H2020 5G-Coral Project

Grant No. 761586

D2.2: Refined design of 5G-CORAL edge and fog computing system and future directions

Abstract

This deliverable provides the final release of the 5G-CORAL Edge and Fog Computing System (EFS) architecture and design. The deliverable extends the initial EFS design [1] as follows: (1) describing the EFS workflows and the EFS data models; (2) extending the analysis of EFS messaging protocols to incorporate Zenoh and RESTful publish/subscribe messaging; (3) verifying the feasibility of EFS reference design through implementation and experimentation of seven different use cases and (4) a study on EFS resource monitoring.

Document properties

Document number	**D2.2**
Document title	D2.2: Refined design of 5G-CORAL edge and fog computing system and future directions
Document responsible	Industrial Technology Research Institute (ITRI)
Document editor	Samer Talat (ITRI)
Editorial team	Alain Mourad (IDCC), Charles Turyagyenda (IDCC), Chenguang Lu (EAB), Samer Talat (ITRI), Ibrahiem Osamah (ITRI)
Target dissemination level	Public
Status of the document	Final
Version	1.0

List of contributors

Partner	Contributors
ADLINK	Gabriele Baldoni
IDCC	Charles Turyagyenda, Giovanni Rigazzi
ITRI	Samer Talat, Ibrahiem Osamah, Gary Huang
NCTU	Hsu-Tung Chien
TELCA	Aitor Zabala Orive
SICS	Bengt Ahlgren, Saptarshi Hazra
EAB	Chenguang Lu
AZCOM	Riccardo Ferrari, Giacomo Parmeggiani

Production properties

Reviewers	**Antonio De La Oliva, Alain Mourad, Samer Talat, Chenguang Lu, Ibrahiem Osamah**

Document history

Revision	Date	Issued by	Description
1.0	1 June 2019	ITRI	D2.2 ready for publication

Disclaimer

This document has been produced in the context of the 5G-Coral Project. The research leading to these results has received funding from the European Community's H2020 Programme under grant agreement N° H2020-761586.

Table of Contents

List of Figures

List of Tables

List of Acronyms

3GPP	3rd Generation Partnership Project
AMQP	Advanced Message Queuing Protocol
AP	Access Point
API	Application Programming Interface
AR	Augmented Reality
ARP	Allocation and Retention Priority
CBOR	Concise Binary Object Representation
CD	Computing Devices
CPU	Central Processing Unit
C-V2X	Cellular Vehicle-to-everything
DASH	Dynamic Adaptive Streaming over HTTP
DDS	Data Distribution Service
DSRC	Dedicated Short Range Communications
EFS	Edge and Fog Computing System
EFS-VI	EFS Virtualisation Infrastructure
EPC	Evolved Packet Core
ETSI	European Telecommunications Standards Institute
FoV	Field of View
GNSS	Global Navigation Satellite System
HTTP	HyperText Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IP	Internet Protocol
IR	Image Recognition
IQ	In-phase and Quadrature components
LTE	Long Term Evolution
MAC	Media Access Control
MEC	Mobile Edge Computing
MME	Mobility Management Entity
MQTT	Message Queue Telemetry Transport
NB-IoT	Narrow-Band IoT
NFV	Network Functions Virtualisation
OCS	Orchestration and Control System
OSS	Operation Support System
P2P	Peer-to-Peer
PHP	Hypertext Preprocessor
QCI	QoS Class Indicator
QoS	Quality of Service
RAM	Random-Access Memory
RAN	Radio Access Network
RAT	Radio Access Technologies
REST	Representational State Transfer
RSSI	Received Signal Strength Indication
RSU	Road-Side Unit
RTMP	Real-Time Messaging Protocol
SDR	Software-Defined Radio
SSID	Service Set Identifier
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UE	User Equipment
URL	Uniform Resource Locator
vAP	Virtual Access Point
VIM	Virtualisation Infrastructure Manager
vMME	Virtual Mobility Management Entity
VNF	Virtual Network Functions
VR	Virtual Reality
XMPP	Extensible Messaging and Presence Protocol

Executive Summary

One of the key targets of 5G-CORAL is to provide the ultra-low latency requirements. Especially, when the end users with smart devices desire a high-quality service. In order to achieve this ambitious target, 5G-CORAL system utilize the distributed Edge and Fog Computing System (EFS) which has networking, computing, and storage capabilities closer to the end users. In this deliverable, the final version of an integrated and virtualized networking and computing solution adopting virtualized functions, user and third-party applications, and context-aware services are blended together on top of EFS. In this work, we outline the main features of final release of the 5G-CORAL EFS architecture and design. In the final EFS, the services for collection, aggregation, and publishing, use of radio and network context information applications, and virtualized functions are pointed out. Also, refined EFS applications using EFS services from multiple Radio Access Technologies (RAT) and the transport and core networks are developed to improve network KPIs and user QoE. In Summary, this deliverable addresses the following aspects of the 5G-CORAL EFS: the refined EFS design, EFS service platform and messaging, the refined EFS design for the 5G-CORAL use-cases and the EFS monitoring. The following highlights the main achievements in this deliverable:

- Data models for the EFS APIs. In particular, a refined description of EFS internal and external interfaces (E1-E4). Also, E2 interface provides the connectivity enabling distributing and sharing service data between EFS functions and EFS applications via EFS service platform.
- A MQTT-based reference design of EFS service platform is presented.
- A study of Zenoh, NATS, DDS, MQTT and Kafka messaging. The results show that Zenoh and NATS outperform other protocols. These two are recommended to consider if high performance is needed.
- EFS workflows for service discovery and integration especially when a service of a deployed application/function is utilised by another application/function.
- EFS implementations for the 5G-CORAL use cases. The performance is evaluated in each use case by experiments. The experiment results show the benefits of adopting the 5G-CORAL design in service delivery, computation offload, and bandwidth reduction and improve multi-RAT support. Also, the association between the EFS and OCS is investigated for the use cases.
- EFS heterogeneous resource monitoring in the context of 5G-CORAL is also addressed. An open-source tool is used to preform resource monitoring which fits into the EFS design.
- EFS service platform data storage engine design using distributed databases that are consistency, availability and partition tolerance.

Future work is anticipated to focus more extensive study for a large-scale EFS deployment integrating multiple use cases running on the same EFS, which is closer to real business deployment. Another future direction can be the possibility to incorporate the capabilities of machine learning, AI techniques and big data handling into EFS, as well as the interactions and extensions with Cloud. This would require a further extension of the EFS design and make the EFS more intelligent and optimized.

1 Introduction

In contrast to previous mobile communication technologies, 5G promises to support a variety of emerging applications including Mixed (Augmented/Virtual) Reality (AR/VR), Cloud Robotics, Connected Vehicles and several Internet-of-Things (IoT) use cases; some of which require very low end-to-end latency (~0.1-20 milliseconds). This ultra-low latency requirement is extremely challenging to deliver through a purely centralized architecture.

5G-CORAL addresses the ultra-low latency requirement by leveraging the concept of "intelligent edge" to provide networking, computing, and storage capabilities closer to the end users. This is realized through an integrated and virtualized networking and computing solution where virtualized functions, context-aware services, and user and third-party applications are blended together to offer enhanced connectivity and better quality of experience. The 5G-CORAL system constitutes two major building blocks, namely (i) the Edge and Fog Computing System (EFS) subsuming all the edge and fog computing substrates offered as a shared hosting environment for virtualized functions, services, and applications; and (ii) the Orchestration and Control System (OCS) responsible for managing and controlling the EFS, including its interworking with other (non-EFS) domains (e.g., transport and core networks, distant clouds, etc.).

The first deliverable of WP2 [1] provided the initial design of the 5G-CORAL EFS and addressed the following aspects: i) the EFS requirements; ii) the EFS architecture including internal and external interfaces; iii) a comprehensive survey, analysis and selection of the EFS Service platform messaging/communication protocols; and iv) a baseline EFS design for the 5G-CORAL use cases.

This second deliverable provides a refinement of the 5G-CORAL EFS design by addressing the gaps identified in [1] as follows: (1) Describing the EFS workflows and the EFS data models; (2) Extending the analysis of EFS messaging protocols to incorporate Zenoh and RESTful publish/subscribe messaging; (3) Verifying the feasibility of EFS reference design through implementation and experimentation of seven different use cases and (4) a study on EFS resource monitoring. The rest of the deliverable is structured as follows:

Section 2 presents an evolution of the 5G-CORAL EFS design that was initially presented in [1] particularly addressing, the EFS workflows and the EFS data models.

Section 3 presents a refinement of the survey and the analysis of the EFS messaging/communication protocols. The refined analysis extends the study in [1] by incorporating Zenoh and RESTful publish/subscribe as potential EFS messaging protocols.

Section 4 presents the refined EFS design and implementation for each of the 5G-CORAL use cases, namely: Robotics, Virtual Reality, Augmented Reality, Multi-RAT IoT, Connected Cars, High-speed Train, and SD-WAN. The refined design addresses the following aspects, per use case, namely: (1) Decomposition of the use case(s) into their constituent EFS entities and their respective interworking(s); (2) Functional validation and experimental verification; (3) Conclusions and future directions.

Section 5 presents a study of EFS resource monitoring and highlights the following key aspects: analysis of state-of-the-art (SoA) monitoring framework, mapping the monitoring approaches to 5G-CORAL and EFS design.

Finally, in **Section 6**, a conclusion is presented summarizing the findings of this deliverable, as well as setting the prospects for future directions.

2 Refined EFS Design

This section presents an evolution of the 5G-CORAL EFS design that was initially presented in [1] particularly addressing, the EFS workflows and the EFS data models. First, we provide an overview of the 5G-CORAL EFS components and the corresponding interfaces in section 2.1. Second, we present the EFS E2 interface and data models in section 2.2. Finally, we present a description of the EFS workflows in section 2.3.

2.1 Overview of 5G-CORAL architecture and EFS components

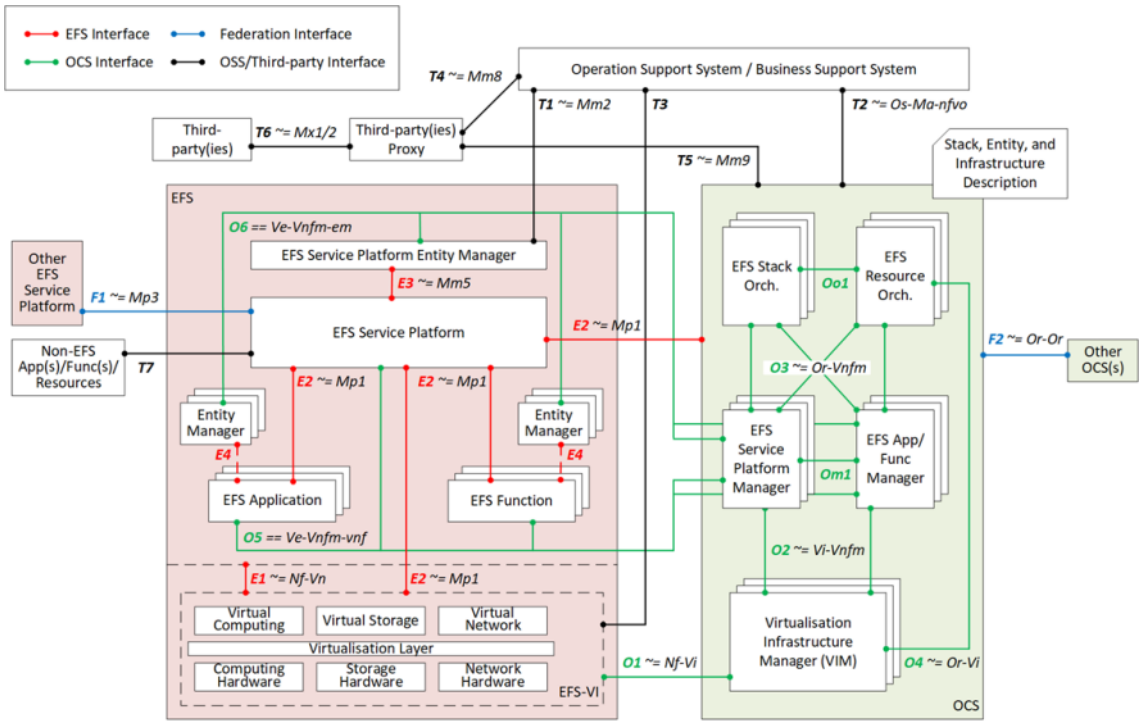


FIGURE 2-1: 5G-CORAL SYSTEM ARCHITECTURE

Figure 2-1 presents the 5G-CORAL system architecture [42] composed of the following two sub-systems, namely.

- **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 a service platform, functions and applications on top of the available resources and may interact with other domains' EFSs.
- **Orchestration and Control System (OCS):** an OCS is a logical system responsible for 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 other domains' OCSs.

The EFS constitutes the following components [1].

- **EFS Virtualization Infrastructure (EFS-VI):** The EFS virtualization infrastructure (EFS-VI) is the totality of the hardware and software components that build up the environment in which EFS entities (i.e. EFS applications, EFS functions and EFS service platform) are deployed, managed and executed. The EFS-VI is geographically distributed across several locations and composed of Fog nodes and Edge nodes.

- **EFS entities,** namely: EFS applications, EFS functions, the EFS service platform and their respective entity managers. An EFS entity is comprised by at least one atomic entity. An atomic entity is an unpartitionable computing task executed in the EFS.
    - **EFS Function:** A software entity comprised of at least one atomic entity deployed in EFS for networking infrastructure.
    - **EFS Application:** A software entity comprised of at least one atomic entity deployed in EFS for end users and third parties.
    - **EFS Entity Managers:** Analogous to the ETSI NFV element managers, the EFS entity managers are responsible for FCAPS management of the EFS service platform, Functions and Applications. This includes configuration management, fault management, Security management, accounting and collecting performance measurement results.
    - **The EFS Service Platform:** A logical data exchange platform constituting: (1) Data storage to keep the collected information from applications/functions and edge/fog resources. (2) Messaging/communication protocols to gather/provide information from/to applications/functions and edge/fog resources.

2.1.1 EFS internal and external interfaces

Table 2-1summarizes the EFS interfaces according to Figure 2-1. There are two categories of EFS interfaces namely internal and external interfaces. The former handles the message exchanges within the EFS while the later communicates with the non-EFS entities such as the OCS, the Operation Support System/Business Support System (OSS/BSS) and the Non-EFS applications/functions/resources. WP2's refinement of the EFS design focused on the internal EFS interfaces, namely: E1, E2, E3 and E4.

TABLE 2-1: EFS INTERFACES

ID	ETSI NFV/MEC ref. point	Description
E1	ETSI NFV: Nf-Vn	This is the reference point between the EFS virtualisation infrastructure (EFS-VI) and the EFS entities, i.e. EFS applications, EFS functions, the EFS service platform and their respective entity managers.
E2	ETSI MEC: Mp1	This is the reference point between the EFS service platform and the following: EFS applications, EFS functions, EFS virtualisation infrastructure and the OCS.
E3	ETSI MEC: Mm5	This is the reference point between the EFS Service platform and the EFS Service platform entity manager.
E4	ETSI MEC: Mm5	This is the reference point between the EFS application/EFS functions and their respective entity managers.
O1	ETSI NFV: Nf-Vi	This is the reference point between the Virtual Infrastructure Manager (VIM) and the EFS virtualisation infrastructure (EFS-VI).
O5	ETSI NFV: Ve-Vnfm-Vnfm	This is the reference point between EFS functions or applications and the EFS Service Platform Manager.
O6	ETSI NFV: Ve-Vnfm-em	This is the reference point between the entity managers of functions, applications and EFS service platform and the EFS Service Platform Manager.
T1	ETSI MEC: Mm2	This is the reference point between the EFS service platform entity manger and the Operation Support System/Business Support System (OSS/BSS).
T3	None	This is the reference point between the EFS virtualisation infrastructure (EFS-VI) and the Operation Support System/Business Support System (OSS/BSS). ETSI NFV has an interface between NFVI and OSS/BSS, however, this interface is not named and is classified under "other references"

ID	ETSI NFV/MEC ref. point	Description
T8	None	This is the refence point between the EFS service platform and the Non-EFS applications, functions and resources. There is no equivalent interface both in ETSI NFV and ETSI MEC.
F1	ETSI MEC: Mp3	This is the reference point between the EFS service platform and other EFS Service platform(s).

The E1 interface is tightly coupled to the virtualization technologies adopted by the EFS Virtualization Infrastructure (EFS-VI), i.e. either docker, LXD/LXC containers, KVM virtual machines, etc. A detailed description of the E2 interface is provided in section 2.2. The E3 and E4 interfaces are implementation-specific interfaces between the EFS service platform and its entity manager, and the EFS application/functions and their entity managers, respectively. The role of these entity managers includes providing life cycle management for their respective entities by interacting with the OCS over the O6 interface and carry out other configuration tasks required for execution on the EFS platform, such as, network configuration and providing internal configuration files. The entity managers are deployed together with their respective entities by the OCS.

The EFS entity managers interact with the OCS over the O6 interface ([5], Figure 2-1) which has similarities with the NFV Ve-Vnfm-em interface [6]. The latter interface has two non-mandatory services to be provided by the entity managers:

- Virtual Network Functions (VNF) indicator: a subscription and retrieval service for indicator values that provide information about VNF behaviour (i.e., EFS entity behaviour).
- Life-cycle management coordination: a service supporting coordination of life-cycle management functions for VNF instances and their components (i.e., EFS entities and their atomic entities). Possible operations are: CreateSnapshot, RevertToSnapshot and ChangeCurrentVnfPackage.

Furthermore, the interface has five services to be provided by the OCS (and used by the entity managers): life-cycle management, performance management, fault management, policy management and snapshot package management.

2.2 EFS E2 interface and data models

Figure 2-2 illustrates the EFS service platform interface "E2". It is the reference point between the EFS service platform and a number of different entities of the architecture, including the EFS applications and functions, as well as the OCS. As the EFS is compliant with ETSI MEC [2], we adopt the corresponding ETSI MEC interface "Mp1" [3][4] as the basis for the EFS E2 interface (REST API to the left in Figure 2-2). The EFS service platform also provides an MQTT publish/subscribe message bus as the main mechanism for connecting the EFS entities (to the right in the Figure 2-2). In ETSI MEC terminology, this is a *platform-provided transport*, meaning that the communication relies on the platform for delivering messages, as opposed to direct communication between client and service. In the rest of this section, we describe how the "Mp1" is used and extended by the EFS.

A newly deployed EFS application or function needs a way to discover the EFS service platform and through that discovery understand how to connect to the E2 service interface. This discovery procedure is described in Deliverable D3.1, Section 4.3 [5]. Different methods can be used depending on the network infrastructure. One possible method is to use DNS-SD, where a service

record for the EFS is created in DNS together with a default DNS domain configured via DHCP (or equivalent), or alternatively via mDNS.

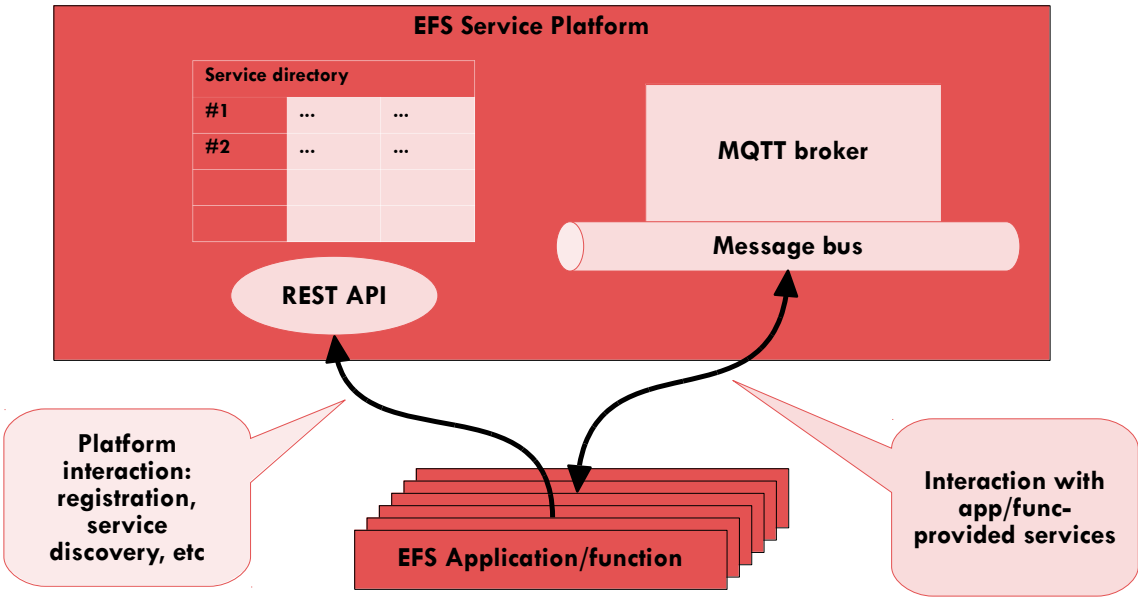


FIGURE 2-2: OVERVIEW OF THE E2 SERVICE INTERFACE OF THE EFS

ETSI MEC Mp1 is a REST-based API that includes functionality for registering and finding services. Alternatively, an EFS implementation can map the Mp1 REST interface to the MQTT pub/sub message bus, and thus only use MQTT as the transport for the E2 interface. In this case, discovery of the EFS service platform is equivalent to discovery of the MQTT broker. The methods for creating and updating a service resource and retrieving information about a service resource are listed in Table 2-2[4]. The root of the REST resource URI is "{apiRoot}/mp1/v1", where "{apiRoot}" is received as part of the platform discovery, as described in the previous subsection.

TABLE 2-2: METHODS IN ETSI MEC MP1 FOR HANDLING SERVICE RESOURCES AND QUERYING FOR TRANSPORTS

Resource name	Resource URI	HTTP method	Meaning
A list of meService	/services	GET	Retrieve information about a list of meService resources
		POST	Create a meService resource
Individual meServices	/services/{serviceId}	GET	Retrieve information about a meService resource
		PUT	Update the information about a meService resource
Transports	/transports	GET	Retrieve information about available transports

The main data structure used by these methods is the "ServiceInfo" resource which describes the properties of a particular service. Table 2-3 lists the attributes of ServiceInfo with comments on how they are used in the EFS.

TABLE 2-3: SERVICE INFO DATA STRUCTURE – FIRST FOUR COLUMNS FROM GS MEC 011

Attribute name	Data type	Cardinality	Description	EFS use
serInstanceId	String	0..1	Identifier of the service instance assigned by the MEPM/mobile edge platform. Shall be absent in POST requests, and present otherwise.	The EFS platform assigns UUIDs (version 4 – random) for this attribute.
serName	String	1	The name of the service. This is how the service producing mobile edge application identifies the service instance it produces	Name of the EFS service. This is also part of the MQTT topic prefix.
serCategory	CategoryRef	0..1	A Category reference […]	In EFS, the category is used to name the EFS application or function the service is part of.
Version	String	0..1	The version of the service.	Service version, also used to form the topic prefix.
transportId	String	0..1	Identifier of the platform-provided transport to be used by the service. [...]	Normally "MQTT", referring to the EFS-provided MQTT transport
transportInfo	TransporInfo	0..1	Information regarding the transport used by the service. […]	Not normally used by the EFS
Serializer	SerializerTypes	1	Indicate the supported serialization format of the service.	Normally "JSON"

Information about the MQTT message bus transport provided by the EFS platform can be retrieved by EFS applications/functions using the E2 interface with the GET method on the /transports resource (see Table 2-1). This method returns a list with a TransportInfo data structure as presented in Table 2-4.

