H2020 5G-CORAL Project

Grant No. 761586

D3.2 – Refined design of 5G-CORAL orchestration and control system and future directions

Abstract

This deliverable presents the refined design of the 5G-CORAL orchestration and control system, namely OCS, with emphasis on orchestration and federation. A distributed resource orchestrator is described along with an optimization algorithm for volatile and federated environments. Next, this document analyses the monitoring and live procedures needed by the identified 5G-CORAL use cases. Finally, it presents an experimental validation of some selected OCS features. Lessons learnt and future directions regarding the OCS conclude the document.

Document properties

Document number	D3.2
Document title	Redefined design of 5G-CORAL orchestration and control system and future directions
Document responsible	UC3M
Document editor	Milan Groshev
Editorial team	Milan Groshev, Luca Cominardi
Target dissemination level	Public
Status of the document	Stable
Version	1.0

List of contributors

Partner	Contributors
ADLINK	Gabriele Baldoni
IDCC	Giovanni Rigazzi
ITRI	Samer Talat, Ibrahiem Osamah, Gary Huang, Chen Hao Chiu
NCTU	Li-Hsing Yen
TELCA	Aitor Zabala Orive, Pedro Bermúdez
UC3M	Luca Cominardi, Milan Groshev, Kiril Antevski, Jorge Martin-Pérez, Sergio Gonzáles, Nuria Molner

Production properties

Reviewers	Li-Hsing Yen, Luca Cominardi, Giovanni Rigazzi, Samer Talat, Aitor Zabala Orive, Carlos Guimaraes, Alain Mourad

Document history

Revision	Date	Issued by	Description
1.0	31 May 2019	UC3M	Public release

Disclaimer

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

3GPP	3rd Generation Partnership Project
AAU	Active Antenna Unit
AMQP	Advanced Message Queuing Protocol
AP	Access Point
API	Application Programming Interface
App	Application
AR	Augmented Reality
ARIA	Agile Reference Implementation of Automation
ARM	Acorn RISC Machine
ASE	Autoscaling Engine
AWS EC2	Amazon Elastic Compute Cloud
BSS	Business Support System
CAM	Cooperative Awareness Message
CAPEX	Capital expenditure
CD	Computing Device
CHA	Capacitated house allocation
CLAMP	Platform for designing and managing control loops.
CPU	Central Processing Unit
CRIU	Checkpoint and restore in user space
D2D	Device to Device
DA	Adapted deferred acceptance
DA-T	Adopted deferred acceptance with transfer
DB	Database
DC	Data Centre
DENM	Decentralized Environmental Notification Message
CLI	Command-line interface
DNS	Domain Name System
EAP	Extensible Authentication Protocol
EBS	Amazon Elastic Block Store
eCDF	Experimental Cumulative Density Function
EFS	Edge and Fog computing System
EMS	Element Management System
ENI	Experiential Networked Intelligence
EPA	Enhanced Platform Awareness
ETSI	European Telecommunications Standards Institute
FPGA	Field-programmable gate array
FSM	Finite State Machine
FS	File System
GPIO	General Purpose Input/Output
GRE	Generic Routing Encapsulation
GTP	GPRS Tunnelling Protocol
GUI	Graphical User Interface
HDD	Hard Disk Drive
HST	High Speed Train
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
HV	Hypervisor
HW	HardWare
I/O	Input/Output
IaaS	Infrastructure-as-a-Service
ID	Identifier
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IoT	Internet of Things
IP	Internet Protocol
JSON	JavaScript Object Notation
K8	Kubernetes

KDL	Kullback-Leibler distance
KPI	Key Performance Indicator
KVM	Kernel-based Virtual Machine
LCM	Life Cycle Management
LRF	Local resource first
LSF	Local service first
LSO	Local service only
LTE	Long Term Evolution
LXC	LinuX Containers
LXD	Next generation system container manager
MAC	Media Access Control
MANO	MANagement and Orchestration
MEC	Multi-access Edge Computing
MEF	Metro Ethernet Forum
MP	Maximal profit
MQ	Messaging queue
MQTT	Message Queuing Telemetry Transport
MUSIC	Multi-site State Coordination Service
NAS	Network Attached Storage
NFS	Network File System
NFV	Network Function Virtualisation
NFVO	Network Function Virtualization Orchestrator
NGINX	Open source software for web serving
NS	Network Service
NSD	Network Service Descriptor
OASIS	Open standards. Open source.
OBU	On Board Unit
OCS	Orchestration and Control System
ONAP	Open Network Automation Platform
ONF	Open Networking Foundation
OOM	ONAP Operation Manager
OPEX	Operating expenditure
OPNFV	Open Platform for NFV
OS	Operating System
OSM	Open Source MANO
OSS	Operations Support System
OVS	Open Virtual Switch
P2P	Peer to Peer
PaaS	Platform-as-a-Service
pc	Pre-copy
PNF	Physical Network Functions
PoS	Point of Sale
QoE	Quality of Experience
QoS	Quality of Service
RAM	Random Access Memory
RAT	Radio Access Technologies
REST	Representational state transfer
RO	Resource Orchestrator
ROS	Robot Operating System
RPC	Remote Procedure Call
RTT	Round-trip time
sc	Stop-and-copy
SD-WAN	Software Defined Wide Area Network
SDC	Service Design and Creation
SDN	Software Defined Network
SDK	Software Development Kit
SFC	Service Function Chaining
SHA	Secure Hash Algorithm
SLA	Service Level Agreement
SO	Stack Orchestrator
SSD	Solid State Disk

TCP	Transmission Control Protocol
TMPFS	Temporary file system
TOSCA	Topology and Orchestration Specification for Cloud Applications
UDP	User Datagram Protocol
UE	User Equipment
URI	Uniform Resource Identifiers
URL	Uniform Resource Locator
USB	Universal Serial Bus
UUID	Universally Unique IDentifier
VDU	Virtualisation Deployment Unit
VIM	Virtualisation Infrastructure Managers
VL	Virtual links
VLAN	Virtual Local Address Network
VM	Virtual Machine
VNF	Virtual Network Functions
VNFC	VNF Components
VNFD	VNF Descriptors
VNFM	VNF Manager
VPN	Virtual Private Network
VR	Virtual Reality
VVP	VNF Validation Program
VXLAN	Virtual Extensible Local Area Network
WAN	Wide Area Network
WP	Work Package
XML	eXtensible Markup Language
YAML	Human-readable data-serialization language
ZSM	Zero touch network & Service Management

Executive Summary

This second and last deliverable from 5G-CORAL Work Package 3 focuses on the refined design of the Orchestration and Control System (OCS). It first identifies the gaps of existing orchestration systems with regards to the edge and fog environment. The resulting analysis serves as the basis for designing next the 5G-CORAL OCS targeted at filling all the identified gaps. Emphasis is put on state distribution and federation optimization, including optimal placement of functions and applications in a volatile environment. An experimental validation of selected features of the refined OCS design is also presented.

The key achievements in this deliverable are highlighted below:

- Analysis and comparison of existing VIM and Orchestrators against 5G-CORAL OCS requirements for edge and fog environments;
- Design of a distributed VIM and Orchestrator leveraging a distributed key-value store to cope with resource-constrained devices and error-prone environments [1];
- Proposal and validation of a descriptor, namely EFS Stack, enabling zero-touch deployment at orchestration level;
- Proposal and evaluation of a placement algorithm addressing the volatility of the resources comprising the virtualization and computing fabric;
- Characterization and analysis of monitoring requirements and triggered procedures at OCS level for dynamically adapting to varying environment conditions [2][3];
- Characterization and evaluation of OCS federation including pricing insight, federation formation dynamics, and advanced resource provisioning;
- Experimental assessment of selected OCS features, such as automated deployment, federation establishment, container-based migration, and network-based D2D communication establishment;
- Refactoring of fog05 to act as distributed VIM following the state distribution paradigm proposed in this deliverable [4];
- Prototyping and publication as open source of f0rce (i.e., fog orchestration engine) implementing the distributed orchestrator paradigm proposed in this deliverable [5].

1 Introduction

The overall concept of the 5G-CORAL Orchestration and Control System (OCS), including its benefits, challenges, requirements, and architecture was introduced and described in depth in the first deliverable of WP3 (see D3.1 [6]). In brief, the OCS has the following tasks: (*i*) to build and maintain the EFS, by enabling automatic discovery of available EFS resources, integrating and federating them into a unified hosting environment, despite their heterogeneity, multiple owners and volatility (e.g., on the move); (*ii*) to manage the lifecycle of the EFS Functions, Applications, and Service Platform, by performing their instantiation, live migration and scaling to dynamically adapt to changing requirements and monitoring information.

The first deliverable of WP3 [6] mainly focused on the support of heterogeneous and dynamic resources, dynamic migration, monitoring, and third-parties interaction with the OCS. This resulted in the initial design of some of the OCS components, namely VIM and EFS Entity Descriptor. Moreover, D3.1 proposed a baseline solution for resource discovery and integration across multiple access technologies, such as IEEE 802.11, 3GPP, Bluetooth/ZigBee, and Ethernet. Finally, D3.1 introduced the concept of resource federation and three federation models.

Departing from those findings, this second deliverable elevates the focus from the VIM up to the EFS Orchestrator and presents validation results for the OCS components. The different scope between D3.1 and D3.2 (the present document), as well as the OCS components being investigated, is highlighted in Table 1-1. Specifically, D3.2 addresses the design of the EFS Resource Orchestrator and the EFS Stack Descriptor in Section 2. This includes a distributed key-value store and a placement algorithm suited for volatile environments. Then, Section 3 analyses the monitoring requirements and identifies the necessary OCS procedures for each of the 5G-CORAL use cases, resulting in a novel container-based migration mechanism. Section 4 proposes a baseline solution for resource federation and allocation, including pricing insight and in-sourcing and out-sourcing of resources between distinct administrative domains. Section 5 presents the experimental validation of some of the OCS features, such as automated deployment enabled by the EFS Stack, federation instantiation, live migration, and network assisted Device-to-Device (D2D) communication. Finally, Section 6 presents the lessons learnt while Section 7 draws the conclusions and future directions for the OCS.

TABLE 1-1: SCOPE OF D3.1 AND D3.2 DOCUMENTS

	D3.1	D3.2
Architecture	Design of the overall OCS architecture, including the OCS components and interfaces	No refinement at architectural level, refinement done at OCS component level (e.g., VIM, Orchestrator, etc.)
VIM	Design of Finite State Machine (FSM) abstraction for EFS Entities, EFS Entity Descriptor, support of dynamic and heterogeneous resources and virtualization substrates	Experimental validation of heterogeneous virtualization substrates, integration with the EFS Resource Orchestrator
EFS Manager	Defined scope, interaction and interfaces	Monitoring procedures addressed for each use case with focus on migration and scaling of EFS Entities
EFS Orchestrator	Defined scope, interaction and interfaces	Design and validation of EFS Resource Orchestrator with focus on state distribution, resource federation and federation algorithms
EFS Stack Descriptor	Defined scope and high-level information model	Design and experimental validation of EFS Stack Descriptor
EFS Entity Descriptor	Defined scope and detailed information model	Refinement, integration in the EFS Stack Descriptor and experimental validation

2 Refined design of the OCS

In this section, we first provide an overview of the OCS architecture to help the reader to better understand this document. Next, we evaluate the capabilities of the most popular orchestration solutions suitable for 5G-CORAL and several conclusions are drawn in Section 2.2. The outcome of this study is then used to support the design of the EFS Stack Orchestrator and EFS Resource Orchestrator in Section 2.3. Finally, Section 2.4 describes and validates the placement algorithms of the EFS Resource Orchestrator.

2.1 Overview of 5G-CORAL architecture and OCS components

The following paragraphs summarise the main concept and components of the OCS as introduced in D3.1 [6]. While no architectural refinement is performed in this document, the internal design of some of the OCS components is further refined in the following sections.

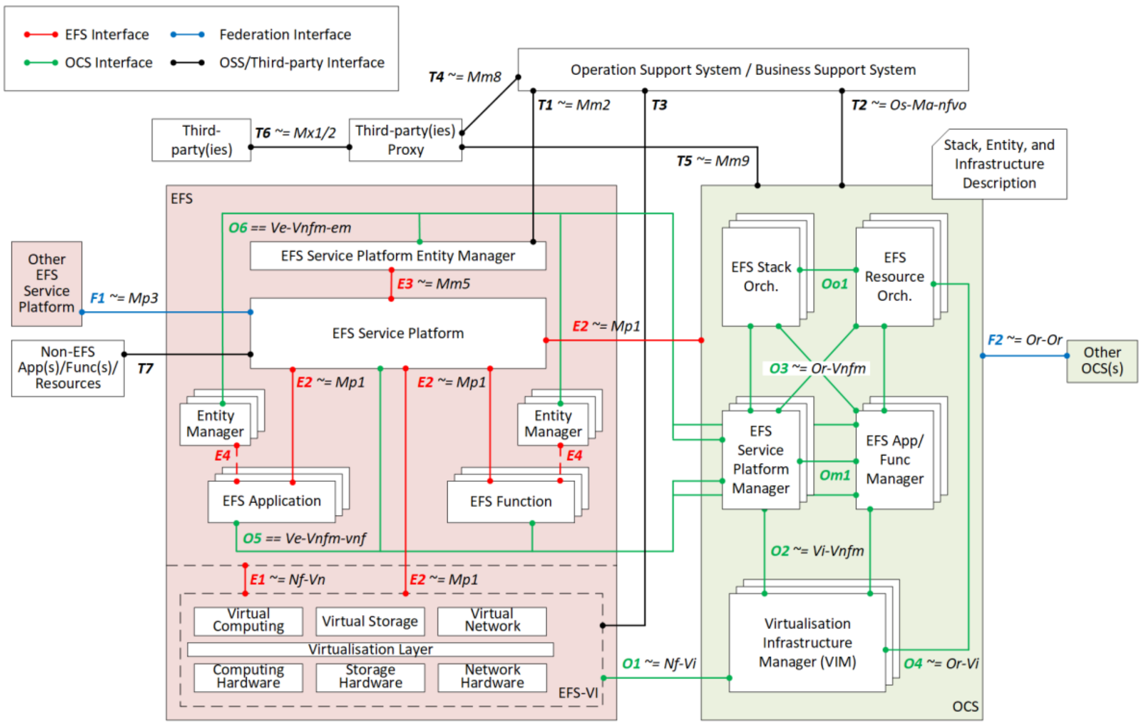

FIGURE 2-1: 5G-CORAL SYSTEM ARCHITECTURE

Figure 2-1 shows the 5G-CORAL system architecture with the two main components:

- **Edge and Fog computing System (EFS):** an EFS is a logical system subsuming Edge and Fog resources that belong to a single administrative domain. An EFS provides service platforms, functions, and applications on top of available resources, and may interact with other EFS domains. See D2.1 [7] and D2.2 [8] for additional information on EFS.
- **Orchestration and Control System (OCS):** an OCS is a logical system in charge of composing, controlling, managing, orchestrating, and federating one or more EFS(s). An OCS comprises Virtualisation Infrastructure Managers (VIMs), EFS managers, and EFS orchestrators. An OCS may interact with OCSs of other administrative domains.

The OCS components, which are shown from bottom to top in Figure 2-1, are:

- A **Virtualisation Infrastructure Manager (VIM)** comprises the functionalities that are used to control and manage the interaction of the service platforms, functions, and applications with the edge and fog resources under its authority;

- An **EFS Manager** is responsible for the lifecycle management (e.g. instantiation, update, scaling and termination) of the service platforms, functions, and applications in the EFS;
- An **EFS Orchestrator** is in charge of the orchestration and management of edge and fog resources and composing the EFS. An EFS Orchestrator comprises an EFS Resource Orchestrator and an EFS Stack Orchestrator. An **EFS Resource Orchestrator** supports accessing the edge and fog resources in an abstracted manner independently of any VIM. An **EFS Stack Orchestrator** is responsible for the EFS Stack lifecycle management operations (e.g. instantiation, update, query, scaling and termination);
- An **EFS Stack** can be viewed architecturally as a forwarding graph of functions and/or application interconnected by supporting edge and fog resources and/or service platforms. An EFS Stack extends the ETSI NFV Network Services by also considering interconnections with applications and service platforms;
- An **EFS Stack Descriptor** extends the ETSI NFV Network Service Descriptor by also considering applications and service platforms in addition to network functions. It describes the requirements and interconnections of one or more EFS Functions and EFS Applications between them or with the EFS Service Platform;
- An **EFS Entity Descriptor** extends and combines ETSI NFV VNF and ETSI MEC App descriptors to uniformly describe the various characteristics of EFS Functions, EFS Applications, and EFS Service Platform. EFS Entity Descriptors are referenced and included into an EFS Stack Descriptor.

2.2 Analysis of existing orchestrators

In this section, we explore some of the most prominent orchestration solutions emerged from open-source communities, research projects and standardization groups, with the goal of assessing their benefits and their limits with respect to the 5G-CORAL framework. For the sake of completeness, in Appendix 9 we provide an exhaustive review of each orchestrator. We first introduce their key capabilities and highlight the specific features required in 5G-CORAL that are not yet supported. Also, we report in more details whether functional and non-functional requirements (see D3.1 [6]) are met or not. Also, Table 2-1 and Table 2-2 help the reader to understand how the reviewed orchestrators satisfy the 5G-CORAL requirements as well as to quickly identify which features are fully or partially supported.

Among the existing capabilities suitable for 5G-CORAL, we note that auto-scalability and fault-management are well supported by the most popular orchestrators, such as Open Source MANO, ONAP and OPNFV, as well as monitoring plugins and the presence of a pub/sub-based event engine, which are relevant features in 5G-CORAL. Moreover, some orchestrators, such as ONAP, provide support for complex lifecycle operations, including healing, scaling and recovery policies that can be defined at design time. It is also worth noting that Network Service Descriptor (NSD)[1] onboarding and basic validation are extensively supported by most of the orchestrators reviewed. In terms of non-functional requirements, we point out that large-scale deployment and multi-tenant support feature in all the solutions, which are key capabilities in 5G-CORAL.

By contrast, federation is not yet supported by most of the orchestrators. When supported, like in the Kubernetes (K8s), it relates to the federation of multiple instances of the same orchestrator. Similarly, dynamic resource discovery and dynamic migration are not supported, which are crucial operations within the 5G-CORAL framework to ensure automatic service deployment and zero-touch management. As an example, ONAP does not currently provide clear guidelines on

[1] Network Service Descriptor (NSD) is the terminology used in ETSI NFV [10] to describe a graph of Virtual Network Functions and their requirements. In 5G-CORAL we use the term EFS Stack to encompass and unify both ETSI NFV and ETSI MEC descriptors.

how to discover and add physical resources at runtime, neither does Cloudify, which does not track the resource availability in the managed infrastructure. Ultimately, the lack of such features and capabilities raises the need for enriching 5G-CORAL and incorporating new features into the framework, as it will be described in detail in the following sections.

TABLE 2-1: OVERVIEW OF 5G-CORAL FUNCTIONAL REQUIREMENTS AND RELATED SUPPORT PROVIDED BY THE ORCHESTRATORS

Functional Requirement	OSM	Open Baton	ONAP	Cloud. ify	OPNFV	Apache ARIA	K8s
Support of harvesting computing capabilities from low-end resources	Yes	No	No	Partial	Yes	Partial	No
Support of harvesting computing capabilities from mobile resources	Partial	No	No	Partial	Partial	Partial	No
Support of discovery, configuration, monitoring, allocation, etc. of relevant hardware capabilities	Yes	Partial	No	Yes	Partial	Partial	Yes
Support of integration including at runtime of heterogeneous resources in terms of software and hardware capabilities	Yes	Yes	Yes	Yes	Yes	Partial	Partial
Support of federation including at runtime of OCS components	No	No	Partial	No	No	Partial	Partial
Support of the interworking with resources external to the OCS	Yes	Yes	Yes	Yes	Yes	Partial	Partial

TABLE 2-2: OVERVIEW OF 5G-CORAL NON-FUNCTIONAL REQUIREMENTS AND RELATED SUPPORT PROVIDED BY THE ORCHESTRATORS

Non-Functional Requirement	OSM	Open Baton	ONAP	Cloud. ify	OPNFV	Apache ARIA	K8s
Support of deployment of OCS on low end devices	No	No	No	No	No	Partial	No
Support of deployment of OCS on mobile devices	No	No	No	No	No	Partial	No
Availability and self-healing mechanisms in error-prone environments	No	Partial	Yes	Yes	Partial	Partial	Yes
Support of large deployments in terms of number of resources and geographic areas	Yes	Yes	Yes	Yes	Yes	Partial	Yes
Support of plugins for extensibility	Yes	Yes	Yes	Yes	No	Partial	Partial
Capability to adapt to workload changes by provisioning and de-provisioning resources in an automated manner	No	Partial	Yes	No	Partial	Partial	Yes
Support of multiple tenants participating and co-existing in the same environment	Yes	Yes	Yes	Yes	Yes	Partial	Yes

2.3 Design of a distributed OCS

As it can be seen from Table 2-2, the 5G-CORAL non-functional requirements for the OCS are far from being met by current orchestrators. Particularly, today's implementations are tailored to datacentre environments where resources are fixed, and high bandwidth is available. However, this assumption is not true for fog and edge environments where heterogeneous resources are geographically distributed. This makes it difficult to support OCS deployment on low-end devices which may also be mobile and distributed across multiple locations. In D3.1 [6], we introduced the development of an OCS prototype (i.e., VIM) which had started under the name of *fog05* and the initial code was released as open source on GitHub [4]. D3.1 focused on defining a plugin-based architecture and a Finite State Machine (FSM) abstraction for the EFS Entities. In this deliverable, we tackle the problem of how to distribute the various OCS components, including the VIM (e.g., fog05) and the newly designed EFS Resource and Stack Orchestrators. The prototype of the orchestrators, dubbed as *f0rce* (i.e., fog orchestration engine), is published as open source on GitHub [5].

The key idea for enabling the OCS deployment on low-end and mobile device is to move away from the monolithic and datacentre-focused implementation. That is, the VIM and the Orchestration should be decomposed in atomic functionalities and their internal state distributed across the network. In this way, each resource can contribute to the overall OCS functionalities and the same functionalities can be replicated within the network to provide increased fault-tolerance and availability. State distribution can be thought as a distributed database where the information meaningful for the OCS is stored. However, in contrast to classical database design where data is meant to be persistently stored, state distribution in our case relates more to the capability to store the runtime information useful for any OCS procedures. In practical terms, this can be reduced to storing the internal variables of OCS in such a way that they can be read and written anywhere. Therefore, we consider a distributed key-value store as the most suitable choice for distributing the OCS state information.

2.3.1 Distributed key-value store

Key-value stores work in a very different fashion from the better-known relational databases (e.g., MariaDB, MySQL, etc.). Relational databases pre-define the data structure in the database as a series of tables containing fields with well-defined data types. In contrast, key-value stores treat the data as a single opaque collection (e.g., associative arrays or hash tables), which may have different fields for every record. This offers considerable flexibility and more closely follows modern concepts like object-oriented programming. Inherently, a distributed key-value store is a key-value store whose data is not stored in a single location but rather at different locations across the network. Several approaches exist for distributing the store: full replica, partial replicas, on-demand, etc., differing on the amount of data being replicated and on the timeliness of replication. However, the traditional approaches and existing implementations are not well suited for constrained, mobile and very distributed resources as in the edge and fog environment. This is because they have been designed with a data-centre infrastructure in mind.

The approach adopted in 5G-CORAL is a distributed key-value store characterised by eventual consistency, scalability and location transparency. This allows to share data across distinct devices along the cloud-to-thing continuum and across different technologies and networks. As a result, the OCS is provided with a unified access to those data so that each portion of the OCS only needs to retain, store and manage the status information that are local to the specific node. That is, data is globally accessible without requiring local replication as in traditional key-value stores. In this way, the OCS as a whole can access data that are locally managed by each portion of the OCS without the need to know where the data resides, providing location transparency. In order to fully support such characteristic, the distributed key-value store needs to leverage a transport protocol tailored for such scenario. In 5G-CORAL we consider Zenoh [9] as reference transport protocol for the distributed key-value store. Additional information, as well as performance evaluation, can be found in D2.2 [8]. Finally, eventual consistency informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item return the last updated value. This allows the OCS to keep operating (to some extent) upon failure of some of its components.

The distributed key-value store organizes the data in a tree structure following the Uniform Resource Identifier (URI) definition [11]. Hence, the key of each value has the following format:

$$/s_1/s_2/\ldots/s_n$$

As an example, let's consider the key "/ocs/vim/id1/entity/id2/info" as to contain the information regarding the entity with *id2* under the control of vim *id1*. By using the URI format, it is possible to use wildcard and queries when accessing the data, thus enabling a fine control on the data. For example, the key "/ocs/vim/id1/entity/*/info" can be used to access the data of all the entities under the control of vim *id1*. Each value is defined as a tuple:

$$v = < e, c, t >$$

Where e is the encoding, c represents the content and *t* is a logical timestamp for ordering. In addition, a pub/sub mechanism is considered for notification to promptly react to changes in the internal state of OCS. For instance, the EFS Resource Orchestrator can subscribe to the monitoring information of a given resource and being notified whenever the RAM consumption is updated. Finally, a Remote Procedure Call (RPC) is considered in order to allow different OCS components and portions to interact with each other without the need to store the data in the network.

Finally, the following primitives are defined:

- **Put, update, remove, get:** data are published via put/update. OCS components can then query the data with get. Finally, remove deletes the data from the data store;
- **Subscribe/unsubscribe:** an OCS component can subscribe to specific keys (including wildcards) and being notified whenever the value associated to that key changes. Unsubscribe removes the subscription;
- **Register_eval, unregister_eval, eval:** OCS components can expose functionalities to other components by registering specific functions for RPC. OCS components can remotely execute functionalities via the eval primitive.

In the following section, we report few examples on how to use the distributed key-value store concept and primitives to implement the VIM and Orchestrator, namely fog05 and f0rce.

2.3.2 Distributed VIM

Each EFS resource participating in the distributed VIM is requested to run an *agent* for the management of the node. Specifically, such agent takes care of advertising the node to the other nodes composing the distributed VIM, instantiating and terminating the EFS Entities on the node, etc. Moreover, the agent needs to keep and share the state of the EFS resources in such a way that all the nodes composing the distributed VIM can cooperate and operate as a single logical entity. Figure 2-2 illustrates the EFS resource (i.e., node)[2], the agent and the various distributed storages envisaged for enabling a distributed VIM along the cloud-to-thing continuum. Three types of storage are considered for the VIM:

[2] The agent running on each node may simultaneously support multiple virtualization technologies. The necessary support at VIM level is provided by configurable plugins which expose a Finite State Machine (FSM) abstraction for EFS Entities. More information is available in D3.1 [6].

- **Local storage:** this storage is used to store the status of the compute node locally on the node itself. This storage is used for communication between the agent and various plugins running on the node[3]. This storage also stores the configuration of the node and the real time status of the node. This storage is not shared on the network.
- **Constrained storage:** this storage is a special case of the local storage and it is needed by those resource-constrained devices uncapable of running the VIM agent (e.g., microcontrollers). In this case, a powerful node can host the local storage of a third node which in turn it is updated it over the network (e.g., using TCP, Zenoh, etc.). In practical term, the powerful node acts as a proxy for the constrained device in similar fashion as happens today with an IoT gateway.
- **Global storage:** this storage is shared across the network and stores the global state of the whole VIM. It is worth mentioning that this storage as a whole includes the complete state of the VIM. However, each node is not required to store locally all the state. Indeed, each node contains a portion of the overall state which can be combined with other portions from the other nodes to form the global state.

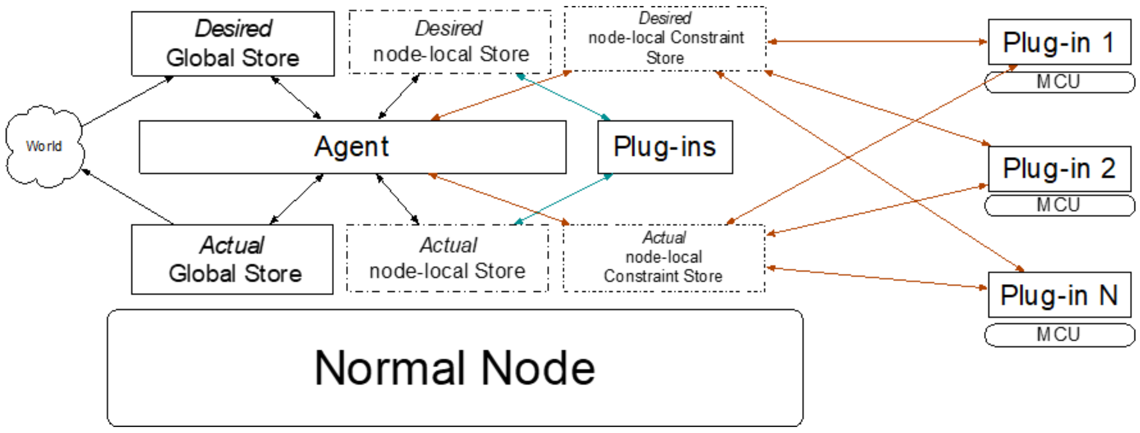

FIGURE 2-2: VIM AGENT

Summarising, each compute node that is powerful enough to run an agent will have its own instance of the distributed key-value store. Such instance takes care of the *node-local* information and portion of the *global* information. In the case of constrained devices, the *agent* and the *store* are remotely hosted on a third node acting as a proxy. Finally, one of the main duties of the agent is to bridge information across the different storages in the VIM in such a way that information can be read from the *global storage* and written on the *local storage* and vice versa.

To tackle the volatility and the errors that could occur in a distributed environment, each of the three types of storage is decomposed in two sub-storages:

- **Actual storage:** it stores the actual stable state;
- **Desired storage:** it stores the next desired stable state.

This separation allows to implement atomic transactions in the VIM. An atomic transaction is an indivisible and irreducible series of operations such that either all occur, or nothing occurs. By doing so, the consistency on the VIM global state is guaranteed in case of errors. Indeed, in the unfortunate case of some operation failing, all the VIM components and nodes can roll back to the state stored in the actual storage. Only when all the operations succeed, the actual storage is then updated. Using an analogy coming from control theory, the desired storage can be seen as

[3] Details about the plugin-based architecture of the VIM to support multiple virtualization technologies can be found in D3.1 [6].

the set point (i.e., the state to achieve) and the actual storage as the expected exit of the system. Combining the three main storages with the two sub-types, it turns out that the distributed VIM consists of a total of six storages:

- **Desired global storage:** it stores the desired state of the whole VIM. It can be used to send requests to the VIM;
- **Actual global storage:** it stores the actual state of the whole VIM. It can be used to retrieve information from the VIM;
- **Desired node-local storage:** it stores the desired state of a single node. This store is hidden inside the VIM and is used for VIM operations. It can be written only by the VIM.
- **Actual node-local storage:** it stores the actual state of a single node. It is internal to the VIM and it can be written only by the VIM agent running on the node hosting it;
- **Desired node-local constrained storage:** like the *desired node-local* but for constrained compute nodes;
- **Actual node-local Constrained Storage:** like the *actual node-local* but for constrained computing nodes.

After having described how the state is distributed in the VIM, the following describes how data is organised in the distributed key-value store. Specifically, it describes the URI structure used by the VIM to archive data separation between the different storages, minimizing data replication, and support multi-tenancy. Namely, in order to minimize differences between the six storages, the VIM adopts a three tree structures:

- **Global Tree:** for both *actual/desired global* storages;
- **Local Tree:** for both *actual/desired* node-local storages;
- **Constrained Local Tree:** for both *actual/desired* node-local constrained storages.

This allows to easily switch between the *actual* and *desired* storages facilitating the development and making the information in the different storages semantically coherent.

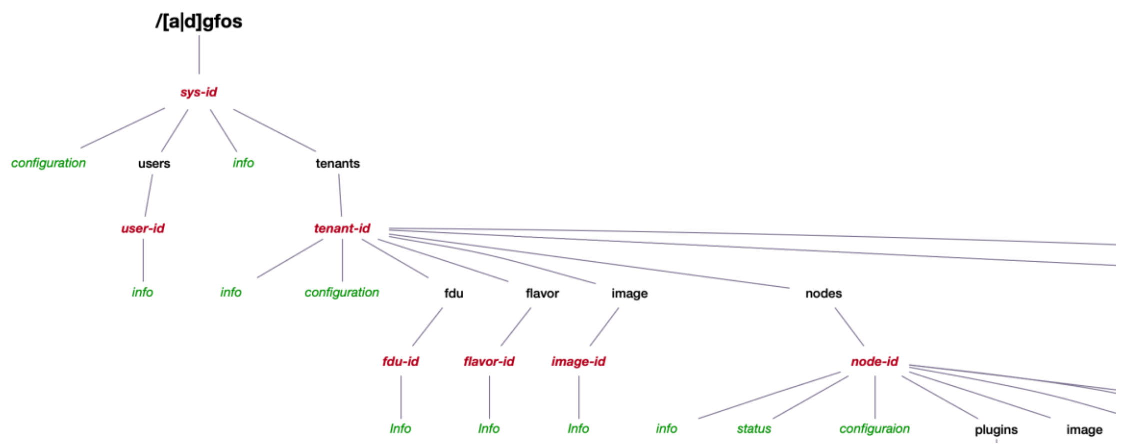

FIGURE 2-3: DISTRIBUTED VIM GLOBAL STORAGE URI TREE

Figure 2-3 depicts part of the URI structure for the *global storage*[4]. It is possible to see an organization in systems, that can be mapped to administrative domains and tenants in order to facilitate the multi-tenancy support. It is worth highlighting that portions of this tree are meant to be stored in persistent storages and replicated through the whole VIM, like information about the tenants, users and configurations. The other two tree structures can be considered as portions of

[4] For sake of space and readability it is not possible to report the full tree structure in this document. The full structure is available on GitHub: https://github.com/eclipse/fog05

the *global* tree. In particular, they can be seen as sub-trees starting from the *node-id*, thus limiting the global view of the VIM and pointing to the information relevant to a specific node.

To better understand how to use the trees, let's consider the example of an EFS Entity already onboarded on a given node that needs to be executed. The information about the EFS Entity is stored in the *actual global* tree in the following key:

/agfos/id1/tenants/id2/nodes/id3/fdu/id4/instances/id5/info

The tenant owning the EFS Entity is identified by *id2*. The node is identified by *id3* and the EFS Entity is identified by *id4*. Finally, the specific EFS Entity instance[5] is identified by *id5*. Let's call this key as *key1*. In this example, the information contained in *key1* returns that EFS Entity is in the CONFIGURED state[6]. The goal is to execute the EFS Entity that implies changing its state to RUN. In order to achieve the state transition, it is necessary to write the target state in the *desired global* store at the following URI (let's call it *key2*):

/dgfos/id1/tenants/id2/nodes/id3/fdu/id4/instances/id5/info

The value written in *key2* is the same value that is contained in *key1* with status field updated to RUN. The write in the *desired global* storage causes the triggering of the agent in the node *id3*, that (*i*) verifies if the instance is actually in the node, (*ii*) finds the plugin in charge of the instance, and (*iii*) writes the target state required in the *desired local* store using a different key (let's call it *key3*):

/dgfos/id3/runtimes/id6/fdu/id4/instances/id5/info

This write triggers the plugin *id6* (e.g., LXD runtime) to start the EFS Entity. Upon successful operation, the agent updates the instance state in the *actual local* store. This update results in the agent updating the new instance state on *key1* on the *actual global* store. At this point, the execution request is considered finalized. The information in the *desired global* store is finally removed and the transaction is complete.

2.3.3 Distributed EFS Stack and Resource Orchestrator

Like the distributed VIM presented in Section 2.3.2, we leverage a distributed key-value store also for the design of the 5G-CORAL distributed EFS Stack and Resource Orchestrator. Figure 2-4 shows the components and the internal architecture of the EFS Stack and Resource Orchestrator. For what concerns the EFS Stack Orchestrator (EFS-SO) (shown in orange in Figure 2-4), multiple instances can be available, where each instance may be responsible of a subset of the overall EFS Stacks managed by the OCS. The information model of the EFS Stack as treated by the EFS-SO can be found in Appendix 10. The main features of the EFS-SO are, therefore, the following:

- Expose a REST API to the users/OSS/BSS to manage the lifecycle of the EFS Stacks, including onboarding, instantiation, and termination. EFS-SO REST API could be presented as a Graphical User Interface (GUI) to ease the interaction with the human user. An example of such GUI is presented later in Section 5.1;
- Validate the EFS Stack according to the information model of Appendix 10;
- Contact the EFS-RO to enforce lifecycle management decisions;
- Keep a catalogue of existing EFS Stacks in the system.

For what concerns the EFS-RO (shown in green in Figure 2-4), three main components can be identified:

[5] An EFS Entity may have multiple instances, e.g., for load balancing and/or high availability.
[6] Additional information on the Finite State Machine (FSM) abstraction can be found in D3.1 [6].

- **EFS Resource Orchestrator Engine:** it oversees keeping track of the resources under its control and allocating/arbitrating the usage of those resources. It works on an abstract network graph representation of the underlying infrastructure. It is also in charge of mapping the EFS Stack (which is represented in form of a graph) received by the EFS-SO onto the underlying infrastructure. Section 2.4 proposes a placement algorithm for volatile environments;
- **VIM connector:** it oversees the connection to the VIMs and translates the abstractions used in the EFS-RO Engine to the different VIM implementations. This includes authentication with the VIMs, implementations of VIM-specific APIs and information models. For example, the information model reported in D3.1 [6] can be used with fog05 as a VIM. Moreover, the VIM connector retrieves the infrastructure graph (i.e., nodes and links) from the VIM and exposes it to the EFS-RO Engine;
- **Cloud connector:** it oversees the connection to various Clouds. It fulfils the same tasks as the VIM connector, without retrieving the infrastructure graph. By definition, public Clouds (e.g., Amazon Web Services, Microsoft Azure, etc.) do not expose their internal infrastructure topology.

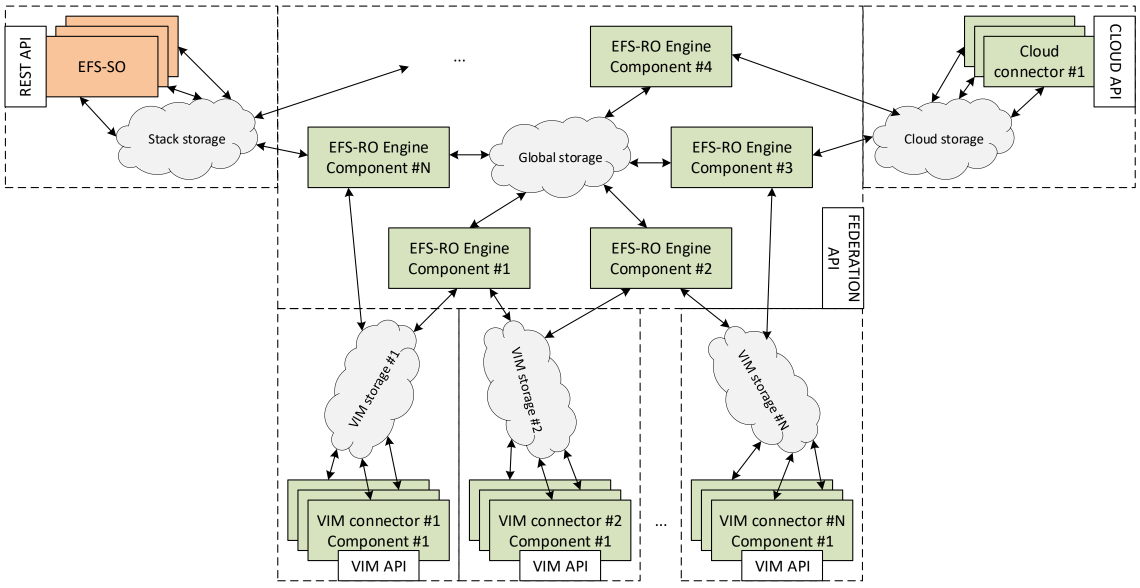
REST API
EFS-SO
Stack storage
...
EFS-RO Engine Component #4
EFS-RO Engine Component #N
Global storage
EFS-RO Engine Component #3
Cloud connector #1
Cloud storage
CLOUD API
EFS-RO Engine Component #1
EFS-RO Engine Component #2
FEDERATION API
VIM storage #1
VIM storage #2
VIM storage #N
VIM connector #1 Component #1
VIM connector #2 Component #1
VIM connector #N Component #1
VIM API
VIM API
VIM API

FIGURE 2-4: EFS-RO COMPONENTS AND INTERNAL ARCHITECTURE

Multiple instances of each EFS-RO component (i.e., EFS-RO Engine, VIM connector, and Cloud connector) can be available for high-availability and redundancy, by leveraging a distributed key-value store. In this way the internal state of the EFS-SO and EFS-RO is distributed across the network in different instances which can be retrieved at any time. Four storages are considered:

- **Stack storage:** it contains the information shared between the EFS-SO and the EFS-RO regarding the existing EFS Stacks. There is no information stored about the underlying infrastructure available in this storage;
- **Global storage:** it contains the information about the overall underlying infrastructure, including the status, and the mapping of the EFS Stack onto the underlying infrastructure;
- **VIM storages:** they contain the information about the infrastructure managed by the different VIMs and the status of the entities running on each VIM. Additionally, each VIM reads and writes information from a separate storage in order to avoid direct leaks between VIMs;

- **Cloud storages:** they contain the information about the status of the entities running on each Cloud. Additionally, each Cloud connector reads and writes information from a separate storage in order to avoid direct leaks between Cloud;

Similar to the distributed VIM proposed in Section 2.3.2, in order to tackle the volatility and the errors that could occur in a distributed environment, each of the four storage types is further decomposed in two additional sub-storages:

- **Actual storage:** it stores the actual stable state;
- **Desired storage:** it stores the next desired stable state.

For additional information about the actual and desired storages see Section 2.3.2. Figure 2-5 shows the URI tree for the EFS-RO global storage. In order to minimize the differences between the global stack (referenced as tenant), vim, and cloud storages, the global storage is structured in such a way the vim, stack, and cloud storages are sub-trees of the global storage. Specifically, the vim tree has the vim-id as a root while the stack storage has the tenant-id in as root.

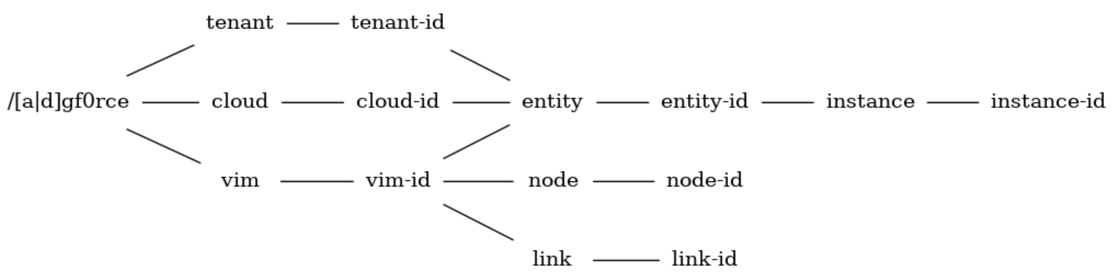

FIGURE 2-5: DISTRIBUTED EFS-RO GLOBAL STORAGE URI TREE

In the following we provide an example of how to use the proposed storages. Let's consider the case of instantiating an EFS Stack starting from the EFS-SO. A user (e.g., tenant id1) uploads the EFS Stack descriptor on the EFS-SO. After validating the descriptor, the EFS-SO writes on the desired stack storage (i.e., dsf0rce) the descriptor to notify the EFS-RO. The URI is the following:

/dsf0rce/tenant/id1/entity/id2

At this point the EFS-RO starts the onboarding of the entity (i.e., adding the entity to the catalogue). Once the onboarding is completed, the EFS-RO writes the descriptor from the desired to the actual storage. Next, the EFS-SO may request the instantiation of the entity by writing the desired state to:

/dsf0rce/tenant/id1/entity/id2/instance/id3

At this point, the EFS-RO execute the placement algorithm to identify the target VIM with the necessary resources to host the entity. Next, the EFS-RO writes the desired state to the VIM:

/dvf0rce/domain/id1/ entity/id2/instance/id3

As a next step, the VIM connector proceeds to instantiate the entity on the VIM. Upon successful instantiation, the VIM connector writes the state of the entity on the actual storage. In turn, the EFS-RO writes the actual state on the global storage and on the stack storage to finally notify the EFS-SO.

The resulting implementation of the EFS-SO and EFS-SO has been published as open source under the name of f0rce (i.e., fog orchestration engine) [5]. This implementation and exemplary procedure are then experimentally validated and evaluated in Section 2.3.3.

2.4 Placement algorithm for volatile environments

As introduced in D3.1 [6], the EFS Stack harmonizes and extends the ETSI MEC and ETSI NFV information model to encompass information that is relevant in the edge and fog environment, such as I/O devices, network interfaces, hardware accelerators, and location constraints. In D3.1 [6] we reported the information model relevant at VIM level. In this deliverable, we extended that information model up to the orchestrator level. By doing so, there is no need for the developer to specify the target infrastructure for the deployment upon on-boarding[7]. Instead, the EFS Resource Orchestrator identifies the most suitable resource for running the EFS Stack based on the requirements. The full EFS Stack information model can be found in the Appendix 10. The process of mapping the EFS Stack onto the underlying infrastructure is called *placement*. In the following, we design a placement algorithm suitable for the edge and fog environment where the following constraints are considered:

- EFS Atomic Entities requirements (e.g., CPU, memory, disk);
- Virtual Links (VL) requirements (e.g., bandwidth, delay);
- EFS Stack location;
- Radio Access Technologies (RAT);
- Infrastructure volatility;
- Infrastructure devices' lifetime (e.g., remaining battery of a fog node).

