SUPERFLUIDITY

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

Research and Innovation Action GA 671566

DELIVERABLE D2.2:
FUNCTIONAL ANALYSIS AND DECOMPOSITION

Deliverable Type:	Report
Dissemination Level:	Public
Contractual Date of Delivery to the EU:	01/06/2016
Actual Date of Delivery to the EU:	01/06/2016
Work package Contributing to the Deliverable:	WP2

Editor(s): Bessem Sayadi (Nokia FR)

Author(s): Bessem Sayadi (NOKIA-FR), Lionel Natarianni (NOKIA-FR), Erez Biton: (NOKIA-IL), Omer Gurewitz (ALU-IL), Giuseppe Bianchi (CNIT), Nicola Blefari-Melazzi (CNIT), Stefano Salsano (CNIT), George Tsolis (CITRIX), Francisco Fontes (ALB), Carlos Parada (ALB), Pedro A. Aranda (Telefónica, I+D), Ignacio Berberana (Telefónica, I+D), Costin Raiciu (UPB), Dirk Griffioen (Unified Streaming), Philip Eardley (British Telecom), John Thomson (OnApp), Julian Chesterfield (OnApp), Michael J. McGrath (Intel), Pedro de la Cruz Ramos (Telcaria Ideas S.L.), Juan Manuel Sánchez Mateo (Telcaria Ideas S.L.), Raúl Álvarez Pinilla (Telcaria Ideas S.L.), Matei Popovici (UPB), Costin Raiciu (UPB).

Internal Reviewer(s) Stefano Salsano (CNIT)

Abstract: This document presents an initial assessment of the Superfluidity reusable components in different domains, including the NFV, the Cloud RAN and mobile edge computing domains. We proceed to decompose existing monolithic network functionality into reusable components. The principle of Reusable Functional Block is introduced. Example applications of this concept are concept are presented in heterogeneous domains: Radio Access Networks, Mobile Edge Computing, fixed networking equipment, packet processing state machines. A set of use cases are introduced at the end of this document, presenting examples of usability of the RFBs.

Keyword List: Superfluidity, Reusable Functional Block, Decomposition, C-RAN, MEC, State Machine

VERSION CONTROL TABLE			
VERSION N.	PURPOSE/CHANGES	AUTHOR	DATE
1	First release	Superfluidity project	31/05/2016
1.1	Revision	Stefano Salsano	01/06/2016
1.2	Final Revision	Nicola Blefari Melazzi	01/06/2016

Executive Summary

To instantiate services on-the-fly, run them anywhere in the network (core, aggregation, edge) and shift them transparently to different locations, Superfluidity introduced the concept of Reusable Functional Block (RFB). This capability is a key part of the converged cloud-based 5G future - it will enable innovative use cases in the mobile edge, empower new business models and allow almost instant roll-out of new services, thus reducing investment and operational costs. In this document, we present the work conducted on the concept of RFB and reports the first results. The goal is not to build a definitive and exhaustive list of reusable components that should be supported, but rather to start understanding the benefits of the concept and identify their most important applications.

Superfluidity proposes to decompose the architecture into elementary radio and network processing primitives and events, which can then be exploited as basic modules of more comprehensive (and traditionally monolithic) network functions and services. The decomposition of monolithic functions into RFBs permits a flexible placement, as well as the incorporation of adequate virtualization techniques like Containers (e.g. Docker), UniKernels, and full Virtual Machines, by considering real-time constraints.

It is not possible here to perform the functional decomposition in an exhaustive way, covering all the envisaged 5G application areas. In this deliverable, the project has focused on a set of different contexts: Cloud Radio Access Networks, Mobile Edge Computing platform, generic NFV environments, fixed networking equipment and packet processing state machines. In each of this context a (non-exhaustive) functional analysis and decomposition into RFBs has been performed.

Finally, a subset of the use cases described in D2.1 has been revisited, having in mind the identification of the needed RFBs for their implementation.

INDEX

List of Figures

List of Tables

List of Abbreviations

ABR	Adaptive Bit Rate
API	Application Program Interface
AR	Augmented Reality
CDN	Content Distribution Network
CRUD	Create , Read , Update , Delete
DDoS / DoS	(Distributed) Denial of Service
DPI	Deep Packet Inspection
DRM	Digital rights management
ETSI ISG	European Telecommunications Standards Institute Industry Specification Group
GGSN / P-GW	Gateway GPRS Support Node / Packet Gateway
HARQ	Hybrid Automatic Repeat Request
IDPS	Intrusion Detection and Protection System
IETF	Internet Engineering Task Force
IoT	Internet of Things
ISA	Instruction Set Architecture
KPI	Key Performance Indicator
LCM	Life Cycle Management
LIPA-SIPTO	Local IP Access and Selected IP Traffic Offload
LTE	Long Term Evolution
LTM	Late TransMuxing
M2M	Machine to Machine
MEC	Mobile Edge Cloud
MIMO	Multiple-Input Multiple-Output
NFV	Network Function Virtualization
NFVI	NFV Infrastructure
NDP	Neighbour Discovery Protocol
NIC	Network Interface Card

NS	Network Service
ONF	Open Networking Foundation
OSS	Operational Support Systems
OTT	Over-the-top
PDP/PDN	Packet Data Protocol/Packet Data Network
PoC	Proof of Concept
PNF	Physical Network Function
QoE	Quality of Experience
QoS	Quality of Service
RAN	Radio Access Network
RFB	Reusable Functional Block
RNC	Radio Network Controller
RRH	Remote Radio Head
RRM	Radio Resource Management
SDN	Software Defined Networking
SDO	Standard Defining Organization
SFC	Service Function Chaining
S/Gi	Reference point defined by 3GPP between the mobile packet core and PDN (Gi is between GGSN and PDN; SGi is between P-GW and PDN)
SLA	Service Level Agreement
SON	Self-Organising Network
TCP	Transmission Control Protocol
UE	User Equipment
UMTS	Universal Mobile Telecommunications System
vCS	virtual Convergent Services
vHGW	virtual Home GateWay
VM	Virtual Machine
VFN	Virtual Network Function
VNFC	Virtual Network Function Component

VNFM	Virtual Network Function Manager

1 Introduction

1.1 Objective of the document

Network Function Virtualization (NFV) is emerging as a new architectural concept for the design and implementation of communication networks. Running network functions in software is a departure from the current monolithic approach to building and deploying hardware appliances. Such appliances are difficult to customise, upgrade and scale and lead to vendor lock-in. The main benefits of running network functions completely in software are envisioned to be:

1. Ease of developing new functionality
2. Seamless scale-out by adding more resources and load balancing across them.
3. Easy upgrade and more agile bug fixing.

Despite being a relatively recent proposal, the specification / standardization of the NFV paradigm has already achieved significant results ([1] to [4]) and there are already implemented solutions. The current architecture is built around the concept of VNFs (Virtual Network Functions) that can be composed to provide Network Services. The VNFs are instantiated and executed over a hosting environment denoted as VNF Infrastructure (VNFI).

In order to fully exploit the NFV potential, re-thinking the way of designing and modelling the telecommunication services is needed. Before NFV, the focus of modelling was on interfaces between physical boxes, with the identification of reference points and of the protocols exchanged across the reference points. The physical boxes were closed; hence, there was no need to specify their internal structure. With the advent of virtualization technologies and NFV, services can be realized by combining functional blocks that are much like software components. Such components can be deployed and executed over a distributed computing infrastructure, composed of a set of big Datacentres and a very large number of distributed resources closer to the access networks (this is referred to as *Fog computing* [10] or *Mobile Edge Computing* [11]). In the 5G context, this scenario is extended to the Radio Access infrastructure, which turns in a *Cloud-RAN* [12].

An optimal allocation of processing components in this highly distributed cloud environment is the key to optimize performances, reduce costs (operational costs and/or equipment buying costs) and achieve higher efficiency. Hence, the desire to decompose functions up to a very high granularity and to extend this approach in a unified manner to Radio Access functionality and to Data Plane "microscopic" forwarding operations. From an ideal perspective, it should be possible to decompose a service in an arbitrary way (from the point of view of the needed resources) and map it in the most convenient way into the set of resource providers (e.g. processing hosts) offered by the infrastructure.

An open issue is the definition (and standardization) of a language (or a set of languages) that allow expressing the composition of processing entities at different levels, from "high level" VNFs to Data

Plane "microscopic" forwarding operation and to Radio Access functionality. The heterogeneity of the platforms at the different levels (from x86 processors based on Unix-like Operating System, to specialized boards based on TCAMs and NetFPGAs) makes it difficult to define a consistent unified framework. A fundamental step in this direction is the definition of a set of reusable components that are common across many network-processing functions. These reusable components will serve two main purposes:

- They will allow more complex functionality to be built by combining small, purpose-specific building blocks
- They will allow each building block to be implemented in a variety of ways, including generic software running on x86/ARM, NetFPGA, specialised ASICs, TCAMs, and so forth.

The element that combines these reusable components is the network programming language. As in general purpose programming languages, it is likely there will be a wealth of languages defined and used. Existing language examples include the configurations used by the Click Modular Router [13] targeted at software routers functionality, and GNU Radio [14] targeted at software radio processing.

The reusable components will act as the API offered by the network OS to network applications. As in all APIs, there is great freedom in the implementation, as long as the API is observed – and this will allow software/hardware vendors to compete in offering implementations of the same functionality and network operators to decide which solutions it wants to deploy based on requirements, derived from its own operations, or taken from its customers.

This document offers an initial assessment of the reusable components in different domains of the networking landscape, including the core network, the Cloud RAN and mobile edge computing. We proceed to decompose existing monolithic network functionality into reusable components, and report our key observations to date. Our goal is not to build a definitive list of reusable components that should be standardised, but rather to start understanding what are the useful functionalities that might be the candidates for standard reusable components.

A fundamental question when decomposing monolithic network functions is to select the right level of granularity. There is a direct trade-off between reusability and performance. On the one hand, fine-grained functions may be easier to reutilise to build complex applications, but have higher instantiation costs and per-flow or global state management across components will become complex and create further overheads. Coarse-grained functions are more efficient, however they will be difficult to reuse. This work does not seek to dictate the appropriate granularity for reusable components; instead, it decomposes monolithic functions using the *appropriate* building blocks for each part of the 5G network, as determined by experts in those areas. The outcome is informative rather than normative, and it provides a concrete starting point for the 5G architecture.

Orthogonal to the granularity of the reusable components, the process of building complex functionality depends on ability of all the involved parties' to ***understand*** what each reusable component is supposed to do: what its allowed inputs are and how the outputs are obtained from the inputs.

1.2 Structure of the document

Section 1 introduces the objective of this document (section 1.1) and its structure (section1.2). Section 2 summarizes relevant state of the art, namely ETSI NFV, where the principle of decomposition of a network service into a set of virtual network function is introduced, Small Cell Forum where a first functional RAN split is given, and two European projects METIS and iJOIN, where different functional split for RAN, including additional parameters such us backhaul is proposed.

In order to understand the decomposition approach that will be presented in Section 4, Section 3 introduces the Superfluidity innovation and architectural aspects, representing a joint work with WP3. In particular, Section 3.1 summarizes the innovations and research challenges of Superfluidity. Section 3.2 presents the status of the Superfluidity architecture design. Section 3.3 introduces the concept of Reusable Functional Block. A proposal for an RFB Composition Language is given in section 3.3.1.

Section 4 starts by introducing a methodology that can be followed to describe the different RFBs. A template is proposed, including the description of the interfaces across different RFBs, their timing requirements and their environment execution requirement (Hardware, need for acceleration or not...). In sections 4.1, 4.2 and 4.3, we proceed to decompose existing monolithic network functionality, related respectively to Cloud-RAN, Mobile Edge Computing and generic NFV environments, into reusable components, and report our key observations to date. Our goal is not to build a definitive list of reusable components that should be standardised, but rather to start understanding what are the useful functionalities that might be considered for each component. In section 4.4 we present the example decomposition of a fixed network equipment (in particular a security appliance) into RFBs. In section 4.5, to emphasize the RFB concept and go deeply into the decomposition principle, a nano-decomposition based on packet processing state machines is introduced.

In Section 5, several RFB chains are considered through example use cases. A detailed description of the considered use cases is available in our Deliverable D2.1.

Section 6 concludes this report.

2 State of the Art

Network Function Virtualization (NFV) is emerging as a new architectural concept for the design and implementation of communication networks. There have been in recent times some European projects and standardization forums focusing on addressing this solution for the upcoming 5G challenges.

2.1 Standardization fora

In this section, we focus on some standardization forums related to the Superfluidity scope. One of them is the ETSI Network Functions Virtualisation Industry Specification Group, which works in the definition of a consistent approach and common architecture for the hardware and software infrastructure needed to support virtualised network functions. The second one, the Small Cell Forum, supports the wide-scale adoption of the small cells deployment. The third one is ONF (Open Networking Foundation), the body that is standardising the OpenFlow protocol for Software Defined Networks.

2.1.1 ETSI NFV

The NFV architectural model is based on the concept of Virtualised Network Functions, which envisages the implementation of network functions as software-only entities. The NFV standardisation in ETSI has considered the use cases for NFV in [1] and the architectural aspects in [2][3]. In particular, the overall architectural framework for NFV is described in [2], while [3] provides further details on the architecture of VNFs. The environment that hosts the VNFs is called NFV Infrastructure (NFVI). The orchestration aspects, i.e. how VNFs can be chained and how their life cycle can be managed, are dealt with in [4] and are commonly referred to as NFV Management and Orchestration, in short MANO. It is interesting to note that moving from the overall framework definition ([2]) to the VNF architecture ([3]) and finally to the management and orchestration aspects ([4]), the level of description changes from a general one to a concrete and detailed one. The last one includes a set of restrictions in its models so that the specification is much closer to the implementation. Figure 1 shows, at a high level of abstraction, the whole architecture proposed by NFV ETSI Industry Specification Group.

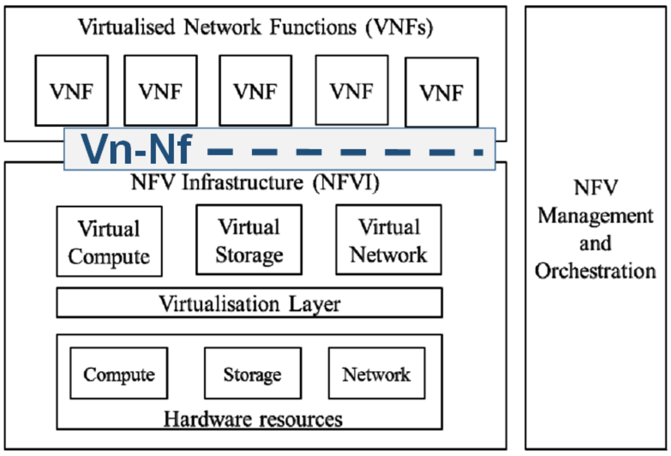


Figure 1: High level NFV framework

The interface or *reference point* between VNFs and the NFV Infrastructure is called Vn-Nf. The Vn-Nf is a special kind of reference point. In fact, a typical interface or reference points describes the information exchanged between two functional entities and may provide details of the related protocols. On the other hand, the Vn-Nf needs to describe how the NFV Infrastructure can host a VNF, which resources it can provide and several other non-functional characteristics. In short, the Vn-Nf interface needs to describe the execution environment offered by the NFVI to a VNF.

VNFs can be chained with other network functions, both VNFs and Physical Network Functions (PNFs), to realize a Network Service (NS) that achieves the desired overall functionality or service that the network is designed to provide. Most current network services are defined by statically combining network functions. This combination can be expressed using an *NF Forwarding Graph*, which focuses on the relations that express connectivity between network functions. This allows different VNFs to be deployed over the virtualised infrastructure to support End-to-End (E2E) network services and be applicable to diverse use cases and operator network scenarios with minimal integration effort and maximum reuse. Figure 2 illustrates the representation of an end-to-end network service that includes a nested NF Forwarding Graph.

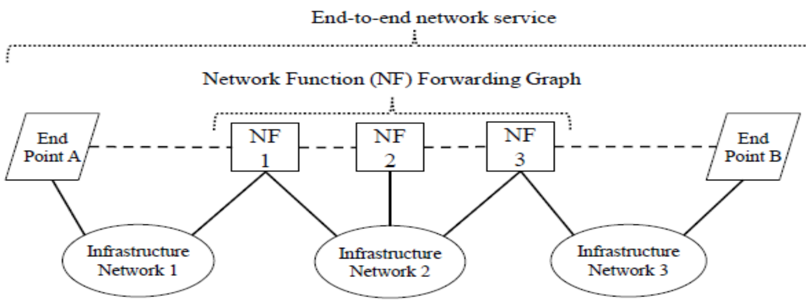


Figure 2: Graph representation of an end-to-end network service

According to [3], VNFs can be decomposed into VNF Components (VNFCs), which are software modules defined by the VNF Provider in order to structure a VNF according to many factors, e.g. the prioritization of performance, scalability, reliability, security and other non-functional goals, the integration of components from other VNF Providers, operational considerations, the existing code base, etc. VNFs are implemented with one or more VNFCs and it is assumed that VNFC Instances map 1:1 to the NFVI Vn-Nf interface. The assumption is that the NFV Infrastructure offers a single type of support for running VNFs, which is a Virtualised Container. Figure 3 shows an evolving view of the NFV architecture framework, clarifying the situation when two or more VNFCs are instantiated.

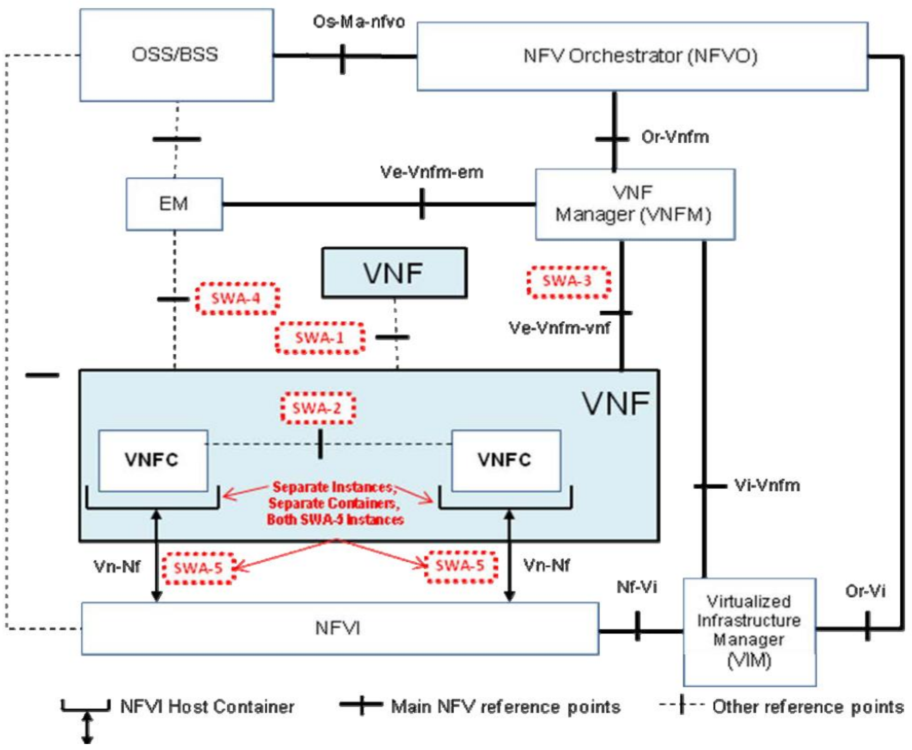


Figure 3: VNF interfaces and NFV reference points

The main contribution of MANO part is the definition of *descriptors* that can be used to characterize Network Services, VNFs, VNF Forwarding Graphs, Virtual Links, and Physical Network Functions in a consistent model. A MANO compliant orchestrator implementation takes these descriptors in input and it is able to deploy the VNFs and the Network Services over the NFV Infrastructure. An open source implementation of MANO, called OpenMANO, provides a nice explanation and some examples of the most important descriptors in [5]. Recently, ETSI has formed a group called Open Source Mano (OSM) whose goal is to deliver an open source MANO stack using accepted open source tools and working procedures.