TABLE 2-4: TRANSPORTINFO RESOURCE DESCRIBING THE PLATFORM-PROVIDED MQTT TRANSPORT

Attribute name	Data type	Cardinality	Description	Value for EFS-provided MQTT transport
Id	String	1	The identifier for this transport.	A UUID
Name	String	1	The name of this transport.	"EFS service transport"
Description	String	0..1	Human-readable description of this transport.	"EFS platform-provided default transport for EFS services"
Type	TransportTypes	1	The type of the transport.	MB_TOPIC_BASED
Protocol	String	1	The name of the protocol used.	"MQTT"
Version	String	1	The version of the protocol used.	"3.1.1"
Endpoint	EndPointInfo	1	Information about the endpoint to access the transport.	Specifies one or more MQTT URIs for accessing the transport.
Security	SecurityInfo	1	Information about the security used by the transport.	OAuth 2.0 security information.
implSpecificInfo	Not specified	0..1	Additional implementation specific details of the transport.	This field can be used to indicate the supported MQTT QoS levels

As mentioned above, EFS services are normally interacted with over the EFS platform-provided MQTT message bus. A particular EFS service freely defines the data format of its messages, but it is recommended to use JSON as the serialiser. The EFS however imposes a structure for the MQTT topics. Similar to URIs in ETSI MEC, the topic prefix to be used by services is defined as: {apiRoot}/{apiName}/{apiVersion}/{serInstanceId}, where the fields are defined as follows:

- {apiRoot} is defined by the TransportInfo for the MQTT transport
- {apiName} uniquely names the EFS service or group of services using a particular API definition – "serName" in the ServiceInfo data structure
- {apiVersion} is a version identifier for the service – "version" field in ServiceInfo
- {serInstanceId} is an identifier for the particular instance of the service – "serInstanceId" field in ServiceInfo. The rationale for this field is that many instances of a service may run in parallel, and clients may select, using topic wildcards in the MQTT subscription, to interact with all instances or only certain instances, depending on, for example, the performance or location.

Several of the 5G-CORAL proof-of-concepts (PoCs) provide EFS services that can be used by other applications. Data models (JSON schemas) for the messages of some of these services are provided in Appendix 7.

2.3 EFS workflows

Figure 2-3 illustrates an example workflow when the OCS deploys an EFS application/function within the EFS, and the subsequent steps taken by the entities of the application/function and by the EFS service platform. The example assumes that a service of the deployed application/function is then used by another application/function. The steps are as follows:

1. **The OCS deploys an EFS application/function.** The OCS decides where the virtual images/containers with the EFS atomic entities of an EFS application/function should execute, and arranges for the images to be deployed. The details of this process is not in the scope of this deliverable.
2. **The EFS application/function starts.** The EFS atomic entities of the application/function starts. They find the EFS service platform interface (EFS E2 interface) using the mechanisms outlined in Deliverable 3.1 [5], Section 4.3.
3. **The EFS application/function registers with the EFS service platform.** The EFS application/function registers its services, if any, with the EFS service platform by creating one or more service resources using the EFS E2 interface. In the same process, the transport used by the service is specified.
4. **A second EFS application/function finds the EFS service.** Another EFS application/function finds the services by querying for services using the EFS E2 interface. The information received includes a URI or similar where the service can be accessed.
5. **The second EFS application/function carries out its operation.** The application/function accesses the desired EFS service using the information received from the EFS service platform.

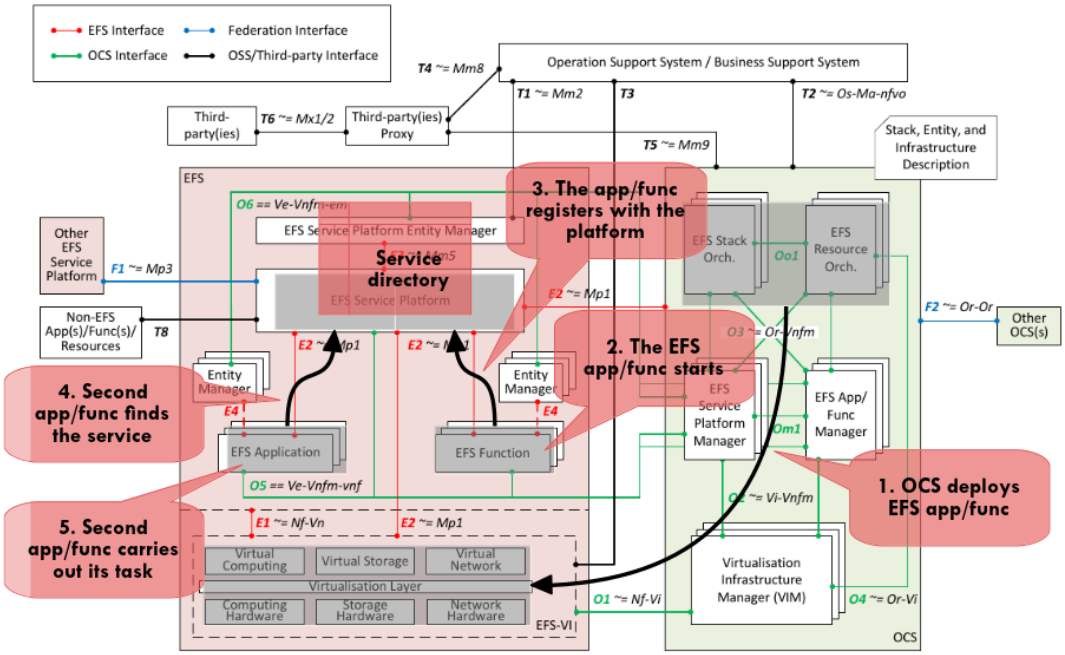


FIGURE 2-3: EFS WORKFLOW

3 EFS service platform and messaging protocols

This section refines the survey and the analysis of the EFS messaging/communication protocols presented in [1]. The refined analysis extends the study in [1] by incorporating Zenoh and RESTful publish/subscribe as potential EFS messaging protocols. First, we provide an overview of the EFS service platform in Section 3.1. Then, we present a survey on Zenoh and RESTful publish/subscribe messaging protocols in Section 3.2. Finally, we introduce a refined analysis of the EFS messaging/communication protocols Section 3.3.

3.1 EFS service platform

The EFS service platform is a logical data exchange platform within EFS consisted of (i) data storage to keep the collected information from applications/functions and edge/fog resources, and (ii) communication protocol to gather/provide information from/to applications/functions hosted in edge/fog resources. The role played by the EFS service platform can be deemed as a 'middleman' in charge of storing and distributing the subscribed data of a service to the data subscribers, while the service data are published by the data publishers and organized as EFS services by the EFS service platform, Figure 3-1. It specifies the protocols and mechanisms for data communication, storage and management and serves both EFS and non-EFS functions and applications though APIs. The non-EFS functions and applications are hosted outside of EFS, such as on the Transport Network and Core network, as well as distant clouds. For example, the Radio Access Network (RAN) functions can publish the RAN context information and the platform can abstract and organize the information as a RAN context service. The subscribing applications of the RAN context service get the context information and use them for their own purposes. For example, a load balancing application can avoid using overloaded RATs based on the RAN context information.

The EFS service platform collects data from EFS Applications/Functions and publish the collected data to EFS Applications/Functions that consume data. In order to push data to the targeted entities, the messaging protocol is a key ingredient of the EFS Service Platform design. Instead of devising new message protocols, WP2 has examined and analyzed several existing messaging protocols, as detailed in [1]. Sections 3.2 and 3.3 extend this analysis to incorporate Zenoh and RESTful publish/subscribe messaging.

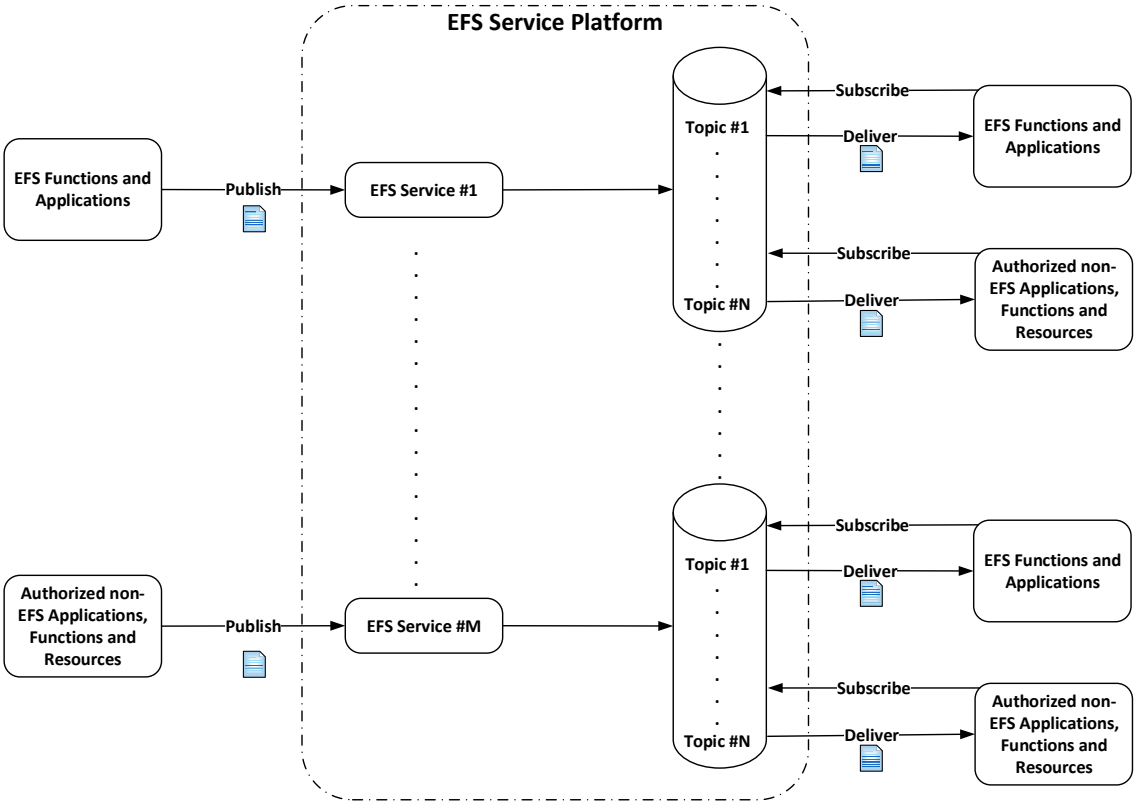


FIGURE 3-1: PUBLISH/SUBSCRIBE MESSAGING AMONG EFS ENTITIES

3.2 Extended survey of EFS messaging/communication protocols

In [1] WP2 studied the following messaging/communication protocols: DDS, MQTT, AMQP, Extensible Messaging and Presence Protocol (XMPP), Kafka, NATS, and Confluent. In light of the adoption of RESTful APIs, by 3GPP service-based architecture (SBA) 5G Core network, WP2 investigated the feasibility of RESTful publish/subscribe as a potential messaging/communication protocol for the EFS. Additionally, Zenoh was also considered in accordance with the OCS experimental framework in WP3.

3.2.1 Zenoh

Zenoh's [17] goal is to bring data-centric abstractions and connectivity to devices that are constrained with respect to the node resources, such as compute, storage, power, and the networking. Zenoh applications coordinate by autonomously and asynchronously writing and reading data into a data space while being decoupled in time and space. The abstraction of a time decoupled data-space is essential in supporting applications that can have sleep cycles and specifically in decoupling the availability of data with the availability of the application that wrote it. Zenoh relies on resources to identify the information to be exchanged between readers and writers, and on resource properties to specify the properties of exchanged data. A Zenoh resource is a closed description for a set, if the cardinality of the set is one we call it a trivial resource. A Zenoh resource is described by means of a URI [17] which may only include path expansions.

The data read and written by Zenoh applications is associated with one or more resources identified by a URI, that may contain '?', '*' and '**' wildcards. '?' matches exactly a single character excluding the path separator, '*' matches any number of characters excluding the path separator, and '**' matches any number of characters including the path separator.

Zenoh Pub/Sub mechanism provides peer-to-peer, client-to-broker and broker-to-broker communication. It provides a scalable routing mechanism for many-to-many communication with different levels of reliabilities. The protocol is designed to be lightweight in both computational and networking overheads. It can leverage both connection-oriented transports as well as connection less packet-based transport, which implies that it can run on top of L3 or L2 networks.

Detailed benchmarking of Zenoh against other well-known Pub/Sub protocols is provided in section 3.3 while a comparison between Zenoh and NATS is provided in Appendix 9.

3.2.2 RESTful publish/subscribe messaging

WP2 investigated a variant of Kafka, namely Kafka REST as an example of RESTful publish/subscribe messaging protocols. Kafka in its native version is not RESTful, however, a Kafka REST Proxy (Figure 3-2) provides a RESTful interface to a Kafka cluster [19] [20]. This makes it easy to: produce and consume messages; view the state of the cluster; and perform administrative actions without using the native Kafka protocol or clients. Some use cases that could benefit from this configuration include reporting data to Kafka from any frontend application built in any language and ingesting messages into a stream processing framework that doesn't yet support Kafka.

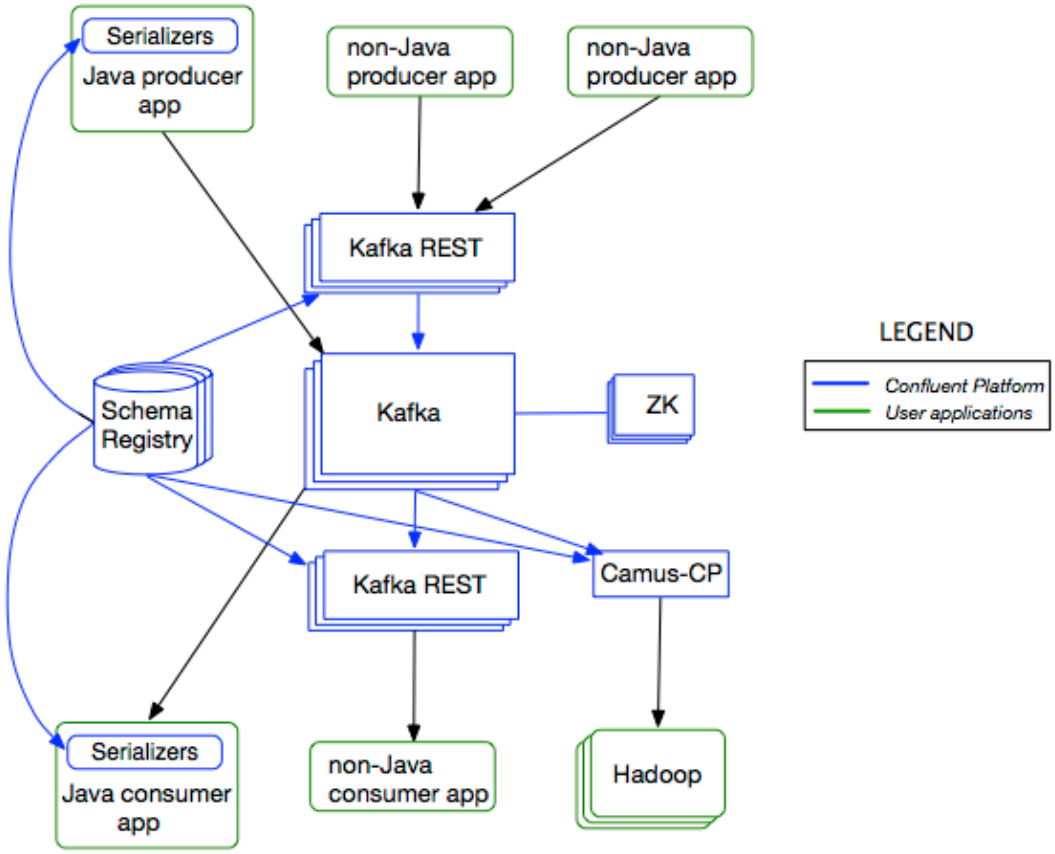


FIGURE 3-2: KAFKA REST PROXY ARCHITECTURE [43]

Kafka uses a pooling system for its notifications and a Transmission Control Protocol (TCP) connection is not maintained. In each request a TCP/HTTP connection is established. When a request is responded to, the session finalizes. If a connection needs to be established by consumer or producer, Kafka REST Proxy could be substituted by a proxy with WebSocket [21].

3.3 Refined analysis of EFS messaging/communication protocols

WP2 conducted an experimental performance evaluation of: NATS, DDS, MQTT, Zenoh and Kafka REST. The objective of the experiment was to compare the performance of the messaging protocols in terms of throughput and messages transmitted per seconds; analyzed over varying payload sizes.

The experimental setup, constituted a single compute node, running Ubuntu 16.04, Intel i7@4.0GHz and 32GB of RAM (Figure 3-3). For each protocol, the test was repeated with 1 million messages for each payload size. All the protocols were tested in brokered deployment, meaning that there was a client publishing to a broker and a client subscribing to the broker. All the results were taken from the subscriber side while the QoS used for all protocols was the standard best effort QoS. Table 3-1presents the raw values recorded during the experimentation.

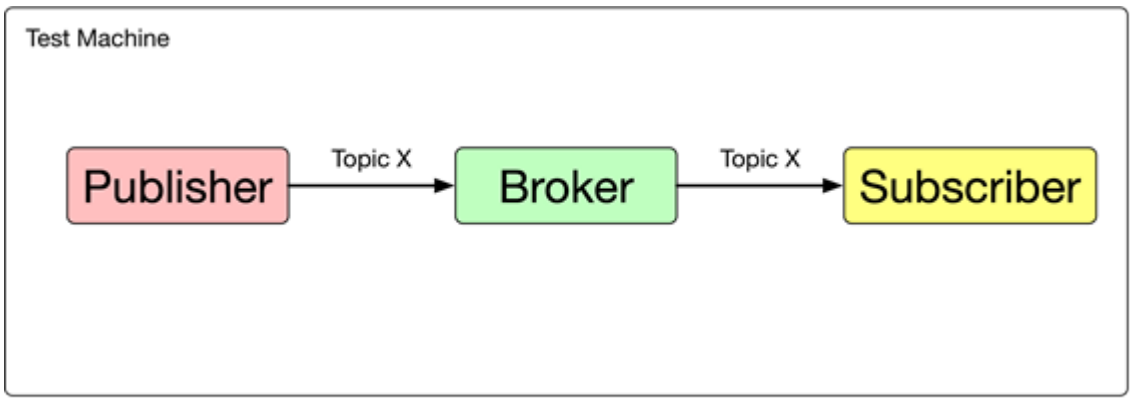


FIGURE 3-3: EXPERIMENTAL SETUP

TABLE 3-1: PUB/SUB MESSAGING PROTOCOLS RESULTS

	Zenoh		NATS		DDS (Cyclone)		MQTT		Kafka REST [40]	
Payld.	**msgps**	**Mbps**	**msgps**	**Mbps**	**msgps**	**Mbps**	**msgps**	**Mbps**	**msgps**	**Mbps**
8	9048065	579	3625473	232	1313241	84	37552,82	2	9156	55
16	10704897	1370	3468433	443	1316622	168	37035,48	4	8784	63
32	10199553	2611	3451773	883	1248549	319	37035,48	9	8499	69
64	8379649	4290	3265401	1671,	1191086	609	37035,48	18	7231	72
128	5757057	5895	2944434	3015	1044868	1069	35195,55	36	6557	78
256	3564289	7299	2466174	5050	839787	1719	39238,76	80	5578	85
512	2063265	8451	1810187	7414	619562	2537	41087,68	168	5116	98
1024	1071489	8777	1276843	10459	352754	2889	36364,3	297	4976	110
2048	505801	8287	727003	11911	186471	3055	28218,86	462	4001	117
4096	265037	8684	352835	11561	96194	3152	18043,11	591	3759	123
8192	134767	8832	185106	12131	54913	3598	7859,93	515	3222	134
16384	70067	9183	88435	11591	30019	3934	4239,57	555	2765	152

Figure 3-4 presents the throughput comparison among: Zenoh, NATS, Eclipse Cyclone DDS, Eclipse Mosquitto for MQTT and Kafk REST. It was observed that Zenoh exhibited the best throughput performance at small payload sizes. It was also observed that the best usage of multithreading occurs at higher payload sizes where NATS exhibited the best throughput performance. At high payload sizes; DDS, MQTT and Kafka REST all had a throughput performance less than half of NATS and Zenoh.

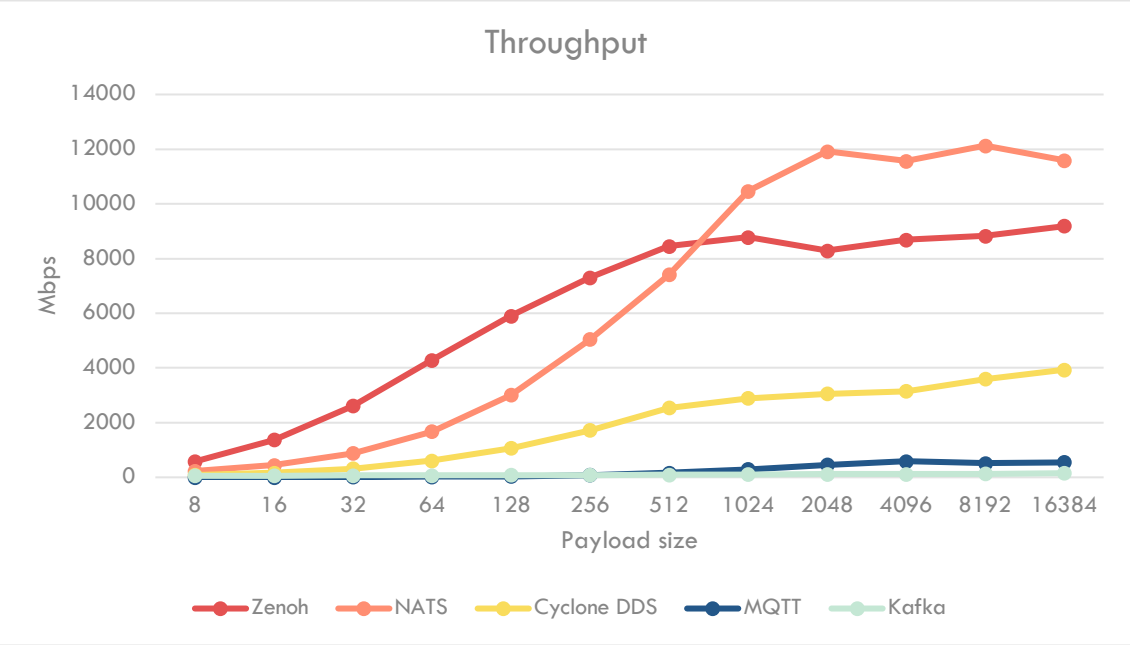


FIGURE 3-4: THROUGHPUT OVER PAYLOAD SIZE

Figure 3-5: presents the messages transmitted per second comparison among: Zenoh, NATS, DDS, MQTT and Kafka REST.