2.4.1 EFS Stack analytical modelling

The EFS Stack is encoded as a **directed labelled graph** G_{EFS}, with its EFS Atomic Entities $v \in V(G_{EFS})$ and virtual links $E(G_{EFS})$. Every EFS Atomic Entity $v \in V(G_{EFS})$ imposes an amount of cpu $c(v)$, memory $m(v)$, and disk $k(v)$; and it needs to be deployed on hardware equipment capable of providing such resources. Similarly, an EFS Atomic Entity may require to be executed within a specific geographical region (e.g., area, location), which we describe as a circle $B\big(p(v), s(v)\big)$ of center *p*(*v*) and radius *s*(*v*). Additionally, it may require a set of Radio Access Technologies (RATs) that must be available on the edge/fog node in order to be executed.

In the EFS Stack, the VLs are directed edges $(v_1, v_2) \in E(G_{EFS})$ interconnecting two EFS Atomic Entities. They represent the traffic flowing in a specific direction (from v_1 to v_2) with a specific bandwidth requirement $b(v_1, v_2)$ in Mbps. In the EFS Stack, the end-to-end propagation delay is controlled throughout the delay constraints of the VLs, and the physical links used to transport their traffic have to satisfy the imposed delays $d(v_1, v_2)$, $\forall (v_1, v_2) \in E(G_{EFS})$.

2.4.2 EFS Virtualization Infrastructure analytical modelling

Similar to the EFS Stack, the compound of infrastructure resources is a **directed graph** G_{infra} that includes the edge and fog resources (e.g., nodes, servers, switches, antennas, etc.). Every EFS resource is a node $h \in V\big(G_{infra}\big)$ with a CPU $c(h)$, memory $m(h)$, disk $k(h)$, and a set of RAT features $r(h)$. The connection between the infrastructure nodes is done with directed edges (h_1, h_2) that belong to the edges of the infrastructure graph $E\big(G_{infra}\big)$, and each of them provides a traffic capacity $b(h_1, h_2)$ ensuring an end-to-end delay $d(h_1, h_2)$.

Given the volatility of the edge and fog environment, each EFS resource is characterized by a reliability parameter $\nu(h) \in [0,1]$ to rank their capability of providing an uninterrupted service. In our work, this parameter is multiplied by a time interval (t_0, t_1) to determine for how long the

[7] The EFS Resource Orchestrator takes the decision of where deploying each EFS Atomic Entity. Therefore, such information needs to be provided to the VIM and properly described in the descriptor at VIM level.

resource is continuously available. In our model the reliability of a link between two infrastructure nodes is given by the minimum of the two nodes that it connects (i.e., $\nu(h_1, h_2) = \min\{\nu(h_1), \nu(h_2)\}$). Regarding the cost of using the infrastructure (i.e., pricing), we associate the costs $\rho_c(h)$, $\rho_m(h)$, $\rho_k(h)$, for the usage of the CPU, memory and disk, respectively. Finally, $\rho_b(h_1 h_2)$ represents the cost of allocating a Mbps on a link $(h_1, h_2) \in E(G_{infra})$. Table 2-3 reports the notation used in the placement algorithm for a quick reference.

TABLE 2-3: PLACEMENT ALGORITHM NOTATION

Notation	Type	Description
$\boldsymbol{G_{EFS}}$	Graph	EFS Stack directed labelled graph
$\boldsymbol{V(G_{EFS})}$	Set	EFS Atomic Entities composing the EFS Stack
$\boldsymbol{E(G_{EFS})}$	Set	EFS Stack VLs
$\boldsymbol{G_{infra}}$	Graph	Infrastructure directed graph
$\boldsymbol{V(G_{infra})}$	Set	Infrastructure nodes (e.g., fog, edge, cloud, switches)
$\boldsymbol{E(G_{infra})}$	Set	Infrastructure links
$\boldsymbol{c(v)}$	Parameter	CPU required by EFS Atomic Entity v
$\boldsymbol{m(v)}$	Parameter	Memory required by EFS Atomic Entity v
$\boldsymbol{k(v)}$	Parameter	Disk required by EFS Atomic Entity v
$\boldsymbol{b(v_1, v_2)}$	Parameter	Bandwidth required by VL (v_1, v_2)
$\boldsymbol{d(v_1, v_2)}$	Parameter	Maximum delay required by VL (v_1, v_2)
$\boldsymbol{r(v)}$	Set	Radio technologies required by EFS Atomic Entity v
$\boldsymbol{p(v)}$	Parameter	Centre of region $B(p(v), s(v))$ where EFS Atomic Entity v must be deployed
$\boldsymbol{s(v)}$	Parameter	Radius of region $B(p(v), s(v))$ where EFS Atomic Entity v must be deployed
$\boldsymbol{r(h)}$	Set	Radio technologies offered by infrastructure node h
$\boldsymbol{p(h)}$	Parameter	Coordinates of infrastructure node h
$\boldsymbol{\rho_c(h)}$	Parameter	CPU unit cost at infrastructure node h
$\boldsymbol{\rho_m(h)}$	Parameter	Memory unit cost at infrastructure node h
$\boldsymbol{\rho_k(h)}$	Parameter	Disk unit cost at infrastructure node h
$\boldsymbol{\rho_b(h_1, h_2)}$	Parameter	Bandwidth unit cost at link (h_1, h_2)
$\boldsymbol{\nu(h)}$	Parameter	Reliability of infrastructure node h
$\boldsymbol{\nu(h_1, h_2)}$	Parameter	Reliability of link (h_1, h_2)
$\boldsymbol{\delta_h(v)}$	Variable	Binary variable to tell if EFS Atomic Entity v is deployed at infrastructure node h
$\boldsymbol{\delta_{h_1,h_2}(v_1, v_2)}$	Variable	Binary variable to tell if VL (v_1, v_2) is deployed at link (h_1, h_2)

2.4.3 Placement heuristics

Upon the arrival of an EFS Stack instantiation request, the placement algorithm needs to decide if an infrastructure node h is capable of hosting the EFS Atomic Entity v, i.e., $\delta_h(v) = 1$, and if the traffic between (v_1, v_2) can be steered over an infrastructure link (h_1, h_2), i.e., $\delta_{h_1,h_2}(v) = 1$. Such decision affects the consumption of resources across the infrastructure, and how much delay is induced by the propagation delay of the selected physical links.

We denote $\kappa_h(v)$ as the cost of deploying an EFS Atomic Entity $v \in V(G_{EFS})$, and $\kappa_{h_1,h_2}(v_1, v_2)$ as the cost of mapping VL $(v_1, v_2) \in E(G_{EFS})$ on top of link $(h_1, h_2) \in E(G_{infra})$. Formally they can be defined as:

$$\kappa_h(v) = \rho_c(h) \cdot c(v) + \rho_m(h) \cdot m(v) + \rho_k(h) \cdot k(v), \quad h \in V(G_{infra}),\ v \in V(G_{EFS}) \tag{1}$$

$$\kappa_{h_1,h_2}(v_1, v_2) = \rho_b(h_1, h_2) \cdot b(v_1, v_2), \quad (h_1, h_2) \in V(G_{infra}),\ (v_1, v_2) \in E(G_{EFS}) \tag{2}$$

The proposed placement algorithms solve the following optimization problem in a greedy fashion:

$$\begin{aligned} \min \sum_{(h_1,h_2)\in E(G_{infra})} \sum_{(v_1,v_2)\in E(G_{EFS})} & \delta_{h_1}(v_1)\kappa_{h_1}(v_1) + \delta_{h_2}(v_2)\kappa_{h_2}(v_2) \\ & + \delta_{h_1,h_2}(v_1, v_2)\kappa_{h_1,h_2}(v_1, v_2) \end{aligned} \tag{3}$$

$$\text{s.t.}: \sum_{v\in\backslash V(G_{EFS})} c(v)\delta_h(v) \le c(h), \quad \forall h \in V(G_{infra}) \tag{4}$$

$$\sum_{v\in\backslash V(G_{EFS})} m(v)\delta_h(v) \le m(h), \quad \forall h \in V(G_{infra}) \tag{5}$$

$$\sum_{v\in\backslash V(G_{EFS})} k(v)\delta_h(v) \le k(h), \quad \forall h \in V(G_{infra}) \tag{6}$$

$$\sum_{(v_1,v_2)\in E(G_{EFS})} b(v_1, v_2)\delta_{h_1,h_2}(v_1, v_2) \le b(h_1, h_2), \quad \forall (h_1, h_2) \in E(G_{infra}) \tag{7}$$

$$\sum_{(h_1,h_2)\in E(G_{infra})} d(h_1, h_2)\delta_{h_1,h_2}(v_1, v_2) \le d(v_1, v_2), \quad \forall (v_1, v_2) \in E(G_{EFS}) \tag{8}$$

$$\sum_{h\in V(G_{infra})} \sum_{\gamma\in r(v)} 1_{r(h)}(\gamma) \cdot \delta_h(v) > 0, \quad \forall v \in V(G_{EFS}): |r(v)| > 0 \tag{9}$$

$$\delta_h(v)D\big(p(v), p(h)\big) \le s(v), \quad \forall h \in V(G_{infra}),\ v \in V(G_{EFS}): s(v) \neq \infty \tag{10}$$

Where $D: \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ denotes the Haversine distance [12] between two coordinates. Equations (3)-(10) represent the optimization problem that minimizes the deployment cost of those solutions that keep below the available resources and meet location constraints and radio requirements of the EFS Atomic Entities. In the following sections, two heuristic algorithms are proposed: the first focuses on cost optimization and the second on lifetime maximization.

2.4.3.1 Cost greedy heuristic

Our first heuristic aims to minimize the deployment cost stated in (1) and (2), while meeting all the other constraints. It iterates through each VL present in the EFS Stack, and then finds the cheapest infrastructure nodes capable of hosting the EFS Atomic Entities and the VLs. Then it looks for the shortest path to steer the virtual link traffic, using as weight for the shortest path graph algorithm the bandwidth cost of each link. Algorithm 2-1 illustrates the pseudo-code of the algorithm.

ALGORITHM 2-1: GREEDY COST HEURISTIC

```
     Data: $G_{EFS}, G_{infra}$
     Result: $\{\delta_h(v)\}_{h\in G_{infra}, v\in G_{EFS}}, \{\delta_{h_1,h_2}(v_1,v_2)\}_{h_1,h_2\in G_{infra}, v_1,v_2\in G_{EFS}}$
1.   for $(v_1,v_2)\in E(G_{EFS})$:
2.       $h_1 \leftarrow cheapest\_host(v_1)$ if $\delta_h(v_1) < 1, \forall h\in G_{infra}$
3.       $h_2 \leftarrow cheapest\_host(v_2)$ if $\delta_h(v_2) < 1, \forall h\in G_{infra}$
4.       $G'\_\{infra\} \leftarrow \langle V(G\_\{infra\}), E(G\_\{infra\}) \setminus \{ (h\_1,h\_2): b(h\_1,h\_2) < b(v\_1,v\_2) \vee d(h\_1,h\_2) > d(v\_1,v\_2) \} \rangle$
5.       $path \leftarrow Dijkstra(G'_{infra}, weight = \rho_b)$
6.       $consume\_resources(h_1, v_1)$
7.       $consume\_resources(h_2, v_2)$
8.       $consume\_resources(path, (v_1, v_2))$
9.       $\delta_{h_1}(v_1) = 1$
10.      $\delta_{h_2}(v_2) = 1$
11.      $\delta_{h_1,h_2}(v_1,v_2) = 1, \quad \forall (h_1,h_2)\in path$
12.  end for
```

Where $cheapest\_cost(v_1)$ is a function that finds the minimum cost host to deploy EFS Atomic Entity v_1, and $consume\_resources(h_1, v_1)$ allocates the CPU, memory and disk for v_1 on host h_1, and $consume\_resources(path, (v_1, v_2))$ allocates bandwidth for VL (v_1, v_2) along a physical path.

2.4.3.2 Fog greedy heuristic

In this second heuristic, rather than minimizing the deployment cost, the algorithm tries to maximize the lifetime of the deployed EFS Stack. That is, objective function (3) becomes:

$$\max \sum_{h\in V(G_{infra})} \sum_{v\in V(G_{EFS})} \delta_h(v) l(h) \tag{11}$$

which implies that the selection of hosts for each EFS Atomic Entity v now depends on the reliability provided by the infrastructure node. For example, imagine that an EFS Atomic Entity v periodically sends sensor-related information. Such EFS Atomic Entity is expected to run in the time interval (t_0=12:00, t_1=15:00). Then, let's consider a fog compute node with reliability $\nu(h_1) = 0.8$ and a second one with reliability $\nu(h_2) = 0.5$. This means that h_1 guarantees v to be available for $0.8 \cdot 3$ hours, while it would only be available only for $0.5 \cdot 3$ hours at h_2. Then, no matter the deployment cost, the fog greedy heuristic will choose h_1. The same procedure is done when looking for the physical links that steer each VL traffic, so the weight in the Dijkstra heuristic will be the link reliability. Algorithm 2-2 illustrates the pseudo-code of the algorithm.

ALGORITHM 2-2: FOG GREEDY HEURISTIC

```
     Data: $G_{EFS}, G_{infra}$
     Result: $\{\delta_h(v)\}_{h\in G_{infra}, v\in G_{EFS}}, \{\delta_{h_1,h_2}(v_1,v_2)\}_{h_1,h_2\in G_{infra}, v_1,v_2\in G_{EFS}}$
1.   for $(v_1,v_2)\in E(G_{EFS})$:
2.       $h_1 \leftarrow reliable\_host(v_1)$ if $\delta_h(v_1) < 1, \forall h\in G_{infra}$
3.       $h_2 \leftarrow reliable\_host(v_2)$ if $\delta_h(v_2) < 1, \forall h\in G_{infra}$
4.       $G'\_\{infra\} \leftarrow \langle V(G\_\{infra\}), E(G\_\{infra\}) \setminus \{ (h\_1,h\_2): (h\_1,h\_2) < b(v\_1,v\_2) \vee d(h\_1,h\_2) > d(v\_1,v\_2) \} \rangle$
5.       $path \leftarrow Dijkstra(G'_{infra}, weight = \nu)$
6.       $consume\_resources(h_1, v_1)$
```

```
7.          $consume\_resources(h_2, v_2)$
8.          $consume\_resources(path, (v_1, v_2))$
9.          $\delta_{h_1}(v_1) = 1$
10.         $\delta_{h_2}(v_2) = 1$
11.         $\delta_{h_1,h_2}(v_1, v_2) = 1, \quad \forall (h_1, h_2) \in path$
12.   end for
```

Where $reliable\_host(v_1)$ finds the most reliable host to deploy v_1.

2.4.4 Performance evaluation

For the sake of testing the performance of the two heuristics, we consider the reference infrastructure architecture and EFS Stack composition described in Appendix 11. In the following we report the performance evaluation based on simulations.

Figure 2-6 shows the ratio of cost per hour of deploying the reference EFS Stack comprising 5 EFS Entities. Results show that although the cost greedy algorithm is supposed to minimize costs, it stays always above the fog greedy deployments in terms of cost/hour. Since the fog greedy algorithm looks for more reliable nodes to deploy the EFS Entities, this causes mappings with higher lifetime, i.e., the EFS stack runs for longer time before stopping due to errors (such as running out of battery). This leads to a larger denominator in the cost/hour resulting in a better lifetime cost as shown in Figure 2-6.

In this performance evaluation, we consider the deployment of the reference EFS Stack for $t_1 - t_0$=24 hours, and we increase the average volatility of fog and edge nodes from $\mu_f = 0.1$ to $\mu_f = 0.5$, and $\mu_e = 0.01$ to $\mu_e = 0.1$, respectively.

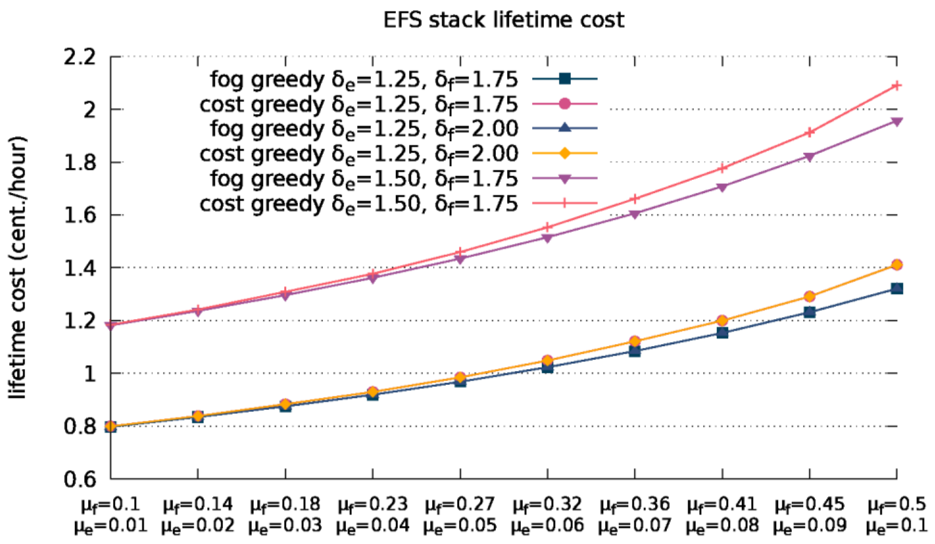

FIGURE 2-6: LIFETIME COST OF A REFERENCE EFS STACK AS VOLATILITY INCREASES

The experiment varies the values of fog and edge resources prices, δ_f and δ_e, respectively. Figure 2-6 shows that higher values of δ_f and δ_e lead to higher lifetime cost, as both fog and edge resources become more expensive. And among δ_f and δ_e, the most important parameter is δ_e, since those EFS Entities deployed at the edge are the ones contributing more for the deployment cost. In fact, the two scenarios $(\delta_e = 1.25, \delta_f = 1.75)$ and $(\delta_e = 1.25, \delta_f = 2)$

yield same results for both heuristics, since the increase of resource cost in the fog with respect to the edge is negligible.

As final remark, Figure 2-6 shows that as the infrastructure nodes increase their volatility (as μ_f, μ_e increase), the cost of mapping the EFS Stack increases as well because the lifetime of mapped EFS Stacks decreases.

2.4.5 Conclusions

After running the fog and cost greedy algorithms for the scenario described in Appendix 11, Section 2.4.4 showed that the fog greedy algorithm leads to better mappings than the cost greedy algorithm in terms of lifetime cost ratio.

Finally, results showed that volatile nodes within the infrastructure harm the lifetime cost of the EFS Stack. This means that the pricing of the edge resources has higher influence in the EFS Stack lifetime cost, than the pricing of fog resources. Therefore, ensuring a good level of connectivity and availability between the users and the edge resources may result in a lower overall cost since the edge resources appear *less volatile* from the user's perspective.

3 Live procedures and migration in the OCS

The edge and fog environments are highly dynamic due to the heterogeneity of the resources. Monitoring allows to collect metrics that capture such dynamicity and provides the OCS with inputs to make appropriate decisions. Hence, the OCS can perform and optimize the system lifecycle based on the data collected and provided by the EFS. Monitoring data can be either exposed by the resources or by the EFS Entities. Additional information on how to perform the monitoring in the EFS is available in D2.2 [8]. After having analysed each 5G-CORAL use case, a novel migration approach is proposed for EFS Entities targeted at downtime minimization.

3.1 OCS live procedures in 5G-CORAL use cases

This section first analyses how the OCS can leverage monitoring data in the context of each of the 5G-CORAL use cases, such as augmented reality navigation, virtual reality, fog-assisted robotics, high-speed train, and software defined wide area network. Next, it analyses the procedures required in each of those use cases.

3.1.1 Augmented reality navigation

The augmented reality (AR) navigation use case comprises one scenario envisioned in a shopping mall. The scenario envisions the end user being navigated in the shopping mall with help of AR navigation and map navigation. To that end, the end users require a stable navigation service access provided by AR navigation applications. A virtual AR navigation application is in the form of an EFS Application. This application allows visual indicators to show on screens to navigate end users. In addition, the application further connects to the Localization module in the form of an EFS Function to navigate the users with the current user location shown on the corner map in screen. These EFS Function and EFS Application are bundled together in a single EFS Stack for the complete deployment and lifecycle management of the AR Navigation services. Furthermore, OCS is aware of that an AR navigation application re-distributes navigation requests to other nearby applications if the AR navigation service load at the application is heavy. Therefore, the OCS, by taking such behaviour into consideration, is able to responsively instantiates a new application nearby once the burst situation happens, instead of scaling up the capability of the busy application.

3.1.1.1 Orchestrated Offload Mechanism of AR Navigation Service

In this OCS procedure, the EFS Service platform is capable of provisioning Wi-Fi Access and AR Navigation service in the Shopping Mall. Figure 3-1 shows the procedure: based on Resource Utilization information provided by the EFS Service platform, the instantiation of a new AR Navigation application is triggered. Such procedure is described as follows:

(A.0) An EFS App/Func Manager continuously performs the CPU utilization and network bandwidth utilization check to identify whether or not an AR Navigation Application is in a busy state (i.e., each of the utilizations is over a pre-defined threshold). In addition, the EFS App/Func Manager receives localization statistics from a Localization function in order to monitor user trajectory, which synthesizes both Bluetooth beacon-based Localization data service and image recognition Localization data service. The involved reference point is O5.

(A.1) Based on the information received from AR navigation applications and the Localization function, the EFS App. Manager decides whether a new instantiation of an AR Navigation application near the busy AR navigation application at the EFS Service Platform is required. If so, the EFS App. Manager requests the EFS Stack Orchestrator for a new AR navigation application deployment with the suggested deployment sites (Target EFS URI(s)). The involved reference point is O3.

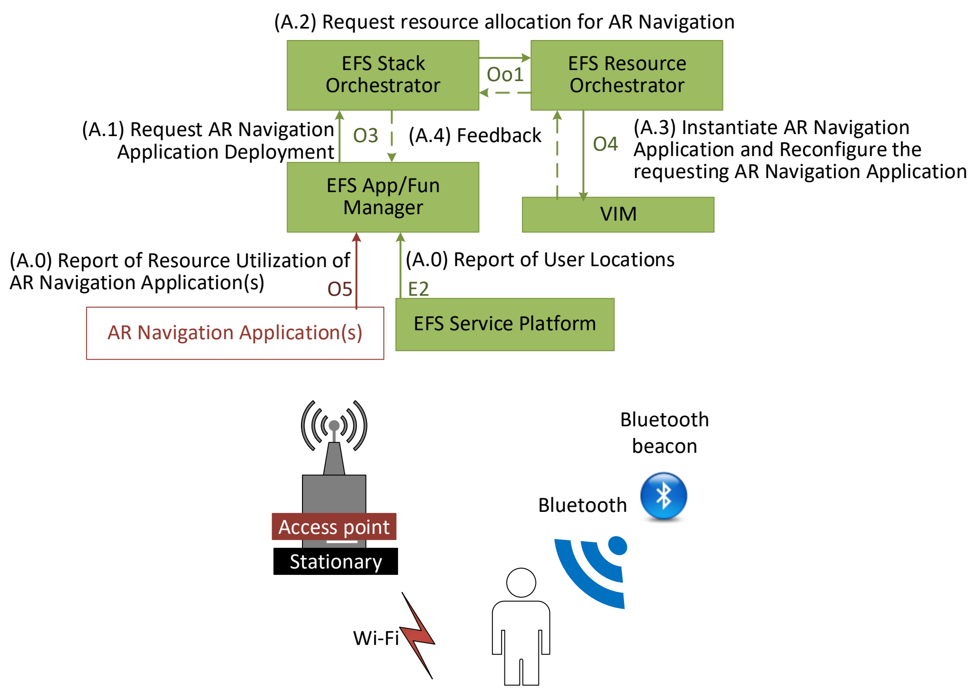
(A.2) Request resource allocation for AR Navigation
EFS Stack Orchestrator
Oo1
EFS Resource Orchestrator
(A.1) Request AR Navigation Application Deployment
O3
(A.4) Feedback
O4
(A.3) Instantiate AR Navigation Application and Reconfigure the requesting AR Navigation Application
EFS App/Fun Manager
VIM
(A.0) Report of Resource Utilization of AR Navigation Application(s)
O5
(A.0) Report of User Locations
E2
AR Navigation Application(s)
EFS Service Platform
Bluetooth beacon
Bluetooth
Access point
Stationary
Wi-Fi

FIGURE 3-1: OCS WORKFLOW FOR THE AR NAVIGATION APPLICATION DEPLOYMENT

(A.2) The EFS Stack Orchestrator then contacts the EFS Resource Orchestrator for allocating the required resources (e.g., CPU, RAM) on the EFS. Optionally, the EFS Stack Orchestrator may request a reconfiguration of the busy AR navigation applications and the other low-loaded AR navigation applications so as to enable offloading from busy applications to low-loaded applications. For example, once OCS found an over-loaded AR navigation application, the OCS associates the application with another application which is usually low-loaded by reconfiguring their behaviour with an offloading mechanism so that the over-loaded is able to re-distribute some AR navigation user requests to the low-loaded one. The involved reference point is Oo1.

(A.3) If the deployment request can be satisfied, the EFS Resource Orchestrator instructs the VIM to initiate a new AR navigation application at a nearby EFS Service platform and optionally to reconfigure the busy application. The involved reference point is O4.

(A.4) Feedback is provided to all the OCS components on the result of the procedure (e.g., successful or not). The involved reference points are O4, Oo1, and O3.

Table 3-1 reports the information exchanged during the EFS instantiation procedure.

TABLE 3-1: INFORMATION EXCHANGED IN THE EFS APPLICATION INSTANTIATION PROCEDURE

RP	Src	Dst	Information	Action	ID
O5	EFS Application	EFS App/Func Manager	Resource ID, Function Instance ID, Resource Utilization Status	Consume EFS Services related to the Resource Utilization information.	A.0
O3	EFS App/Func Manager	EFS Stack Orchestrator	Target Function Instance ID, Target Resource ID	Request the instantiation of the Function Instance ID to the target Resource ID	A.1
	EFS Stack Orchestrator	EFS App/Func Manager	Instantiation status	Feedback on the requested Instantiation	A.3
Oo1	EFS Stack Orchestrator	EFS Resource Orchestrator	EFS Stack Descriptor (Function Instance ID,	Request the instantiation of the EFS Application	A.2

			Links), Target Resource ID	described by EFS Stack Descriptor	
	EFS Resource Orchestrator	EFS Stack Orchestrator	Instantiation status	Feedback on the requested Instantiation	A.3
O4	EFS Resource Orchestrator	VIM	EFS Stack Descriptor (Function Instance ID, Links), Target Resource ID	Request the instantiation of the EFS Application described by EFS Stack Descriptor	A.2
	VIM	EFS Resource Orchestrator	Instantiation status	Feedback on the requested Instantiations	A.3

3.1.2 Virtual Reality

The VR use case consists of a 360-degree live video streaming delivered to multiple end users equipped with a mobile phone or VR goggles capable of processing such multimedia content. Video input is generated by multiple 360 cameras connected to a DASH server located in a remote server, while a local edge server and multiple fog nodes are deployed to reduce end-to-end latency and enhance system scalability.

Key component of this use case is the EFS orientation application. This application is provided by the EFS platform and physically runs inside the fog nodes. Its main goal is to forward information on the visual orientation of each end user to the local edge server that exploits these data in order to optimize the video streaming delivery. The lifecycle of the EFS orientation application is managed by the OCS, which can scale the service in and out depending on the number of end users requesting the video streaming. In the following, we show how the OCS can adopt an offloading mechanism to accommodate more users whenever the fog node resources, i.e., CPU processing power, are not enough.

3.1.2.1 Orchestrated Offload Mechanism of VR Navigation Service

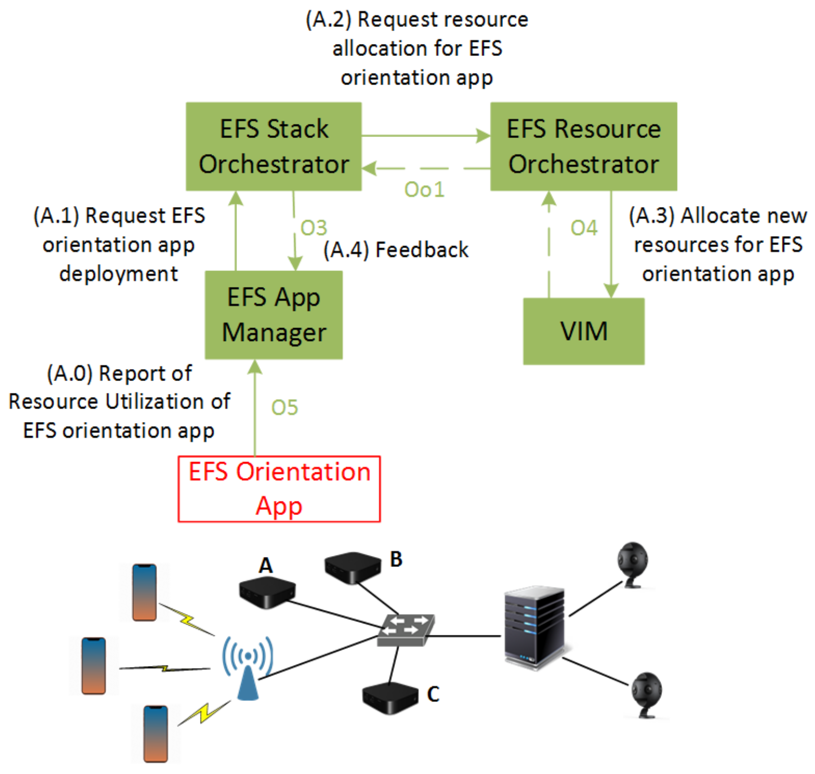
(A.2) Request resource allocation for EFS orientation app
EFS Stack Orchestrator
EFS Resource Orchestrator
Oo1
(A.1) Request EFS orientation app deployment
O3
(A.4) Feedback
O4
(A.3) Allocate new resources for EFS orientation app
EFS App Manager
VIM
(A.0) Report of Resource Utilization of EFS orientation app
O5
EFS Orientation App
A
B
C

Figure 3-2: OCS workflow for VR application deployment

In this OCS procedure, the EFS Service Platform is capable of providing the orientation application in the Shopping Mall by orchestrating resources running on the fog nodes A, B and C. Figure 3-2 shows the procedure: based on Resource Utilization information provided by the EFS orientation application, new resources for the orientation application are allocated based on the user demand. The detailed procedure is as follows:

(A.0) The EFS App manager continuously monitors the CPU utilization and network bandwidth utilization provided by the EFS orientation app to identify whether new service instances must be created. The involved reference point is O5.

(A.1) If a new instance is necessary, EFS App. Manager requests the EFS Stack Orchestrator for a new orientation app deployment. The involved reference point is O3.

(A.2) Next, the EFS Stack Orchestrator contacts the EFS Resource Orchestrator for allocating the required resources (e.g., CPU, RAM, storage) on the EFS Service platform nearby. The involved reference point is Oo1.

(A.3) If the deployment request can be accommodated, the EFS Resource Orchestrator instructs the VIM to allocate new resources. The involved reference point is O4.

(A.4) Feedback is provided to all the OCS components on the result of the procedure (e.g., successful or not). The involved reference points are O4, Oo1, and O3.

Table 3-2 reports the information exchanged in the EFS application instantiation procedure.

TABLE 3-2: INFORMATION EXCHANGED IN THE EFS APPLICATION INSTANTIATION PROCEDURE

RP	Src	Dst	Information	Action	ID
O5	EFS Orientation App	EFS App. Manager	Resource ID, Function Instance ID, Resource Utilization Status	Consume EFS app info related to the Resource Utilization information.	A.0
O3	EFS Func Manager	EFS Stack Orchestrator	Target Function Instance ID, Target Resource ID	Request the instantiation of the Function Instance ID to the target Resource ID	A.1
	EFS Stack Orchestrator	EFS Func Manager	Instantiation status	Feedback on the requested Instantiation	A.4
Oo1	EFS Stack Orchestrator	EFS Resource Orchestrator	EFS Stack Descriptor (Function Instance ID, Links), Target Resource ID	Request the instantiation of the EFS Application described by EFS Stack Descriptor	A.2
	EFS Resource Orchestrator	EFS Stack Orchestrator	Instantiation status	Feedback on the requested Instantiation	A.4
O4	EFS Resource Orchestrator	VIM	EFS Stack Descriptor (Function Instance ID, Links), Target Resource ID	Request allocation of new resources for EFS orientation app	A.3
	VIM	EFS Resource Orchestrator	Instantiation status	Feedback on the requested Instantiations	A.4

3.1.3 Fog-assisted robotics

The Fog-assisted Robotics use case comprises two different scenarios, both envisioned in a Shopping Mall scenario. The first scenario envisions the robots cleaning the common areas of the shopping mall. The second scenario, instead, envisions the delivery of goods by a group of robots working synchronously. In both scenarios, robots are connected via Wi-Fi and move in the Shopping Mall to accomplish the different tasks. To that end, the robots require constant Wi-Fi coverage wherever they go. The Wi-Fi connectivity is provided by a virtual Access Point in the form of an EFS Function. This function allows the robots to communicate with their control engine, which is deployed in the form of EFS Application. In the second scenario (delivery of goods) we also establish a low-latency Device-to-Device communication in order to maintain better coordination between the robots (e.g., moving in formation). The D2D connectivity is delivered as Wi-Fi P2P in the form of an EFS Function. These EFS Functions and EFS Application are bundled together in a single EFS Stack for the complete deployment and lifecycle management of the Fog-assisted Robotics services.

3.1.3.1 Migration of virtual AP based on Wi-Fi signal level

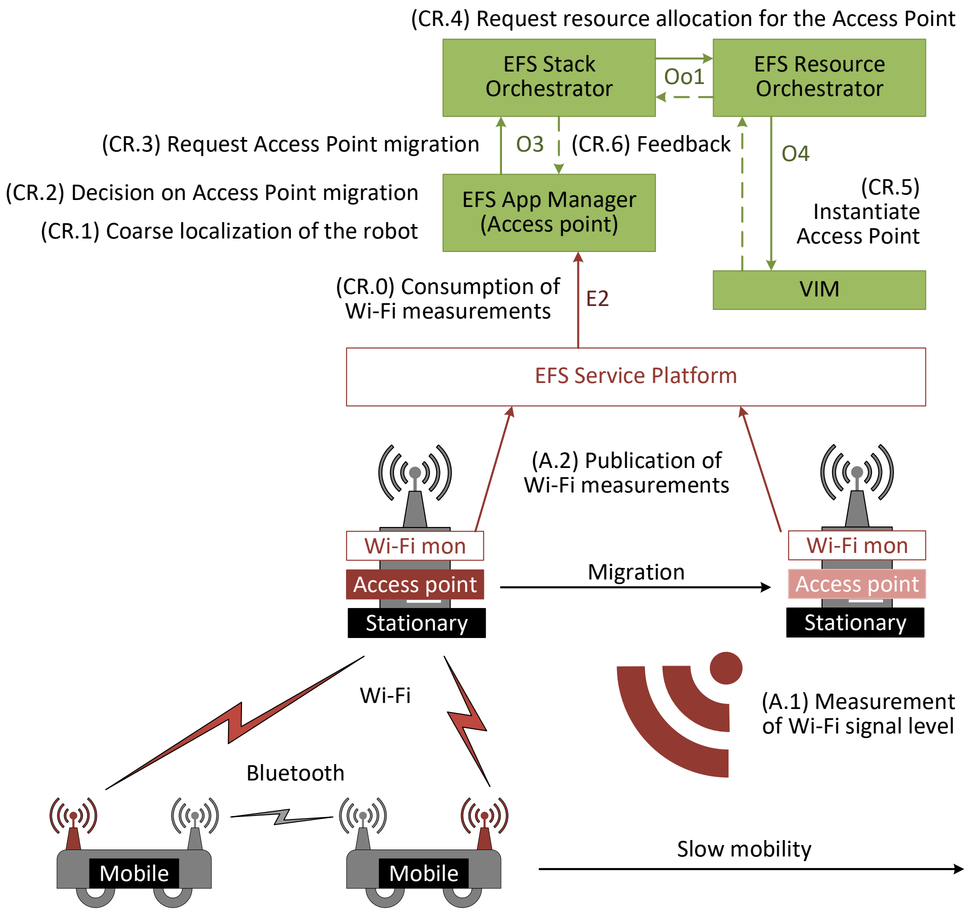
(CR.4) Request resource allocation for the Access Point
EFS Stack Orchestrator
Oo1
EFS Resource Orchestrator
(CR.3) Request Access Point migration
O3
(CR.6) Feedback
O4
(CR.2) Decision on Access Point migration
(CR.1) Coarse localization of the robot
EFS App Manager (Access point)
(CR.5) Instantiate Access Point
(CR.0) Consumption of Wi-Fi measurements
E2
VIM
EFS Service Platform
(A.2) Publication of Wi-Fi measurements
Wi-Fi mon
Access point
Stationary
Migration
Wi-Fi mon
Access point
Stationary
Wi-Fi
(A.1) Measurement of Wi-Fi signal level
Bluetooth
Mobile
Mobile
Slow mobility

FIGURE 3-3: OCS WORKFLOW FOR THE VIRTUAL AP MIGRATION BASED ON WI-FI SIGNAL LEVEL

In this OCS procedure, an EFS Function Manager is deployed and dedicated to the virtual Access Point in order to detect the movement of the robots and trigger the migration of the EFS Function so as to provide full connectivity coverage in the Shopping Mall. Figure 3-3 shows the procedure which relies on an EFS Service providing measurements and information regarding the signal level as seen by all the Wi-Fi-capable EFS resources. Such EFS service can provide the signal level of individual Wi-Fi stations as received at the virtual Access Point. The procedure of the measurement is the following:

(A.1) A dedicated EFS Application (i.e., Wi-Fi mon in Figure 3-3) runs on every Wi-Fi-capable EFS Resource and performs the corresponding measurements on the signal level.

(A.2) The Wi-Fi mon application publishes the signal level measurements via an EFS Service through the EFS Service platform. The involved reference point is E2.

The OCS procedure for the migration of the virtual AP based on Wi-Fi signal level is the following:

(CR.0) The EFS Func Manager associated to the virtual Access Point periodically consumes the EFS Service providing the Wi-Fi signal level as seen from the EFS Resources. The involved reference point is E2.

(CR.1) Based on this information, the EFS Func Manager monitors the coarse location of the robots. This is a step internal to the EFS Func Manager.

(CR.2) Based on the coarse localization of the robot, the EFS Func Manager decides when a migration of the virtual Access Point is needed (e.g., the robots are closer to a given EFS Resource than the one they are currently connected to). This is a step internal to the EFS Func Manager.

(CR.3) The EFS Func Manager contacts the EFS Stack Orchestrator to request the migration of the EFS function. The involved reference point is O3.

(CR.4) The EFS Stack Orchestrator then it contacts the EFS Resource Orchestrator for allocating the required resources (e.g., CPU, RAM, storage) on the target EFS Resource. The involved reference point is Oo1.

(CR.5) If the migration request can be satisfied, the EFS Resource Orchestrator instructs the VIM to migrate the virtual Access Point to the target EFS Resource. The involved reference point is O4.

(CR.6) Feedback is provided to all the OCS components on the result of the procedure (e.g., successful or not). The involved reference points are O4, Oo1, and O3.

TABLE 3-3: INFORMATION EXCHANGED IN THE EFS FUNCTION MIGRATION PROCEDURE

RP	Src	Dst	Information	Action	ID
E2	EFS Service Platform	EFS Func Manager	Resource ID, Wi-Fi station IDs, Wi-Fi signal level	Consume EFS Services related to the Wi-Fi information of surrounding Wi-Fi stations.	CR.0
O3	EFS Func Manager	EFS Stack Orchestrator	Function Instance ID, Dst Resource ID	Request the migration of the Function ID to the target Resource ID	CR.3
	EFS Stack Orchestrator	EFS Func Manager	Migration status	Feedback on the requested migration	CR.6
Oo1	EFS Stack Orchestrator	EFS Resource Orchestrator	Function Instance ID, Src Resource ID, Dst Resource ID	Request the migration of the Function ID from Src Resource ID to Dst Resource ID	CR.4
	EFS Resource Orchestrator	EFS Stack Orchestrator	Migration status	Feedback on the requested migration	CR.6
O4	EFS Resource Orchestrator	VIM	Function Instance ID, Src Resource ID, Dst Resource ID	Request the migration of the Function ID from Src Resource ID to Dst Resource ID	CR.5
	VIM	EFS Resource Orchestrator	Function Instance ID, Src Resource ID, Dst Resource ID	Feedback on the requested migration	CR.6

3.1.3.2 Low-latency D2D communication based on Localization

In this OCS procedure, an EFS Function Manager is deployed and dedicated to the D2D communication in order to monitor the location of the robots and establish or terminate the Wi-Fi P2P channel. Figure 3-4 shows the procedure which relies on an EFS Service providing localization information regarding the current coordinates of the robots. The procedure of the measurement is the following:

(A.1) A dedicated EFS Application (i.e., Localization mon in Figure 3-4) runs in the Robot intelligence and performs probabilistic localization of the robots. The probabilistic localization is based on adaptive (or KDL-sampling) Monte Carlo localization approach. By employing the data from the LiDAR, the robot pose is traced on a known map.

(A.2) The Localization mon application publishes the coordinates of the robots via an EFS Service through the EFS Service platform. The involved reference point is E2.

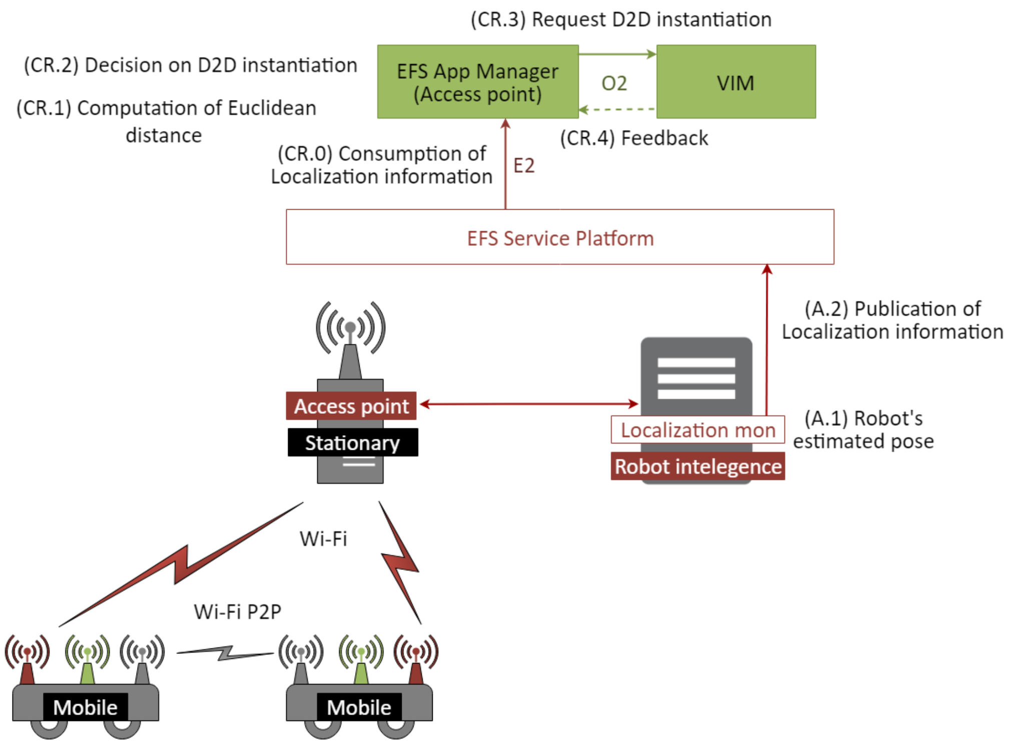
(CR.3) Request D2D instantiation
(CR.2) Decision on D2D instantiation
EFS App Manager (Access point)
O2
VIM
(CR.1) Computation of Euclidean distance
(CR.4) Feedback
(CR.0) Consumption of Localization information
E2
EFS Service Platform
(A.2) Publication of Localization information
Access point
Stationary
Localization mon
Robot intelegence
(A.1) Robot's estimated pose
Wi-Fi
Wi-Fi P2P
Mobile
Mobile

FIGURE 3-4: OCS WORKFLOW FOR THE D2D COMMUNICATION BASED ON LOCALIZATION

The OCS procedure for the lifecycle management of Low-latency D2D communication based on localization is the following:

(CR.0) The EFS Func Manager associated with the D2D connection periodically consumes the EFS Service providing the 2D localization coordinates on the map for the robots. The involved reference point is E2.

(CR.1) Based on this information, the EFS Func Manager computes the Euclidean distance. This is a step internal to the EFS Func Manager.

(CR.2) Based on the Euclidean distance between the robots, the EFS Func Manager decides when the D2D connection can be established (e.g., the robots are closer to a given EFS Resource than the one they are currently connected to). This is a step internal to the EFS Func Manager.

(CR.3) The EFS Func Manager contacts the VIM in order to instantiate the D2D connection according to the Wi-Fi Direct procedure [13]. The involved reference point is O2.

(CR.4) Feedback is provided by the VIM on the result of the instantiation procedure (e.g., successful or not). The involved reference point is O2.

Table 3-4 reports the information exchanged in the EFS function migration procedure.

TABLE 3-4: INFORMATION EXCHANGED IN THE EFS FUNCTION MIGRATION PROCEDURE

RP	Src	Dst	Information	Action	ID
E2	EFS Service Platform	EFS Func Manager	Robot ID, Robot x point on the map, Robot y point on the map	Consume EFS Services related to the localization information of the robots	CR.0
O2	EFS Func Manager	VIM	Function Instance ID, Dst Resource ID	Request the migration of the Function ID to the target Resource ID	CR.3
	VIM	EFS Func Manager	D2D status	Feedback on the requested instantiation	CR.4

3.1.4 High-Speed Train

In the high-speed train use case, the EFS platform plays the key role of monitoring and managing the EFS applications on-board. In the example below, we describe a potential benefit of employing the monitoring feature, consisting of a procedure to allow the EFS application on-board to migrate to another EFS node residing in shopping mall. The ability of monitoring and migrating EFS application at run-time is mission-critical in order to retain the edge service availability for the large number of users.

