SUPERFLUIDITY

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

Research and Innovation Action GA 671566

Deliverable I.3.1:

Initial system architecture and programming interface specification

Deliverable Type:	Report
Dissemination Level:	CO
Contractual Date of Delivery to the EU:	31/03/2016
Actual Date of Delivery to the EU:	31/03/2016 (V1) 12/07/2016 (V2)
Workpackage Contributing to the Deliverable:	WP3

Editor(s):	Erez Biton (NOKIA-IL), Stefano Salsano (CNIT)
Author(s):	Carlos Parada (ALB), Francisco Fontes (ALB), Isabel Borges(ALB), Omer Gurewitz (BGU), Philip Eardley (BT), Giuseppe Bianchi (CNIT), Nicola Blefari Melazzi (CNIT), Luca Chiaraviglio (CNIT), Stefano Salsano (CNIT), George Tsolis (CITRIX), Christos Tselios (CITRIX), Michael J. McGrath (Intel), Filipe Manco (NEC), Felipe Huici (NEC), Lionel Natarianni (NOKIA-FR), Bessem Sayadi (NOKIA-FR), Erez Biton: (NOKIA-IL), Danny Raz (NOKIA-IL), John Thomson (OnApp), Julian Chesterfield (OnApp), Michail Flouris (OnApp), Anastassios Nanos (OnApp),

	Haim Daniel (Red Hat), Livnat Peer (Red Hat), Pedro de la Cruz Ramos (Telcaria Ideas S.L.), Juan Manuel Sánchez Mateo (Telcaria Ideas S.L.), Raúl Álvarez Pinilla (Telcaria Ideas S.L.), Pedro A. Aranda (Telefónica, I+D), Costin Raiciu (UPB)
Internal Reviewer(s)	

Abstract:	This internal report summarizes the initial system architecture and the programming interface specifications. This initial results shows the consideration that lead to the architecture design, as well as initial discussion on the programming languages.

Keyword List:	Architecture, reusable functional blocs, programmable network functions

INDEX

List of Figures

List of Tables

Glossary

(TO BE FILLED OUT IN THE FINAL VERSION OF THE DELIVERABLE)

SUPERFLUIDITY DICTIONNARY	
TERM	DEFINITION

Table 1: SUPERFLUIDITY Dictionary.

1 Introduction

1.1 Deliverable Rationale

This internal report is the result of the architecture discussions as part of Task 3.1 and 3.2. It follows the work on the technical and business requirements in I2.2 and it is performed in parallel to the work on the use cases and requirements of Deliverable D2.1 [23] and on Functional Analysis and Decomposition of Deliverable D2.2 [25]. With focus on softwarization of network functions, a major target of this deliverable is to promote a "divide et impera" approach to the design and operations of network services, with the notion of Reusable Function Blocks (RFB) as elementary primitives. RFBs can be integrated and composed together in a programmatic manner to devise more comprehensive (composed) network functions in a rapid and cost-effective way. RFBs can be "orchestrated" over their *execution platforms* in order to achieve an optimal efficiency in the utilization of the needed resources (e.g. computing, networking, storage). We specifically highlight the general (abstract) concept of RFB and we discuss the underlying issues and the necessary architectural support, raising the need (and role) for a RFB Execution Environment. We then proceed by casting this general approach to a variety of more concrete scenarios, ranging from traditional NFV frameworks where composability and reusability of relatively comprehensive network functions (or blocks) is already widely established, to more advanced data plane programmability approaches where it appears convenient to perform a more fine-grained (elementary) decomposition and where the role of a tailored execution environment appears less obvious.

1.2 Quality Review

Review Team member responsible of the deliverable: _____

VERSION CONTROL TABLE			
VERSION N.	PURPOSE/CHANGES	AUTHOR	DATE
V1	First submission	Erez Biton	31-03-2016
V2	Revised version		11-07-2016

1.3 Executive summary

1.3.1 Deliverable description

In this internal report, the Superfluidity architecture is presented. The architecture is based on the following main concept: (i) standardization convergence, (ii) reusable functional blocks and (iii) programmable and portable interfaces.

1.3.2 Summary of results

(TO BE FILLED OUT IN THE FINAL VERSION OF THE DELIVERABLE)

2 Overall architectural framework for Superfluidity

Current cellular networks are facing significant challenges. In 2015 mobile data traffic grew by nearly 70% with video streaming services contributing over 50% of that growth. Smartphones have been a key driver in this growth in combination with the rollout of LTE networks. By the end of 2015, there were nearly a billion LTE subscribers globally from nearly 500 commercial networks. In the US mobile network traffic for AT&T network grew by a 100,000% from 2007 to 2014 and their video traffic doubled from 2014 to 2015. The challenge is therefore in the increasing mobility and increasing bandwidth requirements of end-users

The explosion in the Internet of Things (IoT) is rapidly driving up mobile network traffic. It is expected that there will be over 30 billion connected devices by 2020 divided into massive IoT and critical IoT. To address the explosion in mobile traffic and connected devices, service providers need to perform better capacity management, which can react dynamically to the network conditions and device's traffic. There is also a need to handle the heterogeneity mobile networks in terms of access technologies, and deployment [1].

5G is expected to support multiple industries/user business cases, multiple access technologies, multiple services and multiple tenants. 5G systems will rely on advancements in the radio connectivity (massive MIMO, beam-forming, etc), in the deployment options (small cell, multi-RAT, etc), in the network capability like mobile Edge Computing and in the latest advancement in the SDN and NFV framework.

One can classify the targeted 5G use cases into three broad categories [2], namely: (i) Enhanced Mobile Broadband (eMBB), (ii) Massive IoT, and (iii) Critical IoT. eMBB covers a variety of new challenging use cases including: Immersive Internet, Augmented Reality, Virtual Reality, Ultra High Definition (UHD) Content Delivery, to cite a few. Critical IoT covers use cases like eHealth, Vehicular communication, etc. It is important to highlight here the stringent requirements that 5G should fulfill for these use cases in terms of ultra low latency (~1ms) and reliability (`5 nines'). The Superfluidity project has analysed a set of use cases to help in the process of gathering requirements for the definition of the architecture. The outcome of this activity has been reported in Deliverable D2.1 [23].

Superfluidity aims to build 5G network addressing the use cases requirements and unifying the network approaches supporting:

1. Agility: Quick and flexible service creation and deployment.
2. Mobility: Flexible placement and rapid migration of services.
3. Management of Heterogeneity: Not just abstracting heterogeneity, but taking advantage of it.

4. Functional composition/decomposition: Services are constructed by programmatic composition of *Reusable Functional Blocks* (RFB).
5. Efficient RFB implementation, e.g. by means of hardware (HW) and software (SW) acceleration when needed.
6. Security by design: RFBs are verified to be secure before deploying them.

The Superfluidity architecture will rely on a proper combination of a set of emerging technologies in different areas. Beyond the utilization of multiple technologies, the big innovation is on the way those technologies can be integrated into a single conceptual environment, taking advantage of synergies among them. Superfluidity leverages the works conducted individually in many forum, namely ETSI NFV, ETSI MEC, ONF, 3GPP and TMForum and advances them. Figure 1 depicts a high-level view of the overall architectural framework. In this section, we assume the reader is familiar with the different involved technologies. In Section 6 we provide the basic tutorial information and a short report on the state of the art.

In the top of Figure 1 the different involved components (C-RAN, MEC, vCore and DC) are shown, while in the bottom the different types of DCs involved (Cell-site, Local, Regional and Central).

Below components indication, the big OSS blue box represents the traditional operational support systems, which deals with all the components in order to create services for end users.

On the bottom, an Extended-NFVI represents an evolution of the ETSI NFVI concept, considering the additional heterogeneity of its nature (including hardware, hypervisors, and other execution environments), and the federation of DCs at different geographies and different types. This extended-NFVI is common to all components, easing resource management and allowing an agile (“superfluid”) orchestration of services.

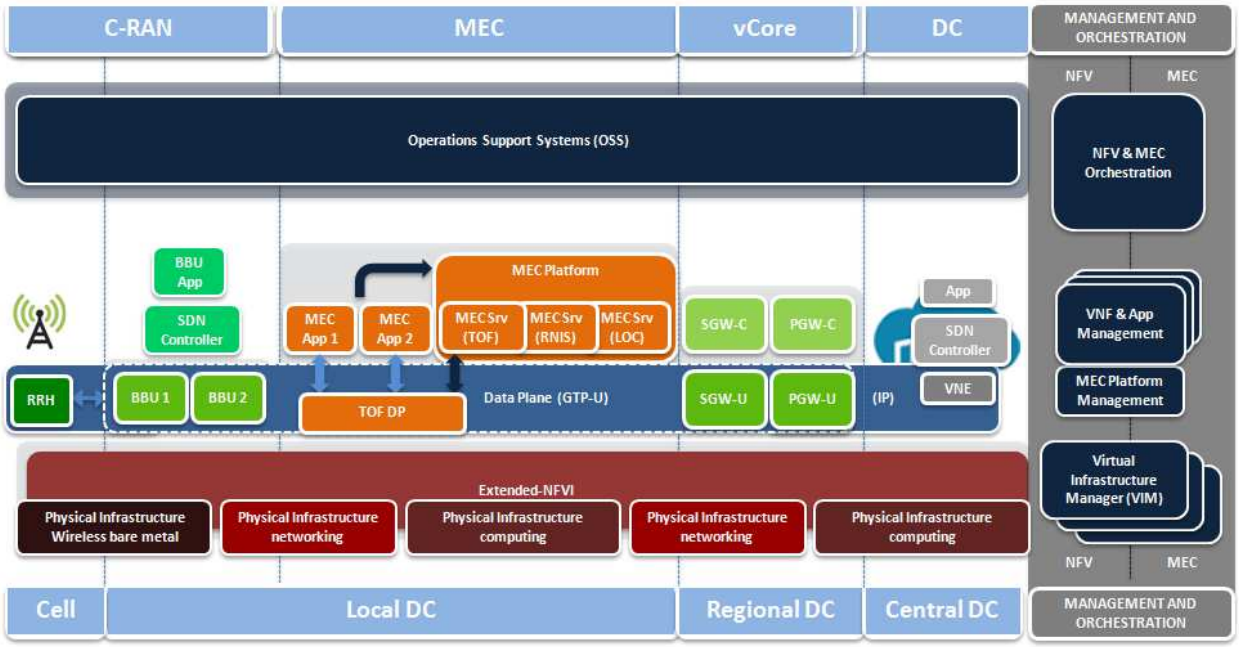


Figure 1: Overall Architectural framework for Superfluidity

Starting from the left, the C-RAN component is split into 2 blocks, corresponding to the RRH and the BBU parts, the first residing in the cell-site and the second in a local DC (not far away from the RRH). This shows the usual C-RAN view, where multiple approaches can be considered for the functional split among them. The BBU boxes can be implemented as NFVs and also follow the SDN concept, as depicted in the figure, by using an SDN Controller and building a BBU App on top, which control the whole BBU network element operations.

The MEC component appears next and can be deployed in the same location (DC) as C-RAN. As C-RAN concentrates in the local DC a large number RANs, this is the right place to run MEC applications (*Note: Other options are also possible, e.g. closer to the core; however, it would result in higher latencies*). The MEC environment uses the same infrastructure as the C-RAN or other VNFs, making the solution very convenient and efficient.

The vCore component is the next component to appear, comprising the central nodes of a mobile network. This Core runs in the common Extended-NFVI, usually located in regional DCs, easing the agility and fluidity when the nodes need to be migrated and/or scaled. The Figure depicts the expected evolution of mobile core networks towards the SDN model, where control plane and data plane components are completely separated, and the former ones fully control the latter ones on data processing tasks. (*Note: The components shown are the ones resulting from splitting the 4G/LTE Core elements*).

The DC component corresponds to the traditional datacenter segment, where a large number of services are deployed. Those are located at central points and deal with a large amount of resources. Beyond traditional services, central DCs also implement NFV and SDN technologies,

in order to control the network components inside the DC. For this reason, in this segment we show a generic VNE (Virtual Network Element), the SDN controller and a generic App element.

Finally, the right most vertical box is the common management and orchestration layer, which is the brain of the whole system. This vertical is an NFV-like set of functions, which allows for an integrated view of everything: network, services, DCs and services, managing the entire system in a superfluid way, taking advantage of a common extended-NFVI and achieving an end user centric view of the ecosystem.

In the bottom of this vertical box, a VIM-like set of components takes care of the resource management on the different DCs, interacting among them to build a federated environment. In the middle, a set of VNF and App Managers handle the lifecycle details of particular VNFs and Applications. Also, a Platform Manager is used to manage service APIs and applications access to those APIs. Finally, orchestrators are able to orchestrate VNFs to create complex services and take the best decisions for Apps to be deployed and serve customers.

3 Superfluidity Architecture

The vision of the Superfluidity project is to move from the current architectural approaches based on monolithic network components/entities and their interfaces, to an approach where network functions can be programmatically composed using “building blocks” and the deployment of these “building blocks” over the underlying infrastructure is highly dynamical, allowing a continuous real-time optimization.

The idea of decomposing relatively complex network functions in more elementary parts is certainly not novel by itself, and has been explored in many scenarios, from higher level Network Function Virtualization frameworks to low level programmable switches/routers based on blocks (e.g. Click routers) to even lower level approaches entailing the identification of very elementary primitives (e.g. OpenFlow actions). However, while discussing such different approaches, we realized that most of the work in this area has remained quite focused on the specific domain scenario, and only a limited attempt to find out similarities and general principles among the different approaches has been to date carried out. At the cost of appearing perhaps abstract (but we will cast such general ideas into more concrete scenarios in the next sections), goal of this section is to try to understand whether there are some common architecture elements and principles which are shared by composition-based approaches.

3.1 Overview

The decomposition of high-level monolithic functions into reusable components is based on the concept of *Reusable Functional Blocks* (RFBs). A RFB is a logical entity that performs a set of functionality and has a set of logical input/output ports. In general, a Reusable Functional Block can hold state information, so that the processing of information coming in its logical ports can depend on such state information. RFBs can be composed in graphs to provide services or to form other RFBs (therefore a Reusable Functional Block can be composed of other RFBs). The RFB decomposition concept is applied to different heterogeneous environments.

In order to achieve the vision of superfluid composition, allocation and deployment of “building blocks” (RFBs) the Superfluidity project focuses on the following pillars:

Decomposition of monolithic functions into Reusable Functional Blocks in different heterogeneous environments - Superfluidity targets to decompose: network services into basic functional abstractions; node-level functions into more granular packet processing primitives; heterogeneous hardware and system primitives into block abstractions.

Measurements based RFB operations - This pillar consists in the definition and realization of tools for real time monitoring of RFB performances and of the status of the environments

(e.g. resource utilization). The information provided by such tools can be used to drive the RFB operations (e.g. allocation of resources to RFBs).

Semantic specification of Block Abstractions – A crucial aspect in composition-based approaches is to provide the "programmer" with a clear and unambiguous understanding of what each block is expected to do, and how it processes input data and produces an output.

High Performance Block Abstractions Implementation - This pillar consists of deriving block abstractions for each of the underlying hardware components, and to optimize their performance.

3.2 RFB heterogeneity: different abstraction levels and granularity

An important peculiarity of the Superfluidity approach is that the (de)composition is applied to different heterogeneous environments, from network-wide composition of (virtual) network functions to node-level composition of packet processing blocks, to composition of signal processing blocks. Actually, it is possible to operate recursively at the different levels, for example network-wide services can be decomposed in node-level RFBs, that can be decomposed in smaller packet processing RFBs.

An important property is the level of formal detail that characterizes the description of an RFB. In some environments, the RFBs can be seen as *small*, elementary building blocks to which it is possible to associate a description of their behaviour using some formal language. In this case, starting from the formal description of the RFBs functionality, it is conceptually possible to compose RFBs and derive the properties of the composed RFB. We can define models that operate on the RFBs with high granularity, the languages that describe the RFB are able to capture the functional "input/output" behaviour of the RFB with all details.

In other environments, the RFBs are high-level components, which are expected to respect some functional specification (e.g. a large number of protocol specification documents). In this case, it is not possible to have a complete description of the RFB functionality using some formal language. The models operate on RFBs with coarse granularity (e.g. on high-level service building blocks). The RFB descriptions in this case are not able to completely capture the "input/output" behaviour of the RFBs, they can be interested to describe non-functional properties like requirements on the underlying infrastructure supporting the RFB execution.

Given this heterogeneity and considering the target framework for Superfluidity (see Figure 1) it is not possible to build from scratch a *clean slate* 5G system that homogeneously follows a new Superfluidity architecture design. Rather, the Superfluidity architecture provides a unified vision, combining, adapting and extending existing models and languages.

3.3 RFB Operations and RFB Composition views

In the definition of the Superfluidity architecture and models, the RFB concept is used in two complementary ways. On one hand, it can be used to model the *allocation* of service components to an execution platform, with the proper mapping of resources. We refer to this approach as *RFB Operations*. On the other hand, it can be used to explicitly model the *composition* of RFBs to realize a service or a component. We refer to this approach as *RFB Composition*. The two approaches are not mutually exclusive and in some scenarios they can be combined.

Considering these two approaches and the heterogeneity of the target environments that we have discussed in the previous section, the Superfluidity architectural modelling based on RFBs can support the definition of different types of frameworks:

Operation/Orchestration frameworks: These frameworks concerns the deployment of high-level RFBs onto the supporting execution environments in an optimal way, taking into account the real time status of resources (e.g. computing, networking, storage). As an example, we can refer to the current ETSI NFV approach, with the allocation of VNFs (Virtual Network Function) to their execution Environment (NFVI).

Composition frameworks: These frameworks concerns the automatic composition of RFBs to build other functional components or service. They can be a sort of "compilers" or "builders" that produce executables for composed RFBs, or they can be concerned with formal correctness check of RFBs chains. The features of these frameworks depend on the type of RFBs and execution environment they are focused on. As an example we can refer to a provisioning framework that matches node-level function abstractions (e.g. a firewall) with the available hardware/block abstractions (e.g. packet processing primitives) in order to *automatically* derive high performance and meet end-user SLA requirements, i.e., without forcing developers to understand low-level system details.

3.4 Superfluidity Architecture

In the previous section, we have introduced the concept of Reusable Functional Blocks and proposed their use in Operation and Composition frameworks. Let us further proceed in the Superfluidity architecture modelling that will support such frameworks.