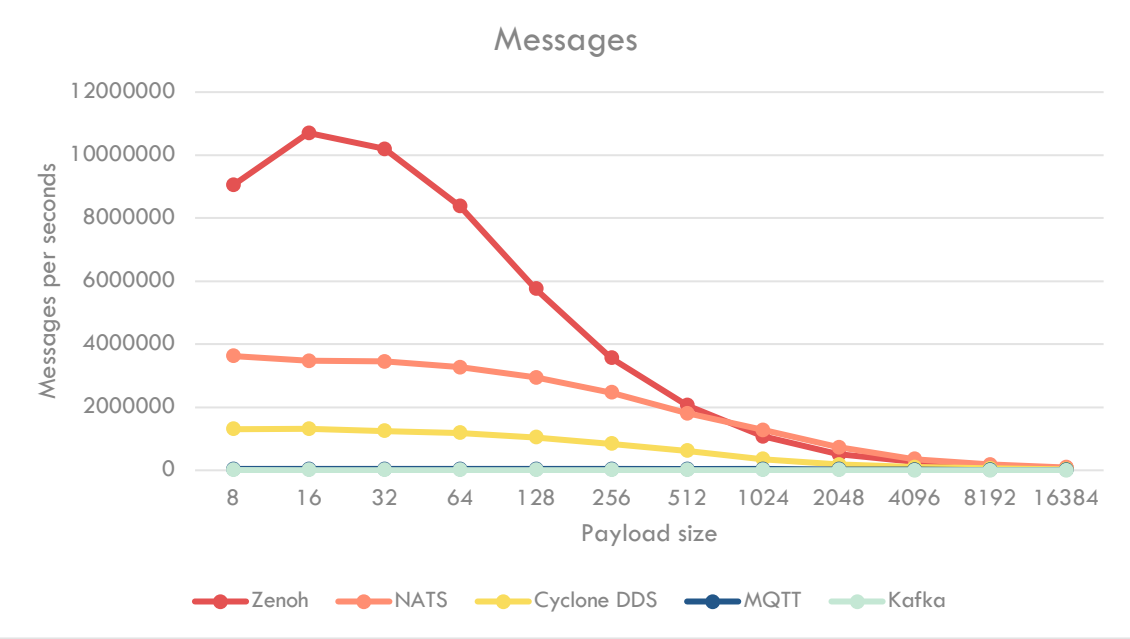


FIGURE 3-5: MESSAGES PER SECOND OVER PAYLOAD SIZE

As expected, increasing the payload size reduces the number of messages transmitted per second for each protocol. It was observed that for small payload sizes, Zenoh transmitted the largest number of messages per second; this is due to Zenoh's wire efficiency, i.e. very low overhead. The performance of all the protocols is comparable for payload sizes greater than 1024 bytes.

The analysis highlights the fact that new Pub/Sub protocols, for example NATS and Zenoh, are designed to have high performance both in terms of throughput and messages transmitted per seconds. However, presently support for these new protocols is fairly limited by the number of client libraries available in the public domain.

While in [1], WP2 adopted DDS and MQTT as the reference/baseline messaging protocols for the EFS, this analysis reveals that enhancements to the EFS could benefit from adopting new Pub/Sub protocols, e.g. NATS and Zenoh.

4 Refined EFS design for 5G-CORAL use-cases

This section presents the refined EFS design for each of the 5G-CORAL use cases, namely: Robotics (Section 4.1), Virtual Reality (Section 4.2), Augmented Reality (Section 4.3), Multi-RAT IoT (Section 4.4), Connected Cars (Section 4.5), High-speed Train (Section 4.6), and SD-WAN (Section 4.7). The refined design addresses the following aspects, per use case, namely: (1) Decomposition of the use case(s) into their constituent EFS entities and their respective interworking(s); (2) Functional validation and experimental verification; (3) Conclusions and future directions.

4.1 Robotics

The Fog-assisted Robotics (FIGURE 4-1) use case comprises of two different scenarios, both envisioned in a Shopping Mall scenario. The first scenario envisions the robots cleaning 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.

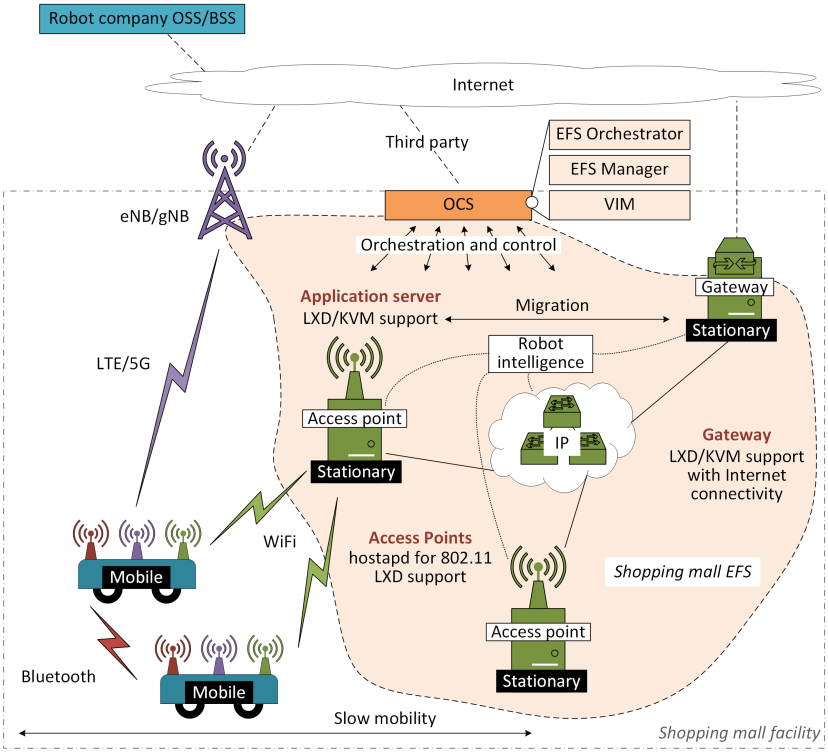


FIGURE 4-1: FOG-ASSISTED ROBOTICS IN THE SHOPPING MALL

4.1.1 Refined EFS design and functional validation

In this use case, the focus is on the delivery of goods by a group of robots working synchronously. Data related to the stock level of each shop is collected and analysed at the EFS and is used to

determine which good needs to be delivered to which shop. Some items may be too large for one robot alone and would require the synchronized operation of two or more robots to carry it. Thanks to the vicinity of the brain to the robots, it is hence possible to achieve tight coordination between the robots.

In the EFS, the location service of the robots is consumed by a robot intelligence application in the EFS, so it can calculate the route that the robots should take to arrive at the point of delivery. Note that the location of the robot can be evaluated via different means. For instance, LiDAR could be used by the robot to figure out its own location, and such location can be published to the EFS service platform for other applications/functions to consume. The robot intelligence application instructs the movement of the robot via wireless connectivity, the protocol functionalities of which are also hosted in the EFS. It is worth noting that, the EFS computing tasks for this use case, such as robot intelligence, service platform, and radio connectivity functions, can be migrated among different EFS resources (e.g. Fog CDs) along the route. The placement of the EFS computing tasks is transparently handled by the OCS. The EFS entities involved in the robotics use case, as well as their interconnection, is illustrated in Figure 4-2. Table 4-1 presents a description of the robotics use case EFS entities depicted in Figure 4-2.

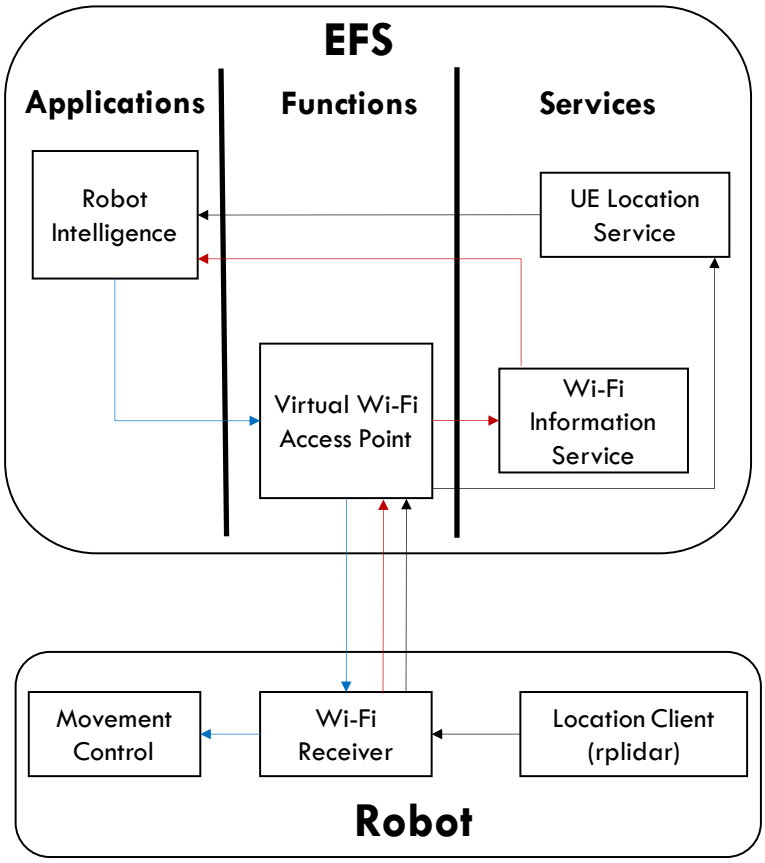


FIGURE 4-2: EFS ENTITIES INTERCONNECTION FOR THE ROBOTICS USE CASE

TABLE 4-1: SUMMARY OF EFS ENTITIES FOR ROBOTIC USE CASE

EFS Entity	Description
Robot Intelligence App	EFS application in charge of controlling and guiding the robot towards the point of delivery. This application provides the robot intelligence which is located inside the EFS platform.

	The application computes the optimum path for a robot to reach the point of delivery. It consumes data provided by the User Equipment (UE) location service and Wi-Fi information service.
Virtual Wi-Fi Access Point	EFS function enabling infrastructure-to-robot communication which is essential for robot navigation. Commands to control the robot are sent over Wi-Fi connections managed by virtual APs, which allows seamless Wi-Fi connectivity for a roaming Wi-Fi client and avoids connection disruptions. This function is also employed to help robots establish Bluetooth D2D communication for accurate movement synchronization.
Wi-Fi Information service	EFS service which provides Wi-Fi network information for each connected client (e.g., a robot) data regarding: the signal level; transmission and reception bit rates; number of retransmission and packet losses at data link level; and number of successfully transmitted/received bytes and packets.
UE location	EFS service which provides the UE position consumed by the robot intelligence application to perform the route computation. Robot location can be obtained through different techniques, such as employing LiDAR or iBeacons technology.

4.1.2 Use-case specific implementations and experimental verification

Figure 4-3, presents the implemented Edge/Fog robotics system that comprises two separates but interacting subsystems, i.e. the robotic system (shown in blue) and the EFS (shown in red).

The robotic system was implemented using the most widespread robotic framework, i.e. Robot Operating System (ROS) [22], which provides a meta-operating environment for developing and testing multi-vendor robotics software. In ROS, each software component is called ROS node. ROS also provides a publish-subscribe messaging framework (i.e. TCPROS) via a specific node, namely the ROS master node. By connecting to the ROS master, ROS nodes can register and locate each other. Once registered, nodes can exchange data via configurable topics in a peer-to-peer fashion. The robotics subsystem was implemented as various ROS components distributed across the robot and the EFS. The robot was equipped with motored-wheels and odometry sensors (Odometry is the use of data from motion sensors to estimate changes in position over time. E.g., motor encoders). ROS components running on the robots are essentially drivers that are in charge of: reading data from the sensors (e.g., odometry); sending the readings to the EFS; and executing the driving instructions received from the robot intelligence application. The robot intelligence application acts as a ROS master and it is also in charge of driving the robot based on the available information. The communication between the robot and the robot intelligence crosses over a Wi-Fi link and the wired network connecting to the EFS. A wireless information service is available locally at the EFS and is consumed by the robot intelligence application.

The implemented wireless information service provides Wi-Fi context regarding the clients connected to the system. The Wi-Fi network information service provides for each connected client (e.g., a robot) data regarding: the signal level; transmission and reception bit rates; number of retransmission and packet losses at data link level; and number of successfully transmitted/received bytes and packets. Additionally, the following link layer configuration(s) is provided: wireless channel; beacon interval; preamble and slot time (i.e., short/long); QoS support; and authorization/authentication status. The Wi-Fi information is published in JSON format to an MQTT-based EFS service platform.

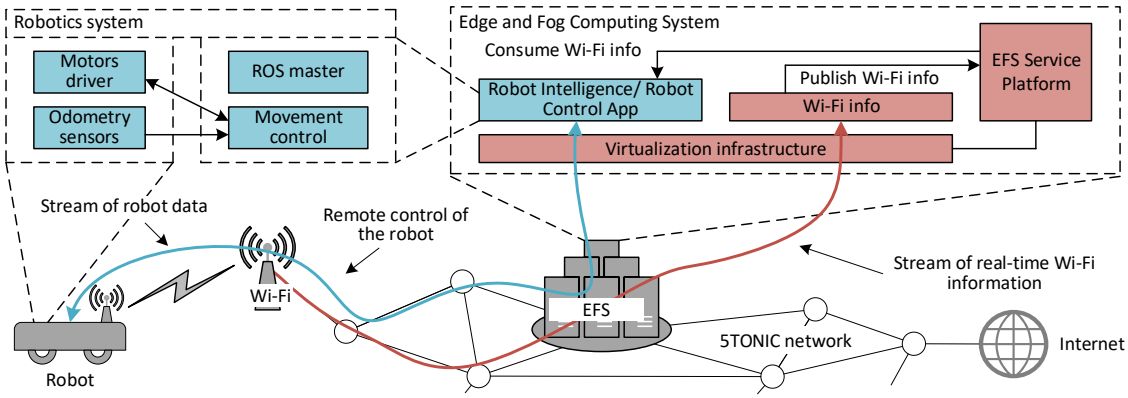


FIGURE 4-3: ROBOTICS LOGICAL SYSTEM

Experimental Setup

To evaluate the Fog-assisted robotics scenario, we built an experimental[1] environment in the 5TONIC laboratory comprising all the components shown in Figure 4-3. The goal of the experiment was to show how the Fog/Edge controlled robotics paradigm improves current Cloud robotics techniques. For the mobile robot, we used the ROS-compatible Kobuki robotics platform. The mobile robot maximum speed was 0.75 m/s, while its minimum speed was 0.1 m/s. The sampling frequency for reading the odometry sensor data from the robot's wheels was 16.6 Hz (i.e., odometry sensor data is refreshed every 60 ms). When driving at full speed (0.75 m/s), the robot covers a distance of 4.5 cm in 60 ms. This results in a precision of 4.5 cm in the robot driving at full speed since odometry sensor data cannot be updated with a frequency higher than 16.6 Hz. In the case of minimum speed (0.1 ms), the precision is 0.6 cm. It is worth highlighting that the sampling frequency value is limited by our robot's hardware. Different robotics platforms may offer higher sampling frequency and consequently better precision. The mobile robot is controlled in a closed loop by the robot intelligence application. The closed loop starts with the robot intelligence (running in the EFS) sending movement commands to the motor drivers (running on the robot) using ROS messages, published to a specific topic devoted to movement commands. The movement command consists of a tuple (speed, distance), where the speed parameter represents the velocity that the robot should maintain while driving, and the distance parameter represents the distance that should be reached upon receiving the movement command. Therefore, the distance parameter represents the movement granularity instead of the final driving destination. Upon receiving a movement parameter through the wireless link, the motor driver initiates the movement in the robot's wheels. The movement is uninterrupted for a length equal to the received distance parameter with constant velocity equal to the received speed parameter. The loop is then closed by the robot continuously sending-back the odometry sensor data to the robot intelligence application in the EFS. The robot intelligence analyses and combines the odometry data together with the Wi-Fi context information to generate a new (speed, distance) tuple, which will serve as input to the next turn of the closed loop.

All iterations of the experiment were performed in a closed and straight hallway (3m wide, 30m long) at 5TONIC laboratory (Figure 4-4). Each experiment consisted of the robot intelligence driving the robot on a straight line for 15m. The starting position of the robot was placed in the middle of the hallway approximately 7 meters away from the Wi-Fi AP having a thin office wall (approximately 15 cm) separating the two. Then, the robot accelerates from the starting position

[1] It is worth noting that the experimental setup considered in WP2 employs a single robot (i.e. adaptive robot control algorithm leveraging Wi-Fi information service) while the experimental setup considered in WP3 employs two robots (i.e. OCS triggering a network assisted D2D connection to reduce the latency of control messages between the robots) [41].

to the target velocity (e.g., min, max, etc.) and it drives in accordance with the closed-loop mechanism. After having travelled for 15 m, the robot stops. During the driving, an additional thicker wall (approximately 25 – 30 cm) separates the robot from the Wi-Fi AP. At the end of the driving, the robot is approximately 22 m away from the Wi-Fi AP.

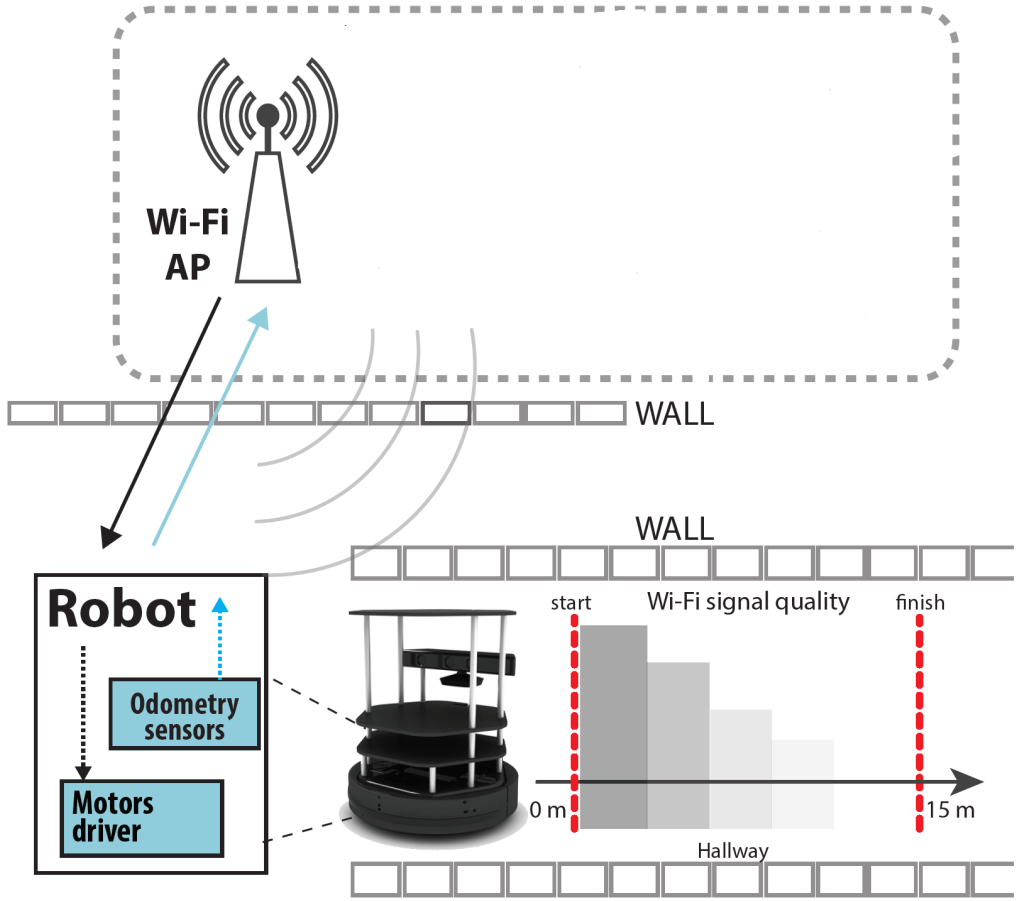


FIGURE 4-4: FLOOR PLAN AND ROBOT ROUTE

The experiment also designed and implemented a control algorithm (Figure 4-5) which is able to adapt the robot driving speed based on the Wi-Fi information service. The aim of the algorithm was to obtain a displacement accuracy similar to the one obtained while driving at the lowest speed, while reaching the target destination faster. Through this algorithm we showcased the benefits of consuming context information to control the robot, nonetheless, we acknowledge that more advanced and optimal algorithms than the one proposed can be eventually be designed. On the one hand, during the experiment, we collected the information of the Wi-Fi signal every 10 ms. We observed that the Wi-Fi signal level presents significant oscillations in case of averaging it over a short time window (e.g., 50 ms). That is, two subsequent average measurements may report considerably different Wi-Fi signal levels. On the other hand, if we take a longer time window (e.g., 500 ms), the oscillations between subsequent average measurements were substantially reduced and the Wi-Fi signal varied in a smoother way. Based on this finding, the control algorithm used the Wi-Fi signal level obtained by averaging it over a fixed time frame. Given the robot's speed bound between 0.1 m/s and 0.75 m/s, a time frame of 500 ms was considered. The computed Wi-Fi signal was then combined with the robot's odometry sensor data for adapting the robot's speed. Figure 4-5 shows the pseudo-code of the control algorithm. The robot intelligence, extracts in real-time the current signal level from the Wi-Fi EFS information

service, stores it in a circular buffer and computes the moving average of the Wi-Fi signal level. For each movement command, the adaptive speed and the adaptive distance are re-calculated. We observed that packet retransmissions and failures start increasing for signal values below -71 dBm, hitting their maximum between -79 and -81 dBm. Based on this observation, the control algorithm adapts the driving robot's speed to the maximum (0.75 m/s) for an average Wi-Fi signal level higher than -71 dBm or to the minimum (0.1 m/s) for an average Wi-Fi signal level equal or lower than -81 dBm. Between -71 dBm and -81 dBm, the control algorithm linearly adapts the robot's speed to the Wi-Fi signal level (e.g., 0.425 m/s with -76 dBm).

```
1: procedure ComputeRobotSpeed
2:     info ← GetCurrentWiFiInfo();
3:     buffer ← buffer.removeOldestWiFiInfo();
4:     buffer ← buffer.add(info);
5:     signalLevel ← buffer.average();
6:     if signalLevel > −71 dBm then speed ← 0.75;
7:     else if signalLevel < −81 dBm then speed ← 0.1;
8:     else speed ← (signalLevel+81 dBm)/ 10 dBm + 0.1;
```

FIGURE 4-5: ADAPTIVE SPEED CONTROL ALGORITHM

Experimental Results

This section evaluates the adaptive speed control algorithm and compares it with scenarios not making use of any context information. The following three scenarios are evaluated.

- The robot drives at minimum speed (0.1 m/s).
- The robot drives at maximum speed (0.75 m/s).
- The robot uses our control algorithm to drive at adaptive speed.

We performed 10 experiments for each scenario (minimum speed, maximum speed, adaptive speed). In addition to the Wi-Fi information we recorded the odometry sensor data directly in the robot itself. This is because the data from the odometry sensors is not timestamped and sending it over the Wi-Fi channel would not be suitable for measuring the speed and acceleration experienced by the robot (due to risk of transmission failures over Wi-Fi). The measured data was aggregated and analysed to produce the results on Figure 4-6. Figure 4-6 has four different graphs; on each graph the x-axis is the distance travelled during the experiment, from the start (0 m) to the end (15 m). The first subgraph from the top presents the Wi-Fi signal level (y-axis on the left) and the transmission errors over the robot driving path (y-axis on the right). It was noticed that there is a significant decay on the Wi-Fi signal quality in the last 5 meters of the driving path reflected by an exponential increase of the transmission errors. The remaining three graphs of Figure 4-6 present, for each evaluated scenario: the speed, the acceleration and the driving time as measured via the odometry sensor data. Despite the acceleration and the speed having different units (m/s and m/s^2, respectively), they share the same y-axis on the left since they present the same range of values. The y-axis on the right represents the elapsed driving time since the start of the experiment run.

In the minimum speed experiment, the robot speed was set constant to 0.1 m/s from the start to the end. Similarly, the acceleration presents a constant value in the order of few cm/s^2. Driving at such low speed results in a smooth run that is not affected by the degradation of the Wi-Fi channel in the last segment of the path, since the slowness of the movement allows more time to recover from possible transmission errors and further retransmissions. As a drawback, the robot requires approximately 160 seconds to complete each experiment run.

As expected, the maximum speed experiment is the one requiring less time (approximately 27 seconds). The impact of the decreasing Wi-Fi signal quality can be seen in the acceleration curve (notably in the last 5 meters of the path) where the acceleration fluctuates due to increased packet delay and consequently a delayed reaction, resulting in a stop-drive effect of frequent braking and spurring acceleration to full-speed. As a consequence, to the effect of the stop-drive behaviour, the driving direction is deviating from the straight driving path.