3.1.4.1 EFS application migration from on-board to on-land based on mobile connection

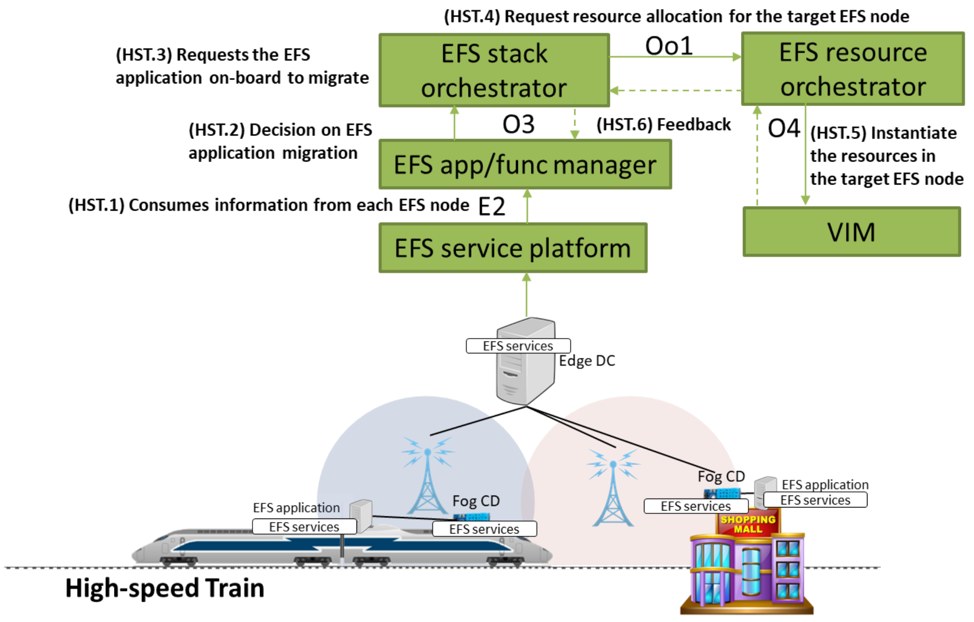
(HST.4) Request resource allocation for the target EFS node
(HST.3) Requests the EFS application on-board to migrate
EFS stack orchestrator
Oo1
EFS resource orchestrator
(HST.2) Decision on EFS application migration
O3
(HST.6) Feedback
O4
(HST.5) Instantiate the resources in the target EFS node
EFS app/func manager
(HST.1) Consumes information from each EFS node
E2
EFS service platform
VIM
EFS services
Edge DC
Fog CD
EFS application
EFS services
Fog CD
EFS services
EFS application
EFS services
EFS services
SHOPPING MALL
High-speed Train

FIGURE 3-5: OCS WORKFLOW FOR EFS APPLICATION MIGRATION FROM ON-BOARD TO ON-LAND BASED ON MOBILE NETWORK CONNECTION

In this OCS procedure, the OCS makes the migration decision and moves EFS application on-board to on-land based on the information from the EFS service platform, to provide the service continuity for each user using the edge service on-board. Figure 3-5 illustrates the OCS workflow for the EFS application migration. We assume that the EFS platform is capable of collecting migration-related information on each node in order to support the EFS application migration operation.

The OCS procedure for EFS application migration from on-board to on-land based on mobile network connection is as the following:

(HST.1) The EFS Func Manager associated with the edge DC periodically consumes the EFS services as mobility info, QoS, UE IP, EFS app ID and source EFS node ID. The involved reference point is E2.

(HST.2) Based on the mobility info, the EFS Func Manager decides when a migration of the EFS application is needed to retain the service continuity for each user using the edge service on-board. If it decides to do the migration, the EFS Func manager selects the EFS application to be migrated based on UE IP, EFS app ID and source EFS node ID and selects the destination EFS node based on mobility info. This output as source EFS node ID, destination EFS node ID, EFS app ID is then forwarded to the EFS stack orchestrator via the O3 interface.

(HST.3) The EFS stack orchestrator enforces the request received from the EFS Func manager.

(HST.4) The EFS stack orchestrator requests the EFS resource orchestrator for resource allocation for the destination EFS node by communicating over the Oo1 interface.

(HST.5) If the migration request can be satisfied, the EFS Resource Orchestrator instructs the VIM to instantiate the resources in the destination EFS node for edge service migration. The involved reference point is O4.

(HST.6) Feedback is provided to all the OCS components on the result of the procedure (e.g., successful or not). The involved reference points are O4, Oo1, and O3.

Table 3-5 reports the information exchanged in the EFS application migration procedure.

TABLE 3-5: INFORMATION EXCHANGED IN EFS APPLICATION MIGRATION PROCEDURE

RP	Src	Dst	Information	Action	ID
E2	EFS Service Platform	EFS Func Manager	Mobility info, QoS, UE IP, EFS app ID, src EFS node ID and other info	Consumes info from EFS service platform	HST.1
O3	EFS Func Manager	EFS Stack Orchestrator	Src EFS node ID, dst EFS node ID, EFS app ID and other info	Request the migration of the EFS application	HST.2
	EFS Stack Orchestrator	EFS Func Manager	Migration status	Feedback on the requested migration	HST.2
Oo1	EFS Stack Orchestrator	EFS Resource Orchestrator	Dst EFS node ID, EFS app ID and other info	Request resource allocation for the dst EFS node	HST.4
	EFS Resource Orchestrator	EFS Stack Orchestrator	Migration status	Feedback on the requested migration	HST.4
O4	EFS Resource Orchestrator	VIM	Dst EFS node ID, EFS app ID and other info	Instantiate the resources in the dst EFS node	HST.5
	VIM	EFS Resource Orchestrator	Migration status	Feedback on the requested migration	HST.5

3.1.5 Software Defined Wide Area Network (SD-WAN)

Software Defined Wide Area Network (SD-WAN) technology is the new generation of Wide Area Networks (WANs) which leverages Software Defined Network (SDN) in the scope of WANs. This use case integrates Edge and Fog infrastructure to virtualize network functions in order to provide a low latency and distributed network service that permits the deployment of an organization's WAN interconnecting the headquarters, branches and Cloud. The envisaged scenario is the shopping mall, where branch shops can use this service to establish a local network and to connect it to the company WAN. Also, a Point of Sale (PoS) application is defined, where banks can establish a secure connection with the shopping mall using SD-WAN. Those shops that use the payment service will connect to the WAN access point with their POSs and process payments. Isolation features make this scenario feasible to become a multi-tenancy environment, while resilience, fault-tolerance and flexibility are features that also enhance the use case.

3.1.5.1 Traffic balancing switching between LTE and broadband interface

This use case presents a Fog CD where a Kubernetes cluster has been installed. A couple of LXD containers are instantiated which execute different functions in order to get a reliable procedure. This procedure focuses on how to provide PoS terminals with a connection to the bank avoiding a direct VPN connection. For the case of PoS terminals, nowadays it can be expected that all the shops in the shopping mall accept credit card payments. Usually the procedure is to establish a secure connection between the terminal and the bank, which then processes the payment. Leveraging SD-WAN, many PoS can connect to the SD-WAN access point, which will establish a single secure connection with the bank, instead of multiple connections for each device. Also, the coverage can be enhanced inside the shopping mall leveraging the IEEE 802.11 access points.

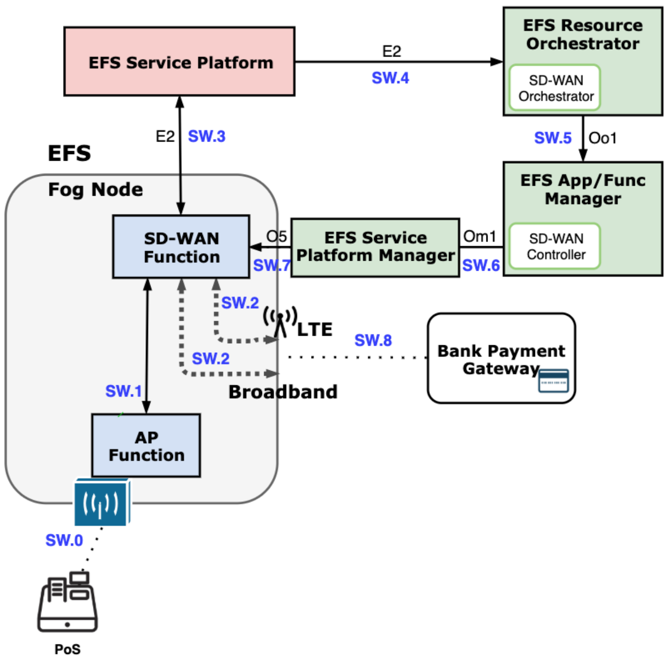

FIGURE 3-6: OCS WORKFLOW FOR TRAFFIC LOAD BALANCING BETWEEN LTE AND BROADBAND INTERFACES

The steps involved in the above procedure are described in the list below:

(SW.0) Point of Sale sends payment information to the AP function (e.g., Dockerized Access Point) by AP interface (i.e., dongle USB)

(SW.1) Information is forwarded to the SD-WAN function.

(SW.2) SD-WAN function collects statistic information about physical interfaces (i.e., LTE and Broadband), and current use of the network such as RTT latency or bandwidth currently being used.

(SW.3) The statistics measured at the physical interface are sent to the EFS Service Platform (Publisher/Subscriber node) by the SD-WAN function which publishes interfaces-related information to the service.

(SW.4) The EFS Resource orchestrator is subscribed to the EFS Service Platform where the SD-WAN monitoring data will be aggregated.

(SW.5) EFS Resource Orchestrator processes the aggregated data in the SD-WAN orchestrator and when necessary commands the EFS App/Func Manager to take action and rebalance the flows going through SW.3.

(SW.6) The EFS App/Func Manager processes the balancing actions sent by the resource orchestrator and using the SD-WAN controller forwards them to the EFS Service Platform Manager in order to activate the most efficient interface of both.

(SW.7) EFS Service Platform Manager acts as a proxy, forwarding the commands from the SD-WAN Controller to the SD-WAN function in order to trigger a change in the flows/paths installed.

(SW.8) SD-WAN sends payment information (received in second step) by the interface chosen by SD-WAN Orchestrator. Payment info arrives to its destination (e.g., Bank Payment Gateway).

Table 3-6 reports information exchanged in the EFS function procedure.

TABLE 3-6: INFORMATION EXCHANGED IN THE EFS FUNCTION PROCEDURE

RP	Src	Dst	Information	Action	ID
E2~=Mp1	EFS Service Platform	EFS Resource Orchestrator	Metrics/Statistics from physical network interfaces	Consume information from EFS Service Platform	SW.4
E2~=Mp1	SD-WAN Function	EFS Service Platform	Metrics/Statistics from physical network interfaces	Publish information to the EFS Service Platform	SW.3
Oo1	EFS Resource Orchestrator	EFS App/Func Manager	Commands to rebalance flows/paths	Requests the SD-WAN EFS Manger to rebalance flows/paths	SW.5
Om1	EFS App/Func Manager	EFS Service Platform Manager	Instructions to activate the optimum interface	Request an interface rebalancing	SW.6
O5	EFS App/Func Manager	SD-WAN function	Commands to activate the optimum interface	Proxies requests from the EFS App/Func Manager to SD-WAN function	SW.7

3.2 Common OCS features overview and container-based migration

To enable provisioning of EFS functions and applications on top of low-power edge devices, OCS provides lifecycle management support for lightweight virtualization technologies including system-based and application-based containerization. On the one hand, the system container behaves like a standalone Linux system. That is, a system container such as Linux Container (LXC/LXD) has its own root access, file system, memory, processes, networking and can be rebooted independently from the host system. On the other hand, the application container isolates an application from other applications running on top of the same host kernel and operating system. An application container such as Docker encapsulates its necessary libraries, configurations and dependencies without affecting the host system and other applications.

TABLE 3-7: COMMON OCS FEATURES ACROSS 5G-CORAL USE CASES

Title	Use Case	Description
Scale out, native app	**VR**	EFS Service platform is capable of providing the orientation application in the Shopping Mall by orchestrating resources running on the fog nodes. Based on Resource Utilization information provided by the EFS orientation application, new resource for the orientation application are allocated based on the user demand.
Scale out, LXC and docker	**AR navigation**	EFS Service platform is capable of provisioning Wi-Fi Access and AR Navigation service in the Shopping Mall. An instantiation of a new AR Navigation application is based on Resource Utilization information provided by the EFS Service platform.
Migration, docker	**High-Speed Train**	OCS make the migration decision and migrate EFS application on-board to on-land based on the information from the EFS service platform to provide the service continuity for each user using the edge service on-board. We assume that the EFS platform is capable of collecting migration-related information on each node in order to support the EFS application migration operation.
Migration, LXD	**Fog-assisted robotics**	EFS Function Manager is deployed and dedicated to the virtual Access Point in order to detect the movement of the robots and trigger the migration of the EFS Function so as to provide full connectivity coverage in the Shopping Mall.

TABLE 3-8: SPECIFIC OCS FEATURES OF SOME 5G-CORAL USE CASE

Title	Use Case	Description
Scale up, docker pods	**SD-WAN**	It focuses on container provisioning in order to resize them to guarantee the best performance and 5G-CORAL KPIs fulfilment. Monitoring framework measures PODs/Dockers/VMs parameters such as computing (CPUs, RAMs), storage (HDD/SSD), and networking (virtual interfaces).

After the detailed description of the OCS monitoring procedures involved in each use case present in 5G-CORAL project, one can distinguish some common features as reported in Table 3-7 and Table 3-8. This is the case for migration and scale up.

Container migration can be classified into stateful and stateless. In stateless migration (aka *cold* or *offline* migration), the state of the container is not preserved when the container is relocated to the destination node. In the case of stateful migration (aka *live* migration), the state of the container is retained when the container is restored at the destination node. There are three schemes of stateful migration as follows:

- ***stop-and-copy*** - freezes the container, checkpoints its state, copies the container image and its state to the destination then restores the state from the checkpoint [14].
- ***pre-copy*** - performs iterative state checkpointing while the container is running till the amount of in-memory change is at minimum, then concludes with a shorter stop-and-copy [15]. Iterative checkpointing reduces the size of the final checkpoint which is performed while the container is frozen. This minimizes the time required for the final checkpoint and the time required to copy the checkpoint to destination.
- ***post-copy*** - performs a short stop-and-copy to move essential state data, then starts the container at the destination and retrieves the rest of the data when required [16]. This type of migration has a very small downtime, but containers may suffer from performance degradation due to the time needed to wait for the requested memory pages.

Table 3-9 shows a summary of the pros and cons of these migration schemes.

TABLE 3-9: PROS AND CONS OF STOP-AND-COPY, PRE-COPY AND POST COPY MIGRATION SCHEMES

Feature/Scheme	stop-and-copy	pre-copy	post-copy
Downtime	Longest – includes the time required to checkpoint and copy the entire state.	Short – only includes the time required for the last iteration of checkpoint and copy.	Shortest – only includes the time required to checkpoint and copy the essential state.
Migration time	Short – the total migration time is short because it is done in one iteration.	Long – depends on the number of iterations. The more iterations, the longer the total migration time.	Long – depends on the running application and the amount of time it requires to retrieve the entire state from the source.
Application performance	Affected only during downtime.	Affected only during downtime.	Affected during downtime and also due to latency during the retrieval of state from the source while application is running.
Network utilization	Low – only one copy of the state is transferred.	High – the total state size accumulatively grows with the number of iterations.	Low – only one copy of the state is transferred.

In the case of traditional hypervisor-based virtualization, virtual machine (VM) migration is well investigated [17] and many successful solutions are commercially available. For instance, a pre-copy based VM migration scheme is presented in [15]. An active VM continues to run in the course of in-memory data iterative pre-copying. During a consecutive iteration, only dirty pages are transferred. At last, a final state copy is performed while the VM instance is frozen and then transferred to the destination host. This way, the amount of downtime is greatly reduced when compared to a pure stop-and-copy scheme. Although VM migration is a mature technology, it relies on hypervisors and most of the existing solutions are tailored for data centre environment where network-attached storage (NAS) and specific virtualization technology are utilized. NAS enables all the host machines in a data centre to access a network-shared storage which reduces the time spent during the copying stage. However, in a scenario where migration takes place between edge nodes, the state and local-disk storage have to be copied over wide area network (WAN).

Recently, container migration has caught more attention from the research community [18] [20]. Especially, since containerization offers many advantages over traditional hypervisor-based virtualization such as resource efficiency and performance. This fact enables the instantiation of lightweight containerized applications suitable for IoT services [20]. In [18], container migration mechanism is developed for power efficiency optimization in heterogeneous data centre. This work assumes that the source and destination hosts have access to a NAS and thus container data is not copied over WAN. Furthermore, a framework for migrating edge containerized applications is presented in [19]. The proposed framework is the first to consider MEC environment for system container migration. Fundamentally, the framework is a layered model which aims to reduce the downtime incurred by the migration process. While the presented results show reduction in downtime as a result of layering, the framework relies on stop-and-copy migration which is not an efficient method for containers with large in-memory state.

Migration is introduced in the High-Speed Train and Fog-assisted Robotics use cases. From implementation point of view, in High-Speed train the migration refers to docker containers while in Fog-assisted Robotics it points to LXD technology. As a brief summary, in High-Speed the EFS service platform provides information to the OCS, which migrates on-board EFS applications to on-land to provide service continuity. In Fog-assisted Robotics, the EFS Function Manager is deployed and dedicated to the virtual access point in order to detect the movement of the robots and trigger the migration of the EFS Function so as to provide full connectivity coverage in the Shopping Mall.

The migration feature demands the application of some OCS functional requirements, listed below:

- Support of harvesting computing capabilities from mobile resources. Migration and mobility are closed characteristics. Thus, in order to feed the OCS with relevant information, it must collect capabilities from mobile resources, e.g. train or robot.
- Support of discovery, monitoring, allocation, etc. of relevant hardware capabilities. With the objective of selecting the best target resource to migrate EFS applications and functions, OCS must gather information about possible destination EFS resources.
- Support of federation including at runtime OCS components. Migration could imply a migration out of the current domain. Hence, federation is needed to manage instantiation of EFS entities among different domains.

With refer to non-functional requirements, please find listed the ones tagged as important for migration.

- Availability and self-healing mechanisms in error-prone environments. The migration procedure should provide recovery mechanisms if errors are produced while migrating an EFS entity. Thus, this non-functional requirement gains importance.

Experimental validation and performance assessment of the migration feature can be found in Section 5.3.

The second common feature identified in three use cases is scale out. In VR, it is described scale out of native applications, in AR it refers to LXD and docker containers. Finally, IoT multi-RAT focus only on docker. This feature aims to create more instances of an EFS application or function when, for instance, resources reach a defined limit. As a brief overview, VR use case pretends to allocate resources based on user demands analysing resource utilization information provided by the EFS orientation application. The AR use case scales out the AR navigation application in the shopping mall based on resource utilization information provided by the EFS service platform. Finally, IoT multi-RAT intends to scale out the virtualize communication stack function in a new node when it is under heavy load.

Regarding what functional requirements apply to this feature, it is listed below the most relevant ones.

- Support of harvesting computing capabilities from low-end resources. It is a key characteristic which allow the OCS to know the state of the function or the application and trigger the scalability of it.
- Support of harvesting computing capabilities from mobile resources. Similar to the one before, OCS should collect computing data from mobile resources and get an overview of how the system is behaving and take proper actions.
- Support of discovery, configuration, monitoring, allocation, etc. of relevant hardware capabilities. To scale up the function, OCS needs to discover hardware capabilities where instantiate the new function, besides configuring it, set the monitoring to get the status, etc.
- Support of integration including at runtime of heterogeneous resources in terms of software and hardware capabilities. Different type of resources can be used to start a new function, so this is a desired requirement to be fulfilled.

- Support of federation including at runtime of OCS components. Federation can be leveraged to user another domain with more resources that the actual and increase the number of entities serving.
- Support of the interworking with resources external to the OCS. Functions could be scaled out in the cloud, if the scenario requires it, so this is another requirement.

In relation to the non-functional requirements for scale out, some of them are important to this feature.

- Availability and self-healing mechanisms in error-prone environments. The deployment of a new function is a process where error can be encountered, and therefore availability and self-healing have to be accomplished.
- Support of large deployments in terms of number of resources and geographic areas. Scaling out the system may imply a large number of functions deployed. Thus, depending on the purpose and design of it, the application may grow up being formed by a large number of resources extended in different locations.
- Capability to adapt to workload changes by provisioning and deprovisioning resources in an automated manner. This refers to scale up the underlying resources, which will be used to scale up functions and applications. Both can be linked, and if a system requests more capabilities, first is to load resources and after scale the entities.

Finally, one additional feature is described (see Table 3-8) but only used in one particular use case, that is the scale up in the SD-WAN use case. The scale up of containers is similar to scale out, commented before. The container itself is reconfigured on demand to increase or decrease the capabilities and the resources allocated. The scale up procedure does not create a new container.

4 Federation and resource provisioning

This section introduces the federation concept in 5G CORAL. Particular focus is given to the federation of resources between different administrative domains. Furthermore, a general system model is used in order to analyse and validate profit-maximized federations and advanced resource provisioning.

4.1 Federation of resources

Federation has been described in [6] and [21] as a mechanism for integrating multiple administrative domains at a different granularity into a unified open platform where the federated resources can trust each other at a certain degree.

Each administrative domain is composed of set of computing/storage/networking devices that shape the underlying infrastructure of a single administrative domain. As mentioned in [6], multiple administrative domains may exist in a same service area. Considering the 5G-CORAL environment, the underlying infrastructures of multiple administrative domains are in constant adjacency. The nearness of various technologies opens a spectrum of possibilities for deployment of different EFS services/applications that rely on multiple underlying infrastructures. By cooperation among administrative domains and losing the strict boundaries, the inclusion of external resources is feasible. The process of adopting external resources provided by another peering/provider domain for the goal of deploying an EFS service/application is called **federation of resources**.

How an administrative domain would benefit from a federation of resources? In 5G-CORAL environment, each administrative domain has its own underlying infrastructure as EFS resources. The quantity of the set of EFS resources varies from large to a set of few EFS resource per administrative domain. In both cases, large or few amount EFS resources, each underlying infrastructure is limited. The limitation can be in terms of capacity, lack of certain technology, user accessibility, etc. In order to expand the limitation without extending the CAPEX and/or OPEX, the administrative domains can use federation feature. The federation as concept allows the administrative domains to maintain the service level without service interruption and high expenses. Depending on the inter-domain interactions, the global welfare of the administrative domains may increase with adoption of federation feature. In environment close to the edge of the network where the infrastructure resources are volatile, through the use of resource federation, the stability can be increased.

In order to enable the federation of resources through 5G-CORAL platform, the whole process of federation goes through several steps. First, it is mandatory to identify all the stakeholders/actors that are part of a certain use case scenario (see Section 4.1.1). Next, a proper model of interaction between all the involved parties or stakeholders has to be established (see Section 4.1.2). Finally, the process of resource federation implemented by setting up how EFS resources interact and establish multi-domain connections between each other using the 5G-CORAL system (see Section 4.1.3, Section 4.1.3.1, Section 4.1.3.2, Section 4.1.3.3 and Section 4.1.4).

4.1.1 Federation roles

The federation procedure is dependent on the setup scenario or the circumstances that demand multiple administrative domains to enable federation among themselves. In the federation process a domain can play two roles: **consumer** and **provider**. Consumer role has the administrative domain that requests federation of resources or resources from external domain to be included as part of its domain/services. The provider role is when the administrative domain provides set of resources to an external (consumer) domain under certain conditions. In each federation scenario there are at least a single consumer domain and a single or multiple provider domains. Administrative domains that have the underlying infrastructure in a near proximity (e.g. same geo-location, co-exist in mutual coverage area, etc.,) are keener to employ federation than administrative domains that are distant (e.g., domain in separate countries).

Prior to any federation procedure, the administrative domains need to define the relationships among themselves in each case they interact as provider and/or consumer roles. The relationships are set on business level in terms of trust policies. These agreements can be statically set in advance (e.g., long time before any federation interaction) or they can be dynamically set, minutes range before any federation procedure. The static agreements or pre-established (Section 5.2 in [21]) are useful for administrative domains that would expect frequent interaction among themselves, usually neighbouring administrative domains. The agreements set up all the terms for both consumer and provider roles, the pricing models, the trust policies, the security level among the administrative domains. For instance, in a cooperative neighbouring interaction, the terms and policies for general resource federation can be set in manner that is better for the provider, while for particular use-cases a different set of terms and usage polices can be favourable for consumer. These agreements in pre-agreed federation are usually long-term agreements with fixed pricing (subscription based), but any length or pricing can be applied. More information regarding the agreements and the pricing can be found in D3.1 [6].

Dynamic or open federation (Section 5.2 in [21]) relationships are set on-line, minutes prior to establishing any federation of resources or services. These agreements usually define roles in a particular use-case. They contain similar terms and policies; however, they are mostly short-term with dynamic pricing policies. The open federation is usually competitive following an auction model of reserving resources (Section 3.5 in [6]). Moreover, as in an open federation, the administrative domain decides dynamically whether to join or leave an existing federation. The administrative domain does not need to make decisions at predetermined time, so the duration of its federation membership is not fixed. Federation in this case is dynamically formed in a distributed, bottom-up manner.

For particular use-cases, the static approach would have pre-determined roles and amount of resources that each provider domain provides to the consumer domain. The time to request, reserve and use federated resources is shorter than in the open-federation manner. Moreover, administrative domains form a federation based on a (long-term or short-term) agreement so that their membership remains unchanged for an extended period of time. Also, mutual agreements are required for any membership change to an existing federation. Federation in this case can be formed by a central entity in an offline, top-down manner. Table 4-1 compares dynamic federation with static federation.

TABLE 4-1: COMPARISONS BETWEEN STATIC AND DYNAMIC FEDERATION

Feature	Dynamic Federation	Static Federation
Membership Change Frequency	High	Low
Membership Change Approach	Autonomous, distributed, bottom-up	Central controlled, top-down
Stability	Potentially unstable	Stable

In 5G-CORAL we are focusing on the pre-determined federation model. The open-federation model is left for further study. The adjacent administrative domains settle general agreements and agreements that support their use cases. The agreements contain the interaction models and the way that the federation is going to be implemented. Next section dives into the details of the interaction model.

4.1.2 Federation interaction model

Once the federation between 5G-CORAL administrative domains is defined as static or pre-established (as explained in Section 5.2 in [21]). The next step is to define the interaction between the 5G-CORAL platform at each domain. The interaction between the administrative domains can be on hierarchical or peer-to-peer level. The approach of the 5G-CORAL is to apply peer-to-peer cooperative model of interaction. In D3.1, three cooperative models are introduced for EFS resource federation:

- Trust model
- Loan model
- Concession model

The loan model is preferable for the open federation, while the concession model for the non-volatile resources and the trust model is well suited for long-term inter-domain relationships. For these reasons and since the static method is adapted in 5G-CORAL, the trust cooperative peer-to-peer model is most suitable at this point. In this static model the pricing can be fixed or posted-scheme that goes through subscription-based charging scheme (monthly or yearly based) [6]. Additional to the defined federation model, each administrative domain may introduce sub-models for specific use-cases that needs to be translated to well-defined SLAs. Moreover, the specific use-case would be seen as a case where different SLA agreements providing better conditions is in place instead of the agreement for a general federation. For example, for a certain administrative domain that provides specific set of services over Wi-Fi access, it may set up specific SLA agreements with neighbouring domains over their Wi-Fi radio resources.

4.1.3 Inter-domain connection (F2 interface)

Next, an administrative domain establishes links to all federated domains on the OCS level via the F2 interface. For example, if administrative domain A has established federation agreements with administrative domain B and administrative domain C then there will be two links on the F2 interface, one from OCS A towards OCS B and another one from OCS A towards OCS C. The F2 interface is an interface for inter-connection of peer-to-peer OCS platforms residing in different administrative domain. The document focuses on the resource federation, hence the communication through the F2 interface would be mainly towards the federation of resources related operations. Having that in mind, the communication on F2 interface is between EFS Resource Orchestration modules.

The EFS Resource Orchestrator module supports accessing the edge and Fog resources in an abstracted manner independently of any VIMs, as well as governance of service platform/function/application instances sharing resources in the EFS [6]. In the federation (SLA) agreements the administrative domains share the endpoints (e.g., IP addresses, URL, etc.,) of their EFS Resource Orchestrators. The endpoints are used to enable communication through the F2 interface. The communication on the F2 interface is composed of three phases: advertisement phase, instantiation phase, and termination phase (shown on Figure 4-1).

To successfully perform the federation, EFS Resource Orchestrators belonging to different domains will communicate via interface F2 to execute a federation message exchange. Within the message exchange, the consumer domain EFS RO has to start the procedure, and the provider EFS RO will suggest a feasible node to be federated (advertisement/discovery phase). Then, the consumer EFS RO will accept or decline the offered resource (negotiation phase), answering to the provider EFS RO. The EFS RO should interact with the VIM and the EFS Application/Function Manager to complete the process (instantiation phase). Figure 4-4 describes further the interaction of the federation interface F2 with the rest of components in the OCS by using a sequence diagram. Figure 4-4 is further detailed in Section 4.1.4.

4.1.3.1 Advertisement/negotiation phase

The advertisement phase or negotiation phase is when all inter-connected administrative domains request/offer set of EFS resources. An administrative domain as a consumer role requests federating resources from other provider domains, whereas an administrative domain as a provider role offers available resources for federation and negotiate over their usage (e.g., duration, pricing, etc.). The providers of federated resources can periodically update their capabilities or reply the offered resources per request. The periodic update of currently available resources for federation would enable all peering administrative domains to have updated global view and rapidly decide for the optimal resources. However, the mobility and volatility of the 5G-CORAL resources demand frequent message exchange on the F2 interfaces, which due to delays or traffic congestions may produce inaccurate updates of the global view. To overcome this issue, the provider EFS Resource Orchestrator advertises the available resources for federation only upon received request from a (potential) consumer domain. The request/advertise approach would allow each administrative domain to apply policies and prioritize requests. For example, domain B may respond to a request arrived from a highly ranked domain A and not respond to a request from lower ranked domain C, in case that both requests arrived at the same time at domain B. In this way, by applying the policies, the signalling overhead is significantly reduced.

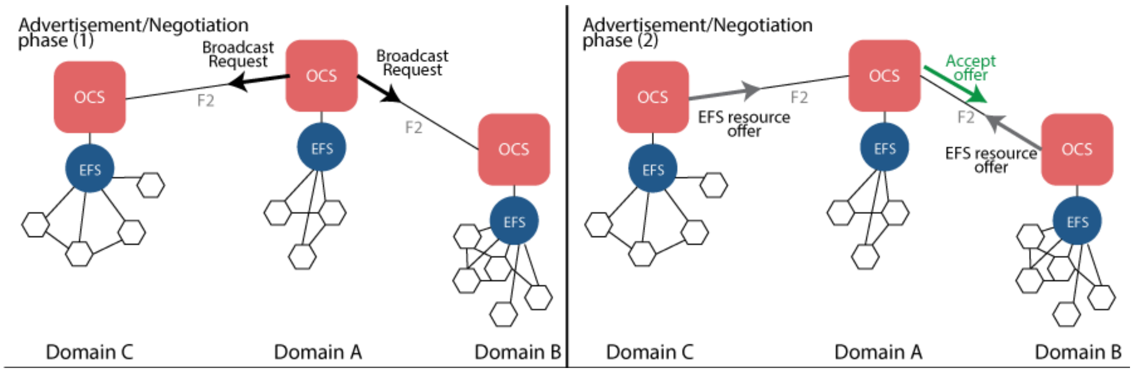

FIGURE 4-1: OCS FEDERATION INTERACTION – ADVERTISEMENT/NEGOTIATION PHASE

Once the consumer domain has the need of adapting federated resources, the constituent EFS Resource Orchestrator prepares a request for federation. The request is multi-casted towards the peering administrative domains according to the demands needed (e.g. geo-location of the resource). For example, as Figure 4-1 shows, domain A broadcasts requests to neighbouring domains (domain B and domain C). The potential provider domains (B and C) generate their offers/advertisements of available resources for federation and respond to the request. The consumer domain A accumulates the responses for a certain time (e.g., once a timeout for received offers expires) and then ranks the received advertisements. As shown on Figure 4-1, the consumer domain A chooses the optimal set of resources (from domain B) and the EFS Resource Orchestrator sends reservation requests (Accept offer) to the chosen provider domain B. The chosen providers confirm the reservation request and that is the last message exchange for the advertisement/negotiation phase.

During the negotiation phase, parties should take into account the federation stability, which could be affected by at least two factors. First, mobility and volatility of EFS resource may later invalidate the usability of federated resources that have been offered. Second, the provider domain may unilaterally retract federated resources that have been offered to some consumer domain and provide another consumer domain with the retract resources as a means to earn more profit. Generally speaking, if a participant can earn more profit by leaving a federation, the federation will fall apart; if a group of participants can all earn more profits by leaving a federation and forming another one, the federation will fall apart. This scenario may not be avoided if administrative domains earn their own profits individually, as in the case of peer-to-peer federation model. However, if all participants share the total profit in the federation (a group federation), instability of federation can be avoided by an appropriate allocation of federation profits to members.

4.1.3.2 Federation instantiation phase

The instantiation phase begins when the provider EFS RO confirms incoming request for reservation of available resources. Then the EFS Resource Orchestrator sends reservation request to the VIM on the O4 interface. From the three planes (management, control and data plane), only the management plane is not federated. The provider domain keeps the EFS resource attached to the local management plane. The VIM reconfigures the control and data plane of the resources that are being reserved. Once both planes are reset to idle, the operation is confirmed from the VIM to the EFS Resource Orchestrator. In order to connect the reserved resources with the consumer domain, the EFS Resource Orchestrator issues request to the EFS Application/Function Manager to instantiate tunnelling function (e.g., SDN-WAN function) on top of the reserved resources (see Figure 4-2). The tunnelling (SDN-WAN) function is instantiated in order to create secure tunnel and grant orchestration privilege to the consumer (external) domain over the control and data plane of the reserved resources. Note that the management plane of the reserved resources would remain orchestrated by the constituent EFS Resource Orchestrator and VIM for the whole duration of the federation process.

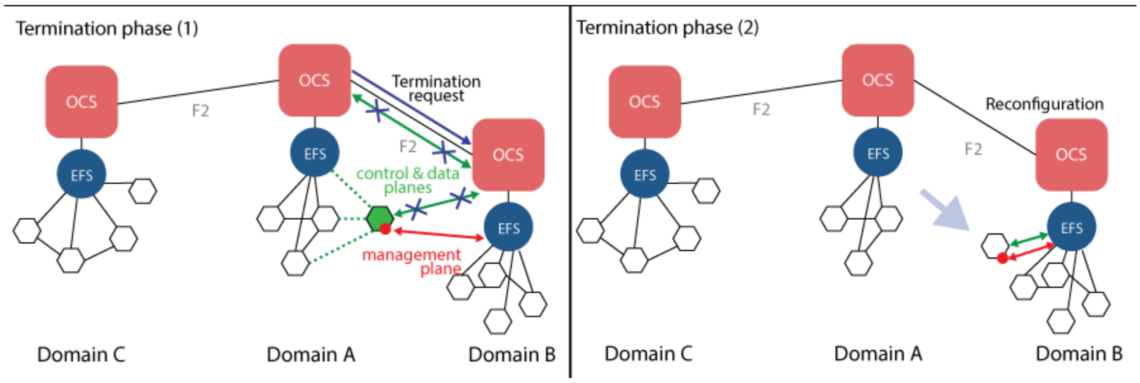

FIGURE 4-2: OCS FEDERATION INTERACTION – TERMINATION PHASE

Upon instantiation of the tunnelling (SDN-WAN) function, the EFS Application/Function Manager exchanges security parameters (e.g., security keys) or provides the ID and the IP address of the tunnelling (SDN-WAN) function to the EFS Resource Orchestrator. The EFS Resource Orchestrator provides this set of information (ID and IP address) on the F2 interface along with a confirmation that the reserved resource is ready to be federated by the consumer domain. The consumer EFS Resource Orchestrator receives the information and instructs already instantiated consumer SDN-WAN function to establish the tunnel. After the tunnel is established, the resources are federated and ready to be used by the consumer domain. The consumer EFS Resource Orchestrator sends confirmation to the provider EFS Resource Orchestrator and the charging process is initiated.

4.1.3.3 Federation termination phase

When the consumer domain wants to terminate the federation of the resources, the consumer domain sends termination request to the provider EFS-RO on F2. The provider EFS RO initiates termination of the SDN-WAN function to the local EFS Application/Function Manager. Once this operation is done, the provider EFS RO sends reconfiguration request to the VIM. Both (control and data) planes are reconfigured to retrieve the reserved resources and make them available in the local domain. The VIM notifies the provider EFS RO for concluded reconfiguration and the provider EFS RO stops the charging and/or accounting process. The provider EFS RO notifies the consumer EFS RO that the federation has terminated successfully and optionally provides the charging information.

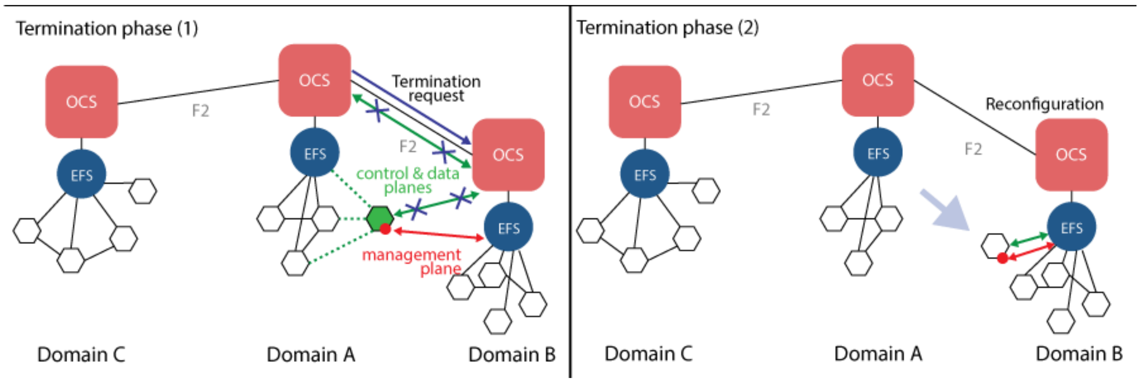

FIGURE 4-3: OCS FEDERATION INTERACTION – TERMINATION PHASE

During the termination phase parties should take into account that a federated EFS resource is stable if it can be used for an extended time so both the provider and consumer domains can benefit from it. Instability of federated EFS resource incurs high signalling costs without real benefits. There are several reasons for a federated EFS resource to be unstable. One occurs to mobile EFS nodes (fog nodes). If a fog node is a part of the EFS resource of a provider domain, offering it to a consumer domain may risk the possibility of losing connection with it possibly due to its movement.

4.1.4 Federation of resources

This subsection describes how the federation of resources is done in 5G-CORAL jointly with the designed federation interface (F2) endpoints. Figure 4-4 illustrates a sequence diagram describing the whole workflow of the static federation, including the interaction between two domains. This includes the messaging exchange between each of the OCS components involved in the federation, for both inter and intra federated domains. Additionally, in Table 4-2 and Table 4-3 describe the federation interface (F2), describing in detail the endpoints involved (e.g., action, body and description) in every of the identified federation phases (see Section 4.1.3.1, Section 4.1.3.2, and Section 4.1.3.3).

TABLE 4-2: FEDERATION ADVERTISEMENT INTERFACE ENDPOINTS

Phase	End Point	Verb	Body	Description
Advertising/Discovery/Negotiation	/federation /discover	GET	None	Retrieve active offers
		POST	offer_uuid	Starts the federation process. Generating in the provider domain an offer, which it will return jointly with an offer_uuid. The details of the offer can be the main characteristics of the offered fog node, which should be similar to the ones specified in the SLA.
		DELETE	offer_uuid	Rejects an offer.
		PUT	offer_uuid	Asks for a new node to federate, automatically rejecting the node offered.
	/federation /reserve	POST	offer_uuid	Reservation request, from an active offer. Reserve the resources, in order to be ready for the instantiation phase. Returns the confirmation.
		GET	offer_uuid	Retrieve reserved resources.

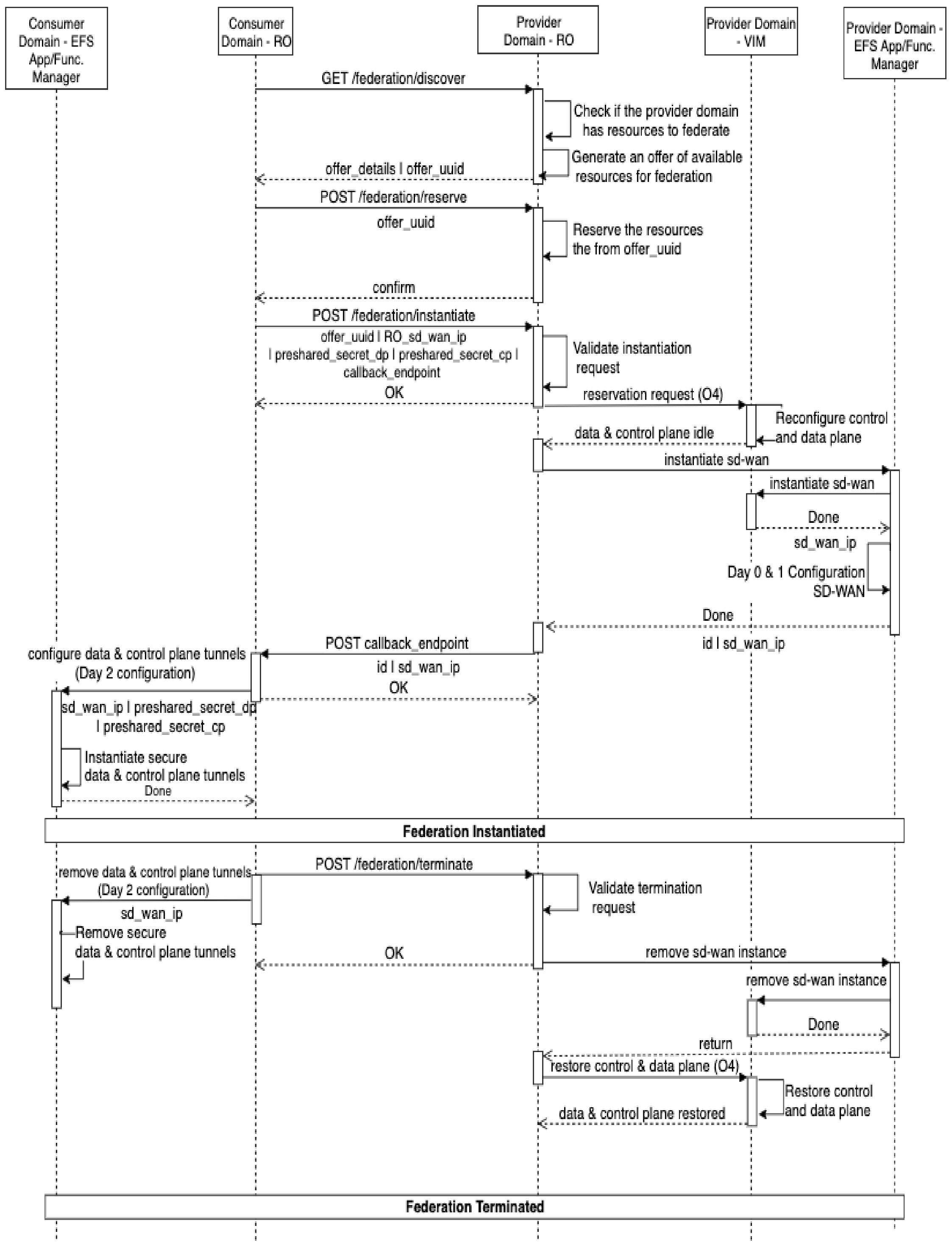
Consumer Domain - EFS App/Func. Manager
Consumer Domain - RO
Provider Domain - RO
Provider Domain - VIM
Provider Domain - EFS App/Func. Manager
GET /federation/discover
Check if the provider domain has resources to federate
Generate an offer of available resources for federation
offer_details | offer_uuid
POST /federation/reserve
offer_uuid
Reserve the resources the from offer_uuid
confirm
POST /federation/instantiate
offer_uuid | RO_sd_wan_ip | preshared_secret_dp | preshared_secret_cp | callback_endpoint
OK
Validate instantiation request
reservation request (O4)
Reconfigure control and data plane
data & control plane idle
instantiate sd-wan
instantiate sd-wan
Done
sd_wan_ip
Day 0 & 1 Configuration SD-WAN
Done
id | sd_wan_ip
POST callback_endpoint
id | sd_wan_ip
OK
configure data & control plane tunnels (Day 2 configuration)
sd_wan_ip | preshared_secret_dp | preshared_secret_cp
Instantiate secure data & control plane tunnels
Done
Federation Instantiated
POST /federation/terminate
remove data & control plane tunnels (Day 2 configuration)
sd_wan_ip
Remove secure data & control plane tunnels
Validate termination request
OK
remove sd-wan instance
remove sd-wan instance
Done
return
restore control & data plane (O4)
Restore control and data plane
data & control plane restored
Federation Terminated

FIGURE 4-4: SEQUENCE DIAGRAM FOR OCS RESOURCE FEDERATION

TABLE 4-3: FEDERATION INSTANTIATION AND TERMINATION INTERFACE ENDPOINTS

Phase	End Point	Verb	Body	Description
Instantiation	/federation /instantiate	POST	offer_uuid, RO_sd_wan_ip, preshared_secret_dp, preshared_secret_cp, callback_endpoint	Accept an offer, including some necessary details to instantiate the federation, such as where should the tunnel be created (RO_public_ip) and the control and data plane shared secrets in order to stablish all secure tunnels towards the consumer domain. Returns immediately as the instantiation process can take some time. Once instantiation process finishes, the provider domain notifies the consumer domain that everything is ready at the callback_endpoint.
		GET	offer_uuid	Retrieves current status of a federation instance identified by its offer_uuid.
Termination	/federation /terminate	POST	offer_uuid	Terminates the federation of resources identified by an offer_uuid.