RFBs needs to be characterized and described, and we need platform-agnostic node-level and network-level "programs" describing how the RFBs interact, communicate, and connect to each other so as to give rise to specific (macroscopic, and formerly monolithic) node components, network functions and services. A language that supports the description and the interaction of RFBs is referred to as *RFB Description and Composition Language (RDCL)*. At the state of the art, we believe it is not possible to have a single language that is capable to describe the different

types of RFBs in the heterogeneous environments and properly capture the needed features to support the RFB Operations and Composition views. Therefore, the Superfluidity architecture will deal with a set of RDCLs, targeted to the different environments. The Superfluidity project will extend the existing RDCLs where needed to support the dynamicity of the allocation/deployment on the underlying infrastructure and to support the coordination of operations at the different heterogeneous environments.

The heterogeneous computational and execution environments, supporting the execution (and deployment) of the RDCL scripts and the relevant coordination of the signal/radio/packet/flow/network processing primitives are referred to as RFB Execution Environments (REE). A set of REEs considered in the context of Superfluidity is described in Section 4.6.

The Superfluidity architecture shown in Figure 2 describes the relation among the RFB, RDCL and REE concepts. The figure highlights that the model (with due technical differences) recursively apply at different levels.

For example we can consider the network-level shown in Figure 2 as the typical NFV level of the ESTI model, where RFBs are meant to be relatively large functions (e.g. a load balancer, a firewall, etc., represented by the RFB #1, #2, #3) and the REE is a network-wide orchestrator complemented by an SDN controller. In turns, a specific high-level network function can be implemented over a programmable network node using in turns a decomposition into more elementary sub-blocks (e.g. packet processing primitives of a programmable router or OpenFlow-like elementary forwarding and processing actions, described as RFB #a, #b, #c).

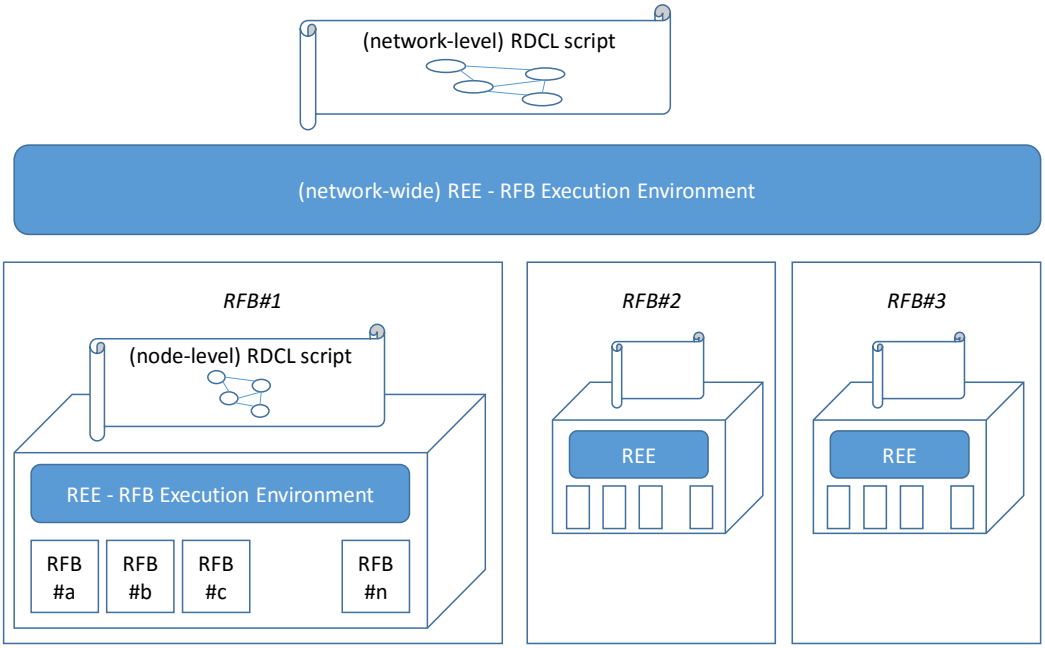


Figure 2: Superfluidity Architecture

Under this vision, operators can formally describe a desired network service as a composition of platform-agnostic, abstract elementary processing blocks (RFBs). Vendors will be in charge of providing efficient software implementations of such elementary blocks, possibly casting them to underlying hardware accelerated facilities. The (cloud-based) infrastructure will provide the brokerage services to match the design with the actual underlying hardware, to control and orchestrate the execution of the RFBs comprising the designed service, and to permit dynamic and elastic provisioning of supplementary dedicated computational, storage, and bandwidth resources to the processing components loaded with peak traffic. As long as different vendor platforms expose the same set of RFBs, and as long as they are clearly specified in terms of semantics (i.e. as long as RFBs implemented by different vendors produce the same output for a same input), how they are specifically implemented (e.g., in HW or in SW) is not nearly a concern for the network programmer. A standardization of such set of RFBs (again, think to the OpenFlow analogy in terms of standardized actions) is all needed to guarantee that an application which suitably composes RFBs running on a given platform (e.g. a SW switch) can be migrated on a different platform (e.g. a bare metal HW switch) which supports the same set of RFBs.

3.4.1 Reusable Function Blocks (RFBs)

The decomposition of high-level monolithic functions into reusable components is based on the concept of Reusable Functional Block (RFB). A Reusable Functional Block can be composed of other RFBs. RFBs can be composed in graphs to provide services or to form other RFBs. In general, a Reusable Functional Block can hold state information, so that the processing of information coming in its logical ports depends on such state information.

A Reusable Functional Block is characterized by a set of logical ports of different types: e.g. data plane ports, control plane ports, management plane ports. A logical port represents a flow of information going in and/or out of the RFB. RFBs can be characterized by their resource needs (e.g. storage, processing) or load limitations (e.g. maximum number of packets/s or number of different flows). RFBs characterization can be augmented with formal description of their behaviour, as needed to formally derive the behaviour of a graph composing a set of RFBs.

A fundamental (and often only implicitly assumed) property of the RFB is the environment on which the RFB can run (or can be hosted), referred to as RFB Execution Environment (REE).

To make an example of RFB Execution Environment let us consider the NFV Infrastructure (NFVI) under standardization by ETSI. In this context, the RFB can be mapped into the concepts of VNF (Virtual Network Function) and of VNFC (Virtual Network Function Component). For example, a RFB that can be mapped 1:1 against a single Virtualization Container (using the ETSI terminology) corresponds to an ETSI VNFC (VNF Component). In this case, its execution environment is a Virtualization Container hypervisor.

Another example of RFB Execution Environment is a modular programmable router architecture like Click [15]. In the Click architecture, a set of components called Click elements can be composed by means of a directed graph (called configuration). In this context, a RFB corresponds to a Click element and will have as execution environment a Click router.

Figure 3 highlights the difference between the current model considered in the ETSI standardization and the proposed approach. In the ETSI model, there is a fixed hierarchy between VNF Groups (defined in [3] but not considered in [4]). VNFs and VNF Components. VNF Components cannot be further decomposed. In the proposed model, all elements are RFB and the decomposition can be iterated an arbitrary number of times.

To make an example of hierarchy, a Click router instance can be a VNF Component in a VNF. Its hosting_environment is a VC hypervisor. The Click instance is described by means of a directed graph of elements (called configuration). Each element is a RFB that has the Click router as execution environment. As shown in Figure 4, in the current modelling approach it is not possible to further decompose the VNFC.

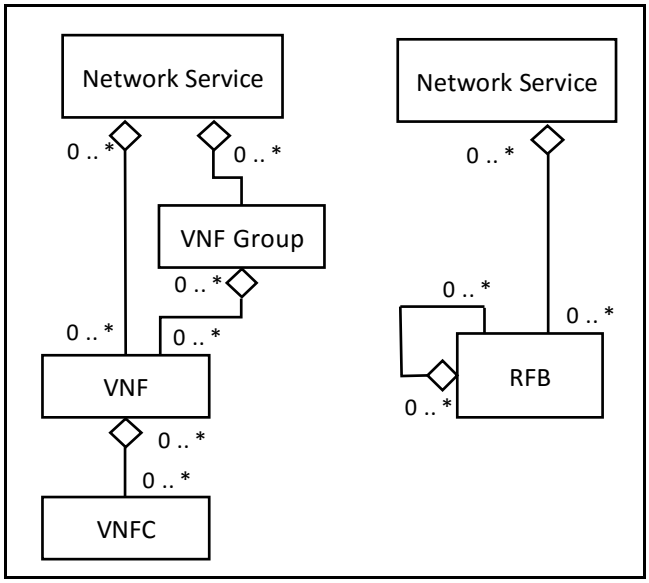


Figure 3: Class diagram for the current approach (left side) and the proposed one (right side).

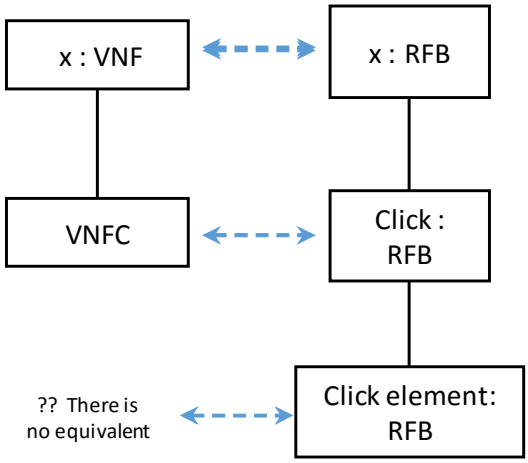


Figure 4: Example of a hierarchy of RFBs.

RFBs can be composed to provide high-level functions and services but also to form other RFBs, as a RFB can be composed of other RFBs. The decomposition of RFBs into other RFBs can be iterated an arbitrary number of times, unlike VNF Components that cannot be further decomposed.

3.4.2 RFB Description and Composition Language (RDCL)

The concept of RFB lends itself to different usage scenarios. An RFB *Composition* framework may consider that logical RFBs can have more than one possible execution environment (i.e. they can be realized using different technologies and frameworks). Elementary radio, packet, and flow processing primitives and events can be formally specified and described independently of the specific underlying hardware as Reusable Functional Blocks, and then implemented and automatically selected/instantiated so as to match the underlying hardware facilities while taking advantage of the relevant accelerators (e.g., GPUs or FPGAs) if/when available. On the other hand, we can have RFBs bound to a specific RFB Execution Environment, and an *Orchestration* framework that only cares about coordinating their execution and allocating them to resources in an optimal way within the specific Execution Environment.

The different scenarios and the heterogeneous RFB Execution Environments imply that a set of different tools are needed to describe the RFBs and their composition. We refer to these tools as RFB Description and Composition Languages (RCDLs).

Considering the NFV Infrastructure under standardization by ETSI, some of the features that VNF, Virtual Machine and Compute Host Descriptors should contain for the appropriate

deployment of VNFs on Virtual Machines over Compute Hosts in a telco datacenter are described in [6].

An approach towards a recursive definition of RFBs has been recently proposed in [22].

An important usage scenario for the RFBs composition is the formal correctness check of a configuration before its deployment. This requires RFB Description and Composition Languages that are able to express the functional properties of a RFB with a formal (semantic) language.

Coming to RFBs that describe packet processing operations A key to portability is the ability to describe a potentially complex and stateful network function as the combination of RFBs, so as to obtain a platform-agnostic formal description of how such RFBs should be invoked, e.g. in which order, with which input data, and how such composition may possibly depend (and change!) based on higher level "states". Indeed, by writing an high level network function as a composition of blocks already pre-implemented in the underlying platform(s), migration of such function from a platform to another (e.g., from a software node to a more capable hardware accelerated node, or from a node placed in the core to an edge node) would be as simple as moving the configuration itself, and would not require transfer and run-time booting of Virtual Machines.

While the idea is simple, its actual specification may be all but trivial. Different languages may be more appropriate to different network contexts (e.g. for node-level programmable switch platforms opposed to network-wide NFV frameworks). And, to the best of our knowledge, as of now there is not a clear cut candidate standing out. Indeed, the typical approach, used in block-based node platforms (such as the Click router, or even in Software Defined Radio platforms) of formally modelling a composition of blocks as a Direct Acyclic Graph of such blocks may be insufficient, as it does not permit the programmer to introduce the notion of "application level states" and hence dynamically change (adapt) the composition to a mutated stateful context. Languages in the IFTTT family (If This Then That), recently introduced in completely different contexts (such as web services or Internet of Things scenarios) may find application also in the networking context given their resemblance with the matching functionality frequently used in forwarding tasks, although, again, stateful applications may require extensions. It is worth to mention that the SUPERFLUIDITY project is specifically addressing novel languages formalized in terms of eXtended Finite State Machines, which appear suited to generalize the widely understood OpenFlow's match/action abstraction, and at the same time appear to retain high-speed implementation feasibility.

3.4.3 RFB Execution environment (REE)

Having a set of blocks (i.e., the RFBs), and having a language which describes how they are composed does not conclude our job. In the proposed architectural framework, we also need an entity or a framework in charge of "executing" such a composition. For an extreme analogy with

ordinary computing systems, in addition to the processor instruction set and the programming (machine) language, we also need a Central Processing Unit which is in charge to execute the workflow and directives formally specified by the programmer using the language. In most generality, we refer to such framework as “RFB Execution Environment” (REE). The role of the REE is to concretely instantiate a desired RDCL script instance, control its execution, maintain and update the application-level states, trigger reconfigurations, and so on. In traditional network-wide NFV scenarios, this role is generally played by an orchestrator, whereas in VNFs implemented in software within a container, or a Virtual Machine, this role is played by the VM itself. Conversely, the case of high-speed bare metal switches where configuration and per-flow state maintenance must occur at wire speed is a bit more problematic: in most architectures this role may not be clearly identified. Delegating this role to the overlying software/control stack may not be the most performance effective solution and it may lead to performance bottlenecks and slow-path operation.

The REE can be seen as a generalization and enhancement of the NFVI (NFV Infrastructure). In the current model the infrastructure (NFVI) provides resources and an external orchestrator coordinates them (but it only instantiates VMs and connects them according to the graph that describes the network services). In the envisaged model, the enhanced infrastructure REE provides the means to execute arbitrarily complex RDCL scripts operating at different levels.

4 Programming interfaces

Considering the architecture proposed in Figure 2, we identify the logical entities that are involved in the realization of services. We consider that the architecture can recursively be applied to a number of levels, therefore we define a generic approach that can be applied at all levels. At each level, we identify a REE Manager and a REE User. The REE User requests the realization/deployment/execution of a service / service component described using a RDCL script to the REE manager. The REE Manager is in charge of deploying/executing the RCDL script using the resources in its REE. Within a REE, the REE Manager interact with REE Resources Entities that are needed to support the realization of the RCDL script.

Hence, we can identify two main APIs. The first one is the API used by the REE User that wants to deploy a service or a component into an RFB Execution Environment. We refer to it as the User-Manager API (UM API). The second one is the API used by the REE Manager to interact with the resources in its REE. We refer to it as the Manager-Resource API (MR API). Figure 5 illustrates these concepts, showing how the APIs can be mapped to different levels. Note that we refer to the APIs considering information exchange in both directions for each APIs.

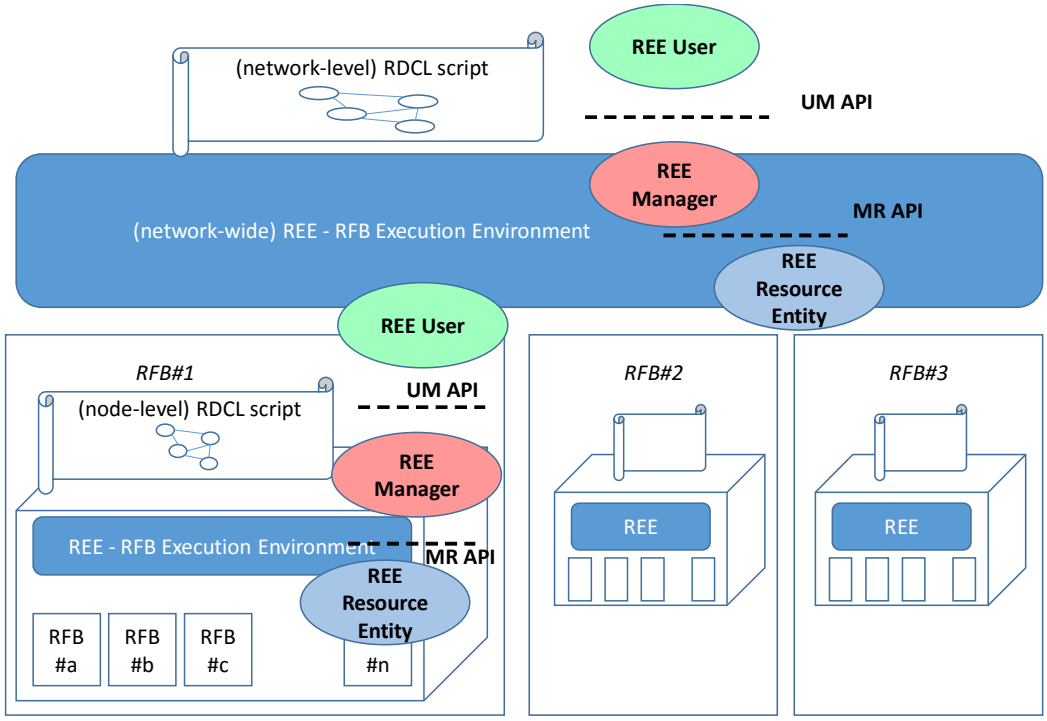


Figure 5: Superfluidity Architecture APIs

The APIs for the different levels may use different modelling approaches, languages and tools, considering that Superfluidity will not build them from scratch but will rely on existing work.

Wherever possible, the project will try to relate the APIs at different level and proceed towards a harmonized vision.

