Operations team as to the requirement to do so is namely Zabbix.

The graphs below are extracted from the CloudSigma Zabbix VM master host, which monitors the largest CloudSigma cloud location. As mentioned, these Zabbix hosts are deployed on VMs of varying specs, relative to the cloud location being monitored. Each Zabbix host is set up to also monitor their own load. As can be seen, in the long-term, the server load is minimal and varies between below 0.5% and 2.5% load.

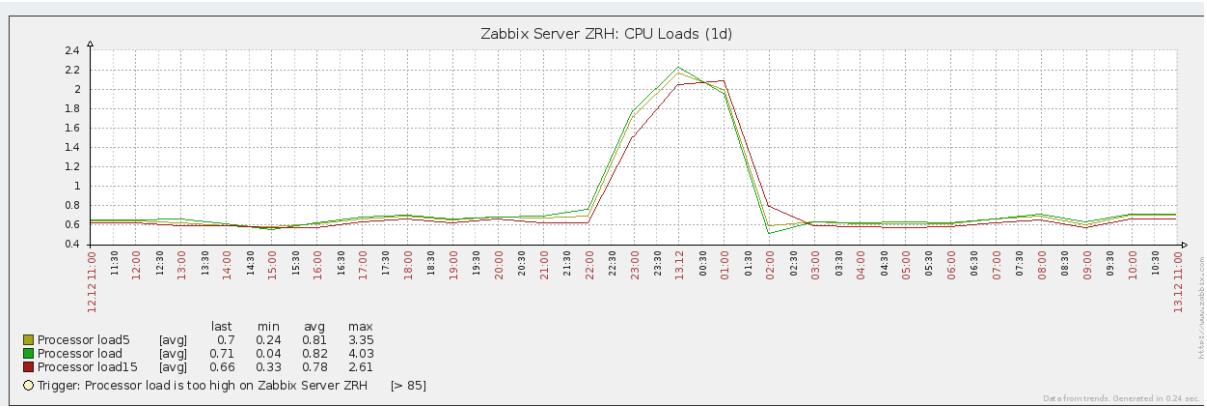
The memory requirements of Zabbix can also be observed to be under 512MB, which includes memory buffers. The remainder is consumed by Linux caching.

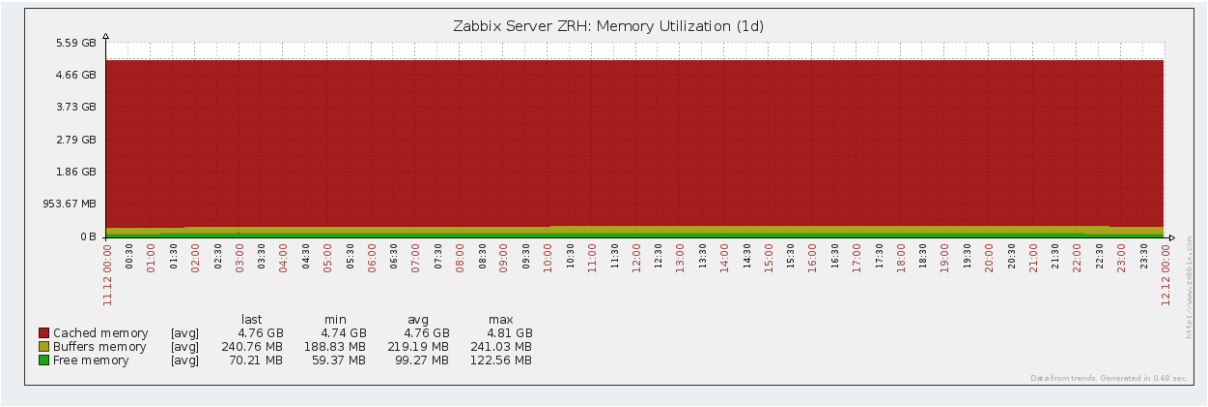




A.2.1 VM load during no-event day

If we drill down to a typical calm day where no triggers were tripped, we can observe that the average system load is typically between 0.6 and 0.8 percent, with the exception being around midnight, when crontab instantiates a series of system events. At this point, the system load rises to approximately 2 percent. On this typical non-event day, 4 triggers were breached, with the typical number being under 10. RAM usage can also be observed as being consistent.