4.2 Profit maximization in a federated environment

Federation of resource among multiple administrative domains is beneficial in many ways. For example, it lowers the rate of EFS resource request denial due to local resource shortage. We can turn all types of benefits into revenue and assume that the sole reason for any EFS node to participate in a federation is to maximize its profit. We shall analyze how to form profit-maximized federations among multiple administrative domains coexisting in a geographical area that are able to share EFS resource technically.

This task resolves a management-plane issue: organizing a set of EFS nodes into disjoint administrative domains (each corresponds to an EFS federation of one or more EFS nodes). Each organizing result is a partition of the set of EFS nodes called *federation structure*. The goal is to seek a federation structure that has the highest total profit. This mission faces challenges due to autonomous behavior of EFS nodes: an EFS node may join or leave a federation at its own will and may not be willing to transfer its profit to or share its profit with other members in the federation. Without agreements among participating EFS nodes, an optimal federation structure, even exists theoretically, is not *stable* and thus cannot be achieved in reality.

Possible agreements among participants include way of participation (see Section 4.1.2) and profit allocation. These agreements affect stability of the federation.

4.2.1 Instability in dynamic EFS federation

In dynamic federation, the federation structure keeps changing with the existence of roaming fog nodes. Even if all EFS nodes are stationary, the federation structure may still be unstable if EFS nodes individually form federations to maximize their own profits.

TABLE 4-4: PROFITS OF EFS NODES IN DIFFERENT FEDERATIONS

EFS node	Own profit	Profit in {A, B}	Profit in {B, C}	Profit in {A, C}	Profit in {A, B, C}
A	5	8	-	6	7
B	6	8	10	-	9
C	4	-	5	7	6

Consider a simple scenario consisting of EFS nodes A, B, and C with their profits in different federations shown in Table 4-4. Suppose that initially all EFS nodes work alone. Node A requests to form a federation with B to maximize its profit. B will accept A's proposal because B's profit can also be increased in federation {A, B}. After that, C cannot join the federation because that will decrease A's profit from 8 to 7. On the other hand, B has the incentive to leave federation {A, B} and form another federation with C. C will accept B's proposal because it will get higher profit than being working alone. After that, because now A works alone, C has the incentive to leave federation {B, C} and form another federation with A. A will accept C's proposal due to higher profit. Now it is A's turn to leave the federation and form a federation with B. The same scenario will then repeat itself.

4.2.2 Profit allocation: fairness and stability

Profit allocation mechanism allocates the total profit of a federation to each member. The allocation should reflect each member's contribution (i.e., fair) and ensure stability. A well-known mechanism is based on Shapley value [22], which accounts for marginal contribution of each member. A member's marginal contribution is the change of the total profit when it joins the federation. Formally, letting $v(S)$ be the total profit in any federation S, federation member i's Shapley value in federation F is defined as:

$$\phi_i(F) = \sum_{S \subseteq F \setminus \{i\}} \frac{|S|!\,(|F| - |S| - 1)!}{|F|!} \left(v(S \cup \{i\}) - v(S)\right). \tag{1}$$

The use of Shapley value in profit allocation achieves individual fairness. More specifically, the profit allocated to a participant is not less than the payoff when it does not participate. However, Shapley-value-based profit allocation incurs high computation cost.

Banzhaf value [23] based on marginal contribution can also be used for profit allocation that guarantees fairness. Compared with Shapley value, Banzhaf value requires less computation overhead to compute. The Banzhaf value for member i in federation F is defined as:

$$\beta_i(F) = \frac{1}{2^{m-1}} \sum_{S \subseteq F \setminus \{i\}} \left(v(S \cup \{i\}) - v(S)\right), \tag{2}$$

where $m = |F|$. The normalized Banzhaf value is defined as:

$$B_i(F) = \frac{\beta_i(F)}{\sum_{j \in F} \beta_j(F)}. \tag{3}$$

The profit of federation F that is allocated to member i is proportional to $B_i(F)$:

$$x_i(F) = B_i(F) v(F). \tag{4}$$

A federation is stable only if the profit allocation mechanism gives no member the incentive to leave the federation to work alone or join another federation. Let $F_i = \{sp_1, sp_2, \dots\}$ be the set of all members in a federation F_i. Let $v(F_i)$ be the total profit in federation F_i. Let x_j be the profit allocated to each $sp_j \in F_i$. An allocation $(x_j)_{sp_j \in F_i}$ is feasible if

$$v(F_i) = \textstyle\sum_{sp_j \in F_i} x_j. \tag{5}$$

Let vector $(x_j)_{sp_j \in F_i}$ be a feasible allocation for F_i. If there exists another feasible allocation $(y_j)_{sp_j \in H}$ for some sub-federation $H \subset F_i$ such that $y_j \geq x_j$ for all $sp_j \in F_i$ and $y_k > x_k$ for some $sp_k \in F_i$, then H has a Pareto improvement on the allocation $(x_j)_{sp_j \in H}$. The existence of a Pareto improvement on the allocation of any subset $H \subset F_i$ implies instability of the federation

F_i because all members in H could leave F_i to form a new federation without profit reduction and at least one member can receive a higher profit. In that case, we say that H *blocks* F_i. The goal of profit allocation is to find a feasible allocation for each federation F_i such that no $H \subset F_i$ can block F_i.

4.2.3 Identifying Best Federation Structure

A straightforward approach to identifying optimal federation structure is to examine every possible federation structure. This approach is not computationally efficient because the number of federation structures is an exponential function of the number of participants [24]. In fact, finding the optimal federation structures is NP-complete.

A commonly adopted approach to optimal federation structure is merge-and-split. Refer to Algorithm 4-1. The algorithm forms the initial structure S that consists of singleton federations only, where each singleton federation is an EFS node. In the merging phase, the algorithm randomly picks up a pair of federations to sees whether merging them into one is beneficial. Unlike in tradition clouds, where any two federations could be considered for possible merging, merging two federations F_i and F_j into one is beneficial only if some EFS node in F_i is able to provide its resource to another EFS node in F_j or vice versa subject to latency constraint. We define $f_{i,j} = 1$ if the request from sp_i can be served by sp_j while meeting the latency constraint t_i. Based on f, we define sharable relation $\perp$ on federations. For any two federations F_i and F_j, $F_i \perp F_j$ iff $\exists sp_p \in F_i, \exists sp_q \in F_j, f_{p,q} = 1$. Therefore, merging F_i and F_j should be considered only if $F_i \perp F_j$ or $F_j \perp F_i$. We use $\mathcal{F}$ to keep the set of all possible pairs of federations in S for which merging should be considered.

ALGORITHM 4-1: MERGE-AND-SPLIT FEDERATION FORMATION MECHANISM

```
initial state: $S \leftarrow \{\{sp_1\},\{sp_2\},\dots,\{sp_n\}\}$
repeat
    F $\leftarrow \{\{F_i,F_j\} | F_i,F_j \in S, F_i \perp F_j$ or $F_j \perp F_i\}$
    while F $\neq \emptyset$ do
        repeat
            randomly select $(F_i,F_j) \in$ F
            F $\leftarrow$ F $\setminus \{\{F_i,F_j\}\}$
        until can_merge$(F_i,F_j)$ or F $= \emptyset$
        if *can_merge*$(F_i,F_j)$ then
            $S \leftarrow S \setminus \{F_i,F_j\}$
            $S \leftarrow S \cup \{F_i \cup F_j\}$
            F $\leftarrow \{\{F_i,F_j\} | F_i,F_j \in S, F_i \perp F_j$ or $F_j \perp F_i\}$
        Endif
    end while
    *redo* $\leftarrow$ *false*
    for all $H \in S$ such that $|H| > 1$ do
        for all partitions $\{F_i,F_j\}$ of $H$ do
            if *can_split*$(F_i,F_j)$ then
                $S \leftarrow S \setminus \{H\}$
                $S \leftarrow S \cup \{F_i \cup F_j\}$
                *redo* $\leftarrow$ *true*
                Break
            endif
        end for
    end for
until $redo$ = $false$
return $S$
```

For a pair of federations F_i and F_j, function can *can_merge*(F_i, F_j) returns whether F_i and F_j should be merged together. On merging, S is updated by removing both F_i and F_j from it and adding the union of $\{F_i\}$ and $\{F_j\}$ into it. $\mathcal{F}$ is also updated accordingly.

For any two (possibly singleton) federations F_i and F_j such that $F_i \cap F_j = \emptyset$, a necessary condition for $H = F_i \cup F_j$ to be a stable federation is

$$v(H) \geq v(F_i) + v(F_j). \tag{6}$$

If (6) does not hold, either F_i or F_j blocks H for any feasible allocation for H. Even if (6) holds, whether H is stable also depends on the profit allocation for H. Let $x_k(F)$ denote the profit allocated to $sp_k \in F$. We define binary relation $\succeq$ on federations as:

$$F \succeq F' \text{ iff } \forall sp_i \in F \cap F', x_i(F) \geq x_i(F') \tag{7}$$

and also, relation $\equiv$

$$F \equiv F' \text{ iff } \forall sp_i \in F \cap F', x_i(F) = x_i(F') \tag{8}$$

Finally, $F \succ F'$ if $F \succeq F'$ and $F \equiv F'$ does not hold.

Some approaches allow merging F_i and F_j into H only if $H \succ F_i$ and $H \succeq F_j$ or $H \succ F_j$ and $H \succeq F_i$. Algorithm 4-2 allows a merging only if the merging improves every member's profit.

ALGORITHM 4-2: FUNCTION CAN_MERGE(F_i, F_j)

```
H ← $F_i \cup F_j$
for all $sp_k \in H$ do
    if $sp_k \in F_i$ and $x_k(H) \leq x_k(F_i)$ then
        return false
    else if $sp_k \in F_j$ and $x_k(H) \leq x_k(F_j)$ then
        return false
    end if
end for
return true
```

When there is no more federation pair in $\mathcal{F}$ to check, the algorithm proceeds to the splitting phase. It checks all possible partitions of every non-singleton federation H in S to see if H should be split into two subsets. Whenever a splitting occurs, the algorithm goes back to the merging phrase with the updated S.

Several conditions can be used for splitting up a federation H into two disjoint subsets F_i and F_j. The condition could be when the splitting improves at least one member's profit without decreasing any other's ($F_i \succ H$ and $F_j \succeq H$ or $F_j \succ H$ and $F_i \succeq H$) [25] when the splitting has a Pareto improvement on one subset ($F_i \succ H$ or $F_j \succ H$) [26], or when all members in one of the subsets have the same or higher profits after the splitting ($F_i \succeq H$ or $F_j \succeq H$) [27]

In our simulation, function *can_split*(F_i, F_j) returns *true* if $F_i \succ H$ and $F_j \succeq H$ or $F_j \succ H$ and $F_i \succeq H$, where $H = F_i \cup F_j$.

4.2.4 Profit-Maximizing Resource Provisioning Configuration

For a specific federation, maximizing the total profit of the federation involves configuring the allocation of resource among participated EFS nodes in the federation. This is usually a sub-problem to solve in finding out the best federation structure.

We formulate a simple profit-maximization model considering unit price and unit cost of resource usage, unit communication cost between EFS nodes, and the ability to communicate without breaking latency constraint between EFS nodes. We assume a federation of n EFS nodes $F = \{sp_1, sp_2, \ldots, sp_n\}$. Each EFS node sp_i has a resource capacity C_i with unit cost c_i. We assume that all home requests of sp_i (resource requested by EFS applications/services of sp_i) have been aggregated with total amount r_i and payment per unit of resource requested p_i. Some portion of r_i can be served by EFS nodes other than sp_i. We use $q_{j,k}$ to denote the amount of resource provided by sp_j to the home requests of sp_k. A **resource provisioning configuration** is to set up all $q_{j,k}$'s for every sp_j and sp_k in the same federation to maximize total profit.

If sp_j serves the home requests of sp_k, it incurs extra communication cost that is estimated by the amount of resource provided by sp_j to sp_k times $b_{k,j}$, the unit cost of the communication link from sp_k to sp_j. Therefore, when sp_j serves the home requests of sp_k, the unit profit is $p_k - c_j - b_{j,k}$. We define an indication variable $f_{k,j}$ to denote whether the service provided by sp_j to the home requests of sp_k meets the associated latency constraint, where $f_{k,j} = 1$ indicates 'yes' and $f_{k,j} = 0$ otherwise.

The profit of the federation F is the maximal profit that can be achieved by resource provisioning configuration:

$$v(F) = \max_{q_{j,k}} \sum_{sp_j \in F} \sum_{sp_k \in F} (p_k - c_j - b_{j,k}) \cdot q_{j,k} \cdot f_{k,j}. \tag{9}$$

The resource provisioning configuration is subject to **capacity constraint:**

$$\sum_{k=1}^{n} q_{j,k} \leq C_j,\ \forall sp_j \in F \tag{10}$$

and **demand constraint:**

$$\sum_{j=1}^{n} q_{j,k} \leq r_k,\ \forall sp_k \in F. \tag{11}$$

More constraints are possible when additional request demands and serving policies are imposed. We consider the following four possible cases (we define request's home EFS to be the EFS where the request arises).

- Case 1 local service only (LSO): Requests can only be served by home EFS nodes. There is no need to form federation because all EFS nodes work alone. That is, $q_{j,k} = 0$ for all $sp_k \neq sp_j$.
- Case 2 local service first (LSF): Requests are served by non-home EFS nodes only when home EFS does not have enough resource. On the other hand, EFS must provide enough resource to home requests before offering residual capacity to guest requests. That is, $q_{j,j} = \min(C_j, r_j)$ for all sp_j. This setting ensures that requests receive at least the same amount of resource as in LSO and the total profit is at least the same as in LSO.

- Case 3 maximal profit (MP): EFS nodes collaborate to maximize the total profit of the federation while requests can be served by any EFS nodes in the federation.
- Case 4 local resource first (LRF): Requests can be served by any EFS nodes in the federation and the maximal amount of resource that sp_j can provide to guest requests is limited by $\sum_{k\neq j} q_{j,k} \leq \max(C_j - r_j, 0)$. The limitation does not imply that sp_j should allocate $\min(C_j, r_j)$ units of resource to its home requests. Other EFS nodes in the same federation with cheaper residual resource may serve the home requests of sp_i.

4.2.5 Performance evaluation

We have conducted extensive simulations to study the performance of the merge-and-split approach to maximal-profit federation structure. The performance metrics under investigation include total profits in the federation structure and the total amount of resource allocated to requests. The four different cases of request demand and serving policies mentioned in Section 4.2.4 were tested. The details of the simulations are in Appendix 12.1.

A factor that significantly affects the results is cooperation intensity p among EFS nodes. We model EFS nodes as vertices in a directed graph, where there is an edge from nodes sp_k to sp_j if sp_k can serve sp_j's request without violating latency constraint. Cooperation intensity p is defined to be the ratio of the number of directed edges to the maximal possible number of directed edges in the graph.

4.2.5.1 Impact of cooperation intensity

Figure 4-5 and Figure 4-6 show how the total profit in the federation structure and the total amount of allocated resource changed with increasing p. Because LSO allows no resource sharing, the performance with LSO is not affected by p. The performance with all other three cases improves as p increases. Among them, the highest total profit is with MP while the largest amount of allocated resource is with LSF. The performance with LRF is between these two cases.

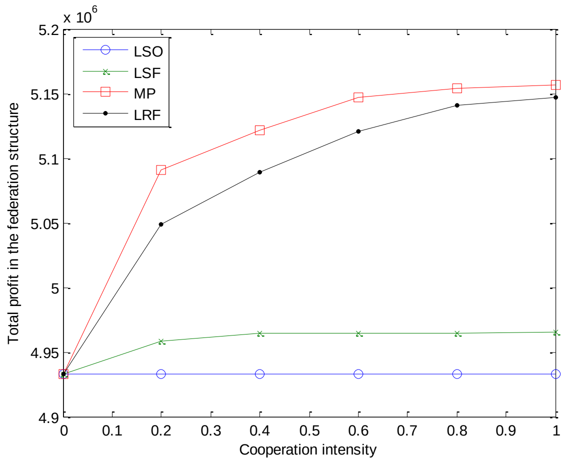

FIGURE 4-5: TOTAL PROFIT IN THE FEDERATION STRUCTURE *VS* COOPERATION INTENSITY

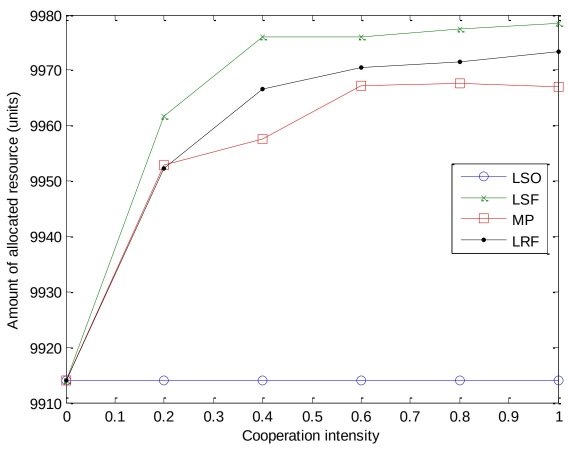

FIGURE 4-6: AMOUNT OF ALLOCATED RESOURCES IN THE FEDERATION *VS* COOPERATION INTENSITY

4.2.5.2 Impact of demand-to-supply ratio

We then study how the resource demand-to-supply ratio affects the performance. This was done by fixing the mean resource capacity to $\mu_k = 1{,}200$ units and varying the mean requested resource units μ_r from 700 to $1{,}450$ units. The results are shown in Figure 4-7 and Figure 4-8. When μ_r is less than $\mu_k = 1{,}200$, the mean capacity, the demands are lower than the supplies so both the total profit and the amount of allocated resource increase linearly as μ_r increases. When $\mu_r \geq \mu_k$, the amount of allocated resource is limited by μ_k. Still, the total profit could be improved by appropriate **resource provisioning configuration** as MP demonstrates in Figure 4-8.

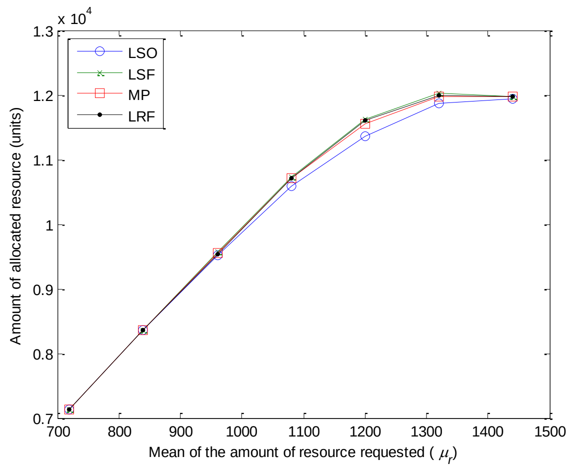

FIGURE 4-7: AMOUNT OF ALLOCATED RESOURCE IN THE FEDERATION *VS* MEAN UNIT OF RESOURCES REQUEST

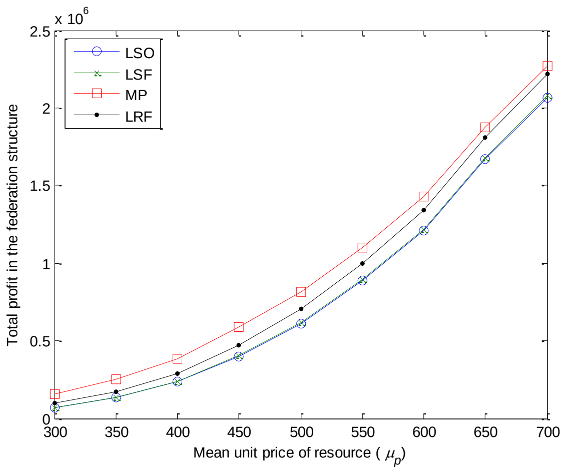

FIGURE 4-8: TOTAL PROFIT IN THE FEDERATION *VS* MEAN UNIT PRICE OF RESOURCES

4.2.5.3 Impact of price-to-cost ratio

We next investigate the impact of the price-to-cost ratio on performance. This was done by fixing all parameters but μ_p, the mean unit price of resource (the mean unit cost of resource was set to $\mu_c = 500$). From the result shown in Figure 4-9, we can see that resource requests are generally fulfilled with LSO and LSF. On the other hand, more profits can be earned with MF and LRF (see Figure 4-8). The extra profits come at the cost of low request acceptance rates. The cost is particularly significant when the price-to-cost ratio is low.

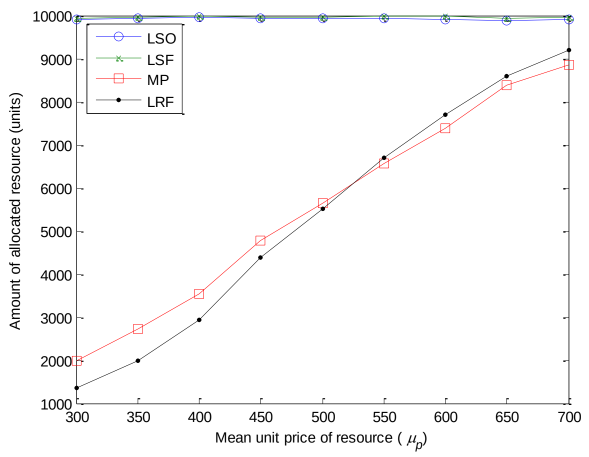

FIGURE 4-9: AMOUNT OF ALLOCATED RESOURCE IN THE FEDERATION STRUCTURE VS. MEAN UNIT PRICE OF RESOURCE

4.2.5.4 Conclusions

The results showed that federation always increases profits. Maximal profits can be earned with MP but sometimes at the cost of reduced amount of allocated resource (when the price-to-cost ratio is low, EFS nodes would rather not serve low-price requests). With LSF, the amount of allocated resource is not lower than the case of no federation (i.e., LSO) yet the profits can potentially be improved. This serving policy is thus recommended.

4.3 Advanced resource provisioning in federated EFSs

In Section 4.2.4, we use a simple model for the problem of finding resource provisioning configuration that maximizes the total profit in an EFS federation. In this section, we further extend this model by considering the following settings:

- **Different resource types:** EFS nodes own different types of physical resource (e.g., CPU, memory, storage, etc.) and provide different flavours of virtualized resource (e.g., different instance types of virtual machines) to EFS applications/services.
- **Multi-objective:** The configuration maximizes not only the service provider's profit but also the user's payoff (i.e., considering the quality of the service offered to requests and also possible payment).
- **Distributed dispatch:** Resource requests are directly sent to target EFS nodes. There is no central entity that dispatches all requests toward their target EFS nodes.
- **Different pricing models:** We consider two pricing models: free-of-use and pay-per-use. The former does not involve monetary exchange and is considered the default model for resource provisioning within a federation. The latter case suitably applies to resource provisioning across different federations. We also consider negotiable payments between resource requestors and providers.

4.3.1 System model

We consider a federation of EFS nodes $F = \{sp_1, sp_2, \dots, sp_n\}$. Each EFS node sp_i is a single computing substrate located in the same geophysical area. Let R denote the set of all different types of physical resource (CPU, memory, storage, etc.). If we exclude special hardware resource, R is universally defined for all nodes. Let C_i^r denote the amount of resource type $r \in R$ at node sp_i. We denote the capacity of node sp_i by $\mathbf{C}_i = (C_i^1, C_i^2, \dots, C_i^{|R|})$.

Virtualized computation resource could be in the form of virtual machine (VM), container, or others. We assume the use of VM and a limited number of VM instance types (called flavours), which has been supported by cloud service providers. Table 4-5 shows the VM instances types offered by Amazon EC2 in US West Region[8].

TABLE 4-5: EXAMPLES OF VM INSTANCE TYPES

Metric	Medium (m=1)	Large (m=2)	XLarge(m=3)	2XLarge (m=4)
CPU	1	2	4	8
Memory (GB)	3.75	7.5	15	30
Storage (GB)	4	32	80	160

As a need to deploy EFS functions/applications, requests for virtualized resource will come to the EFS Resource Orchestrator. In general, each request includes an EFS Stack Descriptor that consists of the following parameters:

- a list of VM instances requested together with corresponding images;
- a directed graph that describes the chaining of these VMs;
- optionally a location indicator that specifies a certain point or area to deploy each VM;
- a latency constraint associated with the whole request.

[8] https://aws.amazon.com/ec2/instance-types/

In reality, EFS functions or applications may demand computation resource located at different geographical areas so that the request should be simultaneously served by more than one node (instead of one). In that case, the EFS Resource Orchestrator is in charge of splitting the request into multiple parts, one toward each node.

In the trust cooperative model, each EFS node may receive requests for virtualized resource from EFS functions or applications within its administrative domain (called home requests) or from other EFS nodes in the same federation (called guest requests). There are several possible policies for EFS Resource Orchestrator to handle incoming requests. For example:

- [P1] Treating home and guest requests equally;
- [P2] Granting home requests first and then allocating residual capacity to guest requests. This is identical to local service first (LSF) in Sec. 4.2.4;
- [P3] Granting home requests first and reserving a portion of the capacity for future home requests. If there is still residual capacity, then allocate it to guest requests.

We consider a general model with which EFS Resource Orchestrator can take any one of these policies.

4.3.2 Request dispatch by OCS

Suppose that the federated EFS system F receives a set of m requests $Q = \{q^1, q^2, \dots, q^m\}$. For each request q^j, all the EFS nodes that currently has sufficient resource capacity to serve it and meets the location and latency constraint are *qualified* nodes for q^j. Dispatching resource requests to qualified nodes is straightforward if every node has enough capacity to serve all requests toward it. If this is not the case, only some requests can be granted. The selection of requests to grant is to maximize the number of *targeted requests*. The definition of targeted requests depends on the serving policy (P1~P3) taken by each EFS node.

- If an EFS node takes P2 or P3, only home requests are targeted.
- If an EFS node takes P1, all requests are targeted.

The optimization problem is closely related to *bin-packing problem*, where objects of different volumes (resource requests in our case) are to be packed into a finite number of bins (EFS nodes in our case) each of same volume. The bin-packing problem has been known NP-hard. The following features differentiate the dispatch problem from the bin-packing problem.

- Nodes in the dispatch problem are not of the same capacity;
- Not all requests can be served even if all nodes are used, and we aim to maximize the number of requests served. In the bin-packing problem, all objects can be packed, and the goal is to minimize the number of bins used;
- Not every qualified node offers a request the same quality of service (QoS; e.g., application latency). We want to dispatch requests to qualified nodes that offer them QoS as high as possible.

4.3.3 Objectives of payment-free request dispatch

We consider the objective of maximize the number of requests granted in parallel with the objective of offering requests QoS as high as possible. This is a multi-objective optimization problem, for which optimal solutions are computationally difficult to find. We decompose it into two sub-problems.

Let the dispatch result of Q to F be represented by a set of indication variables $\{x_i^k\}_i^k$, where $x_i^k = 1$ if request q^k is dispatched to node sp_i and $x_i^k = 0$ otherwise. If all requests are targeted, the objective of OCS is to maximize the total number of granted requests:

$$\max \sum_i \sum_k x_i^k. \tag{12}$$

The objective of each request $q^k \in Q$ is to maximize its own spare latency:

$$\max \sum_i \left(x_i^k \times \left(t_{\max}^k - t_i^k\right)\right), \tag{13}$$

where $t_{\max}^k$ is the latency constraint associated with q^k and t_i^k is the estimated latency when q^k is served by sp_i. These two objectives are subject to capacity constraint and non-split constraint (requests cannot be split and can be dispatched to at most one node). Let $d^{k,r}$ be the amount of physical resource type $r \in R$ needed by request q^k. The capacity constraint is:

$$\sum_k \left(d^{k,r} \times x_i^k\right) \leq C_i^r, \forall r \in R,\ \forall sp_i \in F. \tag{14}$$

The non-split constraint is:

$$\sum_i x_i^k \leq 1, \forall\, q^k \in Q \tag{15}$$

$$\forall\, x_i^k \in \{0,1\} \tag{16}$$

4.3.4 Procedure for payment-free request dispatch

We propose a distributed on-line approach where each node locally and independently selects requests to serve. The role of OCS is to identify for each request all qualified nodes with ranks determined by the QoS they offer and communicate with nodes on behalf of each request. The procedure of this approach is as follows:

1. After the OCS receives a request, it forwards the request to each node;
2. Each node checks to see if it is qualified for the request. A node is qualified if it has enough capacity to serve the request and the service meets the latency constraint associated with the request. If a node is qualified, it also estimates the resulting latency. The node then sends back the result to the OCS;
3. After all nodes reply back their results, the OCS creates a preference list which ranks all qualified nodes for the request;
4. When the OCS has a set of requests to dispatch, each with a preference list, the OCS sends all requests in parallel to their most preferred nodes;
5. Each node may receive more than one requests and may need to select some requests to serve. Based on its own decision, the node responds with either a grant or a reject message to each request;
6. The request procedure completes when a request receives a grant. When a request receives a reject instead, it removes the node from its preference list. If the list is not empty, go to Step 4. Otherwise, the request terminates without being served.

This procedure is a many-to-one matching proposed for college admissions problem. The difference is that each college has a fixed and known quota (for students) while nodes in our problem do not: the number of requests that can be served by a node actually depends on the amount of resource requested and the node's capacity.

4.3.4.1 Requirements for a node being qualified

A node checks whether the latency constraint is met by estimating the communication latency between chaining VMs and also access delay. The procedure to verify whether a node sp_i has enough capacity to serve request q^k follows. First, the amount of VM instances of each type requested by q^k is summarized as $\mathrm{vm}^k = (vm^{k,1}, vm^{k,2}, \dots, vm^{k,|VM|})$, where $vm^{k,j}$ is the requested number of VM instances of type j and $|VM|$ is the number of VM types supported. Since each VM instance type demands a specific amount of physical resource of various types (as Table 4-5 shows), OCS then converts vm^k into a demand vector $\mathrm{d}^k = (d^{k,1}, d^{k,1}, \dots, d^{k,|R|})$, in which $d^{k,r}$ specifies the amount of physical resource type $r \in R$ needed by request q^k. Node sp_i has enough capacity to serve q^k if $\mathrm{C}_i \geq \mathrm{d}^k$.

4.3.4.2 Node's preference on requests

The global objective of maximizing the total number of granted targeted requests (12) is decomposed into individual goal of each node: maximizing the total number of locally granted targeted requests. A greedy approach is to serve requests with lowest resource demands first. This corresponds to a preference function $P_{i,j}(q^k)$ of each node sp_i defined on request q^k:

$$P_i(q^k) = \sum_{r=1}^{|R|} \left(w_i^r \times \left(1 - \frac{d^{k,r}}{C_i^r} \right) \right) \tag{17}$$

where $\sum_r w_i^r = 1$. Parameter w_i^r is a weight that indicates the relative importance (or scarceness) of physical resource type r among all at node sp_i. The summation of all the weights at the node equals one. The term $d^{k,r}/C_i^r$ represents the ratio of the amount of physical resource type r demanded by q^k to sp_i's capacity. For example, if two CPU cores are requested by q^k and sp_i's capacity of CPU cores is four, then the ratio is 0.5.

4.3.4.3 Request's preference on nodes

Based on the results sent back by all qualified nodes, OCS forms a latency vector $\mathrm{t}^k = (t_1^k, t_2^k, \dots, t_n^k)$ for request q^k, where t_i^k is the estimated latency when sp_i serves q^k. With t^k, OCS creates a preference list for q^k based on the following preference function:

$$P^k(sp_i) = t_{\max}^k - t_i^k, \tag{18}$$

where $t_{\max}^k$ is the latency constraint associated with q^k. All requests prefer nodes with high spare latencies.

4.3.5 Payment-Based Request Dispatch

We consider the case that guest requests need pay to EFS service providers for allocated resource. This corresponds to inter-EFS request dispatch. We consider pay-per-use pricing model with dynamic pricing. EFS service providers here are resource sellers while requests are buyers. The selling prices are negotiated between the selling and the buying parties. This is more economically efficient than fixed pricing because resource price is set according to the forces of demand and supply.

For this problem we have the following assumptions:

- A set of m requests $Q = \{q^1, q^2, \dots, q^m\}$ coming to the federated system.
- Each request $q^k \in Q$ is associated with a demand vector $\mathrm{d}^k = (d^{k,1}, d^{k,2}, \dots, d^{k,|R|})$, in which $d^{k,r}$ specifies the amount of physical resource type $r \in R$ needed by request q^k.
- Each request q^k has a budget v^k, which is the maximal price that the requester is willing to pay for q^k. This value is private and not known by EFS service providers.
- $t_{\max}^k$ is the latency constraint of q^k.

- Besides resource descriptor, each request q^k to a qualified EFS system sp_i also includes an offered price b_i^k that is specific to sp_i. The value of b_i^k is related to the QoS provided by sp_i to q^k, and does not exceed v^k.
- t_i^k is the estimated latency when sp_i serves q^k.
- If request q^k with offered price b_i^k is rejected by node sp_i, it can either raise its offered price to $b_i^k + \varepsilon \leq v^k$ and resubmit the request to sp_i again, or sends the request with another offered price to another node.
- C_i^r is the amount of resource type $r \in R$ in EFS system sp_i. We denote the capacity of node sp_i by $\mathbf{C}_i = (C_i^1, C_i^2, \ldots, C_i^{|R|})$.
- Each EFS system sp_i also keeps the unit cost of each resource type by vector $\mathbf{c}_i = \{c_i^1, c_i^2, \ldots, c_i^{|R|}\}$, where c_i^r is the unit operation cost of resource type r in sp_i.

4.3.5.1 Objectives

Let the dispatch result between Q and SP be represented by a set of indication variables $\{x_i^k\}_i^k$, where $x_i^k = 1$ if request q^k is dispatched to EFS system sp_i and $x_i^k = 0$ otherwise. Let $\theta_i(q^k) = c_i^1 \times d^{k,1} + c_i^2 \times d^{k,2} + \cdots + c_i^{|R|} \times d^{k,|R|}$ be the cost of EFS system sp_i when it serves request q^k. If all requests are targeted, the objective of OCS i is to maximize its own profit defined as:

$$\sum_k \left(x_i^k \times \left(b_i^k - \theta_i(q^k)\right)\right). \tag{19}$$

Each request q^k aims to minimize its payment and also latency. A possible objective function can be defined as to maximize its payoff $(v^k - b_i^k)$ per unit latency:

$$\max \sum_i \left(x_i^k \cdot \frac{v^k - b_i^k}{t_i^k}\right). \tag{20}$$

These two objectives are subject to capacity constraint (21):

$$\sum_k (d^{k,r} \times x_i^k) \leq C_i^r, \forall r \in R,\ \forall sp_i \in SP, \tag{21}$$

non-split constraint (22, 23):

$$\sum_i x_i^k \leq 1, \forall\, q^k \in Q, \tag{22}$$

$$\forall\, x_i^k \in \{0,1\}, \tag{23}$$

and budget constraint (24):

$$0 < b_i^k \leq v^k. \tag{24}$$

4.3.5.2 EFS's preference on requests

The objective of each OCS i is to maximize its own profit as defined in (19). Because different requests come with different sizes (amounts of resource requested) with different offered prices, this falls into the 0/1-knapsak problem, which has been known NP-complete. A common-adopted greedy approach is to serve first requests with the highest ratio of profit to the amount of demanded resource. This corresponds to a preference function $P_i(q^k)$ of each EFS system sp_i defined on request q^k:

$$P_i(q^k) = \frac{b_i^k - \theta_i(q^k)}{\sum_{r=1}^{|R|}\left(w_i^r \cdot \frac{d_i^{k,r}}{C_i^r}\right)}, \tag{25}$$

where $\sum_r w_i^r = 1$.

4.3.5.3 Request's preference on nodes

Request q^k's preference on EFS system sp_i depends on whether the estimated latency t_i^k exceeds the latency constraint $t_{\max}^k$. If it does not, the preference value is defined to be its payoff $(v^k - b_i^k)$ per unit of latency. Otherwise, the preference is negative (say, -1). Formally:

$$P^k(sp_i) = \begin{cases} \frac{v^k - b_i^k}{t_i^k} & \text{if } t_i^k \leq t_{\max}^k, \\ -1 & \text{otherwise.} \end{cases} \tag{26}$$

4.3.6 Performance Evaluation

We conducted a series of simulations to investigate the performance of the proposed mechanisms and compare it with that of others. We considered request dispatches both with and without payments. The details of the simulations are in Appendix 12.2.

4.3.6.1 State-of-the-Art Mechanisms Tested

We tested several matching mechanisms, including Capacitated House Allocation (CHA) [28], adapted Boston [29], and adapted Deferred Acceptance (DA) [30] CHA is to allocate a set of houses to a bunch of agents. Every house has a capacity which specifies the maximal number of agents that it can accommodate, and agents can have preference on houses. CHA does not well fit our problem due to the following reasons. First, CHA considers only one-sided preference while both requesters and nodes have preference in our problem. Second, agents are assumed to have equal size and the maximal number of agents that can be accommodated in each house is fixed and known. In contrast, requests come with different sizes (amounts of requested resource) in our problem so an EFS node may fulfill the aggregated demand of three requests but not that of another two.

In Step 5 of the procedure shown in Section 4.3.4, an EFS node may have already accepted some requests but have to reject some other requests later due to insufficient residual capacity. When this happens, it is an issue whether the EFS node should retract a previous grant to make room for a new request simply because the new one has a higher preference function value than the previous. If we allow retraction, it is a variant of deferred-acceptance (DA) algorithm [30], which possesses a property that nodes may tentatively accept requests. If acceptance is always firm (cannot be retracted), the approach is a variant of Boston [29].

We also tested random matching and no offloading (denoted by No-Share). In No-Share, requests were always dispatched to the EFS nodes co-located with the respective serving access points of the requests.

4.3.6.2 Results of payment-free request dispatch

Figure 4-10 shows how the number of served requests changes with increasing number of requests using different approaches. DA clearly outperforms all others, followed by Boston. CHA and Random roughly performed the same. They performed better than No-Share only with few requests. The reason is that nodes in CHA did not have preference on requests, so the set of requests that were granted when a node did not have enough capacity was not carefully determined. This is like Random.

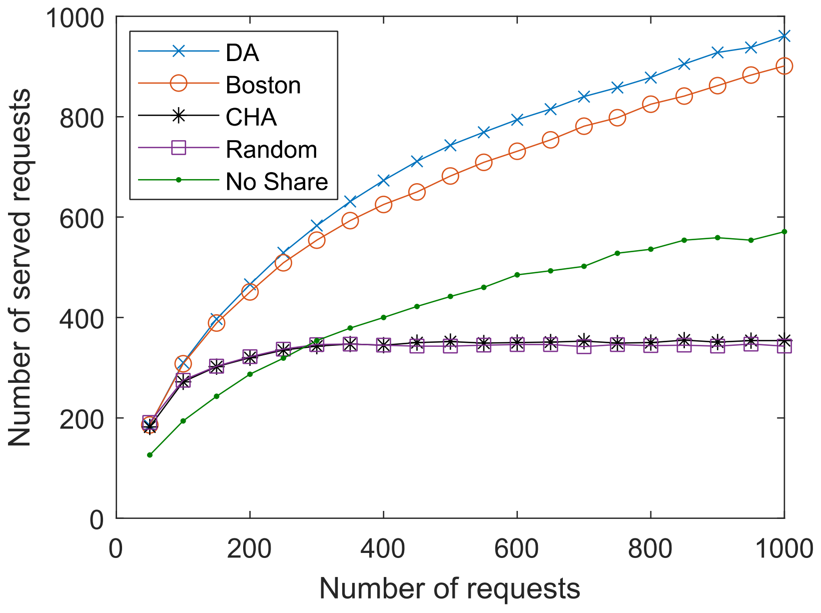

FIGURE 4-10: TOTAL NUMBERS OF SERVED REQUESTS IN PAYMENT-FREE REQUEST DISPATCH

Figure 4-11 shows the average latency per granted request. Here No-Share had the lowest latency, which is reasonable because only local (home) requests could be granted. Random had the highest latency, which is also predictable. The superiority of Boston over DA comes from the property that once Boston grants a request, it never retracts the grant. Therefore, granted requests tended to be dispatched to their most preferred EFS nodes. In contrast, DA may retract a request grant to make room for another request that is preferable. Therefore, granted requests were less likely to be matched to their most preferred nodes. Together with Figure 4-10, we can see that this strategy is to trade requester's preference for node's preference.

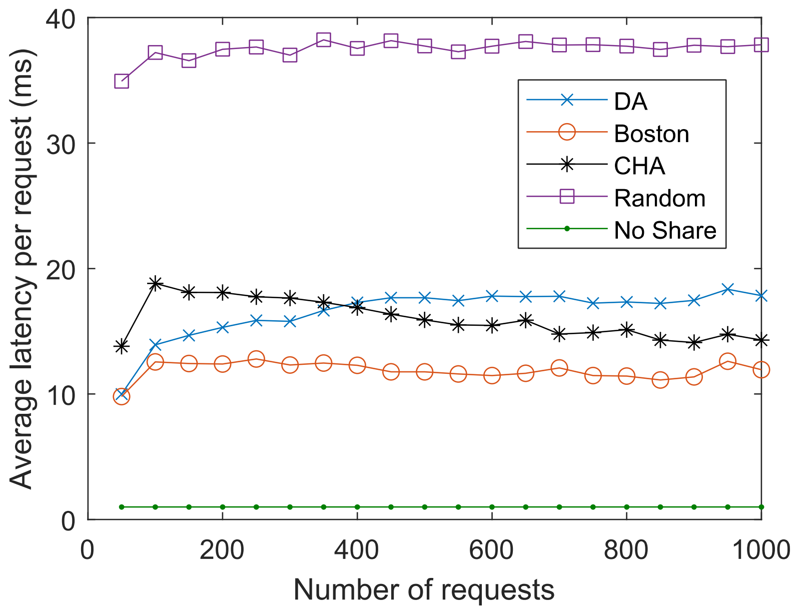

FIGURE 4-11: AVERAGE LATENCY PER REQUEST IN PAYMENT-FREE REQUEST DISPATCH

4.3.6.3 Results of payment-based request dispatch

For payment-based request dispatch, we primarily considered DA with transfer [31] (referred to as DA-T). In DA-T, different requesters may have different settings on the increments of their bids (the value of δ) when the proposed bids are not accepted. Generally speaking, nodes prefer higher δ value while requesters prefer lower. We used a parameter λ to set up the maximal number of times that each requester is allowed to raise its bid toward the same node. It indirectly controls the granularity of $\delta_{i,j}^{k}$ (δ for each q_i^k toward $s_{i,j}$) as follows:

$$\delta_{i,j}^{k} = \frac{v_i^k - a_{i,j}^k}{\lambda}, \tag{27}$$

where $a_{i,j}^k$ is the asked price $s_{i,j}$ provided to q_i^k (the minimal selling price). In the simulations, we assumed that $a_{i,j}^k = \theta_{i,j}(q_i^k)$.

Figure 4-12 shows the average latency per granted request in payment-based request dispatch. The performance of Random and Boston was expected. DA-T with $\lambda = 10$ had a lower latency than DA-T with $\lambda = 4$. This can be justified as a small granularity of bid increment ($\lambda = 10$) gave requests more chances to be considered by their most preferred nodes (before switching to less preferred servers in their preference lists).

Figure 4-13 shows total revenue of the system. Though Boston gave granted requests low latencies, the revenue of the system was nearly the same as Random. The reason is that it did not give EFS nodes the opportunity to replace low-profit requests with high-profit ones. DA-T outperformed Boston because it allows such replacements. Here large granularity of bid increment ($\lambda = 4$) gave the system higher revenue, which is intuitive. However, the gap is not significant.