Finally, the bottom graph shows the motion behaviour for the experiment using the proposed adaptive speed control algorithm. A first observation is that the acceleration and deceleration in this case was smoother. At start, the robot accelerates to full speed, since the received signal level is in the safe zone above -71 dBm. After crossing the -71dBm threshold, the robot speed was linearly reduced following the decrease of the Wi-Fi signal strength, reaching the end of the path driving at minimum velocity. Regarding the driving time, the robot reaches the finish line approximately 10 seconds later than in the maximum speed experiment. Nonetheless, it is still approximately 120 seconds faster than the minimum speed experiment while performing a smooth ride.

As concluding remarks, the results show that there was a trade-off between speed and smooth movement of the robot. By adapting the velocity of the robot with information on the quality of the Wi-Fi channel, the robot was able to move with maximum speed where the Wi-Fi signal channel was good and smoothly lowered the speed in the areas of weak Wi-Fi signal coverage, thus cancelling any stop-drive effect.

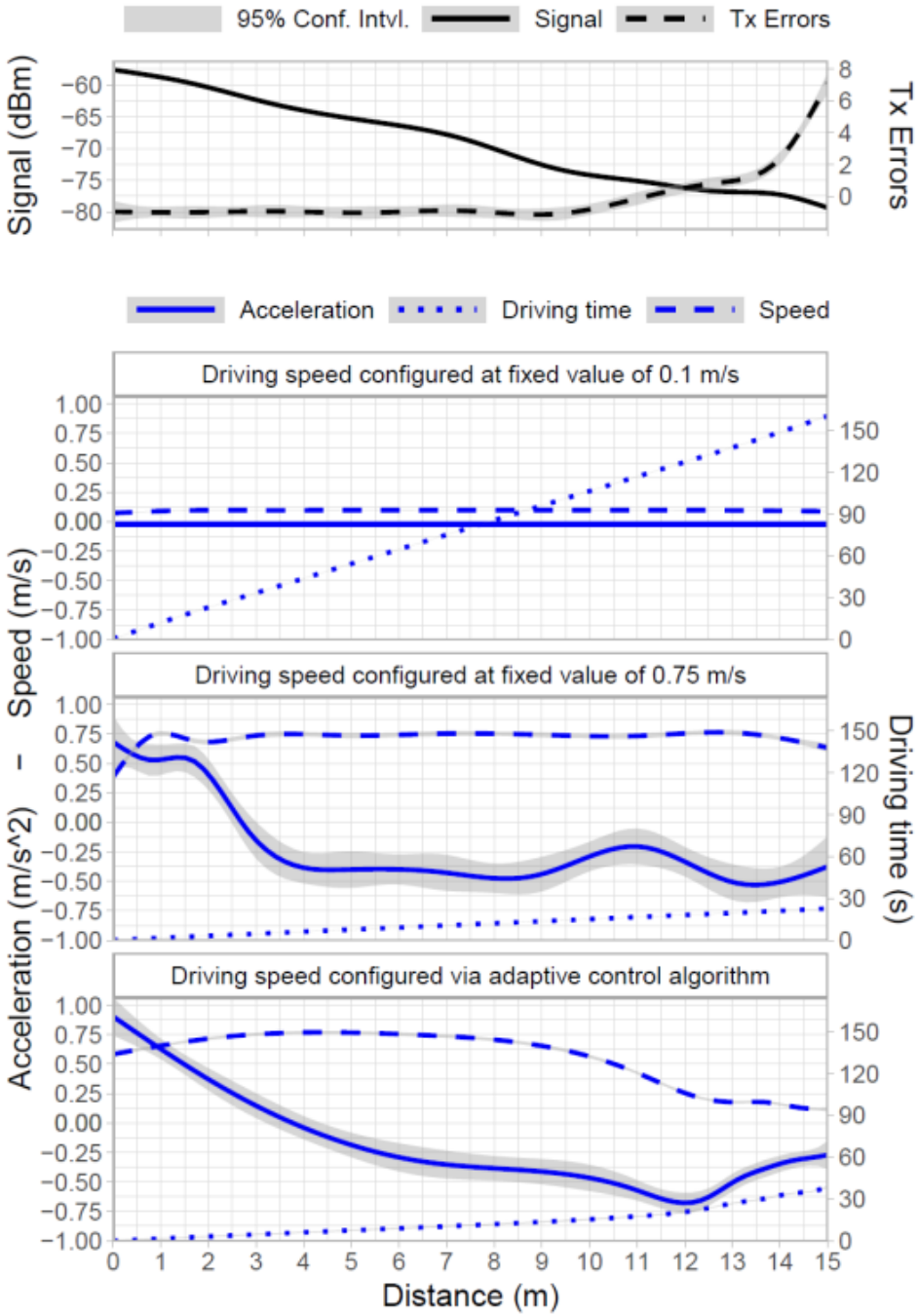


FIGURE 4-6: SPEED, ACCELERATION, AND DRIVING TIME

4.1.3 Conclusions and future directions

This use case highlights the opportunities offered by Edge/Fog computing. One of the key differentiating features of Edge/Fog computing is the possibility for applications running at the Edge to consume context information, e.g., about the network. This can be used to optimize the robotics systems operations in ways otherwise impossible in the Cloud robotics framework. We have designed an Edge/Fog assisted robotics system blending together the Robot Operating System (ROS) that offers a common development framework for robotics applications and the 5G-CORAL EFS platform.

We performed a set of experiments to characterize the relation between the robot control delay and the Wi-Fi signal strength. The resulting characterization was used as a baseline for designing, implementing and experimentally evaluating a control algorithm that consumes context information about the Wi-Fi signal and adapts the robot's speed for a smoother driving. Our experimental results showed that adapting the robot's speed based on the Wi-Fi signal provided by the EFS information service can effectively produce a smoother driving at high speeds. This improvement allows the robot to operate faster compared to the case of ignoring the context information from the network.

The following future work is therefore expected: enhancement of the robotics connectivity from Wi-Fi to 5G along with the corresponding 5G radio network information service; designing of more advanced control algorithms and extending the robotic use case to consider drones.

4.2 Virtual Reality (VR)

Augmented Reality (AR), Virtual Reality (VR) and Mixed Reality (MR) have become prominent technologies within the wide spectrum of video applications available in different markets such as cinema, gaming, education, healthcare, sports and advertisement. While VR offers an immersive virtual user experience and AR augments a real or virtual environment by adding elements for interaction with the user, MR provides a reality-virtuality continuum consisting of different combinations and variations of real and virtual objects co-existing in the same environment. For instance, a 360° video streaming service can be thought of as an MR application, since users can panoramically watch a real video scene by seamlessly adapting the Field of View (FoV) or viewport, i.e., the fraction of omnidirectional view of the scene, as the viewing orientation changes.

The 360° video streaming service is classified as an enhanced Mobile Broad-Band (eMBB) service due to the significantly high bandwidth and constant data rate requirements. For instance, considering High Efficiency Video Coding (HEVC) compression, a live video service with 60 frames-per-second and 8K resolution requires 361 Mbps in order to ensure smooth content play.

This use case showcases the viewport adaptive 360° video streaming technology delivered over the 5G-CORAL solution[2] (EFS and OCS). The viewport adaptive streaming technology reduces the bandwidth required to deliver 360° video while preserving the user's quality of experience, by leveraging the user's viewing orientation, i.e. the portion of the 360° video being watched by the user is delivered in high quality resolution while the rest is delivered in low quality resolution.

The 5G-CORAL solution decomposes the End-to-End (E2E) viewport adaptive 360° video streaming service into microservices hosted on the appropriate Edge/Fog device based on their computational requirements. Additionally, the 5G-CORAL solution offers seamless orchestration

[2] The 5G-CORAL solution targets a holistic Edge/Fog solution with particular focus on integrating the constrained mobile and volatile Edge/Fog devices that are mostly present in the RAN.

and control of the E2E viewport adaptive 360° video streaming service across three tiers of computing nodes (Low, medium and high ends). The benefits of the 5G-CORAL solution include:

- Enhanced flexibility in deploying and managing heterogeneous resources in multi-tier system architecture
- Zero-touch configuration and instantiation of the VR end-to-end service through the distributed orchestration and control delivered by the OCS
- Energy-efficient deployment solutions by offloading resource-demanding computing tasks from terminals

4.2.1 Refined EFS design and functional validation

The EFS entities involved in the VR use case, as well as their interconnection, are illustrated in Figure 4-7. Table 4-2 presents a description of EFS entities depicted in D2.1[1].

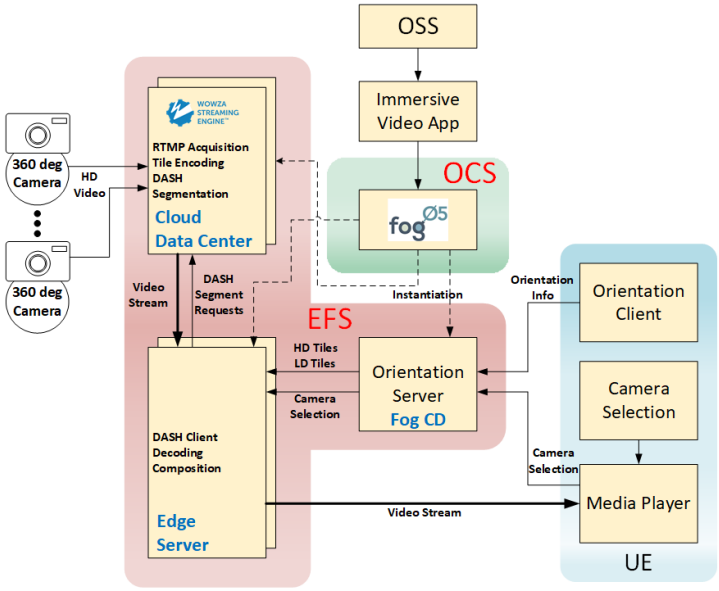


FIGURE 4-7: EFS ENTITIES INTERCONNECTION FOR THE VR USE CASE

TABLE 4-2: SUMMARY OF EFS ENTITIES FOR VR USE CASE

EFS Entity	Description
Real-Time Messaging Protocol (RTMP) acquisition	EFS application responsible for performing the RTMP acquisition enabling persistent connections and low-latency communications. It consumes data provided by the camera and sends the output data stream to the tile encoding application.
Tile encoding	EFS application to perform the tiled 360 video encoding. It processes the data stream coming from the RTMP acquisition application and provides the DASH segmentation module with the tiled encoded data stream.
DASH segmentation	EFS application in charge of segmenting the data stream encoded by the tile encoding application through DASH, which is consumed by the DASH client application

DASH client	EFS application to reassemble DASH segments sent by the DASH segmentation application. The output data is then sent to the decoding application. This application also uses the UE orientation information provided by the UE orientation service, which provides the user's view angle.
Decoding	EFS application performing the decoding of tiled video streams sent by the DASH client. The decoded video stream is then delivered to the composition EFS application.
Composition	EFS application responsible for re-composing tiled video streams into 360 video frame at the client side. This component receives tiled video streams decoded by the decoding EFS function.
UE orientation	EFS function responsible for selecting which tile has to be sent to the UE based on the orientation information provided by the orientation client. The selected tile is communicated to the DASH client.

4.2.2 Use-case specific implementations and experimental verification

Figure 4-7 and Figure 4-8, present the logical and physical implementation of the VR E2E viewport adaptive 360° video streaming using the distributed edge and fog computing platform developed in 5G-CORAL. We consider the viewport adaptation technique based on adaptive tile-encoding streaming, where the 360° video is partitioned in small tiles, which are independently encoded and transmitted according to the viewing orientation, and then stitched together to recompose the 360° frame. To compensate for the extra computational complexity needed to continuously adapt the video stream quality, we spread the computing tasks across three different tiers, namely, fog, edge and cloud, according to their respective computational and latency requirements. The novel contributions can be outlined as follows:

- Different computing processes, including DASH coding/decoding and video frame composition, are distributed across three different tiers, i.e., cloud, edge and fog, thus increasing system scalability and reducing the latency.
- Specific GPU-intensive tasks are offloaded from the client. This results in a reduced terminal complexity as well as improved interoperability, as some of the protocols and video codecs employed may not be supported by legacy devices.
- A novel orchestration and control framework, i.e., 5G-CORAL OCS, is adopted, which enables management, monitoring and orchestration of diverse resources spanning across the three computing tiers, thus facilitating the deployment, operation and lifecycle maintenance of the 360° video streaming service.

The End-to-End 360° video streaming service consists of four major entities, namely: video source, EFS, OCS, and User Equipment (UE). In the following, we describe their respective roles and the information exchanged between the interfaces.

- Video source: two or more 360° cameras, each capturing separate event(s) happening in different area(s) of the interest, stream live video content to a cloud data centre, that represents the EFS entry point. This is implemented by connecting the cameras to a streaming engine, e.g. Wowza Streaming Engine[23], and establishing a Real Time

Multimedia Protocol (RTMP) live or Real Time Streaming Protocol (RTSP) stream session, that ensures a TCP-based persistent connection and low-latency communication.

- EFS: built upon three different compute tiers, i.e. cloud data centre, hosting powerful processing units and located on cloud provider premises; edge server, located closer to the end user equipment (UE) and providing limited compute capabilities; and fog computing devices (CD), resource-constrained devices operating in the UE proximity. The EFS hosts all the essential functions, applications and services to deliver the live video stream.
    - After RTMP acquisition, the tile-encoding app running on the data centre performs tile-based High-Efficiency Video Coding (HEVC) encoding, thus partitioning each video frame into three tiles (3 x 1 uniform tiling), each capturing a 120° viewing angle, which are encoded at either high or low-quality resolution.
    - Next, the DASH segmentation app packetizes the bitstream data into multiple chunks, i.e., DASH segments, which are requested by the DASH client running on the edge server. Furthermore, the decoding and composition app decodes the tiled video streams and composes the 360° video frame for the UE, respectively.
    - A key EFS component of our solution is the fog CDs deployed whose main task is to gather the UE viewing orientation and convey it, via HTTP REST API, to the DASH client hosted at the edge server. The DASH client utilises the UE viewing orientation to determine the portion of the 360° video that must be delivered at high quality.
    - The DASH client is able to quickly recompose the video frame by decoding the correct tiles, depending on the orientation information sent by the fog CDs. It is worth noting that all the EFS software processes are implemented as native applications.
- UE: the user terminal consists of three components, namely, a media player, a camera selector and the orientation client. The first two elements are managed by the user, whereas the latter runs in the background and forwards information on the user orientation to the orientation service located on the fog CD.

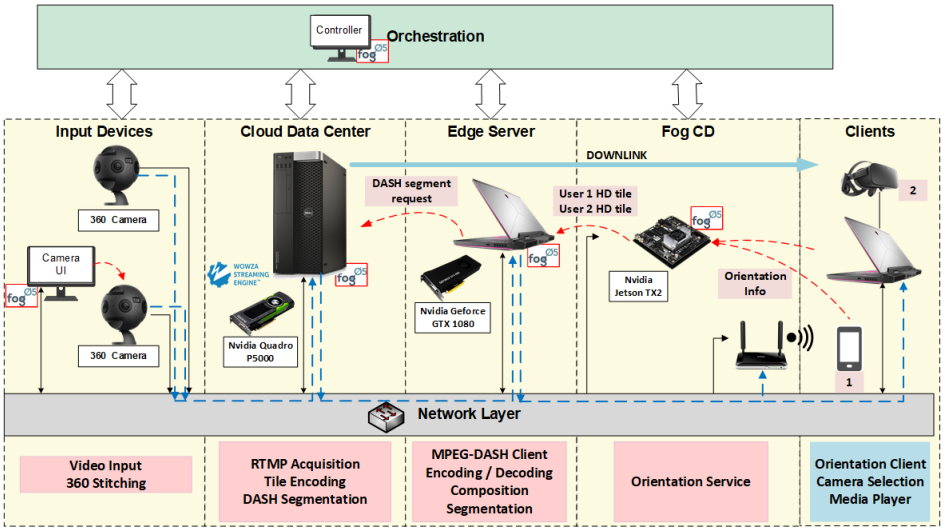


FIGURE 4-8: VR END-TO-END PHYSICAL IMPLEMENTATION BUILDING BLOCKS

Figure 4-8 presents the physical implementation that consists of a multi-tier computing, storage and networking platform capable of conveying multimedia traffic generated by two or more Insta360 Pro cameras to fixed and mobile clients. Each component is equipped with a Gigabit Ethernet network adapter and is connected to the network layer represented by a Gigabit Ethernet switch,

whereas the phone terminal is connected to a Wi-Fi IEEE 802.11ac access point. The tasks performed by each layer are listed at the bottom of Figure 4-8. The top layer hosts the orchestration component, which deploys and manages all the entities running the Fog05 agent. This is achieved by using a laptop, raspberry pi, or any computer equipped with a screen and running the Fog05 agent, in order for the operator to execute scripts and verify the successful function onboarding.

As previously described, the 360° live video streaming is initiated by the cameras connected to Wowza Streaming Engine hosted by the cloud data centre. To trigger the RTMP session, a computer laptop running the Fog05 agent is also plugged to the camera, thus allowing automatic session instantiation and termination. The Insta360 Pro camera is equipped with 6 fisheye lenses and can perform real-time stitching of 4K video sequences. Next, the data centre handles the tile-based HEVC encoding by leveraging computing resources provided by the Nvidia Quadro P5000 GPU. Our data centre consists of a Dell Precision Tower 7810 equipped with an Intel Xeon E5-2670 v4 2.30 GHz CPU, 64 GB RAM and 1 TB HDD storage. In addition, this machine runs a DASH server compliant with the latest MPEG immersive Omnidirectional Media Format (OMAF) standard.

The DASH segments are then received by the edge machine represented by a Dell Alienware 17 laptop equipped with an Intel Core i7-8750H CPU, 16 GB RAM, 128 GB SSD and an Nvidia GeForce GTX 1080 graphic card, able to execute complex tasks such as tile decoding and video frame composition. Furthermore, the edge server exploits the orientation information supplied by the fog node. To this end, we use an Nvidia Jetson TX2 development board, consisting of a Jetson TX2 module, which embeds a powerful GPU and two ARM CPUs. It is worth pointing out that although this platform features a high-performance 256-CUDA core graphic processor, we solely rely on the available CPU power, as the user orientation tracking doesn't require hardware acceleration.

The orientation service computes in real-time the video stream tiles that must be encoded in high definition by processing the orientation info, i.e., yaw, roll and pitch, periodically reported by the terminals according to the orientation report rate system parameter. Also, a count-down timer is associated with each reported tile: the timer can be configured by setting the orientation decay period system parameter and is periodically decremented and reset whenever a new matching orientation is reported.

Finally, the edge server transmits the optimized DASH video stream to the clients. Specifically, we consider two types of video terminals, i.e., a Samsung S9+ Android-based mobile phone and an Oculus Rift VR headset connected to a computer laptop. Additionally, we developed an Android application and an Oculus Rift application both of which feature a user media player capable of reporting the orientation info as well as selecting one of the video streams according to the user choice.

Experimental Setup

The following experiment was conducted by running a 5-Minute pre-recorded 360 video sequence stored on the data centre that feeds the Wowza streaming engine. The results were generated by executing the same experiment for 10 repetitions and by recording the metrics every second. A summary of the system parameters employed is presented in Table 4-3. The experiment only considered a stationary phone terminal and randomly changed its orientation in each repetition.

TABLE 4-3: SYSTEM PARAMETERS

System Parameter	Value
Video Resolution	4K (3840x2160)
Video Bitrate	15 Mbps
Video Frame Rate	30 fps
Orientation Report Rate	10 Hz
Orientation Decay Period	250 ms

To show the impact of the video processing load on the data centre, we retrieved the GPU load, the GPU power consumption and the memory usage by using the GPU-Z tool that supports NVIDIA video cards. The GPU-Z tool measures the following three metrics.

- The first metric gives an estimation of how active the GPU is in a given interval.
- The second metric indicates the power in Watts consumed by the GPU,
- The third metric reports the memory used.

During the experiment we recorded all the GPU-Z metrics for three different configurations, namely: idle, i.e., the video streaming service is inactive; split mode, where all the computing tasks, except for the orientation service running on the fog CD, are distributed between the data center and the edge server; no split mode, where all the tasks are executed by the data center. Additionally, we obtained the bandwidth consumed in downlink by the terminal. Moreover, we considered three different streaming modes in order to highlight the benefits of the adaptive tile encoding strategy. The information on the bandwidth is obtained by using a network monitoring tool called Wireshark, which allows to monitor the bandwidth consumed by distinct applications on the same machine.

Experimental Results

Figure 4-9 shows the GPU load, power consumption and memory used by the cloud data centre measured through GPU-Z. As expected, the GPU computing load distribution between the data centre and the edge server results approximately in a 7% reduction of the GPU load, which translates into more processing capacity available for other services running in the data centre. Furthermore, the split mode leads to a small reduction of the power consumed by the data centre. Specifically, up to 2 Watts can be saved by offloading some GPU processing onto the edge server. This also means that most of data centre power is spent executing the tile encoding and the DASH segmentation together with the Wowza streaming engine. Finally, it is worth pointing out that our approach helps to reduce the data centre memory occupancy, with a memory saving equal to 600 MB in comparison with the no split mode.

The Empirical CDF of the downlink data rate observed at terminal side is shown in Figure 4-10. We note that the adaptive tile-encoding requires roughly a bandwidth equal to 21 Mbps, whereas a non-optimized approach encoding all the tiles with maximum quality leads to a bandwidth consumption equal to 31 Mbps, thus to a 33% increment. Obviously, this is due to the larger size of the DASH segment requested by the local edge server, which results in a higher bandwidth consumption over the link between the remote edge server and the client. Also, by employing a fully low-quality encoding, the required data rate decreases to only 2 Mbps, though the QoE is heavily affected by the lack of high-quality tile being watched by the user.

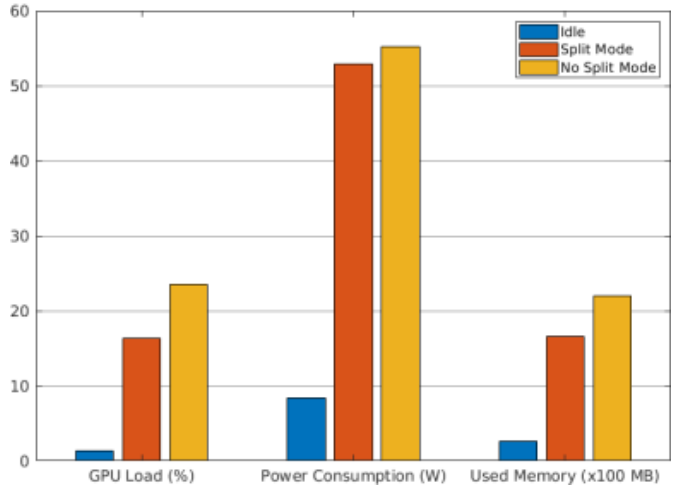


FIGURE 4-9: GPU LOAD, POWER CONSUMPTION AND MEMORY USAGE ON THE CLOUD DATA CENTRE.