The conceptualization of VNF and of VNFC defined in current architecture of ETSI NFV can be generalized into the RFB concept of the Superfluidity architecture. RFBs are envisaged as a way to decompose high-level monolithic functions into reusable components. They facilitate the Superfluidity goal to allocate resources (processing, networking, storage…) dynamically and efficiently. An RFB can be executed on what we call “hosting environment”. For example, using the ETSI terminology, VNFC entities can be executed on a so-called “Virtualization Container”, which constitutes its hosting environment. In the current model, VNFs and VNFCs correspond to Virtual Machines running in hypervisors. The Superfluidity architecture extends this state of the art in two directions. On one hand, it includes the concept of very lightweight Virtual Machines supporting RFBs with fine granularity. In this case, the hosting environment is a hypervisor specialized in supporting tiny Virtual Machines. On the other hand, the Superfluidity architecture will support heterogeneous hosting environments that can be used to support the RFBs. Examples of such environments are the like the click modular router, or extended finite state machines (XFSM) based on OpenFlow, or modular Software Radio processing platforms.

2.1.2 Small Cell Forum

Small Cell Forum supports the wide-scale adoption of small cells and the delivery of integrated Heterogeneous Networks (HetNets). Its mission is to accelerate small cell deployment to change the shape of mobile networks and maximise the potential of the mobile internet. The term *small cells* refers to low-powered radio access nodes that operate in licensed and unlicensed spectrum and typically have a range of 10 metres to several hundred metres. These contrast with a typical mobile macrocell that might have a range of up to several tens of kilometres. The term covers femtocells, picocells, microcells, metrocells and public Wi-Fi.

It works to remove barriers, drive standards, ensure interoperability and support the deployment of small cells worldwide. Currently their members are driving solutions that include small cell/Wi-Fi integration, SON (Self Organizing Network) evolution, virtualisation of the small cell layer, driving mass adoption via multi-operator neutral host, ensuring a common approach to service APIs to drive commercialisation and the integration of small cells into 5G standards evolution.

SCF splits small cells into two components, a central small cell where functions are virtualized, and a remote small cell with non-virtualized functions [26]. The central small cell will serve multiple remote small cells. The small cell layers and functions are investigated with a top-down approach where gradually more functions are moved from the remote small cell to the central small cell. These small cell split points progressively result in: Centralized services, Centralized RRC, Centralized PDCP, Centralized RLC, Centralized MAC and Centralized PHY. A summary of the split architectures for the use cases is given in Figure 4. Functions to the left of the split are virtualized in the central small cell (VNF), while functions to the right reside in the remote small cell (PNF).

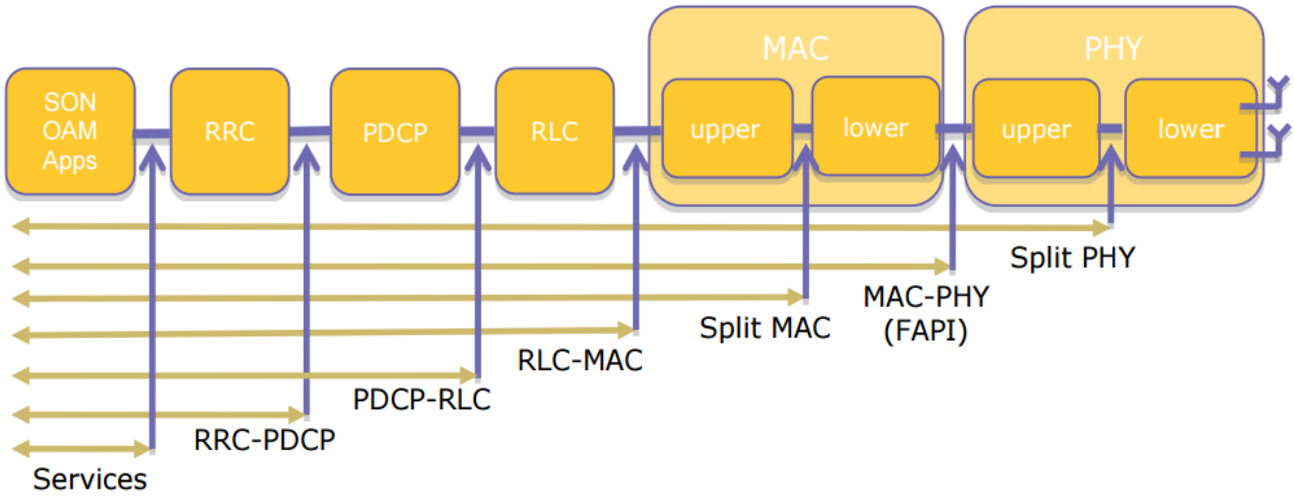


Figure 4: Functional splits proposed for Small Cells (source extracted from Small Cell Forum)

2.1.3 ONF's OpenFlow approach to decomposing switching functionality

In this section, we discuss how the networking community has worked in the direction to decompose the switching functionality in order to offer programmability of the network elements. The OpenFlow protocol is under standardization by the Open Networking Foundation (ONF). In OpenFlow, some level of switch programmability is accomplished by providing the operator with the ability to associate a set of (pre-implemented) very elementary and stateless actions (forward a packet, drop a packet, rewrite a header field, encapsulate the packet, etc.) to the outcome of a matching process which broadly identifies the flow. We refer to this as OpenFlow "*match/action*" abstraction. The most interesting (and winning!) feature of this abstraction is its independence on the underlying platform: as long as two switches support the same header matching facilities and the same set of well-defined OpenFlow actions, an OpenFlow configuration can be ported from a switch to another, irrespective on how the matching process and the set of actions is implemented (be it in HW or in SW).

However, OpenFlow was designed with the desire for rapid adoption [23], i.e., as a pragmatic attempt to address the dichotomy between i) flexibility and ability to support a broad range of innovation, and ii) compatibility with commodity hardware and vendors' need for closed platforms. Ultimately, OpenFlow provided "only" an abstracted model for a flow table, insufficient to permit "true" network programmability and stateful operation.

The aftermath is that most of the network programming frameworks proposed so far in the SDN arena circumvent OpenFlow 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. Getting back to the previous load balancer example, an OpenFlow-based implementation of a load balancer requires to involve an external controller in the load balancing decision.

2.2 Related work in progress

In this section, we focus on two European projects whose work is related to Superfluidity's objectives. Both projects METIS and iJOIN address the problems expected for future mobile technologies from the perspective of wireless communication, RAN and backhaul network.

2.2.1 EU Metis

METIS is among the largest coordinated efforts worldwide dedicated to 5G wireless technologies. Its main objective is to generate European consensus on the future global mobile and wireless communication systems. METIS also aims to provide an overall system concept as a technical objective.

The essence of the system concept developed to address the 5G challenges consists in three generic services: extreme Mobile BroadBand (xMBB), ultra-reliable Machine-Type Communication (uMTC) and massive MTC (mMTC); and four main enablers: lean system control plane, dynamic RAN, localized traffic flows and a spectrum toolbox. xMBB provides increased data rates, as well as improved QoE for moderated rates. mMTC allows connectivity for a large number of devices with energy and cost restrictions. uMTC provides reliable communication to time-critical services and applications. A lean system control plane provides control information necessary to guarantee latency and reliability, dynamic RAN allows the rapid deployment of nomadic access points, localized content flows reduce the load of the backhaul and finally, the spectrum toolbox is a set of tools that allow 5G systems to operate with great spectrum flexibility.

The functional architecture of METIS is explained in [27]. Four high-level building blocks have been identified: Central Management Entities (they cover network overarching functionalities), Radio Node Management (the BBs that provide radio functionalities that affect more than one node), Air Interface (per node radio functionality) and Reliable Service Composition (self-explanatory). The first three high-level BBs contains a number of smaller building blocks Building Blocks that can be Common or Horizontal Topic Specific depending on its functionality as seen in Figure 5.

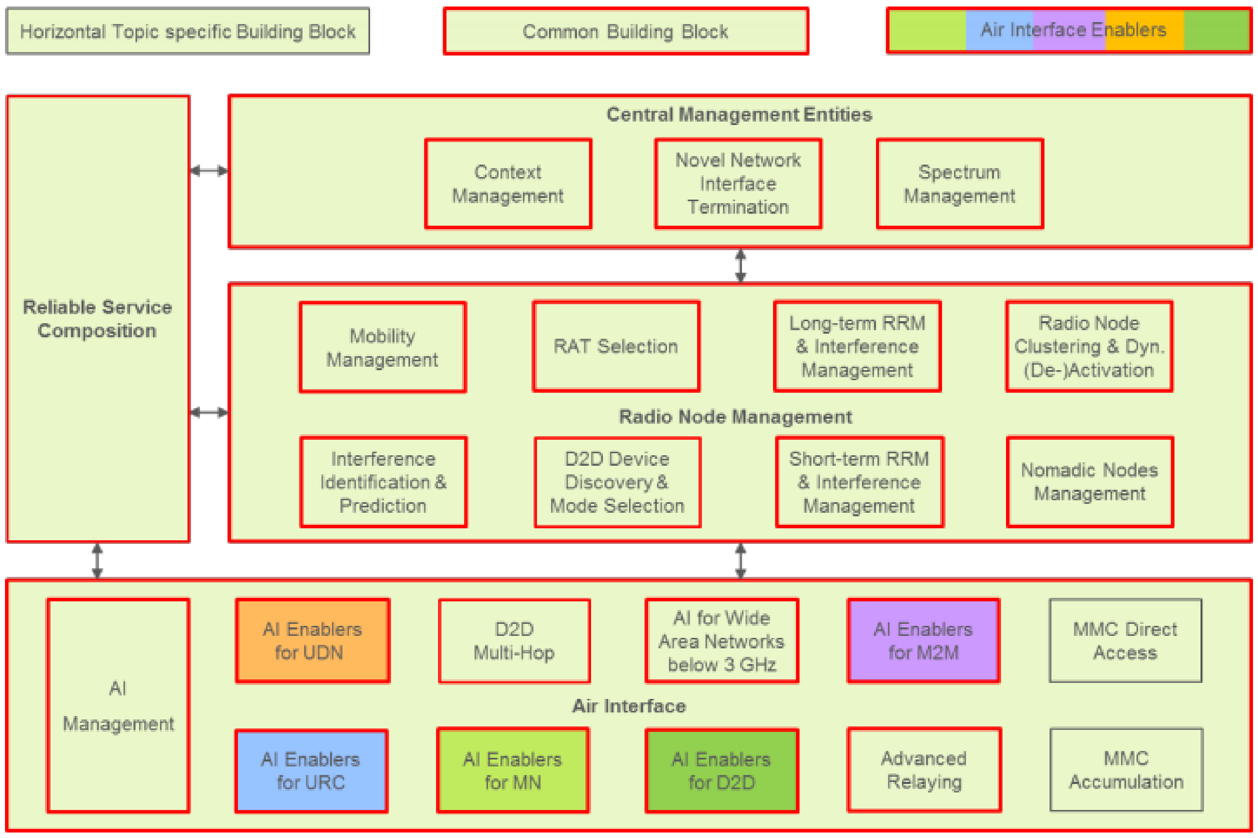

Horizontal Topic specific Building Block
Common Building Block
Air Interface Enablers
Central Management Entities
Context Management
Novel Network Interface Termination
Spectrum Management
Reliable Service Composition
Mobility Management
RAT Selection
Long-term RRM & Interference Management
Radio Node Clustering & Dyn. (De-)Activation
Radio Node Management
Interference Identification & Prediction
D2D Device Discovery & Mode Selection
Short-term RRM & Interference Management
Nomadic Nodes Management
AI Management
AI Enablers for UDN
D2D Multi-Hop
AI for Wide Area Networks below 3 GHz
AI Enablers for M2M
MMC Direct Access
Air Interface
AI Enablers for URC
AI Enablers for MN
AI Enablers for D2D
Advanced Relaying
MMC Accumulation

Figure 5: Building Blocks of the overall METIS system.

METIS approach to the 5G challenges is oriented to wireless access networks, unlike Superfluidity, which aims for a new architecture for the whole network. In addition, despite the similarity in the terminology between MEITS' BBs and Superfluidity's RFBs, it is important to note that the BBs refer to specific parts and functionalities of the METIS architecture, while RFBs could be any virtualized network service component.

2.2.2 EU iJOIN

iJOIN is a European project that has received funding from the EU's 7th Framework Programme. It aims to address the problem of rapidly increasing traffic volume in mobile networks. It does so, introducing a new concept "RAN as a Service". In RANaaS, RAN functionality is centralised through an open IT platform based on cloud infrastructure. This platform integrates smaller cells and heterogeneous backhaul, including a wireless one that connects antennas located in places where a wired connection cannot be provided. The IT platform and backhaul are optimized taking into account the possibility that antennas may not be connected to the control station through an optic fibre.

The iJOIN functional architecture described in [28], is divided in three layers that interact between each other: the physical layer, the MAC layer and the network layer. The physical layer is responsible for the actual transmission of messages over the wireless channel. It also exploits the interference of signals by means of cooperative uplink detection and downlink transmission schemes. The MAC layer is concerned with the MAC, RRC and RRM in RAN and in the backhaul network. This includes resource allocation, QoS scheduling, interference coordination and connection control, e.g. for handovers. The network layer is responsible for the efficient operation of the network in terms of cost, energy consumption, mobility support and latency. The interaction between the physical layer and the MAC layer focuses on the exchange of channel information such as SNR. The network layer provides information about the backhaul configuration and measurements to the other two layers.

Like METIS, this project takes a RAN and backhaul centric approach to the problems 5G will encounter, while Superfluidity aims to achieve the possibility to instantiate services on the fly in any part of the network.

3 Superfluidity innovation and architecture

3.1 Superfluidity research challenges

Superfluidity project focuses on three innovative pillars, namely:

- *Converged architecture design*: The main objective here is to provide an architecture that is location, time, and platform independent while providing high performance, ease-of-use, and security mechanisms to prevent the deployed services from harming each other or the network as a whole.
- *Platform-Independent Block and Function Abstractions*: Superfluidity targets to decompose (1) heterogeneous hardware and system primitives into block abstractions and (2) network services into basic functional abstractions. It will design a provisioning framework that matches function abstractions with the available hardware/block abstractions in order to *automatically* derive high performance and meet end-user SLA requirements, i.e., without forcing developers to understand low-level system details.
- *High Performance Block Abstractions Implementation*: This pillar consists of deriving block abstractions for each of the underlying hardware components, and to optimize their performance.

These three pillars allow us to move from the current architectural approach based on monolithic network components/entities and their interfaces, to an approach where components can be "constructed" via the programmatic composition of elementary "building blocks".

In our view, a specific 5G network deployment should comprise the combination of:

1) Elementary radio, packet, and flow processing primitives and events, formally specified and described independently of the specific underlying hardware, but 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.
2) Platform-agnostic node-level and network-level "programs" describing how these primitives interact, communicate, and connect to each other so as to give rise to specific (macroscopic, and formerly monolithic) node components, network functions and services.
3) A heterogeneous computational and execution environment, supporting the execution (and deployment) of such "programs" and the relevant coordination of the signal/radio/flow/network processing primitives.

Under this vision, operators will formally describe a desired network service as a composition of platform-agnostic, abstract elementary processing blocks. 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 elementary blocks 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.

This document offers an initial assessment of the reusable components in different domains of the networking landscape including the Cloud RAN and Mobile Edge Computing. We proceed to decompose existing monolithic network functionality into reusable components, and report our key observations to date. Our goal is not to build a definitive list of reusable components that should be standardised, but rather to start understanding what are the useful functionalities that might be the candidates for standard reusable components.

When discussing functional decomposition, we posit that two (not necessarily exclusive) directions can be taken. The first direction consists in identifying relatively "high-level" network functions (load balancers, firewalls, caches, analytics and monitoring functions, etc.) and "chain" such functions so as to deploy value-added network services. This direction is already quite established and somewhat mature in the networking community; for instance, it (implicitly) appears chosen by most of the current ongoing work in the NFV arena. At least in principle, the model for interconnecting such functions is also relatively established, and entails the formal description of a Direct Acyclic Graph (DAG), which connects the output of one or more function blocks to the input of other function blocks. This model has indeed found an extremely successful adoption not only in the NFV arena, but also in the configuration of software routers such as Click. In practice, several challenges still remain open for such an "high level" decomposition approach, and Superfluidity will address many key research topics in this field, including but not limiting to the necessary clear specification of the semantics of each involved block, not an easy task when blocks are complex and stateful.

A second decomposition direction, which we descriptively refer to as **"nano-decomposition"**, is certainly less explored. 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. Obviously, such a more aggressive decomposition model would ease the problem of semantics (the involved stateless primitives are much more elementary, and more easily specified and agreed upon, than an high level network function), and would give much more flexibility to the programmer since it would entail fine-grained customization of network functions. Getting back to the example of a load balancer, we would be in principle able to "program" how the load balancer should take internal decisions on each specific flow. But at the same time, by shifting "intelligence" (e.g. flow state handling) from the implementation of the function block to the composition of "dumb", stateless, primitives, this direction opens new research challenges, and specifically:

1) how to formally describe the way such elementary functions should be combined together so as to model a stateful behaviour using stateless primitives and in a platform-agnostic manner (a DAG most likely not being appropriate, as there is no immediate and simple way to formalize flow/device states using a DAG, nor trigger state updates with a DAG), and
2) how to design a concrete reference architecture that support such an approach (e.g., which internal architecture component should store and maintain states, how states are updated, which internal processing components should be envisioned, and so on).

The process of building complex functionality depends on ability of all the involved parties' to ***understand*** what each reusable component is supposed to do: what its allowed inputs are and how the outputs are obtained from the inputs. This is equivalent to the POSIX contract: what are the allowed ranges and formats for input parameters for system calls, and what is the expected output, error conditions and so forth. Another example is the contract between the CPU and the programmer: what each CPU-level instruction does, what the inputs and outputs are, etc.

This API contract is fundamental to decouple implementation from utilisation, yet it is completely missing today in works that aim to virtualize network functions such as ETSI NFV. To understand why this is the case, consider a simple firewall reusable component. At first sight, its semantics are obvious: it will drop all traffic not wanted by the network operator. However, there are many subtleties:

- What type of traffic does it accept beyond TCP and UDP? Does it filter traffic in tunnels, and if so which ones?
- Does it allow stateful processing (e.g. allow outgoing traffic and related incoming traffic)?
- Does it scan TCP options and drops unknown ones?
- Does it parse payloads and remove viruses?