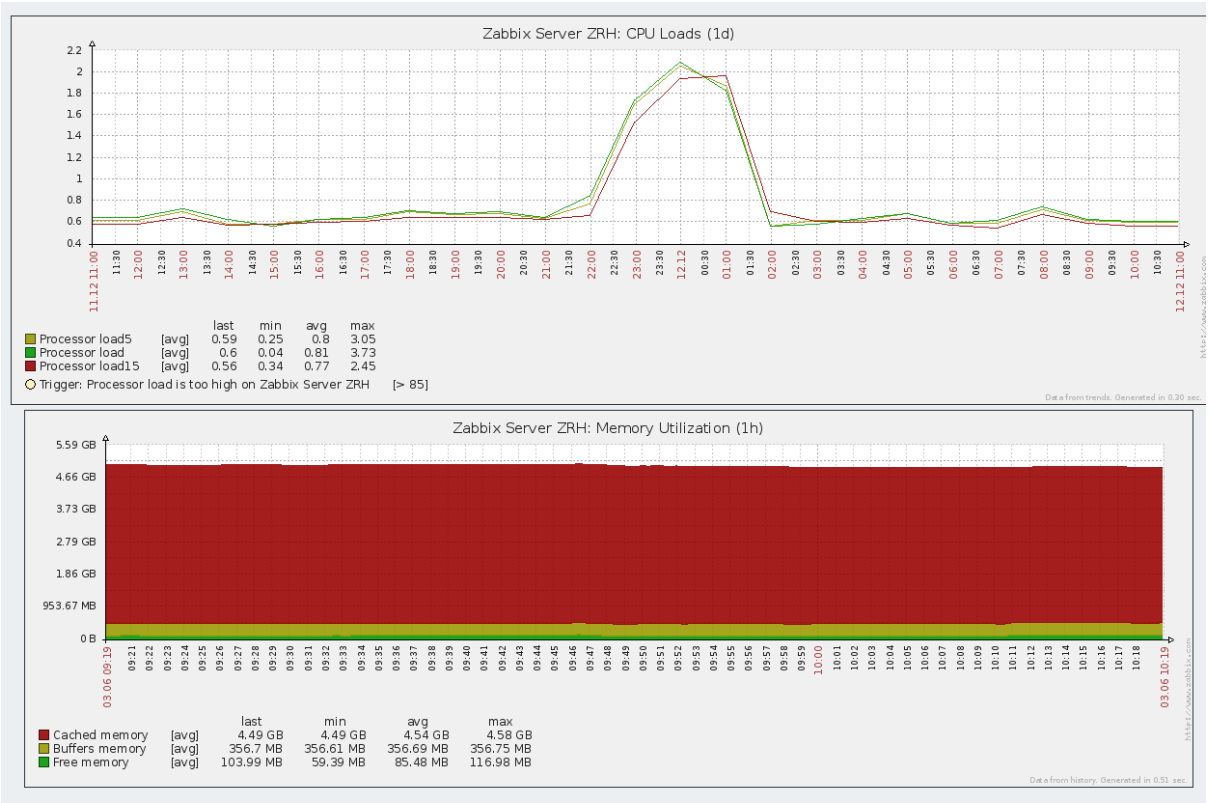




A.2.2 VM load during trigger-breach day

If we compare this to a day during which host issues did occur, we can observe a very similar load graph, indicating that the Zabbix master server was not placed under any additional strain. This can be explained by the fact that Zabbix' load is consistent for the number of events it monitors, with the fact that triggers having been breached resulting in the queuing of outbound trigger notifications. On this day when a compute or storage node has experienced issues, 118 triggers have been breached and 118 notifications queued and sequentially forwarded. RAM load again stays consistent.

OBJ



These findings go to show that Zabbix has very modest system requirements in order to monitor thousands, if not tens of thousands of items, at the same time evaluating thousands of

triggers. When triggers are fired, system load does not increase, at least at these levels of load. This goes a long way to establish that one MaaS instance can comfortably handle a very large scale service monitoring scenario.

In the above scenarios, Zabbix was deployed on infrastructure far outweighing the service's requirements. Typical load seldom breached 1%, with system events raising this to just over 2%. A Zabbix / MaaS instance could be deployed on a far lower-specced VM or container, with the aim of evaluating and establishing the minimum resource requirements of MaaS (Zabbix) in order to meet the SLA expectations of the service.