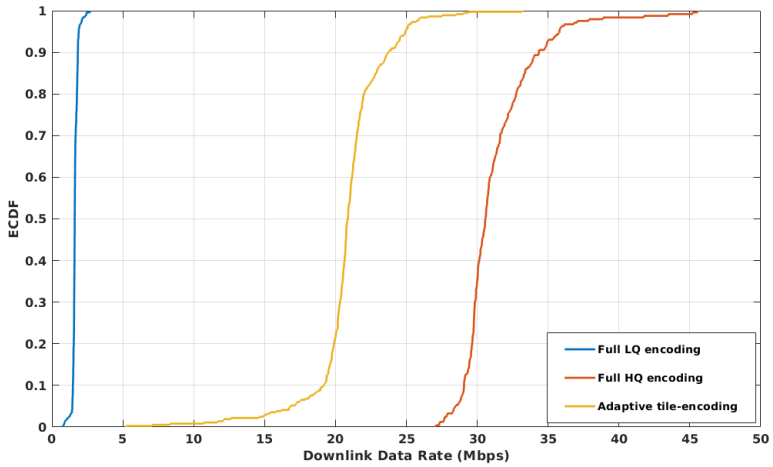


FIGURE 4-10: ECDF OF THE DOWNLINK DATA RATE

4.2.3 Conclusions and future directions

The 5G-CORAL solution decomposes the end-to-end 360° video streaming service into micro-services which are then distributed across three computing tiers, namely cloud, edge, and fog, in order of proximity to the end user client. The solution uses an adaptive viewport technique whereby only the field of view capturing the end user client orientation is delivered in high quality whereas the rest of the 360° scene is delivered in low quality, yielding good bandwidth saving. In addition, all the three compute tiers, composed of heterogeneous computing resources, are orchestrated and controlled using a unified orchestration and control system (OCS) based on Fog05.

Performance evaluation has been conducted using physical testbed using real hardware equipment. The evaluation/experimentation measured metrics such as the GPU load, power consumption, memory usage, downlink and uplink data rates. These measurements clearly demonstrated the benefits of the proposed solution compared to a conventional approach where the 360° video streaming service is executed out of the Cloud.

Measurement results show that our approach can alleviate the footprint of the 360° video delivery service on a cloud data centre by reducing the GPU load, the consumed power and the memory usage. Furthermore, we evaluated the bandwidth needed in uplink and downlink by the edge

server to deliver the video content and compared the adaptive tile-encoding approach with two non-optimized solutions, where the all the tiles are encoded at low or high quality. In particular, we observed a bandwidth reduction equal to 33% and 5%, respectively in downlink and uplink, when the adaptive tile-encoding is employed with respect to the full high-quality approach.

While the measurements reported considered only a single user scenario, it is anticipated that with our distributed solution more significant bandwidth saving gains can be achieved especially in dense environments where several users share the same field of view. This is thanks to the potential of aggregation across multiple users with same orientation angles.

The performance evaluation with multiple users is planned for future work. Furthermore, as a next step, we plan on carrying the 360° video streaming out of cameras on-board moving devices, such as robots or drones, using 5G wireless connectivity to the fog and edge tiers. This will help obtain more insights on the latency measures and the deployment topology in these mobile scenarios.

4.3 Augmented Reality (AR)

Augmented Reality (AR) is a powerful technology which brings new quality to the way we perceive the surrounding world. The goal is to understand the video stream recorded by the camera of the user device and add digital content (image or animation) on top of it in order to augment the video end user is observing (e.g. from the phone's screen). In 5G-CORAL, we are aiming at providing a continuous indoor AR navigation experience for the clients at the shopping mall. The objective is to augment the user recorded video frames with a navigation arrow similar to the popular car navigation application. The user will see a guiding line grounded in the real world image displayed on his screen so that it will remind a real object, a pointer, to the desired destination as depicted in Figure 4-11. Moreover, user will be able to see shop promotions on their screen whenever he/she passes by the store. These special offer will enhance the shopping experience for the mall's client. It is important to highlight that utilization of multi-RAT architecture is very vital to enhance AR performance. Indeed, the utilization of different RAT information will reduce the end-to-end latency and provide a better user experience for the AR user. Where, the utilization of different RAT can help in collecting the context information to determine approximate user location. Then, an image recognition (IR) will require much smaller database size for estimating accurate user location. In addition, the processing time of AR computing will be reduced dramatically due to less image processing with smaller database.

FIGURE 4-11: AR LIVE NAVIGATION IN SHOPPING MALL

4.3.1 Refined EFS design and functional validation

Functions, services, and applications comprising on AR Navigation use case and the way they are interconnected is depicted in Figure 4-12. The IR Application processes input video frames sent by the UE and location data via iBeacon Localization Data, Service, and communicates with the IR Localization Data Service. After the IR processing, the application can determine in which zone of the shopping mall a given UE is located.

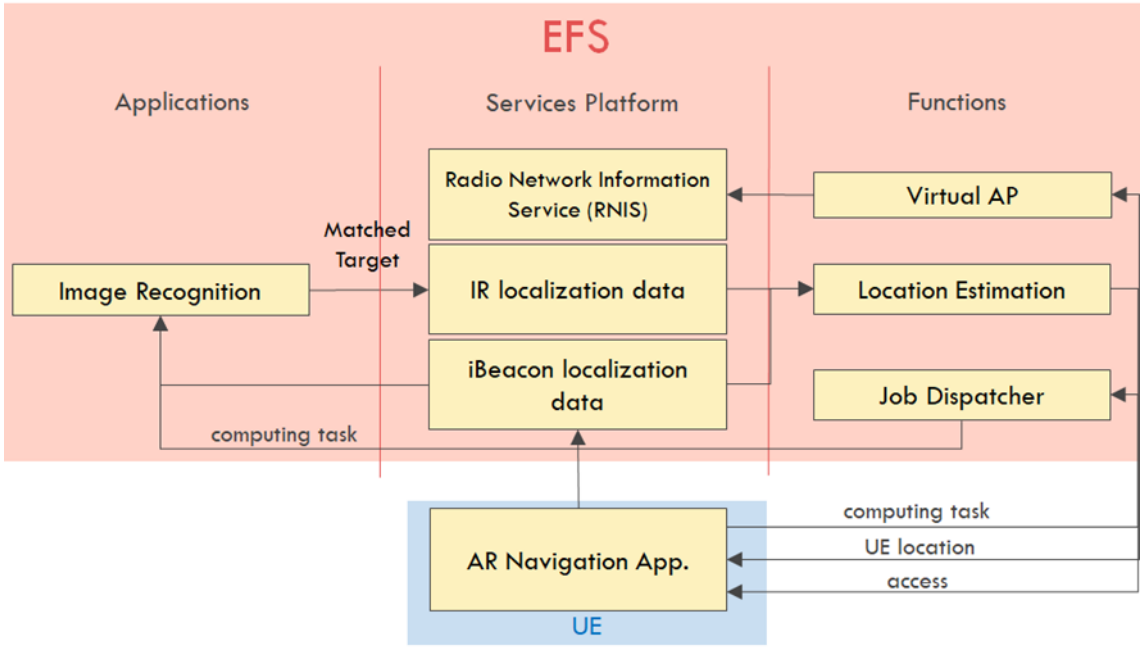


FIGURE 4-12: AR NAVIGATION EFS DESIGN

Three EFS service entities have been designed in the AR Navigation use case. The RNIS service exposes the received signal strength of a user towards an Access Point (i.e., Virtual AP Function).

The IR Localization Data Service consumes an output of the Image Recognition (IR) application and provides information about which area of the shopping mall where the UE is located. The iBeacon Localization Data Service conveys iBeacon ID and signal strength with respect to the UE receiving the beacon signal. This service can estimate the approximate location of the UE based on the vicinity of the UE to the iBeacon with assumption of iBeacon location known. The iBeacon localization data is provided by the UE and consumed by the Image Recognition application and the Location Estimation function.

Three EFS function entities have been specified in the AR Navigation use case. Virtual access point (vAP) is a network function that was designed to provide customized WiFi access service to a specific client. When the client moves, its assigned vAP moves along. Therefore, from the client perspective, it will still be connected to the same AP, yet in fact, it is connected to the same vAP but different physical AP. Localization estimation function estimates relative UE location (e.g. X, Y, Z coordinates) in the indoor environment. It combines location information from multiple localization sources including iBeacon localization data service, Image Recognition application, Phone's gyroscope data and possibly any other location information from other source (applications/functions). The Job dispatcher function dispatches the image recognition tasks among multiple computing substrates in order to balance the load among the fog nodes which are part of this application.

4.3.2 Use-case specific implementations and experimental verification

The environment described in the AR Navigation use case drastically decreases the need for the video frame to travel from user's phone all the way to the remote data centre. Fog Computing Devices (CD) host by shopping mall's Wi-Fi access points are deployed to which places proximate to the end user. Networked Fog nodes can offload computing capability of the remote data centre. The application finds a connection to the Image Recognition application (EFS application) deployed on the Fog CDs distributed around the shopping mall. Each Fog CD is coupled with a Wi-Fi access point controlled by the OCS. While iBeacon is used to broadcast messages, which help to infer the location of the mobile phone, the Wi-Fi AP allows for basic connectivity of the UE with the rest of the network.

Figure 4-13 and Figure 4-14 show the experimental environments with native AR application and AR container (LXD), both of them are using TX2 as FogCDs, respectively, in which execution flow and latency monitored during the experiment trials are showed. According to the experimental results in the environment with native AR application, the connection establishment latency, measured by Timers 1, 2 and 3, is 278ms. The experimental Connection establishment result in the environment with AR container is 316ms, which is similar to the result of native AR application.

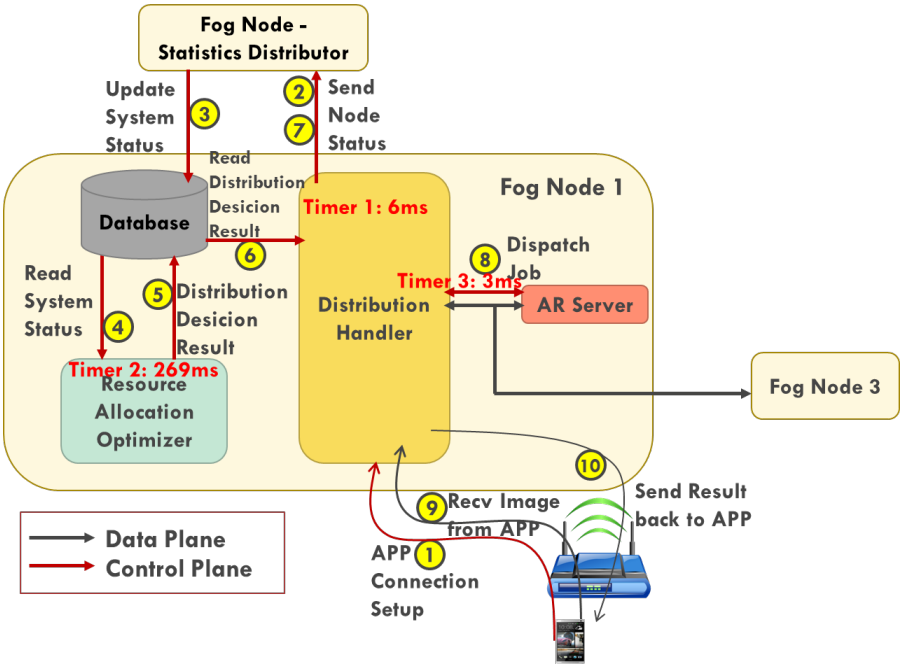


FIGURE 4-13: DISTRIBUTED AR – EXECUTION FLOW (NATIVE APPLICATION)

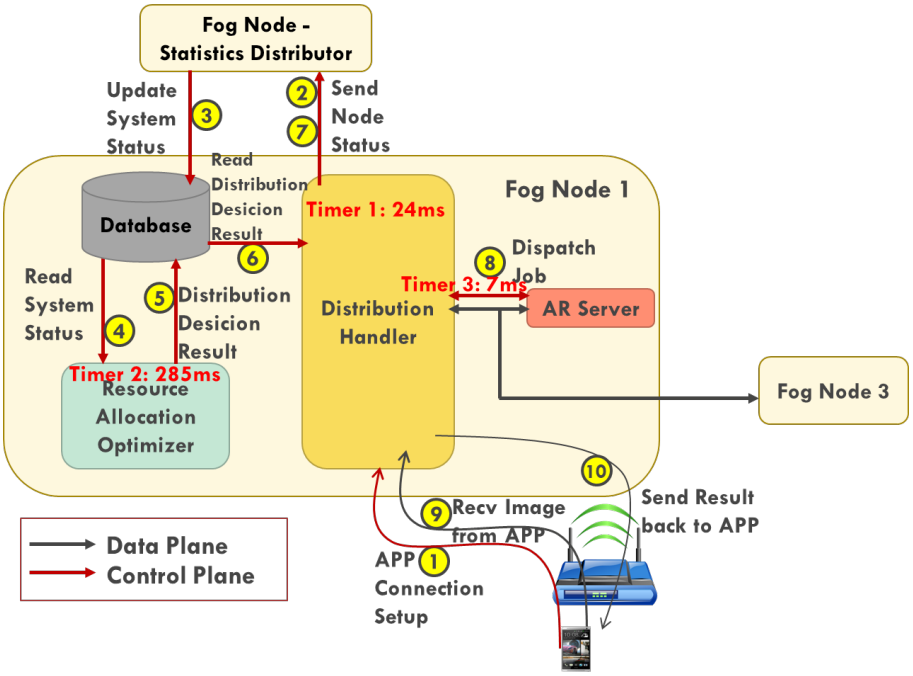


FIGURE 4-14: DISTRIBUTED AR – EXECUTION FLOW (LXD CONTAINER)

4.3.3 Conclusions and future directions

In an infrastructure supporting user mobility such as shopping mall use case, fog computing architecture has been adopted to realize AR navigation application where the communication delay is significantly reduced on such architecture. The experimental result shows us a proof-of-concept that the edge and fog computing architecture is possible to meet the requirements of AR navigation application. However, to make a considerable amount of shopping users satisfied simultaneously, the future direction in this use case is to design and develop a distributed computing

mechanism which is able to organize Fog CDs in proximity in a cooperative way so as to tackle and response timely bursts of AR navigation requests.

4.4 Multi-RAT IoT

In this 5G-CORAL use case, the main idea is to investigate the possibility to have one radio network infrastructure (instead of parallel network deployments) to serve multiple IoT RATs. The IoT baseband and upper layer functions are centralized and cloudified to an Edge Cloud environment. The main benefits are increasing network flexibility, reducing network cost, and increasing system scalability and future proofing.

4.4.1 Refined EFS design and functional validation

In this project, we provide a basic reference design of the Multi-RAT IoT use case to showcase the feasibility of implementing the three key EFS elements, i.e. EFS function, EFS service and EFS application, as briefly explained below, illustrated in Figure 4-15 and summarized in Table 4-4 in some more details.

- EFS function: IoT communication stack functions for various IoT RATs are implemented as software that can run on the Edge and they are virtualized as Docker containers, which are orchestrated using Kubernetes for life-cycle management, scaling, load balancing etc.
- EFS service: MQTT is used as an example of EFS platform design. In-phase and Quadrature components (IQ) service is published from the IoT communications to a MQTT broker.
- EFS application: As an example, the interference analyzer application subscribes to IQ services from a MQTT broker and uses the subscribed IQ samples for interference analysis.

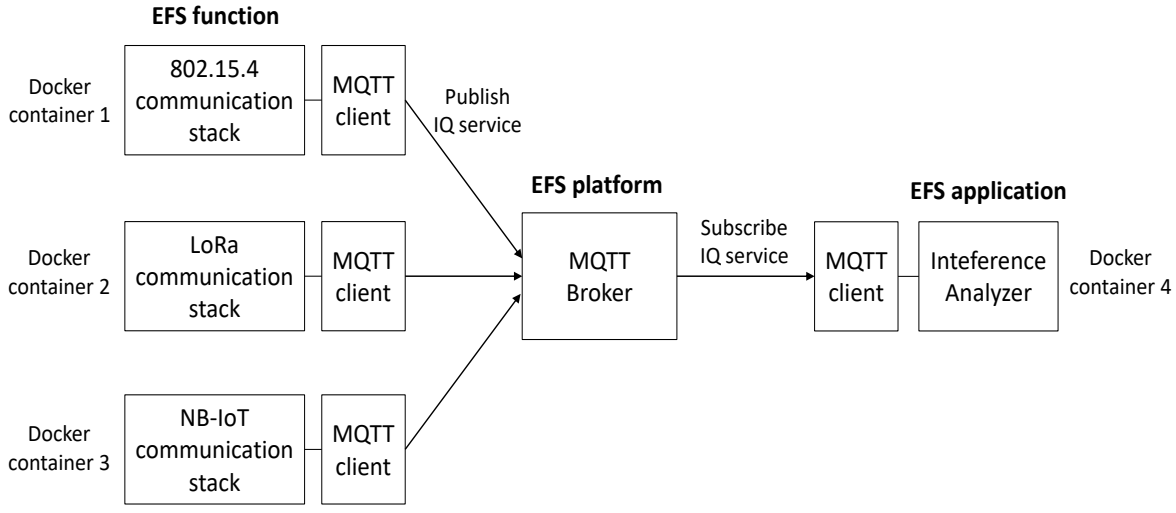


FIGURE 4-15: ILLUSTRATION OF REFINED EFS DESIGN FOR MULTI-RAT IOT USE CASE

TABLE 4-4: SUMMARY OF EFS ENTITIES FOR IOT MULTI-RAT USE CASE

EFS Entity	Description
IoT communication stacks	These are the **EFS functions** that implement various IoT communication stacks for different RATs, e.g. IEEE 802.15.4, LoRa, NB-IoT. Basically, the communication stack function softwarises and virtualize the communication protocol (including lower layers like L1/L2 and higher layers like L3) implementation on EFS. For user data, in uplink, each communication stack function demodulates the IQ samples received from radio heads. In downlink, it modulates the user data to IQ samples and send to radio head where the IQ samples are converted to radio signals and sent to the air interface.
MQTT clients and brokers	Use MQTT as the **EFS service platform**, as agreed in the consortium, mainly due to its simplicity and software maturity. In this use case, each IoT communication stack function is connected to a MQTT client to publish its services to other EFS functions or applications which subscribed the services. The MQTT brokers handles the data pub/sub mechanism following the MQTT protocol.
IQ service	This is an **EFS service** developed in this project. Basically, upon requests, radio head listens to its air interface and send IQ samples to the communication stack functions. The communication stack functions do some pre-processing and then send the IQ data via their MQTT client.
Interference analyzer	This is an **EFS application** developed in this project. Interference is a key problem in wireless communication. The current approach with transceivers embedded in access points can't give a good picture about interferences due to its limited processing resources in hardware. The purpose of this application is to provide a tool on the Edge which can utilize the Edge resources to analyse the interference situation and thus construct a more accurate picture based on interference statistics obtained from the IQ service data subscribed with MQTT.

The methodology for functional validation is to develop a PoC testbed being developed in both WP2 and WP4 in this project and perform functional and performance tests. At this time when writing this deliverable, the IoT communication stack functions supporting IEEE 802.15.4, LoRa and NB-IoT have been passed basic functional tests in lab environment. In addition, the functionality of IEEE 802.15.4 and NB-IoT implementations have been successfully demonstrated in two public events of EUCNC 2018 and ICT 2018, as well as in the midterm reviews in Taiwan and Vienna. The following list the functions that have been tested.

(1) IEEE 802.15.4: full-stack implementation supporting 3 frequency channels from PHY layer to application layer with bi-directional communications between the softwarised communication stack function in the Edge and commercial IoT devices (i.e. Zolertia firefly).
(2) LoRa: PHY and MAC layer implementation with bi-directional communications between the softwarised communication stack function in the Edge and commercial IoT devices (i.e. Pycom FiPy)
(3) NB-IoT: downlink PHY (NPSS, NPSCH) implementation with simplified upper layer implementation which supports sending signals and messages from the softwarised communication stack function in the Edge and a self-developed SDR-based NB-IoT receiver.

In this use case, the main research and development work is on the softwarization of the IoT communication stacks of multiple RATs. Other EFS elements of EFS service platform and EFS

applications are dependent to the IoT communication stack functions. They are more part of the testbed integration work in WP4. So, the functional tests on these parts will be done in the integration phase in WP4.

4.4.2 Use-case specific implementations and experimental verification

In the following, some key aspects and more details regarding the EFS design and implementation for the Multi-RAT IoT use case are presented. The performances of different aspects are evaluated and verified by experimental results with the developed PoC testbed.

4.4.2.1 RH-Edge interface

The interface between radio heads and the Edge needs to be efficiently designed to avoid overloading the transport network in between, as well as minimizing the latency. The following presents how we address this aspect and more details about our implementations to improve the interface efficiency. Although some aspects mainly consider two RATs of IEEE 802.15.4 and NB-IoT for PoC, the insights learned can be extended to other RATs.

In this use case, the physical layer processing is done at the Edge and the Radiohead is responsible for the configuration and management of the Software Defined Radio (SDR). The SDR converts the radio channel information into digital streams of In-Phase and Quadrature samples. These samples need to be transported to and from the Edge for the receive and transmit data flow chains respectively.

As the physical layer of different RATs are virtualized and the physical instantiation is determined by the OCS, the RH-Edge Interface should be a logical interface. Internet Protocol (IP) provides a mature and diverse collection of transport protocols over logical interfaces and hence is chosen for this use-case. The protocol specifications of the RATs define timing constraints for the transactions between the gateway and other nodes. In order to satisfy these requirements, the transport of samples over the RH-Edge interface should have low-latency. The throughput requirements of the transport protocol are determined by the type and the number of RAT instances using a single interface. For example, a single 802.15.4 instance requires a data rate of approximately 128 Mbps (note that this can be further compressed, as to be discussed later on about fronthaul compression regarding NB-IoT implementation. For example, the bit rate can be compressed by 4 times by using a more efficient data format.).

Considering these requirements, the base implementation used ZeroMQ as the transport protocol. ZeroMQ uses TCP for the transport of samples over the RH-Edge interface. TCP provides reliability with guaranteed delivery of packets with flow-control and error correction mechanisms. But these methods increase the transport overhead resulting in higher latency. Considering the wide adoption of User Datagram Protocol (UDP) as the transport protocol for video and audio streaming applications which have similar transport requirements as our use case, we developed a UDP-based transport as the default RH-Edge transport. The UDP based source and sink blocks for the GNU Radio are implemented which allow us to transport GNU Radio specific metadata along with the data payload. This helps in communicating status and control flags between the Radiohead and Edge dataflow chains.

Experimental results

To test the performance of difference transports between the edge and Radiohead components, we set up a testbed with two Intel Hades Canyon NUCs running the edge and radio head flowgraphs. The two NUCs are connected by a 1 Gbps Ethernet link. The radio head is connected to the USRP for communicating with a Zolertia Firefly node. The radio head uses the Squelch filter for RX packet filtering. We use ping command over a TUN interface for benchmarking the round-trip times. The UDP data payload size was set to 8192 bytes for a single packet.

Figure 4-16 shows the use of UDP in the RH-Edge results in much lower round-trip times as compared to TCP. The UDP based transport has higher packet loss which can be attributed to lack of reliability mechanisms in the UDP transport protocol. We also observe that the use of echo filtering on the TCP transport helps improve the round-trip times. This is because, without echo filtering, the MAC ACK mechanisms can be triggered by the echo messages, resulting in MAC retransmissions and hence longer round-trip times.

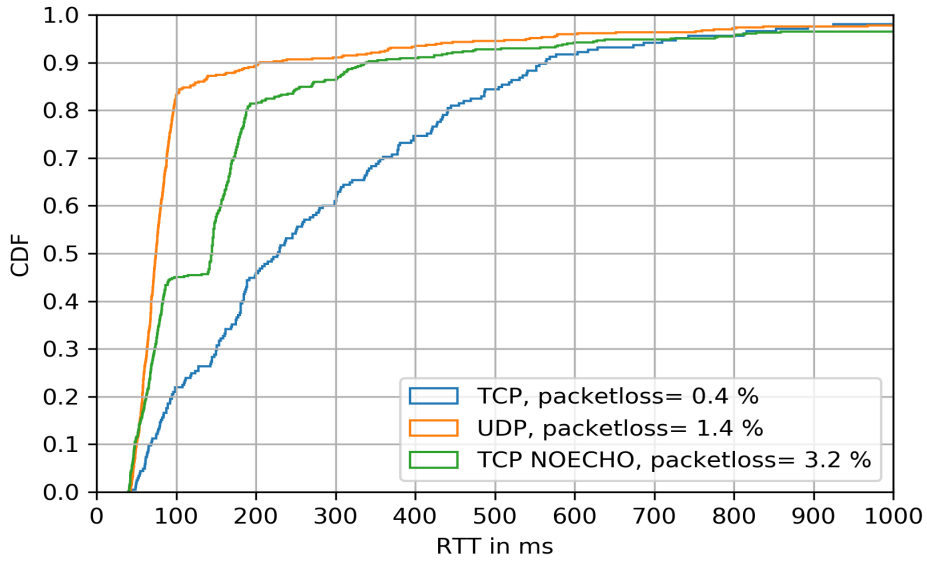


FIGURE 4-16: CDF OF RTT FOR PING MEASUREMENTS WITH DIFFERENT TRANSPORT PROTOCOLS

Rx Packet detection is a general problem with any packetized wireless protocol. We present our 802.15.4 implementation for an example. However, Tx echo handling is USRP specific problem.