4.1 API for collecting performance information from the NFV Infrastructure

The NFVI is responsible for providing the execution environment for RFB's and will have a significant influence on both the performance and behaviour of RFB's depending on its level of heterogeneity. Heterogeneity within the NFVI appears in at least three major forms. Firstly, the virtualised infrastructure environment will mostly likely be comprised of different generations and types of X86 CPU's. Additionally the compute nodes of the NFVI will have different PCIe devices such as accelerators, network cards with different features sets, chipsets, storage types etc. Secondly, the state of the compute nodes will vary significantly depending on what type workloads are running and the number of workload instances. Finally, the physical location of resources, which maybe important for some RFB's where proximity to users is required to deliver the best possible user experience is required. Therefore, detailed information on the types (physical and virtual), their state and location is required in order to provide the necessary information required at orchestration level to make either informed initial placement, migration or scaling decisions. According to our architectural model, this corresponds to a Manager-Resource API.

Information about the physical and virtualised resources maybe available from the Virtualised Infrastructure Manager (VIM) used to manage the infrastructure resource. For example, in an OpenStack cloud environment infrastructure information is available from the various service databases such as NOVA and Neutron, via standardised REST API's. However the granularity of information available or may not be sufficient depending on the explicit needs of the Orchestrator and algorithms used to make RFB placement decisions. The information available in other VIM environments such as Mesos or Kubernetes may also vary significant. In an SDN environment, network information from the SDN controller also needs to be considered in order complete network topology picture. For example OpenDaylight exposes various types of topology information via REST based API's.

The picture increases in complexity in multi-VM environment where commonality (type, format, quantity) in resource information across in the various VIMs will not exist. Therefore, a centralised infrastructure repository that collates the infrastructure information from the various VIMs into a common repository and exposes the information via a standard set of REST API's maybe advantageous as it simplifies the required interactions between the orchestrator and NFVI. This type of approach can also be used to store the physical location of resources via some of geo tagging or other form location specific identifier that is consumable at an orchestration level. Collectors designed for each VIM environment can ensure that the information is retrieved from each environment is consistent and concurrent. The information

in the repository maybe also augmented with additional network information from SDN controllers. Secondly, this approach is also provide greater flexibility in how topology of workloads and their allocated resources (physical and virtual) are represented.

Telemetry plays a key role in monitoring the state and utilisation of the infrastructure resources within the NFVI. Monitoring of resources maybe available as a service within different VIM environments. For example in OpenStack the ceilometer service provides data on the utilisation of the physical and virtual resources comprising deployed clouds via standard API. However, the granularity of the available data is limited to metric such as CPU and memory utilisation, which may or may not be sufficient depending on the type of workload to be monitored. A standalone metrics platform, which can support fine-grained monitoring of resource across different virtualised (VMs, containers, unikernels) and non-virtualised environments maybe necessary. Collection of network metrics from SDN controllers may also be necessary such as what flows are connected to which ports on either physical or virtual switches. The data collected can be written to various common databases such as MySQL, MariaDB, OpenTSDB and exposed at an orchestration level either through predefined queries or preferentially through an API such as REST in order to provide abstraction from the underlying telemetry collection mechanisms, which either may be evolved or be changed over time. The metrics collected may also processed in some predefined manner and results stored in the database repository. For example, data could be interrogated for SLA or KPI violations and then used to trigger corrective actions by the Orchestrator when necessary.

4.2 API toward the 3rd parties that want to deploy applications over Superfluidity

[This is a User-Manager API.]

[See activity in Task T5.3]

4.3 Optimal function allocation API (for packet processing services in a programmable nodes)

[This is a User-Manager API]

[Given a description of a network wide end-to-end configuration, how to map that in physical allocations at different levels – See activity in Task T5.1]

4.4 Configuration/deployment API for OpenStack VIM

[This is a Manager-Resource API]

[It is offered by a VIM like Openstack, it allows an orchestrator to ask for resources, for example the HEAT language can be used – See activity in task T6.1]

4.5 Semantic correctness check API for packet processing services

[This is a User-Manager API]

[A network provider that wants to check the correctness of the implementation of some services based on the RFB services - See activity in Task T6.3]

4.6 NEMO: a recursive language for VNF combination

Currently, there is a lot of activity going on to use NFV in the network. From the point of view of the orchestration, Virtual Network Functions are blocks that are deployed in the infrastructure as independent units. They provide for one layer of components (VNF components – VNFCs), i.e. a set of VNFCs accessible to a VNF provider that can be composed into VNFs. However, there is no simple way to use existing VNFs as components in VNFs with a higher degree of complexity. In addition, VNF Descriptors (VNFDs) used in different open source MANO frameworks are YAML-based files, which despite being human readable, are not easy to understand.

On the other hand, there has been recently an attempt to work on a modelling language for networks (NEMO). This language is human-readable and provides a NodeModel construct to describe nodes that supports recursiveness. In this draft, we propose an addition to NEMO to make it interact with VNFDs supported by a NFV MANO framework. This integration creates a new language for VNFDs that is recursive, allowing VNFs to be created based on the definitions of existing VNFs.

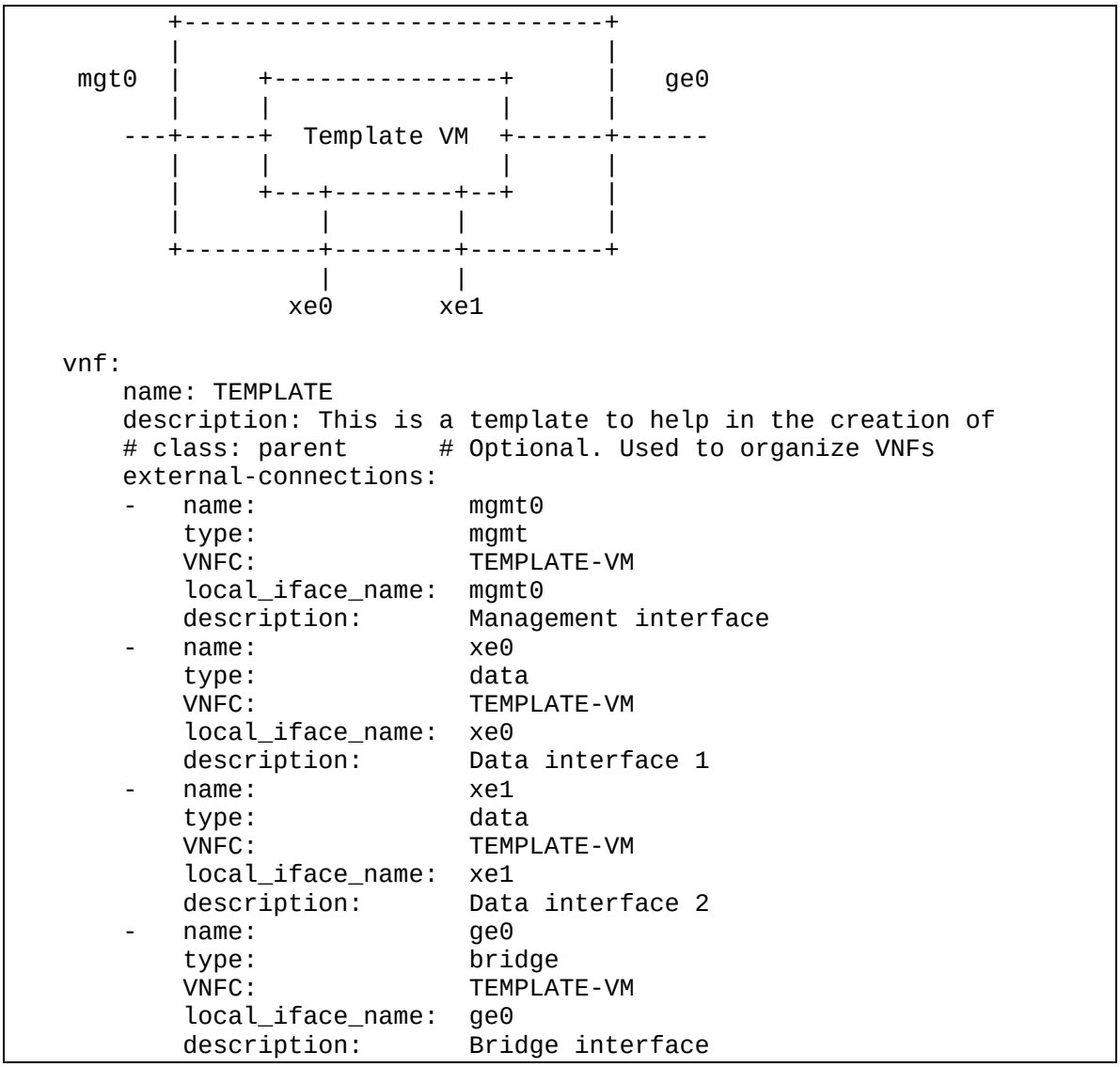
Hereafter, we use OpenMANO and OSM descriptor references as an example for the lowest level descriptors that are imported into NEMO. Conceptually, other descriptor formats like TOSCA can also be used at this level.

4.6.1 Virtual network function descriptors tutorial

Virtual network function descriptors (VNFDs) are used in the Management and orchestration (MANO) framework of the ETSI NFV to achieve the optimal deployment of virtual network functions (VNFs). The Virtual Infrastructure Manager (VIM) uses this information to place the functions optimally. VNFDs include information of the components of a specific VNF and their interconnection to implement the VNF, in the form of a forwarding graph. In addition to the

forwarding graph, the vnfd includes information regarding the interfaces of the VNF. These are then used to connect the VNF to either physical or logical interfaces once it is deployed.

There are different MANO frameworks available. For this draft, we will concentrate on the example of OpenMANO [1], which usesYAML [2]. Taking the example from the (public) OpenMANO github repository, we can easily identify the virtual interfaces of the sample VNFs in their descriptors:




```
vnf:
    name: TEMPLATE
    description: This is a template to help in the creation of
    # class: parent      # Optional. Used to organize VNFs
    external-connections:
    -   name:              mgmt0
        type:              mgmt
        VNFC:              TEMPLATE-VM
        local_iface_name:  mgmt0
        description:       Management interface
    -   name:              xe0
        type:              data
        VNFC:              TEMPLATE-VM
        local_iface_name:  xe0
        description:       Data interface 1
    -   name:              xe1
        type:              data
        VNFC:              TEMPLATE-VM
        local_iface_name:  xe1
        description:       Data interface 2
    -   name:              ge0
        type:              bridge
        VNFC:              TEMPLATE-VM
        local_iface_name:  ge0
        description:       Bridge interface
```

Figure 6: Sample VNF and descriptor (source: OpenMANO github)

4.6.2 NEMO tutorial

The Network Modeling (NEMO) language is described in[I-D.xia-sdnrg-nemo-language]. It provides a simple way ofmdescribing network scenarios. The language is based on a two-stage process. In the first stage, models for nodes, links and other entities are defined. In the second

stage, the defined models are instanciated. The NEMO language also allows for behavioural descriptions. A variant of the NEMO language is used in the OpenDaylight NEMO northbound API [3].

NEMO allows to define NodeModels, which are then instanciated in the infrastructure. NodeModels are recursive and can be built with basic node types or with previously defined NodeModels. An example for a script defining a NodeModel is shown below:

```
CREATE NodeModel dmz
  Property string: location-fw, string: location-n2,
    string: ipprefix, string: gatewayip, string: srcip,
    string: subnodes-n2;
  Node fw1
    Type fw
    Property location: location-fw,
      operating-mode: layer3;
```

Figure 7: Creating a NodeModel in NEMO

4.6.3 Proposed enhancements to NEMO

In order to integrate VNFDs into NEMO, we need to take into account two specifics of VNFDs, which cannot be expressed in the current language model. Firstly, we need a way to reference the file which holds the VNFD provided by the VNF developer. This will normally be a universal resource identifier (URI). Additionally, we need to make the NEMO model aware of the virtual network interfaces.

Referencing VNFDs in a NodeModel

As explained in the introduction, in order integrate VNFDs into the NEMO language in the easiest way we need to reference the VNFD as a Universal Resource Identifier (URI) as defined in RFC 3986 [RFC3986]. To this avail, we define a new element in the NodeModel to import the VNFD:

```
CREATE NodeModel NAME <node_model_name>
  IMPORT VNFD FROM <vnfd_uri>
```

Referencing the network interfaces of a VNF in a NodeModel

As shown in Figure 6, VNFDs include an exhaustive list of interfaces, including the interfaces to the management network. However, since these interfaces may not be significant for specific network scenarios and since interface names in the VNFD may not be adequate in NEMO, we propose to define a new element in the node model, namely the ConnectionPoint.

```
CREATE NodeModel NAME <node_model_name>
  DEFINE ConnectionPoint <cp_name> FROM VNFD:<iface_from_vnfd>
```

An example

Once these two elements are included in the NEMO language, it is possibly to recursively define NodeModel elements that use VNFDs in the lowest level of recursion. Firstly, we create NodeModels from VNFDs:

```
CREATE NodeModel NAME SampleVNF
    IMPORT VNFD from https://github.com/nfvlabs/openmano.git
/openmano/vnfs/examples/dataplaneVNF1.yaml
    DEFINE ConnectionPoint data_inside as VNFD:ge0
    DEFINE ConnectionPoint data_outside as VNFD:ge1
```

Figure 8: Import from a sample VNFD from the OpenMANO repository

Then we can reuse these NodeModels recursively to create complex NodeModels:

```
CREATE NodeModel NAME ComplexNode
    Node InputVNF TYPE SampleVNF
    Node OutputVNF TYPE ShaperVNF
    DEFINE ConnectionPoint input
    DEFINE ConnectionPoint output
    CONNECTION input_connection FROM input TO InputVNF:data_inside
        TYPE p2p
    CONNECTION output_connection FROM output TO ShaperVNF:wan
        TYPE p2p
    CONNECTION internal FROM InputVNF:data_outside TO ShaperVNF:lan
        TYPE p2p
```

Figure 9: Create a composed NodeModel

This NodeModel definition creates a composed model linking the SampleVNF created from the VNFD with a hypothetical ShaperVNF defined elsewhere. This definition can be represented graphically as follows:

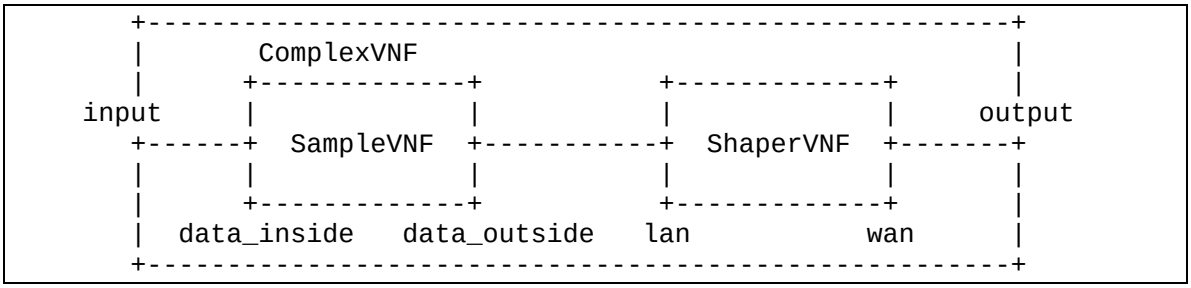


Figure 10: Representation of the composed element

In ETSI NFV, a network service is described by one or more VNFs that are connected through one or more network VNFFGs. This is no more than what is defined in the composed NodeModel shown in Figure 10. By using NEMO, we provide a simple way to define VNF forwarding graphs (VNF-FGs) in network service descriptors in a recursive way.

5 Instantiation of the RFB concept in different environments

5.1 Management and Orchestration in a “traditional” NFV Infrastructure

The cloud infrastructure is the basis of the emerging cloud technology. It allows the creation virtual entities, with compute, storage and networking resources, appearing as if they were physical. The use of hypervisors (e.g. KVM) is still the most common virtualization technology; however, container-based technologies (e.g. dockers) are getting momentum. ETSI NFV refers to this execution environment as NFV Infrastructure (NFVI).

Independently of the virtualization technology in use, this infrastructure needs to be managed with a high degree of agility, in order to allow rapid changes in the virtual entities, taking this way advantage of the virtualization paradigm. At the same time, as services are often very complex, an orchestration function must make the job of building coherent end2end services using virtual entities. Although significantly different, management and orchestration functions are referred by ETSI NFV together as the MANO block.

There are many ways for organizing the MANO macro block, along different administrative domain and locations. This is valid for the two main types of services approached in this project: network functions (e.g. eNB, EPC) and applications (e.g. Augmented Reality, Video Analytics). For a more elaborated description about this topic, please refer to section 4 of I6.1 [24].

5.1.1 NFVIs and Services

Today, different cloud infrastructures (NFVIs) are usually devoted to run specific services (see Figure 11-a). However, the utilization of a common NFVI for multiple services (see Figure 11-b) can provide a more efficient use of resources, taking advantage of synergies and reducing costs.

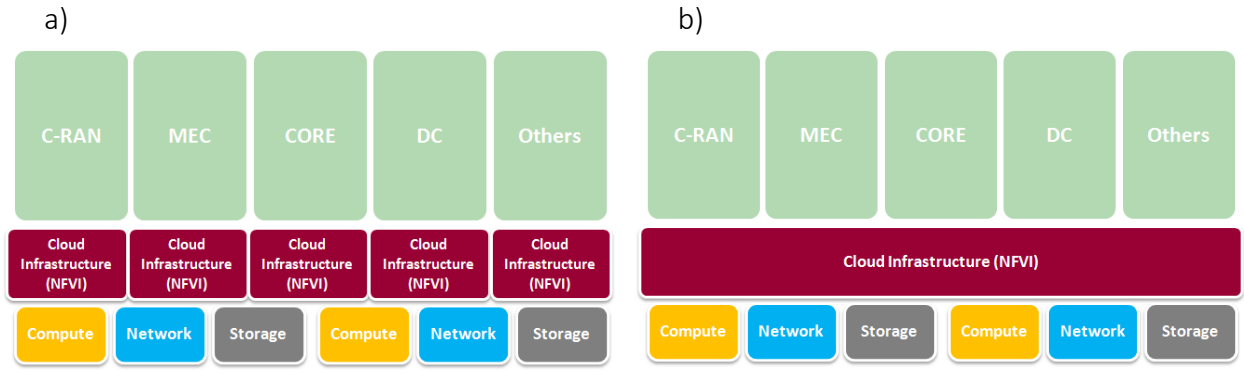


Figure 11: NFVIs per Service.