A.3 VRRM implementation details

The flowchart of VRRM Server is presented in Figure 41. Before appearance of Graphical User Interface (GUI), there are multiple initialisations to be done. At first, the server makes a connection to the SQL Server[9]. The database is used to speed up the procedure by not re-calculating intermediate steps or results. Considering the VRRM procedure, in specific, calculating the PDF of Radio Resource Units (RRUs), Radio Access Technologies (RATs), and the network are the most time consuming tasks during the evaluations and simulations. By saving the intermediate results during the calculation of probability functions, it is possible to decrease significantly the processing time. In order to improve the performance of the SQL-Server, the PDFs are saved in binary files and a pointer to relative file is placed in the SQL table. Using this technique, the size of SQL database file decreases and the running time is reduced to couple of minutes.

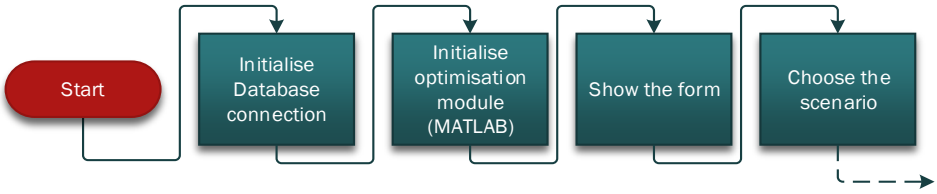


Figure 41 VRRM module flow chart.

After having database connected, the optimisation module is initialised. The function for optimisation is developed in MATLAB[10] and converted to a managed Dynamic Library Link (DLL). Having all the pre-requisite modules loaded, the control panel appears. By choosing "OAI Integration" scenario and clicking the "Load scenario" button, the VRRM Server follows the flowchart illustrated in Figure 42. After loading the real-time charts, the TCPLink module starts in listening mode. Upon reception of a new message, it is checked for the "Scenario Descritpion" message, MSG1 (Table 11). Then, the message is parsed and the scenario details are extracted. In the next step, the RAT module that contains the properties of the radio resources is created and initialised. After initialising the Common Radio Resource Management (CRRM) and VRRM modules, the VNOs based on the received scenario are formed. Finally, the services are added to the VNOs. In the next stage, the VRRM model optimisation is solved, the results are outputted into the excel file, and the new policy flag is set.

[9] http://www.microsoft.com/en-us/server-cloud/products/sql-server/default.aspx

[10] http://www.mathworks.com

VRRM responses to the policy update requests of OAI, MSG2 (Table 11) by the new set of policies when the new policy flag is set through MSG4 (Table 11). In other cases, the server just responds with MSG3 (Table 11) to inform the OAI that there is no update available.

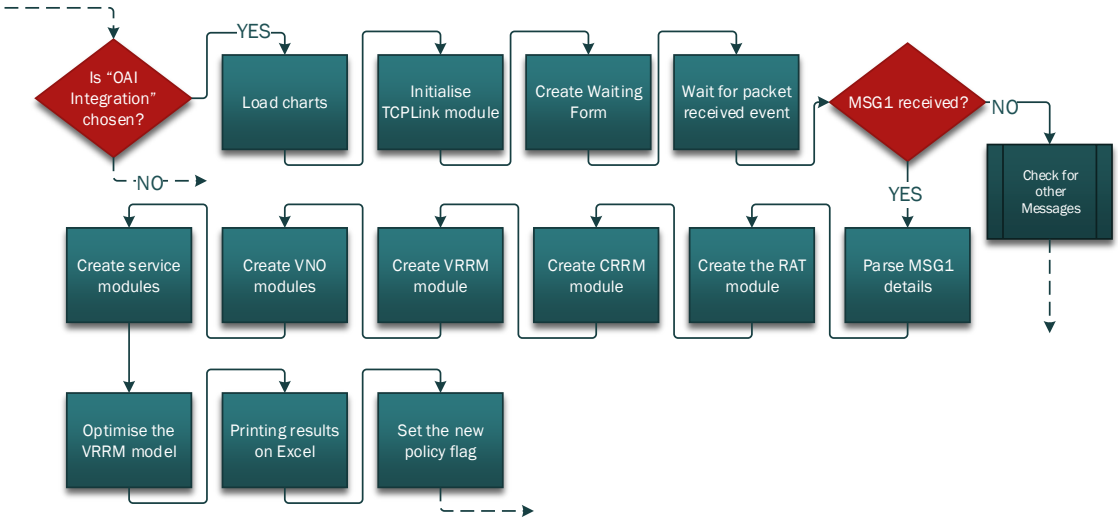


Figure 42 VRRM module flowchart for OAI integration.