The SDR continuously streams radio samples to the Radiohead. Most of these radio samples do not correspond to 802.15.4 packets and hence do not add any value to the IoT gateway located at the Edge. The transfer of these samples results in wastage of network and compute resources. We need to detect which samples correspond to 802.15.4 packets at the radio head and only stream the relevant samples to the Edge.

We evaluated three different filters for this purpose. The first iteration was done using a moving average RSSI filter which was followed by exponential moving average squelch filter. Finally, we implemented a preamble filter which correlates the incoming sample stream with the sample sequence for the IEEE 802.15.4 preamble. All these filters are threshold based gated filter. If the output of the filter for the incoming filter is above the threshold, then the samples are forwarded to the Edge using the RH-Edge interface otherwise they are dropped.

In order to compare the performance of the filters, an experiment was designed. Since the objective of the filter is to correctly filter out the unnecessary samples while forwarding the correct samples, the data size of the output samples and the number of packets detected from those data files were chosen as the output metrics for this experiment. A Zolertia firefly was programmed to

send 802.15.4 packets at regular intervals of 8 seconds. A USRP was used as the receiver which fed the three filters running simultaneously. The output of these filters was saved to files and decoded using the 802.15.4 PHY module in GNU Radio. The decoded packets were parsed using Wireshark. The size of the output file is reported as Data Size, while the number of packets decoded correctly by Wireshark is shown as Packets Detected in Table 4-5. This test was carried out in a controlled office environment with low traffic on the radio channel reducing the chances of over the air collision.

TABLE 4-5: RX PACKET DETECTION EXPERIMENTAL RESULTS

Filter	Data Size	Packets Sent/ Packets Detected
RSSI	10.9 GB	41/41
Squelch	2.9 GB	41/41
Preamble Detector	307 MB	41/41

Our results show that the preamble detector is the best filter among the three as it was able to correctly decode all the packets while reducing the data size transferred by 97% and 89% in comparison to the RSSI filter and Squelch filter respectively. Both the RSSI and Squelch filter works on the input power of the incoming samples. So, they are generic and any relative power signal will be forwarded. On the other hand, the preamble detector is designed for only 802.15.4 packets and is not dependent solely on the input power hence it helps in better filter performance.

Due to poor isolation of the TX and RX channels on the USRP, the transmitted signals are loop backed and received by the receive chain. This results in unnecessary data sent from the radio head to the edge and further computation at the edge. Since there is no solution found in the literature, we come up with our method of ensuring half duplex operation, as illustrated in Figure 4-17. Initially the receiver is turned on (state S0). When the transmission burst starts we turn off the receiver (move to state S1). Since there is a certain delay from the transmission delay between the GNU Radio and the USRP operations, we need to take that into account when we are gating the receiver. On receiving the end of the burst, we wait for a certain time (in our case 700 μs) before turning on the receiver.

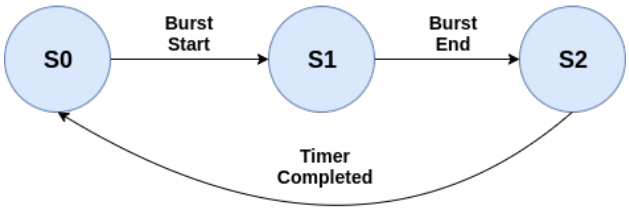


FIGURE 4-17: STATE TRANSITION FOR ECHO HANDLING

NB-IoT is designed to be compatible with LTE frame structure and maximize the reuse of LTE transceiver functionalities. Such a design facilitates the in-band and guard-band deployment in the LTE band [24]. In this work, we focus on downlink (specially NPSS and NPDSCH) to showcase one implementation example as one RAT in the Multi-RAT IoT use case.

As presented before, the key idea of this use case is to centralize baseband processing to the Edge. It is beneficial to reduce the fronthaul (FH) data sent between radio heads and the Edge. The required FH data rate can be expressed as

$$R = f_s b$$

where f_s is the sampling rate of IQ sample and b is the number of bits per complex IQ sample.

The bandwidth of a NB-IoT signal is 180 kHz which comprises of 12 subcarriers with subcarrier spacing of 15 kHz to be compatible to LTE implementation. According Nyquist theory, the minimum sample rate required for a complex baseband is the bandwidth. In practice, the sample rate should be set moderately higher than the bandwidth, to remove the aliasing effect and relax the anti-aliasing filter design. For NB-IoT, 240 Ksps seems a good choice, which can be generated by a 16-point FFT.

However, in NB-IoT, each slot of 0.5ms comprising of 7 OFDM symbols has two cyclic-prefix (CP) lengths. The CP on the first symbol in each slot is 5.2 us long while the CP on the following six symbols is 4.7 us long, as illustrated **in Figure 4-18**. To have an integer number of samples of these two CPs, the minimum sample rate is 1.92 Msps, which is 8 times higher than 240 Ksps. With 1.92 Msps, the first CP is 10 samples long while the second CP is 9 samples long. In addition, it requires 128-point FFT which requires much more complexity than 16-point FFT with 240 Ksps.

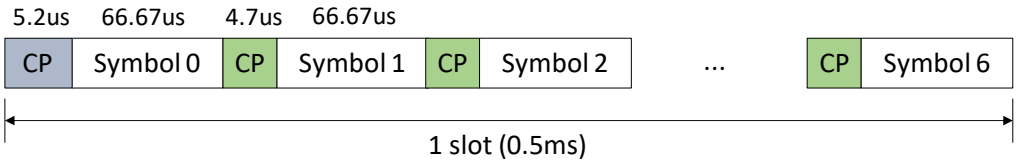


FIGURE 4-18: NB-IOT SLOT STRUCTURE

Figure 4-19 shows the measurement results with sample rate of 1.92 Mbps. At the USB interface, the number of bits per IQ sample is 32 bits, i.e. 16-bit fixed point format for each I/Q sample. At the Edge-RH interface, the number of bits per IQ sample is 64 bits, i.e. 32-bit floating point format for each I/Q sample used in GNU Radio. The measurement verifies that the required FH bit rate is quite high (~125 Mbps) for one NB-IoT cell. It should be also note that the USB-interface bit rate is only half of that of the Edge-Radio interface because of more bits are used to represent IQ samples in GNU Radio. When we aggregate many cells to the edge, this would require a large bandwidth on the FH network and thus increase the cost. Therefore, there is a need to reduce FH bit rate. In this project, we refer to this as FH compression.

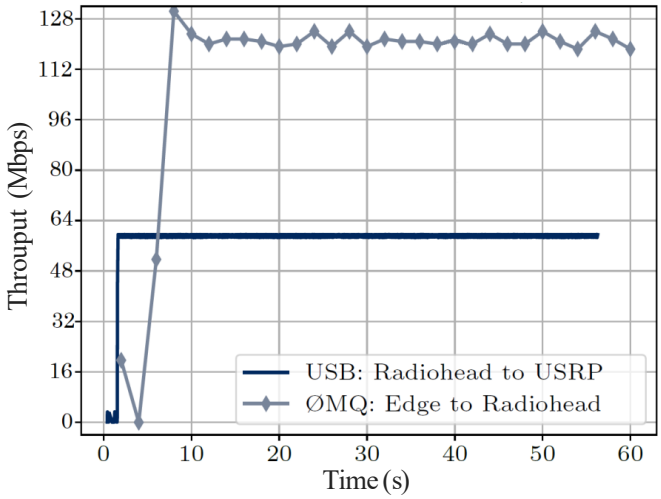


FIGURE 4-19: MEASURED FRONTHAUL THROUGHPUT WITHOUT COMPRESSION

As discussed above, the current implementation is highly over-sampled. This indicates that FH can be largely compressed by reducing the sample rate. For example, FH bit rate can be reduced by 8 times by reducing the sample rate from 1.92 Msps to 240 Ksps. To achieve this, we take the functional split principle [25] and move the function for adding CP to radio head, as illustrated in Figure 4-20 (b). In this way, the sample rate of the samples going through FH is reduced to 240 Ksps comparing to the case without compression of 1.92 Msps. We implemented the compression in our testbed. As shown in Figure 4-21, the measurement results show that the bit rate is reduced to about 16 Mbps. It verifies 8 times compression due to implement 8 times sample rate reduction.

In theory, the fronthaul can be further compressed by reducing the number of bits per IQ sample. Having 64 bits per IQ sample is redundant obviously. After all, the original data format used in USRP is only 32 bits per IQ sample. Based on our experience and expertise in the FH area, it is feasible to compress it down to 16 bits. Combining both functional split and efficient data format, the FH bit rate per NB-IoT signal can be compressed to 4Mbps, which is reasonably low to fronthaul many cells between radio heads and the Edge.

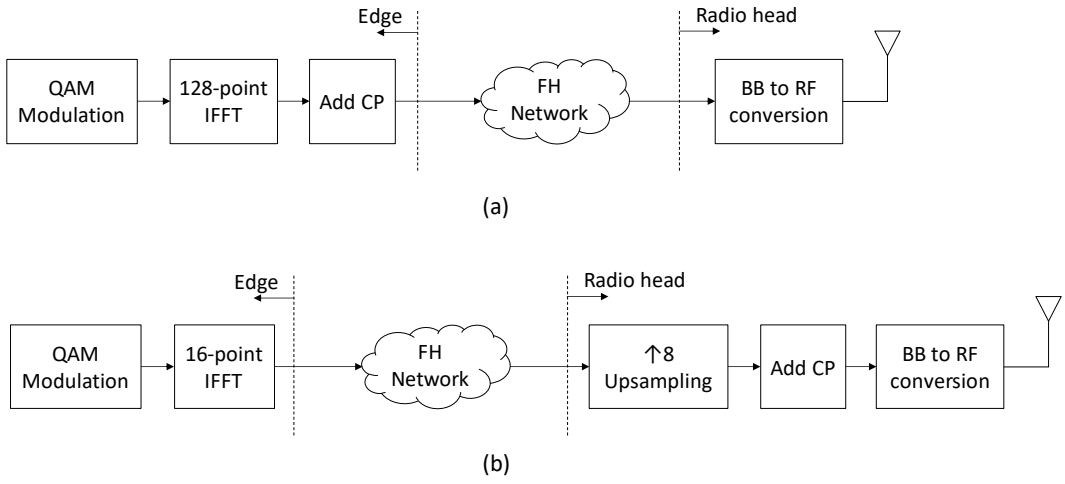


FIGURE 4-20: DOWNLINK BLOCK DIAGRAM (A) WITHOUT COMPRESSION AND (B) WITH COMPRESSION

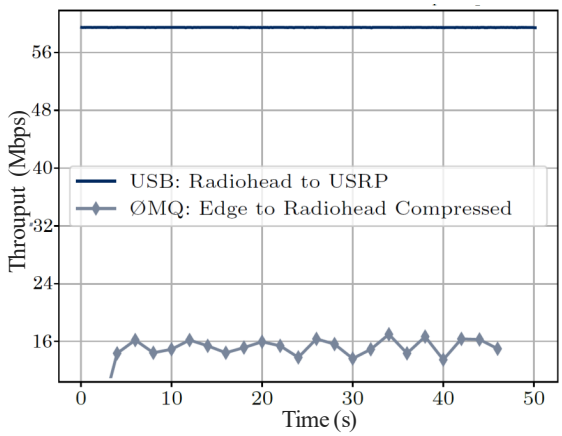


FIGURE 4-21: MEASURED FRONTHAUL THROUGHPUT WITH COMPRESSION

4.4.2.2 Multi-channel design

Massive IoT use cases require high network capacity and availability to provide reliable connections to a dense collection of sensor nodes. In order to address these demands, we propose a multi-channel IoT Gateway that is able to communicate over multiple radio channels, ensuring nodes operating on different radio channels can be part of the same network. This increases the capacity of the network as well as its availability. Since in our use case, the encoding and decoding of radio technologies are done in software, this approach can be extended to multiple radio technologies on multiple radio channels using a single antenna.

As a proof of concept, we designed a three-channel IEEE 802.15.4 gateway. The SDR samples three adjacent radio channels. These wideband radio samples are segmented into samples for individual radio channels using a polyphase channelizer for the receive chain. For the transmit chain, the samples from individual channels are coalesced to form the wideband signal which is transmitted using the USRP. Our 802.15.4 setup for a single channel described in D4.1[26] can then be used for the individual transmit and receive chains for each channel. Figure 4-22 and Figure 4-23 show the block diagram of the multi-channel transmitter and receiver implementation.

FIGURE 4-22: MULTI-CHANNEL RECEIVER IMPLEMENTATION

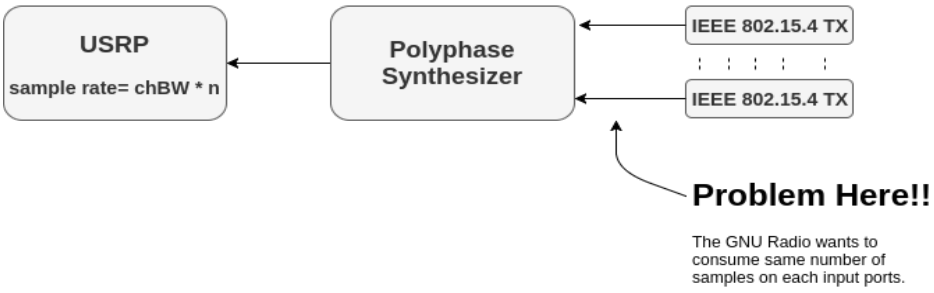


FIGURE 4-23: MULTI-CHANNEL TRANSMITTER IMPLEMENTATION

Our 802.15.4 TX generates bursts of IQ samples corresponding to 802.15.4 packets. The polyphase synthesizer is designed as a synchronous block in GNU Radio, in order to have proper

matrix dimensions for the IFFT operations. This implies that the polyphase synthesizer will consume the same number of samples on all input ports. So, if the length of the IQ samples coming from different 802.15.4 TX processes is different, then only the smallest message will get transmitted together with the same number of samples from the other messages. The remaining samples have to wait for samples to be available on the port with the smallest message size. The radio transmission for these messages has been illustrated in Figure 4-24, where there is a short discontinuity of the transmitted signal. The receiver on the Zolertia Firefly will see this discontinuous signal as a corrupted packet.

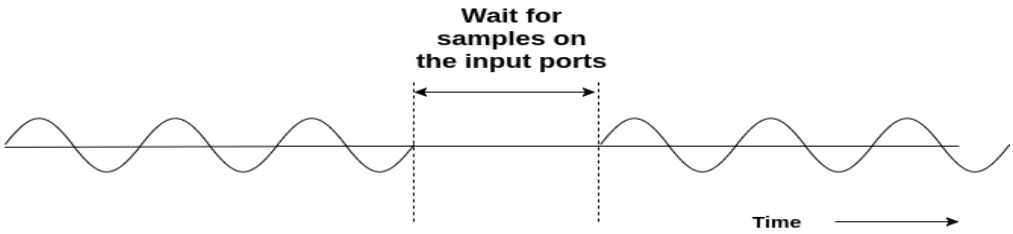


FIGURE 4-24: BREAKDOWN OF RADIO TRANSMISSION.

In order to alleviate this problem, we tested different approaches. We tried to make all the MAC packets of the same size by appending zeros after the CRC. The zeros get modulated as well by the PHY and result in corrupted packets on the Firefly because the RF driver on the Firefly processes received packet based on the length of the reception, instead of using the actual length field in the packet. Another approach is to send zeros on the ports that do not have data to output. If the 802.15.4 TX ports produce the packet data slower than the consumption rate, zeros are going to be inserted between the samples of the same packet. This results in having the radio transmission breakdown problem in the middle of the packet. This is always going to be the case with the split case as the data rate for the UDP/TCP varies.

Considering the drawbacks of these two methods, we buffer the incoming samples for an 802.15.4 packet in order to overcome the problem of variable data production rate. Our IEEE 802.15.4 TX provides metadata about a sample stream, like the start of a packet and end of a packet using GNU Radio tags. We make use of these tags, to properly identify the beginning and end of packets. We output samples when a packet is buffered in the internal buffers. If any other ports have some packets buffered, then we output zeros on this output port. The flowchart for a single port is shown in Figure 4-25. We use a pair of threads to read from the input buffers and write to the output buffers respectively in order to increase the data throughput.

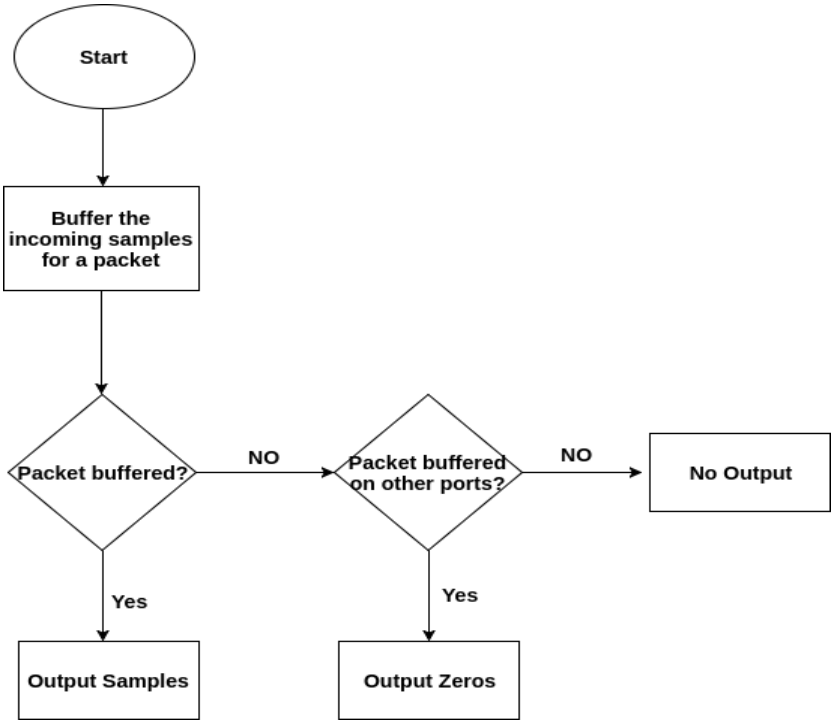


FIGURE 4-25: FLOWCHART FOR OUR METHOD FOR A SINGLE INPUT PORT - OUTPUT PORT COMBINATION

In our initial implementation, we used a vector of queues for our internal buffers, which leads to data rate problems as each write to this vector exceeds the capacity and a new vector is created with the old elements copied to this new vector. This slows down the read and write processes. In order to avoid this problem, we used large ring buffers in our implementation. This helps alleviate the data rate problem. In order to handle burst traffic, the USRP needs to be sent proper tags for managing the transmission process. Otherwise, the USRP will wait for internal buffers to be filled up which leads to radio transmission breakdown in case the packet transmission has ended on the GNU Radio flowgraph. It is difficult to determine when our burst will end. Hence, we append zeros to the end of each transmission and attach the End of Burst tag to the last sample of these zeros as illustrated in Figure 4-26. These zeros are appended to the physical layer sample stream which can be interpreted as no radio transmission from the USRP. Thus, these zeros do not result in packet corruption on the Firefly.

FIGURE 4-26: STRUCTURE OF EACH TRANSMISSION

We design an experimental setup illustrated in Figure 6. We have a host computer running our IoT Multi-channel gateway process for three channels. The receive and transmit chains are designed as shown in Figure 4-22and Figure 4-23 respectively. We use the 802.15.4 channels 24, 25 and 26 for this experiment. The Zolertia Fireflies are configured as shown in Figure 4-27, with one device on each channel. We use the ping command for our measurements. The experimental results are presented for 500 ping messages.

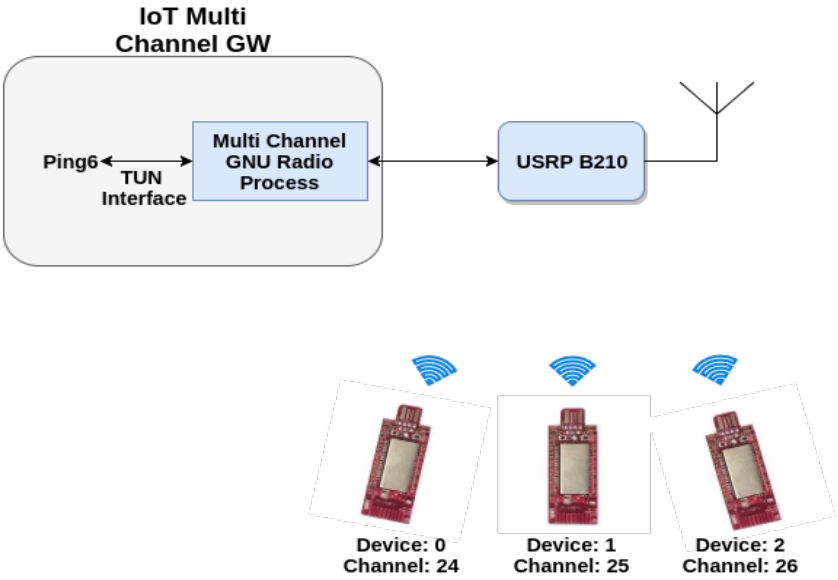


FIGURE 4-27: EXPERIMENTAL SETUP

Figure 4-28 shows the impact of simultaneous ping on multiple channels on the ping results for a single channel. The graph also shows the CDF for ping results from our single channel Implementation and when using a native Contiki border router. The results show our multi-channel implementation has higher round trip times compared to the single channel implementation. We hypothesize that this is mainly due to the buffering of all the samples of a packet needed to solve the discontinuity of radio transmission and also the extra computation needed for the polyphase synthesizer and channelizer. As we increase the number of channels on which we ping simultaneously, the reliability decreases. And we see a higher variance in the ping results. We think packet corruption requiring multiple retransmission when we transmit on multiple channels in the main reason for these variations.

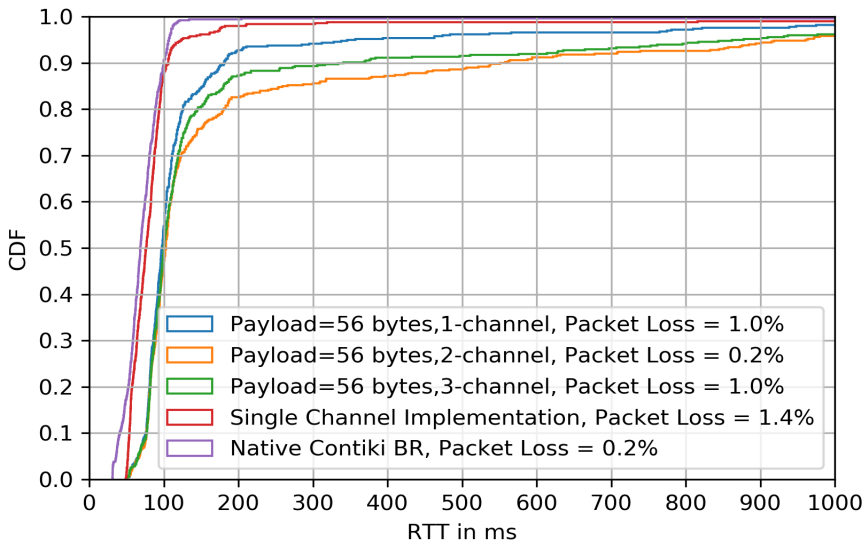


FIGURE 4-28: CDF OF RTT FOR PING MEASUREMENTS WITH 56 BYTES PAYLOAD

In Figure 4-29, we see that packet loss increases with increasing ping data payload size. The data payload size does not have a significant impact on the round-trip times of the ping data packets. As the buffering time for samples from different message sizes would be different but the lack of significant differences for these different data payload sizes highlights the main delays in our multi-channel implementation as compared to the single channel implementation is mainly occurring from the polyphase synthesizer and channelizer.