5.1.2 VIMs and NFVIs

A Virtualized Infrastructure Manager (VIM) function is responsible to manage the infrastructure (NFVI). A single VIM can manage an NFVI, which is probably the simples approach (see Figure 12). However, a single centralized VIM can also manage all NFVIs (see Figure 13). This latter option can face some limitations, depending on the size of the infrastructure. For this reason, a hybrid approach can be also a good compromise solution (see Figure 14).

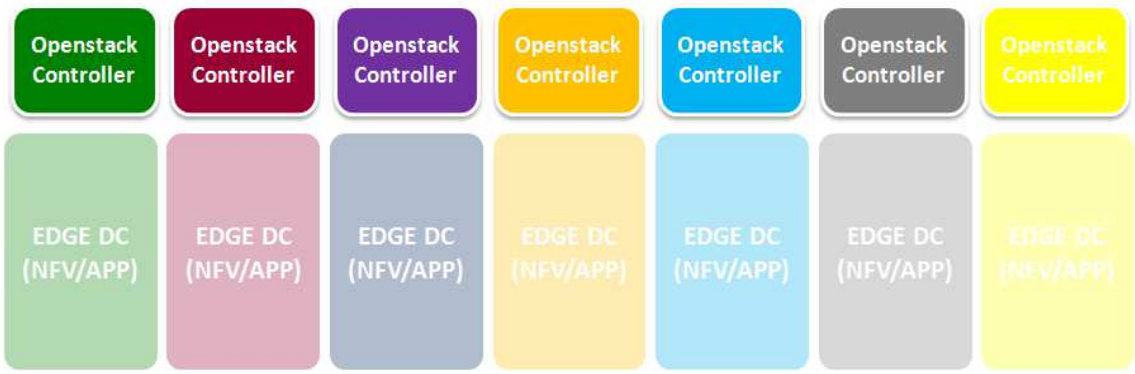


Figure 12: VIMs per NFVI.

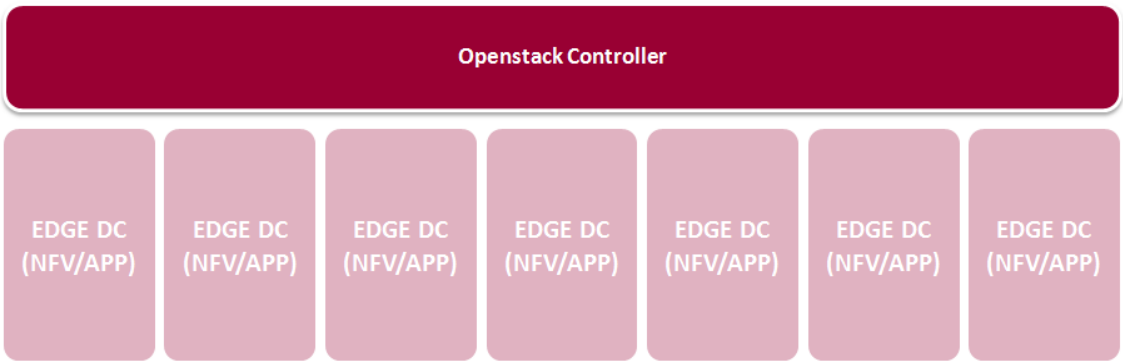


Figure 13: Single VIM for all NFVIs.

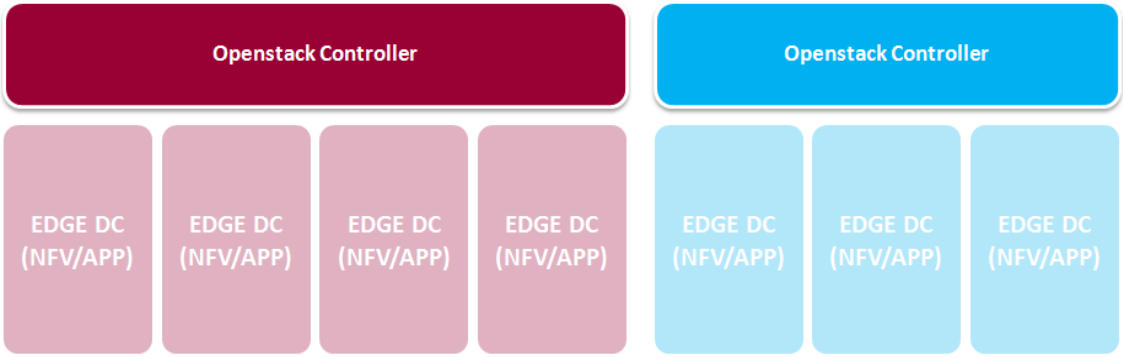


Figure 14: Hybrid; multiple NFVIs per VIM.

5.1.3 Orchestration and Services

The orchestration of complex services can be performed using different approaches. One solution is to have a top-level Orchestrator, capable to manage all services in all locations (see

Figure 15). However, this is very unlikely, since the orchestration task can be very specific and, in the real world, many service providers will bring their own Orchestrator attached. For this reason, a more reasonable approach should be the use of multiple Orchestrators, one per service (See Figure 16).

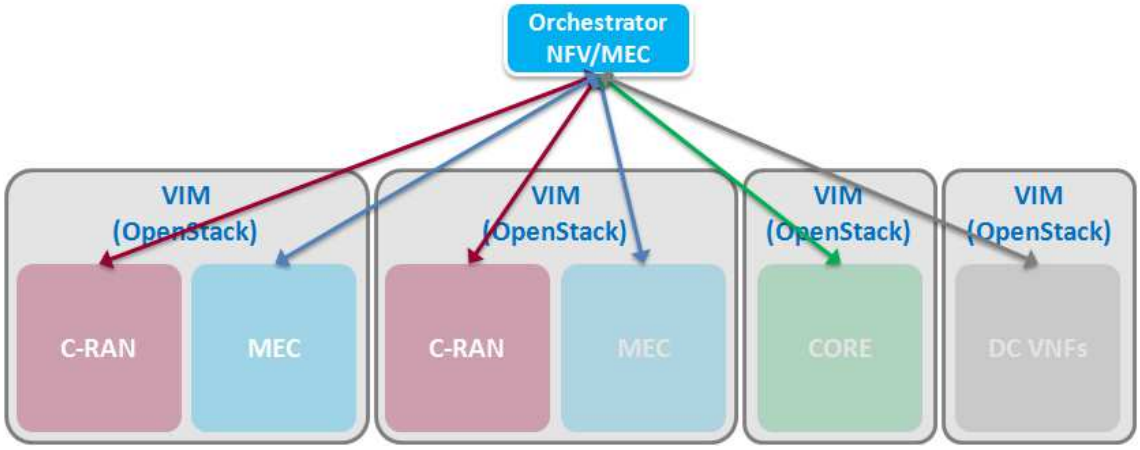


Figure 15: One generic Orchestrator for all Services and locations.

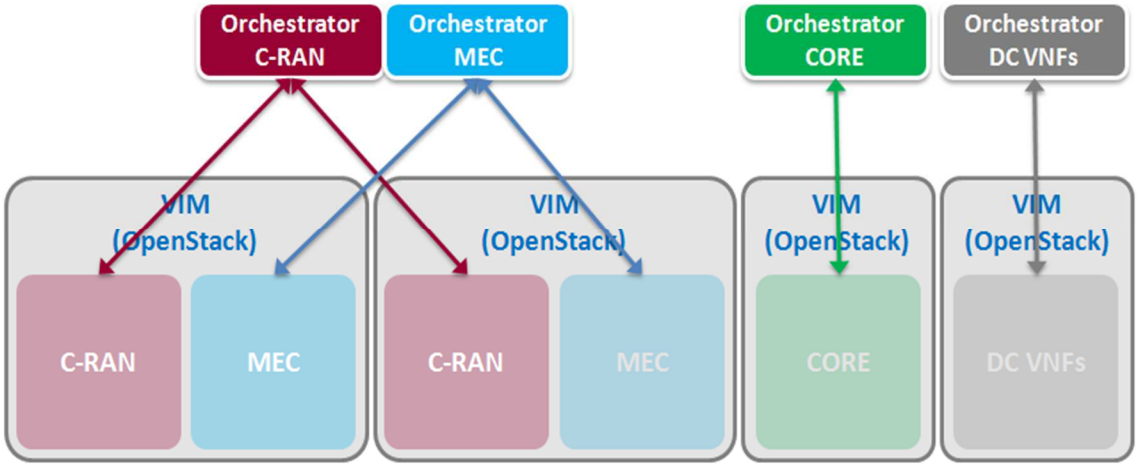


Figure 16: One specialized Orchestrator per Service (for multiple locations).

5.1.4 Orchestration Integration

In case where multiple service Orchestrators exist, one per service, there is a need for some integration between them and, eventually, an additional (top) level of Orchestration. The Figure 17 depicts three integration models: (1) in the North-South integration, all service Orchestrators communicate only with a top-level Orchestrator, which simplifies integration as service Orchestrators only need to integrate with the top-level one; (2) in the East-West integration, service Orchestrators communicate with each others', making integration more complex. There is also a hybrid approach, which comprises both models altogether (1+2).

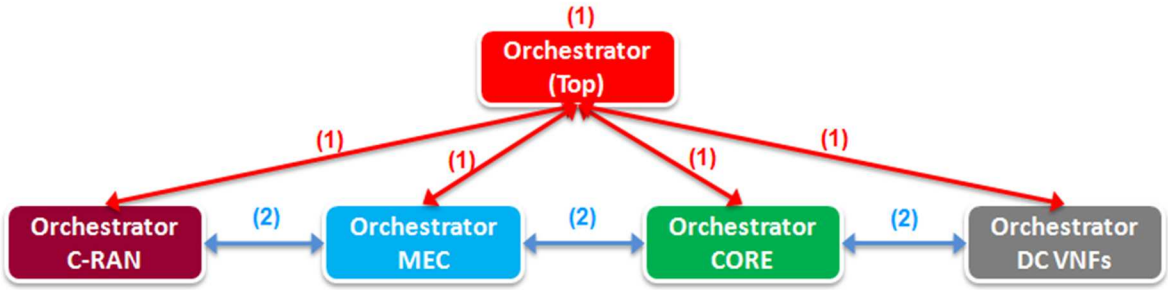


Figure 17: North-South Integration(1), East-West Integration(1) and Hybrid Integration (1 and 2).

5.2 "Traditional" NFVI infrastructure: VNF scaling with feedback from measurements

5.3 Container based NFVI infrastructure for 5G RANs

Today, the network is a network of elements (e.g., an e-NodeB) hosting a set of functions. However, using virtualization technology to address 5G requirements, it makes sense to re-examine this functional split across set of RFBs to introduce the required 5G flexibility. In addition, the same functionality, such as authentication/authorization, is currently implemented in different network elements (i.e., packet data network gateways and mobility management entities) to ensure vendor compatibility. Through the decomposition, each functionality (authentication, authorization, radio resource management, turbo decoding, fast Fourier transform, etc.) will be dedicated to an RFB, thus allowing a modular design of 5G. Any update of any specific RFB is now doable without interfering with other RFBs.

In Deliverable D2.2 [25], a full analysis of the different processes supported in each layer of the RAN stack (Physical, MAC, RLC, RRC, PDCP) has been conducted. A classification of the identified processes is proposed based the nature of the process (control or data plane), synchronization to low layer time (TTI), scope of the process (user based or Cell based). Based on that classification, a set of RFBs are proposed.

Due to real time requirements in the RAN part (for example, the scheduler has to decide each 1ms on the way the frame filled), we select the container technology as an environment of execution of RFB of the RAN. This choice is motivated by the performance of the container technology compared to Virtualization Machine (using Xen or KVM as hypervisor) in terms of boot time, RTT (ping flood) and throughput (using iperf) as it is measured in [20]. The structure of each technology is depicted in the following figure:

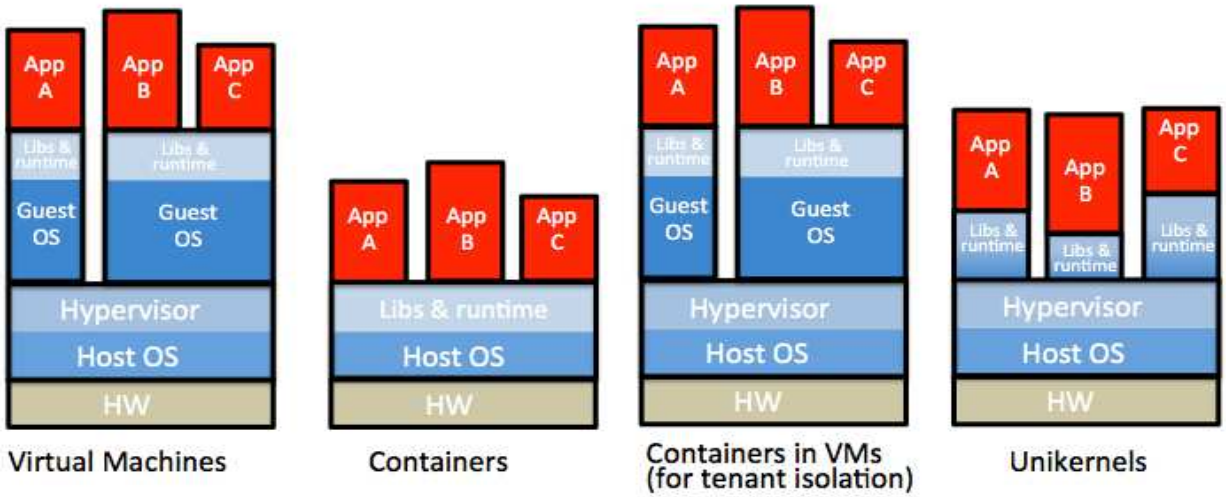


Figure 18: Virtual Machine, Container and Unikernel (source [21])

Docker is a new leading container based technology that offers a more efficient and lightweight approach to application deployment. Its eco-system offers a simple way to define a service chain as a multi-container application considering all the dependencies in a single file thanks to docker compose command. The service chain structure and configuration is held in a single place, which makes spinning up services simple and repeatable.

The Compose file is a YAML file defining services, networks and volumes. YAML stands for *"YAML Ain't Markup Language"* taking concepts from programming languages such as C. It share a set of similarities to XML.

5.4 "Traditional" NFVI infrastructure : the MEC scenario

A key enabler for low-latency scenarios of 5G networks is the concept of Mobile Edge Computing (MEC) [17], whose main idea is to deploy computing capabilities near to end users. ETSI has specifically devoted an ISG called MEC (*Mobile Edge Computing*), to standardize the way application developers and content providers can run their services from the edge MEC supporting Radio Access Network (RAN) processing and third party applications. The MEC technology brings a set of advantages: i) ultra-low latency, ii) high bandwidth, iii) real-time access to radio network information and iv) location awareness. As a result, operators can open the radio network edge to a third party, allowing them to rapidly deploy innovative applications and services towards mobile subscribers, enterprises and other vertical segments. One of the goals of Superfluidity is to integrate MEC in the overall architecture, so that the MEC platform can rely on the same physical resources of an Extended-NFVI.

By introducing the notion of RFB in MEC, it becomes easier to: i) scrutinize constituent elements of the aforementioned components, ii) distinguish those that can be virtualized, implemented and re-utilized in the most efficient way, and iii) provision a network capable of handling its resources with a higher degree of granularity.

Applications are the most dynamic entities of the MEC Architecture. The whole architecture is devoted to provide an agile environment to MEC Apps, making them closer to end users, even when those are moving along different attachment points. The utilization of reusable functional blocks (RFBs) in MEC ecosystems takes advantage of the fact that Applications are usually built based on multiple components, typically associated to different layers, e.g. load balancers, webserver, database, back-end. The reutilization of these components permits a more flexible management of MEC Apps. Namely, it allows the deployment of the appropriate computation capacity for each layer, and can easily accommodate MEC Apps changes (scale in or scale out) to the performance requirements along the time.

Other functional blocks (other than MEC Apps) can also benefit from the concept of RFBs. The MEC architecture uses a network function (NF) to inspect and offload certain traffic towards the appropriate MEC Application to serve the users from the edge. This NF can be virtualized (VNF) or not. Although this function is out of the scope of the MEC standardization work, it is usually referred to as TOF (Traffic Offloading Function), since this was the name used at the beginning of the ETSI MEC activity. Note that the TOF name has been removed from the final MEC architecture picture. When implemented as VNF, the TOF can also be deployed as a set of RFBs, as it is composed by smaller components such as, GTP encap/decap, Traffic Filter, Router, or D/SNAT, among others. TOF can take advantage of this modularity by deploying and scaling this VNF according to the performance requirements.

Finally, the Management and Orchestration components of the MEC Architecture can also be decomposed into smaller and reusable blocks. However, as those components are not expected to be deployed or scaled dynamically (although it could make sense under some circumstances), the utilization of the RFB concept is more limited. It is just a matter of software development good practices.

5.5 Unikernel based Hypervisors

5.6 Click-based environments

Click [15] is a software architecture that decomposes the functionality of a packet-forwarding node into elements that are interconnected to implement arbitrarily complex functions. The Click elements operate on packets (e.g. at IP or Ethernet level) by performing inspections and/or modifications of the different fields. A large number (hundreds) of built-in Click elements are available, including for example Packet Classifiers, Rate Monitor, Bandwidth Shapers, and Header Rewriters. The interconnection graph of Click elements is called a configuration, which is described by a declarative language. The language allows the definition of compound elements composed of other elements. In Superfluidity, Click is one of the RFB Execution Environments. In

our vision, a single Click element represent a RFB and a configuration that combines Click elements represents again a (composed) RFB. In [19], it is shown how a Click configuration can be executed in minimalistic virtual machines that runs in a specialized hypervisor called ClickOS. ClickOS Virtual Machines can be seen as RFBs and can be composed to implement the desired packet processing services.

To give an example of the Click capacity to build complex forwarding behaviour, the functionality of a Cisco ASA (Adaptive Security Appliance) [7] in a real environment (UPB's Computer Science Department network) has been analysed. The corresponding Click configuration has been reconstructed. Figure 19 shows a simplified "Out-In" traffic pipeline of the ASA model. The ASA model contains over 200 Click elements. It performs VLAN switching, routing, traffic filtering, static and dynamic NAT. To generate an ASA model, we have developed a tool, which parses a subset of ASAs' configuration file, and builds its corresponding Click model. We have validated our generation tool using "black-box" testing. We have sent TCP, UDP, raw-IP and ICMP packets through a real ASA and its model, and compared the results. A detail of the actual Click elements used by the TCP traffic classification block is shown in Figure 20.