VRRM responses to the policy update requests of OAI, MSG2 (Table 11) by the new set of policies when the new policy flag is set through MSG4 (Table 11). In other cases, the server just responds with MSG3 (Table 11) to inform the OAI that there is no update available.

A.3.1 Changes to OAI to support multiple VNOs/Groups

In addition to development of VRRM server, the OAI also needs some changes to support multiple VNOs or groups operating on the same infrastructure. One fundamental requirement is the gather statics per VNO/group instead of whole UEs.

OAI store UEs key performance indicators, e.g., total data rates offered to the UEs in downlinks. Based on the codes that gathers these statistics, new set of codes has been added to gather the same statistics but for each group. The group-based statics that has been added to OAI are:

- Group MAC PDUs (Protocol Data Units) per sub-frame (`dlsch_group_pdus_tx`),
- Group transferred size per sub-frame (`dlsch_group_bytes_tx`),
- Group download data rate per sub-frame (`dlsch_group_bitrate`),
- Group MCS (Modulation and Coding Scheme) per sub-frame (`dlsch_group_mcs`),
- Group active UEs per sub-frame (`dlsch_group_active_ue`).

These statistics are used in scheduler for enforcing the VRRM's policies and management of radio resources. However, the reports for the VRRM server contain statistics over an observation window. This observation window may be as long as the decision window discussed in VRRM modelling. In order to calculate this long-term statistic, the statistics obtained in each sub-frame are collected. Theis task is fulfilled by using the already implemented concept of listing in OAI. The following set of list variable was added:

- List for group bitrate (`dlsch_listg_bitrate`),

- List for group Modulation and Coding Scheme (MCS) (`dlsch_listg_mcs`),
- List for group active UEs (`dlsch_listg_active_ue`).

After initialising these variables in MAC layer, they are ready to be used. Figure 43 shows the flowchart of collecting, calculating, and reporting the long-term statistics. As it is shown, when the VNOs statistics become available in each sub-frame, they are pushed into relative lists. At the end of each sub-frame, a counter is placed to count the number of sub-frames. When the value of counter become equal to the length of observation window, the average of statistics pushed into list are calculated and reported to VRRM server using the "Real-Time Report" message, MSG5 (Table 11). Finally lists are cleared and the sub-frame counter is set to zero.

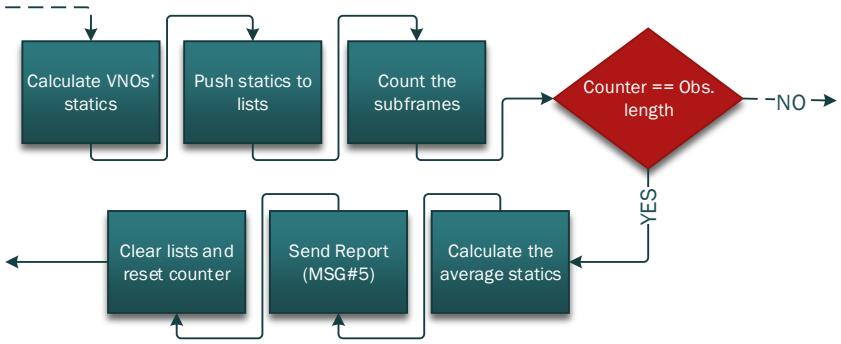


Figure 43 flow chart of calculating and reporting long term statics.

A.3.2 Changing the algorithm of MAC scheduler in order to support the groups policies

The downlink scheduler flowchart originally implemented in OAI is shown in Figure 44. Storing the UEs' DLSCH buffers is the first step in the scheduling process. Then, the number of RBs required to transmit the packets in each UEs buffer is calculated. Sorting the UEs by their priorities, they are pre-allocated with the available RBs. Based on the output of the pre-allocation, the actual allocation is done in two rounds.

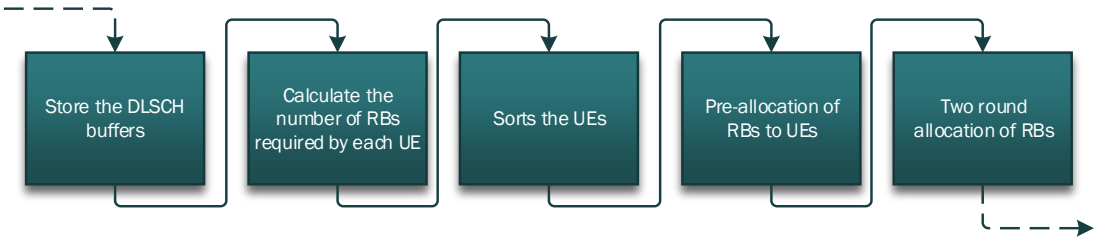


Figure 44 Flowchart of downlink scheduler.