All of these are plausible functionalities of the firewall, and they need to be expressed clearly so that users of the firewall component know exactly what it does, so that they compose it correctly with other reusable components. For instance, using tunnelling before the firewall may completely disable it if the firewall does not deal with that type of tunnel; and using a proxy after the firewall means that any filtering based on source and destination addresses may be rendered useless.

A key contribution of Superfluidity is to recognise the importance of associating semantics to each reusable component in a way that allows to safely compose them into correctly functioning network-wide applications. In particular, we identify two main approaches:

- The use of pre and post-conditions in the API to ensure type safety when traffic crosses multiple reusable components.
- The use of a modelling language to describe, at higher level, what each reusable component is doing. We can then use symbolic execution to understand how different components would when applied together to traffic.

3.2 Superfluidity architecture

Figure 6 shows the reference architecture as specified so far in Superfluidity and mainly in WP3. It is an end-to-end architecture including the wired part, the data centre and the core components, the wireless part composed of Cloud-RAN component. Mobile Edge Computing (MEC) is a new technology platform, which constitutes the last component considered in the project. These components are depicted in the top of Figure 6 where the bottom depicts the different types of data -centres involved (Cell-site, Local, Regional and Central).

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 orchestration of services (superfluid).

The Superfluidity project aims to design a unified and high performance distributed cloud platform technology for radio and network functions support as well as their migration. In our vision, C-RAN, MEC and information technology cloud technologies are integrated, by adopting a “divide and conquer” architectural paradigm in order to create the “glue” that can unify heterogeneous equipment and processing into a dynamically optimised, superfluid network.

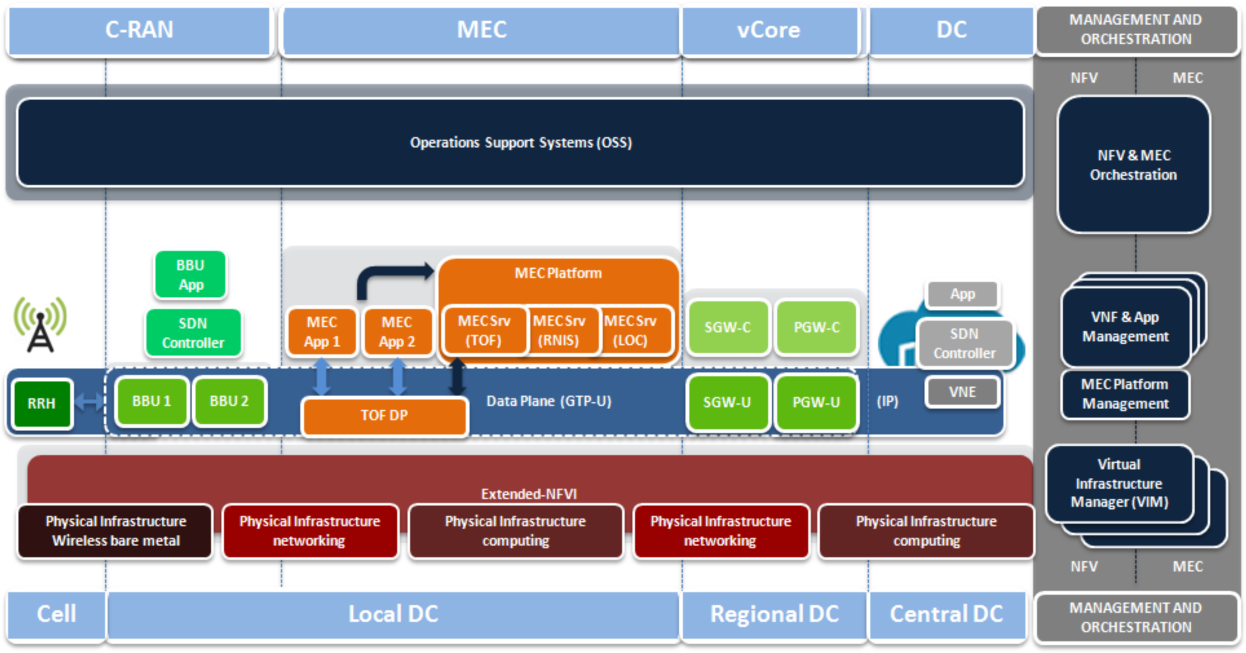


Figure 6: Overall Superfluidity Architecture.

Hereafter, we briefly introduce C-RAN and MEC.

- Cloud Radio Access Network:

  The main idea behind Cloud-RAN (CRAN) is to pool the Baseband Units (BBUs) from multiple base stations into centralised BBU pool for statistical multiplexing gain. A minimum of critical functions remain at the radio head (RRH, Remote Radio Head) whose main function is frequency shifting. With CRAN, it is then possible to design very tight coordination between cells and to maximise the radio capacity in bps/MHz/cell. Additionally, by leaving only the RRH on-site with a compact power supply, CRAN facilitates antenna site engineering and provides footprint reduction, as well as shorter time to install and lower rental costs.

- Mobile Edge Computing:

  A key enabler for low latency scenarios of 5G networks is the concept of Mobile Edge Computing (MEC), whose main idea is to deploy computing capabilities near the end users. In particular, MEC allows supporting Radio Access Network (RAN) processing and third party applications. This technology brings a set of advantages: i) proximity to users, ii) ultra-low latency, iii) high bandwidth, iv) real-time access to radio network information and v) location awareness. As a result, operators can open the radio network edge to 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 such that the MEC platform can rely on the same physical resources of the Extended-NFVI.

Toward a flexible 5G system, Superfluidity proposes to decompose these monolithic components into a Reusable Functional Block (RFB) to have a more fluid network. In a decomposed vision, instantiating a user service will be translated as a combination of the right RFBs each other's. Programmable and portable network functions, independent from the underlying platform

3.3 Key elements for achieving superfluid network programmability

The three main elements of a programmable superfluid network are:

- A set of basic/elementary (domain-specific) "***Reusable Function Blocks***" (RFBs), which are building blocks used to compose high-level functions and services. A RFB is a logical entity and it is the generalization of the concept of VNF (Virtual Network Function) and of VNFC (Virtual Network Function Component). RFBs can be composed to provide high-level functions and services but also to form other RFBs, as an RFB can be composed of other RFBs. 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 (see below) 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.

- An "**RFB Composition Language**" (RCL). 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". The actual specification of such language is 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.
- A "**RFB Composition Execution environment**" (RCE). 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. In most generality, we refer to such framework as "RFB Composition Execution environment" (RCE). The role of the RCE is to concretely instantiate a desired RCL script instance, control its execution, maintain and update the application-level states, trigger reconfigurations, and so on. As we want to address different levels of functional decomposition, we need to target an heterogeneous RCE, capable of controlling the execution of the RCL scripts over different platforms. The RCE 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 RCE provides the means to execute arbitrarily complex RCL scripts operating at different levels[1].

[1] To make an example, a Click router instance can be a VNF Component in a VNF. Its hosting environment is a hypervisor. The Click instance is described by means of a directed graph of elements (called configuration). In the current modelling

Figure 7 summarizes our proposed conceptual architectural model for programmable and portable network functions, independent from the underlying platform. The figure clearly highlights that the model (with due technical differences) may ***recursively*** apply at different hierarchy levels in the network. Indeed, one can envision it at the typical NFV level, where RFBs are usually meant to be relatively large functions (e.g. a load balancer, a firewall, etc.) and the RCE 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. OpenFlow-like elementary forwarding and processing actions).

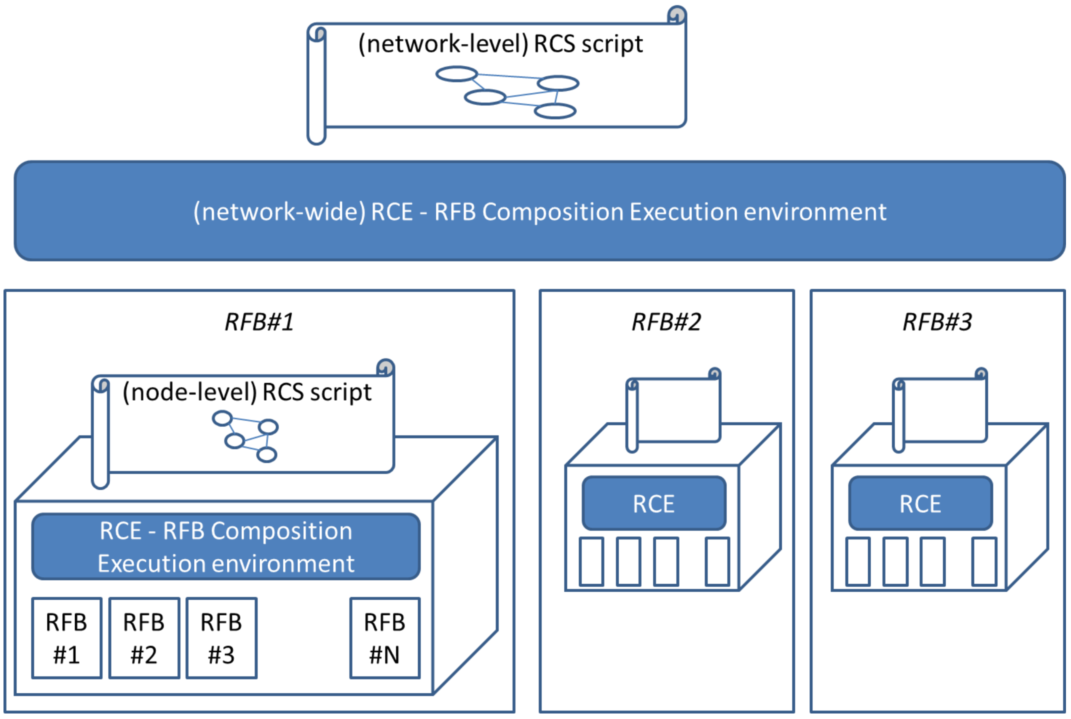

(network-level) RCS script
(network-wide) RCE - RFB Composition Execution environment
RFB#1
RFB#2
RFB#3
(node-level) RCS script
RCE - RFB Composition Execution environment
RFB #1
RFB #2
RFB #3
RFB #N
RCE
RCE

Figure 7: Conceptual model for Reusable Functional Blocks (RFBs) and RFB Composition Execution environments (RCEs)

3.3.1 A possible RFB Composition Language: NEMO

NEMO (Network Modelling) started as an effort to produce a human-readable representation of networks and network configurations in the IETF. The language proper is defined in [30]. Although there were some efforts in forming a Working Group at the IETF'94 in Prague, this effort has been abandoned. Currently, it is used in the IBNEMO project, an official project that is working on defining

approach it is not possible to further decompose the VNFC. In our vision, each element is a RFB that has the Click router as hosting environment and each RFB can in principle be further decomposed.

an Intent Based northbound interface using the NEMO language in the OpenDaylight (ODL) Beryllium release. The NEMO language defines Models for Nodes and Links and then allows users to instantiate them.

Figure 8 shows a service graph composed by three VNFs and four virtual links (VLs) as proposed in the Management and orchestration (MANO) framework [9]. NEMO can accommodate the VNFs in in NodeModel component and the VLs in LinkModel components.

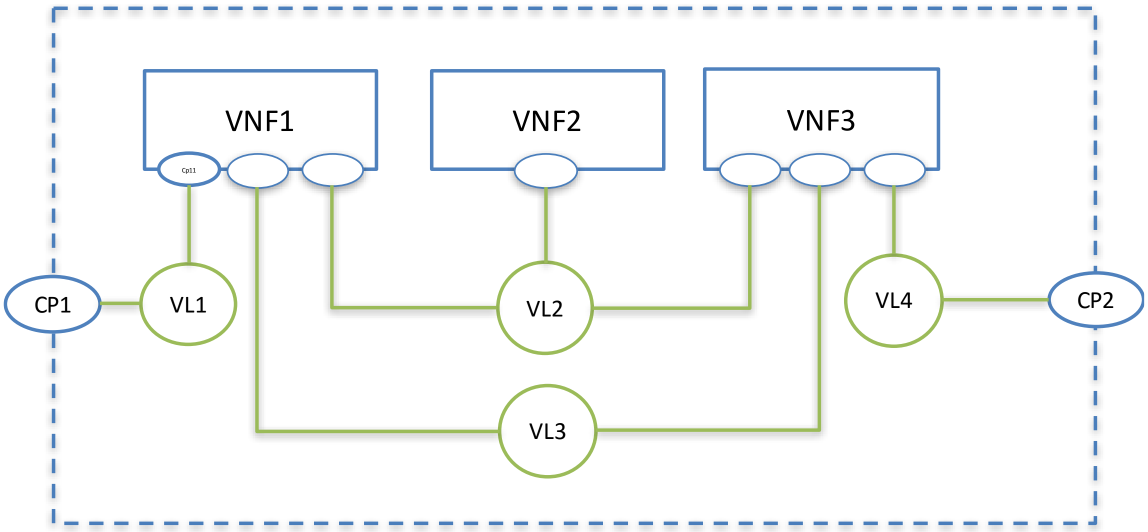


Figure 8: A service graph composed of virtual network functions and virtual links

Since NEMO is recursive in nature (you can use NodeModels in NodeModel definitions), it could represent a simple, yet powerful way of representing RFBs and their relationships in the scope of Superfluidity. Therefore, the project is analysing the NEMO applicability to the Superfluity concepts, and identifying the gaps and needed extensions.

4 Flexible network design: from monolithic functions to a set of RFBs

The first objective of Superfluidity project is to propose a flexible, an open and a programmable 5G data plane and relevant APIs for network functions' convergence. To entail that, Superfluidity proposes to decompose the architecture into elementary radio and network processing primitives and events, to be exploited as basic modules of more comprehensive (and traditionally monolithic) network functions and services. The decomposition of monolithic functions into reusable functional blocks (RFB) permit a flexible placement at different physical elements in the network (Cell-site, Local, Regional and Central), as well as the incorporation of adequate virtualization techniques (Docker, UniKernel, Virtual Machine) considering real-time constraints. Successful accomplishment of such goal will permit unprecedented reusability of radio and network processing tasks, and simplified/fastened deployment time, at reduced cost (reduced component-specific personnel's expertise needed). Vendor-neutrality and open interfaces will permit interoperability; more primitives that are elementary will reduce the entry barrier to third party stakeholders.

In the following subsections, we summarize the main reusable functional blocks identified in different environments: CRAN, MEC, generic NFV environment, fixed networking equipment, packet processing state machines.

We identified a methodology that can be used to facilitate the identification and description of Reusable Functional Blocks. It is based on the following template:

- **Description:** It described the scope of the functional blocks, its main processes, and functionalities.
- **Interface/Affinity with other functional blocks:** To implement any applicative service (e.g. video streaming session) or a network service (e.g. IMS), a set of functional blocks have to be chained. So it is necessary to know the neighbouring of each functional blocks called here affinity. For example, a MAC functional block could not be chained directly to the Mobility management functional block of the core. This affinity is essential to build the blueprint of a service or a network slice.
- **Synchronous or asynchronous characteristic**: it indicates the characteristic of the time synchronization at the level of TTI.
- **Control or Data plane**: In mobile network, we have two types of information data and control. Many control information are executed in order to monitor the state of the devices their respective mobility across cells... The data plane is related to the data consumed by the device. Therefore, it is important to identify the nature of the Functional block.
- **Hardware affinity**: Each functional block will take advantage, if possible, from the virtualization, except some functional blocks will be physically implemented e.g. conversion A/D, Amplifier... However, due to real time requirements of some components in Superfluidity architecture like the RAN, some acceleration in the hardware platform is

required. So for each functional block we specify its requirement on the need on the acceleration or not and what is the hardware platform/environment needed.

- Domain: the 5G architecture will be composed by three main domains: Front-End cloud (composed of a set of CRPI GWs), EDGE cloud and central cloud as it is depicted in Figure 9:

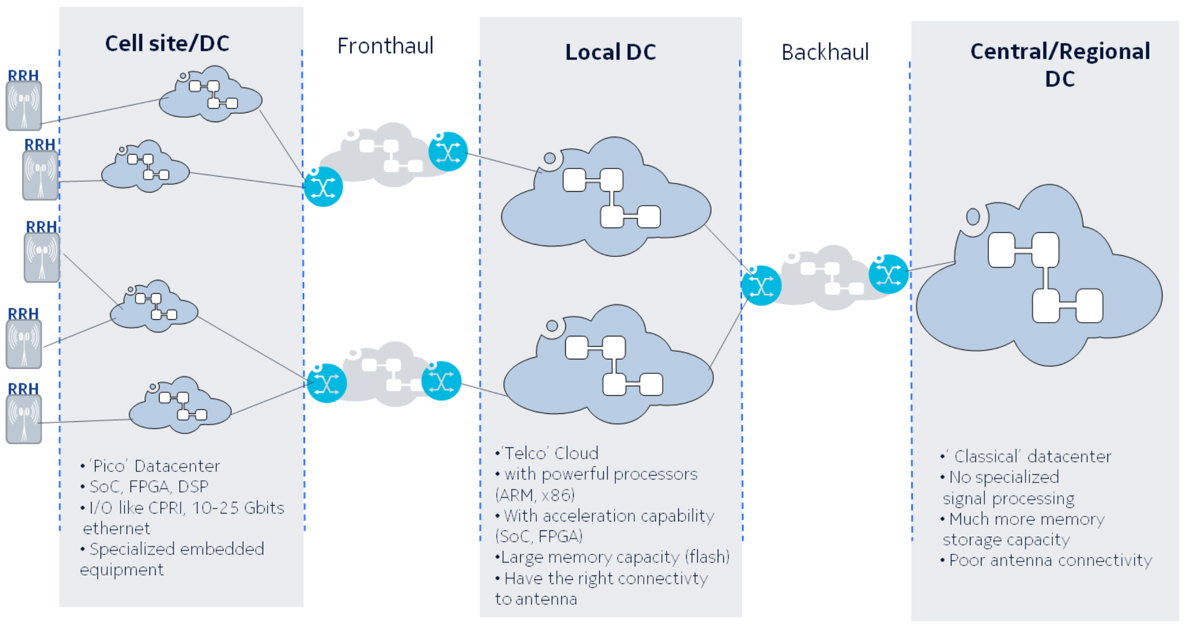

Cell site/DC
Fronthaul
Local DC
Backhaul
Central/Regional DC
RRH
RRH
RRH
RRH
RRH
• 'Pico' Datacenter
• SoC, FPGA, DSP
• I/O like CPRI, 10-25 Gbits ethernet
• Specialized embedded equipment
• 'Telco' Cloud
• with powerful processors (ARM, x86)
• With acceleration capability (SoC, FPGA)
• Large memory capacity (flash)
• Have the right connectivty to antenna
• 'Classical' datacenter
• No specialized signal processing
• Much more memory storage capacity
• Poor antenna connectivity

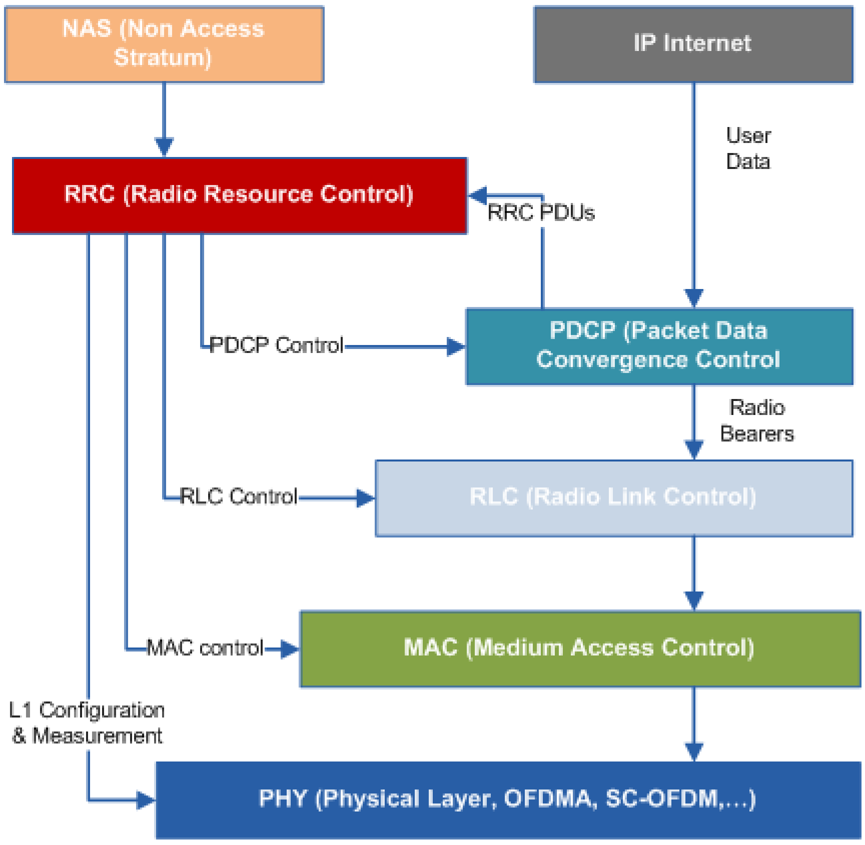
Figure 9: Different domain considered in Functional block placement: Central DC, Regional DC, Local DC and Front-end DC

For each functional block, the decision for placement should follow the following rules:

- Any functional block could theoretically instantiated/placed to any of the three domains: with virtualization, we are able to run a core functional block even in a front-end cloud.
- Traditionally monolithic functions are likely to be split between several domains to bring the required flexibility. For example, the physical layer will be split between front-end cloud and edge cloud; the mobile core functions will be split between central cloud and EDGE cloud. The splitting allows a lean and independent upgrade of the different functional blocks.
- Each FB will have real time constraints, which should be considered to determine the best place. For that, it is important to identify the needed timing for each FB:
    - In case of a hard real time function (ms), the FB need to stay within the vicinity of the antenna (Front-End cloud or EDGE cloud)
    - In case of soft real time functions (100s of ms - 1s), we can relax the location; EDGE Cloud or even Central cloud
    - In case of offline RFBs (e.g. Machine learning), the FB could be located in the Central cloud because of large storage capabilities.