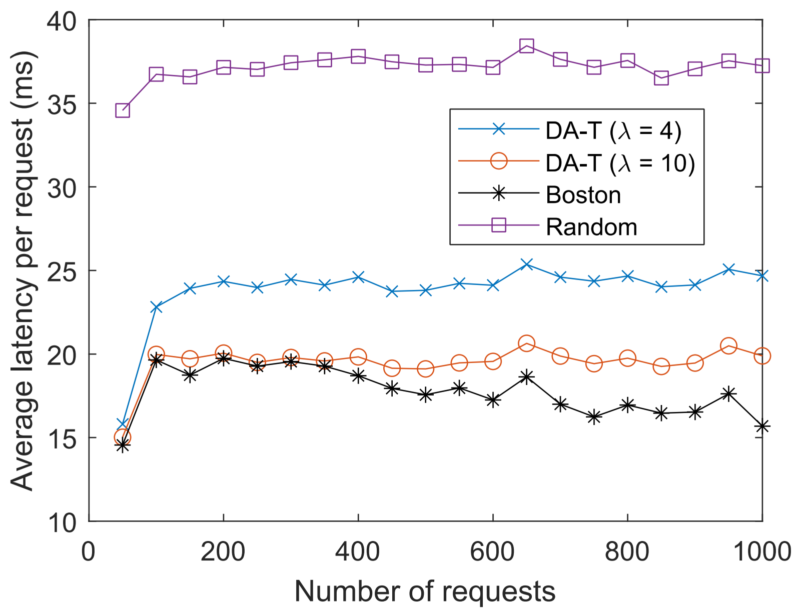

FIGURE 4-12: AVERAGE LATENCY PER REQUEST IN PAYMENT-BASED REQUEST DISPATCH

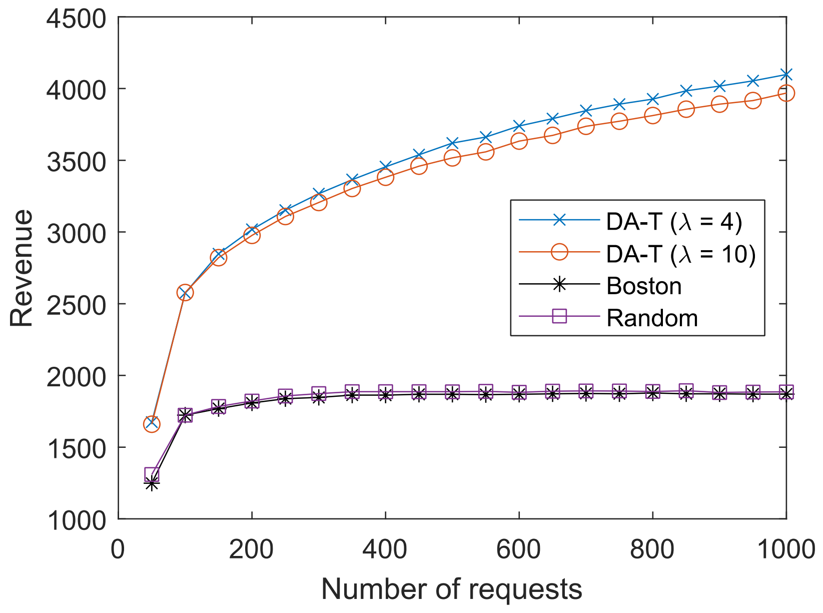

FIGURE 4-13: TOTAL REVENUE IN PAYMENT-BASED REQUEST DISPATCH

4.3.6.4 Conclusions

The results showed that in payment-free request dispatch, good dispatch approaches can serve more requests while still meeting latency constraints. Among them, DA serves more requests than the counterparts. For payment-based request dispatch, good dispatch mechanism like DA-T can have high revenue while still meeting latency constraints.

5 OCS experimental validation

This section presents the experimental validation of the OCS designed in WP3 with focus on the automated deployment of EFS Entities, OCS federation, migration of EFS Entities and network assisted D2D communication.

5.1 Automated deployment

In this section, we present and discuss the approach adopted to validate the automated deployment of services and applications within the 5G-CORAL platform. Such capability enables the so-called zero-touch deployment, which creates and manages the end-to-end service by reducing the need for human operator intervention. In 5G-CORAL, this operation translates into deploying an EFS Stack which implies the onboarding and the instantiation of each EFS Entity and Service included in the descriptor. Figure 5-1 illustrates all the steps involved.

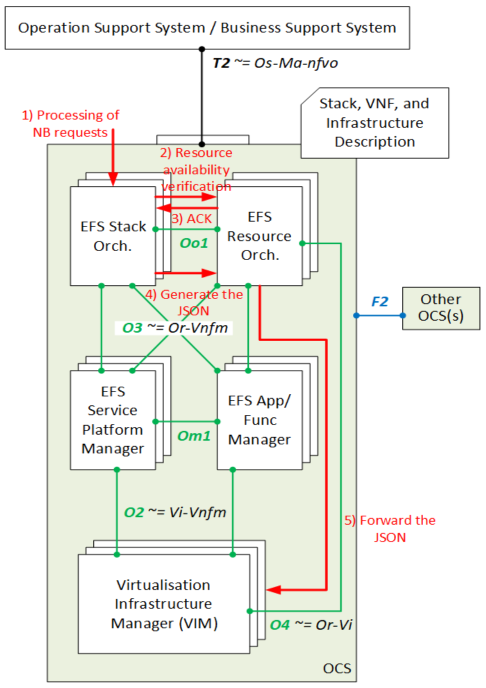
Operation Support System / Business Support System
T2 ~= Os-Ma-nfvo
Stack, VNF, and Infrastructure Description
1) Processing of NB requests
2) Resource availability verification
3) ACK
EFS Stack Orch.
EFS Resource Orch.
Oo1
4) Generate the JSON
O3 ~= Or-Vnfm
F2
Other OCS(s)
EFS Service Platform Manager
Om1
EFS App/ Func Manager
O2 ~= Vi-Vnfm
5) Forward the JSON
Virtualisation Infrastructure Manager (VIM)
O4 ~= Or-Vi
OCS

FIGURE 5-1: WORKFLOW FOR ON-BOARDING AND INSTANTIATING AN EFS STACK

During the first phase, the EFS Stack Orchestrator (SO) processes north-bound App onboarding requests sent by the OSS (1). Next, the EFS SO verifies that the EFS platform contains sufficient resources to onboard the App by querying the EFS Resource Orchestrator (RO) (2). Once the EFS SO has received a positive acknowledgment from the EFS RO (3), a JSON network descriptor is generated and sent to the EFS RO (4). Finally, the EFS RO forwards the App instantiation request to the VIM (5), which creates the EFS App instance.

In order to validate and assess the automated deployment procedure, we evaluate multiple software implementations and virtualization technologies as well as different OCS components. Particularly, Table 5-1 shows the Hypervisors, VIMs and Orchestrators under test. Regarding the hypervisors, we consider two container-based hypervisors (i.e., Docker and LXD) and one virtual machine-based hypervisor (i.e., KVM). Regarding the VIMs, we consider fog05 and OpenStack while for the Orchestrator we consider f0rce and Kubernetes (k8s). It is worth highlighting that fog05 and f0rce refer to the VIM and Orchestrator implementations following the 5G-CORAL OCS design guidelines and developed as part of WP3 work. The code of fog05 and f0rce have been made available as open source on GitHub [4][5].

TABLE 5-1: OCS SOFTWARE IMPLEMENTATION DETAILS AND COMPONENTS UNDER TEST

Component	Software Implementation	Version
Hypervisor	Docker	18.06.1
	KVM	2.5.0
	LXD	3.12
VIM	fog05	0.2
	OpenStack	Ocata
Orchestrator	f0rce	0.1
	k8s	1.13

To properly compare the above implementations, we consider the deployment of an EFS Stack composed of a single atomic EFS App. The reason of choosing such scenario is driven by the fact that such scenario is the minimal set of functionalities supported by all the software implementations under test. By doing so, the contribution of each OCS component to the overall deployment time can be clearly isolated as explained later in Section 5.1.1. The baseline image for the EFS App is Alpine Linux, a minimal Linux distribution suitable for virtualization and cloud alike environments which is available for Docker, LXD and KVM runtimes. In our tests, we have installed a webserver (i.e., NGINX) in every image as exemplary software that can interact over the network. Table 5-2 reports the image size for the three different virtualization technologies.

TABLE 5-2: EFS APP CHARACTERISTICS AND CONFIGURATIONS

Virtualization technology	Image version	Image size
Docker	Alpine Linux	23.5 MB
LXD	Alpine Linux	4.7 MB
KVM	Alpine Linux	242.1 MB

In case of using f0rce as orchestrator, we have created the EFS Stack descriptor as shown in Table 5-3. The EFS Stack descriptor follows the information model detailed in Appendix 10 and it is provided to the EFS Stack Orchestrator for onboarding and instantiation. Figure 5-2 shows the web-based interface of the EFS Stack Orchestrator which can be used by a human operator to onboard and trigger the instantiation of the EFS Stack. In case of the other hypervisors, VIMs and Orchestrators, we used instead the exposed APIs to achieve the automated deployment.

TABLE 5-3: EFS STACK DESCRIPTOR FOR INSTANTIATING THE LXD-BASED EFS APP ON F0RCE

```
{
    "efs-app-descriptor": {
        "uuid": "fc958662-ccec-4791-a082-0330c02b08b7",
        "name": "test_app",
        "vendor": "UC3M",
        "version": "1.0",
        "soft-version": "",
        "ocs-version": "1",
        "vdus": [
            {
                "vdu_uuid": "d94d309d-8414-4717-879b-8f5a98efc130",
                "vdu_name": "example_vdu_alpine",
                "vdu_image": {
                    "uri": "file:///home/user/bench.tar.gz",
                    "checksum":
"769d3b2c476c46b9dd57e280821bdd6a1694a8f643247b4d70096643f6d5d472",
                    "format": "tar.gz"
                },
                "vdu_computational_requirements": {
                    "cpu_arch": "x86_64",
                    "cpu_min_count": 1,
                    "ram_size_mb": 128.0,
                    "storage_size_gb": 1.0
                },
                "vdu_interfaces": [
                    {
                        "name": "eth0",
                        "is_mgmt": false,
                        "mac_address": "be:ef:be:ef:00:01",
                        "virtual_type": "PARAVIRT",
                        "internal_cp": ""
                    }
               ],
               "vdu_hv_type": "LXD",
               "vdu_internal_connection_points": [],
               "vdu_depends_on": [],
               "vdu_lcm_hooks": {
                      "migration_type": "COLD"
               }
            }
        ],
        "virtual_links": [],
        "service-produced": [],
        "description": "Simple deployment"
    }
}
```

The infrastructure is composed by one EFS Resource acting as compute node and by a second EFS Resource acting as controller. In order to compare the different orchestrator and VIM implementations (see Table 5-1) on the same hardware, both EFS Resources are equipped with an Intel Xeon E5-2620 (i.e., 32 logical cores) running at 2.1GHz, 128 GB of RAM, 512 GB of storage, and 2 Ethernet interfaces at 10 Gbps. It is worth mentioning that the usage of more constrained resources would have made impossible a direct comparison between different software with high computing requirements. Finally, both EFS Resources run Ubuntu 18.04 Server.

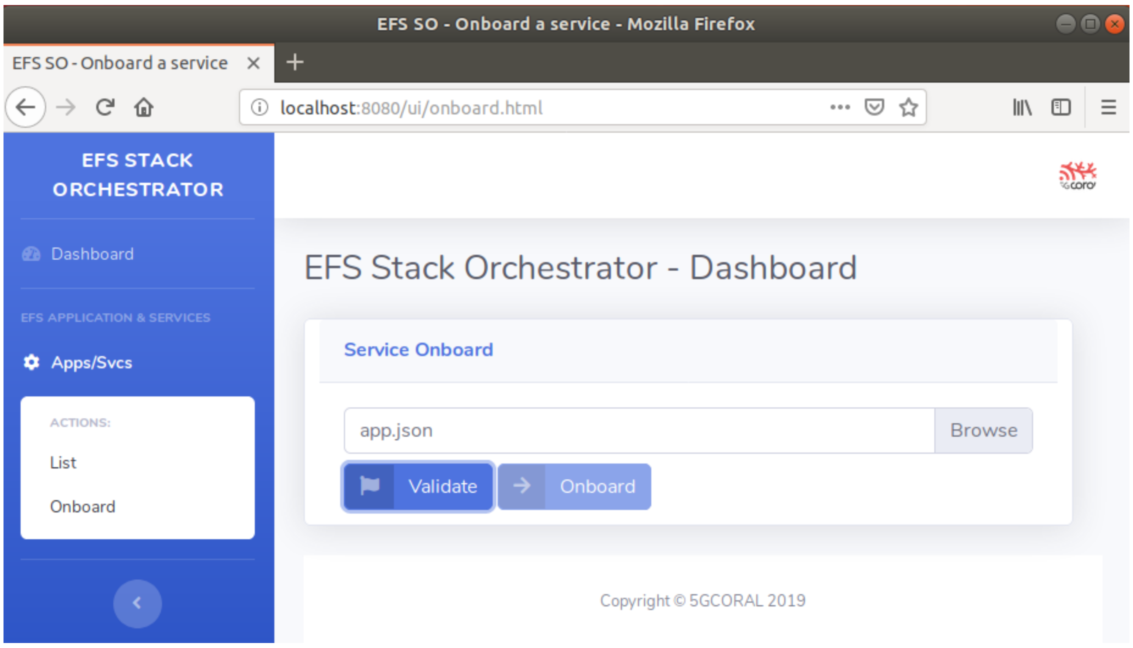
EFS SO - Onboard a service - Mozilla Firefox
EFS SO - Onboard a service
localhost:8080/ui/onboard.html
EFS STACK ORCHESTRATOR
Dashboard
EFS APPLICATION & SERVICES
Apps/Svcs
ACTIONS:
List
Onboard
EFS Stack Orchestrator - Dashboard
Service Onboard
app.json
Browse
Validate
Onboard
Copyright © 5GCORAL 2019

FIGURE 5-2: EFS STACK ORCHESTRATOR WEB-BASED INTERFACE

5.1.1 Results

For each Hypervisor, VIM and Orchestrator reported in Table 5-1, we have performed 250 automated deployments of the baseline Alpine Linux image. For each deployment we have then measured the overall deployment time, which is the time elapsed from the initial instantiation request sent to the Hypervisor/VIM/Orchestrator till the moment the webserver running inside the EFS App becomes reachable.

Moreover, to normalize the deployment time we already make available a copy of the image on the compute node. By doing so, the OCS does not need to copy the EFS App image over the network. Additionally, by employing a single compute node, the results better highlight the impact of the virtualization technology independently of the overhead that might be introduce by the placement algorithm when a larger number of nodes is used. As a result, the measured deployment time can be considered as a lower bound.

Figure 5-3 reports the experimental Cumulative Density Function (eCDF) for each of the configurations and tests. From top to bottom, first the results regarding the Docker virtualization technology are presented. In this scenario there are two parallel settings: fog05 + Docker and k8s + Docker. Specifically, the software implementations under test are Docker as hypervisor, fog05 as VIM, and Kubernetes (k8s) as Orchestrator. It is worth highlighting that k8s here is considered as Orchestrator, however k8s provides the full stack (VIM + Orchestrator). The figure in the middle reports the results for the LXD virtualization technology. In this case, the software implementations under test are LXD as hypervisor, fog05 as VIM, and f0rce as Orchestrator. The last figure at the bottom reports the results for the KVM virtualization technology. In this case we the software implementations under test are KVM as hypervisor, fog05 as VIM, and OpenStack as second VIM.

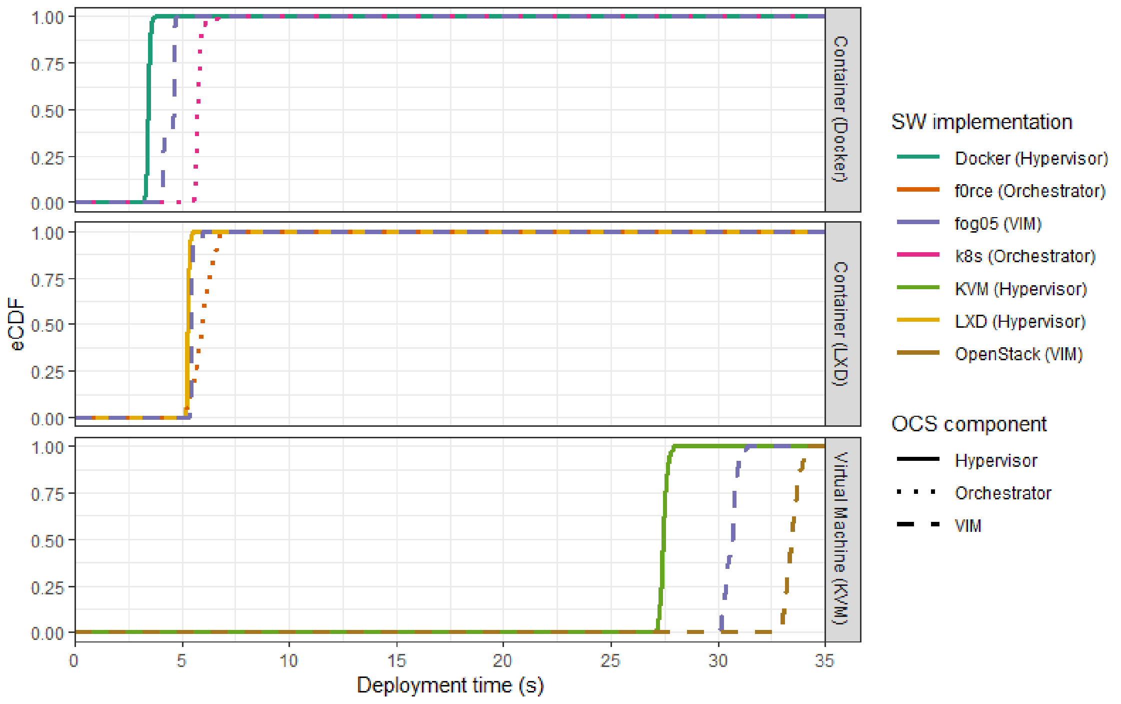

FIGURE 5-3: EXPERIMENTAL DEPLOYMENT TIME OF AN EFS STACK WITH AN ATOMIC EFS APP

The breakdown of the deployment time, along with the most significant statistical properties, is reported in Table 5-4. Starting by analysing the hypervisors, results show that Docker provide the fastest deployment time. If we compare the image sizes reported in Table 5-2, we can see that the Docker-based image weights 92.6 MB while the LXD-based image weights 4.7 MB. Nevertheless, Docker achieves a faster deployment time (3.421s vs 5.276s). It is worth reminding that in these tests the images are pre-provisioned on the compute node, therefore the images are not copied over then network. Nonetheless, Docker and LXD manage the instantiation in different ways: while Docker adopts a differential approach (i.e., it copies on disk only the differences between a baseline image and the target image/instance), LXD copies the whole image for each instance to be created. Moreover, LXD adopt a Virtual Machine-like management of the instances compared to the App-centric management adopted by Docker. This means that the LXD instance needs to execute additionally operating system-like procedures (e.g., services start-up, filesystem check, etc.) which are not instead executed in Docker. These aspect result in a longer deployment time for LXD compared to Docker. The longest deployment time is provided by KVM (i.e., 27.449s). This is natural since a Virtual Machine instance requires to go through the whole boot process in the same way as a physical machine. This means that a separate kernel runs in the Virtual Machine to implement the whole hardware-control logic, which is not required in a container-based environment.

TABLE 5-4: STATISTICAL CHARACTERISTICS OF THE EXPERIMENTAL DEPLOYMENT TIME (S)

Virt. Tech.	Component	Min	Max	Mean	Median	Std Dev.
Docker	Docker	3.223	3.756	3.421	3.413	0.086
	fog05	4.027	4.733	4.444	4.601	0.245
	k8s	5.498	6.920	5.752	5.730	0.168
LXD	LXD	5.128	5.975	5.276	5.265	0.076
	fog05	5.292	7.047	5.479	5.432	0.316
	f0rce	5.355	10.125	5.950	5.931	0.444
KVM	KVM	27.085	27.906	27.449	27.435	0.163
	fog05	30.055	31.351	30.591	30.645	0.280
	OpenStack	32.785	34.197	33.444	33.411	0.323

Regarding the VIM performance we can see how these are strictly bounded to the hypervisor performance. Specifically, we can see how the deployment time measured with fog05 for all the three virtualization technologies is ~15% higher than the hypervisor. This is due to the additional operations that the VIM needs to perform on the infrastructure to properly configure and interconnect the EFS App being instantiated. In case of using KVM as hypervisor, it can be seen that VIM provided by OpenStack is slower than the VIM provided by fog05. However, OpenStack performs additional operations regarding the authentication and authorization of the EFS App which are not implemented in the version of fog05 under test. In case of Docker, k8s adopts a *monolithic* approach by closely coupling the VIM and the Orchestrator in such a way it is hard to separate their functionalities. By having a broader look, we can observe how the deployment time measured at Orchestrator level follows the same trend observed at VIM level. In the two reported measurements (i.e., Docker with k8s and LXD with f0rce), we observe that the biggest component of the deployment time is the hypervisor following a little overhead introduced by the Orchestrator. It is worth remarking that these results do not consider the scalability of placement algorithms available in the different implementations since we are considering only one compute node.

5.1.2 Conclusions

The EFS Stack descriptor has been experimentally validated via the fog05 and f0rce implementations. The EFS Stack descriptor has hence been used by the EFS Stack Orchestrator and the EFS Resource Orchestrator provided by f0rce to instruct fog05 the deployment of a reference EFS App. Results show that the main factor contributing to the overall deployment time is the hypervisor. Container-based virtualization technologies (i.e., Docker and LXD) are ~6 times faster in deploying and executing the EFS App compared to KVM. Moreover, the image footprint of LXD is ~5 times smaller than Docker. Finally, in these tests we have considered the image to be pre-provisioned on the compute node. However, in a realistic scenario the image should be provisioned on-demand on the compute nodes. In case of limited network connectivity and storage space, LXD would be a better choice compared to Docker.

5.2 Federation

This section evaluates the static federation mechanisms described in Section 4.1. In this scenario, two administrative domains exchange SLAs and federation parameters statically previous to the federation discovery phase. This means that SLAs are assigned statically without the possibility to update them dynamically once the federation is established. The objective of this validation scenario is to test how a use case such as the SD-WAN can be adapted and consequently enhanced with mobility capabilities by leveraging the federation interface (F2). The federation mechanisms envisioned in 5G-CORAL, enable the sharing of EFS resources, allowing use case owners to deploy their applications/functions/ services across the whole cloud-to-thing continuum. The SD-WAN use case considers a Point-of-Sale (PoS) leveraging the federation mechanisms developed in WP3 in order to offload user traffic in different locations so that web applications at the edge/fog can be precisely located. As a result, the owner of the PoS does not need to own EFS resources in all locations where he desires to instantiate resources. He leverages the federation to use other domain EFS resources for offloading non-critical/sensitive applications. This concept is conceived from WAN optimization use cases, where content is precisely cached at the border of the network to avoid sending unnecessary data across the network.

The scenario to validate comprises two domains playing the following roles: one domain is a consumer domain in the federation while the second one is a provider in the federation. The main feature being validated is the ability of steering dynamically the control plane of EFS resources. Allowing different OCSs to share the control of an EFS resource.

The OCS components deployed in this scenario are the VIM (i.e., fog05) using LXD containers, the EFS Resource Orchestrator, the SD-WAN EFS Function manager and the PoS EFS Application manager. The EFS Functions, applications and services deployed in this scenario are the SD-WAN function, the PoS web application, the PoS customer & inventory database application, and the host mobility detection AP function. The experimental setup including the EFS and OCS components is depicted in Figure 5-4.

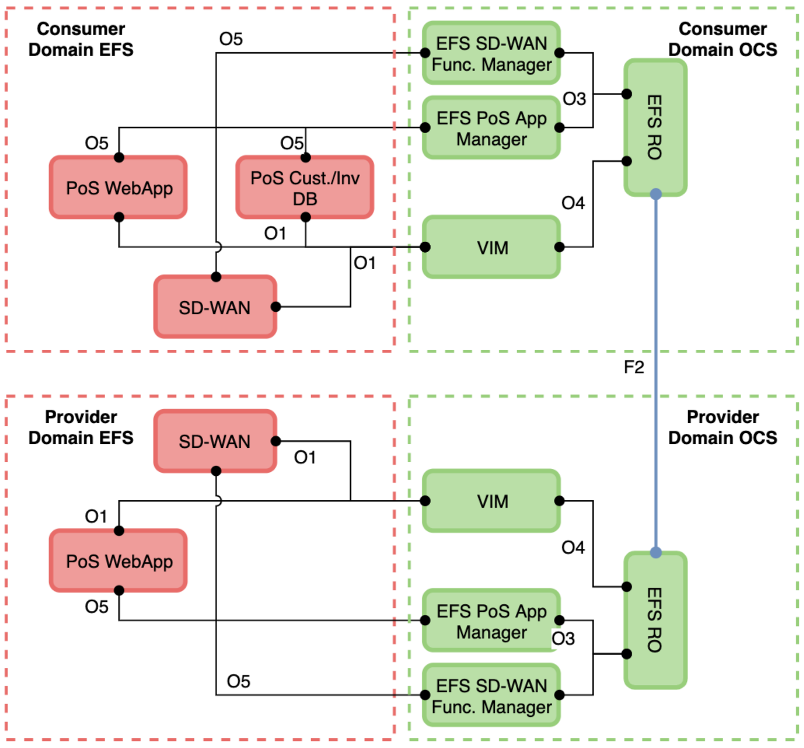

FIGURE 5-4: FEDERATION ARCHITECTURAL COMPONENTS UNDER VALIDATION

Based in the scenario described in Figure 5-4, the federation results are validated in three separate phases as illustrated in Figure 5-5 (not that a green arrow represents upstream traffic, while a red arrow represents downstream traffic):

- **Phase 0:** it represents a user directly connected to its home domain. The PoS application and the PoS customer and inventory database are located in the home domain.
- **Phase 1:** it represents a user roaming to a provider domain, which only provides him connectivity to its home or consumer domain. The PoS application and the PoS customer and inventory database are still located in the home domain.
- **Phase 2:** it represents the offloading phase, were the consumer domain instantiates in the provider domain the PoS web application, offloading traffic from the PoS terminal to its nearest PoS web application.

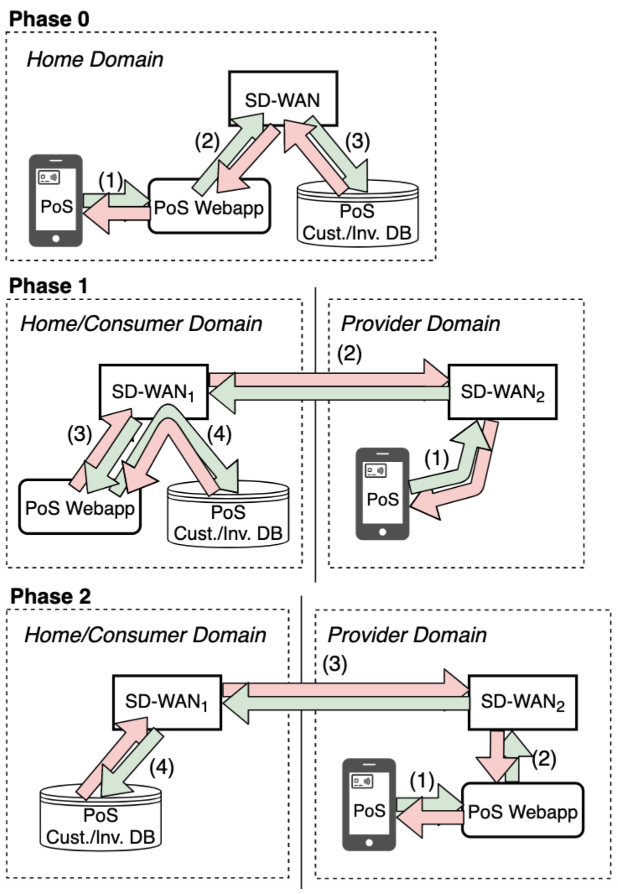

FIGURE 5-5: PHASES OF FEDERATION VALIDATION

TABLE 5-5: MAPPING OF COMMUNICATION ENDPOINTS, PHASES AND INDEXES

Communication endpoints	Phase	Index
PoS terminal and PoS Webapp	0	(1)
	1	(1), (2), (3)
	2	(1)
PoS Webapp and PoS Cust./Inv. DB	0	(2), (3)
	1	(3), (4)
	2	(2), (3), (4)

Table 5-5 maps the communication of the different applications involved in the PoS use case with the indexes in Figure 5-5, which represent each hop that traffic needs to take when communication between EFS elements involved in this use case. If the communication path is composed by a less indexes this means that the communication has less hops to transverse, which can be related to a lower latency, higher bandwidth and better jitter metrics measured in Section 5.2.1.

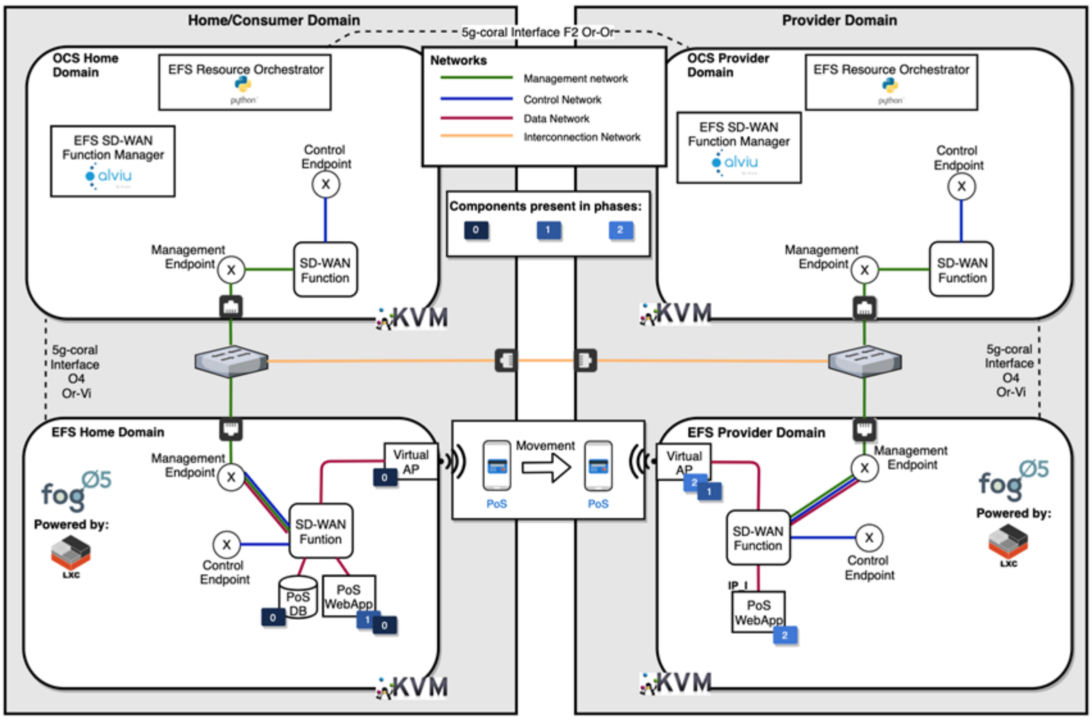

FIGURE 5-6: EXPERIMENTAL SETUP FOR FEDERATION VALIDATION

The hardware employed in the experimental scenario is composed of two Dell Latitude E5550 laptops with 8GB of RAM, Intel i5-5300U CPU and 500 GB of HDD. Each of the laptops will be used to simulate a pair of EFS OCS, composing a domain. Each EFS has a single compute node. Both laptops are interconnected using a single ethernet interface, which serves as the interconnection network for the two domains. Additionally, the laptop integrated network card serves as the domain Wi-Fi AP's, representing the EFS access network. To emulate the domain OCS and EFS, we opted for virtual machines virtualized using KVM, this approach allows us to isolate the OCS and EFS completely. Both EFS and OCS guest OS are based on Ubuntu 16.04 LTS 64bits allowing fog05 to deploy functions/applications/services using LXD container technology. Moreover, to deploy the OCS components developed for this use case, native Linux applications and LXD containers where used. Finally, as a PoS terminal we are using an extra Dell Latitude E5550 laptop, which includes google chrome browser and the necessary tools to test network performance (i.e., ping and iperf3).

5.2.1 Results

This presents the measurements gathered for validating the federation mechanism, i.e., latency, jitter, bandwidth and the deployment time in each of the federation phases. Metrics measured for phase 0, represent an ideal scenario where there is no mobility; the end user is directly connected to its home domain, where all the functions/applications/services that the use case needs are provided in a single EFS. The metrics measured from phase 1, represent a scenario where the user has moved to another domain, while the necessary core functions/applications/services are still deployed at its home domain. The provider domain is only providing a L3 secure tunnel connectivity to its home domain. Finally, the metrics measured from phase 2 use the scenario of phase 1 and on top it adds the offloading of web application traffic to a local cached web application in the provider domain.

Table 5-6 reports the results from RTT latency tests carried out for the three phases. For each test the packet size, packet rate and the number of samples taken are set statically to 1400 Bytes, 0.2 packets per second (pps) and 100 samples. When we move between phase 0 and phase 2, results show that the latency on average has increased between the PoS Webapp and the PoS Cust./Inv. DB in ~1.5ms. This result highlights the impact of the SD-WAN function in the federation. Latency between the PoS Terminal and the PoS Webapp is on average between 4.5 ms and 5 ms with a high standard deviation, justified by the radio interface used to access the applications in the EFS. Notice that from phase 1 to phase 2 the average latency has increased ~1ms, which can translate to better radio conditions while executing experiment 1.

TABLE 5-6: FEDERATION RTT LATENCY RESULTS IN MS

Phase	From	To	Tool	Max	Min	Avg.	Std. Dev.
0	PoS Terminal	PoS Webapp	ping	57.74	1.537	4.491	6.567
	PoS Webapp	PoS Cust./Inv. DB	ping	0.242	0.069	0.095	0.032
1	PoS Terminal	PoS Webapp	ping	17.77	1.815	4.56	3.783
	PoS Webapp	PoS Cust./Inv. DB	ping	0.391	0.071	0.089	0.033
2	PoS Terminal	PoS Webapp	ping	34.595	2.013	5.792	5.334
	PoS Webapp	PoS Cust./Inv. DB	ping	2.912	0.894	1.595	0.297

TABLE 5-7: FEDERATION JITTER RESULTS IN MS

Phase	From	To	Tool	1 Mbps	10 Mbps	28 Mbps
0	PoS Terminal	PoS Webapp	iperf3	-	2.327	1.602
	PoS Webapp	PoS Cust./Inv. DB	iperf3	0.074	0.011	0.005
1	PoS Terminal	PoS Webapp	iperf3	-	2.203	2.937
	PoS Webapp	PoS Cust./Inv. DB	iperf3	0.074	0.011	0.005
2	PoS Terminal	PoS Webapp	iperf3	1.83	2.42	1.123
	PoS Webapp	PoS Cust./Inv. DB	iperf3	-	0.258	0.112

Table 5-7 describes the results from jitter tests carried out for the three phases. Jitter is measured using iperf3 tool in UDP mode, using three different bandwidths (1 Mbps, 10 Mbps and 28 Mbps) which allows us to understand the jitter under different load conditions. Results show that the jitter increased by nearly a factor of two between phase 0 and 1 under the heavy load scenario (28 Mbps). The increase in jitter in phase 1 is later rectified in phase 2, resulting in similar jitter conditions as the ideal case of phase 0. These results tell us that federation and the use of offloading can under a heavy loaded network decrease the jitter of the end user.

TABLE 5-8: FEDERATION BANDWIDTH RESULTS IN MBPS

Phase	From	To	Tool	Avg. TCP	Avg. UDP
0	PoS Terminal	PoS Webapp	iperf3	17.88 Mbps	24.6 Mbps
	PoS Webapp	PoS Cust./Inv. DB	iperf3	34.78 Gbps	29.7 Mbps
1	PoS Terminal	PoS Webapp	iperf3	15.2 Mbps	21.9 Mbps
	PoS Webapp	PoS Cust./Inv. DB	iperf3	34.78 Gbps	29.7 Mbps
2	PoS Terminal	PoS Webapp	iperf3	18.8 Mbps	27 Mbps
	PoS Webapp	PoS Cust./Inv. DB	iperf3	188.6 Mbps	24.8 Mbps

Table 5-8 describes the results from bandwidth tests carried out for the three phases. Bandwidth is measured using the iperf3 tool in both modes, TCP and UDP. For the UDP mode, bandwidth is manually increased until packet loss is experienced. Results show that the bandwidth available when accessing the PoS Webapp from the PoS terminal decreased from phase 0 to phase 1, which explains the end user movement from the consumer to the provider domain. Now, between phase 1 and 2 the bandwidth available using TCP and UDP has increased, justified by the offloading of the PoS Webapp to the provider domain, which tells that the available bandwidth at phase 2 to the PoS Webapp is similar to phase 0 under ideal conditions.

Table 5-9 reports the deployment times for each of the EFS components involved. The results show that the SD-WAN function has the lightest impact in the federation deployment time, as opposed to the PoS Webapp which has the heaviest impact. Additionally, the results show that the connection of the consumer and provider federation domains take roughly 0.7 s.

TABLE 5-9: FEDERATION DEPLOYMENT TIMES (S) FOR EACH OF THE COMPONENT ON THE EFS

SD-WAN	IPsec tunnel	PoS Webapp	PoS Cust./Inv. DB
16.67 s	0.669 s	51.154 s	31.29 s

5.2.2 Conclusions

From the results we can conclude that the federation of EFS resources allows domains to extend their capabilities dynamically across multiple domains, connecting securely users between different domains or even offloading certain functions/applications/services to other domains, maximizing the utilization of resources at the edge and fog. Federation instantiation has some overhead, such as the instantiation of the SD-WAN function and the deployment of secure channels connecting both data and control planes. For use cases which require a fast federation instantiation, the SD-WAN function can be already pre-provisioned in the fog node, which is going to be federated, reducing in ~17 seconds the total time to instantiate the federation. Once the SD-WAN is instantiated the only overhead left is the control and data plane switching from the fog node which requires the EFS VIM to rewire internally the fog node and the SD-WAN to connect the control and data plane of the fog node to the consumer domain securely. The optimized federation instantiation mechanism can be deployed in less than 5 second, allowing the fog node rewiring a sufficient margin of ~4.3 seconds.

Regarding the deployment of some use cases, the federation mechanism has proved efficient and flexible enough to dynamically instantiate an application at the provider domain, which allows certain traffic to be offloaded. As stated in Section 5.2.1, results show that the offloading by leveraging federation is capable of improving the end user QoE by improving latency, jitter, and bandwidth overall, providing a seamless handover to end users who roam across different domains. Finally, we can conclude that the presented results represent a sort of lower bound given by the static federation. In case of a dynamic federation (see Section 4.1), parties could involve a negotiation phase prior to the resource federation which may result in a longer resource federation establishment.

5.3 Migration of EFS function and application

In this section, we present and discuss the approach that is adopted in 5G-CORAL platform to enable the migration of EFS functions and applications. This approach allows the OCS to migrate system and application containers between EFS nodes. Specifically, we develop a pre-copy migration scheme (as described in Section 3.2) while considering the sources of prolonged migration downtime. The enabling technologies for our proposed scheme include LXC[9], checkpoint and restore in user space (CRIU)[10], and remote file synchronization (rsync[11]). The experimental setup is shown in Figure 5-7. CRIU is utilized to dump the state of the migrating containers. The local-disk and the state of the containers are copied by using *rsync* for its remarkable speed and efficiency. To migrate a function or an application between EFS nodes with minimal downtime, the following steps are taken:

1. **Local disk-copy:** the container base-image is assumed to be available in all edge nodes to reduce traffic overhead and to keep the total migration time to minimal. Local-disk synchronization is performed to copy application related files.

[9] https://linuxcontainers.org/
[10] https://criu.org/Main_Page
[11] https://rsync.samba.org/

2. **Iterative pre-copy:** all the pages of the container including the running applications are checkpointed then copied to the destination EFS node while the container continues to run. Next, pre-copy iterations are performed to checkpoint and copy only the memory pages that have changed (dirtied) since the last checkpoint.
3. **Stop-and-copy:** the container gets frozen in this step then a final checkpoint and copy are performed. The downtime observed by the user occurs during this step.
4. **Restore-and-terminate:** the container is restored in the destination EFS node and the frozen container in the source node gets terminated.

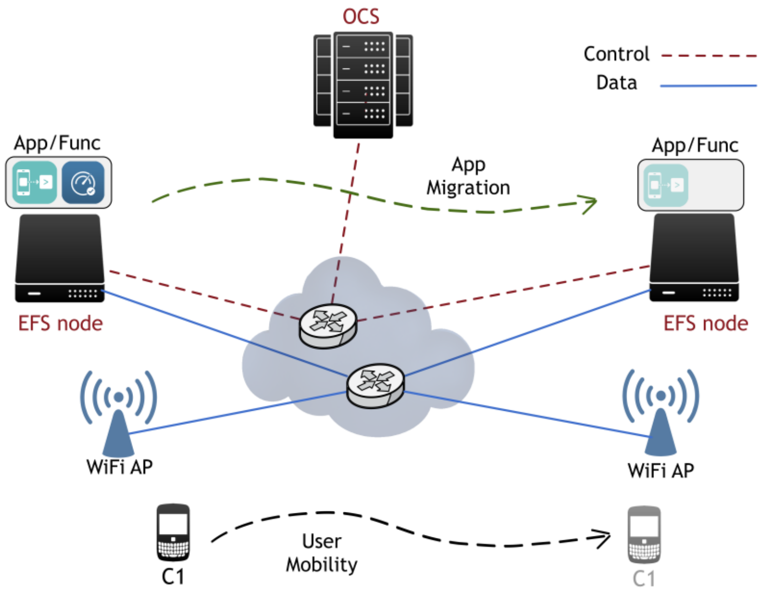

FIGURE 5-7: MIGRATION EXPERIMENTAL SET-UP

It is important to note that checkpoint and restore functions of CRIU are computational expensive. Checkpoint function collects the process tree and resources, freeze the process, then write them to files. The restore function reads the files, resolves shared resources, fork the process tree then restore the process resources. Both functions perform I/O operations which are generally slow especially on rotational block devices such as hard disk drive (HDD). To improve the migration scheme, we include the following enhancements on the EFS nodes:

- **Low-latency computing capabilities:** Linux general-purpose kernels fail to provide time guarantees for time-critical applications [32]. Hence, we incorporated low-latency computing into the EFS nodes. In addition, we scaled the CPU performance to avoid latency caused by waking up from idle state.
- **Fast storage:** HDD uses mechanical mechanism to persistently store data in blocks of 512 byte. As such, I/O operations experience seeking time delays (i.e., the time it takes the disk head to find the target track). Here, we utilize temporary file system (TMPFS) to enhance the performance as it allows short-term files to be written and read without generating disk I/O.

To benchmark container migration, we implemented stop-and-copy (*sc*) migration scheme reproducing the results presented in [33] and evaluated its downtime against our proposed pre-copy (*pc*) migration scheme. The migration experiments were carried out between two EFS nodes. In this experiment, we evaluate the downtime during the migration of blank LXC system containers and application container running both Ubuntu and Alpine Linux releases. Figure 5-7 shows the experimental setup and Table 5-10 provides the hardware and software details used in this experiment. The interconnection between the source and destination EFS nodes is 1 Gbps.

TABLE 5-10: HARDWARE AND SOFTWARE SPECIFICATIONS USED IN THE EXPERIMENTAL SET-UP

	Characteristic	Description
EFS node Hardwar	Model	ProLiant DL160
	CPU	Intel Xeon 2.10GHz
	RAM	124GiB DIMM 2400
	Network	2 x I350 Gigabit
Software & Containers	General-purpose kernel	4.4.0-145-generic
	Low-latency kernel	4.4.0-145-lowlatency
	LXC1	3.0.3
	CRIU	3.11
	Container (C1)	System container (Ubuntu 16.04)
	Container (C2)	Application container (Ubuntu 16.04)
	Container (C3)	System container (Alpine 3.7)
	Container (C4)	Application container (Alpine 3.7)

5.3.1 Results

The obtained results are based on average values of 30 trails for each presented case. Figure 5-8 shows the downtime of stop-and copy (sc) and pre-copy (pc) migration schemes. The left y-axis represents the observed downtime while the right y-axis shows the size of the accumulative checkpoint files in megabytes for the respective container type and migration scheme. Since the rate of dirty pages for the blank containers are minimal, most of the downtime is attributed to the common steps (i.e., final checkpoint → state copy → restore) of both migration schemes rather than being dominated by the time taken to copy large in-memory state.

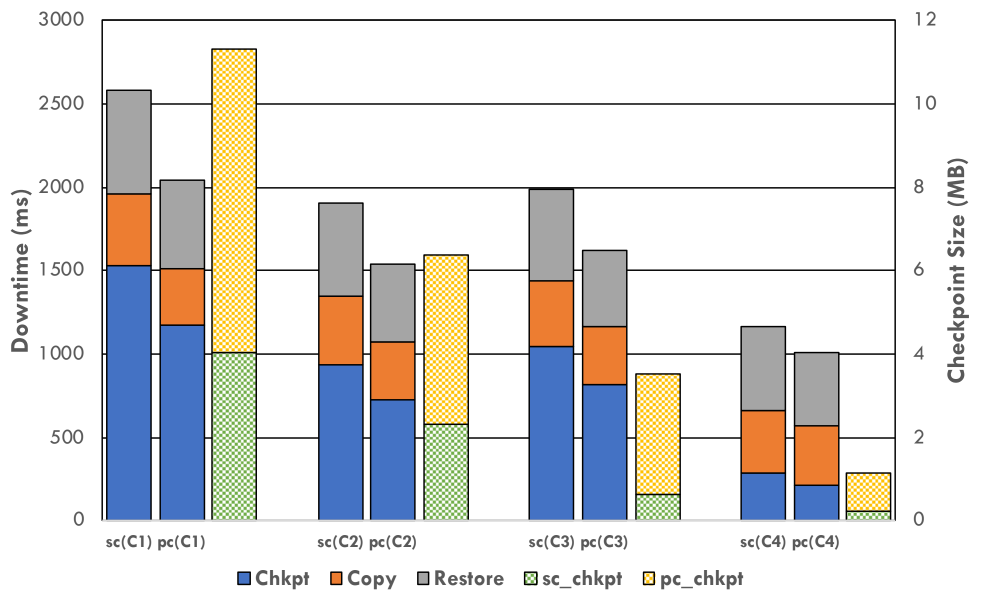

FIGURE 5-8: MIGRATION DOWNTIME COMPARISON BETWEEN STOP-AND-COPY (SC) AND PRE-COPY (PC) SCHEMES FOR DIFFERENT CONTAINERS

The container state size of every iteration in pre-copy procedure depends on the rate of dirty pages. For instance, in the case of Ubuntu system container (C1), the downtime due to the checkpointing process are 1.53 s and 1.17 s for the *sc* and *pc*, respectively. The overall average migration downtime for the two schemes are 2.58 s and 2.05 s, respectively. To highlight the features of the obtained datasets from the trails, we plot the empirical cumulative distribution function (eCDF). Figure 5-9 shows the eCDF for migrating a blank Ubuntu 16.04 application container (C2) using *sc* and *pc* migration schemes. The x-axis represents the downtime in milliseconds of each of the experiments (total of 30 experiments for each migration scheme) while the y-axis represents the cumulative percent. For example, the downtime values at the 80th percentile are 1948 ms and 1565 ms for *sc* and *pc*, respectively. Also, the range of the downtime values for *pc* is smaller compared to *sc* which indicates less variation in the case of pc scheme.