In the following, the changes introduced to these function to support multiple VNOs or groups are described.

The first change that has been done in the scheduler procedure is the sorting of UEs. Originally, the UEs were sorted based on the following aspects:

- *HARQ round*: the users that are in their second round of HARQ are given higher priority,

- *Bytes in the buffer*: the users that have more bytes to receive are moved to the beginning of the user list.
- *Maximum time of SDU creation*: the users with more tolerance of delay pushed to the back of list opening up space for other users.
- *UE ID*: in the case all the other criteria are the same, UE ID is the last sorting criterion.

This function was changed in order to put the users of VNOs with higher priority on the top of the list before checking the other criteria. Based on the policies received from VRRM, each VNO is received a portion of available RBs. The goal is to start the user list with the users of VNO with biggest share of RBs. The flowchart of sorting UEs is depicted in **Error! Reference source not found.**. The changes s presented in blue while the rest with green.

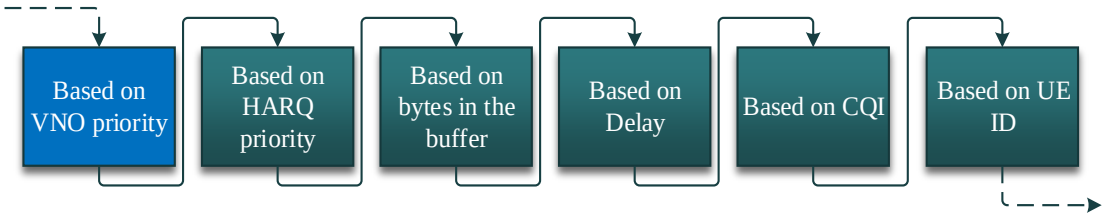


Figure 45 UE sorting algorithms

In the next step, the pre-allocation sequence is modified to support the possibility of having multiple VNOs (or groups) with different priorities.

The pre-allocation initially was implemented as it is shown in the green and red blocks in Figure 46. When there are active UEs, the required RBs for fully serve each of them is calculated. On the case there are not enough resources to serve all the active UEs with at least minimum RB units (i.e., 2 RBs in the implementation of OAI), the minimum RB units are pre-allocated to all of them. Otherwise, the average RB per UE obtained in the last step is pre-allocated. It is obvious that no pre-allocation is done when there are no active UEs. Based on the sorted UE list, each terminal that has to receive the control information is granted with the required RBs. UE with data traffic receive the average RBs. In situation, when the required RB for a user is smaller than the average RBs, UE is just pre-assigned only the required RBs.

The pre-allocation procedure is changed by calling a new function that maps the assigned data rate of each VNO to its share of available resources. For each frame the sharing of available resources is updated based on the policies received from VRRM, minimum data rate for each VNO, and key performance indicators, namely, the average MCS per VNO. In a first step the number of RBs per VNO is calculated by dividing minimum data rate assigned by average MCS achieved. The adaptation to the maximum number of RBs available for the eNodeB is considered in a second step, in order to avoid overbooking.

Then, the same pre-allocation procedure is done but this time only per VNO. In better words, the average RBs per UEs is calculated for UEs of each VNO based on their share of the available resources obtained in first step. The flowchart of the pre-allocation procedure with the new algorithms illustrated by the light blue and orange blocks in Figure 46 shows how the changes are implemented. It is worth noting that the primary goal was to implement the support for virtualisation of radio resources with minimum changes in OAI.