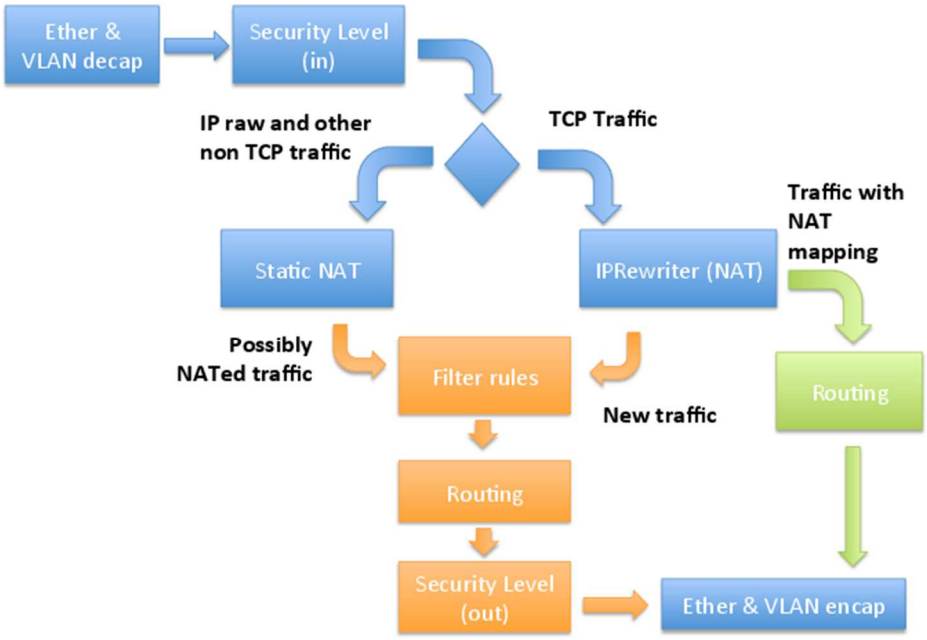


Figure 19: High-level decomposition of Cisco ASA

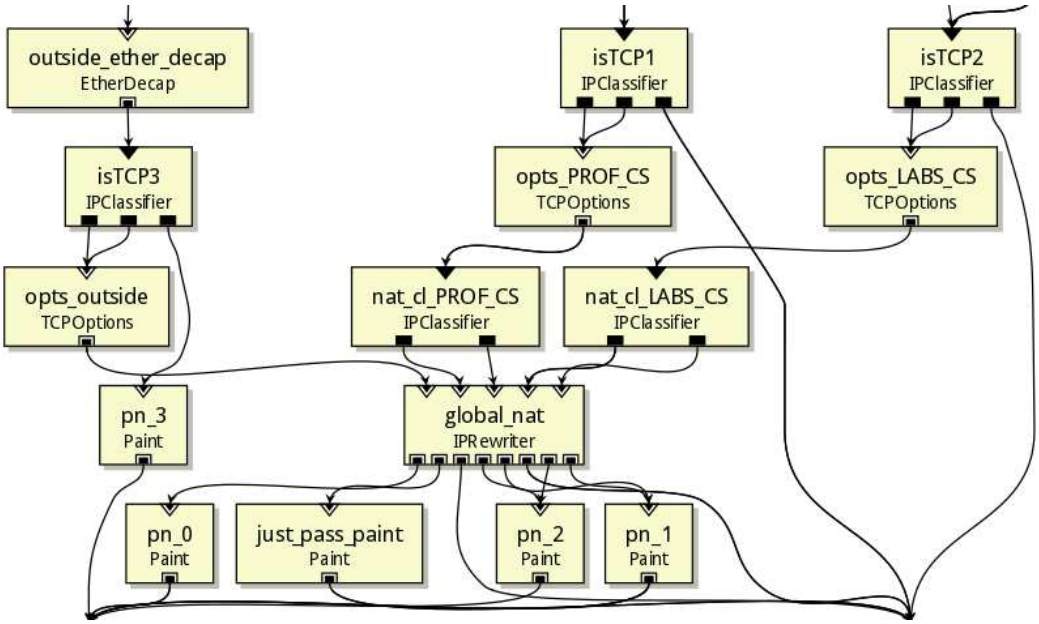


Figure 20: detail around TCP traffic classification block.

5.7 Radio processing modules

[See activity in Task T4.3]

5.8 Programmable data plane network processor using eXtended Finite State Machines

In Deliverable D2.2 [25], section 4.5, we have introduced a convenient platform agnostic programming abstraction for data plane network processing tasks (e.g. switching, routing, traffic processing, etc), based on eXtended Finite State Machines (XFSMs). With respect to the architecture terminology introduced in this deliverable, in D2.2 we have listed:

1. The relevant (nano) RFBs, i.e. the set of (in this case very) elementary primitives the programmer can use as "instructions" to program a network processor node, and
2. The RDCL (RFB Description and Composition Language), which in this case was conveniently found to be an XFSM, being an XFSM an abstract and platform agnostic formal model which permits to combine elementary actions, events, conditions, and ALU-like arithmetic and logic micro-instructions into a stateful behavioural formal model.

It remains to specify how an XFSM can be "executed" by a software or hardware platform, i.e. it remains to specify how, concretely, an RFB Execution Environment (using section 3's notation) for a high performance data plane network processor should be engineered. In the remainder of this section we complete this final step, and propose an initial reference architecture for a

programmable network processor, which we call Open Packet Processor (OPP) which supports processing tasks formally described in terms of XFSMs.

Unless otherwise specified, we will refer to OPP in terms of reference HW design. The reasons is twofold. The first one is technical: of course an HW design is more challenging and restrictive over a SW one, as it needs to show that the proposed reference RFB Execution Environment is capable by design to process each packet in a limited and bounded number of clock cycles per packet. The second reason is more strategical: we believe, and we will argument in the next "digression" section 5.8.1, that virtualization should not restrict to SW executed on commodity CPUs (e.g. x86 or ARM) but should also entail domain-specific general purpose processors, for which our OPP is a concrete proposal. The reader just interested in the OPP architecture may skip section 5.8.1, and directly move to the next section 5.8.2.

5.8.1 Digression: network function virtualization on domain-specific HW

While the specification currently in progress within the ETSI Network Function Virtualization (NFV) ISG retains a quite high level of generality in terms of Infrastructure specification (namely, the underlying NFVI, Network Function Virtualization Infrastructure), reliance on commodity processors such as x86 or ARM as platforms for running Virtual Machines implementing virtualized network functions is often implicitly assumed.

While such a choice seems of course optimal in terms of (low) cost and (maximum) flexibility, it brings about obvious performance concerns, especially when dealing with network functions which require intensive packet-level processing at (ideally) line rate. Take for instance very basic network functions such as switching. Despite the extensive effort carried out by companies specialized and research groups specialized in software acceleration on commodity computing platforms, the gap between HW-based switching chips and SW-based ones is still significant and not going to decrease in the future. (see Figure 21, taken from a presentation of Nick McKeown, Stanford University, 2014).

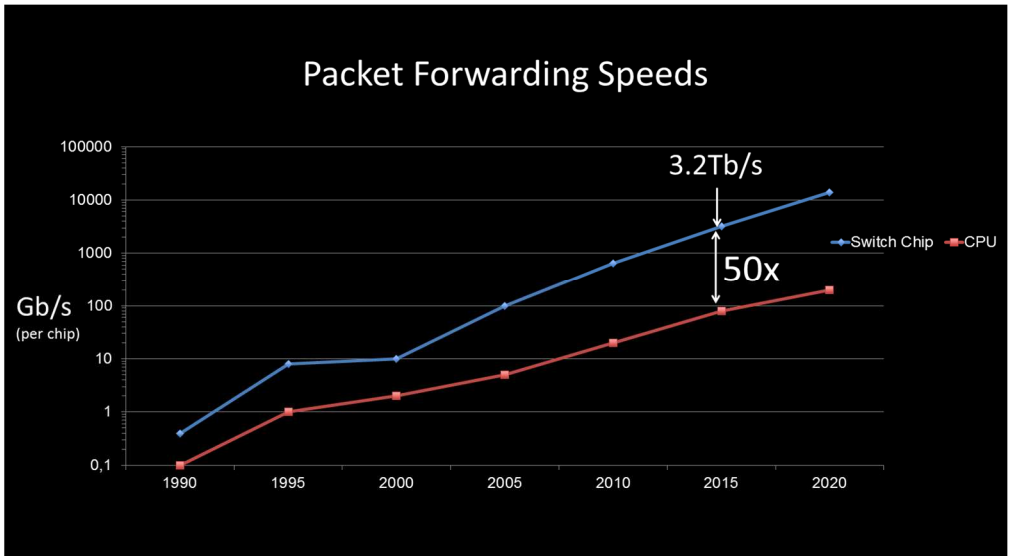


Figure 21: HW vs SW switches forwarding speeds

Indeed, massive I/O parallelization is possible only in an HW switching chip. And, some very basic match/lookup primitives, such as wildcard matching, are extremely efficient when implemented in HW (e.g., on commodity TCAMs), unlike their SW implementation counterpart (at least in the most general wildcard match case, so far there is no algorithm which can perform at O(1) speed/complexity). The situation becomes even worse when dealing with flow/packet-level functions such as traffic classifiers or policers, traffic control algorithms (say an AQM algorithm), packet schedulers, packet-level monitoring tasks, security-related functions (such as DPI or stateful firewalls), and so on. Running such functions directly on the switches' fast-path is today unaffordable with commodity CPUs, and in fact 'pure' SW switches usually divert such flows over the slow path, whereas commercial brand-name switches rely on custom HW.

Domain-specific platforms.

Interestingly, networking is a field where general-purpose commoditized domain-specific platforms are somewhat still lagging behind. In fields such as graphics and video/image processing, general purpose commodity programmable platforms, namely GPUs, have taken up since long time, and provide the applications' developers with a dedicated HW platform which retains full flexibility and programmability, but which is also tailored and performance-optimized towards the very special type of processing needed in this field. The same holds in fields such as signal processing (with DSP platforms), and to some extent also in the field of cryptography (with crypto accelerators). Conversely, in the networking field, bare metal switches are just starting to emerge, and (besides the pioneering work in OpenFlow which however is functionally limited to support "just" programmable Flow Tables, and besides the more recent work on P4 where, however, a reference architecture for a P4 target chipset – albeit already claimed to be manufactured by Barefoot - is still not made public) a clear and neat

interface between hardware functions and their software configuration and control is still missing.

A closer look at the above issue reveals that the real crux of the matter is not only technological, but it also (and mostly) revolves around the identification of what should be the network and traffic processing "primitives" or blocks supported by a platform, and how such blocks should be combined to produce meaningful and realistically complex network functions. In other words, the problem is not (only) related to the feasibility and viability of a programmable platform for developing network functions, but stems into the i) appropriate functional decomposition of an higher-level network function into smaller and reusable network and processing blocks, and into ii) the identification of how to formally describe (and run-time enforce) how such blocks combine together to form a desired higher level function. An enlightening example of what has been done so far in this field, and hence what is still missing, is OpenFlow. On one side, OpenFlow has clearly identified elementary "actions" which are pre-implemented in the switch and which can be therefore exploited by a third party "programmer", and has left full room for extensions in the supported set of actions. On the other side, the OpenFlow match/action abstraction that the programmer can use to configure a desired Flow Table is largely insufficient to formally describe more complex (e.g. stateful) flow processing tasks. As a result, most of the today's network programming frameworks circumvent OpenFlow's limitations by promoting a "two-tiered'' programming model: any stateful processing intelligence of the network applications is delegated to the network controller, whereas OpenFlow switches limit to install and enforce stateless packet forwarding rules delivered by the remote controller.

In the next sections, we describe our proposal for an Open Packet Processor. It starts from the OpenFlow model of decoupling actions from matches, but it further permits the programmer to describe the switch operation via a behavioural XFSM-based model which can be thus used as network processor's programming language.

5.8.2 The Open Packet Processor Architecture

In this section, we discuss into details a novel switch reference architecture which permits to concretely support a **decomposition approach, which we descriptively refer to as "nano-decomposition"** (see Deliverable D2.2, section 4.5). Rather than employing relatively high level functions, the idea is to define a set of extremely elementary and stateless actions and primitives as building blocks. In such model, we would not handle, say, a "load balancer function" but we would rather handle much more elementary forwarding and processing instructions (for instance, send packet from flow X to port Y, increment a counter, etc). To make a concrete example, a load balancer would not anymore be treated as a basic function block, but might be in turns "programmed" using a suitable combination of such very elementary primitives.

Challenges and requirements

In order to properly exploit "nano-decomposition", we wish to emerge with a packet processing architecture which holds the following four properties.

(1) Ability to process packets directly on the fast path, i.e., while the packet is traveling in the pipeline (nanoseconds time scale). More specifically, we would like to emerge with a solution which, *if implemented in HW*, would meet the requirement of *performing packet processing tasks in a deterministic and small (bounded) number of HW clock cycles*. Indeed, as duly discussed in the introduction, network functions should not be restricted to be implemented on commodity CPUs, but should be eventually able to exploit, when high performance is mandated, hardware platforms. The reason is that a software program running on a commodity x86 or ARM platform might not sustain network functions which require intensive packet-level processing at multi gbps line rate.

(2) Efficient storage and management of per-flow stateful information. As anticipated in Deliverable D2.2, section 4.5, we envision nano-decomposition as based on very elementary stateless actions or instructions. Hence, stateful management of flows is now mandated to the platform, opposed to the function block. Stateful management is required to permit the programmer to further use the *past* flow history for defining a desired per-packet processing behaviour (for instance, if a threshold number of packets has arrived for a flow, reconfigure the forwarding decision). As shown next, this can be easily accomplished by *pre-pending* a dedicated storage table (concretely, an hash table) that permits to retrieve, in O(1) time, stateful flow information. We name this structure as Flow Context Table, as, in somewhat analogy with context switching in ordinary operating systems, it permits to retrieve stateful information associated to the flow to which an arriving packet belongs, and store back an updated context the end of the packet processing pipeline. Such (flow) context switching will operate at wire speed, on a packet-by-packet basis.

(3) Ability to specify and compute a wide (and programmable) class of stateful information, thus including counters, running averages, and in most generality stateful features useful in traffic control applications. It readily follows that the packet processing pipeline, which in standard OpenFlow is limited to match/action primitives, must be enriched with means to describe and (on the fly) enforce conditions on stateful quantities (e.g. the flow rate is above a threshold, or the time elapsed since the last seen packet is greater than the average inter-arrival time), as well as provide arithmetic/logic operations so as to update such stateful features in a bounded number of clock cycles (ideally one).

(4) Platform independence. A key pragmatic insight in the original OpenFlow abstraction was the decision of restricting the OpenFlow switch programmer's ability to just *select* actions among a

finite set of supported ones (opposed to permitting the programmer to develop own custom actions), and associate a desired action set (bundle) to a specific packet header match. We conceptually follow a similar approach, but we cast it into a more elaborate eXtended Finite State Machine (XFSM) model. Indeed, an XFSM abstraction permits us to formalize complex behavioral models, involving custom per-flow states, custom per-flow registers, conditions, state transitions, and arithmetic and logic operations. Still, an XFSM model does not require us to know how such primitives are concretely implemented in the hardware platform, but "just" permits us to combine them together so as to formalize a desired behaviour. Hence, it can be ***ported*** across platforms which support a same set of primitives.

eXtended Finite State Machines

We recall, from Deliverarable D2.2, section 4.5, that an eXtended Finite State Machine is an abstraction which is formally specified (see Table 2), by means of a 7-tuple **<I,O,S,D,F,U,T>**. Input symbols **I** (OpenFlow-type matches) and Output Symbols **O** (actions) are the same as in OpenFlow. Per-application states **S** permit the programmer to freely specify the possible states in which a flow can be, in relation to her desired custom application (technically, a state label is handled as a bit string). For instance, in an heavy hitter detection application, a programmer can specify states such as **NORMAL**, **MILD**, or **HEAVY**, whereas in a load balancing application, the state can be the actual switch output port number (or the destination IP address) an already seen flow has been pinned to, or **DEFAULT** for newly arriving flows or flows that can be rerouted. With respect to a Mealy Machine, the key advantage of the XFSM model resides in the additional programming flexibility in three fundamental aspects.

(1) **D: Custom (per-flow) registers and global (switch-level) parameters.** The XFSM model permits the programmer to explicitly define her own registers, by providing an array **D** of per-flow variables whose content (time stamps, counters, average values, last TCP/ACK sequence number seen, etc) shall be decided by the programmer herself. Additionally, it is useful to expose to the programmer (as further registers) also switch-level states (such as the switch queues' status) or ``global'' shared variables which all flows can access. Albeit practically very important, a detailed distinction into different register types is not foundational in terms of abstraction, and therefore all registers that the programmer can access (and eventually update) are summarized in the XFSM model presented in Table 2 via the **array D of memory registers**.

(2) **F: Custom conditions on registers and switch parameters.** The sheer majority of traffic control applications rely on *comparisons*, which permit to determine whether a counter exceeded some threshold, or whether some amount of time has elapsed since the last seen packet of a flow (or the first packet of the flow, i.e., the flow duration). The **enabling functions f_i:D→{0,1}** serve exactly for this purpose, by

implementing a set of (programmable) boolean comparators, namely conditions whose input can be decided by the programmer, and whose output is 1 or 0, depending on whether the condition is true or false. In turns, the outcome of such comparisons can be exploited in the transition relation, i.e. a state transition can be triggered only if a programmer-specific condition is satisfied.

(3) U: Register's updates. Along with the state transition, the XFSM models also permits the programmer to update the content of the deployed registers. As we will show later on, registers' updates require the HW to implement a set of update functions $u_i: D \rightarrow D$, namely arithmetic and logic primitives which must be provided in the HW pipeline, and whose input and output data shall be configured by the programmer.