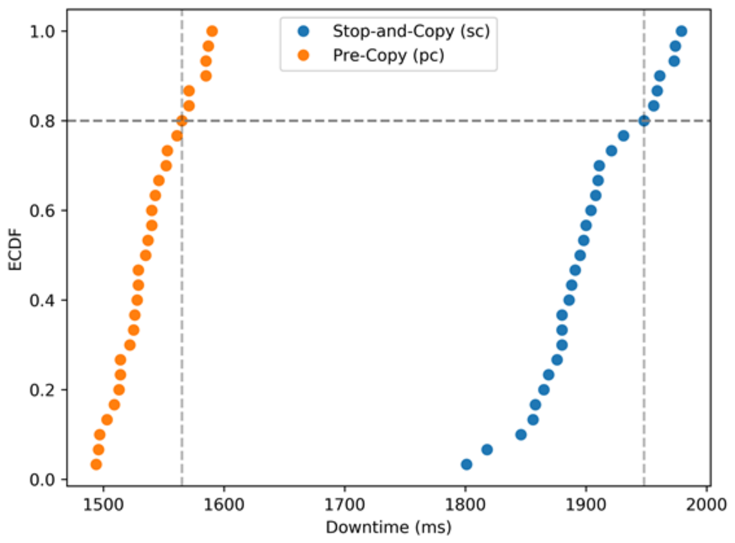

FIGURE 5-9: ECDF OF UBUNTU APPLICATION CONTAINER (C2) MIGRATION

The results clearly show that the *sc* exhibits higher downtime and variation compared to the developed *pc* scheme. As such, the proposed *pc* migration scheme reduces the downtime by approximately 21% when compared to the current state-of-the-art.

5.3.2 Conclusions

Provisioning functions and applications at the network edge through lightweight virtualization technologies proves to be a prominent feature especially for ultra-low latency and reliable vertical services. Besides the benefit of low latency and resource efficiency, supporting user mobility from computing prospective is equally important to maintain a continuous service delivery. To this end, container migration is a key solution to sustaining user quality of experience. In this section, we showed that functions and applications running in one EFS node can be relocated by the OCS to another EFS node with minimal downtime. For that, we developed a pre-copy migration scheme which includes enhancements to EFS nodes namely low-latency computing and fast storage. The experimental results show 21% downtime reduction compared to the current state-of-the-art.

5.4 Network assisted D2D

Network assisted D2D requires from the OCS to perform instantiation, termination and/or healing) of the D2D communication channel. In this section we will describe the experimental validation of the EFS Manager that is responsible for lifecycle management of the EFS functions and applications. In order to perform the experimental validation of the EFS Manager as a reference scenario we will use the fog-assisted robotics use case. Please note that in D2.2 [8], a similar scenario is evaluated and the results are showing how an EFS Service (Wi-Fi mon in Section 3.1.3.1) can be used for creating an adaptive driving algorithm to improve the driving precision robot. For what concerns this document instead, Section 3.1.3.2 gives a detailed explanation of the OCS procedures needed to establish the D2D communication in the referent use case. Based on this description, we implemented our exemplary scenario on an experimental testbed in the 5TONIC [34] laboratory. This testbed is used for a step-by-step experimentation aimed at evaluating the latency reduction of the D2D communication channel.

The set-up in the 5TONIC laboratory is shown in Figure 5-10 and it is composed of two major components: (*i*) Orchestration and Control System and (*ii*) Edge and Fog System. Regarding EFS, we used a set of EFS resources. One Fog CD for providing Wi-Fi connectivity for communication between the robots and the EFS Entities. An EFS Function implements the Wi-Fi Access Point capabilities and it is deployed as an LXD container. In the same Fog CD, we have deployed another container which serves for the EFS Service Platform and its presented as MQTT broker. The Robotic application is implemented as various Robot Operating System (ROS) [12] [35] components distributed across the robots (i.e., Fog CDs) and the edge devices. Table 5-11 lists the main ROS components used in our experimentation. Second Fog CD holds the robot intelligence and the localization monitoring service of the robots. They are deployed in a single virtual machine, while on the robot the ROS components run as native applications.

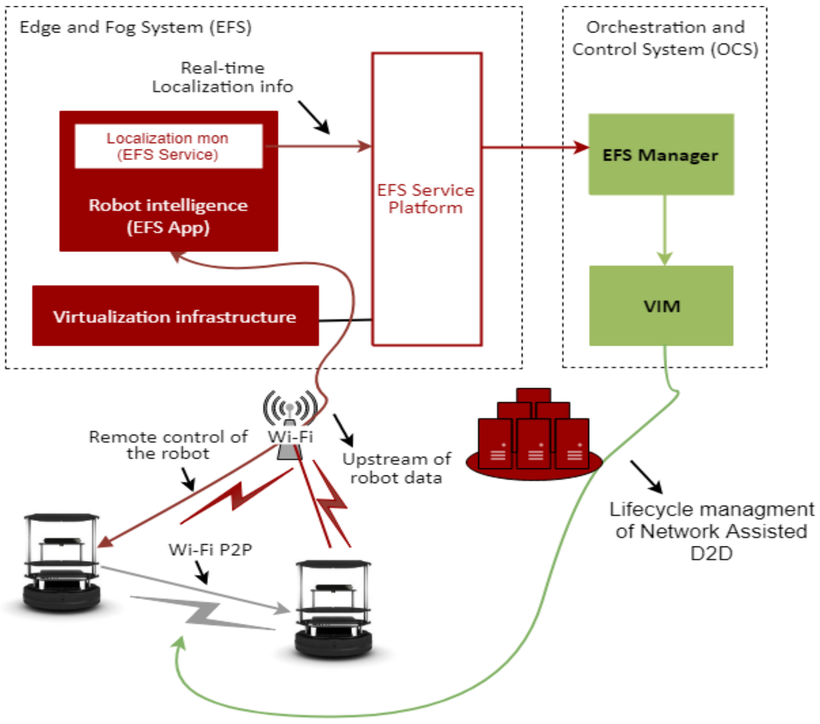
Edge and Fog System (EFS)
Real-time Localization info
Localization mon (EFS Service)
Robot intelligence (EFS App)
Virtualization infrastructure
EFS Service Platform
Orchestration and Control System (OCS)
EFS Manager
VIM
Remote control of the robot
Wi-Fi
Upstream of robot data
Wi-Fi P2P
Lifecycle managment of Network Assisted D2D

FIGURE 5-10: EXEMPLARY SCENARIO LEVERAGING NETWORK-ASSISTED D2D

TABLE 5-11: ROBOTIC SYSTEM ROS COMPONENTS

ROS component	Description
Robots localization	Probabilistic localization for robots moving in 2D [36]. Provides the indoor localization for both robots (Robot1 and Robot2) against a known map.
Experiment App	ROS application that executes the experiment drive sends commands to Robot1.
Kobuki follower	ROS application that follows a robot on a known map while trying to keep constant distance between the robots.
Map server	ROS node that provides map data as ROS service [37].

[12] The Robot Operating System (ROS) is a widely spread framework for writing robot software. It is a collection of tools, libraries, and conventions for creating complex and robust robotics applications.

LiDAR streaming	ROS package that provides support for 2D Laser Scanner [38].
Odometry sensors	ROS node that provides robot specific Odometry data [40].
Kobuki driver	ROS wrapper for the Kobuki driver [39].

The Orchestration and Control System implemented for the experimentation comprises of custom EFS Manager and VIM. The EFS Manager is implemented as a container in the first Fog CD and follows the work-flow described in Section 3.1.3.2 in the same time for the VIM component we have employed fog05 [4], which embodies all the required OCS principles.

In order to validate the benefits and performance of the network assisted D2D, we have designed and compared two experimental scenarios illustrated in Figure 5-11 and Figure 5-12. In the first experimental scenario (see Figure 5-11), all the robot ROS components are in charge of (*i*) reading sensors data (e.g., odometry, laser), (*ii*) send the data to the Robot Intelligence and (*iii*) execute driving instructions received from the Robot Intelligence. On the other hand, the received data in the Robot intelligence is used to perform indoor localization on a known map and to execute the experiment process. The ROS component – Experiment app – navigates the first robot throughout the known map and the ROS component – Kobuki follower – navigates the second robot by following the driving path of the first one while maintaining a constant distance.

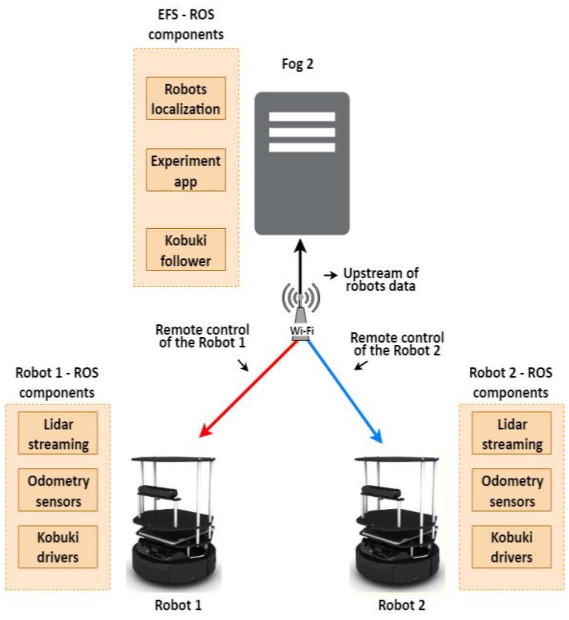

FIGURE 5-11: FULLY CENTRALIZED ROBOTICS CONTROL

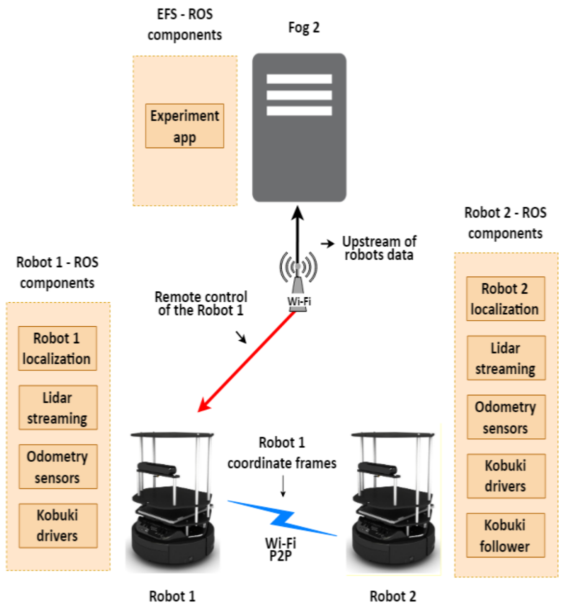

FIGURE 5-12: NETWORK-ASSISTED D2D ROBOTICS CONTROL

In the second scenario (see Figure 5-12) we have a D2D communication between the two robots that is established with the help of the EFS Manager. The distribution of the ROS components in the Robotic system is slightly different. The Robot Intelligence now hosts only the Experiment app. The robots, in addition to the existing ROS components they also host their own probabilistic localization system. This means that each robot is aware of his own position against a known map. Furthermore, the Kobuki follower ROS node is now placed in the second robot. Consequently, the coordinates of the first robot are now consumed via the D2D communication channel. This data is used by the Kobuki follower node in order to navigate the second robot.

Both scenarios consist of the robots driving in a closed and square hallway. The starting positions of the robots is in the beginning of the hallway. Both robots are placed one behind the other with approximate distance between the robots of 0.3 meters. Then, the first robot starts the experiment drive with a constant speed of 0.2 m/s. The second robot follows the first robot trying to keep a constant distance of 0.65 meters.

5.4.1 Results

The publish-consume mechanism for exchanging data between the different ROS components is based on TCP. This means that any interference on the Wi-Fi channel between the robots and the Robot intelligence will produce a retransmission at TCP level, thus introducing an undesired delay and/or packet loss in the close-loop mechanism. Additional delays and/or packet loss in the delivering of the odometry and laser sensor data can result in a significant mismatch of the robots estimated 2D position in the Robot Intelligence. Similarly, additional delays in the delivering of the movement instructions can degrade the smoothness and precision of the driving. For this reason, we carry out 10 experiment runs using the fully centralized robotic control. Each run consists of the Robot Intelligence driving the robots on a straight line for 15 meters. The starting position of the robots is approximately 7 meters away from Wi-Fi access point. Next, the robots accelerate from the starting position to the target velocity (0.2 m/s) and drives for 15 meters. At the end of the driving, the robot stops close to 22 m from the Wi-Fi access point. The Wi-Fi information obtained via the Wi-Fi mon application (see Section 3.1.3.1), was recorded in the Robot Intelligence, while on the robot itself we measured the received navigation commands delay.

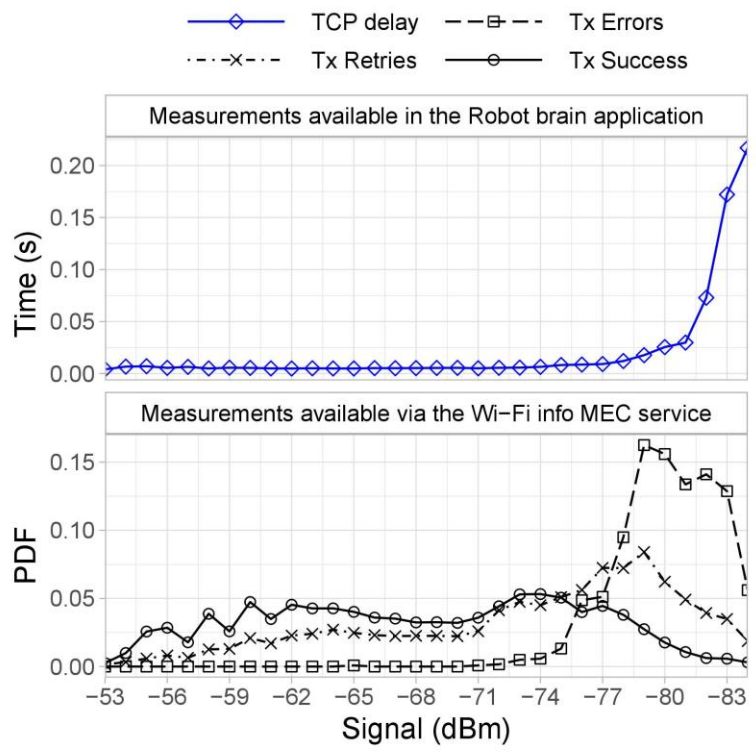

FIGURE 5-13: WI-FI CHANNEL AND DELAY CHARACTERIZATION FOR FOG-ASSISTED ROBOTICS

Figure 5-13 characterizes the quality of the Wi-Fi channel covering our experimental area. With respect to the measurements available via the Wi-Fi mon service, the Tx Retries presents the probability density function of the downlink frames retransmissions. The Tx Errors shows the PDF of the failed transmissions, Tx Success line shows the PDF of all the downlink frames successfully transmitted (from the virtual AP to the robots) and TCP delay presents the ROS driving commands downstream delay. From Figure 5-13 it can be seen that for lower Wi-Fi signal level (below -71 dBm), the probability of having a failed transmission increase. This probability becomes higher than the probability of successful transmission at signal level lower than -77 dBm.

TCP delay measurements confirm this with values as high as hundreds of milliseconds. As a result, we were able to notice certain imprecision in the distance between the two robots. This imprecision was increasing as the robots were moving away from the virtual AP. For signal level below -80dBm (the last 2 meters of the drive) it is very hard to have a successful transmission. This resulted in non-smooth and bouncy movements on both robots.

TABLE 5-12: STATISTICAL CHARACTERISTIC OF FOG-ASSISTED ROBOTICS DOWNSTREAM DELAY (S)

Scenario	Min	Max	Mean	Median	Std Dev.
Centralized	0.001	0.226	0.006	0.002	0.011
D2D	0.001	0.124	0.007	0.003	0.010

Table 5-12 presents the statistical analysis of the downstream delay measured in the robots. The results show that there is no significant difference some of the statistical parameters, such as the average. This is reasonable since we are using Wi-Fi as radio access technology for both experimental scenarios. However, environmental changes (e.g., physical obstacles, like walls and floors) other external interferences can cause peaks in the delay, slow network speed and poor signal level in the centralized scenario. Therefore, the network assisted D2D communications helps at mitigating such scenarios. Based on this observation, we decided to simulate an interfered Wi-Fi channel by introducing artificial delay. With the centralized robotic control (see Figure 5-11) we performed 3 sets of measurements, each containing 10 runs. For the first set of measurements we (*i*) didn't introduce any artificial delay (0 ms), then we introduced (*ii*) 100 ms and (*iii*) 300 ms of delay. With the network assisted D2D robotics control (see Figure 5-12) we performed 1 set of measurements containing 10 experiment runs. All the robotics system components are synchronized and share the same time reference for accurate measurements. Throughout the duration of the experiments, we recorded Euclidean distance between the robots in the Robot Intelligence and in the robots.

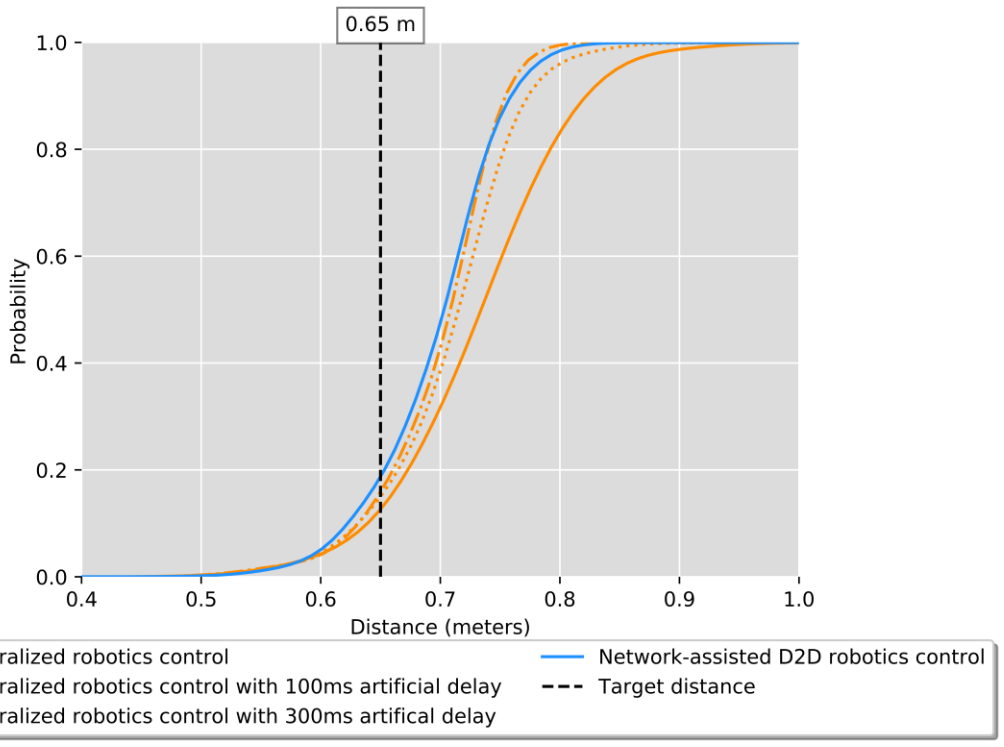

FIGURE 5-14: EXPERIMENTAL CDF OF DISTANCE BETWEEN THE TWO ROBOTS

The obtained data from all the experiments is analysed and aggregated to generate the results presented in Figure 5-14. In overall, Figure 5-14 presents the Cumulative Density Functions (CDF) for each set of measurements. From left to right, first the results regarding the network-assisted robotics control are presented. In this plot we measure the Euclidean distance in the second robot. The second, third and fourth plot reports the results for the centralized robotics control. In order to obtain these three plots, we measure the Euclidean distance in the Robot Intelligence.

TABLE 5-13: STATISTICAL CHARACTERISTICS OF THE DISTANCE (M) BETWEEN THE TWO ROBOTS

Scenario	Artificial Delay	Min	Max	Mean	Median	Std Dev.
Centralized	0 ms	0.473	4.522	0.687	0.699	0.081
	100 ms	0.470	4.517	0.695	0.703	0.065
	300 ms	0.443	4.504	0.718	0.721	0.079
D2D	0 ms	0.474	4.517	0.677	0.683	0.130

The most significant statistical properties of the measured Euclidean distance between the robots is reported in Table 5-13. It is worth mentioning that in our tests we have robots making turns. Since we are measuring the straight-line distance, this leads to shorter distances in our measurement set. In addition to that, artificial delay is added on top of the already existing Wi-Fi delay between the robots and the robot intelligence (i.e., 6-7 ms). Statistical values of the Wi-Fi delay between the robots and the robot intelligence with good network strength, little noise and no congestion are presented in Table 5-12. Starting by analysing the first experimental scenario, we can see that the centralized robotics control measurements provide the most precise maintaining of distance with respect to artificially delayed (with 100 ms and 300 ms artificial delay). This is natural since as we are increasing the delay, the mismatch of the estimated 2D positions of the robots increases. Therefore, we have slower reaction time and increased imprecision in the Robot intelligence. Regarding the network assisted D2D robotics control, we can see how significant improvement regarding the distance is achieved. In the reported measurements (i.e., D2D with Centralized robotics control, D2D with 100 ms artificially delayed Centralized robotics control, and D2D with 300 ms artificially delayed Centralized robotics control) we can observe that by using the D2D communication channel we can arrive closest to our target distance of 0.65 m during the experimental drive. It is worth highlighting that these results do not consider variable artificial delay nor packet loss.

5.4.2 Conclusions

In this section the EFS Manager has been experimentally validated by implementation of the Network assisted D2D feature. Moreover, we showed how to exploit the context information that is available in the edge by making it accessible to the EFS Manager through EFS Services. The experiments demonstrated how the EFS Services can be also beneficial for the OCS itself. The context information consumed by the OCS can be used to perform instantiation, termination and/or healing of EFS applications and function. Results show that D2D connection can be used for maintaining better coordination (e.g., moving in formation) between robots. Furthermore, in cases when there is increased delay on the Wi-Fi channel due to interferences, the low-latency robot-to-robot communication can help to optimize the robotics systems operations and achieve better precision.

6 Lessons learnt

Here, we present the lessons learnt throughout the design, refinement and experimental validation of the 5G-CORAL OCS. Specifically, we focus on the benefits of using the distributed key-value store, on the impact of volatile and resource-constrained devices, on the choice of container-based virtualization and federation, and on the ability of the OCS to operate and reconfigure multi-RAT setups.

Lesson 1: Distributed key-value store enables distributed OCS state information and facilitates OCS deployment in low-end and mobile devices

The 5G-CORAL OCS adopts a novel key-value store concept, which delivers data sharing across different technologies and networks, along the cloud-to-thing continuum. Differently from traditional key-value stores, in 5G-CORAL data are globally accessible and local replication is not required. Furthermore, we assessed the advantages of adopting a distributed VIM, capable of managing resources hosted by heterogeneous nodes, particularly by resource-constrained and mobile nodes. Differently from a centralized VIM, the distributed VIM ensures more flexibility and agility in monitoring, tracking and provisioning resources sitting on different logical layers, namely, cloud, edge, fog and terminals. Moreover, to overcome issues generated by the resource volatility, we introduced the storage decomposition in actual and desired storage, such that service instantiation/termination, polling and any other lifecycle operations can be successfully carried out even when devices are out of coverage or connectivity is disrupted. In turn, this allows the OCS to instantiate and terminate EFS applications, having an accurate visibility of the resources available and taking into account their volatility.

Lesson 2: The introduction of the EFS stack information model allows capturing key information of the edge and fog environment to perform an accurate placement

In 5G-CORAL, the EFS Stack allows to harmonize and extend the ETSI MEC and ETSI NFV information model by collecting information describing the edge and fog environment, such as I/O devices, network interfaces and location constraints. Such information model also includes the orchestration level and facilitates the application development, since the developer is no longer compelled to specify the target infrastructure immediately after the onboarding. The EFS Resource Orchestrator is ultimately the entity responsible for determining the resources fitting the requirements, based on placement algorithms capable of even capturing the volatility of fog entities.

Lesson 3: Deploying services over volatile low-cost resources comes at the cost of increased lifetime expenses

In Section 2.4.3, we discussed placement algorithms for the EFS Stack in scenarios consisting of volatile resources. Our conclusion was that introducing volatile resources in the EFS infrastructure significantly increases the total cost of ownership of the EFS Stack. In other words, the adoption of volatile resources in the 5G-CORAL platform makes a negative impact on the OPEX. Nevertheless, as demonstrated throughout the project, the pooling of such resources brings in a wealth of benefits, including a significant reduction in CAPEX due to their re-utilization and sharing. Moreover, flexible and agile service deployment/maintenance and faster federation may only rely on resources provisioned by fog nodes in the proximity (e.g., due to hardware or location requirements) for certain applications and services. Therefore, we point out that the usage of edge resources is preferable in specific circumstances, e.g., when high resource volatility makes the service lifecycle too hard to manage, thus severely impacting on OPEX KPIs set by service providers and network operators. For instance, operational costs may significantly rise in situations where services are produced by entities running on mobile resources, such as connected cars with resources on board. The continuous mobility may lead to connection drops and disruptions among the consumers of those services. As a result, we advocate that the service producer is better to be located on the edge resources from an OPEX perspective. On the contrary, in a shopping mall setting, fog resources are more persistent due to the limited space, thus ensuring good availability in a certain time frame and location.

Lesson 4: Container-based virtualization solutions ensure faster EFS deployment and migration

As pointed out in Section 5.1, the total deployment time for a single EFS App is heavily influenced by the VIM choice. Container-based virtualization approaches result in faster deployment with respect to traditional VM hypervisor solutions, such as KVM, due to the overhead generated by the separate kernel and the time spent during the boot process. In particular, we note that Docker can ensure the fastest deployment performance as long as an image is already available on the compute nodes, whereas LXD may be more suitable in scenarios where connectivity and storage resources are limited. Furthermore, containers are also employed to enable fast migration of an EFS resource between two nodes, thus ensuring minimal downtime and high QoS.

Lesson 5: Federation can help operators significantly reduce deployment and operational costs

In Section 5.2, we assessed how federation can significantly reduce the time spent to load a web application thanks to the ability of placing the web application container closer to the end user. In turn, federation can be considered a key feature of the OCS for optimizing function and application deployment and efficiently placing containers into convenient locations, such as in proximity of the users requesting the EFS service. By contrast, a single EFS infrastructure may introduce deployment limitations due to the lack of resources and reduced user accessibility, with consequent negative impact on CAPEX and OPEX. Moreover, in Section 4 we pointed out that administrative domains close to each other are more incline to take part in the federation process, assuming the adoption of a trusted cooperative peer-to-peer model. Finally, we proved that federation always increases the profits of the members involved and instability can be prevented by encouraging all the participants to share the total profit.

Lesson 6: Agile deployment and reconfiguration of RATs and multiple communication channels can be delivered by the OCS

In 5G-CORAL, RATs are handled as EFS functions and can be rapidly instantiated, terminated and migrated. As an example, robots can benefit from Wi-Fi coverage through virtual APs deployed by the OCS, or low-latency D2D connectivity can be reliably provided for robots that needs to be coordinated. This is possible thanks to the context information available at the edge, which is delivered to the EFS manager by means of EFS services. As we demonstrated in Section 5.4, the ability of deploying multi-RAT solutions is mission-critical in robotics use cases, as robots need to be able to exploit short-range D2D connectivity to navigate and maintain coordination with high accuracy, particularly in high-interference conditions. In such scenario, the 5G-CORAL OCS can quickly react and reconfigure the network connections by relying on the information stored in the EFS.

7 Conclusions and future directions

This second WP3 deliverable concludes the 5G-CORAL work on the design and validation of an Orchestration and Control System (OCS) for edge and fog environments. Summarising the technical work of WP3, the first deliverable D3.1 [6] identified the opportunities and requirements for a joint edge and fog orchestration system. These led to the early design of the OCS architecture, components and interfaces. The focus of D3.1 [6] was on the bottom part of the OCS components, namely VIM and EFS Entity Descriptor, so as to enable the support of heterogeneous and dynamic resources, dynamic migration, monitoring, and third-parties interaction on the OCS. Moreover, it proposed a solution for resource discovery and integration across multiple access technologies, such as IEEE 802.11, 3GPP, Bluetooth/ZigBee, and Ethernet. Finally, D3.1 [6] introduced the concept of resource federation and three federation models.

Departing from those findings, this second deliverable elevated the focus from the VIM up to the Orchestrator. Based on the gaps identified in the state-of-the-art (see Section 2.2) for distributed edge and fog environments, this deliverable first proposed in Section 2.3 an approach based on a distributed key-value store for implementing a distributed VIM and EFS Stack and Resource Orchestrator. By doing so, the distributed nature of edge and fog environments is also taken into consideration for the OCS and not only for the EFS. An open-source implementation of the distributed VIM and EFS-SO/ EFS-RO has been made available on GitHub under the name of fog05 [4] and f0rce [5], respectively. Next, a placement algorithm suitable for volatile environment has been presented in Section 2.4 showing the impact of pricing at edge and fog tiers on the lifetime of EFS Stacks. An analysis of the 5G-CORAL use cases has been then performed in Section 3.1 on the expected OCS procedures to tackle volatile and mobile environments, resulting in a novel container-based migration mechanism (see Section 3.2) with a reduced downtime compared to existing state-of-the-art. A resource federation mechanism has been proposed in Section 4.1 and evaluated analytically showing the impact of pricing and agreements between different domains in Section 4.2 and 4.3. An experimental validation and evaluation of the OCS has been finally reported in Section 5 highlighting the impact of the virtualization technologies and the benefits of using EFS Services on the overall management system. Section 6 summarised the lessons learnt.

For what concerns future directions, we can conclude that the 5G-CORAL OCS mainly positioned itself as an end-to-end Infrastructure-as-a-Service (IaaS) spanning across fog, edge, and cloud tiers. This means that the OCS user is able to use the EFS-SO to dereference various low-level details of underlying network infrastructure like physical computing resources, location, scaling, migration, etc. This concept is very much aligned with the infrastructure-centric and API-centric concepts of ETSI MEC and ETSI NFV. In the last couple of years, few initiatives were started building on top of ETSI MEV and ETSI NFV in order to allow to customers to develop, run, and manage applications and services without the complexity of building and maintaining the infrastructure typically associated with the delivery of the functions, applications, and services. Specifically, ETSI Experiential Networked Intelligence (ENI) [41] and the ETSI Zero touch network & Service Management (ZSM) [42] aim at closely integrating automation and intelligence mechanisms with the OSS/BSS of the customers (i.e., not only with the OSS/BSS of the infrastructure provider), thus natively supporting the customers' requirements, both operational and business-wise. By integrating such concept in the OCS, a paradigm shift is then envisioned: evolving from a distributed IaaS paradigm towards a distributed Platform-as-a-Service (PaaS) allows to provide the necessary support and data exposure to the clients, which can ultimately take advantage of the edge and fog benefits without dealing with its underlying complexity.

8 References

[1] M. McBride, D. Kutscher, E. Schooler and C.J. Bernardos, "Overview of Edge Data Discovery," IETF, draft-mcbride-edge-data-discovery-overview-01, March 2019.

[2] G. Papadopoulos, P. Thubert, F. Theoleyre and CJ. Bernardos, "SPAWN use cases," IETF, draft-bernardos-spawn-use-cases-00, March 2019.

[3] C.J. Bernardos and A. Mourad, "Autonomic setup of fog monitoring agents," IETF, draft-bernardos-anima-fog-monitoring-00, March 2019.

[4] fog05 project, "fog05: end-to-end compute, storage and networking virtualisation," Eclipse foundation. [Online]. Available: https://github.com/eclipse/fog05 [Accessed 22 May 2019].

[5] fog05 project. "f0rce: fog orchestration engine," Eclipse foundation. [Online]. Available: https://github.com/eclipse/fog05/tree/f0rce [Accessed 22 May 2019].

[6] 5G-CORAL project, "Deliverable D3.1; Initial design of 5G-CORAL orchestration and control system," May 2018. [Online]. Available: http://5g-coral.eu/wp-content/uploads/2018/06/D3.19802.pdf [Accessed 22 May 2019].

[7] 5G-CORAL project, "Deliverable D2.1; Initial design of 5G-CORAL Edge and Fog computing system," May 2018. [Online]. Available: http://5g-coral.eu/wp-content/uploads/2018/06/D2.19803.pdf [Accessed 22 May 2019].

[8] 5G-CORAL project, "Deliverable D2.2; Refined design of 5G-CORAL edge and fog computing system and future directions," May 2019.

[9] A. Corsaro, E. Boasson and O. Hecart, "zenoh: The zero network overhead protocol," July 2018. [Online]. Available: http://zenoh.io/download/pdf/zenoh.pdf [Accessed 23 May 2019].

[10] ETSI, "Network Functions Virtualisation (NFV); Management and Orchestration; Network Service Templates Specification," European Telecommunications Standards Institute, GS NFV-IFA 014, October 2016.

[11] T. Berners-Lee, R. Fielding and L. Masinter "Uniform Resource Identifier (URI): Generic Syntax," IETF, RFC 3986, January 2005.

[12] Robusto, C. (1957). The Cosine-Haversine Formula. The American Mathematical Monthly, 64(1), 38-40. DOI: https://doi.org/10.2307/2309088.

[13] Texas Instruments, "OMAP Wireless Connectivity NLCP WiFi Direct Configuration Scripts," [Online]. Available: http://processors.wiki.ti.com/index.php/OMAP_Wireless_Connectivity NLCP_WiFi_Direct_Configuration_Scripts [Accessed 22 May 2019].

[14] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum, "Optimizing the migration of virtual computers," SIGOPS Oper. Syst. Rev. 36, SI (December 2002), 377-390.

[15] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. 2005. Live migration of virtual machines. In Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI'05), Vol. 2. USENIX Association, Berkeley, CA, USA, 273-286.

[16] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. 2009. Post-copy live migration of virtual machines. SIGOPS Oper. Syst. Rev. 43, 3 (July 2009), 14-26. DOI: https://doi.org/10.1145/1618525.1618528.

[17] Violeta Medina and Juan Manuel García. 2014. A survey of migration mechanisms of virtual machines. ACM Comput. Surv. 46, 3, Article 30 (January 2014), 33 pages. DOI: https://dx.doi.org/10.1145/2492705.

[18] J. Nider and M. Rapoport. 2016. Cross-ISA Container Migration. In Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR '16). ACM, New York, NY, USA, Article 24, 1 pages. DOI: https://doi.org/10.1145/2928275.2933275.

[19] A. Machen, S. Wang, K. K. Leung, B. J. Ko and T. Salonidis, "Live Service Migration in Mobile Edge Clouds," in IEEE Wireless Communications, vol. 25, no. 1, pp. 140-147, February 2018. DOI: https://doi.org/10.1109/MWC.2017.1700011.

[20] R. Morabito, I. Farris, A. Iera and T. Taleb, "Evaluating Performance of Containerized IoT Services for Clustered Devices at the Network Edge," IEEE Internet of Things Journal, vol. 4, no. 4, pp. 1019-1030, Aug. 2017. DOI: https://doi.org/10.1109/JIOT.2017.2714638.

[21] 5G-Coral Project, "Deliverable D1.2; 5G-CORAL business perspectives," August 2018. [Online]. Available: http://5g-coral.eu/wp-content/uploads/2018/09/D1.2-final12828.pdf [Accessed 22 May 2019].

[22] Shapley, L. (1953) "A Value for n-Person Games," In: Kuhn, H. and Tucker, A., Eds., Contributions to the Theory of Games II, Princeton University Press, Princeton, 307-317.

[23] Owen, G. (1975), Multilinear extensions and the banzhaf value. Naval Research Logistics, 22: 741-750. DOI: https://doi.org/10.1002/nav.3800220409.

[24] Tuomas Sandholm et al. "Coalition structure generation with worst case guarantees". In: Artificial Intelligence (July 1999), pp. 209–238.

[25] R. Yu et al., "Cooperative Resource Management in Cloud-Enabled Vehicular Networks," in IEEE Transactions on Industrial Electronics, vol. 62, no. 12, pp. 7938-7951, Dec. 2015. DOI: https://doi.org/10.1109/TIE.2015.2481792.

[26] L. Mashayekhy and D. Grosu, "A Merge-and-Split Mechanism for Dynamic Virtual Organization Formation in Grids," in IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 3, pp. 540-549, March 2014. DOI: https://doi.org/10.1109/TPDS.2013.93.

[27] L. Mashayekhy, M. M. Nejad and D. Grosu, "Cloud Federations in the Sky: Formation Game and Mechanism," in IEEE Transactions on Cloud Computing, vol. 3, no. 1, pp. 14-27, 1 Jan.-March 2015. DOI: https://doi.org/10.1109/TCC.2014.2338323.

[28] Manlove D.F., Sng C.T.S. (2006) Popular Matchings in the Capacitated House Allocation Problem. In: Azar Y., Erlebach T. (eds) Algorithms – ESA 2006. ESA 2006. Lecture Notes in Computer Science, vol 4168. Springer, Berlin, Heidelberg.

[29] Abdulkadiroğlu, Atila, and Tayfun Sönmez. 2003. "School Choice: A Mechanism Design Approach." American Economic Review, 93 (3): 729-747.

[30] D. Gale & L. S. Shapley (1962) College Admissions and the Stability of Marriage, The American Mathematical Monthly, 69:1, 9-15, DOI: https://doi.org/10.1080/00029890.1962.11989827.

[31] Kelso, Alexander S, Jr & Crawford, Vincent P, 1982. "Job Matching, Coalition Formation, and Gross Substitutes," Econometrica, Econometric Society, vol. 50(6), pages 1483-1504, November.

[32] Scordino, C. and Lipari, G. Linux and real-time: Current approaches and future opportunities. In IEEE Internafional Congress ANIPLA. 2006.

[33] A. Machen, S. Wang, K. K. Leung, B. J. Ko and T. Salonidis, "Live Service Migration in Mobile Edge Clouds," in IEEE Wireless Communications, vol. 25, no. 1, pp. 140-147, February 2018. DOI: https://doi.org/10.1109/MWC.2017.1700011.

[34] 5TONIC, "Open-research and innovation laboratory for 5G technologies." [Online]. Available: https://www.5tonic.org/ [Accessed 22 May 2019].

[35] Robot Operating System, "Project documentation." [Online]. Available: http://wiki.ros.org/ [Accessed 22 May 2019].

[36] Robot Operating System, "AMCL Documentation." [Online]. Available: http://wiki.ros.org/amcl [Accessed 22 May 2019].

[37] Robot Operating System, "Map server Documentation." [Online]. Available: http://wiki.ros.org/map_server [Accessed 22 May 2019].

[38] Robot Operatin System, "rplidar Documentation." [Online]. Available: http://wiki.ros.org/rplidar [Accessed 22 May 2019].

[39] Robot Operating System, “Kobuki nodelet Documentation.” [Online]. Available: http://wiki.ros.org/kobuki_node [Accessed 22 May 2019].
[40] Robot Operating System, “Kobuki Odometry Documentation.” [Online]. Available: http://wiki.ros.org/navigation/Tutorials/RobotSetup/Odom [Accessed 27 May 2019].
[41] ETSI, “Experiential Networked Intelligence (ENI); Terminology for Main Concepts in ENI,” European Telecommunications Standards Institute, GR ENI 004 v1.1.1, June 2018.
[42] ETSI, “Zero touch network and Service Management (ZSM); Proof of Concept Framework,” European Telecommunications Standards Institute, GS ZSM 006 v1.1.1, May 2015.
[43] L. Cominardi, L. M. Contreras, C. J. Bernardos and I. Berberana, “Understanding QoS Applicability in 5G Transport Networks,” 2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB), Valencia, 2018, pp. 1-5. DOI: https://doi.org/10.1109/BMSB.2018.8436847.
[44] J. Martín-Pérez, L. Cominardi, C. J. Bernardos, A. de la Oliva and A. Azcorra, “Modeling Mobile Edge Computing Deployments for Low Latency Multimedia Services,” in IEEE Transactions on Broadcasting. DOI: https://doi.org/10.1109/TBC.2019.2901406.
[45] ETSI, “OpenSource MANO,” European Telecommunications Standards Institute. [Online]. Available: https://osm.etsi.org/ [Accessed 22 May 2019].
[46] Open Baton project, “Open Baton - An extensible and customizable NFV MANO-compliant framework.” [Online]. Available: https://openbaton.github.io/ [Accessed 22 May 2019].
[47] Open Baton project, “Open Baton official documentation.” [Online]. Available: https://openbaton-docs.readthedocs.io/en/stable/ [Accessed 22 May 2019].
[48] ETSI, “Network Functions Virtualisation (NFV); Management and Orchestration,” European Telecommunications Standards Institute, GS NFV-MAN 001, December 2014.
[49] ONAP project, “ONAP: Open network automation platform,” The Linux Foundation projects. [Online]. Available: https://www.onap.org/ [Accessed 22 May 2019].
[50] Cloudify Platform Ltd, “Cloudify official documentation.” [Online]. Available: https://docs.cloudify.co/4.5.0/ [Accessed 22 May 2019].
[51] OPNFV project, “Open Platform for NFV (OPNFV),” The Linux Foundation projects. [Online]. Available: https://www.opnfv.org/ [Accessed 22 May 2019].
[52] OPNFV project, “Project proposal: Orchestra,” The Linux Foundation projects. [Online]. Available: https://wiki.opnfv.org/display/PROJ/Orchestra [Accessed 22 May 2019].
[53] ARIA TOSCA project, “ARIA TOSCA Orchestration Engine,” Apache Software Foundation. [Online]. Available: http://ariatosca.incubator.apache.org/ [Accessed 23 May 2019].
[54] Kubernetes, “Production-Grade Container Orchestration,” The Linux Foundation projects. [Online]. Available: https://kubernetes.io/ [Accessed 23 May 2019].

9 Appendix: Analysis of existing orchestrators

This appendix provides an overview of existing orchestrators of reference and, for each of them, it provides a summary and a gap analysis against the OCS requirements as defined in D3.1 [6].

9.1 Open Source MANO (OSM)

ETSI OSM is an operator-led ETSI [45] community that aims at delivering a production-quality open source Management and Orchestration (MANO) stack aligned with ETSI NFV Information Models capable of meeting the requirements of production NFV networks. Figure 9-1 illustrates the OSM architecture and highlights the OSM interaction with VIM and Virtual Network Function (VNF) components. Specifically, OSM interacts with the VIM for deploying the VNFs and configuring the Virtual Links (VLs) interconnecting them. In order for OSM to work, it is required that (*i*) each VIM has an API endpoint reachable from OSM and (*ii*) each VIM provides a management network for configuring the IP addresses of the VNFs. The VIM-provided management network should be reachable from OSM.

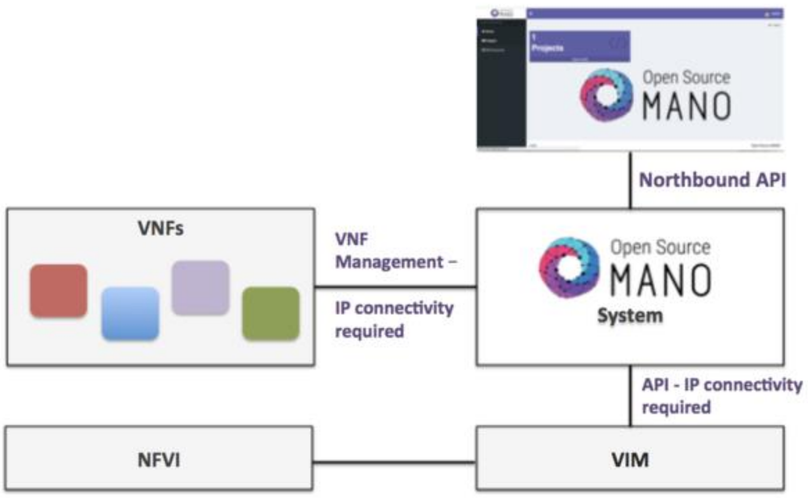

FIGURE 9-1: OSM COMPONENTS[13]

OSM runs in a single server or VM and it requires a minimum of 2 CPUs, 8 GB RAM, 20GB disk and a single interface with Internet access. However, OSM recommends 2 CPUs, 16 GB RAM, 40GB disk and a single interface with Internet access. Moreover, OSM requires Ubuntu 16.04[14] (64-bit variant required) as base image. In the following, Table 9-1 and Table 9-2 report the existing and missing OSM capabilities suitable for the 5G-CORAL OCS.

TABLE 9-1: EXISTING OSM CAPABILITIES SUITABLE FOR 5G-CORAL OCS

Capability	Description
Extensible VIM support	OSM interact with the VIMs via plugins. Additional plugins can be provided to support new VIMs.
NSD and VNFD validation	OSM provide validation of descriptors during the on-boarding.
Day0 and Day1 configurations	OSM leverages on cloud-init for day0 configuration and Juju for day1 configurations.
Monitoring	OSM provides a Prometheus server for VNF monitoring.
Complex lifecycle operation support	By using Juju is possible to have advanced lifecycle operations (e.g., reconfiguration).

[13] Source: https://osm.etsi.org/wikipub/index.php/OSM_Release_FIVE
[14] http://releases.ubuntu.com/16.04/

TABLE 9-2: MISSING OSM CAPABILITIES REQUIRED FOR 5G-CORAL OCS

Capability	Description
Federation	Federation between different OSM deployments is not possible. However, OSM can federate resources from different datacenters.
Dynamic Resources Discovery	OSM is able to manage resources provided by multiple VIMs which in turn need to support resource discovery.
Dynamic Migration Support	Migration of VNFs is not currently supported in OSM.