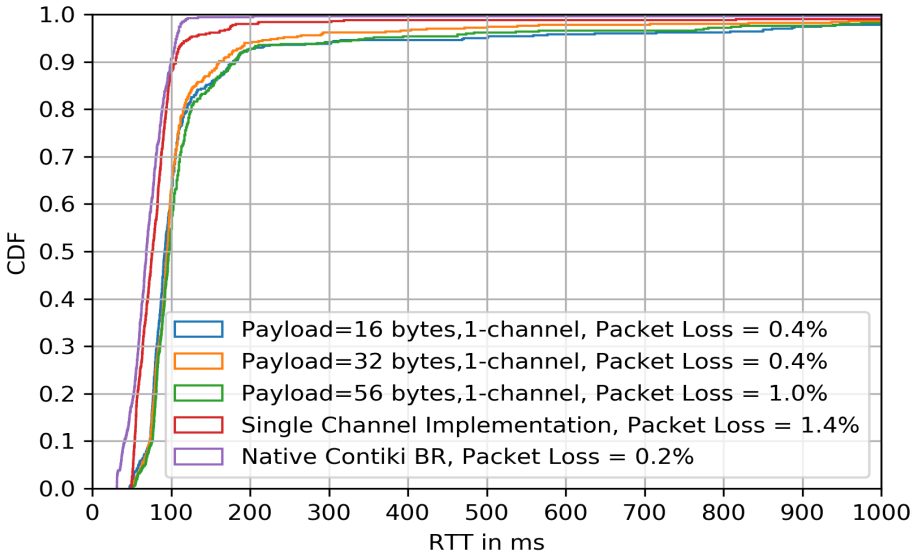


FIGURE 4-29: CDF OF RTT FOR PING MEASUREMENTS WITH DIFFERENT PAYLOAD SIZES

To summarize, we are able to communicate with multiple radio nodes on multiple channels using this approach. The reliability of our implementation can be improved with particular focus on understanding what the main causes of the extra delays are and how to improve the design. Our implementation showed variation in the results from different channels. We attribute this mainly to how we designed our polyphase filter banks, a closer look at the design of these filters is needed.

4.4.2.3 IQ service and interference analysis application

The focus of the EFS implementation for this use case has been on developing the multi-RAT communication stack functions, evaluate the design and showcase the feasibility by PoC testbed development. The work is still ongoing regarding establishing MQTT-based EFS platform with IQ service and the development of the interference analysis application. These are also taken as part of the integration work in WP4 where we will integrate the developed EFS elements in one testbed together with orchestration features. Therefore, more details regarding this part will be reported later in D4.2, the final deliverable of WP4.

4.4.3 Conclusions and future directions

In this deliverable for the Multi-RAT IoT use case, we provided a reference EFS design which is fully based on the 5G-CORAL architecture and comprised of all three key EFS elements in the design example. The design has been functionally tested and verified in lab tests and demonstrated in two public demonstration events at EUCNC 2018 and ICT 2018 in Vienna, as well as two internal demo events at mid-term reviews in Taipei and Vienna, in November and December 2018, respectively.

Implementation-wise in this deliverable, we have addressed two key issues regarding efficient RH-Edge interface design and multi-channel transceiver implementation, which are crucial for

cloudifying multi-RAT communication stacks. The experimental results prove the feasibility of implementing this use case following the 5G-CORAL concept and architecture.

In 5G-CORAL, we focus on proof of concept studies and prototyping works of cloudifying existing IoT stacks, like IEEE 802.15.4, NB-IoT and LoRa. In future works, we plan to dig more into IoT protocols, explore the possibilities to relax timing requirements in the protocols, and investigate the tradeoff between latency and performance. The idea is in the direction to explore new designs of cloud-friendly protocols, instead of being limited by the existing protocol design. This would enable to cloudify more RATs into the Edge and thus further reduce the overall IoT network costs by resource sharing and increase system flexibility and scalability to address the future challenges for serving trillions of IoT devices in the full Digital Society era.

4.5 Connected Car

The connected cars use-case aims at proving the advantages offered by the 5G-CORAL architecture to improve the road safety. The low latency provided by this platform enables collision avoidance algorithms to be really effective. Also, the fact that it is a distributed architecture allows for several devices (potentially all the vehicles in the World) to publish their telemetry information at a high rate. This is possible only if that data traffic remains geographically confined where the car is located, which is also where that data is useful: clearly, a car in a certain city should not be receiving telemetry information from a car located in a different city, or even in a different neighborhood.

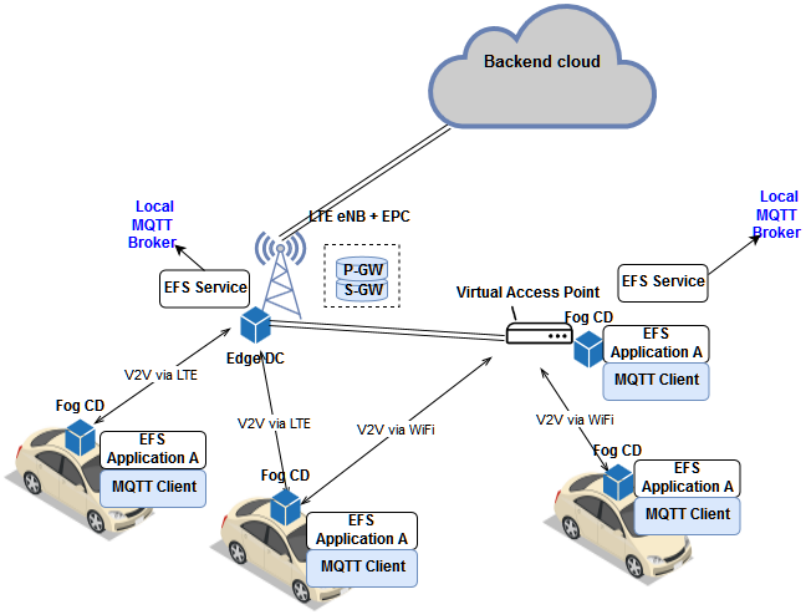


FIGURE 4-30: CONNECTED CARS SCENARIO

The Figure 4-30 describes the connected cars use case scenario, where the cars can be connected via LTE, via WiFi to an RSU or both simultaneously.

4.5.1 Refined EFS design and functional validation

The Figure 4-26 describes the refined EFS designed for the connected cars use case.

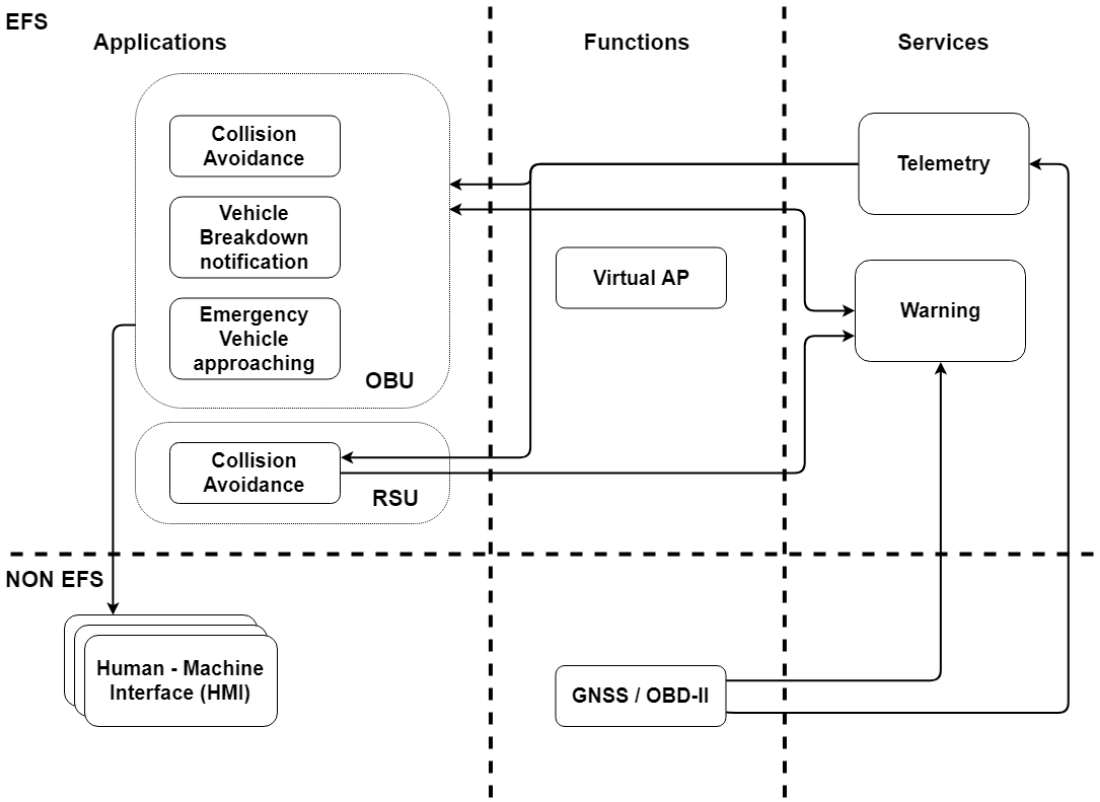


FIGURE 4-31: EFS ELEMENT IN CONNECTED CAR USE CASE

In the connected cars use case, two services are offered on the EFS Service Platform: Telemetry service and Warning service. The first one provides some telemetry information from the vehicles, such as position and speed. The warning service provides notifications related to road hazards. Both services are based on ETSI standards, in particular the ESTI CAM [27] has been adopted for the telemetry service message contents, while the ETSI DENM [28] for the warning messages.

A Road Side Unit (RSU) has been introduced in the design. It provides a secondary RAT access, which could be DSRC (Dedicated Short Range Communications), C-V2X (Cellular Vehicle-to-Everything), etc. For the PoC we decided to use Wi-Fi as a simple way of proving the concept. For this reason, a Virtual Wi-Fi access point can be deployed as an EFS Function.

The EFS applications are the ones consuming and producing the messages that are exchanged over the EFS Service Platform. In particular, the On-Board Unit is capable of:

- Processing the Telemetry messages to warn the user about collision risks (Collision Avoidance application)
- Generating and publishing warnings if a hazardous malfunction is detected (Vehicle Breakdown Notification)
- Warning the user in case of an approaching emergency vehicle (Emergency Vehicle Approaching)

The very same Collision Avoidance application can also be instantiated on the RSU, which collects the telemetry information of several vehicles in a certain area and generates warnings if necessary.

4.5.2 Use-case specific implementations and experimental verification

The On-Board-Unit runs the EFS application as a Legato Framework (www.legato.io) application which allows to access the various sensors and interfaces on the board and, most importantly, keeps the applications monitored and sandboxed so that it is possible to start, stop, install and remove applications.

When started, the application on the OBU reads a configuration file where several parameters can be configured, such as LTE and WiFi connection parameters, MQTT broker details, sampling period of the sensors providing telemetry data and tuning parameters for the application algorithms. A full list of the configuration parameters is available in Appendix 7.1.3.

The application can be divided into two main components. The first one is responsible for generating the telemetry messages (CAMs) and publishing them to the EFS service platform. The second one takes care of receiving the telemetry messages from the other vehicles nearby and does the processing needed to compute if a collision risk is real and, in that case, generate and publish a warning (DENMs) message for the position where the car is located.

In addition to that, a module responsible for generating the warning message according to the ETSI ITS standards has been introduced. In other words, the message needs to be periodically re-transmitted depending on its urgency and it must be terminated if the originating cause has disappeared.

Some management of the warning messages is being done also when the messages are received. Since each warning message has a position and an expiry time attached to it, the OBU that receives one message needs to determine if the car is within the relevance area of the warning and trigger or stop the alarm depending on whether the car enters or leaves such area. Note that this requires the OBU to store all the warning messages it received and remove them only if they expire or if a termination message is received via the service platform by the vehicle that originated the message.

The telemetry messages, which include the vehicle characteristics (length and width), its speed, its location and other details, are generated at a rate which varies between 1Hz and 10Hz depending on how much the heading, speed and position of the vehicle have changed since the last transmission of a message. In other words, the telemetry message is sent by default every second, however the values are monitored ten times as fast and, if there is a big enough change, the data it is transmitted right away. This is done to reduce the network load by doing a minimum pre-filtering of the data, while keeping a high data-rate when needed.

For the demonstration we used a JSON encoded message since it is easier to process and debug across multiple platforms, however we studied other data encoding protocols, namely CBOR (Concise Binary Object Representation) and Protocol Buffers which reduce the payload size respectively by 42% and 82%. The reduced payload size results in a reduced latency, as it is summarized in table 11.

TABLE 4-6: WIFI AND LTE LATENCY MEASUREMENTS (AVG. OVER 2500)

Latency	WiFi	LTE (in TI LAB)
1900 bytes (JSON)	Min: 48.399 ms Avg: 52.555 ms Max: 94.183 ms Stddev: 4.204 ms	Min: 33.399 ms Avg: 48.211 ms Max: 122.080 ms Stddev: 4.073 ms
1020 bytes (CBOR)	Min: 26.137 ms Avg:29.045 ms Max: 76.766 ms Stddev: 4.335 ms	Min: 31.933 ms Avg: 46.648 ms Max: 104.751 ms Stddev: 6.014 ms

346 bytes (Protobuf)	**Min:11.542 ms Avg:15.347 ms Max:57.350 ms Stddev: 5.479ms**	**Min: 20.309 ms Avg: 34.871 ms Max: 79.587 ms Stddev: 3.757 ms**

In this table we collected the minimum, average, maximum latencies and their standard deviations for the exchange of telemetry messages over WiFi and LTE. The WiFi measurements where performed while being connected to a Virtual AP (EFS Function) running on the RSU; The LTE measurement where performed in a 5G-CORAL environment, where the MQTT broker was instantiated very close to the eNB. The table shows how the latency varies with the different encoding formats. In the PoC we will consider the scenario where the OBU is connected to both RATs simultaneously, therefore the resulting latency is equal to the lowest one, with the added benefit of an increased reliability.

4.5.3 Conclusions and future directions

Mainly three aspects have been identified for the future directions: localization accuracy improvement, a more robust V2V RAT and an improvement of the algorithms behind the EFS applications.

The location of the vehicle has currently been determined using a GNSS receiver, which has an accuracy of a few meters. While this is fine for proof of concept and for certain applications (e.g. emergency vehicle approaching notification), there are several possible improvements. First of all, a Kalman Filter can be used to perform sensor fusion between the GNSS position information, the vehicle speed (which can be measured from the angular velocity of the wheels) and the accelerations in the X, Y and Z axis that can be acquired with an Inertial Measurement Unit (IMU). In addition to this, the 5G-CORAL infrastructure can enable data fusion that would not be possible without communication between cars. In practice, a LiDAR can be installed on the vehicles and the distance between nearby vehicles can be published on the localization EFS service. Again, a Kalman filter (or a more advanced Particle Filter) can be used to improve the localization of two vehicles by doing data fusion and combining the LiDAR measured distance with the distance calculated by knowing the coordinates of the two cars.

The second element that can be improved is the second RAT available for the communication with the RSU. The currently used Wi-Fi can be replaced with a more adequate C-V2X based communication, without having to re-design the architecture and logic of the application that manages the Multi-RAT aspect of the connected cars use case.

Finally, the applications can be improved to include more refined collision avoidance algorithms. For example, an algorithm able to predict over a certain time span the movement of two cars can drastically anticipate the warning of a possible collision to the driver and, as well as reduce the number of false alerts.

4.6 SD-WAN

The SD-WAN use case aims to leverage SDN and NFV technologies to provide a secure and reliable interconnection network within the 5G-CORAL platform to connect the edge, fog and cloud and ultimately enable federation. Moreover, SD-WAN functions perform as virtual gateways, establishing a virtual connection among them, which transparently connects elements in different locations under the same virtual network. In the shopping mall scenario, point of sale (PoS) applications are deployed in one of the shopping malls EFS domains. An additional SD-WAN function is deployed at another EFS close to the initial shopping mall EFS domain, this EFS domain could be owned by a train or bus station company located close to the shopping mall. For this use

case we will analyse how federation can be used between these two domains to locate/offload precisely functions, and applications close to the end user. In this scenario we will simulate an end user moving from one federation consumer domain to a federation provider domain, enabling the consumer domain to take control of a fog node at the provider domain by leveraging federation, which will use to instantiate a PoS web application to offload traffic from end users connected to the provider domain. The use of federation with the combination of the SD-WAN function, allows domains to scale outside their own infrastructure.

4.6.1 Refined EFS design and functional validation

The main entity deployed in the use case is the SD-WAN function, acting as a virtual gateway. SD-WAN functions are controlled by two components, the SD-WAN manager and SD-WAN controller, integrated into the SD-WAN EFS function manger, work together to establish a secure virtual network across SD-WAN functions. Through the SD-WAN function, control and data plane traffic flows, allowing the SD-WAN function to selectively separate both planes. In the use case presented in 4.6 the data plane is the traffic to and from PoS terminals, which flows though the PoS Web App and PoS DB, control pane traffic is composed by OCS interfaces with the EFS, which for this scenario will be the communication between the VIM(fog05) and the ESF infrastructure and the communication between the SD-WAN function and the SD-WAN EFS Function Manager.

The next defined function is the Virtual WiFi Access Point, linked to the SD-WAN function in a layer 2 network basis. It will forward data from the WiFi interface to the SD-WAN function, which will route the traffic to the intended destination, which in this use case will be the home/consumer domain PoS Application or the Provider domain offloaded PoS Web Application.

Figure 4-32 shows an example of PoS service composed of two applications: the customer and inventory database (PoS DB) and the PoS web application (PoS Web App). While the database stays in the consumer domain, the web application can be deployed closer to the end user in the federated domain. Both service components are chained together by the SD-WAN function, allowing them to communicate over a secure virtual link.

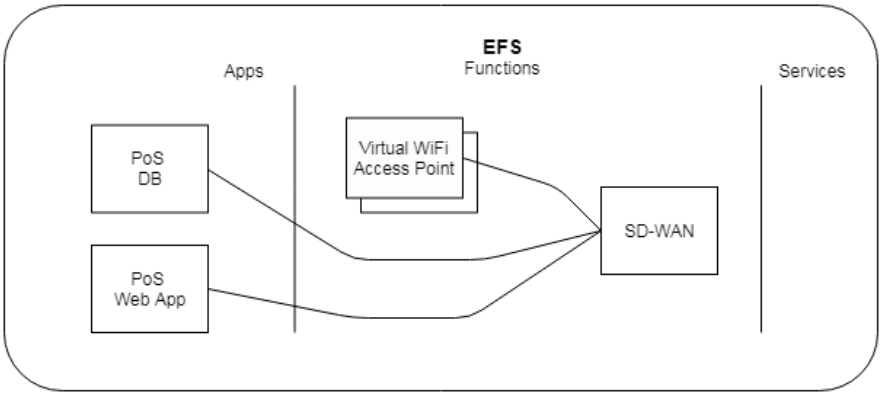


FIGURE 4-32: EFS ELEMENTS IN SD-WAN

4.6.2 Use-case specific implementations and experimental verification

This subsection explains the measurements gathered from the experiment setup, were we are testing the connectivity between a virtual host (representing the EFS PoS WebApp) and a PoS terminal accessing via WIFI both located in in the same EFS domain and leveraging the SD-WAN function to provide the connectivity. The experimental setup is composed by one Dell Latitude E5550 laptop with 8GB of RAM, Intel i5-5300U CPU and 500 GB of HDD, used to simulate the EFS domain. Instantiated on the laptop there is a virtual AP function (LXD container), and a virtual machine (KVM) were the SD-WAN function and the EFS PoS WebApp will be instantiated.

Additionally, a PoS terminal composed of an additional Dell Latitude E5550 laptop with the same characteristics as the one simulating the EFS domain will be sued to execute the validation experiment.

The results provided in Table 4-7 serve as provisional results, which help us to analyse the impact of the SD-WAN in the data plane. Table 4-7 represent latency metrics collected using the ping tool between both the PoS terminal and the PoS WebApp. For this experiment, one hundred ping samples were taken and the period between ping samples is increased from 0.25s to 1s doubling the value each time. Additionally, the size of the ping packets is tuned to 56B, 128B, 256B, 512B; measuring latency over different packet sizes. Ping results are shown in the default ping tool format minimum/average/maximum/standard-deviation.

TABLE 4-7: LATENCY MEASUREMENTS

Latency	T=0.25s	T=0.5s	T=1s
56B packet	7.108/72.124/412.909/96.857 ms	7.410/55.643/374.333/73.563 ms	7.474/61.247/548.430/87.039 ms
128B packet	7.449/67.957/562.680/93.676 ms	7.278/50.165/281.554/60.898 ms	6.936/87.566/558.190/104.784 ms
256B packet	6.611/54.922/412.721/71.129 ms	7.392/67.699/743.926/99.153 ms	7.277/72.065/564.883/118.195 ms
512B packet	7.547/43.844/336.875/58.958 ms	7.307/58.636/329.302/66.779 ms	7.562/55.484/319.116/73.125 ms

Following the latency measurements, the next experiments tries to extract some more fundamental network metrics, such as, Jitter, Bandwidth and service deployment time. The results are presented below:

- Jitter (10 seconds test) (UDP):
  - 15.3 Mbytes transferred
  - 12.8 Mbits/sec
  - 7.544ms of jitter
- Bandwidth (10 seconds test) (TCP):
  - Sender: 13.6 Mbits/sec
  - Receiver: 13.2 Mbits/sec
- Service deployment time:
  - 12 mins for VM deployment using fog05
  - 3 mins for container deployment using fog05

Results from the experiments carried out in this section are diverse, indicating us that there could be a bottle neck in the scenario. The bottle neck which was identified is that the processor (cpu) used during the experiments, which does not have hardware acceleration for cryptographic operations. This type of hardware acceleration in CPUs is only present in medium to high end CPUs, rarely found in constrained devices.

4.6.3 Conclusions and future directions

Based on the results above, after the implementation of the SD-WAN function, the WIFI AP and both PoS service application are ready, there will be a second round of measurements in order to provide detailed measurements of a fully deployed scenario with two fully functional EFS domains,

which will enhance the initial results collection, providing a richer set of metrics from all of the functions/applications/services deployed. In this scenario the user will connect to a web application instantiated in the provider domain closer to its location, enabling us to extract more metrics from the scenario, including access network connection times, offloading function instantiation time, impact in the EFS system.

Regarding the future directions, some of them are listed below:

- Software automation and deployment.
- Detection of when the federation should be triggered, e.g., increase of the number of users connected to an edge or fog node or even mobility detection, were connection and disconnection events to a domain are transmitted through the EFS service platform.
- Migrate from a static federation model to a dynamic federation model.