Finally, we stress that the actual computational step in an XFSM, i.e. the step which determines how the XFSM shall evolve on the basis of an arriving packet, resides in the transition relation $T: S \times F \times I \rightarrow S \times U \times O$, which is ultimately nothing else than, again, a ``map'' (albeit with more complex inputs and outputs than the basic OpenFlow map), and hence is naturally implemented by the switch TCAM, as shown next. In other words, what makes an XFSM a compelling abstraction is the fact that its evolution does not require to resort on a CPU, but can be enforced by an ordinary switch's TCAM.

XFSM formal notation		Meaning
I	input symbols	all possible matches on packet header fields
O	output symbols	OpenFlow-type actions
S	custom states	application specific states, defined by programmer
D	n-dimensional linear space $D_1 \times \ldots \times D_n$	all possible settings of *n* memory registers; include both custom per-flow and global switch registers.
F	set of enabling functions $f_i: D \rightarrow \{0,1\}$	Conditions (boolean predicates) on registers
U	set of update functions $u_i: D \rightarrow D$	Applicable operations for updating registers' content
T	transition relation $T: S \times F \times I \rightarrow S \times U \times O$	Target state, actions and register update commands associated to each transition

Table 2: Formal specification of an eXtended Finite State Machine (left two columns) and its meaning in our specific packet processing context (right column)

Open Packet processor: reference architecture design

To our view, what makes the previously described XFSM abstraction compelling is the fact that it does not necessarily mandate for a SW implementation (which of course is clearly a possibility), but if needed it can be directly executed on the switch's fast path using off the shelf HW and without resorting on any CPU. To support this claim, the next Figure 22 shows a possible reference implementation of an architecture which we refer to as "Open Packet Processor", in

short **OPP**. Such architecture is on purpose described in terms of building blocks, which can therefore be implemented in hardware (and when commenting about implementation issues, by default we will refer to a possible HW implementation as this is the most challenging one since it requires to guarantee a bounded number of clock cycles per block/operation).

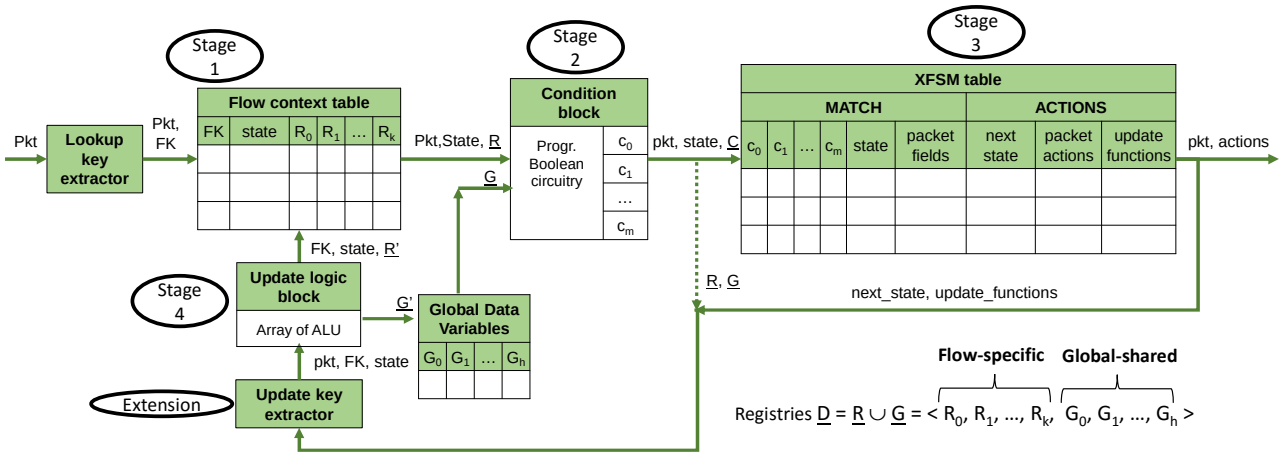


Figure 22. Open Packet Processor Architecture

The packet processing workflow for a packet traveling across the architecture depicted in Figure 22 is best explained by means of the following *stages*.

Stage 1: flow context lookup. Once a packet enters an OPP processing block, the first task is to extract, from the packet, a **Flow Identification Key** (**FK**), which identifies the entity to which a state may be assigned. The flow is identified by an unique key composed of a subset of the information stored in the packet header. The desired FK is configured by the programmer (an IP address, a source/destination MAC pair, a 5-tuple flow identifier, etc) and depends on the specific application. The FK is used as index to lookup a **Flow Context Table**, which stores the *flow context*, expressed in terms of (i) the **state label s_i** currently associated to the flow, and (ii) an array $\underline{R}=\{R_0, R_1, ..., R_k\}$ of (up to) k+1 registers defined by the programmer. The retrieved flow context is then appended as metadata and the packet is forwarded to the next stage.

Stage 2: conditions' evaluation. Goal of the **Condition Block** illustrated in the above Figure (and implementable in HW using ordinary Boolean circuitry) is to compute programmer-specific conditions, which can take as input either the per flow register values (the array $\underline{R}$), as well as global registers delivered to this block as an array $\underline{G}=\{G_0, G_1, ..., G_h\}$ of (up to) h+1 global variables and/or global switch states. Formally, this block is therefore in charge to implement the **enabling functions** specified by the XFSM abstraction. In practice, it is trivial to extend the assessment of conditions also to packet header fields (for instance, port number greater than a given global variable or custom per-flow register). The output of this block is a boolean vector $\underline{C}=\{c_0, c_1, ..., c_m\}$ which summarizes whether the i-th condition is true (c_i=1) or false (c_i=0).

Stage 3: XFSM execution step. Since boolean conditions have been transformed into 0/1 bits, they can be provided as input to the TCAM, along with the state label and the necessary packet header fields, to perform a wildcard matching (different conditions may apply in different states, i.e. a bit representing a condition can be set to "don't care" for some specific states). Each TCAM row models one transition in the XFSM, and returns a 3-tuple: (i) the next state in which the flow shall be set (which could coincide with the input state in the case of no state transition, i.e., a self-transition in the XFSM), (ii) the actions associated the transition (usual OpenFlow-type forwarding actions, such as `drop()`, `push_label()`, `set_tos()`,etc., and (iii) the micro-instructions needed to update the registers as described below.

Stage 4: register updates. Most applications require arithmetic processing when updating a stateful variable. Operations can be as simple as integer sums (to update counters or byte statistics) or can require tailored floating point processing (averages, exponential decays, etc). The role of the **Update logic block** component highlighted in Figure 22 is to implement an **array of Arithmetic and Logic Units (ALUs)** which support a selected set of computation primitives which permit the programmer to update (re-compute) the value of the registers, using as input the information available at this stage (previous values of the registers, information extracted from the packet, etc). The updated registry values are then stored in the relevant memory locations (flow registries and/or global registries). More into details, since we are not relying on any general purpose CPU, but we are free to design in an HW implementation our "own" specific array of ALUs, the above mentioned computational primitives shall be most conveniently implemented as a domain-specific (i.e., specific for packet processing task) instruction set, which includes either micro-instructions typically found in a standard RISC processor (logic, arithmetic and bitwise instructions), as well as dedicated more complex microinstructions useful to traffic control applications (e.g. compute running averages, standard deviations, exponentially weighted running averages, etc). A detailed overview of the candidate microinstructions supported by the array of such ALUs has been reported in Deliverable D2.2, section 4.5.

Extension: Cross-flow context handling. There are many useful stateful control tasks, in which states for a given flow are updated by events occurring on *different* flows. A simple but prominent example is MAC learning: packets are forwarded using the *destination* MAC address, but the forwarding database is updated using the *source* MAC address. Thus, it may be useful to further generalize the XFSM abstraction, i.e. by permitting the programmer to **use a Flow Key during lookup** (e.g. read information associated to a MAC destination address) and **employ a possibly different Flow Key** (e.g. associated to the MAC source) **for updating a state or a register**. The possible Flow Key differentiation between lookup and update phases is highlighted in Figure 22 by the usage of two different **Key Extractor blocks**.

6 State of the art

This section reports on different state of the art activities relevant to the Superfluidity architecture definition process.

A set of ongoing architecture definition activities in SDOs and industry associations are covered in sections 6.1 and 6.2. Section 6.3 tries to identify a convergence approach for NFV, SDN, C-RAN and MEC. Section 6.4 introduces the convergence among Mobile Core of 4G/5G networks and the "central" cloud Data Centres. The sections 6.3 and 6.4 provides the background reasoning for the overall architectural framework introduced in section 2 (Figure 1).

6.1 Latest 3GPP advancement toward 5G

Towards a more flexible and open network, past 3GPP releases, i.e. Rel-11, Re-12, and Rel-13, introduced a set of new features related to machine type communication and RAN sharing capability where some network entities are shared among virtual operators. These trends will be strengthened in forthcoming releases to be natively supported in future 5G networks.

Toward an API-driven customizable architecture, Enhancements for Service Capability Exposure feature introduced in Rel-13 provides means to securely expose the services and capabilities provided by 3GPP network interfaces to external application providers. This feature is used to support small data services as well as machine type communication (MTC).

Currently, flexibility in RAN is supported by the concept of a capacity broker for RAN sharing [11]. The resource allocation is provided on demand to virtual mobile network operator (VMNO) using signalling.

Beyond the RAN sharing concepts, 3GPP has defined two architectures in 3GPP SA2 [12]: Multi-Operator Core Network (MOCN) feature, where each operator has its own Evolved Packet Core (EPC) and a shared eNBs and Gateway Core Network (GWCN) feature, where also the MME is shared between operators.

In parallel to this 3GPP enhancements, cloud computing technologies and cloud concepts gained momentum not only from the information technology (IT) perspective, but also within the telecom world. Emerging processing paradigms such as mobile edge computing (MEC) and Cloud-RAN (C-RAN) have to be accommodated to enable a flexible deployment and to support programmability to meet different stringent requirements in terms of latency, robustness, and throughput.

Recently, the Network Functions Virtualization (NFV) trend of implementing network services as virtualized software running on commodity hardware has gotten a lot of attraction. 3GPP started to study the support of NFV considering the network management perspective in [13].

In Rel-14, some specifications on architecture requirements for virtualized network management have been introduced [14].

It is clear that NFV is an important piece for a convergence solution to build a natively customizable and adaptable 5G network changing the current paradigm of product specific application architecture. Superfluidity goes beyond the NFV framework to foster programmability as a main architectural feature of future 5G network.

6.2 NFV, SDN & MEC Standardization

NFV (*Network Function Virtualization*) is the ETSI group devoted to standardizing the virtualization of *Network Functions* (NFs). It intends to specify the architecture required to accommodate the challenges of the new virtualization paradigm, covering the runtime and management aspects, in order to manage the entire lifecycle of a VNF (*Virtual Network Function*). Furthermore, it also comprises the management of *Network Services* (NSs), which can be built by orchestrating multiple VNFs, according to a *Forwarding Graph* (FG), using a catalog-driven approach.

The ONF (*Open Networking Foundation*) is an organization devoted to promote the utilization of software-defined technologies to program the network. Following a *Software-Defined Networking* (SDN) approach, the network is separated into three different parts: the user-data plane, the controller plane and the control plane. The user-data plane is responsible for forwarding the user data, while the controller plane is composed by SDN controllers, which provide high-level APIs to the control plane above. The control plane is responsible for programming the network, easing the creation of new applications and speeding up the rollout of new services.

MEC (*Mobile Edge Computing*) is the ETSI ISG devoted to standardizing the way application developers and content providers can afford an IT environment to provide their services from the edge. This environment provides ultra-low latency and high bandwidth capabilities, while it has access to operator-provided information, such as location and radio status. ETSI MEC intends to specify an architecture required to accommodate the challenges of computing at the edge, namely providing a dynamic (superfluid) services lifecycle management, allowing applications to cope with the end user mobility, application migration and integration with NFV ecosystems.

The 3GPP (3rd *Generation Partnership Project*) is an organization devoted to producing system specifications regarding cellular networks, including the radio access network, the core network, or services capabilities. It intends to create a detailed description of architectures, network elements, interfaces and protocols, making the interoperation among different vendors and

network providers possible. 3GPP usually relies on existing protocols, like IP, SIP and Diameter, specifying only the particular parameters of operation.

The TMForum (*TeleManagement Forum*) is a telecom industry association devoted to provide guidelines to help operators to create and deliver profitable services. One of the biggest TMForum achievements is the definition of a complete business process (eTOM) and application (TAM) maps, including all activities related to an operator, from the services design to the runtime operation, including monitoring or charging. In order to accommodate the SDN/NFV impacts, the TMForum has created the ZOOM (Zero-touch Orchestration, Operations and Management) program, which intends to build more dynamic support systems, fostering the service and business agility.

In the next section, we further detail the architecture of the preceding Standards Definition Organisations and relate the key architectural concepts to those that we promote in SUPERFLUIDITY.

6.3 Cross Standards architecture

6.3.1 ETSI NFV architecture

The creation of the ETSI NFV ISG intended to bring to the telco sector the appropriate IT tools to take advantage of cloud principles, like on-demand, agility, scalability or pay-as-you-go (PAYG), among others. The hardware and software decoupling and consequent utilization of COTS (*Common Off-The-Shelf*) hardware can also be applied to network functions, leading to a cost reduction, increasing of network agility and vendor independency.

NFV is seen as a critical architectural building block to give mobile operators the necessary tools to address the massive data usage by their customers in the new of wave of expected 5G use cases. NFV enables the “cloudification” of network functions (NFs). For this, the network function has to be implemented apart from dedicated/ specialized hardware, and be able to run on top of COTS. Typical examples of VNFs are routers or firewalls, but it can also be mobile or fixed components, like P-GWs or eNBs. Figure 23 shows the NFV principle.

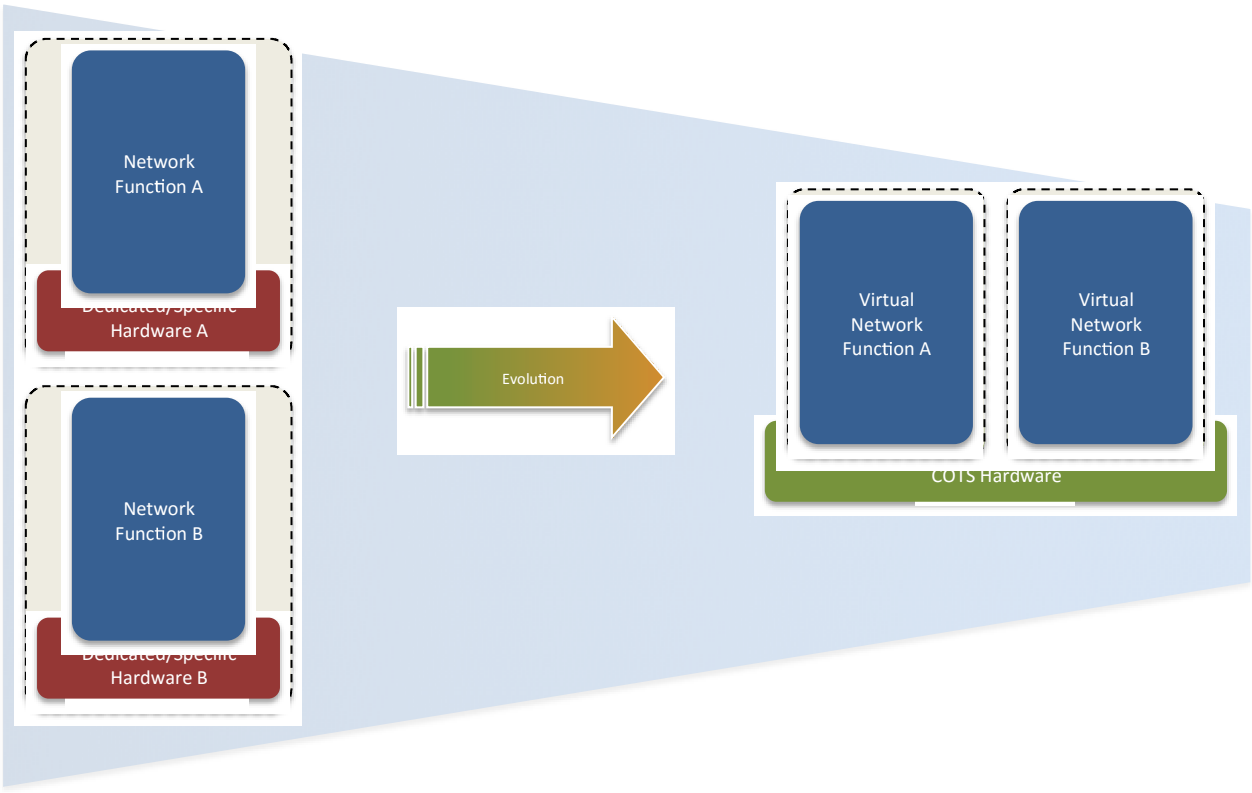


Figure 23: ETSI NFV concept.

The "cloudification" of network functions can be further enhanced by using the management and orchestration environment. In such cases, the environment manages the entire VNFs lifecycle, performing not just the deployment and disposal, but also managing the runtime operations like migration or scaling, according e.g. to the function load, making a more efficient use of resources. Such a platform is also able to orchestrate combinations of VNFs, creating complex *Network Services* (NS).

Figure 24 depicts a simplified version of the full ETSI NFV architecture. On the left side the execution and control (runtime) operations can be seen, whilst the right side shows management and orchestration. The bottom left shows the virtual infrastructure, which comprises hardware resources (COTS), the virtualization layer (e.g. hypervisors) and virtual resources (e.g. VMs, Containers, Disk, VLANs). VNFs run on top of one or multiple VMs and use network resources. On the top left, the Management Support Services (OSSs/BSSs) interact with the Management and Orchestration (right side) and with the VNFs. In the right, on the bottom, the VIM (e.g. OpenStack) interacts with the NFVI to manage resources. On the top right, the Orchestrator and Management module manages the complete lifecycle of VNFs and orchestrate NSs.