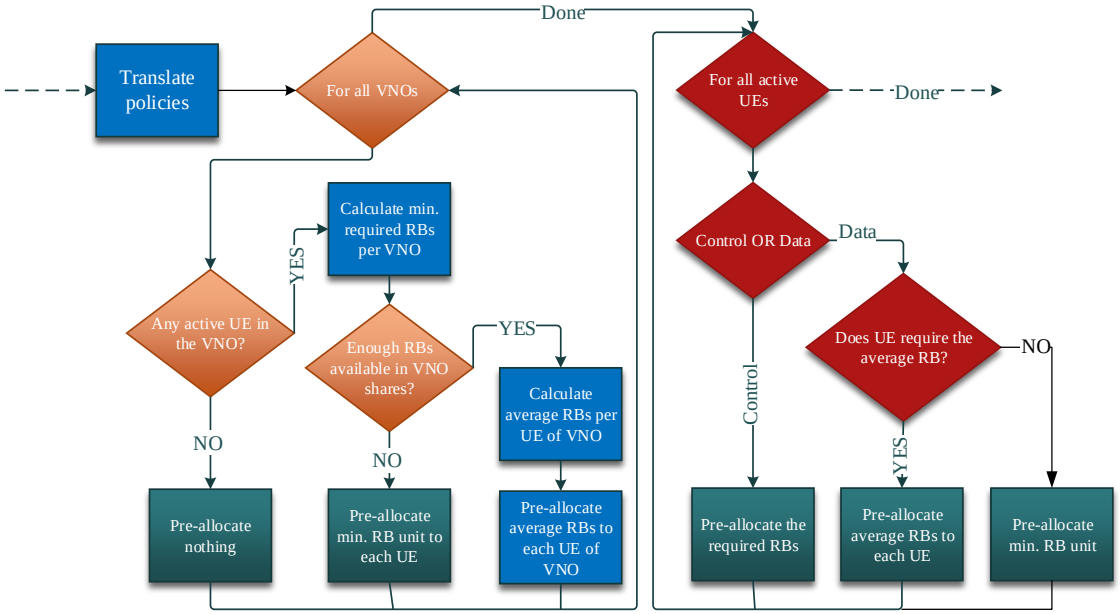


Figure 46 Changed pre-allocation procedure.

A.3.3 Changing the codes to add the groups information into the XML file

In order to facilitate the introduction of information related to VNOs/groups a new set of parameters identified by <PROTOCOL><MAC> was added into the XML file already existing for OAI inputs.

An example of the XML code for the configuration of OAI-VRRM integration is presented in Figure 48.

In the <VRRM> section the parameters for the TCP/IP communication are defined, namely, the IP address of the machine in which VRRM is running (VRRM_IP) and the TCP port (VRRM_PORT). The VRRM_LINK parameter allows running OAI without the connection to VRRM, but maintaining the virtualisation of radio resources for serving different VNOs.

In the following lines the number of groups/VNOs (NUM_GROUPS) and the quantity of frames that are considering for each report sent to VRRM (OBS_WIN_LENGTH) are set.

Finally, in the <GROUPS> section the static information for each group is stated. Three type of groups are considered: BE, BE with minimum guaranteed data rate and Guaranteed. The differentiation between the types of the groups is made based on the values in Gigabit/s assigned to GROUP_MIN_GB and GROUP_MAX_GB, Table 10. The SERVICE_WEIGHT and VIOLATION_WEIGHT are parameters used for the optimization performed by VRRM module, allowing to distinguish among VNOs of the same type.

Table 10 – Group type settings.

Group Type	GROUP_MIN_GB [GB]	GROUP_MAX_GB [GB]
Best Effort (BE)	0	0
BE with min Guaranteed	>0	0
Guaranteed bitrate (BG)	>0	>0

A.3.4 Adding support for bidirectional communication

- The communication protocol implemented between OAI and VRRM module has the same structure for all the messages as it is shown in Figure 47.
- Header and trailer: are placed to determine the beginning and ending of the packets,
- Message number: defines the structure of the data field
- Packet size: 2-byte long field indicates the size of the packet.

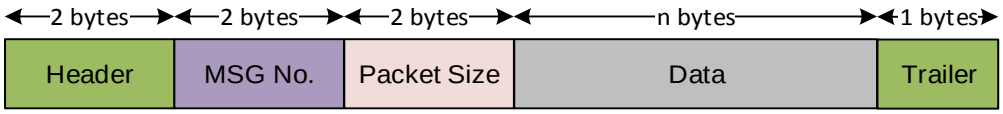


Figure 47 Packet structure.

In the following the implemented messages, Table 11, are briefly described.

Table 11 Messages Summary.