4.1 Description of different C-RAN RFBs

In this section, we describe the list of identified FBs required for a Cloud RAN. Figure 10 depicts the six layers used as reference in the Radio Access Network and the relationship across them.



NAS (Non Access Stratum)
IP Internet
User Data
RRC (Radio Resource Control)
RRC PDUs
PDCP Control
PDCP (Packet Data Convergence Control
Radio Bearers
RLC Control
RLC (Radio Link Control)
MAC control
MAC (Medium Access Control)
L1 Configuration & Measurement
PHY (Physical Layer, OFDMA, SC-OFDM,...)

Figure 10: Monolithic version of the RAN and its different layers

The scope of the six layers is described hereafter:

- **Physical Layer (Layer 1):** The Physical Layer carries all information from the MAC transport channels over the air interface. It takes care of a set of processes related to the user's link (adaptation, power control), cell search for synchronization matter and other measurements for the RRC layer.
- **Medium Access Layer (MAC)**: The MAC layer is responsible for:
  - Error correction through HARQ,
  - Scheduling information reporting,
  - Multiplexing/ de-multiplexing of MAC SDUs from one or different logical channels onto transport blocks (TB) to be delivered to the physical layer on transport channels,
  - Priority handling between UEs,
- **Radio Link Control (RLC):** The RLC operates in 3 modes of operation:
  - Transparent Mode (TM),
  - Unacknowledged Mode (UM),

    - Acknowledged Mode (AM).

  RLC Layer is responsible for error correction through ARQ (Only for AM data transfer), concatenation, segmentation and reassembly of RLC SDUs (Only for UM and AM data transfer).

  RLC is also responsible for re-segmentation of RLC data PDUs (Only for AM data transfer), reordering of RLC data PDUs (Only for UM and AM data transfer), duplicate detection (Only for UM and AM data transfer), RLC SDU discard (Only for UM and AM data transfer), RLC re-establishment, and protocol error detection (Only for AM data transfer).

- **Radio Resource Control (RRC):** The main services and functions of the RRC layer include broadcast of system Information related to the non-access stratum (NAS), broadcast of System Information related to the access stratum (AS), Paging, establishment, maintenance and release of an RRC connection between the UE and E-UTRAN, Security functions including key management, establishment, configuration, maintenance and release of point to point Radio Bearers.
- **Packet Data Convergence Control (PDCP):** PDCP Layer is responsible for Header compression and decompression of IP data, Transfer of data (user plane or control plane), Maintenance of PDCP Sequence Numbers (SNs), In-sequence delivery of upper layer PDUs at re-establishment of lower layers, Duplicate elimination of lower layer SDUs at re-establishment of lower layers for radio bearers mapped on RLC AM, Ciphering and deciphering of user plane data and control plane data, Integrity protection and integrity verification of control plane data, Timer based discard, duplicate discarding,
- **Non Access Stratum (NAS) Protocols:** The non-access stratum (NAS) protocols form the highest stratum of the control plane between the user equipment (UE) and MME. NAS protocols support the mobility of the UE and the session management procedures to establish and maintain IP connectivity between the UE and a PDN GW.

By analysing the different processes supported in each layer which could be classified to control and data plane, some of them requires a synchronization to low layer time (TTI), others could be executed in asynchronous way and some processes are related to a dedicated user and others are related to a cell. So, based on that classification, we propose the following different functional blocks (see Table 1).

Table 1: Description of different C-RAN Functional Blocks

Functional Block (FB)	FB Description	Interface/Affinity with others FBs	Sync/Async	Control/Data Plane	HW affinity	Domain
PHY RRH	A/D; Signal generation and adaptation	It exchanges information with PHY cell functional Block (I/Q samples streaming)	sync	C and U plane	Physical Network Function (not virtualized) Hardware HW specific: FPGA, DSP	Antenna site
PHY Cell	It includes all the processes executed for one cell, e.g. FFT/iFFT, Modulation, Cyclic prefix	It exchanges information with PHY UE and PHY RRH FBs (FFT (I/Q) streaming)	Sync	C and U plane	It requires some acceleration capability FPGA, SoC, DSP, ARM	Antenna site or Front-End Cloud
PHY User (UE)	It covers all the physical processes executed per UE, e.g. FEC (Convolution coding (control) Turbo coding or Polar coding), HARQ,…	It exchanges information with PHY Cell, MAC UE MAC Cell (Bursty packet), and RRC	Sync /Async	C and U Plane	It requires powerful processors ARM, x86 and some acceleration capability FPGA, SoC	Front-End Cloud or EDGE cloud
MAC Cell/Scheduling Real Time	It covers all the processes related to the scheduling and the different optimization features like COMP, ICIC…	It exchanges information with PHY UE, MAC UE, RLC, RRC	Sync/Async	C and U Plane	It requires powerful processors ARM, x86	Front-End Cloud or EDGE cloud
MAC User (UE)	It covers processes related to UE like UE Power control; Padding; Multiplexing of TBs;	It exchanges information with, PHY UE, RRC, RLC (AM/TM/UM) and MAC Cell	Sync/Async	C and U Plane	It requires powerful processors ARM, x86	EDGE cloud
RLC	It includes processes related to segmentation/concatenation of PDCP PDUs based on information	It exchanges information with PDCP, MAC and RRC. It should be co-located with MAC	*Async*	C Plane (RLC TM), U/C plane for RLC AM/UM	Executed on generic HW (x86)	EDGE cloud

	exchange with MAC and PDCP. Several modes are supported: Transparent, Acknowledged and Unacknowledged. Each case could be a separate FB	Cell/UE FBs since the connection is delay sensitive				
PDCP	IT covers functionalities like transfer of user/control plane data, header compression, ciphering, (un) acknowledged data transfer service, in sequence delivery, duplicate discarding ...	It exchanges information with RRC and RLC	Async	U/C plane	Executed on generic HW (x86)	EDGE cloud or Central cloud
RRC Cell	It covers functionalities to convey messages to UE in RRC_idle/ RRC_connected associated with broadcast system information	It connects to all FBs: PHY RRH, PHY Cell, MAC Cell, RLC, PDCP and NAS	Async	C plane	Executed on generic HW (x86)	EDGE cloud or Central cloud
RRC User (UE)	It covers functionalities handling UE control and management: mobility functions (Handover), UE measurements reporting, QoS management, paging	It connects and interacts with MAC UE, MAC Cell, RLC, PDCP and NAS.	Async	C plane	Executed on generic HW (x86)	EDGE cloud or Central cloud
NAS User (UE)	It refers to the user procedures related to signalling between the UE and MME.	It connects and interacts with NAS Core, RRC UE, RRC Cell.	Async	C plane	Executed on generic HW (x86)	EDGE cloud or Central cloud
NAS Core	It refers to network management functions (MMEs load balancing, MME overload control, GTP-C signalling load control…)	It interacts with NAS UE	Async	C plane	Executed on generic HW (x86)	EDGE cloud or Central cloud

Remark: Some of the identified FBs in C-RAN could be decomposed in a finer granularity like PDCP functional block, which could be decomposed into two FBs: the first one focused on user plane grouping functionalities related to header compression and data transfer procedures that are specific to user Plane. The second one could group functionalities and procedures applied for both user plane and control plane like sequence number maintenance, (De)-Ciphering, Integrity Protection and Verification.

As depicted in Figure 11, we move from monolithic RAN functions to a functional blocks graph highlighting the affinity of the different identified FBs.

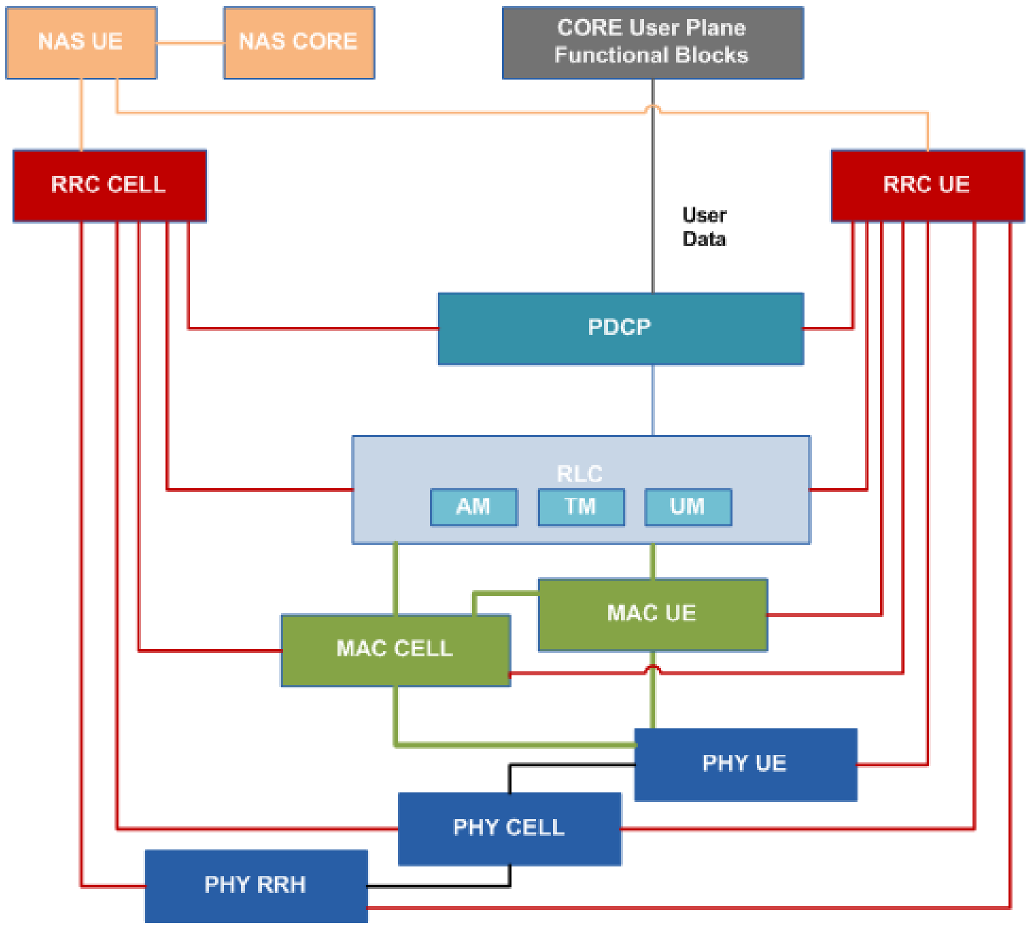


Figure 11: Affinity graph between different C-RAN functional blocks.

4.2 Description of MEC RFBs

4.2.1 MEC Overview

The Mobile Edge Computing (MEC) technology is currently under standardization by the ETSI in the MEC Industry Standard Group (ISG). MEC offers application developers and content providers a cloud computing hosting environment to deploy and run applications at the edge of a mobile network. The MEC environment is characterized by:

- User proximity
- Ultra-low Latency
- High bandwidth

MEC provides a set of services to be consumed by applications running on the MEC environment, such as radio signal conditions, network statistics, or end-user locations, among others. These services, together with the applications' privileged deployment location (at the edge), create a potential to enable brand new use cases, such as advanced IoT solutions (e.g. video analysis), augmented reality, or video streaming, among others, improving significantly the user experience.

MEC can take advantage of existing NFV deployments, due to the complementary use of cloud infrastructures, virtualization layers and management and orchestration components. Operators are increasingly looking at virtualization of their mobile networks and, in particular C-RAN, which is deployed at the edge. The synergy between the deployment of network functions and applications can further support the MEC concept by leveraging the deployment of applications at the edge.

The MEC architecture includes the MEC Host as the key point in the edge. The MEC Host allows 3rd party applications to get deployed on top of a cloud environment, but also allows applications to register and get authenticated/authorized to access to multiple services, available via APIs. Management functions are available to manage both, the applications lifecycle (LCM) and services.

In a centralized point, there is the orchestration layer, responsible to make decisions regarding the instantiation, termination and movement of applications along the different edges.

The Figure 12 depicts the MEC architecture as described by the ETSI MEC [32]. It reflects the influence from the ETSI NFV architecture.

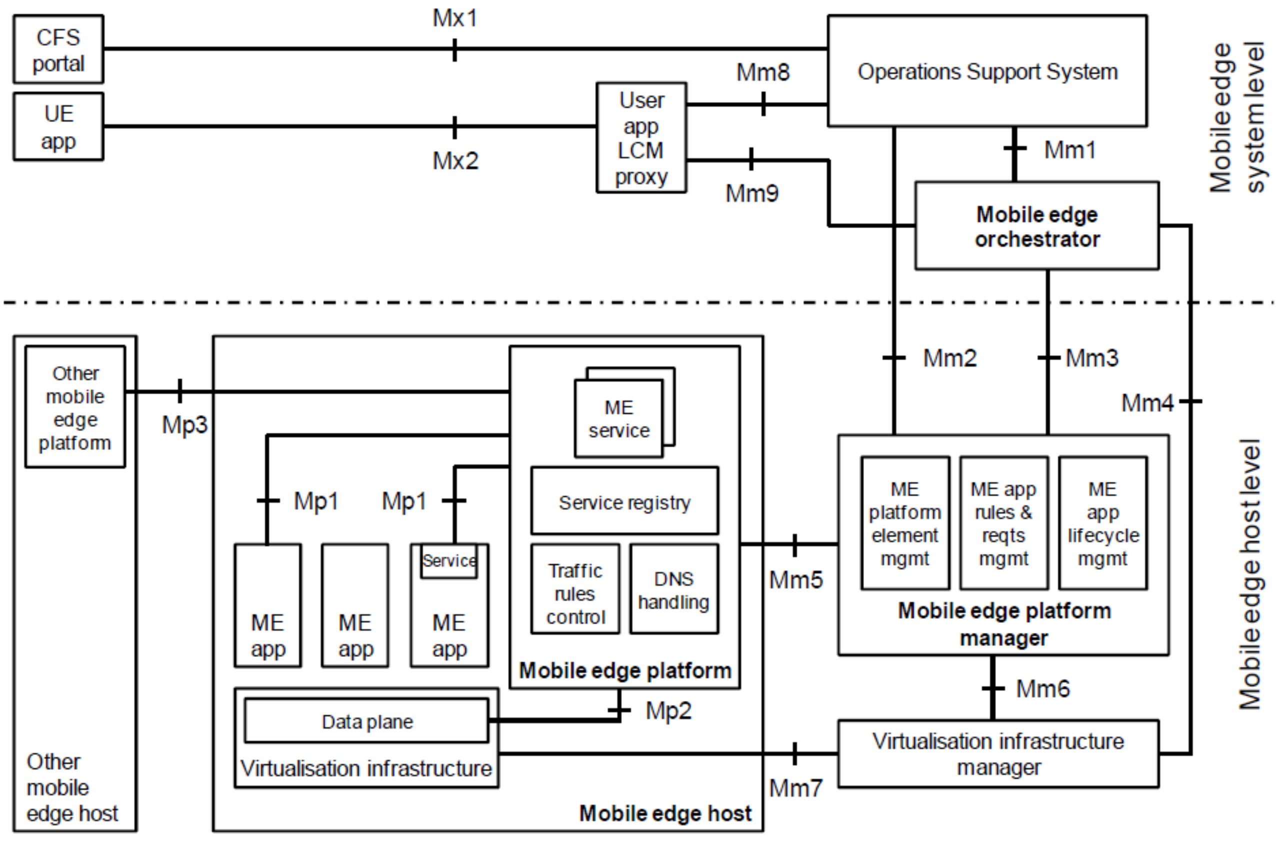


Figure 12: ETSI MEC architecture.

MEC is an important technology for 5G, as it complements NFV/SDN technologies by extending the DC to the edge and to applications (not only network functions). The MEC paradigm plays a pivotal role on 5G since will reduce significantly the latency for the new era of applications.

4.2.2 MEC RFBs

The MEC system comprises two levels of operation: the Host level and the System level. The Host level is located on mobile network edges and includes components like Cloud Infrastructure, Services, Apps and Managers. The System level is located at a centralized point and has a holistic view about the whole system. It includes essentially the MEC Orchestrator.

A full MEC system comprises a single MEC system level and multiple MEC Host levels (Hosts), each located in a particular edge location and associated with a certain mobile network coverage. Figure 13 provides a pictorial representation of the MEC building blocks.