Finally, Table 9-3 and Table 9-4 present the gap analysis of OSM against the functional and non-functional OCS requirements.

TABLE 9-3: 5G-CORAL OCS FUNCTIONAL REQUIREMENTS AND OSM SUPPORT

Functional Requirement	Consideration
Support of harvesting computing capabilities from low-end resources	This is a VIM requirement. OSM supports different VIMs[15] via plugins.
Support of harvesting computing capabilities from mobile resources	This is a VIM requirement. See above.
Support of discovery, configuration, monitoring, allocation, etc. of relevant hardware capabilities (e.g., wireless interfaces, GPIO, GPU, SR-IOV, etc.)	Relevant hardware capabilities need to be exposed by the VIMs. Then, the allocation based on this requirement is available via Enhanced Platform Awareness (EPA).
Support of integration including at runtime of heterogeneous resources in terms of software and hardware capabilities (e.g., different CPU arch, hypervisors, etc.)	OSM is able to manage heterogenous resources as long as they are managed by different VIMs.
Support of federation including at runtime of OCS components	OSM is able to federate only resources coming from different datacentres under his control. No federation is possible between different OSM instances.
Support of the interworking with resources external to the OCS (e.g., cloud-to-thing continuum)	OSM is able to manage Physical Network Functions (PNF) which can be not under the OCS control.

TABLE 9-4: 5G-CORAL OCS NON-FUNCTIONAL REQUIREMENTS AND OSM SUPPORT

Non-Functional Requirement	Consideration
Support of deployment of OCS on low end devices (e.g., battery-limited, form-factor, resource constrained, etc.)	OSM is deployed as a set of Docker containers. It has high computing and networking requirements making impossible to deploy OSM on low end devices.
Support of deployment of OCS on mobile devices (e.g., car, robot, train, etc.)	As above, OSM needs persistent connection with VIMs and VNFs.
Availability and self-healing mechanisms in error-prone environments	If Juju is used as VNFM then self-healing is possible.
Support of large deployments in terms of number of resources and geographic areas	OSM is designed to manage resources across datacentres.
Support of plugins for extensibility	OSM supports only plugins for the VIM. Particularly, it is possible to implement connectors to VIMs.
Capability to adapt to workload changes by provisioning and de-provisioning resources in an automated manner	OSM currently does not support elasticity.

[15] https://osm.etsi.org/wikipub/index.php/VIMs

Support of multiple tenants participating and co-existing in the same environment	OSM supports multiple tenants.

9.2 Open Baton

Open Baton [46] is an Open Source Network Function Virtualization Orchestrator (NFVO) released in 2015 in partnership between Fraunhofer FOKUS Institute and the Technical University of Berlin, allowing the user to create a Network Function Virtualization (NFV) environment based on ETSI NFV MANO specifications [48]. The ultimate goal of the project is to facilitate the integration between cloud infrastructure providers and virtual network function providers in an NFV framework. To this end, Open Baton adopts the ETSI NFV data model to build network services and virtual network descriptors, enabling interoperability and supporting extensibility, thanks to its message bus architecture.

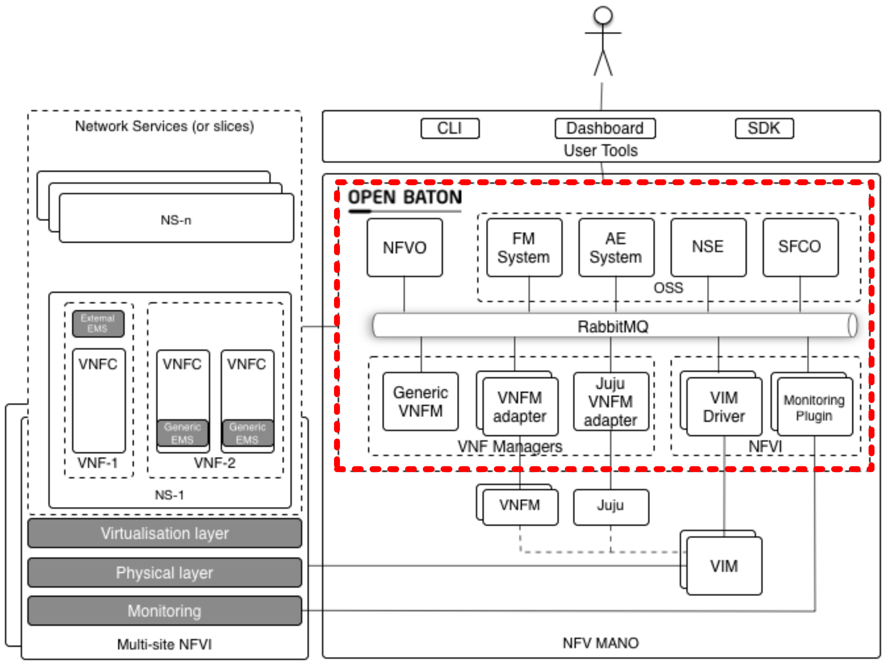

Figure 9-2: Overview of the Open Baton architecture[16]

Figure 9-2 illustrates the key components of the Open Baton architecture [47], which are summarised in the following:

- NFVO (Network Function Virtualisation Orchestrator), designed and implemented following the ETSI MANO specifications;
- Generic VNFM (Virtual Network Function Manager) and EMS (Element Management System), managing VNFs lifecycle based on the descriptors.
- The Generic VNFM provides a Juju VNFM adapter to deploy Juju charms or Open Baton VNFM packages in addition to an autoscaling engine, used for automatic runtime management of scaling operations of the VNFs;
- A monitoring plugin integrating Zabbix as monitoring system;
- An event engine based on pub/sub mechanism to dispatch lifecycle events execution;
- A set of libraries (in Java, Go and Python), used to build a bespoke VNFM;
- A driver-based mechanism for compatibility with various VIMs;

[16] Source: https://openbaton.github.io/documentation/

- A fault management system, used for automatic runtime management of faults which may occur at any level;
- A network slicing engine, used to ensure a specific QoS level for a network slice;

It is also worth pointing out that Open Baton integrates with OpenStack, representing the main VIM implementation. To sum up, Open Baton is primarily designed to extend basic orchestration towards network function management, including a generic VNFM and EMS, and interoperable with other VNFMs. Finally, three main mechanisms are available to extend the environment:

1. Via plugins based on Remote Procedure Calls (RPCs);
2. Via VNFM plugins, through Advanced Message Queuing Protocol (AMQP) messages and REST interfaces between NFVO and VNFM;
3. Via events, generated by the NFVO for each lifecycle event.

Finally, Open Baton requires a minimum of 2 CPUs, 2 GB of RAM and 10 GB of storage. However, it recommends 8 CPUs, 8 GB of RAM and 10 GB of storage. In the following, Table 9-5 and Table 9-6 report the existing and missing OSM capabilities suitable for the 5G-CORAL OCS.

TABLE 9-5: EXISTING OPEN BATON CAPABILITIES SUITABLE FOR 5G-CORAL OCS

Capability	Description
Docker VNFM and VIM driver	Open Baton can deploy containers on top of a running Docker engine. The VNFM and VIM are both needed to deploy network services (NSs) over Docker
Autoscaling engine	This module provides an NFV-compliant AutoScaling Engine (ASE). In addition, the Autoscaling engine uses the plugin mechanism to allow any convenient Monitoring System.
Fault management system	This component handles alarms generated by the VIM and executes actions through the NFVO, thus providing switch-to-standby and heal functionalities.
Monitoring plugin	The plugin mechanism allows Open Baton to conveniently use multiple monitoring systems. An example of monitoring plugin is Zabbix.
Pub/sub-based event engine	The Pub/sub mechanism can be employed to enable interoperation with multiple external VNFMs.

TABLE 9-6: MISSING OPEN BATON CAPABILITIES SUITABLE FOR 5G-CORAL OCS

Capability	Description
Service Federation	Not yet supported, yet Open Baton is expected to become a key enabler to integrate a local testbed with a large federation of 5G oriented infrastructures. As an example, 5G Berlin will federate several testbeds, each with a dedicated scope and purpose [1].
Dynamic Resources Discovery	Open Baton main goal is to extend basic orchestration and no resource discovery capabilities are provided. This may be integrated through a specific plugin.

Finally, Table 9-7 and Table 9-8 present the gap analysis of OSM against the functional and non-functional OCS requirements.

TABLE 9-7: 5G-CORAL OCS FUNCTIONAL REQUIREMENTS AND OPEN BATON SUPPORT

Functional Requirement	Consideration
Support of harvesting computing capabilities from low-end resources	Low-end resources cannot be used to provide additional computing capabilities.
Support of harvesting computing capabilities from mobile resources	Mobile resources cannot be used to provide additional computing capabilities.
Support of discovery, configuration,	Support of discovery, configuration, monitoring

monitoring, allocation, etc. of relevant hardware capabilities (e.g., wireless interfaces, GPIO, GPU, SR-IOV, etc.)	and resource allocation is currently not available. However, mechanisms to provide resource discovery are under development.
Support of integration including at runtime of heterogeneous resources in terms of software and hardware capabilities (e.g., different CPU arch, hypervisors, etc.)	This feature is available in Open Baton. The framework is fairly extensible through plugin support and dedicated SDK library.
Support of federation including at runtime of OCS components	Not available at the moment. Federation support is under development.
Support of the interworking with resources external to the OCS (e.g., cloud-to-thing continuum)	Open Baton enables orchestration of external resources, as long as the correct plugin is available.

TABLE 9-8: 5G-CORAL OCS NON-FUNCTIONAL REQUIREMENTS AND OPEN BATON SUPPORT

Non-Functional Requirement	Consideration
Support of deployment of OCS on low end devices (e.g., battery-limited, form-factor, resource constrained, etc.)	There is no support for deploying OCS on low-end devices.
Support of deployment of OCS on mobile devices (e.g., car, robot, train, etc.)	There is no support for deploying OCS on mobile devices.
Availability and self-healing mechanisms in error-prone environments	Self-healing capabilities are provided by Open Baton, through alarms originated from the VIM and convenient actions carried out by the NFVO.
Support of large deployments in terms of number of resources and geographic areas	Open Baton is highly scalable and features an AutoScaling Engine.
Support of plugins for extensibility	Plugins can be used or developed to extend Open Baton.
Capability to adapt to workload changes by provisioning and de-provisioning resources in an automated manner	The AutoScaling Engine supports workload adaptation through the monitoring system.
Support of multiple tenants participating and co-existing in the same environment	Multi-tenancy is supported via network slicing.

9.3 ONAP

ONAP [49] provides a platform for real-time and policy driven orchestration and automation of both physical and virtual network functions, enabling network and cloud provides to rapidly automate new services in a massive scale (multi-site and multi-VIM support). ONAP provides:

- A *design framework* that allows service specification with respect to all aspects, modelling the resources as well as relationship that make up the service, specify the policies that guide the service behaviour, and specify the analytics and closed-loop events needed for the elastic management of the service.
- An *orchestration and control framework* (Service Orchestrator and Controllers) that is recipe and policy driven, to provide automate instantiation of the service as well as managing the service in an elastic manner.
- An *analytic framework* that monitors the service behaviour during the lifecycle of the service and uses the policies as required to deal with situations that require healing or scaling of the service in an elastic manner.

The ONAP platform (as show in Figure 9-3) provides a unified framework for policy-driven service design, implementation, analytics and LCM for large scale workloads and services: it allows to orchestrate both physical and virtual network function enabling operators to leverage existing network infrastructure.

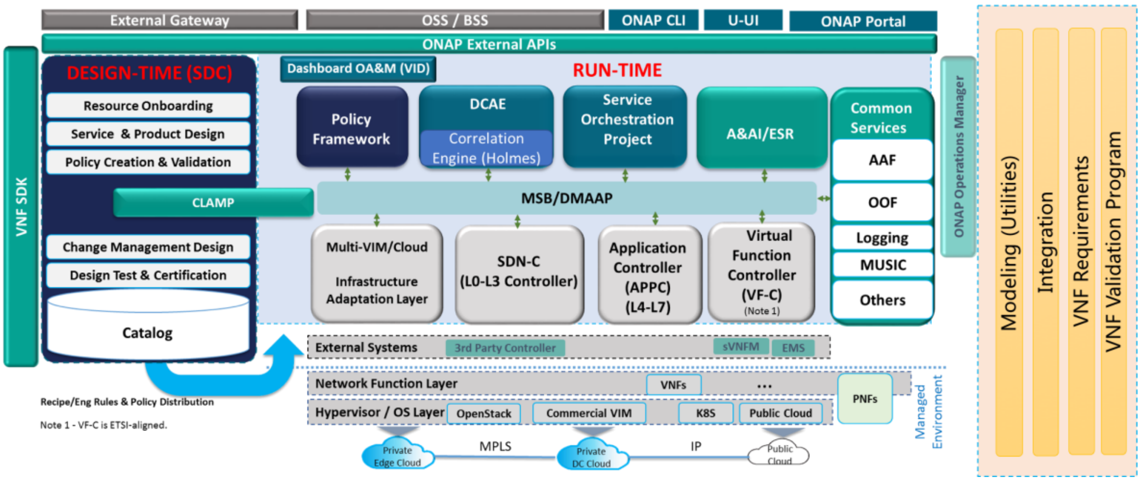

FIGURE 9-3: ONAP PLATFORM COMPONENTS[17]

Specifically, ONAP is functionally composed by a Portal, a Design Time Framework, a Runtime Framework, a Closed-Loop Automation, and Microservices Support. Particularly, the ONAP platform provides common functions that are necessary to construct specific behaviours. In ONAP a service is created and defined using the *ONAP Design Framework Portal* which is a design time component of the whole platform.

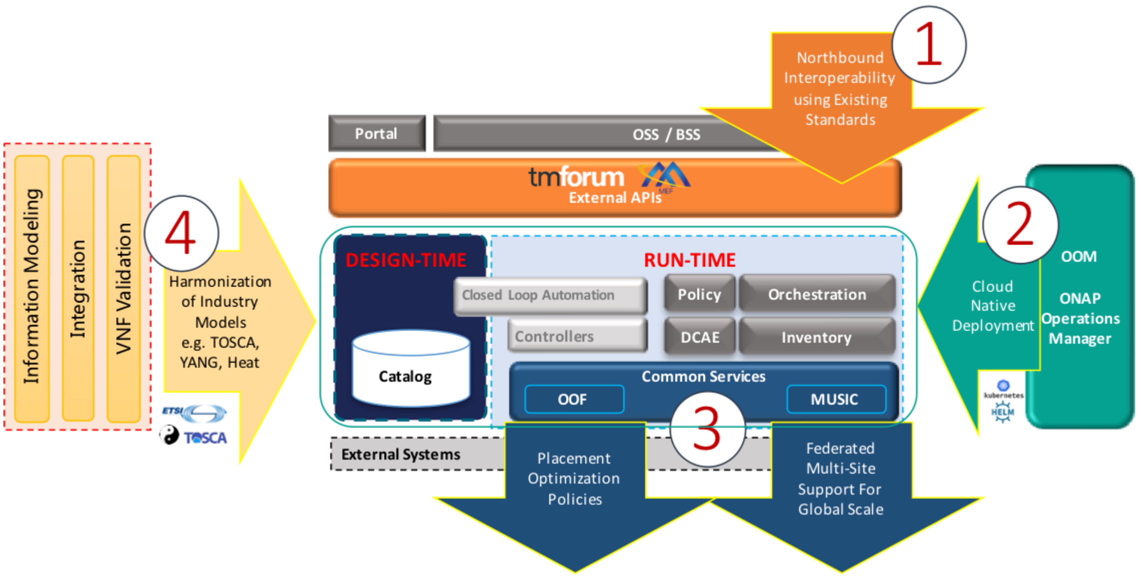

FIGURE 9-4: FUNCTIONAL VIEW OF THE ONAP ARCHITECTURE[18]

Figure 9-4 provides a simplified functional view of the architecture, in which we have:

- Design Time environment for onboarding service and resource into ONAP and designing the required services;

[17] Source: https://docs.onap.org/en/casablanca/guides/onap-developer/architecture/onap-architecture.html
[18] Source: https://docs.onap.org/en/casablanca/guides/onap-developer/architecture/onap-architecture.html

- External API both northbound and southbound that provides interoperability with multi-VIM and Cloud providers;
- OOM, that provides the ability to manage cloud-native installation and deployments to Kubernetes-managed cloud environments;
- Common Services that manage complex and optimized topologies:
  - MUSIC allows ONAP to scale to multi-site environment;
  - ONAP Optimization Framework provides a declarative, policy-driven approach for creating and running optimization applications, like Homing/Placement.
- Information Model and framework utilities to harmonize the topology, workflow and policy models coming from different standards such as ETSI NFV MANO, TM Forum SID, ONF Core, OASIS TOSCA, IETF and MEF.

The *Portal* provides access to design, analytics and operational control/administration functions, via a shared role-based dashboard.

The *Design-Time* Framework is the development environment that allows the definition and creation of resources, services and products: it is composed by (*i*) the *SDC* that provides tools to define, simulate and certify system assets as well as the associated policies, (*ii*) the *VNF SDK* with the *VVP*, which provides the tool to design and validate VNF that can be deployed in the ONAP platform, (*iii*) the *POLICY* component, that deals with the definition of policies, and (*iv*) the *CLAMP* component, used to manage closed control loops, configure it, deploy it and decommission it, as well as to update the loop with new parameters at runtime.

The *Runtime* framework executes all the rules and policies distributed by the design and creation environment. In particular, it is composed by the *SO* that automates the sequences of activities, tasks, rules and policies needed for on-demand creation, modification or removal of network, application or infrastructure services and resources. It provides a high-level orchestration with an end-to-end view of the infrastructure, network and application. The *Controllers* (*SDNC, APPC, VF*-C) are applications which are coupled with cloud and network services and execute the configuration, real-time policies and control of the state of distributed components and services. The VF-C provides an ETSI-compliant NFVO function that is responsible for the lifecycle management of virtual services and the associated physical server infrastructure. In ONAP, the modelling supports different standards:

- VNFD based on ETSI NFV IFA011 v2.4.1 with modification to align with the ONAP requirements;
- VNFD based on TOSCA that is based on ETSI NFV SOL001 v0.6.0;
- VNF Package ETSI SOL004.

ONAP is installed though the ONAP Operation Manager that uses Kubernetes, Docker containers and Helm installer. In the current version, ONAP requires Kubernetes 1.11.2, Helm 2.9, Kubectl 1.11.2, and Docker 17.03.x. This results in the deployment of 14 Virtual Machines/Containers and a minimum hardware requirement of 8 CPUs, 16 GB RAM, and 160 GB of storage. Table 9-9 and Table 9-10 report the existing and missing ONAP capabilities suitable for the OCS.

TABLE 9-9: EXISTING ONAP CAPABILITIES SUITABLE FOR 5G-CORAL OCS

Capability	Description
NSD onboarding and basic validation	NSD are verified in design time by the tool in the SDK and deployed by the runtime framework.
Extensible and highly customizable NSD format	Based on TOSCA. Open for customization.
Mature workflow execution engine with cross-	Workflow definition fully programmable at design time.

dependencies support	
Complex lifecycle operation support	Heal, scaling and recovery policies can be defined at design time
Monitoring support	Monitoring in embedded in the analytics
Arbitrary VIM support	Multi-VIM/Cloud adaptation layer is provided as well as southbound API, unclear how to implement connection to a new VIM
Dynamic migration support	Managed at runtime by the closed-loop controller
Service Federation	Designed to multi-site management, federation may be possible if we consider multi-site ONAP deployment

TABLE 9-10: MISSING ONAP CAPABILITIES SUITABLE FOR 5G-CORAL OCS

Capability	Description
Resources discovery, their utilisation tracking	Unclear how physical resources can be discovered or added at runtime. Resources can be defined at design time and the runtime have to manage them.

Finally, Table 9-11 and Table 9-12 present the gap analysis of ONAP against the functional and non-functional OCS requirements.

TABLE 9-11: 5G-CORAL OCS FUNCTIONAL REQUIREMENTS AND ONAP SUPPORT

Functional Requirement	Consideration
Support of harvesting computing capabilities from low-end resources	Low-end resources cannot be used to provide additional computing capabilities.
Support of harvesting computing capabilities from mobile resources	Mobile resources cannot be used to provide additional computing capabilities.
Support of discovery, configuration, monitoring, allocation, etc. of relevant hardware capabilities (e.g., wireless interfaces, GPIO, GPU, SR-IOV, etc.)	Support of discovery, configuration, monitoring and resource allocation is currently not available. To investigate how to integrate resource discovery.
Support of integration including at runtime of heterogeneous resources in terms of software and hardware capabilities (e.g., different CPU arch, hypervisors, etc.)	Support of heterogeneous resource is available in ONAP.
Support of federation including at runtime of OCS components	MUSIC component uses multi-site deployment as a federation mechanism from an ONAP point of view.
Support of the interworking with resources external to the OCS (e.g., cloud-to-thing continuum)	ONAP can be integrated with 3rd parties VIMs and Clouds.

TABLE 9-12: 5G-CORAL OCS NON-FUNCTIONAL REQUIREMENTS AND ONAP SUPPORT

Non-Functional Requirement	Consideration
Support of deployment of OCS on low end devices (e.g., battery-limited, form-factor, resource constrained, etc.)	The high computing requirements make impossible the deployment of ONAP on low-end devices.
Support of deployment of OCS on mobile devices (e.g., car, robot, train, etc.)	The high computing requirements make impossible the deployment of ONAP on mobile devices.
Availability and self-healing mechanisms in error-prone environments	Self-healing capabilities are provided by ONAP through alarms originated from Closed Loop monitoring (CLAMP).
Support of large deployments in terms of number of resources and geographic areas	ONAP is highly scalable and scaling of VNF is defined at design time in the POLICY component.

Support of plugins for extensibility	ONAP supports plugins for different VIMs.
Capability to adapt to workload changes by provisioning and de-provisioning resources in an automated manner	The scaling defined in POLICY can support on-demand adaptation of the service.
Support of multiple tenants participating and co-existing in the same environment	Multi-tenancy is supported.

9.4 Cloudify

Cloudify [50] is an open-source TOSCA-based cloud orchestration framework, featuring both commercial and community platform releases, widely used in production. Cloudify enables the user to model applications and services and automate their entire lifecycle, including deployment on any cloud or datacentre environment, monitoring all aspects of a deployed application, detecting issues and failure, manually or automatically remediating such issues, and performing ongoing maintenance tasks.

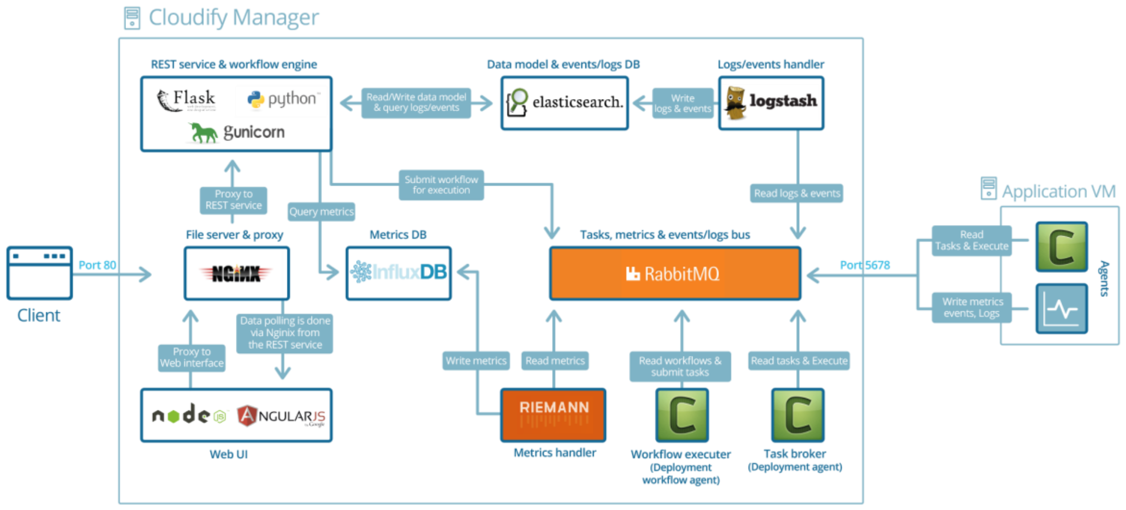

FIGURE 9-5: OVERVIEW OF THE CLOUDIFY ARCHITECTURE[19]

Figure 9-5 illustrates the Cloudify architecture which comprises of the following main components:

- **Cloudify Manager:** consisting of the Cloudify code and a set of open-source applications. The Cloudify Manager architecture is designed to support all potential operational workflows you might require when managing your applications.
- **Cloudify Agents**: representing entities for executing tasks on application hosts. They reside inside the application (e.g., VM), listen to task queues and execute tasks when required. The agents are designed to execute tasks using Cloudify-specific plugins. Note that Cloudify can run in "agentless" mode, which means that agents can use specific plugins to manage hosts without the agents being installed on those hosts. It is possible to specify which server nodes will have agents installed on them in the blueprint.
- **Cloudify Console:** includes a Cloudify Console that provides the same features as the CLI, as well as others.

Table 9-13 lists the Cloudify main features while Table 9-14 and Table 9-15 report the existing and missing Cloudify capabilities suitable for the 5G-CORAL OCS.

[19] Source: https://cloudify.co/guide/3.0/overview-architecture.html

TABLE 9-13: SUMMARY OF CLOUDIFY FEATURES

Capability	Description
Language written	Python, JavaScript (for UI).
Network Service Descriptors format	TOSCA language.
Workflow engine	TOSCA orchestration engine, based on Apache ARIA.
NSDs catalogue	Local FS with per-tenant isolation
Available integrations with external infrastructure providers	OpenStack, Kubernetes, public clouds.
Proxy for the Cloudify REST service and file server	Nginx.
Cloudify REST service	Gunicorn and Flask.
Application model, indexing, logs and events storage	PostgreSQL.
Log and event messages handler	Logstash.
Internal messaging	Async via RabbitMQ.
Build-in Monitoring platform	Riemann.
Cloudify management worker	Celery.
Monitoring sample storage	InfluxDB.
Other Southbound Interfaces	Any, via custom plugins.

TABLE 9-14: EXISTING CLOUDIFY CAPABILITIES SUITABLE FOR 5G-CORAL OCS

Capability	Description
NSD onboarding and basic validation	During NSD onboarding semantic checks are performed which ensures that TOSCA blueprint syntactically is correct. However, low-level dependencies and workflow implementation error might be checked during execution phase only.
Extensible and highly customizable NSD format	TOSCA allows to identify custom node types and associate with these nodes arbitrary metadata.
Mature workflow execution engine with cross-dependencies support	TOSCA-based workflow allows to specify several dependency types among blueprint Nodes, thus forming order of operations during the workflow execution.
Complex lifecycle operation support	Additionally, to common operations like create and delete, Cloudify considers possibility of healing and scaling operations out of the box.
Monitoring support	The Cloudify agents through the Rabbit-MQ messaging platform are reporting their monitor metrics, events and logs. In the blueprint you can configure what metrics (CPU Utilization, Physical Memory, Disk IO, Network IO etc.) you want to be reported. Cloudify monitoring implementation uses Grafana for tracking system metrics.
Arbitrary VIM support	Custom plugin can be developed for any service deployment.
Dynamic migration support	Cloudify can be configured to move VMs from one cloud to another. Also, with correct modelling of the blueprints you can migrate your container-based environment.

TABLE 9-15: MISSING CLOUDIFY CAPABILITIES REQUIRED FOR 5G-CORAL OCS

Capability	Description
Service Federation	Depending on a federation approach selected for 5G-CORAL platform and appropriate logic, custom developments and extensions will be required
Resources discovery, utilisation tracking	Custom development and extension will be required for dynamic resource discovery. Cloudify doesn't track resources availability in the managed infrastructure and just tries to complete appropriate workflow. However, constraints and SLA policy checking might be introduced on a plugin layer.

Cloudify [50] requires a minimum of 2 CPUs, 4 GB RAM, 5 GB disk and two network interfaces. However, Cloudify recommends 8 CPUs, 16 GB RAM, 64 GB of storage. Moreover, it requires Red Hat/CentOS 7.4 to run. In the following, Table 9-16 and Table 9-17 present the gap analysis of Cloudify against the functional and non-functional OCS requirements.

TABLE 9-16: 5G-CORAL OCS FUNCTIONAL REQUIREMENTS AND CLOUDIFY SUPPORT

Functional Requirement	Consideration
Support of harvesting computing capabilities from low-end resources	The Cloudify agent enables support of computing capabilities from low-end resources. Note: There is also the option of "agentless" mode.
Support of harvesting computing capabilities from mobile resources	The Cloudify agent enables support of computing capabilities from mobile resources. Note: There is also the option of "agentless" mode.
Support of discovery, configuration, monitoring, allocation, etc. of relevant hardware capabilities (e.g., wireless interfaces, GPIO, GPU, SR-IOV, etc.)	Support of discovery, configuration, monitoring and resource allocation is currently available. However, Cloudify doesn't track resources availability in the managed infrastructure and just tries to complete appropriate workflow.
Support of integration including at runtime of heterogeneous resources in terms of software and hardware capabilities (e.g., different CPU arch, hypervisors, etc.)	Cloudify has highly extensible architecture through plugin support. This enables runtime integration of heterogeneous resources in terms of software and hardware.
Support of federation including at runtime of OCS components	Not available at the moment. Custom development and extensions will be required.
Support of the interworking with resources external to the OCS (e.g., cloud-to-thing continuum)	Cloudify enables orchestration of external resources, if the correct plugin is available.

TABLE 9-17: 5G-CORAL OCS NON-FUNCTIONAL REQUIREMENTS AND ONAP SUPPORT

Non-Functional Requirement	Consideration
Support of deployment of OCS on low end devices (e.g., battery-limited, form-factor, resource constrained, etc.)	Given the high computing requirements it is impossible to deploy Cloudify on low-end devices.
Support of deployment of OCS on mobile devices (e.g., car, robot, train, etc.)	Given the high computing requirements it is impossible to deploy Cloudify on mobile devices.
Availability and self-healing mechanisms in error-prone environments	Self-healing capabilities are provided by Cloudify. The heal and scale operations are coming out of the box with Cloudify.
Support of large deployments in terms of number of resources and geographic areas	Cloudify is highly scalable when TOSCA Auto-Scaling is enabled.
Support of plugins for extensibility	Cloudify natively has a plugin-based architecture.
Capability to adapt to workload changes by provisioning and de-provisioning resources in an automated manner	Since Cloudify does not track resources availability in the managed infrastructure and just tries to complete appropriate workflow this feature is not available currently in Cloudify.
Support of multiple tenants participating and co-existing in the same environment	Multi-tenancy is supported in the current version of Cloudify.

9.5 OPNFV

OPNFV [51] is a collaborative project supported by the Linux Foundation that brings together service providers, cloud computing and infrastructure providers, as well as developers and users who define new platforms, integrate existing open source frameworks and components, and test, develop and deploy NFV open source projects. The common goal of the service provider promoting this project is to push the evolution of NFV by building a carrier-grade platform that focuses on ensuring interoperability, consistency and high performance across multiple open source components. In this regard, OPNFV continues to integrate with multiple projects and test to drive technology development. OPNFV not only aims at developing and establishing standards, but also works closely with various standards organizations such as ETSI's NFV Internet Standards Organization, IEEE, ONF, etc. to implement the standard NFV reference platform. By integrating components from upstream projects, the community is able to conduct performance and use case-based testing on a variety of solutions to ensure the platform's suitability for NFV use cases.

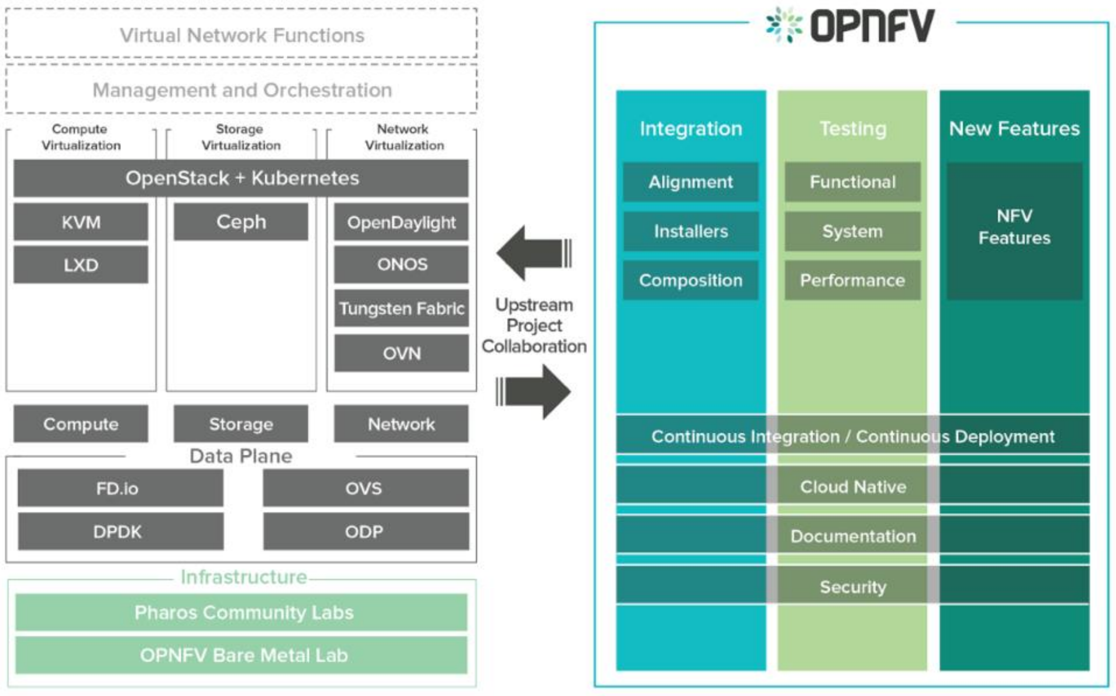

FIGURE 9-6: OPNFV ARCHITECTURE[20]

OPNFV also works upstream with other open source communities to bring contributions and learnings from its work directly to those communities in the form of blueprints, patches, bugs, and new code. Particularly, OPNFV focuses on building NFV Infrastructure (NFVI) and Virtualized Infrastructure Management (VIM) by integrating components from upstream projects such as OpenDaylight, OVN, OpenStack, Kubernetes, Ceph Storage, KVM, Open vSwitch, Linux, DPDK, FD.io and ODP. OPNFV is able to run on both Intel and ARM commercial and white-box hardware, support VM, container and bare metal workloads. These capabilities, along with application programmable interfaces (APIs) to other NFV elements, form the basic infrastructure required for Virtualized Network Functions (VNF) and MANO components.

OPNFV platform architecture (show in Figure 9-6) can be decomposed into the following basic building blocks: hardware, software platform, tooling and testing, applications, and MANO. When OPNFV projects seek orchestration functionalities for their testing scenarios, they usually

[20] Source: https://www.opnfv.org/software

emulate or simulate those functionalities executing different procedures and/or requests in parallel to different components. The Orchestra project [52] aims to integrate Open Baton with existing OPNFV projects for specific scenarios and use cases. The OPNFV-SFC (Service Function Chaining) test cases use Tacker as MANO component. Tacker is an official OpenStack project building a Generic VNF Manager (VNFM) and an NFV Orchestrator (NFVO) to deploy and operate Network Services and VNFs on an NFVI platform.

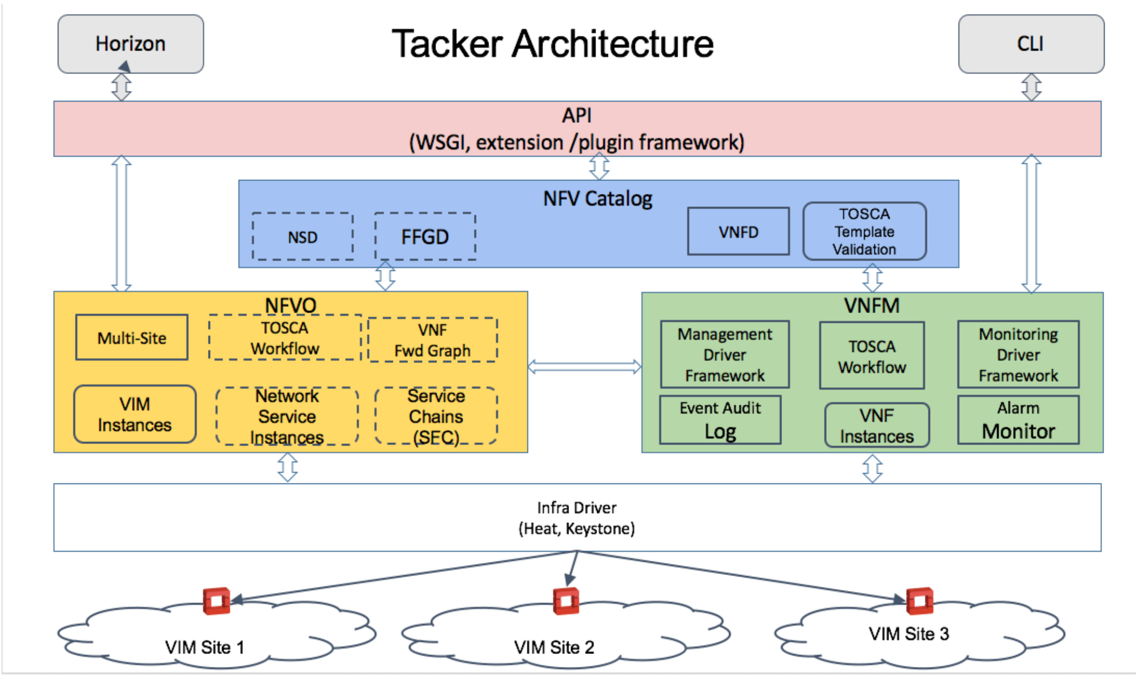

FIGURE 9-7: TACKER ARCHITECTURE[21]

The Tacker architecture is shown in Figure 9-7 and consists of three major components:

1. **NFV Catalog:** it includes VNF Descriptors, Network Services Descriptors, and VNF Forwarding Graph Descriptors;
2. **VNFM**: it performs basic life-cycle of VNF (create/update/delete), enhanced platform-aware (EPA) placement of high-performance NFV workloads, health monitoring of deployed VNFs, auto-healing/auto-scaling VNFs based on policies, and easy initial configuration of VNFs;
3. **NFVO:** it performs templatized end-to-end Network Service deployment using decomposed VNFs, VNF placement policy ensuring efficient placement of VNFs, VNFs connected using an SFC (described in a VNF Forwarding Graph Descriptor), VIM resource checks and resource allocation, ability to orchestrate VNFs across multiple VIMs and multiple sites (POPs).

Tacker uses TOSCA for VNF meta-data definition. More specifically, Tacker uses TOSCA NFV profile schema. For Tacker to work, the system consists of two parts: the tacker system and the VIM systems. In the following, Table 9-18 and Table 9-19 report the existing and missing Cloudify capabilities suitable for the 5G-CORAL OCS.

TABLE 9-18: EXISTING OPNFV CAPABILITIES SUITABLE FOR 5G-CORAL OCS

Capability	Description
Implementing the standard NFV reference platform	It can match with various ETSI's NFV Internet Standards.

[21] Source: https://wiki.openstack.org/wiki/Tacker

Deploying NFVI and VIM	It is possible to establish through integrating components from upstream projects such as OpenDaylight, OVN, OpenStack, Kubernetes, Ceph Storage, KVM, Open vSwitch, Linux, DPDK, FD.io and ODP.
VIM installation	Since the VIM is either OpenStack or Kubernetes, the target VIM installation involves the setup of either system.
Consistency with hardware production	OPNFV is can be run on both Intel and ARM production.
VM and container deployment	White-box hardware can be used to support VM and container.

TABLE 9-19: MISSING OPNFV CAPABILITIES REQUIRED FOR 5G-CORAL OCS

Capability	Description
Federation	It is not supported.

OPNFV requires a minimum of 2 CPUs, 16 GB RAM, 256 GB of storage. However, OPNFV recommends 8 CPUs, 64 GB RAM, 512 GB of storage. Moreover, it requires Red Hat/CentOS/ Ubuntu to run. Table 9-20 and Table 9-21 present the gap analysis of OPNFV against the functional and non-functional OCS requirements.

TABLE 9-20: 5G-CORAL OCS FUNCTIONAL REQUIREMENTS AND OPNFV SUPPORT

Functional Requirement	Consideration
Support of harvesting computing capabilities from low-end resources	Resource support depends on the VIM.
Support of harvesting computing capabilities from mobile resources	Resource support depends on the VIM.
Support of discovery, configuration, monitoring, allocation, etc. of relevant hardware capabilities (e.g., wireless interfaces, GPIO, GPU, SR-IOV, etc.)	Resource support depends on the VIM.
Support of integration including at runtime of heterogeneous resources in terms of software and hardware capabilities (e.g., different CPU arch, hypervisors, etc.)	Resource support depends on the VIM.
Support of federation including at runtime of OCS components	Not supported.
Support of the interworking with resources external to the OCS (e.g., cloud-to-thing continuum)	Not supported.

TABLE 9-21: 5G-CORAL OCS NON-FUNCTIONAL REQUIREMENTS AND OPNFV SUPPORT

Non-Functional Requirement	Consideration
Support of deployment of OCS on low end devices (e.g., battery-limited, form-factor, resource constrained, etc.)	Given the high computing requirements it is impossible to deploy OPNFV on low-end devices.
Support of deployment of OCS on mobile devices (e.g., car, robot, train, etc.)	Given the high computing requirements it is impossible to deploy OPNFV on mobile devices.
Availability and self-healing mechanisms in error-prone environments	OPNFV may support self-healing and auto-scaling via Tacker, especially auto-restart on failures.
Support of large deployments in terms of number of resources and geographic areas	Clustering multiple OPNFV and OpenStack instances may lead to large deployments.

Support of plugins for extensibility	OPNFV can be extended via plugins.
Capability to adapt to workload changes by provisioning and de-provisioning resources in an automated manner	OpenStack provides basic support for load balancing.
Support of multiple tenants participating and co-existing in the same environment	Tacker is a multi-tenant aware VNF Manager.

9.6 Apache ARIA TOSCA

ARIA, which stands for Agile Reference Implementation of Automation, is an open source, TOSCA-based orchestration library, which supports multi-cloud and multi-VIM environments, that can be used by any organization wanting to integrate TOSCA orchestration capabilities into their current and future solutions [53]. Its goal is to accelerate adoption of the TOSCA standard for orchestration with an open governance model by bringing together a large community of contributors to develop solutions more quickly. ARIA TOSCA is an open, light, CLI-driven library of orchestration tools that other open projects can consume to easily build TOSCA-based orchestration solutions. ARIA is now an incubation project at the Apache Software Foundation.

OASIS TOSCA offers a vendor neutral standard for modeling cloud-based applications, while ARIA is an open implementation of the TOSCA specification, allowing complete visibility and free use of all its source code[22]. ARIA offers a library with a programmable interface that allows embedding ARIA into collaborative projects, to enable organizations looking to incorporate TOSCA orchestration capabilities into their solutions. Figure 9-8 illustrates the ARIA architecture.

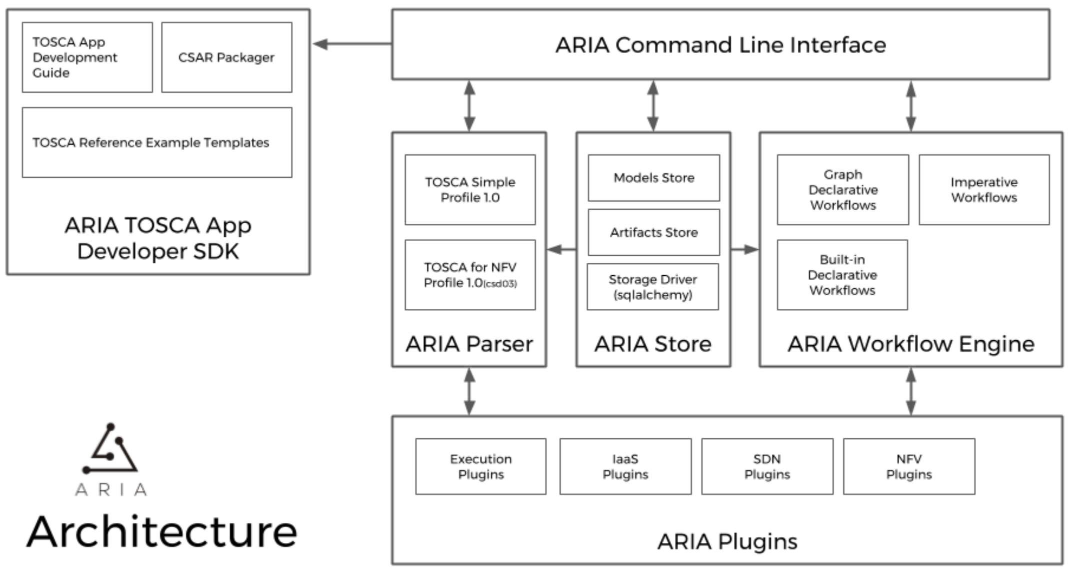
TOSCA App Development Guide
CSAR Packager
TOSCA Reference Example Templates
ARIA TOSCA App Developer SDK
ARIA Command Line Interface
TOSCA Simple Profile 1.0
TOSCA for NFV Profile 1.0(csd03)
ARIA Parser
Models Store
Artifacts Store
Storage Driver (sqlalchemy)
ARIA Store
Graph Declarative Workflows
Imperative Workflows
Built-in Declarative Workflows
ARIA Workflow Engine
Execution Plugins
IaaS Plugins
SDN Plugins
NFV Plugins
ARIA Plugins
ARIA
Architecture

FIGURE 9-8: OVERVIEW OF THE ARIA ARCHITECTURE[23]

Through ARIA, application vendors will be able to test and run their applications easily, from blueprint to deployment, without the former hassle of developing the orchestration engine themselves, simplifying TOSCA certification and validation exponentially. ARIA includes a TOSCA DSL parser, whose role is to interpret the TOSCA template, create an in-memory graph of the application and validate template correctness. TOSCA provides a typing system with normative node types to describe the possible building blocks for constructing a service template, as well as

[22] https://github.com/apache/incubator-ariatosca
[23] Source: http://ariatosca.incubator.apache.org/about/

relationship types to describe possible kinds of relations. Both node and relationship types may define life-cycle operations to implement the behavior an orchestration engine can invoke when instantiating a service template. The template files are written in declarative YAML language using TOSCA normative types. Technology specific types can be introduced via ARIA Plugins without any modifications of the parser code. ARIA natively supports TOSCA Simple Profile 1.0, and TOSCA Simple Profile for Network Function Virtualization. TOSCA Templates include a YAML Topology Template, plugins, workflows, and resources such as scripts and others.