Message	Message Designation	Communication	Comments
MSG1	Scenario Description	OAI → VRRM	Contains the scenarios parameters.
MSG2	Update Request	OAI → VRRM	Request the policies update
MSG3	No Updated Available	VRRM → OAI	Policies are not ready to be sent
MSG4	Policies Update	VRRM → OAI	Contains the updated policies
MSG5	Real-time Report	OAI → VRRM	Contains the operational report

```
<PROTOCOL>
     <MAC>
      <!--Parameters definition for OAI-VRRM communication-->
      <VRRM>
           <VRRM_LINK>1</VRRM_LINK>
           <VRRM_IP>10.154.3.22</VRRM_IP>
           <VRRM_PORT>5000</VRRM_PORT>
      </VRRM>
      <!--Number of groups (VNOs) requesting service-->
      <NUM_GROUPS>2</NUM_GROUPS>
      <!--Statistics report interval (in number of frames) from OAI to
VRRM-->
      <OBS_WIN_LENGTH>20</OBS_WIN_LENGTH>
      <!—Groups static information-->
      <GROUPS>
           <SOURCE_ID>0</SOURCE_ID>
           <GROUP_MIN_GB>0</GROUP_MIN_GB>
           <GROUP_MAX_GB>0</GROUP_MAX_GB>
           <SERVING_WEIGHT>0.04</SERVING_WEIGHT>
           <VIOLATION_WEIGHT>0.36</VIOLATION_WEIGHT>
       </GROUPS>
       <GROUPS>
           <SOURCE_ID>1</SOURCE_ID>
           <GROUP_MIN_GB>3</GROUP_MIN_GB>
           <GROUP_MAX_GB>5</GROUP_MAX_GB>
           <SERVING_WEIGHT>0.06</SERVING_WEIGHT>
           <VIOLATION_WEIGHT>0.54</VIOLATION_WEIGHT>
           </GROUPS>
      </MAC>
</PROTOCOL>
```

Figure 48 Example of XML code for configuration of OAI-VRRM communication and groups configuration

10 References

Alex Henneveld. (2013) CAMP, TOSCA, and HEAT, http://de.slideshare.net/alexheneveld/2013-04specscamptoscaheatbrooklyn

AWS. (2013) AWS CloudFormation, http://aws.amazon.com/en/cloudformation/

Bruno Sousa, Claudio Marques, David Palma, João Gonçalves, Paulo Simões, Thomas Bohnert, and Luis Cordeiro. (2014) Towards a High Performance DNSaaS Deployment, Presented at the MONAMI 2014, Wurburg, Germany, September 22, pp., 77–90

D2.2. (2013) Overall Architecture Definition, MobileCloud Networking Project

D2.5. (2015) Overall Architecture Definition, Mobile-Cloud Networking

D3.1. (2013) Infrastructure Management Foundations – Specifications & Design for MobileCloud framework, MobileCloud Networking Project

D3.2. (2014) Infrastructure Management Foundations – Components First Release, MobileCloud Networking Project

D3.3. (2014) Infrastructure Management Foundations – Components Final Release, MobileCloud Networking Project

D4.3. (2014) Algorithms and Mechanisms for the Mobile Network Cloud, Mobile-Cloud Networking

DoW. (2012) Description of Work, Mobile-Cloud Networking

IEEE JSAC. (2015) IEEEE JSAC Special Issue on Measuring and Troubleshooting the Internet: Algorithms, Tools and Applications, IEEE, http://www.comsoc.org/files/Publications/Journals/jsac/cfp/measuring%20and%20troubleshooting%20the%20internet%20cfp.pdf

Morabito, R., Kjallman, J., and Komu, M. (2015) Hypervisors vs. Lightweight Virtualization: A Performance Comparison, pp., 386–393

Nominum. (2015) DNSperf, http://nominum.com/measurement-tools/

Nyren, R., Edmonds, A., Alexander Papaspyrou, and Metsch, T. (2011) Open Cloud Computing Interface - Core, Open Grid Forum, http://ogf.org/documents/GFD.183.pdf

SAIL. (2013) SAIL, The SAIL project, http://www.sail-project.eu/

Schmidt, R. de O., Hendriks, L., Pras, A., and Pol, R. van der. (2014) OpenFlow-based Link Dimensioning, New Orleans, LA, USA, November, http://eprints.eemcs.utwente.nl/25451/

SURFnet. (2015) GigaPort3: a state-of-the-art network for the Netherlands, SURF, https://www.surf.nl/en/knowledge-and-innovation/innovationprojects/2009/gigaport3.html

Trove. (2014) OpenStack Trove, https://wiki.openstack.org/wiki/Trove