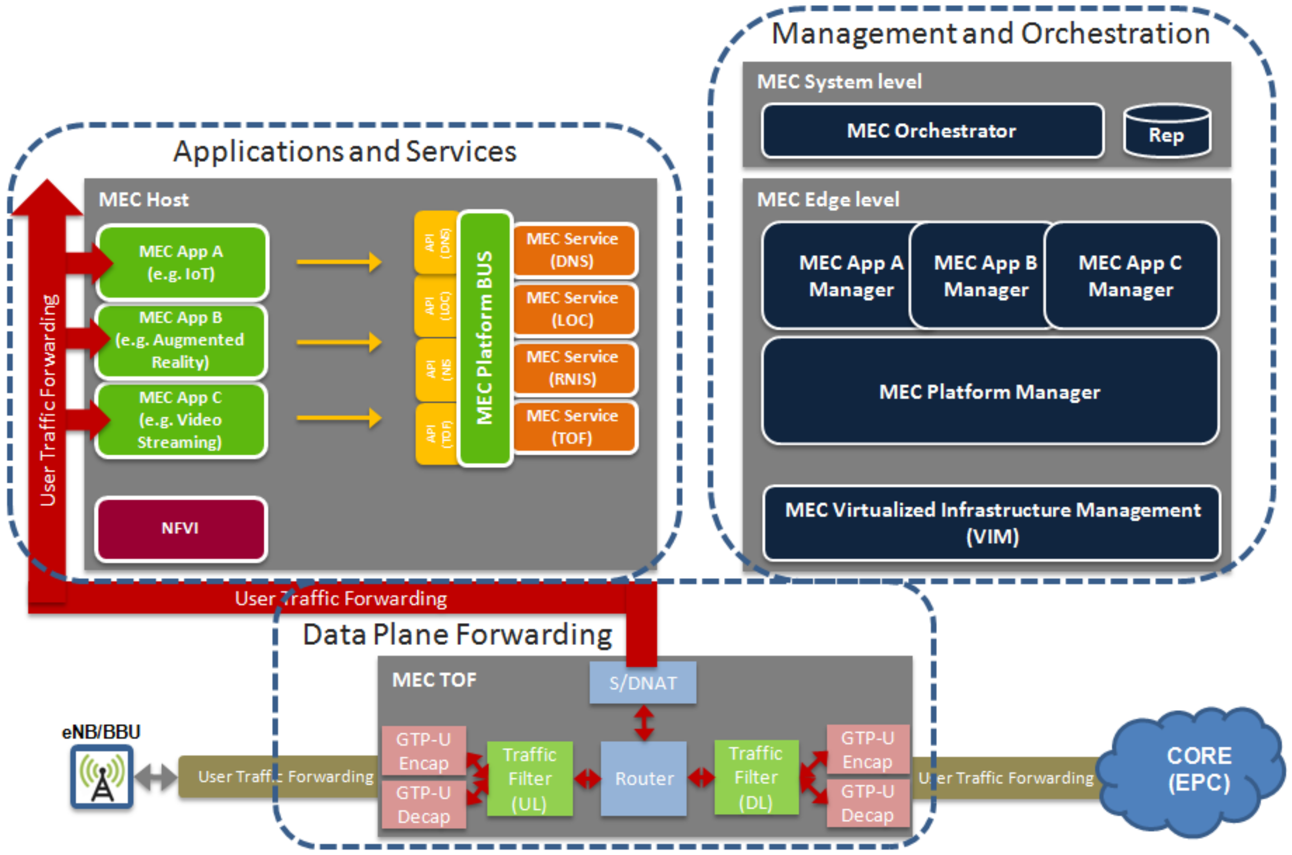


Figure 13: MEC building blocks

The tables below describe the main MEC components, as well as the smaller building blocks of the overall MEC solution.

Table 2: MEC main building blocks

MEC	Description
1. MEC TOF	The MEC TOF (*Traffic Offloading Function*) is located at the user data plane (S1-U), and is responsible to inspect and forward the subscribers' traffic towards the Applications and vice-versa.
2. MEC Host	A MEC Host comprises an NFVI to support the dynamic deployment of Applications, and a platform to allow those Applications to consume (via APIs) Services. Applications may also provide services to other Applications.
3. MEC MANO	The MEC MANO (*Management and Orchestration*) is responsible for managing the MEC Host. It can be divided into two main management components: App Managers (for managing the lifecycle of the Apps) and Platform Managers (for managing the platform, e.g. the Application access to Services).

These main building blocks can be further decomposed into smaller blocks, which may be reused (RFB). These are identified in the following tables.

1. MEC TOF	Description
1.1 GTP-U Decap	Decapsulates IP packets from GTP-U tunnels
1.2 GTP-U Encap	Encapsulates IP packets within GTP-U tunnels
1.3 Traffic Filter	Filters GTP-U and user IP traffic
1.4 Router	Routes GTP-U and user IP traffic
1.5 S/DNAT	Source/Destination Network Address Translation (NAT)

2. MEC Host	Description
2.1 Service (N)	Services to be consumed by Applications (e.g. TRF, LOC, RNIS, Monitoring, etc.). There can be multiple Service (N).
2.2 Service API (N)	Provide Service APIs for Apps access. There can be multiple Services API (N) for a given service.
2.3 Service Bus	Messaging Bus to make Service APIs available for Apps
2.4 MEC Apps (N)	MEC Applications running on the MEC Host environment. There can be multiple Apps (N).

Besides the RFB shown above, the MEC MANO is also made of blocks, but very unlikely reusable.

3. MEC MANO	Description
3.1 App Manager (N)	Manages the entire MEC Apps' lifecycle. There may be 1 App Manager per App (N).
3.2 Platform Manager	Manages the Platform, which includes the Service Bus, the Services and respective APIs.
3.3 MEC Orchestrator	Orchestrates the Apps deployment among the entire MEC System (composed by multiples edges).
3.4 MEC Orchestrator Repository	Stores Orchestration policies and other relevant system information.

4.3 Description of different NFV RFBs

NFV platforms, similar to IT public clouds, are expected to provide various tools to be used by the VNF providers. Those tools are expected to be provided as a service (according to the so called XaaS approach). Load balancer, analytics and state repository are three examples of services to be provided by the NFV platform. In this section, we further decompose those services into RFBs to enable faster instantiation, scaling, and migration of those services based on the traffic and usage needs. Then we focus on the Service Function Chaining functionality and provide its decomposition into logical components.

4.3.1 Load balancer as a functional block

This is a basic example for a functional block – it gets a stream of packets and needs to forward them (in a load balanced way) to a set of servers (services) that do the work on the stream of packets. While this is a very simple example, there are still interesting issues related to this example, which requires a further study.

- The load balancer could be realised as a single (physical) component that can be elastic (grow as needed) or could be a single component (we are referring to a load balancer as a functional block) that cannot scale.
- The load-balancer functional block interfaces should be defined. Specifically, we should further investigate how to define the incoming stream as well as how to define the number of server and their locations.
- Does it have a state (for example sticky load balancing all packets from the same flow go to the same server)? if so, what is a state and how this state is shared if at all?

Per ETSI GS NFV-SWA 001 V1.1.1 (2014-12) [3], among the different types of load balancing, typically 4 models are identified:

1) **VNF-internal Load Balancer**:

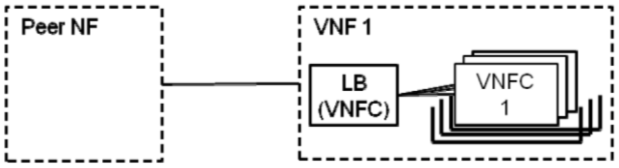


Figure 14: VNF-internal Load Balancer

- One VNF instance seen as one logical NF by a Peer NF. The VNF has at least one VNFC that can be replicated and an internal load balancer (which is also a VNFC) that scatters/collects packets/flows/sessions to/from the different VNFC instances. If the VNFCs are stateful, then LB shall direct flows to the VNFC instance that has the appropriate configured/learned state.
- Examples: VNF Provider specific implementation of a scalable NF.
- Single VNF Provider solution (per definition of VNFCs).
- The VNFM instantiates the LB, which may itself be a VNFC.

2) **VNF-external Load Balancer** (analogous to 1:1 (drop-in) replacement of existing NFs):

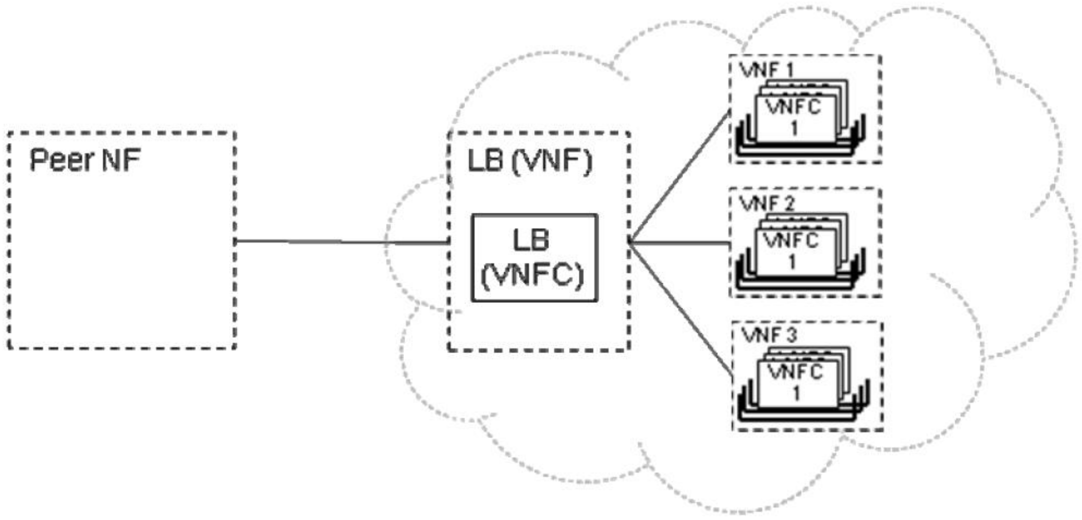


Figure 15: VNF-external Load Balancer

- VNF Instances seen as 1 logical NF by a Peer NF. A load balancer external to the VNF (which may be a VNF itself) scatters/collects packets/flows/sessions to/from the different VNF instances (not the VNFCs!).
- Examples: Application Delivery Controller (ADC) type LB or Direct Server Return (DSR) type LB in front of web-server.
- VNFs may be of different VNF Providers, e.g. to increase resilience.
- If the VNF supports this model, the NFVO may instantiate it multiple times and add a LB (V)NF in front of this pool of VNF instances.
- If the VNFs contain state, then the LB VNF shall direct flows to the VNF instance that has the appropriate configured/learned state.

3) **End-to-End Load Balancing**:

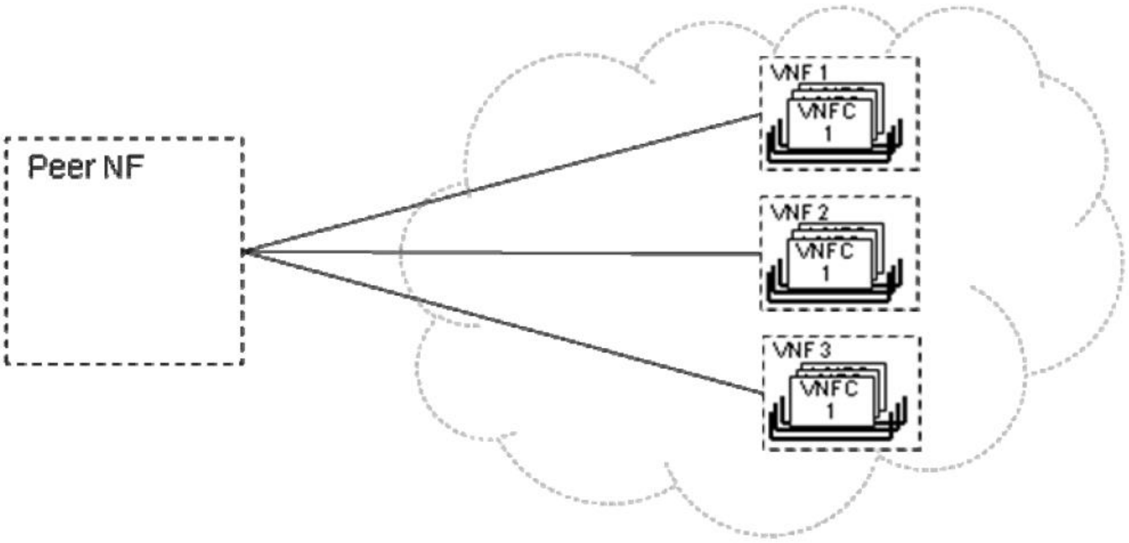


Figure 16: End-to-End Load Balancer

- N VNF instances seen as N logical NFs by a Peer NF. The Peer NF itself contains load balancing functionality to balance between the different logical interfaces, see figure 12. For example:
    - In 3GPP, S1-flex interface between eNBs (= Peer NF) and MME, S-GW.
    - Client-side (DNS-based) load balancing between web servers.
- VNFs may be of different VNF Providers, e.g. to increase resilience.
- If the VNFs contain state, then the LB NF shall direct flows to the VNF instance that has the appropriate configured/learned state.
- The NFVO may instantiate multiple VNFs, but does not instantiate a LB.

4) **Infrastructure Network Load Balancer**

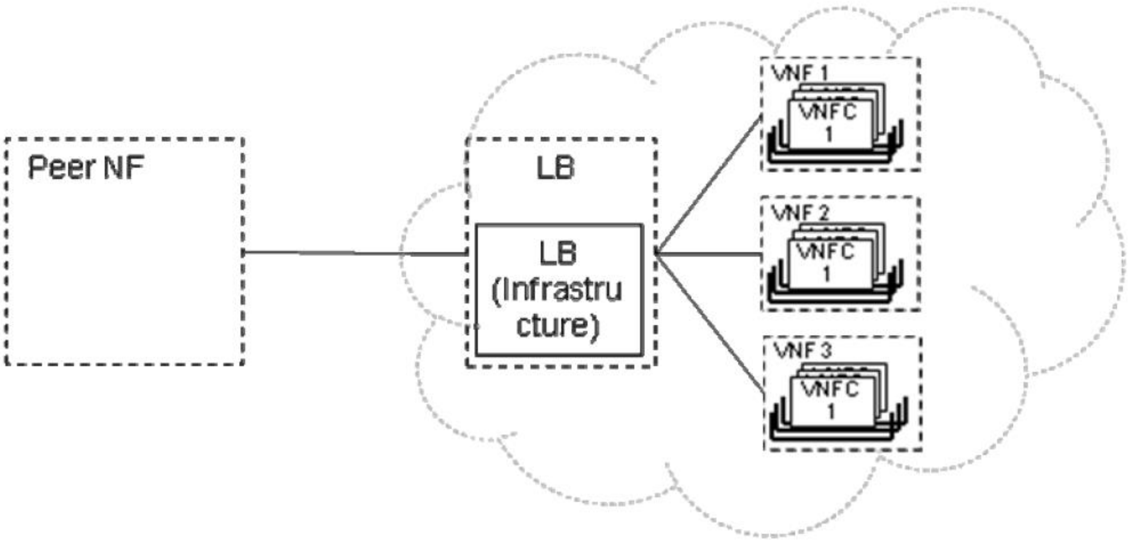


Figure 17: Infrastructure Network Load Balancer

- VNF instances seen as one logical NF by a Peer NF. A load balancer (may be in a hypervisor vSwitch, an OS vSwitch, or a physical box) provided by the NFV Infrastructure that scatters/collects packets/flows/sessions to/from the different VNF instances. Examples: NFV Infrastructure providing load balancing for several Firewall instances.
- VNFs may be of different VNF Providers, e.g. to increase resilience.
- If the VNF and the NFVI supports this model, the NFVO may instantiate it multiple times and configure a load balancer in the NFV Infrastructure that connects to this pool of VNF instances to perform load balancing.

Something important to note is that load balancers need to be reconfigured upon VNF/VNFC instantiation and termination, occasionally under MANO control. In addition, load balancers occasionally support health-checking functions that allow them to automatically stop sending packets to servers that become unresponsive, and resume sending packets once these servers become healthy again.

Another important consideration of in-network services is the so-called firewall load balancing (Figure 18): unlike server load balancing, where the clustering can be realized using one load balancer, network service clustering requires two (logical) load balancers, one on each side of the cluster. In addition, these two load balancers need to ensure that the packets belonging to a TCP/UDP connection go through the same network service device. Many network security services are stateful in nature. They can do better security analysis of the traffic only if all the packets belonging to a connection are seen.

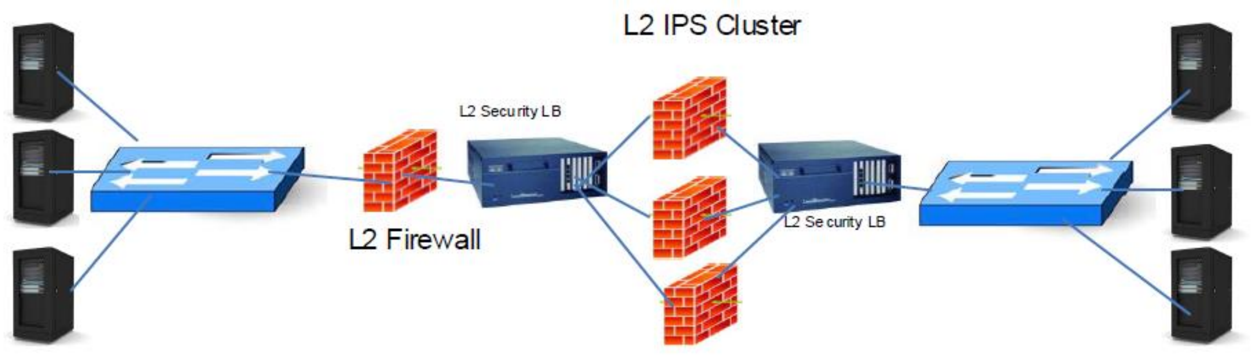


Figure 18: Firewall Load Balancing

4.3.2 Analytics as a functional block

The analytics functional block is a more complex block, where the input here can be a stream of data items and the output can be an alert when an abnormal characteristic in the data is detected. This can be as simple as a threshold based alert notification, or could utilise a very complex machine-learning algorithm that takes history into account.

Snap telemetry system [29] decomposes the Analytics into three main components, namely, collectors, processors, and publishers (as depicted in Figure 19).

- Collectors - Collectors consume telemetry data.
- Processors - Extensible workflow injection. Convert telemetry into another data model for consumption by existing consumption systems.
- Publishers - Store telemetry into a wide array of systems.