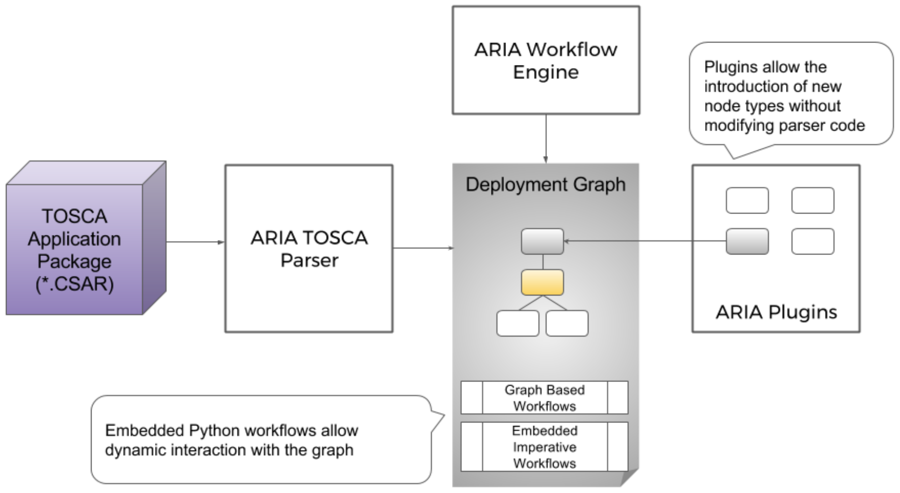
ARIA Workflow Engine
Plugins allow the introduction of new node types without modifying parser code
TOSCA Application Package (*.CSAR)
ARIA TOSCA Parser
Deployment Graph
ARIA Plugins
Graph Based Workflows
Embedded Imperative Workflows
Embedded Python workflows allow dynamic interaction with the graph

FIGURE 9-9: DECLARATIVE MODEL-DRIVEN ORCHESTRATION[24]

ARIA Workflows (see Figure 9-9) are automated process algorithms that allow dynamic interaction with the graph described by the application topology template. ARIA Workflows describe the flow of the automation by determining when which tasks will be executed. A task may be an operation, optionally implemented by a plugin, or other actions, including arbitrary code or scripts. ARIA Workflows can be embedded within the TOSCA Template to be able to access the graph dynamically. Workflows are implemented as Python code using dedicated APIs and a framework to access the graph and the runtime context of the application, the context provides access to the object graph described in the TOSCA template. ARIA comes with a number of built-in workflows - these are the workflows for install, uninstall, scale and heal. Built-in workflows are not special in any way: ARIA supports creating custom workflows that use the same APIs built-in workflows are using.

ARIA Plugins allow extending the TOSCA normative types dynamically by adding new technology-specific node types and relationship types, without changing the code of the ARIA TOSCA Parser. The plugins introduce new node types and the implementation that realizes the logic behind every new node type. The plugin-based types are isolated, allowing to use different versions of the same plugin in a single blueprint - for example support OpenStack Kilo and OpenStack Juno in the same template. It also allows combining types of different technologies - for example OpenStack nodes with VMware, Amazon, or other types such as Router, Firewall, Kubernetes and others. The work of interacting with IaaS APIs, running scripts, Configuration Management tools, Monitoring tools and any other tools used when managing applications is done by the ARIA Plugins. Plugins can be included as part of the application template package and loaded dynamically. ARIA includes set of plugins that can be used as it is or as reference for implementing for new plugins. In the following, Table 9-22 and Table 9-23 report the existing and missing ARIA capabilities suitable for the 5G-CORAL OCS.

[24] Source: http://ariatosca.incubator.apache.org/about/

TABLE 9-22: EXISTING APACHE ARIA CAPABILITIES SUITABLE FOR 5G-CORAL OCS

Capability	Description
NSD onboarding and basic validation	During NSD onboarding ARIA performs semantic checks which ensures that TOSCA blueprint syntactically is correct. However, low-level dependencies and workflow implementation error might be checked during execution phase only.
Extensible and highly customizable NSD format	ARIA provides TOSCA based NSD format which identify custom node types and associate with these nodes arbitrary metadata.
Mature workflow execution engine with cross-dependencies support	ARIA itself natively supports TOSCA-based workflow which allows to specify several dependency types among blueprint Nodes, thus forming order of operations during the workflow execution.

TABLE 9-23: MISSING APACHE ARIA CAPABILITIES REQUIRED FOR 5G-CORAL OCS

Capability	Description
Complex lifecycle operation support	ARIA itself does not support complex lifecycle operation; it depends on whether or not an OCS implements and supports it.
Monitoring support	ARIA itself does not support complex lifecycle operation; it depends on whether or not an OCS implements and supports it.
Arbitrary VIM support	ARIA itself does not support complex lifecycle operation; it depends on whether or not an OCS implements and supports it.
Dynamic migration support	ARIA itself does not support complex lifecycle operation; it depends on whether or not an OCS implements and supports it.
Service Federation	ARIA itself does not support complex lifecycle operation; it depends on whether or not an OCS implements and supports it.
Resources discovery, their utilisation tracking	ARIA itself does not support complex lifecycle operation; it depends on whether or not an OCS implements and supports it.

Apache ARIA does not have any recommended requirements per se. However, it requires Python 2.6/2.7 (Python 3 is currently not supported) and it has been tested under Ubuntu 14.04, Ubuntu 16.04, Centos 6.6, Centos 7, Arch Linux, and Windows 10. Finally, Table 9-24 and Table 9-25 present the gap analysis of ONAP against the functional and non-functional OCS requirements.

TABLE 9-24: 5G-CORAL OCS FUNCTIONAL REQUIREMENTS AND APACHE ARIA SUPPORT

Functional Requirement	Consideration
Support of harvesting computing capabilities from low-end resources	It relies on the tosca specification to specify resource node for low-end resources and relies on orchestrator being developed to implement the harvesting function for such resources. ARIA, playing as an orchestration engine, provides a way to extend the specification for low-end resources and provides an API set between operators and orchestrators to be developed to enable such support.
Support of harvesting computing capabilities from mobile resources	It relies on the tosca specification to specify resource node for *mobile resources* and relies on orchestrator being developed to implement the harvesting function for such resources. ARIA, playing as an orchestration engine, provides a way to extend the specification for mobile resources and provides an API set between operators and orchestrators to be developed to enable such support.
Support of discovery, configuration, monitoring, allocation, etc. of relevant hardware capabilities (e.g., wireless interfaces, GPIO, GPU, SR-IOV, etc.)	To discover, configure, monitor, allocate resource with various hardware capabilities, it relies on the tosca specification to specify resource node of such and relies on orchestrator being developed to implement the functions for such resources. ARIA, playing as an orchestration engine, provides a way to extend the specification for resources

	with various hardware capabilities and provides an API set between operators and orchestrators to be developed to enable such support.
Support of integration including at runtime of heterogeneous resources in terms of software and hardware capabilities (e.g., different CPU arch, hypervisors, etc.)	It relies on the tosca specification to specify such heterogeneous resource node which describes both software and hardware capabilities and relies on orchestrator being developed to implement runtime functions for such resources. ARIA, playing as an orchestration engine, provides a way to extend the specification for such heterogeneous resources and provides an API set between operators and orchestrators to be developed to enable such support.
Support of federation including at runtime of OCS components	It relies on the orchestrator being developed to support federation function.
Support of the interworking with resources external to the OCS (e.g., cloud-to-thing continuum)	It relies on the orchestrator being developed to support such interworking function.

TABLE 9-25: 5G-CORAL OCS NON-FUNCTIONAL REQUIREMENTS AND APACHE ARIA SUPPORT

Non-Functional Requirement	Consideration
Support of deployment of OCS on low end devices (e.g., battery-limited, form-factor, resource constrained, etc.)	It relies on the tosca specification to specify resource node for low end devices and relies on orchestrator being developed to enable such support.
Support of deployment of OCS on mobile devices (e.g., car, robot, train, etc.)	It relies on the tosca specification to specify resource node for mobile devices and relies on orchestrator being developed to enable such support.
Availability and self-healing mechanisms in error-prone environments	It relies on both the tosca specification and implementation of orchestrator to enable an OCS operating in error-prone environments and to provide support of self-healing mechanisms.
Support of large deployments in terms of number of resources and geographic areas	It relies on both the tosca specification and implementation of orchestrator to support such large deployments
Support of plugins for extensibility	ARIA, playing as an orchestration engine, allows tosca-based plugins to extend the specification for newly designed resources. It still relies implementation of orchestrator to support plugins for extensibility
Capability to adapt to workload changes by provisioning and de-provisioning resources in an automated manner	It relies on both the tosca specification and implementation of orchestrator to enable an OCS operating in error-prone environments and to provide support of self-healing mechanisms.
Support of multiple tenants participating and co-existing in the same environment	It relies on the tosca specification and the orchestrator being developed to support multiple tenants and co-existence.

9.7 Kubernetes (K8s)

Kubernetes is a portable, extensible open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation [54]. Kubernetes has a number of features. It can be thought of as:

- a container platform;

- a microservices platform;
- a portable cloud platform.

Kubernetes provides a container-centric management environment. It orchestrates computing, networking, and storage infrastructure on behalf of user workloads. This provides much of the simplicity of Platform as a Service (PaaS) with the flexibility of Infrastructure as a Service (IaaS) and enables portability across infrastructure providers. With Kubernetes application-specific workflows can be streamlined to accelerate developer velocity. However, Kubernetes is not a traditional, all-inclusive PaaS (Platform as a Service) system. Since Kubernetes operates at the container level rather than at the hardware level, it provides some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, logging, and monitoring. However, Kubernetes is not monolithic, and these default solutions are optional and pluggable. Kubernetes provides the building blocks for building developer platforms but preserves user choice and flexibility where it is important. Additionally, Kubernetes is not a mere orchestration system. The technical definition of orchestration is execution of a defined workflow: first do A, then B, then C. In contrast, Kubernetes is comprised of a set of independent, composable control processes that continuously drive the current state towards the provided desired state.

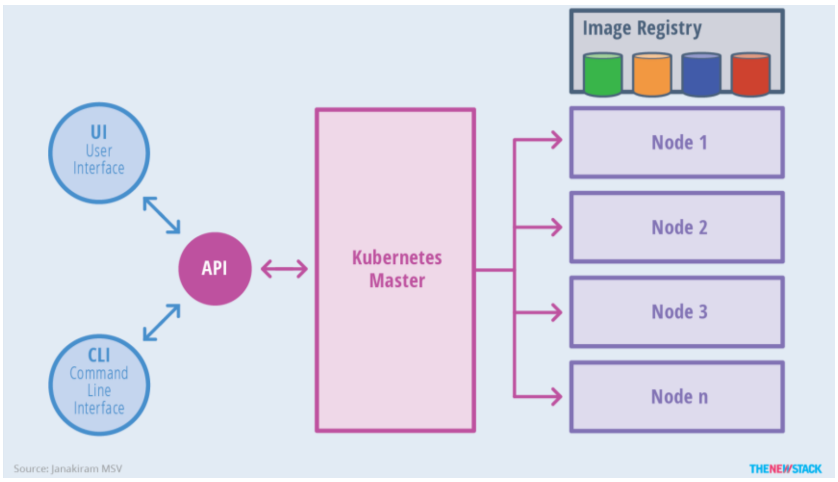

FIGURE 9-10: KUBERNETES ARCHITECTURE[25]

Figure 9-10 illustrates the architecture of Kubernetes which consists of at least one master and multiple compute nodes. The master (see is responsible for exposing the application program interface (API), scheduling the deployments and managing the overall cluster. Each node runs a container runtime, such as Docker, along with an agent that communicates with the master. The node also runs additional components for logging, monitoring, service discovery and optional add-ons. Nodes are the workhorses of a Kubernetes cluster. They expose computing, networking and storage resources to applications. Nodes can be virtual machines (VMs) running in a cloud or bare metal servers running within the data centre. Applications deployed in Kubernetes are packaged as microservices. These microservices are composed of multiple containers grouped as pods (see Figure 9-11). Each container is designed to perform only one task. Pods can be composed of stateless containers or stateful containers. Stateless pods can easily be scaled on-demand or through dynamic auto-scaling.

Contemporary workloads demand availability at both the infrastructure and application levels. In clusters at scale, everything is prone to failure, which makes high availability for production workloads strictly necessary. While most container orchestration engines and PaaS offerings deliver application availability, Kubernetes is designed to tackle the availability of both

[25] Source: https://thenewstack.io/kubernetes-an-overview/

infrastructure and applications. On the application front, Kubernetes ensures high availability by means of replica sets, replication controllers and pet sets. Operators can declare the minimum number of pods that need to run at any given point of time. If a container or pod crashes due to an error, the declarative policy can bring back the deployment to the desired configuration. Stateful workloads can be configured for high availability through pet sets.

For infrastructure availability, Kubernetes has support for a wide range of storage backends, coming from distributed file systems such as Network File System (NFS) and GlusterFS[26], block storage devices such as Amazon Elastic Block Store (EBS) and Google Compute Engine persistent disk, and specialized container storage plugins such as Flocker[27]. Adding a reliable, available storage layer to Kubernetes ensures high availability of stateful workloads.

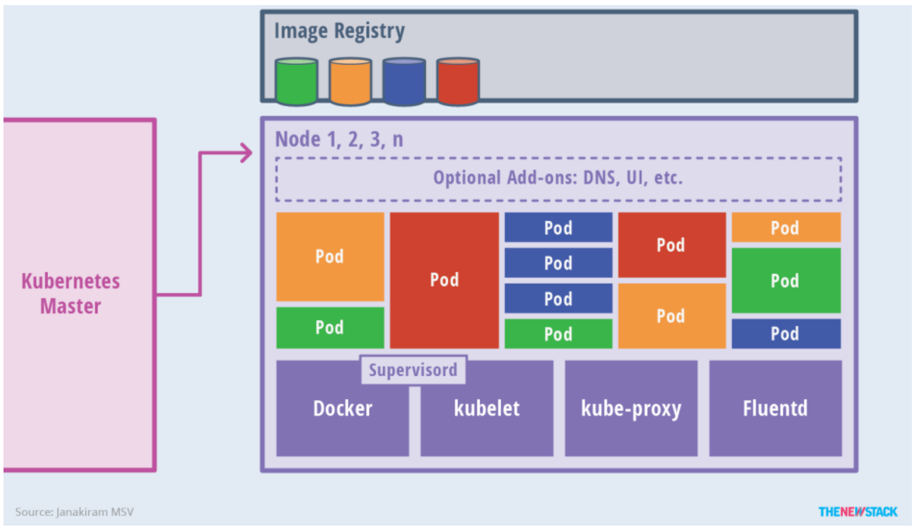

FIGURE 9-11: KUBERNETES NODE ARCHITECTURE[28]

Through federation, it's also possible to mix and match clusters running across multiple cloud providers and on-premises. This brings the hybrid-cloud capabilities to containerized workloads. Customers can seamlessly move workloads from one deployment target to the other. In the following, Table 9-26 and Table 9-27 report the existing and missing Kubernetes capabilities suitable for the 5G-CORAL OCS.

TABLE 9-26: EXISTING K8S CAPABILITIES SUITABLE FOR 5G-CORAL OCS

Capability	Description
Auto-scaling	K8s support horizontal pod auto-scaling, which automatically scales the number of pods based on CPU utilization.
Self-healing	K8s supports pod health checks to ensure availability.
Federation	K8s supports federation for containerized workloads.

TABLE 9-27: MISSING K8S CAPABILITIES REQUIRED FOR 5G-CORAL OCS

Capability	Description
Virtual Machine support	K8s only supports containers.
VIM support	K8s does not interact with any VIM. It's a "monolithic" architecture.
Networking support	K8s focuses on applications and not on network functions. The networking support is tailored to applications while advanced networking operations (e.g., mobile network) are not possible.

[26] https://docs.gluster.org/en/latest/
[27] https://github.com/ClusterHQ/flocker
[28] Source: https://thenewstack.io/kubernetes-an-overview/

Kubernetes requires a minimum of 2 CPUs and 2 GB of RAM on each K8s node. However, it recommends one master node with 4 GB of RAM and 2 CPUs and compute node with total of 10 GB and 4 CPUs. Finally, Table 9-28 and Table 9-29 present the gap analysis of Kubernetes against the functional and non-functional OCS requirements.

TABLE 9-28: 5G-CORAL OCS FUNCTIONAL REQUIREMENTS AND K8S SUPPORT

Functional Requirement	Consideration
Support of harvesting computing capabilities from low-end resources	This is a VIM requirement. K8s adopts a "monolithic" architecture and cannot integrate additional VIMs.
Support of harvesting computing capabilities from mobile resources	This is a VIM requirement. See above.
Support of discovery, configuration, monitoring, allocation, etc. of relevant hardware capabilities (e.g., wireless interfaces, GPIO, GPU, SR-IOV, etc.)	K8s supports Enhanced Platform Awareness (EPA) for relevant hardware capabilities.
Support of integration including at runtime of heterogeneous resources in terms of software and hardware capabilities (e.g., different CPU arch, hypervisors, etc.)	K8s only supports Kubernetes compute nodes as resources.
Support of federation including at runtime of OCS components	K8s supports the federation of multiple K8s instances in different locations.
Support of the interworking with resources external to the OCS (e.g., cloud-to-thing continuum)	K8s only supports the interworking between K8s compute nodes.

TABLE 9-29: 5G-CORAL OCS NON-FUNCTIONAL REQUIREMENTS AND K8S SUPPORT

Non-Functional Requirement	Consideration
Support of deployment of OCS on low end devices (e.g., battery-limited, form-factor, resource constrained, etc.)	Given the medium computing requirements, it is not possible to deploy K8s on low end devices.
Support of deployment of OCS on mobile devices (e.g., car, robot, train, etc.)	K8s networking has been designed with datacentres in mind. Therefore, it does not support mobile devices.
Availability and self-healing mechanisms in error-prone environments	K8s supports self-healing mechanisms for pods and applications.
Support of large deployments in terms of number of resources and geographic areas	K8s supports large deployments via federations.
Support of plugins for extensibility	K8s supports multiple backends at infrastructure level. However, it only supports container-based execution environments.
Capability to adapt to workload changes by provisioning and de-provisioning resources in an automated manner	K8s supports the auto-scaling of pods based on CPU utilization.
Support of multiple tenants participating and co-existing in the same environment	K8s natively supports multi-tenancy at application level and at infrastructure level via federation.

10 Appendix: EFS Stack information model

An initial EFS Stack information model was presented in D3.1[6]. One of the differences of the information model presented in this document compared to what presented in D3.1 [6] is in the different scope. Particularly, D3.1 [6] focused on the information model between the EFS Resource Orchestrator (EFS-RO) and the VIM, while this Appendix focuses on the information model between the users and the EFS-SO. Specifically, Figure 10-1 illustrates the EFS Stack information model as designed by 5G-CORAL. The figure highlights the relations and concepts adopted from the reference standards (i.e., ETSI MEC and ETSI NFV) and the additional information required by 5G-CORAL in order to merge and extend these two frameworks from the Edge down to the Fog.

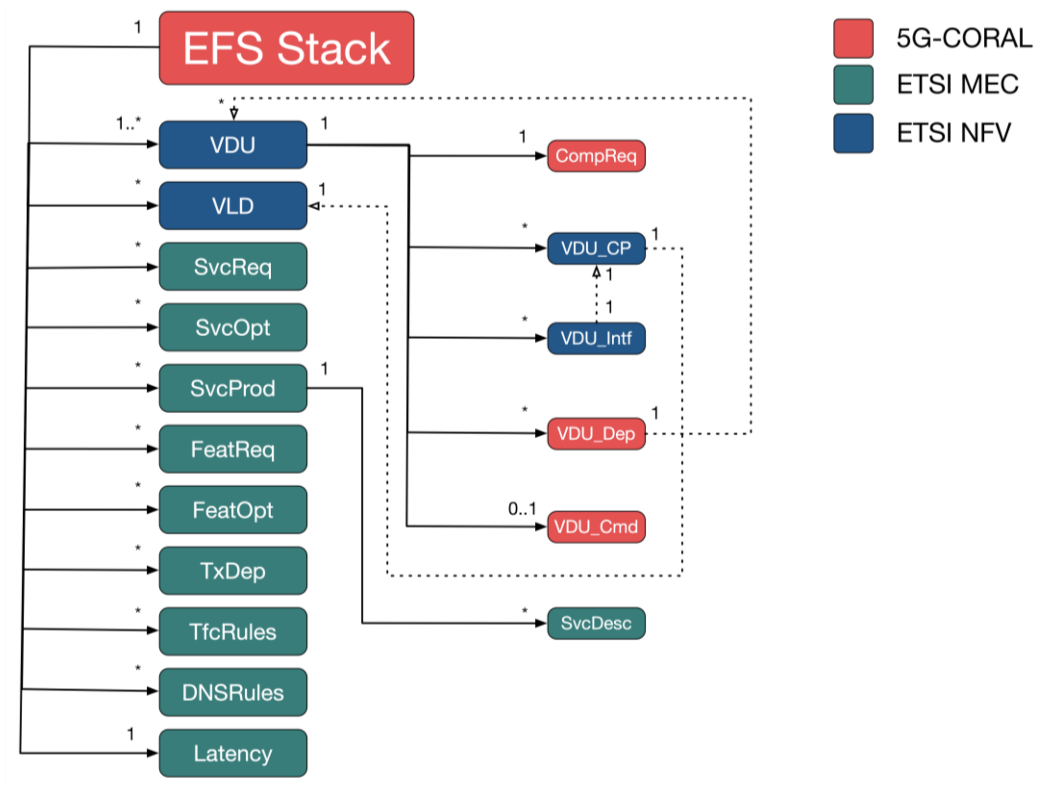

FIGURE 10-1: EFS STACK INFORMATION MODEL

The tables reported in the following describe each field and parameter of the EFS Stack information model. For the sake of similarity, the tables use the same format as used by ETSI NFV and ETSI MEC for their information models.

10.1 Virtualisation Deployment Unit (VDU)

The VDU information element describes the deployment and operational behaviour of a single EFS Atomic Entity.

Name	Type	Cardinality	Description
vdu_uuid	String	1	Unique identifier for the VDU
vdu_name	String	1	Unique name of the VDU
vdu_image	Element	1	Image to be used when instantiating this VDU (see 11.1.1)
vdu_command	Element	1	Command used to start the VDU, present only if vdu_hv_type is BARE (see 11.1.2)

vdu_computation_requirements	Element	1	Computation Requirements for this VDU (see 11.1.3)
vdu_configuration	Element	1	Configuration script used at start of this VDU (see 11.1.4)
vdu_interfaces	Element	0…N	List of virtual interfaces used by this VDU (see 11.1.5)
vdu_hv_type	Enum	1	This of hypervisor needed by this VDU, can be one of {BARE, LXD, KVM, XEN, Docker}
vdu_internal_connection_points	Element	0…N	Internal Connection points defined by this VDU (see 11.1.6)
vdu_io_ports	Element	0…N	Specific I/O ports needed by this VDU (see 11.1.7)
vdu_lcm_hooks	Element	0–1	Hooks/Script called before each LCM action inside the VDU (see 11.1.8)
vdu_depends_on	List	0…N	List of VDUs this VDU depends on (startup dependency)

10.2 Image

The image information element includes the information related to the EFS Atomic Entity package (e.g., location, checksum, format) in case of non-native (e.g., containers) packaging.

Name	Type	Cardinality	Description
uri	String	1	URI for the Image
checksum	String	1	SHA1 checksum to verify the image file
format	String	1	Format of the image (e.g., qcow2, RAW, tar.gz)

10.3 Command

The command information element contains the information for executing a native EFS Atomic Entity.

Name	Type	Cardinality	Description
binary	String	1	Path to the binary file (e.g., file:// /bin/myapp)
args	List	0…N	List of arguments to be passed to the binary

10.4 Computational Requirements

The computation requirements information element includes all the computational and storage requirements for the EFS Atomic Entity.

Name	Type	Cardinality	Description
cpu_arch	String	1	CPU architecture needed by the VDU
cpu_min_freq	Float	1	Minimum CPU frequency required
cpu_min_count	Int	1	Minimum number of vCPU required
ram_size_mb	Int	1	RAM required
storage_size_gb	Int	1	Disk size required
gpu_min_count	Int	1	Minimum number of GPU required
fpga_min_count	Int	1	Minimum number of FPGA required
min_running_time_minutes	Int	1	Minimum running time in minutes in an hour
max_running_time_minutes	Int	1	Minimum running time in minutes in an hour
position	Element	1	Position requirement for this VDU (see 11.1.9)

10.5 Configuration

The configuration information element contains the information related to any eventual configuration script to be executed by the EFS Atomic Entity during the start-up.

Name	Type	Cardinality	Description
conf_type	Enum	1	Configuration script type (CLOUD_INIT, SCRIPT)
script	String	1	Configuration Script

10.6 Interface

The interface information element includes the details required for creating the virtual/physical interfaces required by the EFS Atomic Entity.

Name	Type	Cardinality	Description
name	String	1	Name of the virtual interface
is_mgmt	Bool	1	True if the interface is a management one
mac_address	String	1	MAC address of the interface
internal_cp	Reference	0–1	Reference to an internal connection point connected to this interface
virtual_type	Enum	1	Kind of virtualized interface used (VIRTIO, PARAVIRT, SR_IOV, …)

10.7 Connection Point

The connection point information element allows to interconnect the EFS Atomic Entity with virtual links (see 11.2).

Name	Type	Cardinality	Description
cp_uuid	String	1	Unique UUID for the connection point
vl_id	Reference	0–1	Reference to the virtual link the CP is connected

10.8 IO Port

The IO Port information element includes the details required by the EFS Atomic Entity in terms of hardware IO Ports.

Name	Type	Cardinality	Description
Name	String	1	Name of the IO Port
min_io_ports	Int	1	Minimum number of IO ports needed
io_type	Enum	1	Type of IO ports (I2C, GPIO, BUS, …)

10.9 Life-Cycle Management (LCM) Hooks

The LCM Hooks information element contains the pointers to executables to be run upon LCM operations (e.g., run, stop, migration, etc.).

Name	Type	Cardinality	Description
on_run	String	1	Script called after the VDU is started. This script is executed after the configuration script (see 11.1.4)
on_stop	String	1	Script called before the VDU to be stopped
on_migration _start	String	1	Script called before starting a migration
on_migration _ended	String	1	Script called upon migration completion
migration_type	Enum	1	Specify the kind of migration supported by the VDU: {LIVE, COLD}

10.10 Position

The Position information element expresses location constraints of the EFS Atomic Entity.

Name	Type	Cardinality	Description
lat	String	1	Latitude
lon	String	1	Longitude
radius	Float	1	Radius in meter

10.11 Virtual Link

The Virtual Link information model describes the connectivity type and characteristics.

Name	Type	Cardinality	Description
vl_uuid	String	1	Unique UUID for the Virtual Link
name	String	1	Name of the virtual link
is_mgmt	Bool	1	True if this virtual link is used for management
vl_type	Enum	1	Type of the virtual Link: {ELINE, ELAN}

10.12 EFS Entity/EFS Service

This information model describes the EFS Entity as a whole and includes the EFS Services provided/required by the EFS Entity.

Name	Type	Cardinality	Description
uuid	String	1	Unique UUID of the Ent/Svc
name	String	1	Unique Name of the Ety/Svc
vendor	String	1	Ety/Svc Vendor
soft_version	String	1	Version of the software of the Ety/Svc
ocs_version	float	1…N	List of supported OCS versions by the Ety/Svc
description	String	1	Description of the Ety/Svc
vdus	Element	1…N	VDUs that compose the Ety/Svc (see 11.1)
virtual_links	Element	0…N	Virtual Links that compose the Ety/Svc (see 11.2)
service_required	Element	0…N	Service required by the Ety/Svc to run (see 11.3.12)
service_optional	Element	0…N	Services that are optional for the Ety/Svc (see 11.3.12)
service_produces	Element	0…N	Services produces by the Ety/Svc (see 11.3.10)
feature_required	Element	0…N	EFS Features required by the Ety/Svc to run (see 11.3.13)
feature_optional	Element	0…N	EFS Features optional by the Ety/Svc to run (see 11.3.13)
transport_ dependencies	Element	0…N	Transport dependencies for the Ety/Svc (see 11.3.7)
traffic_rules	Element	0…N	Traffic rules to be created for the Ety/Svc (see 11.3.3)
dns_rules	Element	0…N	DNS rules to be created for the Ety/Svc (see 11.3.2)
latency	Element	1	Latency supported by the Ety/Svc (see 11.3.1)

10.12.1 Latency

The Latency information element expresses the latency requirements of the EFS Entity.

Name	Type	Cardinality	Description
type_unit	Enum	1	Time unit for the latency
latency	Float	1	Max latency supported

10.12.2 DNS Rule

The DNS Rule information element expresses the DNS rules to be configured on the EFS Service Platform for publishing any eventual EFS Service(s).

Name	Type	Cardinality	Description
dns_rule_id	String	1	Unique UUID for the DNS Rule
domain_name	String	1	Domain name for this DNS rule
ip_address_type	Enum	1	IP Address type for this DNS rule: {IP_V4, IP_V6}
ip_addresses	String	1…N	IP addresses for this DNS rule
ttl	Int	1	TTL in seconds for this DNS rule

10.12.3 Traffic Rule

The Traffic Rule information element describes what kind of traffic should be redirected from the underlying network infrastructure to the EFS Entity.

Name	Type	Cardinality	Description
traffic_rule_id	String	1	Unique UUID for this traffic rule
filter_type	Enum	1	Kind of filter: {FLOW, PACKET}
priority	Int	1	Priority of this rule
traffic_filter	Element	0…N	Traffic filter to be applied (see 11.3.4)
action	Enum	0…N	Action to be taken If traffic matches the filter: {DROP, FORWARD_DECAPSULATED, FORWARD_AS_IS, PASSTHOUGH, DUPLICATE_DECAPSULATED, DUPLICATE_AS_IS}
dst_interface	Element	0…1	Destination interfaces for matching traffic (see 11.3.5)

10.12.4 Traffic Filter

The Traffic Rule information element describes the matching rules for redirecting traffic from the underlying network infrastructure to the EFS Entity.

Name	Type	Cardinality	Description
src_addresses	String	0…N	Source addresses
dst_addresses	String	0…N	Destination addresses
src_port	Int	0…1	Source port
dst_port	Int	0…1	Destination port
protocol	String	0…1	Protocol
src_tunnel_address	String	0…1	Source address of the tunnel
dst_tunnel_address	String	0…1	Destination address of the tunnel
qci	Int	0…1	QCI field
dscp	Int	0…1	DSPC field
tc	Int	0…1	TC field

10.12.5 Interface Type

The Interface Type information element describes the type of interface that should be used for redirecting traffic from the underlying network infrastructure to the EFS Entity.

Name	Type	Cardinality	Description
interface_type	Enum	1	Kind of the interface: {MAC, TUNNEL, IP}
tunnel_info	Element	1	Type of tunnel (see 11.3.6)
src_mac_address	String	1	Source MAC address
dst_mac_address	String	1	Destination MAC address
dst_ip_address	String	1	Destination IP address

10.12.6 Tunnel Info

The Tunnel Info information element describes the tunnel configuration that should be used for redirecting traffic from the underlying network infrastructure to the EFS Entity.

Name	Type	Cardinality	Description
tunnel_type	Enum	1	Type of tunnel: {GRE, VXLAN, Zenoh, GTP}
tunnel_src_ address	String	1	Tunnel source address
tunnel_dst_ address	String	1	Tunnel destination address

10.12.7 Transport Dependency

The Transport Dependency information element describes the configuration and requirements for the EFS Service to be consumed/produced by the EFS Entity.

Name	Type	Cardinality	Description
labels	String	0…1	Labels needed for this transport
serializers_list	Enum	0…1	Serialized to be used for this transport: {JSON, XML, PROTOBUF3}
transport	Element	1	Transport information (see 11.3.8)

10.12.8 Transport Descriptor

The Transport Descriptor information element describes the transport information of the EFS Service to be consumed/produced by the EFS Entity.

Name	Type	Cardinality	Description
transport_type	Enum	1	Kind of the transport: {REST_HTTP, MB_TOPIC_BASED, MB_ROUTING, MB_PUBSUB, RPC, RPC_STREAMING, WEBSOCKET}
protocol	String	1	Protocol used by this transport
version	Float	1	Version of the protocol used by this transport
security	Element	1	Security information (see 11.3.9)

10.12.9 Security Info

The Security Info information element includes the authentication tokens to produce/consume the EFS Service by the EFS Entity.

Name	Type	Cardinality	Description
oauth2_info	String	1	OAuth2 Info
grants	Enum	0…1	OAuth2 Grant info: {OAUTH2_AUTORIZATION_CODE, OAUTH2_IMPLICIT_GRANT, OAUTH2_RESOURCE_OWNER, OAUTH2_CLIENT_CREDENTIALS}
token_endpoint	String	1	OAuth2 Token Issue endpoint

10.12.10 Service Descriptor

The Service Descriptor information element includes the details of the EFS Service to be produced/consumed by the EFS Entity.

Name	Type	Cardinality	Description
ser_name	String	1	Name of the service
ser_category	Element	1	Category of this service (see 11.3.11)
version	float	1	Version of this service
transport_supported	Element	1…N	Transport supported by this service (see 11.3.7)

10.12.11 Category

The Category information element defines the category of an EFS Service for better grouping in the EFS Service catalog.

Name	Type	Cardinality	Description
href	String	1	Reference to category in the catalog
uuid	String	1	Unique UUID for this category
name	String	1	Name for this category
version	float	1	Version for this category

10.12.12 Service Dependency

The Service Dependency information element identifies the requirements for consuming an EFS Service from an EFS Entity perspective.

Name	Type	Cardinality	Description
ser_name	String	1	Name of the service needed
ser_category	Element	1	Category of the service needed (see 11.3.11)
ser_transport_ dependencies	Element	1…N	Transport needed for this dependency (see 11.3.7)

10.12.13 Feature Dependency

The Feature Dependency information element identifies the requirements for an EFS Entity in terms of features that need to be supported by the EFS Service Platform.

Name	Type	Cardinality	Description
feature_name	String	1	Name of the needed feature
version	Float	1	Version of the needed feature

11 Appendix: Simulation settings for placement algorithm

This appendix reports the simulations settings used for evaluating the placement algorithms in Section 2.4.

11.1 Simulation settings for EFS Stack and pricing

For the sake of testing the performance of the two placement algorithms based on heuristics (see Section 2.4.3), we consider an EFS Stack composed of 5 EFS Atomic Entities with different sizes and requirements as highlighted in Table 11-1. The reference values, including the pricing, are for an eventual deployment on AWS EC2 in the Paris region[29].

TABLE 11-1: EFS STACK COMPOSITION AND PRICING

Qty	Reference	CPU	RAM	Pricing (AWS Paris)	Tier
1	AWS t3.medium	2	4 GB	$0.0472 per Hour	1x cloud
2	AWS t3.micro	2	1 GB	$0.0118 per Hour	1x cloud, 1x cloud/edge
2	AWS t3.nano	2	0.5 GB	$0.0059 per Hour	1x edge/fog, 1x fog

In our scenario we impose one EFS Entity (i.e., t3.medium) to be always deployed on the cloud. A second EFS Entity (i.e., t3.micro) can be deployed either on the cloud or at the edge. A third EFS Entity (i.e., t3.micro) is deployed at the edge. A fourth EFS Entity (i.e., t3.nano) can be deployed either at the edge or in the fog. Finally, a fifth EFS Entity (i.e., t3.nano) needs to be always deployed in the fog. To encompass the different scales and resource pooling benefits of cloud, edge and fog we introduce a scaling factor for the pricing. In particular, we assume the price for a deployment in the fog to be 1.5 times higher compared to the same deployment in the cloud. Similarly, we assume the price for a deployment in the edge to be 1.2 times higher than the same deployment in the cloud.

11.2 Simulation settings for infrastructure generation

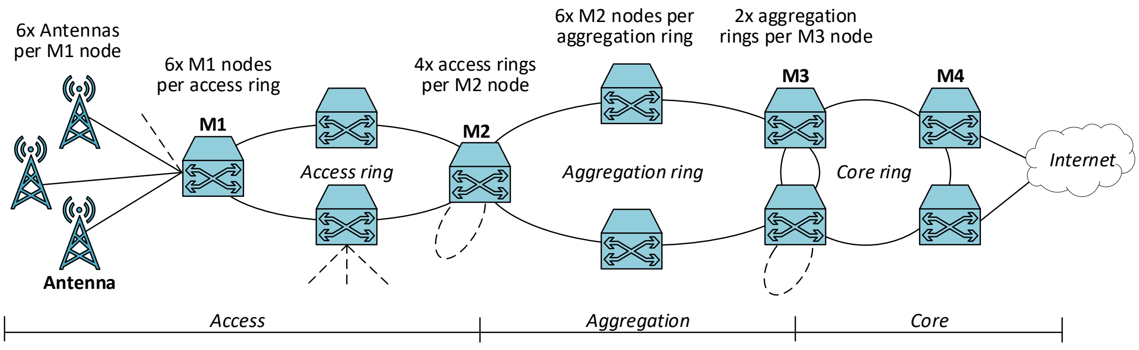
6x Antennas per M1 node
6x M1 nodes per access ring
4x access rings per M2 node
6x M2 nodes per aggregation ring
2x aggregation rings per M3 node
M1
M2
M3
M4
Access ring
Aggregation ring
Core ring
Internet
Antenna
Access
Aggregation
Core

FIGURE 11-1: REFERENCE 5G TRANSPORT NETWORK ARCHITECTURE [43]

In order to simulate a realistic infrastructure, we consider a reference 5G transport infrastructure as proposed in [43] and shown in Figure 11-1. The transport architecture comprises three segments: (*i*) access, (*ii*) aggregation, and (*iii*) core. The access comprises 6 Active Antenna Units (AAUs) for each node M1 connected via a point-to-point link, and 6 nodes M1 connected in a ring topology. Thus, each access ring hence connects a total of 36 AAUs. Next, each aggregation ring comprises 6 M2 nodes, each of which serves as gateway to 4 access rings. Finally, each aggregation ring is served by two M3 nodes for redundancy reasons, while each M3 node provides gateway capabilities to 2 aggregation rings. It is worth noticing that the M1 and M2 nodes are configured in a ring topology (access and aggregation rings, respectively) only at electrical level while at logical level are considered to be connected point-to-point to their

[29] https://aws.amazon.com/ec2/instance-types/

corresponding gateways (M2 and M3, respectively). This means that packets are enqueued only at gateway level and not every time they traverse a node in the ring.

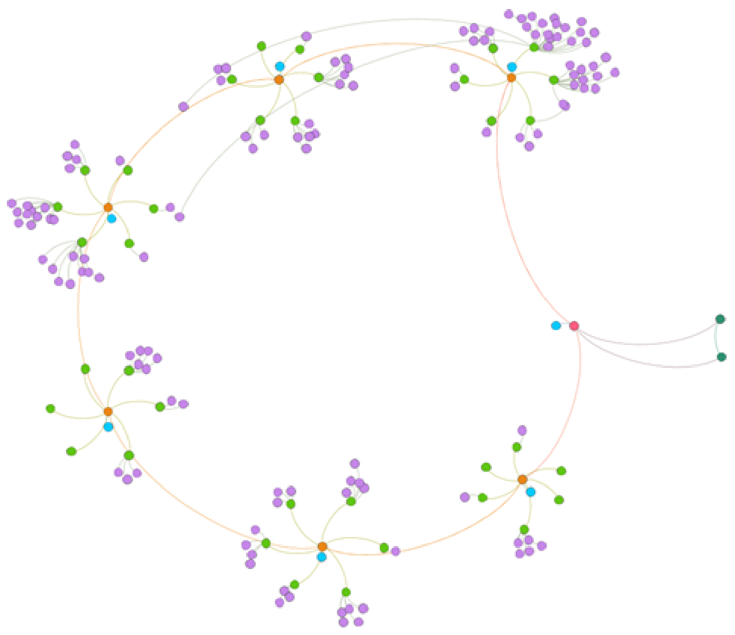

FIGURE 11-2: RANDOMLY GENERATED INFRASTRUCTURE

Starting from the reference transport architecture we randomly generate multiple instances of the infrastructure following the same method proposed in [44] and based on inhomogeneous Poisson point processes with hard-core repulsion. Figure 11-2 shows a random realization of the transport infrastructure. The edge is considered to be a server collocated with a M1 nodes. As an example of edge server, we consider an Azure Data Box[30]. Finally, we consider 128 fog nodes to be collocated with each AAU to encompass for the fog dimension. As an example of fog node, we consider a Raspberry Pi 3 B+[31].

11.3 Simulation settings for infrastructure volatility

To generate the reliability values of the fog nodes and the edge servers, we generate their volatility $(1-\nu(h))$ using exponentially distributed random variables. More specifically, the volatility of an infrastructure node is generated as $\left(1-\nu(h)\right) \sim f(\{Exp\},\lambda)$. Where f is a function that takes the decimal part of a randomly distributed exponential variable and normalizes it in the interval $(\mu \cdot (1 \pm 0.1))$. In our simulation we vary the volatility of fog nodes from $\left((\mu_f = 0.1)\cdot(1\pm 0.1)\right)$ up to $\left((\mu_f = 0.5)\cdot(1\pm 0.1)\right)$. Similarly, the volatilities of the edge nodes vary from $\left((\mu_e = 0.01)\cdot(1\pm 0.1)\right)$ up to $\left((\mu_e = 0.1)\cdot(1\pm 0.1)\right)$. Moreover, we vary incrementally vary the volatility values in ten steps: from $\mu_f = 0.1$ up to $\mu_f = 0.5$ for the fog nodes volatility, and from $\mu_e = 0.01$ to $\mu_e = 0.1$ for the edge nodes volatility. Therefore, in Figure 2-6 (see Section 2.4.4) 100% volatility implies $\mu_f = 0.5$ and $\mu_e = 0.1$.

[30] https://azure.microsoft.com/en-us/services/databox/
[31] https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/

12 Appendix: Simulation settings for federation

This appendix reports the simulations settings used for evaluating the federation performance in Section 4.2 and Section 4.3.

12.1 Simulation settings for federation formation

The following reports the simulation settings leveraged in Section 4.2.4. The performance metrics include the social welfare and the amount of allocated resource in constructed federation structures. We used Gaussian distribution to generate the values of C_i, r_i, c_i, and p_i for each EFS system sp_i. Table 12-1 lists the notations for the means and the standard deviations of these variables with their default values. We used Python library PuLP[32] as an integer programming solver for remote provisioning configuration in each federation. The result of each configuration is averaged over 50 trials.

TABLE 12-1: SIMULATION PARAMETERS FOR FEDERATION FORMATION

Parameter	Description	Default value
$\boldsymbol{n}$	Number of EFS nodes.	10
$\boldsymbol{\mu_k}$	Mean resource capacity of EFS nodes.	1200
$\boldsymbol{\mu_r}$	Mean resource demand	1000
$\boldsymbol{\sigma_k}$	Standard deviation of resource capacity of EFS nodes	110
$\boldsymbol{\sigma_r}$	Standard deviation of resource demand	110
$\boldsymbol{\mu_c}$	Mean unit cost of resource	500
$\boldsymbol{\mu_p}$	Mean unit price of resource.	1000
$\boldsymbol{\sigma_c}$	Standard deviation of unit cost of resource	110
$\boldsymbol{\sigma_p}$	Standard deviation of unit price of resource	110
$\boldsymbol{p}$	Cooperation intensity	0.6

12.2 Simulation setting for resource provisioning in federated environments

The following reports the simulation settings leveraged in Section 4.3.6. We randomly placed ten EFS nodes (numbered from 0 to 9) in a 100 × 100 km^2 area. The capacity of each EFS node was randomly determined with the setting shown in Table 12-2. We varied the number of requests from 50 to 1000. To generate non-uniform distributions of user requests on EFS nodes, the identifier of the serving EFS node of each request was set by applying a floor function to a Gaussian distributed random variable (with mean 5 and standard deviation 2:5) truncated at 0 and 9. We assumed four types of VMs as those offered by Amazon EC2 in US West Region. Each request was a combination of these flavours with most requests demanded Medium and Large VMs. Only a few demanded XLarge and 2XLarge ones. About 60% requests had latency constraints uniformly set in the range from 1 to 100 ms.

TABLE 12-2: SIMULATION PARAMETERS FOR SERVER CAPACITY

Parameter	Distribution	Mean	Standard Deviation
Number of CPU cores	Gaussian	50	10
Amount of memory (GB)	Gaussian	500	50
Amount of storage (GB)	Gaussian	500	100

Other requests did not have latency constraints. When a request was served by the server co-located with the serving EAP, the latency was assumed 1 ms. The latency for serving a guest request was 1 ms plus the propagation delay which is proportional to the physical distance between the request and the serving EFS node.

[32] https://pythonhosted.org/PuLP/