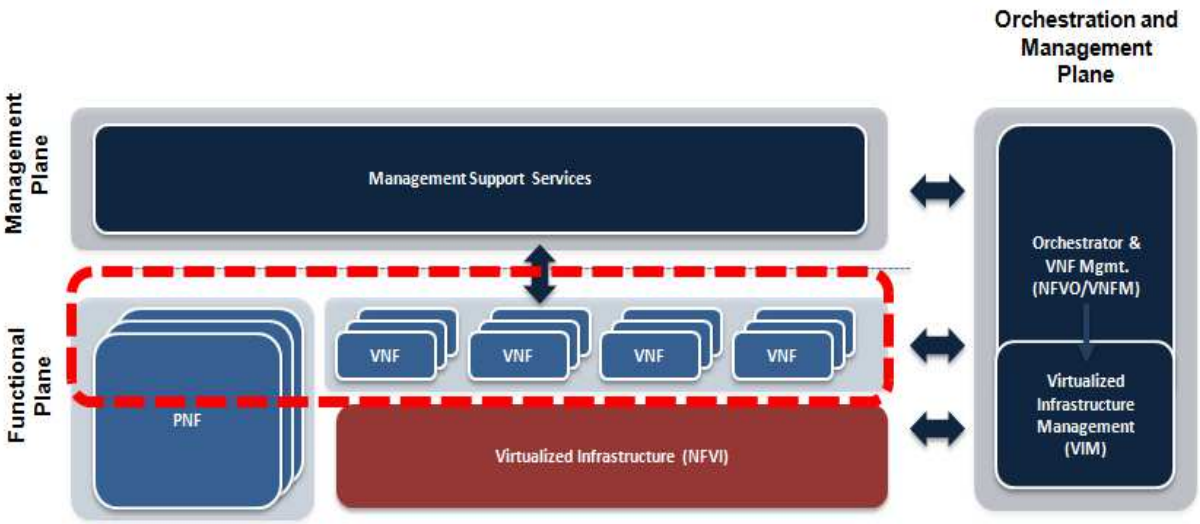


Figure 24: ETSI NFV architecture.

6.3.2 ONF SDN architecture

SDN as core technology in 5G enables operators to simplify service and networking provisioning with visibility across multiple network layers, heterogeneous switch hardware and multi-domain networks. The centralized control of networks SDN provides to operators allows operators to have greater operational flexibility. For example, SDN enables much higher levels of network automation via programmability, which will make 5G networks more adaptable enabling improved service agility. SDN allows operators to dynamically adapt to changing network conditions to support either rapid scale up/down of services or the provisioning of new services, which improves opex by shortening the time to new revenue.

SDN splits the network functions into 3 parts: the user-data plane, the controller plane and the control plane. The user-data plane (or Data Plane) is composed by simple switching Network Elements (NEs), responsible to forward the user traffic according to the basic commands received from the north (controller) interface. The Controller Plane is composed by SDN controllers, which provide basic commands to the south (user-data) and high-level APIs to the north (control or Application Plane). Controller APIs are abstractions used to program the network, speeding up the creation of new services. Figure 25 shows the SDN concept, assuming that the starting point is not a traditional NF relying on a dedicated/specific hardware, but an already virtualized NF (VNF), as described in the section above.

Overall, the NEs forwarding process is controlled by the applications, which use high-level APIs provided by the SDN controllers. The SDN controllers interact with the NEs through low-level southbound APIs to enforce basic forwarding rules, using protocols like CLI, Netconf or Openflow. SDN controllers provide a northbound interface, abstracting the programmer from the network details and making simpler the network service creation. This is one of the key advantages on the SDN model.

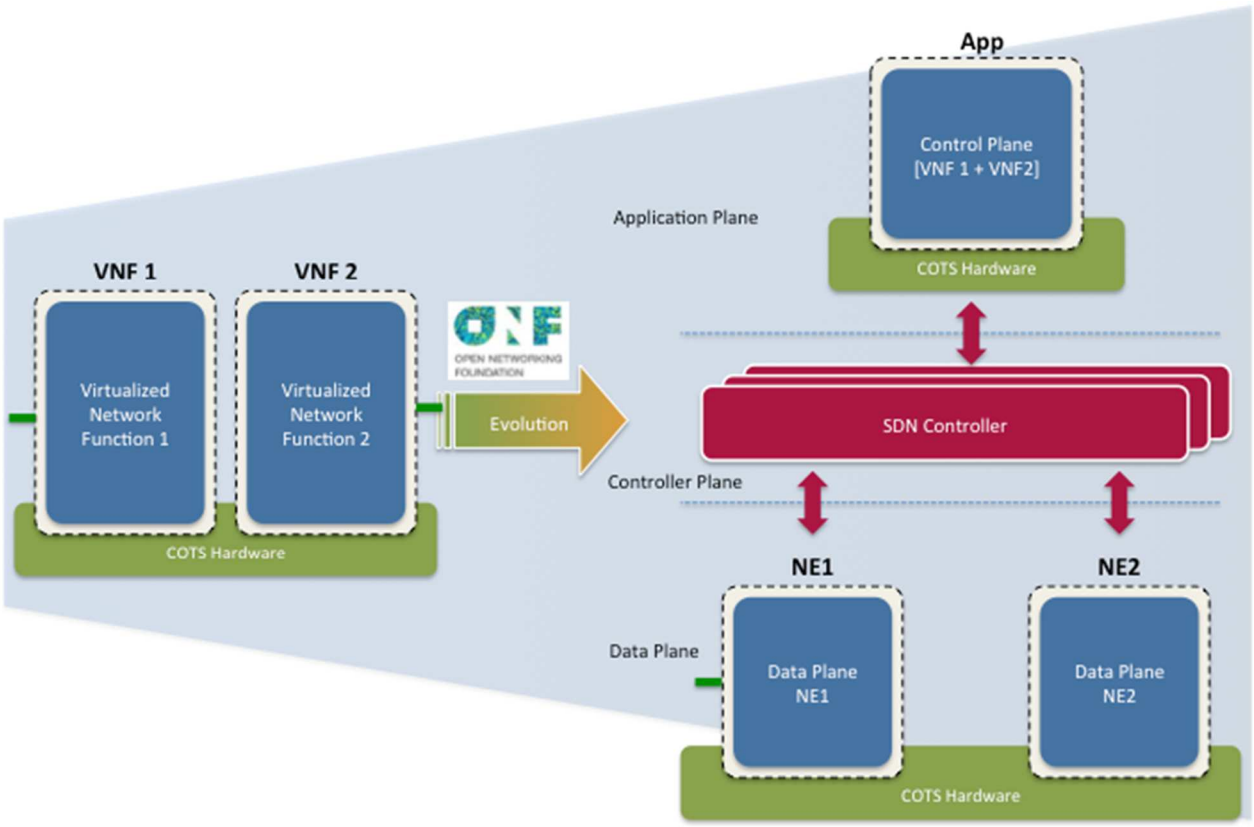


Figure 25: ONF SDN concept.

As shown in Figure 25 above already shows that, when we move to an SDN world, the transformation to the SDN paradigm has not tomay not be on a 1:1 basis. This means that a VNF may not move directly to the SDN paradigm by splitting itself into 3 parts. In fact, a single VNF can result into multiple applications and/or multiple NEs (N:N). To make this clear, an example of "SDNificiation" is provided below.

In this example (see Figure 26), the original scenario is an already set of virtualized routers. Each router has an IPv4 and IPv6 forwarding feature as well as the traditional routing protocols, like OSPF, BGP, etc (for simplicity, we can forget other features). On top of that, operators can configure the routers and build services like IPv4 connectivity, IPv6 connectivity or enterprise VPN, among others. Moving to the SDN paradigm, control and forwarding planes are decoupled. On the bottom, NEs are deployed with forwarding-only capabilities, applied according to the policies provided by the control plane. On the top is implemented the control logic, e.g. OSPF, BGP, as well as all the service logic that permits e.g. the provisioning of a new Access, VPN or VPN site.

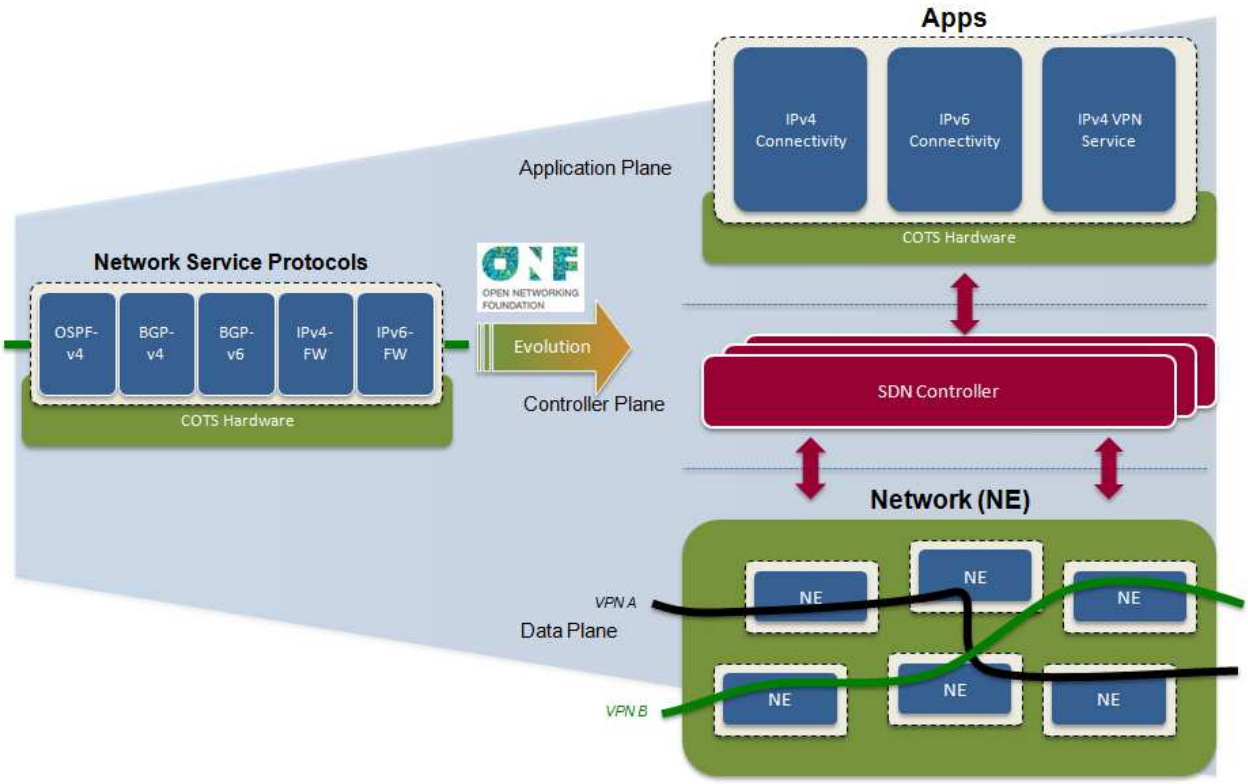


Figure 26: SDN: Example of Transport Network.

In this particular case, there is an N:N mapping between the control plane (SDN Apps) and the user-data plane (NEs); i.e. multiple applications implement different services on top of a common set of NEs. NEs can be virtual software switches (e.g. Open vSwitch) or more performing hardware specific switches.

6.3.3 Converged NFV+SDN architecture

The advent of SDN and NFV are seen as the critical architectural building blocks to give mobile operators the necessary tools necessary to address the massive data usage by their customers in the new of wave of expected 5G use cases. The software focus driving the confluence of NFV and SDN will allow mobile operators to be more responsive to customer demands either dynamically or through on-demand service provisioning via self-service portals.

NFV and SDN were developed by different standardization bodies; however, they are complementary technologies and as a result are often collectively referred as NFV/SDN. Although they have substantial value when exploited separated, in combined they offer significant additional value. The NFV technology brings the possibility of creating new network functions on-demand, placing them on the most suitable location and using the most appropriate amount of resources. However, this requires the SDN to be able to adjust the network accordingly, making network (re)configuration (i.e. programmability) and sequencing functions (chaining).

As NFV and SDN come from different SDOs (Standard Developing Organizations), none of them combines both architectures. Thus, in [16] there is an attempt to propose such combination, taking the ETSI NFV architecture as a starting point and introducing the SDN paradigm.

Starting from the ETSI NFV architecture depicted in Figure 24, we can see the VNFs (in the left part) which represent the virtualized network functions. According to the SDN model, shown in Figure 25, a monolithic VNF is separated into three parts.

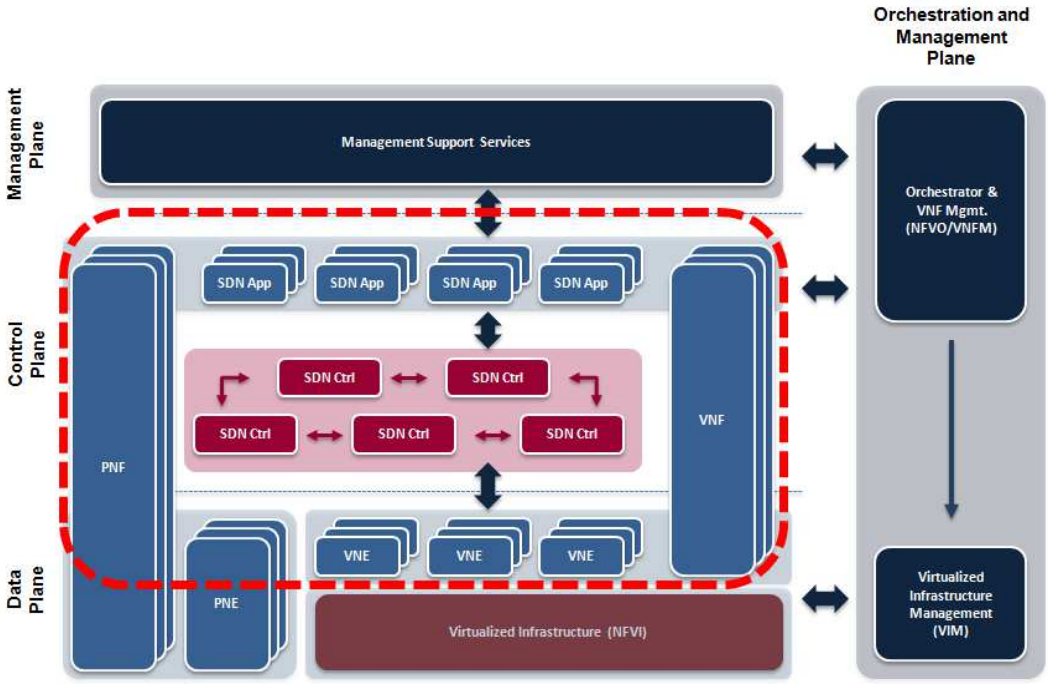


Figure 27: NFV and SDN combined architecture [16].

Note: It should be noted that the red dashed box in Figure 24 and Figure 27 identify the replacement derived from the SDN model.

The combined architecture is not very different from the one defined by the ETSI NFV with the exception of changes regarding the extra additional layers. The naming of the components is another issue that we needed to decide, since similar boxes have different names, depending on whether you take a look at them from the NFV or the SDN perspectives.

In order to consider "legacy" components (non-NFV, non-SDN), we kept the physical NFs in the leftmost side of the dashed red square of the architecture, meaning that we may have *Physical Network Functions* (PNF), which do not apply the NFV and the SDN models. The same way, we may have virtualized NFs (VNFs), but with no SDN capabilities. For those, we kept the *Virtual Network Functions* (VNF) naming, as shown in rightmost side of the dashed red square. In the middle, all components are SDN-aware, meaning that they are split into three layers. In the bottom layer (user-data plane), you may have physical or virtualized *Network Elements* (NE), which can be *Physical Network Elements* (PNEs) or *Virtual Network Elements* (VNEs), respectively. In this case, we opted by names coming from the SDN world, since they describe more clearly the roles they are performing. On the controller layer, we assumed that we may

have multiple controllers and at different levels, naming them all *SDN Controllers* (SDN Ctrl). Finally, for the control layer, we used the naming *SDN Application* (SDN App). In this case, we do not specify if it is virtual or not, but it can be both, although we believe that this layer will be mostly populated by virtual applications, considering that hardware specific utilization is declining.

6.3.4 3GPP C-RAN

The *Cloud Radio Access Network* (C-RAN) is a new mobile network architecture that allows operators to meet the needs of increasingly growing users demands and 5G requirements for achieving the performance needs of future applications. C-RAN intends to move part of the eNB to local clouds, taking advantage of the cloud technologies, elastic scale, fault tolerance, and automation. Using this architecture, a minimum set of hardware critical functions is kept on the physical *Remote Radio Head* (RRH) components, while the remaining virtualized (CPU intensive) part, the Base Band Unit (BBU), is deployed and concentrated in small local datacenters for a large number of cells. Figure 28 depicts the basic concept.

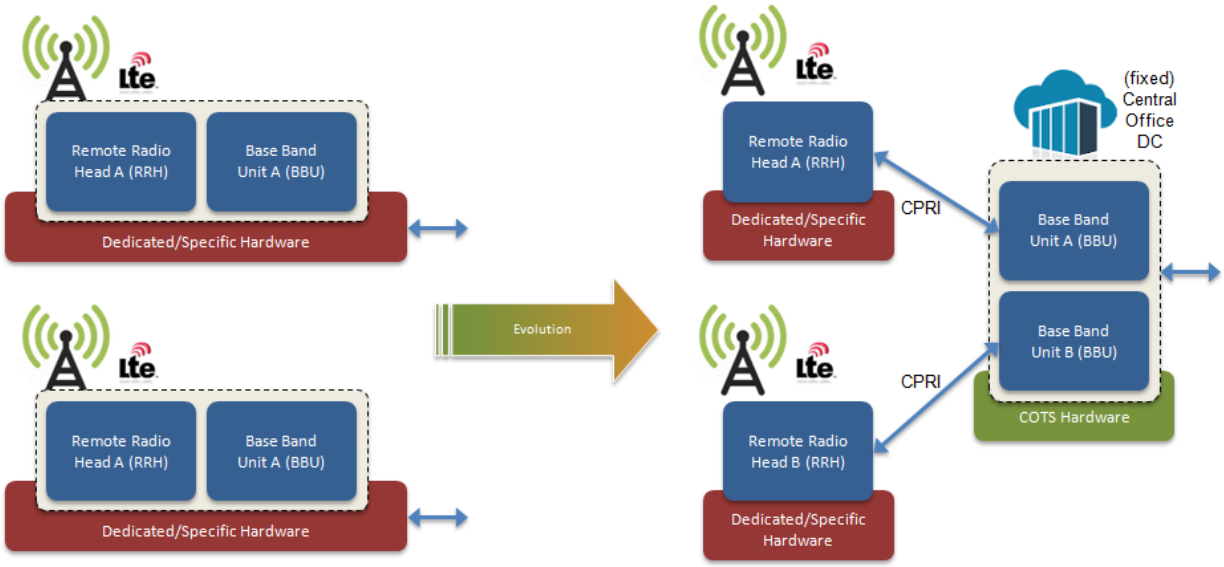


Figure 28: 3GPP C-RAN basic model.