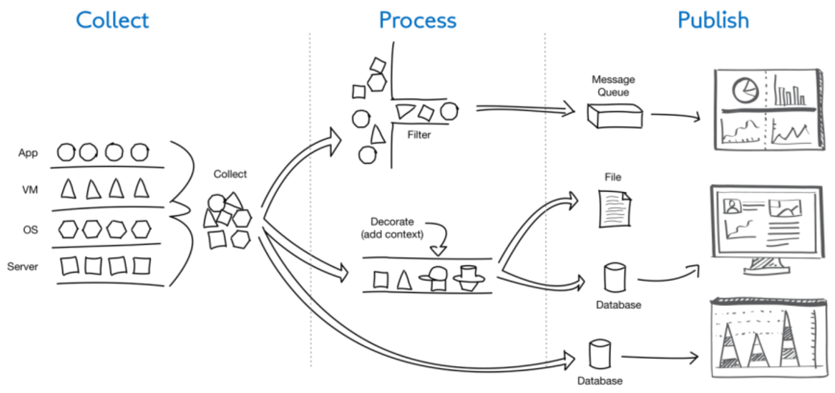


Figure 19: snap decomposition

In Superfluidity we further differentiate the analytics as a service into edge and core blocks to enable real-time analytics at the edge networks that responds in a short timing, as well as offline analytics at the core network. As for the offline analytics, we further decompose it into smaller components, namely, root cause analysis, alarm correlation and policy/rule engine. Figure 20 depicts the proposed decomposition. Although those RFBs are differentiated by the location they are deployed in, we are not limiting the allocation of those blocks, and in fact, RFBs could be shifted from the edge cloud to the central data centre and vs. versa, based on the real time requirements form the analytics engine. That is, for example, if latencies permit, all data collectors can be located at the central data centre.

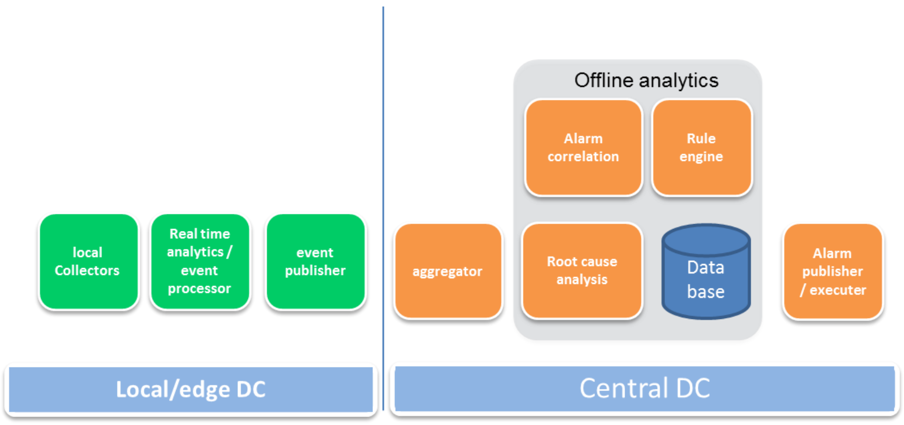


Figure 20: decomposition of analytics block

- Local Collectors

The collectors RFBs collect the information from the various sources, such as infrastructure information (e.g., from Nagios or ceilometer), applications data, user information, etc. The collector RFB should typically be located near the information source to avoid both delays as well as transfer of high traffic loads to the core network.

- Real time analytics/ event processor

This block aims at analysing real time information and provide alerts based on immediate information collected by the telemetry system. Decisions and inputs from the event processor are transferred to the event publisher, either to trigger operation or to transfer processed information for offline analytics.

- Event publisher

This event processor can trigger alarm or any other operation as well as provide aggregated/processed information and alerts to the central aggregator.

- Aggregators

The aggregator may collect information from the local publishers as well as from additional data sources. The information is stored in a repository for offline analytics. The aggregators could be located at the aggregation node or at the core network.

- Alarm correlation

Receives alerts from the aggregators correlates the alarms and provide the minimal sets of alarms. The alarm correlation RFB would probably be located at the central data centre. If all information is local, then the data base as well as the alarm correlation would be located at the edge network.

- Root cause analysis

Identifies the root cause of a failure. Typically, the root cause analysis is based on offline analytics and rely on machine learning techniques,

- Rule engine

Data analytics and predictions would be performed by this RFB.

- **Alarm publisher**

Based on information received by the offline analytics (e.g., alarm correlation RFB), the alarm publisher produces alarm and populate it through the application publishing mechanism (e.g., the support system web page).

4.3.3 State repository as a functional block

An important aspect of moving to the cloud is the ability to scale in a timely manner. To that end, it is clear that VNFs should be redesigned and decomposed in (i) stateless workers, and (ii) state repository distributed across the infrastructure. Such an architecture allows fast instantiation of new workers and synchronisation of their role with the state that is available and synchronised to multiple locations.

In the following, we focus on the cloud RAN use case as an example for the usage of the State repository functional block.

The state for the RAN MAC is composed of a control state keeping the channel information for the users and the priorities of the packets (to meet with the required QoS), as well a data state which is composed of the actual pending packets for transmission. Then the MAC scheduler works on the control data to derive the transmission order in the next frame. The scheduler output is fed to the frame builder that takes the actual packet and constructs the frame or the streams to be transmitted from each antenna.

Accordingly, for the wireless MAC scheduler we could identify two state types, namely, control state, and data state.

- Control state
  - Channel states for the different users to the different antennas/cells.
  - Packet priorities (e.g., deadline, MIR and CIR, PF metric, etc.)
- Data state
  - The data queues for each user (organised according to the traffic QoS)

Now, instead of handling this state information by the VNF (i.e., the wireless MAC function), it would be more efficient to utilise a more generic state repository functional block.

Here, the state functional block has various requirements such as:

- Storage volume: the control state consumes small volume, but the data packets may require high volumes.
- Storage traffic: Both the control and data states updates frequently in about every 1msec (read and write)
- Distribution: MAC process may shift between access nodes (e.g., due to handovers) or between the access and the aggregation node (when transmitting data in a multi-cell MIMO

techniques). Accordingly, the state should be synchronised and available at the surrounding access node and the nearest aggregation node.

4.3.4 Service Function Chaining

Service function chaining enables the creation of composite (network) services that consist of an ordered set of service functions (SFs) that must be applied to packets and/or frames and/or flows selected because of classification.

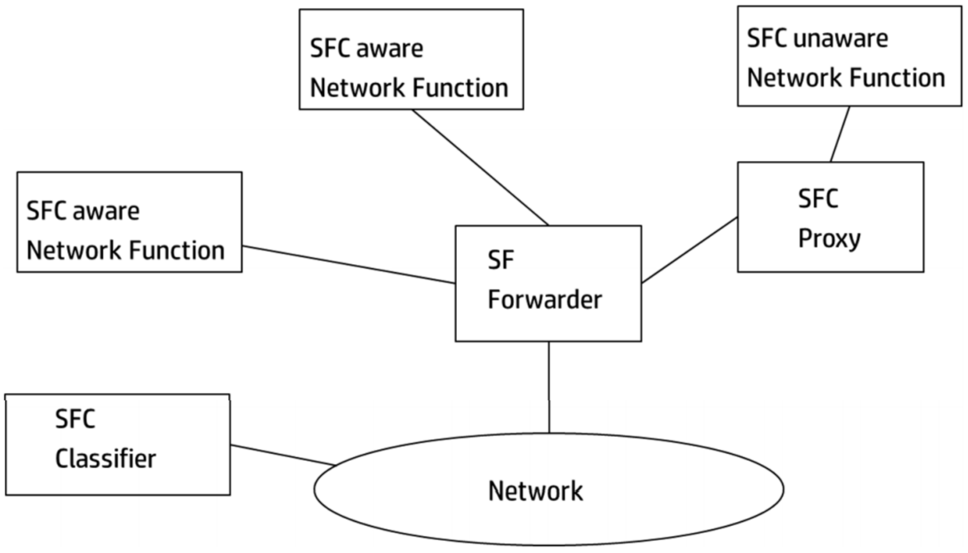

SFC aware Network Function
SFC unaware Network Function
SFC aware Network Function
SF Forwarder
SFC Proxy
SFC Classifier
Network

Figure 21: Service Function Chaining (SFC) architecture

Defined by IETF, the SFC Architecture is described in [33]. The SFC Architecture is built out of architectural building blocks that are logical components. These logical components are:

- SFC-aware and SFC-unaware Service Functions
- SFC Classifier
- SF Forwarder
- SFC Proxy

Components are interconnected using the SFC encapsulation. The SFC encapsulation enables service function path selection. It also enables the sharing of metadata/context information when such metadata exchange is required.

4.3.4.1 Service Function

The concept of a Service Function (SF) evolves; rather than being viewed as a bump in the wire, an SF becomes a resource within a specified administrative domain that is available for consumption as part of a composite service. SFs send/receive data to/from one or more SFFs. SFC-aware SFs receive this traffic with the SFC encapsulation.

While the SFC architecture defines the concept and specifies some characteristics of a new encapsulation – the SFC encapsulation – and several logical components for the construction of SFCs, existing SF implementations may not have the capabilities to act upon or fully integrate with the new SFC encapsulation. In order to provide a mechanism for such SFs to participate in the architecture, an SFC Proxy function is defined. The SFC Proxy acts as a gateway between the SFC encapsulation and SFC-unaware SFs.

4.3.4.2 Service Function Forwarder

The Service Function Forwarder (SFF) is responsible for forwarding packets and/or frames received from the network to one or more SFs associated with given SFF using information conveyed in the SFC encapsulation. Traffic from SFs eventually returns to the same SFF, which is responsible for injecting traffic back onto the network. Some SFs, such as firewalls, could also consume a packet.

The collection of SFFs and associated SFs creates a service-plane overlay in which SFC-aware SFs, as well as SFC-unaware SFs reside. Within this service plane, the SFF component connects different SFs that form a Service Function Path (SFP).

The SFF component has the following primary responsibilities:

1. SFP forwarding
2. Terminating SFPs
3. Maintaining flow state

4.3.4.3 Network Overlay and Network Components

Underneath the SFF there are components responsible for performing the transport (overlay) forwarding. They do not consult the SFC encapsulation or inner payload for performing this. They only consult the outer-transport encapsulation for the transport (overlay) forwarding.

4.3.4.4 SFC Proxy

In order for the SFC architecture to support SFC-unaware SFs (e.g., legacy service functions), a logical SFC Proxy function may be used. This function sits between an SFF and one or more SFs to which the SFF is directing traffic.

The proxy accepts packets from the SFF on behalf of the SF. It removes the SFC encapsulation, and then uses a local attachment circuit to deliver packets to SFC-unaware SFs. It also receives packets back from the SF, reapplies the SFC encapsulation, and returns them to the SFF for processing along the service function path.

Thus, from the point of view of the SFF, the SFC proxy appears to be part of an SFC-aware SF.

4.3.4.5 SFC Classifier

Traffic from the network that satisfies classification criteria is directed into an SFP and forwarded to the requisite service function(s). Classification is handled by a service classification function.

The granularity of the initial classification is determined by the capabilities of the classifier and the requirements of the SFC policy.

For instance, classification might be relatively coarse: All packets from this port are subject to SFC policy X and directed into SFP A, or quite granular: All packets matching this 5-tuple are subject to SFC policy Y and directed into SFP B.

As a consequence of the classification decision, the appropriate SFC encapsulation is imposed on the data, and a suitable SFP is selected or created. Classification results in attaching the traffic to a specific SFP.

4.4 Decomposition of a Cisco Security Appliance (ASA 5510)

4.4.1 Overview

An important step in the transition to 5G is the task of choosing the right building blocks that can be used to easily create new network functionality. Granularity matters: larger building blocks are easier to understand but harder to use to enable other functionality; smaller building blocks may be desirable but is it feasible to decompose complex functionality into such small building blocks? The Click modular router proposes a suite of small elements, each with clearly defined semantics, as a way to enable the creation of custom packet forwarding functionality. A major benefit for Click elements is that they capture basic network functionality which we can easily model using of symbolic execution-friendly language called SEFL and described in the upcoming architecture deliverable D3.1). Thus, any Click configuration (a directed graph of Click elements through which packets flow) can be automatically translated in SEFL and verified. Network administrators can use Symnet – our symbolic execution tool for SEFL - to trace-down bugs, check security policies, etc. Second, Click is directly executable.

Cisco’s Adaptive Security Appliance ASA5510 (short ASA), is a hardware appliance which performs deep-packet inspection, along with basic traffic filtering and NAT. An ASA 5510 is the core-device in University Politehnica of Bucharest Computer Science Department network. In this section we describe how we have expressed the functionality of the ASA box as a Click modular router configuration. Our “ASA-in-Click” can be used to replace the physical ASA for NFV.

4.4.2 A Click model for ASA

The ASA model contains over 200 Click elements. It performs VLAN switching, routing, traffic filtering, static and dynamic NAT, TCP options filtering. Additionally, our model implements Cisco's security levels: each interface is assigned a numeric value – its security level. Traffic from interfaces with lower security-levels cannot reach those with higher security levels. The following table illustrates the Click element types used for each ASA operation.

ASA operation	Click Element Types
Ethernet and VLAN switching	EtherEncap EtherDecap VLANEncap VLANDecap HostEtherFilter
Security Levels	Paint PaintSwitch
Traffic filtering (ACLs)	IPClassifier
Static NAT (IP source/destination rewriting)	IPClassifier IPRewriter
Dynamic NAT	IPClassifier IPRewriter
TCP options filtering	TCPOptions (written by us)

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. We have also used Symnet in order to verify the model. Symnet proved instrumental in finding subtle modelling bugs such as incorrect handling of (specific) return-traffic subject to NAT.

In Figure 22, a simplified traffic pipeline of our ASA model is shown. The following code snippet illustrates Ethernet and VLAN operations for that pipeline:

```
in_PROF_CS -> host_ether_PROF_CS :: HostEtherFilter(00:23:eb:bb:f1:4d)
                    -> in_vlan_PROF_CS :: VLANDecap()
        -> ether_decap_PROF_CS :: EtherDecap()
                    -> pn_PROF_CS :: Paint(80);
```

Packets only destined for the current interface are accepted. VLAN tags are stripped, as well as the Ethernet.

In the second phase, we use Click annotations (Paint & PaintSwitch) to implement Cisco's forwarding security level rule. In the above snippet, 80 is the security level assigned to the "PROF_CS" interface. Packets that are not dropped by the "Security Level" elements move on to the next phase.

The ASA handles TCP and non-TCP traffic differently. We use an IPClassifier element to make this distinction on packets. Non-TCP traffic may or may not be subject to IP source/destination rewriting (without storing mappings), which is handled by "Static NAT", illustrated schematically below:

```
pn_PROF_CS -> … //non-tcp traffic
    -> static_cl_PROF_CS :: IPClassifier(src host 172.16.5.222,-)[0]
    -> static_rw_PROF_CS :: IPRewriter(pattern 141.85.225.152 --- 0 1)
          -> next_stage
static_cl_PROF_CS[1] -> next_stage
```

TCP traffic for which NAT mappings exist (shown in green in Figure 22) is directly moved to the routing phase and subsequently forwarded to the destination.

The entire NAT operation is handled by a single IPRewriter element, illustrated schematically below:

```
global_nat :: IPRewriter(keep 0 1,
                   [pattern p_i nat_i nat_repl_i,]
                   keep no_nat_i no_nat_repl_i,
                   …);
```

In the above snippet, i ranges over all ASA interfaces. If traffic on an interface may be subject to NAT, a "rewrite pattern" p_i is created. nat_i and nat_repl_i identify the output ports for direct (natted) resp. return traffic on interface i. For each interface we also define a "keep" pattern, which stores a mapping without modifying traffic. As before, no_nat_i and $no\_nat\_repl_i$ are the direct resp. return traffic on interface i. We use "keep" in order to identify return traffic for active TCP connections and forward them without passing through the filtering stage.

Any type of new traffic (IP traffic from the "Static Nat" stage, and TCP traffic egressing ports nat_i and no_nat_i, shown in orange in the figure) is subject to filtering (including TCP-options filtering which is realized by a custom-written "TCPOptions" element), and may be dropped, depending on the destinations' security-level. A snippet is shown below:

```
cl_incoming :: IPClassifier(
 src host 37.128.224.6,
 dst net 141.85.228.0/26,
 dst host 141.85.225.151 and dst tcp port 21,…,-)
      -> opts :: TCPOptions()
      -> PaintSwitch (…)
```

The element "cl_incoming" models ASA's access control list "incoming". The "TCPOptions" Click element has been custom-written.

The routing phase is implemented as "destination-based forwarding" using an IPClassifier element. The appropriate VLAN and Ethernet encapsulations are finally applied.

We note that the routing stages for NAT and non-NAT traffic are different, because security level filtering does not apply to the former. We opted for this pipeline structure in order to avoid using different types of Click annotations (in addition to those used by the Security Level box), and to make debugging of the ASA model easier.

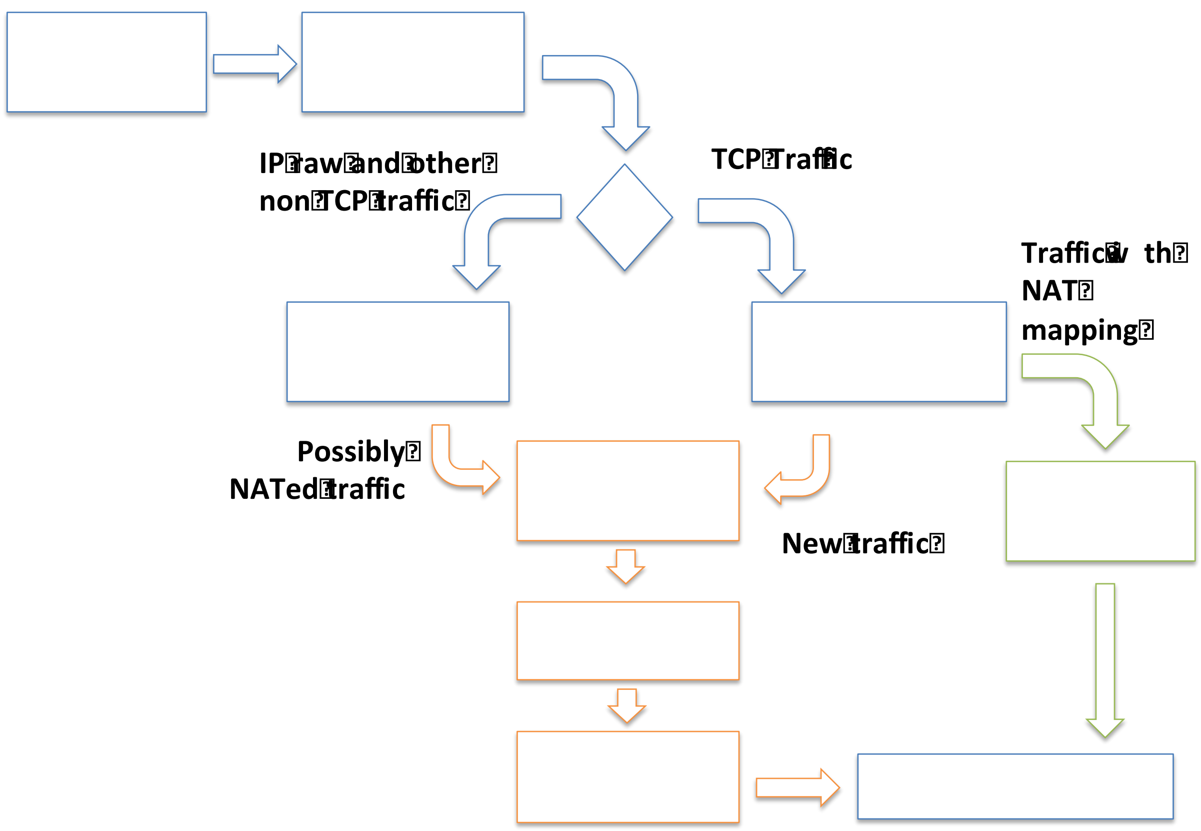


Figure 22: A simplified ASA pipeline

4.5 Nano-decomposition with state machines

In section 2.1.3, we have discussed how the networking community is focusing on the OpenFlow abstraction to provide decomposition of switching functions. In what follows, we propose an approach which not only shows that nano-decomposition appears technically viable without getting rid of the crucial platform-independence property, but also that it may be somewhat cast as an OpenFlow extension. The scope of this deliverable is to provide a functional decomposition into Reusable Functional Blocks, while the architectural aspects should be demanded to the Work Package (WP) 3. On the other hand, we need to introduce some architectural aspects hereafter, taking into account the novelty of the proposed approach.

Our proposal will specifically entail the definition of a platform agnostic programming abstraction based on **eXtended Finite State Machines - XFSMs** (see section 4.5.1), which is a much more general and expressive abstraction than the OpenFlow match/action one, and which permits to describe stateful network processing tasks, opposed to the stateless nature of the OpenFlow match/action abstraction. Based on this abstraction, a set of identified Reusable Functional Blocks for XFSM will be described in section 4.4.2.

4.5.1 Abstracting Flow Processing with eXtended Finite State Machines

To the best of our knowledge, the usage of XFSMs to model in an abstract manner network functions has been pioneered in the field of Wireless Medium Access Control protocols by the Flavia EU project [22]. The relevant first publication [24] indeed shows how to formally describe and "execute" an 802.11 MAC (and any variation thereof, including TDMA schemes) using an XFSM-based bytecode machine language running on an XFSM execution engine, called Wireless MAC Processor, directly implemented in the Wireless Network Interface Card. A network switch is however a completely different device: it must handle possibly millions of flows in parallel and with state transitions in the nanoseconds packet-level timescale (opposed to a "single" wireless channel access operation, where performance in terms of state transitions and maintenance are in the order of the microsecond timescale).

Nevertheless, viability of finite state machines for stateful forwarding the wired/switching domain was proposed and proven viable in [25], which is our starting point for the approach presented next. This work proposed OpenState, a generalization of OpenFlow which (perhaps surprisingly) requires very marginal architecture modifications in existing network switches but can "execute" a state machine to dynamically evolve OpenFlow forwarding rules on the basis of packet-level events. OpenState was however limited to support an extremely simple form of state machines, called Mealy Machines. Specifically, OpenState was conceived starting from the observation that the OpenFlow match/action abstraction can be reinterpreted in formal terms as a "***map***" $T:I \rightarrow O$, where $I=\{i_1, \ldots, i_M\}$ is a finite set of ***Input Symbols***, namely all the possible matches which are technically supported by an OpenFlow specification (being irrelevant, at least for this discussion, to know how such Input Symbols' set I is established, and that each input symbol is a Cartesian combination of all possible header field matches), and $O=\{o_1, \ldots, o_K\}$ is a finite set of **Output Symbols**, i.e. all the possible actions supported by an OpenFlow switch. The "engine" which performs the actual mapping $T:I \rightarrow O$ is a standard TCAM. The obvious limit of this OpenFlow "map" abstraction is that the match/action mapping is statically configured, and can change only upon controller's intervention (e.g. via flow-mod OpenFlow commands). However, as observed in [25], an OpenFlow switch can be trivially extended to support a more general abstraction which takes the form of a ***Mealy Machine***, i.e. a Finite State Machine with output, and which permits to formally model ***dynamic*** forwarding behaviours, i.e. permit to change in time the specific action(s) associated to a same match. It suffices to add a further finite set $S=\{s_1, s_2, \ldots, s_N\}$ of ***programmer-specific states***, and use the TCAM to perform the mapping $T:S \times I \rightarrow S \times O$. While remaining feasible on ordinary OpenFlow hardware, such a Mealy Machine abstraction brings about two key differences with respect to the original OpenFlow abstraction. First, the (output) action associated to a very same (input) match may now differ depending on an (input) state s_i in **S**, i.e., the state in which the flow is found when a packet is being processed. Second, the Mealy Machine permits to specify in which, possibly different, (output) state s_o in **S** the flow shall enter once the packet will be processed, hence it permits to **dynamically modify the flow state, and hence apply to the next packet(s) a different forwarding rule, without the intervention of any external controller.**

While quite interesting, this generalization appears still insufficient to permit the programmer to implement meaningful and realistic packet processing stateful applications, which usually require more than just the ability to define and update a state label. Consider for instance a traffic classifier: it requires to extract features from the packet stream comprising the flow (such as average packet size, statistics on the inter-arrival packet time, etc.), and such feature extraction must rely on arithmetic operations not supported neither by the OpenState Mealy Machine abstraction, nor by the switch architecture proposed in [25]. Or take for instance a traffic policer or shaper implemented as a Token Bucket: policing decisions require to manage timers, and depend on conditions applied to the packet statistics, conditions which are not expressible using a Mealy Machine.

We thus propose to exploit a more compelling and complete abstraction (actually already anticipated in [25], but without any idea of how to support it via an actual architecture), the so-called **eXtended Finite State Machine (XFSM) model**.

Table 3: Formal specification of an eXtended Finite State Machine (left two columns) and its meaning in our specific packet-processing context (right column)

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

As summarized in the above Table 3, this model is formally specified 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** are inherited from the Mealy Machine abstraction [25], and 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 a 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 3 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$\rightarrow${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 ui: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×F×I$\rightarrow$S×U×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 in the next section 4.4.2. 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.

4.5.2 Identification of Reusable Functional Blocks for XFSMs: the "update functions"

With reference to the proposed XFSM abstraction, the goal of a functional (nano-)decomposition is to identify which specific set of functions should be provided as basic building block for the programmer to describe meaningful packet processing and traffic control tasks by means of an XFSM. As said above, the XFSM model summarized in Table 3 is formally specified by a 7-tuple **<I,O,S,D,F,U,T>**. Since the set of states **S**, the content of per-flow registers **D**, and the list of transition relations **T** are freely defined by the programmer, these sets are not relevant to any functional analysis. This is clearly not anymore the case for what concerns Input symbols **I** and Output Symbols **O** (actions). However, we posit that (at least for backward compatibility issues) we may reasonably inherit OpenFlow-type matches and OpenFlow actions, respectively, and indeed significant work is currently being made in the ONF community to both extend the OpenFlow's matching facilities as well as the supported action set. Furthermore, with respect to the enabling functions **F**, these are standard Boolean comparison, so programmable support for logical operators and of their combinations is the only requirement to implement such set. It follows that for the purpose of this section, the only remaining target for a decomposition analysis consists in identifying an appropriate set of **update functions U**.

We recall that, at each step, the specific computations that the Update Logic Block must perform are provided by the output of the XFSM transition, and are expressed in the form of a tuple of micro-instructions typical of a microprocessor (e.g., 32 bit instructions). An use-case-based analysis (taking as use cases non trivial traffic processing functions currently performed at line rate and not programmable in an OpenFlow switch due to limitations in the expressiveness of the match/action abstraction - we specifically considered as use cases a port scan detector, a C4.5 traffic classifier based on a binary tree, and a token bucket.) has been therefore made to determine which candidate update functions should be implemented in the Update Logic Block and hence supported by the deployed domain-specific ALUs.

The next two tables report the findings of our analysis. While Table 4 reports basic ALU functions that are already supported by any basic ALU design, Table 5 specifically lists additional domain-specific functions, which we deem useful in traffic control applications, and which would normally require multiple clock cycles if implemented using more elementary operations. Each instruction comprises an `OPCODE`, followed by a variable number of operands that depend on the specific instruction. Input operands (labelled as IN_i in the next Table 4 and Table 5) can be any among the available per flow registries **R_i**, the global variables **G_i**, or the header fields **H_i** provided by the Extractor block. Output operands (labelled as OUT_i) indicate where the result of the instruction must be written (e.g. in a given per-flow register, or in a global variable). In some instructions, one or more of the operands (labelled as IO_i) are both used as input and output. Finally, some instructions may handle constant operands (labelled as IMM).

The most interesting finding is related to the **Domain-specific operations**, which include the online computation of running averages (Avg) and variances (Var), and the computation of exponentially decaying moving averages (Ewma) which can serve the purpose of a moving average, but which can be incrementally computed and do not require to maintain a window of samples. Usage and implementation details about packet/flow specific instructions are provided in Table 5. The Avg() operation stores the number of samples in IO1, and includes a new sample IN1 in the running average IO2. Similarly, the Var() operation stores the number of samples in IO1, the average of the value IN1 in IO2 and the variance in IO3. Finally, the Ewma() operation[2] was included to permit smoothing. It stores the last timestamp (IN1) of a packet in the register identified by IO1, computes the exponentially weighted moving average of the value IN2 using the equation in and stores the result in IO2.

Table 4: ALU basic instruction set

Type	Instructions	Definition
Logic micro-instructions	NOP	do nothing
	NOT	OUT1 ← NOT(IN1)
	XOR, AND, OR	OUT1 ← IN1 *op* IN2
Arithmetic micro-instructions	ADD, SUB, MUL, DIV	OUT1 ← IN1 *op* IN2
	ADDI, SUBI, MULI, DIVI	OUT1 ← IN1 *op* IMM
Bit-level instructions (shift/rotate)	LSL (Logical Shift Left)	OUT1 ← IN1 << IMM
	LSR (Logical Shift Right)	OUT1 ← IN1 >> IMM
	ROR (Rotate Right)	OUT1 ← IN1 *ror* IMM

Table 5: ALU extended (domain-specific) instruction set

Instruction	Description	Definition

Avg()	Running average (frequently used for feature extraction)	IO1 ← IO1+1
		IO2 ← IO2+(IN1-IO2)/(IO1+1)

[2] Being t_k the last sample time, and $x_{k'}$ a new sample occurring at time $t_{k'}$, for simplicity of HW implementation we may *approximate* the exponentially weighted moving average as $m(t_{k'})=m(t_k)\ \alpha$^$(t_{k'}-t_k) + x_{k'}$, and we use α =1/2 to compute powers as shift operations. The intermediate *decay* quantity in the second line is used just for clarity of presentation.

Std()	Running variance (frequently used for feature extraction)	IO1 ← IO1+1
		IO2 ← IO2+(IN1-IO2)/(IO1+1)
		IO3 ← IO3+((IN1-IO2)^2 -IO3)/(IO1+1)

Ewma()	Exponentially weighted Moving average (latest samples are more important than past ones, so as to run-time smooth measurements and 'follow' a temporal behaviour)	IO1 ← IN1
		decay = 1<<(IN1-IO1)
		IO2 ← IO2/decay + IN2

Note that we have on purpose restricted the set of identified RFBs (update functions) to microinstructions, which, besides being apparently useful to the specific network programmer's needs, are also **computationally effective in terms of implementation**. More precisely we have restricted to functions which may be implemented in HW using at most two clock cycles (to this purpose, the definition of each identified update function explicitly shows inputs and outputs, and is provided in a way that makes evident to the reader how these functions might be implemented in HW). It is worth to mention that, similar to the action set in standard OpenFlow, also the specific instruction set provided by the Update Logic Block is independent of our proposed OPP abstraction, i.e., its extension or improvement (e.g. with further dedicated domain-specific instructions) does not affect the overall OPP design.

5 Reusable component analysis based on use cases

D2.1 examined the use cases for Superfluidity, along with their business and technical requirements. We considered the likely impact on Superfluidity's work under several categories:

- Service agility – in terms of service creation, delivery and management. This includes flexible service function chains, as well as resource allocation that is flexible enough for service continuity, mobility and orchestration
- Cost savings – including simplicity for lower operational costs
- Analytics and metrics – metrics are needed for different purposes and so with different temporal granularities, formats etc. One key purpose of metrics is to inform analytics that provide actionable insights for the orchestrator
- Quality of experience – In-line functions need to add only a small amount of latency, either for all applications or else for selected ones. When considering where to physical place (/distribute) of a chain of functions, we need to be aware of the QoE required
- Building blocks – Reusable Functional Blocks (RFBs) are a way of decomposing high-level monolithic functions into reusable components. There need to be tools to validate the properties of functions that are composed of simpler RFBs.
- Orchestration – the challenge is to compose RFBs to achieve 'macro' functions that meet the requirements of latency, scalability and throughput.
- Converged architecture – this is about handling heterogeneity at different levels: the traffic and endpoints; the services and processing; access technologies and scale
- Platform – to handle service agility, computation and network efficiency
- Scalability – one way of addressing this is through a load balancer that spawns up a new 'service handling instance' as required. Scalability requires support from the platform, orchestrator, telemetry system and so on
- Security – functions need to be verified to be secure before deploying them (often called 'secure by design'). As well as this static analysis, dynamic analysis is needed during actual operation to verify behaviour, detect anomalies and so on.

Perhaps the overall architectural requirement emerging from the above list is that of simplicity. Simplicity has of course always been important in networking and computing, but is particularly crucial in the NFV world, in order to fulfil its requirements for agility and dynamism in terms of creating, delivering and managing services. The challenge includes the scope and scale of the devices, vendors, functions and so on which are involved in the overall service.

There is still a lot of on-going research about how to achieve simplicity, but some themes are emerging from our work:

- Flexible reuse. This is the whole basis of the Reusable Functional Blocks described in Section 4.
- Consistency of interfaces, northbound and southbound of layers. There will be numerous functions and vendors involved in the layers of functionality. Part of the way forward is a consistent, vendor-neutral, layer-neutral way of describing the capabilities of the layer and for managing it. This is explored further in the WP3 deliverables.
- Automatic and programmatic control. Ideally we want to eliminate 'humans from the loop'. Humans are error-prone and slow! Network operators today see the issues; the opportunities for error and slowness will be greatly increased in the NFV world. We need tools that automatically plan, test and validate a change. Again, WP3 deliverables discuss this topic further.
- In-line processing needs to scale and be easy to repair. There are several promising directions including stateless network functions and our finite state machines work discussed in Section 4.4.

We now revisit some of the use cases, by identifying the RFBs required in each case.

5.1 Use Case: On–the-fly Monitoring

Use case	*On –the-fly Monitoring*	
Description in a nutshell	*Continuous increase in bandwidth demands means network-wide deployment of DPI will become increasingly expensive and in most cases unsustainable. There is a requirement to enable relatively cheaper monitoring infrastructures.* *Implementation of DPI like systems with VNF(s) will allow two scenarios (a) dynamic deployment of DPI to monitor selected network segments e.g. specific geographical regions, and (b) implementing multiple DPI deployments for disparate virtual customers sharing the same physical infrastructure*	
Reusable Functional Blocks	***Packet Processor***	Service that looks into packet headers to get information on addresses (source and destination), identify protocols, and understands fingerprints of applications

	Load Balancer	Service for distributing packet flows over multiple instances the DPI service chain
	Encryption/Decryption Module	Service to enable analysis of secured packet flows
	Encryption Key Manager	Service that allows encryption keys to be maintained outside of encryption/decryption module. Useful when same key is applied to multiple use cases e.g. same private key for encrypting packet flow.
	Management Module	Service that manages creation, deletion, and upgrade of patterns and fingerprints of applications
	Policy Control/Enforcer	Service that allows deep packet inspection to control access to network
	Switching	Service to switch packet flows in and out of DPI service chain
	Database	Service to store and maintain pattern/fingerprint definitions used for monitoring different applications

5.2 Use Case: S/Gi-LAN Services on the (Mobile) Edge

Use case	***S/Gi-LAN Services on the (Mobile) Edge***
Description in a nutshell	*In today's mobile networks, services involving traffic management/DPI and transport/content optimization have been traditionally deployed on the Internet side of the GGSN/P-GW, i.e. in the S/Gi-LAN.* *Even though the industry recognizes the utility of these services, always in the context of the Mobile Data Tsunami and the desire of operators to differentiate from their competitors on the basis of QoE, deploying such solutions in a scalable fashion is becoming increasingly challenging/costly.* *Moreover, the lack of accurate visibility on RAN conditions makes it very difficult to deliver traffic management and transport/content optimization in a way that achieves balance between network efficiency and QoE.*

	The flattening and "IP-fication" of the network, in the evolution from UMTS to LTE and, eventually, to 5G, provides opportunities of pushing these services into the RAN and towards the Edge of the network.	
Phases	*Instantiate Service*	Traffic management and optimization services will be instantiated in the Mobile Edge micro-data centres (DCs) specified by the operator, or determined to require them
	Access User Plane	The traffic handling services will utilise the Mobile Edge Computing (MEC) infrastructure to acquire access to user plane traffic.
	Create Data Session	Metadata about user plane traffic (Packet Data Protocol sessions) will be held in an in-memory Session Database. Information will include subscriber and device identifiers, respective bearers, associated IPv4/IPv6 addresses, etc.
	Enrich with Control Plane Information	The entries maintained in the Session Database will be enriched with control plane information, leveraging the corresponding MEC APIs. Such information will involve radio-access type per bearer, radio resource allocation, cell load, throughput guidance, link quality, etc. Please note that this info may have to be updated regularly (probably asynchronously) during data session lifetime.
	Track Data Flows	The traffic handling services will identify (L4) data flows. Initially the focus will be on supporting the tracking of flows that use the TCP and UDP transport protocols (over both IPv4 and IPv6). The services will need to maintain state (in-memory) for each of the identified data flows, at least for ones selected to receive traffic optimization.
	Inspect Content	The content (i.e. payload of TCP or UDP packets) of the identified data flows will be inspected in DPI-like fashion to determine the content type. Note that this may have to be stateful, i.e. associating packets to logical streams. The analysis applied may have to be more elaborate than simple signature/pattern matching, e.g. to identify video transferred using an encrypted transport (HTTPS/QUIC).
	Identify Application	To determine e.g. whether it is appropriate to apply traffic regulation to ABR video, the services will have to infer that the consuming application is a media player.

		This occasionally requires logic across data streams of the same subscriber (e.g. when encryption is in effect).
	Evaluate Policies	The services will combine information maintained in the session database with the state kept for each data flow and with the configuration set by the operator to decide which traffic optimization actions must be applied.
	Apply Transport-level Optimization	The services will apply transport (L4) optimization, initially focusing on TCP traffic, to increase the speed, decrease the latency and maximize the efficiency of delivering content to the UE, moreover in a way that takes into account RAN congestion.
	Apply Application-level Optimization	On top of transport optimisation, the services may also apply application-level traffic management schemes. One example is ABR video aware traffic regulation (pacing), which allows the operator to limit the ABR video quality that should be delivered by the mobile network.
	Calculate Metrics, Emit Flow Records	The transport and application layer metrics gathered as part of delivering the above services will be logged and optionally emitted (as IP flow records) to analytics infra.
	State Migration	If the UE moves to a radio service area that is handled by a different MEC DC, the session & data flow state that is maintained on behalf of the UE will need to be migrated, to the extent that the transition is seamless to the user.
	Tear-Down Service	If the traffic optimization services are not required any more in the particular MEC DC, they shall be torn town.
Reusable Functional Blocks	***Mobile Edge Computing (MEC)***	• MEC Platform (Auth) • MEC Platform (Bus) • MEC Platform Management • MEC Enabler (TRF) • MEC Enabler (TRF API) • MEC Enabler (RNIS) • MEC Enabler (RNIS API)
	Load Balancer	If traffic needs to be balanced between multiple service instances, load balancer must support stickiness on a per sub/UE or data session basis.