This approach has multiple advantages: 1) It brings resource isolation for C-RAN functions that are now software defined and can be deployed and scale using Commodity hardware and off-the-shelf standards components; 2) it reduces the cost per deployed cell, which is an important aspect, as next mobile generations will reduce the cell coverage (and therefore increase the number of cells) as a way to increase bandwidth; 3) by centralizing BBUs in a local DC, the required amount of resources is lower than for traditional scenarios, as today cell-sites are provided with processing power for peak hours; 4) X2 interfaces, among eNBs, become internal links in top of the rack DC network, reducing latency times on mobility situations; 5) it leveraged

C-RAN SDN controllers [14] to improve network capacity and flexibility by providing dynamic control, fined grained resource allocation and better latency control to meet QoS requirements.

There are different C-RAN solutions, which propose different functional splits among the cell-site part (RRH) and the DC part (BBU). This has impacts on the interface between both and the amount of bandwidth required for that. For this reason, this issue is of extreme importance.

6.3.5 ETSI MEC architecture

Mobile Edge Computing (MEC) is a new technology platform, which is currently being standardized by an ETSI Industry Standard Group (SIG). MEC will offer application developers and content providers cloud computing and an IT service environment to deploy and run applications at the edge of a mobile network.

MEC is recognized by the 5G PPP as one of the key emerging technologies for 5G systems (along with NFV and SDN). The MEC environment will be characterized by:

- Proximity
- Ultra Low Latency
- High Bandwidth
- Real-time access to radio network information
- Location Awareness

MEC provides a set of services that can be consumed by applications running on the MEC platform. These services can expose information from such as real time network data e.g. radio conditions, network statistics etc. and the location of connected devices to consuming applications. Exposing these types of information can be used to create context for applications and services with the potential to enable a new ecosystem of use cases, such as dynamic location services, big data, multimedia services and many more, with a significantly improved user experience.

MEC can take advantage of NFV deployments, due to the complimentary use of cloud infrastructure, virtualization layers and management and orchestration components. Operators are increasing looking at virtualization of their mobile networks and, in particular, C-RAN. In this context, operators will have to create small DC infrastructures at the edge, in order to support RAN processing requirements. Having IT infrastructures on the edge, will enable operators to use unused resources to support 3rd parties to deploy applications in an edge computing environment, without having to invest on additional resources specially for that purpose. As a consequence, edge computing becomes cheaper, making it more appealing to network operators, content providers and developers.

The same concept can be applied to fixed networks, where virtualization is being adopted. In fact, for network providers with mobile and fixed operations and traditional central offices can

be used to concentrate both mobile and fixed access, making those places good candidates to provide edge computing applications for all customers. Figure 29 outlines this vision.

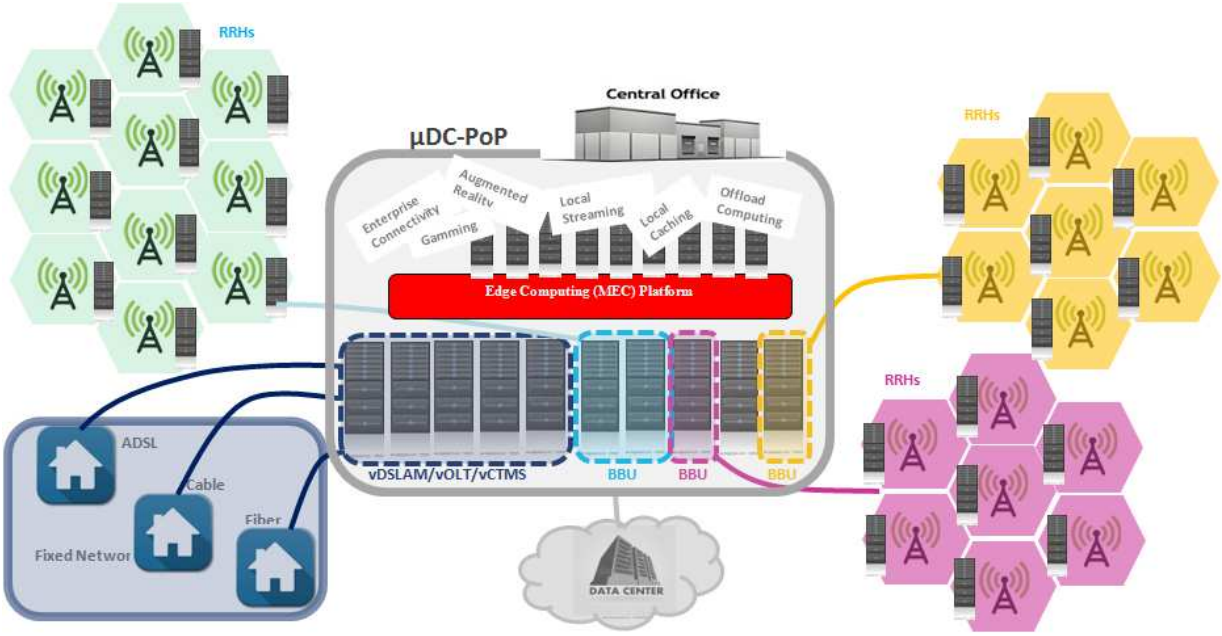


Figure 29: MEC mobile/fixed vision.

The edge computing architecture includes the MEC Platform as the central point. This platform allows 3rd party applications to register and get authenticated/authorized, in order to access a set of services, available via the respective APIs Using the APIs, applications can request the platform to make traffic offload (TOF) to serve customers from the edge, or access to other information provided by the operator.

The NFVI and VIM defined by the NFV are essentially reused to provide a virtual environment this time for the applications. A similar thing happens to the management and orchestration layer, which can be used to manage the applications lifecycle and orchestrate them according to the best interests of end users and application providers. In the management layer, an additional manager can be found, which is dedicated to managing the MEC platform itself, including services and respective APIs. The Figure 30 shows the proposed architecture for MEC.

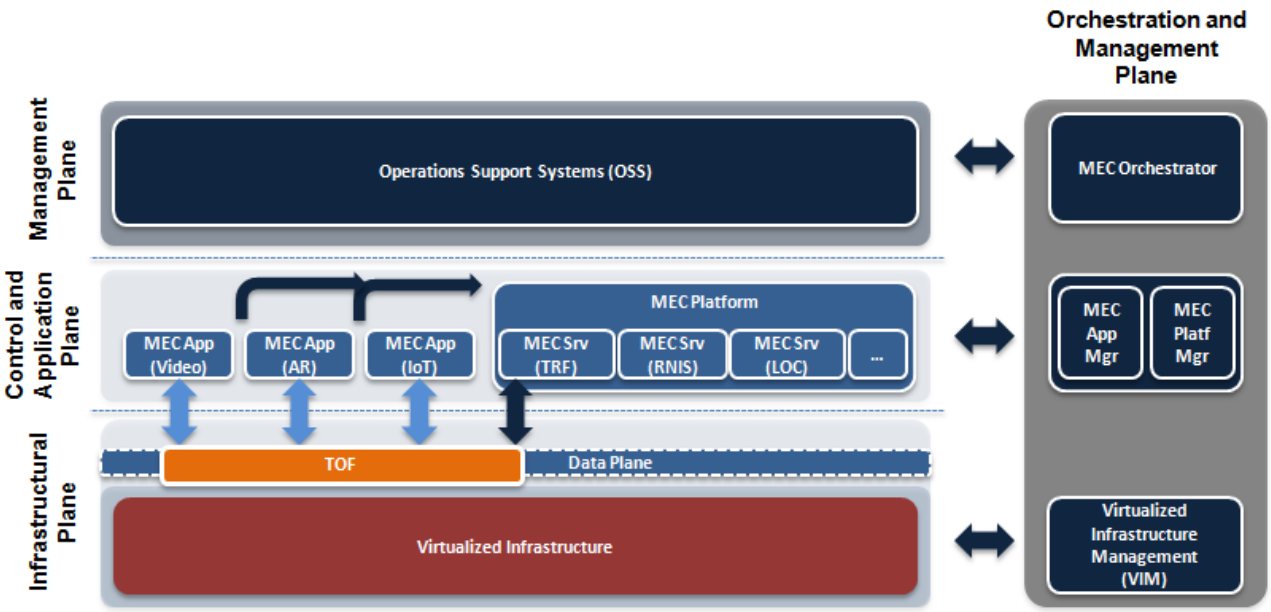


Figure 30: MEC architecture.

It is clear the MEC will be key architecture component of 5G in the advent of mobile edge cloud computing. It is also complementary to the key 5G technology building blocks of SDN and NFV through a programmable paradigm. The MEC paradigm will important element of 5G as it plays a pivotal role in offering increased monetization scope for service providers helping to transform mobile networks.

6.4 Cross domain architecture

6.4.1 Mobile Core

The Mobile Core is the part of a mobile network, which aggregates the traffic and signalling coming from the eNBs. It is composed by functional blocks dealing with the control and data plane functions. In LTE networks, the Mobile Core is called *Evolved Packet Core* (EPC) and has three main functional entities: MME, S-GW and P-GW; additionally, there is a repository: HSS. The MME is a control plane function, while the S-GW and P-GW, although essentially executing data plane functions, they also play some signalling roles. The expected evolution of the Mobile Core is to follow the SDN concept of having a clear split between control and data planes.

Figure 31: depicts this model, which is under study on 3GPP in [3GPP-23.714].

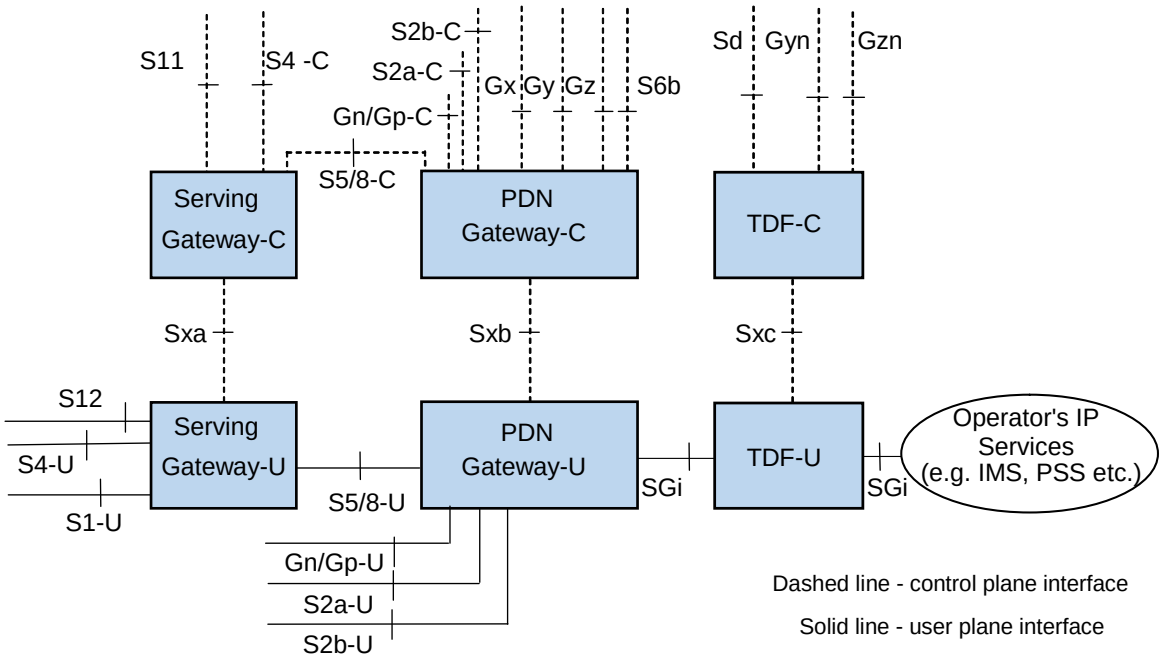


Figure 31: 3GPP Mobile Core [3GPP-23.714].

Although having this separation, all currently existing interfaces remain untouched. New interfaces will be created between the separated control plane and data plane functions. Traditional protocols like Diameter can be used for that, or other more SDN-like may be introduced like Openflow. Another trend could also lead 3GPP to evolve the architecture by merging some of the control functions into a single functional entity; the same may happen for the data plane entities.

6.4.2 Central-DC

Central DCs are big central infrastructures where most public cloud services are located. Comprising a huge amount of resources (CPU, Storage, Networking, etc.), they are extremely efficient since they benefit from scale factors, reducing costs in aspects such as space, cooling, operations, etc.

Central DCs hold today the majority of cloud applications. With the NFV and SDN advent, many DC networking components, such as switches, routers, firewalls or load balancers, can be already virtualized and controller by SDN Controllers.

The caveats for large central DCs are the distance to the end users, which increases the latency, having significant impacts on some applications, and waste of bandwidth along the path. On the other hand, the deployment of bandwidth-intensive applications on such environments may cause bottlenecks on DCs itself and in general on the network providers.

6.4.3 Virtualization and Cloud Computing

The key principle that has enabled the resource and service abstraction from the underlying physical resources in virtualization. Cloud computing has allowed for data center owners to virtualize workloads in virtual machines and then benefit from the consolidation of workloads onto fewer physical machines. The manageability of workloads that are abstracted from the physical infrastructure gave rise to many beneficial features such as high-availability, scale-up and scale-out workloads. Once abstracted sufficiently from the hardware, the workloads could be moved between physical machines to ensure high-availability services. Virtual machines have enabled the first stage of consolidation but there are now many approaches looking into making the Virtual Machine overhead smaller, thereby making it possible to have more workloads on a single machine and to remove the inefficiencies of having Operating Systems that are over-provisioned for the types of workloads that are being run. Containers and chroot jails are not new but there is a lot of interest and work going into minimizing the footprint of virtual workloads. Unikernels are light-weight processes that are designed and compiled for specific purposes, thereby further removing memory management sub-systems and other features that a multi-tasking operating system would normally need to handle.

These light-weight virtual platforms allow for more efficient processing of the workloads. ClickOS [19] is a specialized MiniOS (lightweight Xen VM) instance that is optimized for high packet throughput.

7 REFERENCES

[1] Malik Kamal Saadi, 5G: From Vision to Framework, ABI Research, 2015.

[2] ITU-R M.2083-0: IMT Vision - Framework and overall objectives of the future development of IMT for 2020 and beyond. International Telecommunications Union – Radiocommunication Sector. September 2015.

[3] ETSI ISG NFV: Network Functions Virtualisation (NFV); Architectural Framework. ETSI GS NFV 002 V1.2.1. December 2014.

[4] ETSI ISG NFV: Network Functions Virtualisation (NFV): Virtual Network Functions Architecture. ETSI GS NFV-SWA 001 V1.1.1. December 2014.

[5] ETSI ISG NFV: Network Functions Virtualisation (NFV): Management and Orchestration. ETSI GS NFV-MAN 001 V1.1.1. December 2014.

[6] ETSI ISG NFV: Network Functions Virtualisation (NFV): NFV Performance & Portability Best Practices. ETSI GS NFV-PER 001 V1.1.2. December 2014.

[7] Cisco ASA 5510 Adaptive Security Appliance http://www.cisco.com/c/en/us/support/security/asa-5510-adaptive-security-appliance/model.html

[8] 5G NORMA D3.1: Functional Network Architecture and Security Requirements v1.0. December 31, 2015.

[9] Mijumbi, R. et al.: Network Function Virtualization: State-of-the-art and Research Challenges. IEEE Communications Surveys & Tutorials. DOI: 10.1109/COMST.2015.2477041. Publication pending.

[10] 3GPP TR 22.852, Study on Radio Access Network (RAN) Sharing enhancements, Rel.13, Sep. 2014

[11] 3GPP TS 23.251, Network Sharing; Architecture and Functional Description, Rel.12, Jun 2014.

[12] 3GPP TR 32.842, Telecommunication management; Study on network management of virtualized networks, Rel. 13, Dec 2015

[13] 3GPP TS 28.500, Telecommunication management; Concept, architecture and requirements for mobile networks that include virtualized network functions, Rel. 14, Dec 2015

[14] A. Gopalasingham, L. Roullet, Nessrine Trabelsi, C. Shue Chen, A. Hebbar, et al.. Generalized Software Defined Network Platform for Radio Access Networks. IEEE Consumer Communications and Networking Conference (CCNC), Jan 2016, Las Vegas, United States. 2016.

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, M. F. Kaashoek. “The Click modular router”, ACM Transactions on Computer Systems (TOCS) 18, no. 3 (2000): 263-297

[16] P. Neves, et al., “The SELFNET Approach for Autonomic Management in an NFV/SDN Networking Paradigm”, International Journal of Distributed Sensor Networks, Vol. 2016, 2016.

[17] N. Fernando, S.W. Loke, and W. Rahayu, "Mobile cloud computing: A survey." Future Generation Computer Systems vol. 29, no.1 pp. 84-106, 2013.

[18] G. Bianchi, et al., “OpenState: programming platform-independent stateful openflow applications inside the switch,” ACM SIGCOMM Computer Communication Review, Vol. 44, N. 2, pp. 44-5, 2014

[19] J. Martins et al, “ClickOS and the Art of Network Function Virtualization”, USENIX NSDI ’14, Seattle, USA, 2014.

[20] https://www.ietf.org/proceedings/95/slides/slides-95-nfvrg-2.pdf
[21] https://www.ericsson.com/research-blog/sdn/unikernels-meet-nfv/
[22] P. Aranda, “High-level VNF Descriptors using NEMO”, draft-aranda-nfvrg-recursive-vnf-00, IETF draft, work in progress
[23] P. Eardley (ed.), “Use cases, technical and business requirements”, Deliverable D2.1 of Superfluidity project
[24] H. Daniel, L. Peer (eds.) “Initial design for control framework”, Internal Deliverable I6.1 of Superfluidity project
[25] B. Sayadi (ed.) “Functional Analysis and Decomposition”, Deliverable D2.2 of Superfluidity project