	Service Chaining	Since these services will be probably deployed together with other traffic inspection or traffic processing services, the Mobile Edge infrastructure needs to provide a traffic steering and/or service chaining function.
	Session Database	Stored information will include subscriber (e.g. MSISDN) and device (e.g. IMSI) identifiers, associated bearers, allocated IPv4/IPv6 addresses, etc. Extra information will include radio-access type (per bearer), radio resource allocation, cell load, throughput guidance, link quality, … In case of multiple service instances within the same DC, Session Database will have to be shared between them.
	TCP/UDP header processing	The traffic processing services must be able to process IPv4/IPv6 (L3) headers and TCP/UDP (L4) headers. This can be part of the kernel, but for reasons of throughput and flexibility, a user-plane protocol stack is preferable.
	Performance Enhancing Proxy	Depending on the transport optimization techniques, the network services will have to modify the timing of packets (traffic regulation/pacing), generate ACKs and handle packet retransmissions on behalf of the content server, or even completely replace the congestion control module (the split-TCP middle-box scenario).
	Connection Tracker	To be able to implement analysis/logic on a per logical flow/stream basis, the network services will include a connection/flow tracker. Depending on stateful-ness of the transport optimization technique, associated state may have to be synchronised across instances, or else service migrations may result into connection resets, etc. For similar reasons, service introduction shall only handle new connections, whereas service tear-down must (reasonably) wait for tracked connections to "starve".
	Deep Packet Inspection function	Protocol and content detection will depend on DPI-like signatures.
	Content Detection Algorithms	Traffic must be inspected (on the basis of data flows) to identify the type of content being transferred. Focus would be on identifying content that can be optimized (e.g. ABR video, even if it is encrypted).

	Application Detection function	Traffic must be inspected (on the basis of data flows) to identify the type of application that is consuming the content. Focus would be on identifying applications that can be optimised (e.g. ABR video players).
	Policy Engine	Combining the data-plane and control-plane inputs with the configuration/preferences of the operator to make decisions and apply actions requires a Policy Engine.
	Transport Optimizer	Transport-layer (L4) optimisation, initially focusing on TCP (see also Performance Enhancing Proxy). Objective is to achieve high network speeds, efficient use of network resources, congestion control and low latency.
	TCP Congestion Control function	For the case of split-TCP middle-boxes, the congestion control algorithm can be replaced. Congestion handlers are usually delivered in the form of reusable modules.
	Video Optimizer	Regulation/pacing of video flows is a traffic management action that helps conserve network resources. This, for example, allows the operator to specify the ABR video quality that should be delivered by the mobile network. It may require cooperation with the Transport Optimizer.
	Metrics Engine	For monitoring service performance, detailed KPI metrics will be generated, both on a per data flow basis, but also time-windowed aggregates. The former can be emitted to a big data analytics RFB (see Flow Record Output below). The latter can be read, in the form of counters, via relevant management plane protocols (e.g. SNMP).
	Instrumentation Gathering	Generation and exporting of instrumentation data is almost always required for traffic handling services, since it enables troubleshooting and root-cause analysis of any issues. This includes the capturing of packet traces, etc.
	Flow Record Output	Detailed per flow metrics can be emitted to network analytics platform in the form of a flow record output.
	Management function	Frontend system (CLI/GUI) for managing, configuring, monitoring and maintaining the optimization services
	Performance collector function	Frontend system (web GUI) for evaluating performance and assuring the benefits of the optimization services

	Big Data Analytics	Composition of functional blocks that receives metrics and counters from the traffic handling services (among other inputs) to implement a Network Analytics solution. Based on the historical metrics, and using trending, forecasting and machine learning techniques, insights extracted can be fed back to the traffic handling services, in terms of policies, towards optimizing them further.

5.3 Use Case: On-the-fly Ad Removal Offloading

Use case	***On-the-fly Ad Removal Offloading***	
Description in a nutshell	*Given the proliferation of online advertisements and the fact that a large number of players based their revenue model on them, it is perhaps unsurprising that ad blockers have become commonplace. While certain useful, they do consume CPU cycles as they scan incoming traffic which results in reduced battery life when talking about mobile devices. In this use case we propose to provide an on-the-fly, virtualized ad blocker service that can be run in edge networks, essentially offloading this functionality from mobile devices in order to increase their battery life.*	
Reusable Functional Blocks	***Strip***	RFB to strip L2 headers from packets
	IP Checker	RFB to filter for IP packets and to check that they are sane (e.g., that the checksum is correct)
	Classifier	RFB used to filter for specific traffic flows (e.g., HTTP traffic in case our intent is to block ads in HTTP flows)
	TCP flow reconstructor	RFB to reconstruct the TCP flow in case ads span multiple packets
	HTTP parser	RFB to parse HTTP traffic
	Ad remover	RFB to search for and remove ads from packets.

5.4 Use Case: Rapid and massively-scalable instantiation of high performance (virtual) application instances

Use case	***Rapid and massively-scalable instantiation of high performance (virtual) application instances for high performance storage applications***

Description in a nutshell	*Current VMs are quite large and heavy, requiring a full Linux O/S to be provisioned. We challenge this model to use much more light-weight instances that can be then used for a multitude of applications that are otherwise inaccessible*	
Reusable Functional Blocks	***Command listener***	Listens to ATAoE commands
	Driver domain	Helper domain that contains drivers for particular hardware devices
	Router	Routes commands either to/from local VMs or adds them to the network queue for remote processing
	Logging system	Send/receive logs from all guest instances
	ACL system	Authenticate packets coming in and send to authorised users. Maintain privacy for packets not addressed to other users
	Queues/Buffers and handlers	Utilise local storage/memory space for capturing data/commands while the other data is being processed
	Packet reordering	Given TCP/IP connection orientated, sequentialising system is not included, need to re-order packets to generate the original stream
	VM controller	Given the Dom0 logic is moved to the MicroVisor need to distribute the control logic and make each domU responsible for some of the control state.

5.5 Use Case: Local Breakout (LBO)

Use case	***Local Breakout***
Description in a nutshell	*Local Breakout intends to avoid user traffic to be sent to the mobile network core, when communication parties are on the same edge network (e.g. eNB). On 3GPP networks, by default, all traffic is terminated on the mobile core (PDP/PDN). However, in many cases, knowing that users are attached on the same edge, communications can be short-cut, making the connectivity more efficient. A similar concept may also be applied to fixed networks.*

Reusable Functional Blocks	*User-Data Plane Interface*	Component integrated with the forwarding path, which intends to allow the MEC platform to access the user-data plane traffic
	MEC Platform (Auth)	MEC Platform component devoted to Service authentication and authorization, namely regarding the access to APIs
	MEC Platform (Bus)	MEC Platform component dedicated to Service communication (Service<–>APIs, Service<->Service)
	MEC Platform Management	Component dedicated to manage the MEC Platform, namely, regarding the API exposure, interaction with authentication/authorization modules, etc.
	MEC Enabler (TRF)	Service Enabler plugged into the MEC platform, which provides user-data plane features
	MEC Enabler (TRF API)	Component that makes an API available for Services to access the user-data plane
	MEC Enabler (RNIS)	Service Enabler plugged into the MEC platform, which provides Radio Network Information Systems (RNIS) features
	MEC Enabler (RNIS API)	Component that makes an API available for Services to access RNIS features
	MEC Enabler (Loc)	Service Enabler plugged into the MEC platform, which provides UE Location features
	MEC Enabler (Loc API)	Component that makes an API available for Services to access UE Location features
	MEC Service LBO	Operator or 3rd party Service devoted to provide Local BreakOut (LBO) services using the MEC System
	MEC Service LBO	Component devoted to manage the Local BreakOut (LBO) Service (associated to the LBO Service)

5.6 Use Case: virtual Convergent Services

Use case	***virtual Convergent Services (vCS)***
Description in a nutshell	*The virtual Home GateWay (vHGW) use case intends to move traditional functions (e.g. firewall, parental control, NAT, etc.) residing on the*

	customers' home to a virtual HGW (vHGW) in the cloud. In a convergent scenario, these services apply both to fixed and mobile environments, providing a convergent desired behaviour, either when the user is at home or using a mobile device. *The virtual Set Top Box component complements the use case, extending the usage of a virtual STB to a multi-screen scenario, simplifying the convergent environment, reducing operator investment and making easier the upgrade and deployment of new services, being accessible to the user from any terminal.*	
Reusable Functional Blocks	*User-Data Plane Interface*	*Component integrated with the forwarding path, which intends to allow the MEC platform to access the user-data plane traffic*
	MEC Platform (Auth)	*MEC Platform component devoted to Service authentication and authorization, namely regarding the access to APIs*
	MEC Platform (Bus)	*MEC Platform component devoted to Service communication (Service<–>APIs, Service<->Service)*
	MEC Platform Management	*Component devoted to manage the MEC Platform, namely regarding the API exposure, interaction with authentication/authorization modules, etc.*
	MEC Enabler (TRF)	*Service Enabler plugged into the MEC platform, which provides user-data plane features*
	MEC Enabler (TRF API)	*Component that makes APIs available to let Services access to user-data plane features*
	MEC Enabler (RNIS)	*Service Enabler plugged into the MEC platform, which provides Radio Network Information Systems (RNIS) features*
	MEC Enabler (RNIS API)	*Component that makes APIs available to let Services access to RNIS features*
	MEC Enabler (Loc)	*Service Enabler plugged into the MEC platform, which provides UE Location features*
	MEC Enabler (Loc API)	*Component that makes APIs available to let Services to access to UE Location features*

	MEC Service vCS	*Operator or 3rd party Service devoted to provide virtual Converged Services (vCS) using the MEC System*
	MEC Service vCS (Mgm)	*Component devoted to manage the virtual Convergent Service (vCS) (associated to the vCS Service)*
	MEC Orchestrator	*Component devoted to orchestrate Services lifecycle (deployment, scaling, migration, disposal, etc.)*
	MEC Orchestrator (Policy)	*Component that hold policies used by the MEC Orchestrator to orchestrate the Services lifecycle*

5.7 Use Case: Late transmuxing (LTM)

Use case	*Late transmuxing (LTM) on the (Mobile) Edge*	
Description in a nutshell	*Instead of using the CDN network only as cache, the CDN (edge) nodes could be used to create requested formats when needed, saving bandwidth and storage within the network, increasing edge resource usage and improving user experience.*	
(LTM) Phases	*Request parsing*	Webserver determines handler based on request type (via extension usually) and maps the request to the local or remote location the content (samples and server manifest) is located.
	Upstream sample fetch	(Libfmp4) When needed audio/video samples are not in the local cache from a similar previous request for another protocol, sample have to be fetched upstream.
	Manifest creation	(Libfmp4) Create the appropriate client manifest for the request (HLS, HDS, HSS or DASH).
	Chunk creation	(Libfmp4) Create the appropriate chunk for the request (HLS, HDS, HSS or DASH).
	Encryption/DRM	(Libfmp4) Encrypt chunk and/or signal DRM in the client manifest based on the server manifest settings.
	Output handover	Return created output to Webserver for delivery.
Reusable Functional Blocks	*Mobile Edge Computing (MEC)*	• MEC Platform (Auth) • MEC Platform (Bus) • MEC Platform Management • MEC Enabler (TRF)

		• MEC Enabler (TRF API) • MEC Orchestrator • MEC Orchestrator (Policy)
	Load Balancer	If traffic needs to be balanced between multiple service instances within the same MEC DC, load balancer must support stickiness on a per sub/UE or data session basis.
	WebServer	Linux platform that can support a web-server e.g. Apache or Nginx (or anything that can handle HTTP requests in general), the LTM instance.
	Libfmp4	The USTR muxing software used in the web-server for LTM.
	Storage	Upstream providing mezzanine audio/video samples, accessible over HTTP (accepting range requests, e.g. S3).
	CMS/Policy	Service for LTM instances to fetch configuration (server manifest) from, configuration can change dynamically. The server manifest controls the client/manifest generation and can be provided by the CMS on a per request basis, so rules may be applied (for instance in relation with the video optimizer).
	AutoScaling	The setup should be able to add and remove LTM instances based on load or other metric, automated.
	Cache priming	Latency between storage and LTM instance should be low enough so cache priming (setting up local caches with content to be ready for expected load) could be employed. Alternatively 'prefetch' (fetching next chunk before time by the webserver to have it cache already) may be employed as well.
	Video Optimizer	Regulation/pacing of video flows is a traffic management action that helps conserve network resources. This, for example, allows the operator to specify the ABR video quality that should be delivered by the mobile network. It may require cooperation with the Transport Optimizer.
	Metrics Engine	For monitoring service performance, detailed KPI metrics will be generated, both on a per data flow basis, but also time-windowed aggregates. The former can be emitted to

		a big data analytics platform (see Flow Record Output below). The latter can be read, in the form of counters, via relevant management plane protocols (e.g. SNMP).
	Monitoring	Monitor stream characteristics on LTM instance (bandwidth, sessions, streams).

6 Conclusions

This document concludes the activities of WP2 of the Superfluidity project.

A functional analysis of different domains/platforms, with the identification of functional elements relevant to 5G scenarios and their decomposition into Reusable Functional Blocks (RFBs) has been carried out.

The first considered domain is the Cloud-RAN (Radio Access Network). A set of RFBs are proposed for Cloud-RAN. Each RFB is characterized by a set of features: its interface and affinity to other RFBs, synchronous or asynchronous characteristic, control or data plane, hardware affinity and the domain where the RFB could be instantiated.

The second domain is the Mobile Edge Computing platform. The main MEC components have been identified and a decomposition in smaller functional blocks has been provided.

The third domain is a generic NFV platform. Some key services like load balancer, state repository and analytics have been studied and decomposed into Reusable Functional Blocks. The Service Function Chaining functionality has been analysed and decomposed into logical components.

The fourth domain is a fixed networking equipment and it has been addressed considering the example of a commercial security appliance (the CISCO ASA). We discussed how this appliance has been decomposed using as Reusable Functional Block the element of an open source router platform called Click.

The fifth domain is an innovative environment, i.e. a packet processing state machine. This environment is referred to as XFSM (eXtended Finite State Machine) and represents the evolution of an OpenFlow forwarding engine, enhanced in order to perform a “stateful” packet processing. The set of elementary “actions” of this packet processing engine has been identified.

To emphasize the benefit of the RFB principle, in the last section of this document a set of use cases has been analysed in order to identify the needed RFBs.

Based on the results reported in this document, the work related to the finalization of the Superfluidity architecture will continue in the WP3.

7 References

[1] “Network Functions Virtualisation (NFV); Use Cases” ETSI GS NFV 001 V1.1.1 (2013-10)

[2] “Network Functions Virtualisation (NFV); Architectural Framework”, ETSI GS NFV 002 V1.1.1 (2013-10)

[3] “Network Functions Virtualisation (NFV); Virtual Network Functions Architecture”, ETSI GS NFV-SWA 001 V1.1.1 (2014-12)

[4] “Network Functions Virtualisation (NFV); Management and Orchestration”, ETSI GS NFV-MAN 001 V1.1.1 (2014-12)

[5] OpenMANO Descriptors https://github.com/nfvlabs/openmano/wiki/openmano-descriptors

[6] “Network Functions Virtualisation (NFV); Use Cases” ETSI GS NFV 001 V1.1.1 (2013-10)

[7] “Network Functions Virtualisation (NFV); Architectural Framework”, ETSI GS NFV 002 V1.1.1 (2013-10)

[8] “Network Functions Virtualisation (NFV); Virtual Network Functions Architecture”, ETSI GS NFV-SWA 001 V1.1.1 (2014-12)

[9] “Network Functions Virtualisation (NFV); Management and Orchestration”, ETSI GS NFV-MAN 001 V1.1.1 (2014-12)

[10] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, “Fog computing and its role in the internet of things”, 1st SIGCOMM workshop on Mobile cloud computing (MCC), August 2012, Helsinki, Finland

[11] “Mobile-Edge Computing – Introductory Technical White Paper”, https://portal.etsi.org/Portals/0/TBpages/MEC/Docs/Mobile-edge_Computing_-_Introductory_Technical_White_Paper_V1%2018-09-14.pdf

[12] K.Chen et al., “C-RAN: The Road Toward Green RAN”, White Paper, China Mobile Research Institute (2011).

[13] E. Kohler, R. Morris, B. Chen, J. Jannotti and M. F. Kaashoek, "The Click Modular Router Project," ACM Transactions on Computer Systems, vol. 18, no. 3, pp. 263-297, 2000.

[14] GNU Radio – The free and open software radio ecosystem, Available: http:/gnuradio.org/

[15] Gerardo Garzia, “Introduction to OpenMANO”, presentation on slideshare.net http://www.slideshare.net/movilforum/introduction-to-open-mano (Published on Jun 16, 2015)

[16] openmano descriptors (produced by the Open Mano project) https://github.com/nfvlabs/openmano/wiki/openmano-descriptors

[17] UCLA CS Read, "The Click Modular Router Project," 17 January 2012. [Online]. Available: http://www.read.cs.ucla.edu/click/elements. [Accessed 20 October 2015].

[18] UCLA CS Read, "The Click Modular Router Project," 01 December 2009. [Online]. Available: http://www.read.cs.ucla.edu/click/packages. [Accessed 07 October 2015].

[19] SDNCentral LLC, "SDX Central," [Online]. Available: https://www.sdxcentral.com/nfv-sdn-products-directory/. [Accessed 05 October 2015].

[20] NGMN alliance, “NGMN 5G white paper”, 17 February 2015, Available online: https://www.ngmn.org/uploads/media/NGMN_5G_White_Paper_V1_0.pdf

[21] Sam Newman, “Building Microservices”, February 2015, O’Reilly] Media

[22] FLAVIA EU project (FLexible Architecture for Virtualizable future wireless Internet Access), http://www.ict-flavia.eu/

[23] N. Feamster, J. Rexford, and E. Zegura, “The road to SDN: an intellectual history of programmable networks“, ACM SIGCOMM Computer Communication Review, vol. 44, no. 2, pp. 87--98, 2014.

[24] Tinnirello, I.; Bianchi, G.; Gallo, P.; Garlisi, D.; Giuliano, F.; Gringoli, F., "Wireless MAC processors: Programming MAC protocols on commodity Hardware", INFOCOM, 2012, pp.1269-1277.

[25] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, “OpenState: Programming Platform-independent Stateful OpenFlow Applications Inside the Switch” ACM SIGCOMM Computer Communication Review, vol. 44, no. 2, pp. 44–51, 2014.

[26] Small Cell Forum, Small cell virtualization functional splits and use cases, Document 159.05.1.01 (2015)

[27] METIS Deliverable D6.4 “Final report on architecture”, January 2015. Available: https://www.metis2020.com/wp-content/uploads/deliverables/METIS_D6.4_v2.pdf

[28] iJOIN Project. “D5.3-Final Definition of iJOIN architecture”, April 2015. Available: http://www.ict-ijoin.eu/wpcontent/uploads/2012/10/D5.3.pdf

[29] Intel, Snap – the open telemetry framework, https://github.com/intelsdi-x/snap

[30] https://tools.ietf.org/html/rfc3963

[31] Superfluidity, Deliverable D2.1, “Use cases, technical and business requirements”, March 2016

[32] http://www.etsi.org/technologies-clusters/technologies/mobile-edge-computing

[33] IETF SFC Architecture (RFC 7665). Available at https://tools.ietf.org/pdf/rfc7665.pdf