4.7 High-Speed Train

In LTE networks, MME is the main entity which handles control signaling. It is responsible for initiating paging and authentication of the mobile devices. Also, MME retains location information at the tracking area level for each user and then selects the appropriate gateway during the initial registration process. In inter-MME handover, MME plays a vital part in signalling control in standard procedure. In particular, inter-MME handover involves three control stages. The first stage, the source eNodeB initiates the handover by sending a request message over the S1-MME reference point. In the second stage, the source MME, selects the target MME and configure a messaging tunnel over an interface called S10. The S10 is a control interface between MMEs. The last stage occurs when MME transfers the configuration message to target eNB over S1 interface. Needless to say, the high-speed train use case involved the interaction between MME on-board and on-land as described in D2.1in details. Hence, we proposed to adopt the vMME on-board of train in 5G-CORAL. Notably, the S10 interface will include large amount of control signalling especially when hundreds of passenger devices handover simultaneously. Therefore, we also proposed to enhance the S10 interface to reduce the signalling between on-bored and on-land EFS node. Where, the adopted EFS virtualisation infrastructure has the MME functionality. Also, it is the totality of the hardware and software components that build up the environment.

4.7.1 Refined EFS design and functional validation

In high-speed train, the adopted EFS has different purpose entities as shown in Fig. 5-23. The first EFS entity is video streaming application. This EFS application provide the streaming video for UEs. The second entity is EFS functions which consists from two main part:

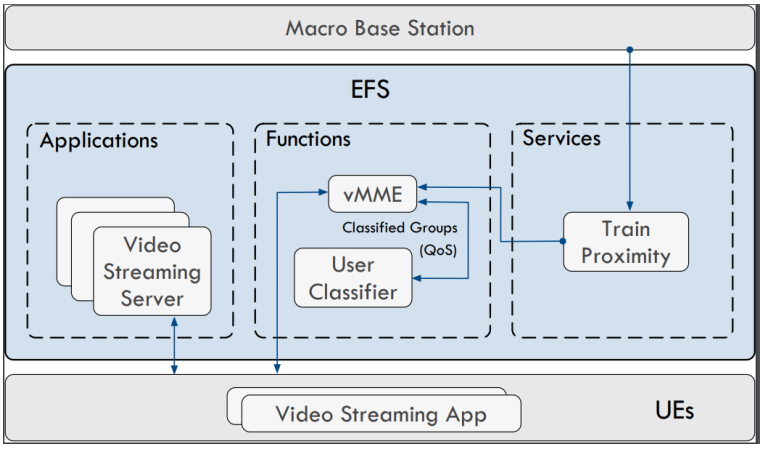


FIGURE 4-33: HIGH-SPEED TRAIN EFS DESIGN

User Classifier function: The UEs on-board have different QoS requirements. The user classifier function classifies the UEs into groups based on context information obtained such as QoS Class Indicator (QCI) and allocation and retention priority (ARP) which will be extracted from vMME. The user classifier function, where the classifications take place in two steps. In the first step, the user classifier extracts context information of UEs and sorts them based on QCI and ARP. Then, it will send the groups in descending QoS order queues to the vMME for handover in the second step.

vMME: With the information provided from user classifier function and train approximate, vMME will execute handover of a group of users as train approach. As the train approaching to the train station, the vMME transmit important UEs context information ahead of time which will reduce the amount of signalling during the handover process itself. In particular, from step 0 to step 3 (see Figure 5-24), vMME will forward the relocation UEs information request to target MME which is the MME in the core network. Also, vMME will receive the relocation response time in this case.

The last EFS entity is the train proximity. This block will be handled in the MME of the core network. The MME monitors several UEs and PCIDs (ID of eNBs), and since the time schedule and the route path of trains are known, the approximate location of the train can be estimated. Then, it executes handover triggering functions for UEs moving to specific eNB. In the two-hop architecture, the MME monitors the CPE on-board. When the train is approaching the station, i.e., the CPE will handover to the specific eNB near the station, then the MME executes the handover triggering function which sends a handover trigger signalling to the vMME.

EFS Node can host several applications, functions and services such as video streaming application, vMME, user classifier function, and train proximity service, respectively. In our experiments, we used NextEPC framework as baseline for vMME. Then, we modify vMME to fit our proposed scheme including modification for S10 interface as elaborated earlier.

In the high-speed train, OCS is responsible to handle the service migration as elaborated in details in D3.2. In our case, the train approximate service and user classifier function will be utilized by OCS to trigger and classify users into groups. As results, the downtime of users which they are moving from on-board to on-land will be minimized.

The flowchart of described enhanced MME implementation is described in Fig. 5-24. Obviously, the steps from 0 to 3 need to be executed ahead of time before handover process is executed in the legacy system.

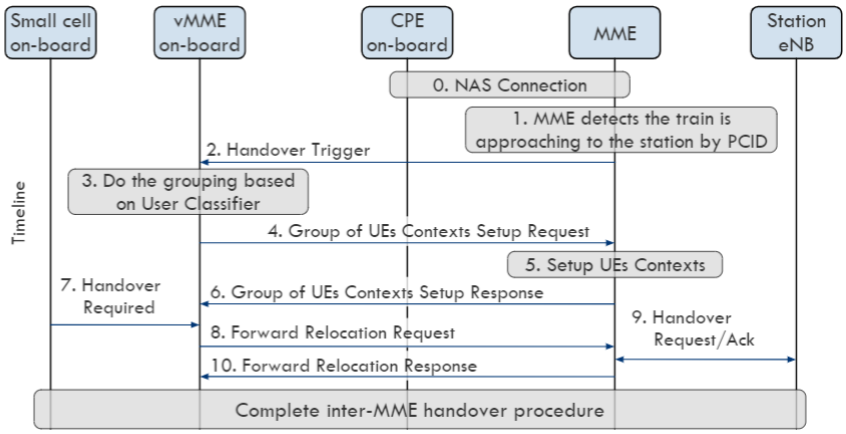


FIGURE 4-34: ENHANCED INTER-MME PROCEDURE FLOWCHART

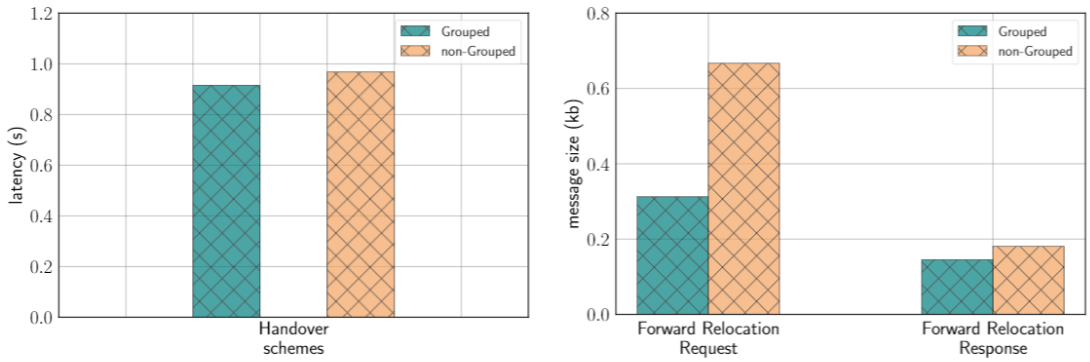
4.7.2 Use-case specific implementations and experimental verification

In the emulation environment of high-speed train, first, the MME, small cell and CPE are powered on. Then, vMME is powered on and connected with HSS in core network. Four emulated UEs connect to the on-board small cell. At the core network, MME connects to the target eNB (on-land). As train mobility is emulated, the MME will send handover trigger to vMME to classify the users into groups based on QoS and ARP. In addition, enhanced inter-MME handover is executed. Finally, we measure the size of control signal packet, handover latency, and downtime then compare the legacy system with the proposed schemes measurements as elaborated in following subsections.

4.7.2.1 Experimental verification

Fig. 5-25(a) represents the comparison of the handover improvement with/without the proposed enhancement for inter-MME. The x-axis represents the enhanced inter-MME (Grouped) and legacy inter-MME (non-Grouped) while the y-axis represents the inter-MME handover time. Obviously, the Grouped handover improved the total handover time slightly comparing with non-Grouped inter-MME handover. In the both cases, the total handover time is large due to two-hop architecture and the emulation environment set up. In real high-speed train, the average handover time is around 200ms. It is worth mention, the latency reduction is not the target of this work and this prove the results in Fig. 5-25 (a) did not create any overhead but reduces the total handover time.

Fig. 5-25 (b) represents the comparison of signalling control messages during handover with Grouped and non-Grouped inter-MME handover. The x-axis represents the forward relocation request and forward relocation response, respectively. The y-axis represents the average control message sizes in bytes. In the case of forward relocation request, the Grouped inter-MME scaled down the average control messages up to 50% per user in comparison to the non-Grouped one. Also, grouped inter-MME reduces forward relocation response up to 25% per user in comparison to the non-Grouped one. Notable, this will reduce the signalling to core network significantly in large scale scenario. At the same time, it will contribute for the application stability at the end user side.



(a) Handover latency (b) Handover control messages overhead

FIGURE 4-35: HIGH-SPEED TRAIN EMULATION RESULTS

4.7.3 Conclusions and future directions

In moving infrastructure scenarios such as train network, the proposed two-hop architecture is adopted to improve on-board user experience by reducing the interaction with on-land base stations. Furthermore, edge clouds and virtualization technologies can be utilized to bring services closer to the traveling users. Nevertheless, when large number of users transit from train to station, a signaling storm and application traffic backhauling become challenges to maintain continuous

service. Our experimental results show that the proposed schemes can reduce the control signaling by 50% when compared to the state-of-the-art.

5 5G-CORAL EFS Monitoring

This section presents the monitoring solution adopted in 5G-CORAL to monitor the distributed EFS resources. First, we provide an overview of 5G-CORAL Monitoring in Section 5.1. Then, we present Prometheus as EFS monitoring platform in Section 5.2. Finally, we provide Prometheus experimental measurements in terms of computing, storage and networking resources under different scenarios in Section 5.3.

5.1 Overview of 5G-CORAL Monitoring

Considering the distributed and heterogenous nature of 5G-CORAL, EFS entities can be instantiated in multiple forms, such as native host applications, containers or even virtual machines. This diversity presents some extra challenges on how 5G-CORAL will tackle monitoring, such as tracking resource utilization from the underlying computing, network, and storage infrastructure or detecting failures. Introducing a monitoring platform will enable 5G-CORAL OCS to be fed by valuable monitoring data and improve its placement, scaling and migration algorithms of the resources located at the EFS. However, the tools which will be used to monitor the EFS, must comply with the 5G-CORAL requirements.

In the 5G-CORAL architecture, EFS monitoring is envisioned as an EFS Service, which leverages E2~=Mp1 EFS interfaces to collect valuable information from the underlying EFS Virtual Infrastructure, EFS applications and Functions, to further expose them to other EFS components (Functions and/or Applications) and OCS components such as EFS Application/Function manager and Resource orchestrator. Additionally, in order to feed the EFS Monitoring service, monitoring agents/probes are additional elements included in 5G-CORAL architecture. These monitoring agents should ideally have a minimal impact on the EFS resources, i.e. computing, storage and networking. Figure 5-1depicts how the EFS monitoring service and agents fit into the 5G-CORAL architecture, including the collection and consumption interfaces from which monitoring data would flow.

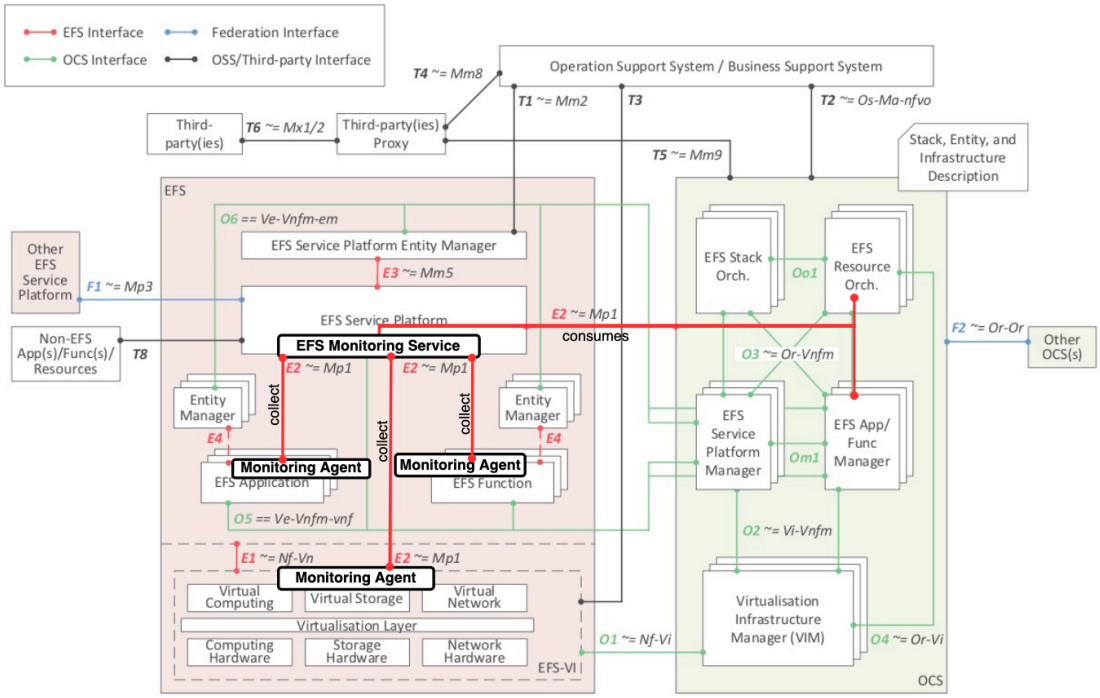


FIGURE 5-1: EFS MONITORING MAPPING TO 5G-CORAL ARCHITECTURE

5.2 Prometheus as EFS monitoring platform

This subsection focuses on monitoring tools and platforms. The metrics exposed from the monitoring tools are detailed in Appendix 8. The different tools described in Appendix 8 cover all kinds of physical and virtual resources used in 5G-CORAL, ranging from virtual machines to containers (docker, LXD) and Linux to Windows operating systems.

Prometheus monitoring platform is an open-source metrics-based time series database, designed for white-box monitoring. It pulls metric data from devices rather than rely on the device to push the metrics (although, push is also available via a gateway). Scalability is supported by deploying many Prometheus servers. Additional features which Prometheus supports include the use of a very simple exposition format, support labels (dimensions/tags) and a single executable. Prometheus data model aggregates all data metrics as time series streams, which are timestamped values of a collection of metrics. These are uniquely identified by its metric name and a set of key-value pairs (labels). Additionally, in case the pull method does not suit the use case requirements Prometheus has an API available to publish data. Prometheus time series is data is usually identified by a metric name and a set of labels (*<metric name>{<label name>=<label value>,…}*) E.g., a time series with metric name api_http_request_total and labels *method="POST"* and *handler="/messages"* could be written as *api_http_request_total{method="POST", handler="/messages"}*.

Prometheus has four client libraries to measure core metrics, namely.

- **Counter** is cumulative metric that represents a single monotonically increasing counter whose value can only increase or be reset to zero on restart.
- **Gauge** represents a single numerical value that can arbitrarily go up and down. For example: measuring temperature/memory usage.

- **Histogram** samples observations and counts them in configurable bucks. For example: request duration or response sizes.
- **Summary** is similar to histogram, samples observations. Provides a total count of observations and a sum of all observed values. For example: request duration and response sizes.

Prometheus relies in data scraping, which is the process of importing data from a web service or API into your own data monitoring, analysis or storage service. By leveraging data scraping Prometheus defines jobs and instances, which are defined by an endpoint you can scrape. A collection of instances with the same purpose or endpoint, which could be replicated for scalability or reliability reasons is called a job. Figure 5-2 represents Prometheus architecture, composed of three main elements, Service Discovery, Rules & Alerts and Local Storage. The detailed description of the internal architecture goes beyond the scope of this deliverable. We utilized a Prometheus integrated component, called cAdvisor. cAdvisor analyses and exposes resource usage and performance data metrics from running containers to Prometheus time series database. Figure 5-3 shows a screenshot of the dashboard where CPU and memory usages. A full list of the metrics supported by cAdvisor are given Table 5-1.

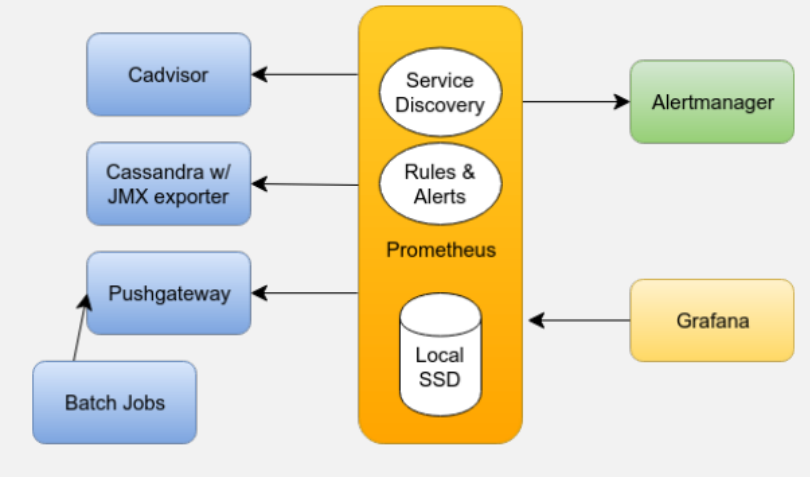


FIGURE 5-2: PROMETHEUS ARCHITECTURE

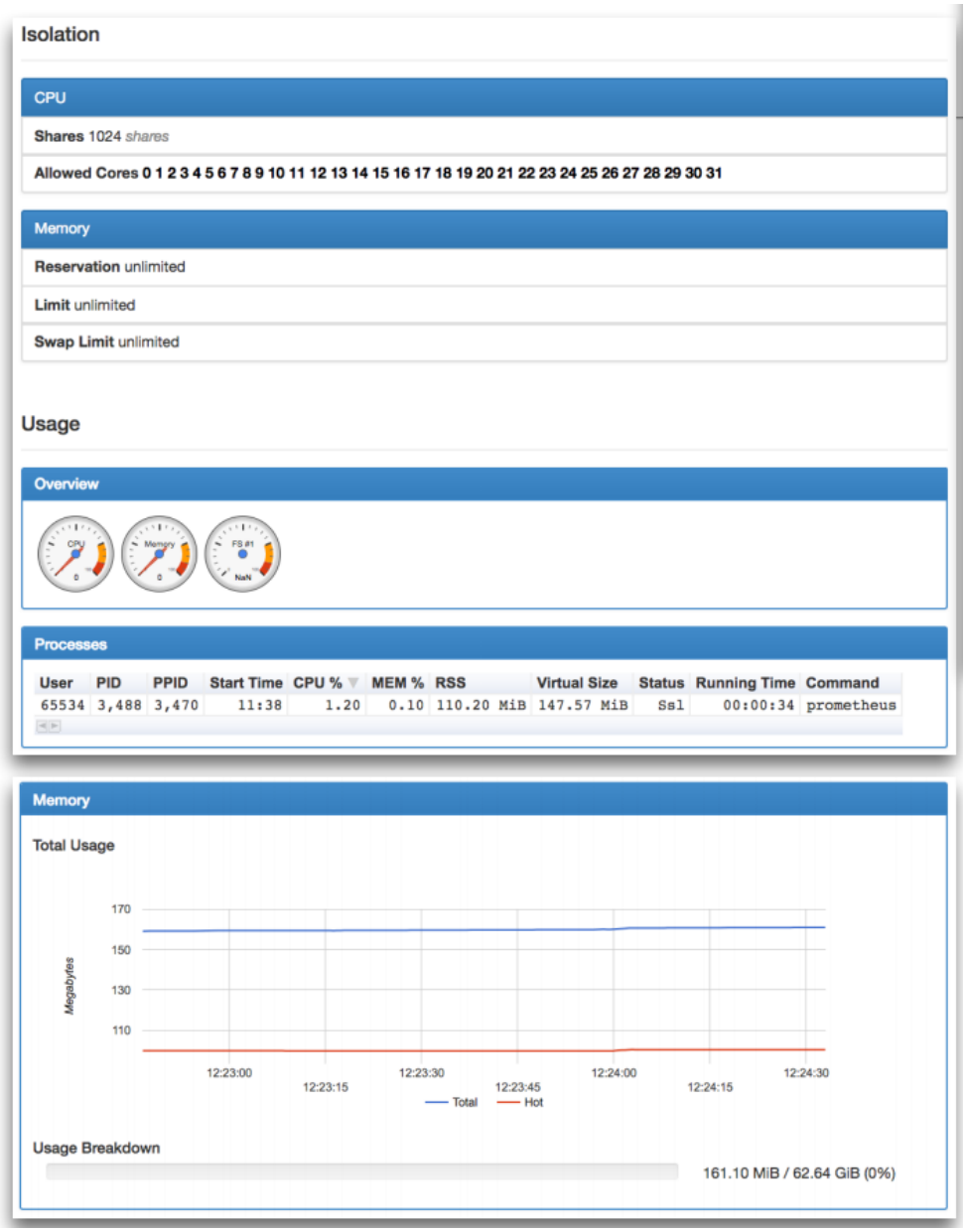


FIGURE 5-3: CADVISOR DASHBOARD

Table 5-1: All Metrics cAdvisor can expose to Prometheus

All Metrics cAdvisor is able to export to Prometheus	
container_cpu_system_seconds_total	container_memory_swap
container_cpu_usage_seconds_total	container_memory_usage_bytes
container_cpu_user_seconds_total	container_memory_working_set_bytes
container_fs_inodes_free	container_network_receive_bytes_total
container_fs_inodes_total	container_network_receive_errors_total
container_fs_io_current	container_network_receive_packets_dropped_total
container_fs_io_time_seconds_total	container_network_receive_packets_total
container_fs_io_time_weighted_seconds_total	container_network_transmit_bytes_total

container_fs_limit_bytes	container_network_transmit_errors_total
container_fs_read_seconds_total	container_network_transmit_packets_dropped_total
container_fs_reads_merged_total	
container_fs_reads_total	container_network_transmit_packets_total
container_fs_sector_reads_total	container_scrape_error
container_fs_sector_writes_total	container_spec_cpu_period
container_fs_usage_bytes	container_spec_cpu_shares
container_fs_write_seconds_total	container_fs_writes_total
container_fs_writes_merged_total	container_last_seen
container_spec_memory_limit_bytes	container_memory_cache
container_spec_memory_swap_limit_bytes	container_memory_failcnt
container_start_time_seconds	container_memory_failures_total
container_tasks_state	

Prometheus configuration allows the monitoring administrator or external system to specify the scrapping rules, which should be applied to collect the metrics. Such configuration is expressed in YAML format, which was designed to serialize data in a human readable way. Prometheus configuration (prometheus.yml). Figure 5-4 shows an example which specifies scraping rules for two different jobs, the *targets* parameter defines the endpoint at which Prometheus should query in order to retrieve the data. Additionally, scraping sources can be the following: Azure[29], Consul[30], AWS EC2[31], Openstack[32], GKE[33], Kubernetes[34], Marathon[35], Triton[36].

```
0 global:
1   scrape_interval: 15s
2
3 scrape_configs:
4   - job_name: 'prometheus'
5     scrape_interval: 5s
6     static_configs:
7       - targets: ['localhost:9090']
8
9   - job_name: 'cadvisor'
10    scrape_interval: 5s
11    static_configs:
12      - targets: ['cadvisor:8080']
~
```

FIGURE 5-4: PROMETHEUS CONFIGURATION (PROMETHEUS.YML)

Prometheus offers the following instrumentation features:

- Client Libraries: Before you can monitor your services, you need to add instrumentation to their code via one of the Prometheus client libraries. These implement the Prometheus metrics types
- Pushing Metrics: Monitor metrics which normally cannot be scrapped.
- Exporters and Integrations: Number of libraries and servers which help in exporting existing metrics from third-party systems as Prometheus metrics. It can also monitor services on MQTT broker[11].

Grafana is one of the preferred tools to visualize the data and metrics collected in Prometheus time series database. Grafana is an open-source platform for data analytics and data visualization, which is capable of reading data from multiple sources including Prometheus. Figure 5-5, an example of a Grafana dashboard is shown.

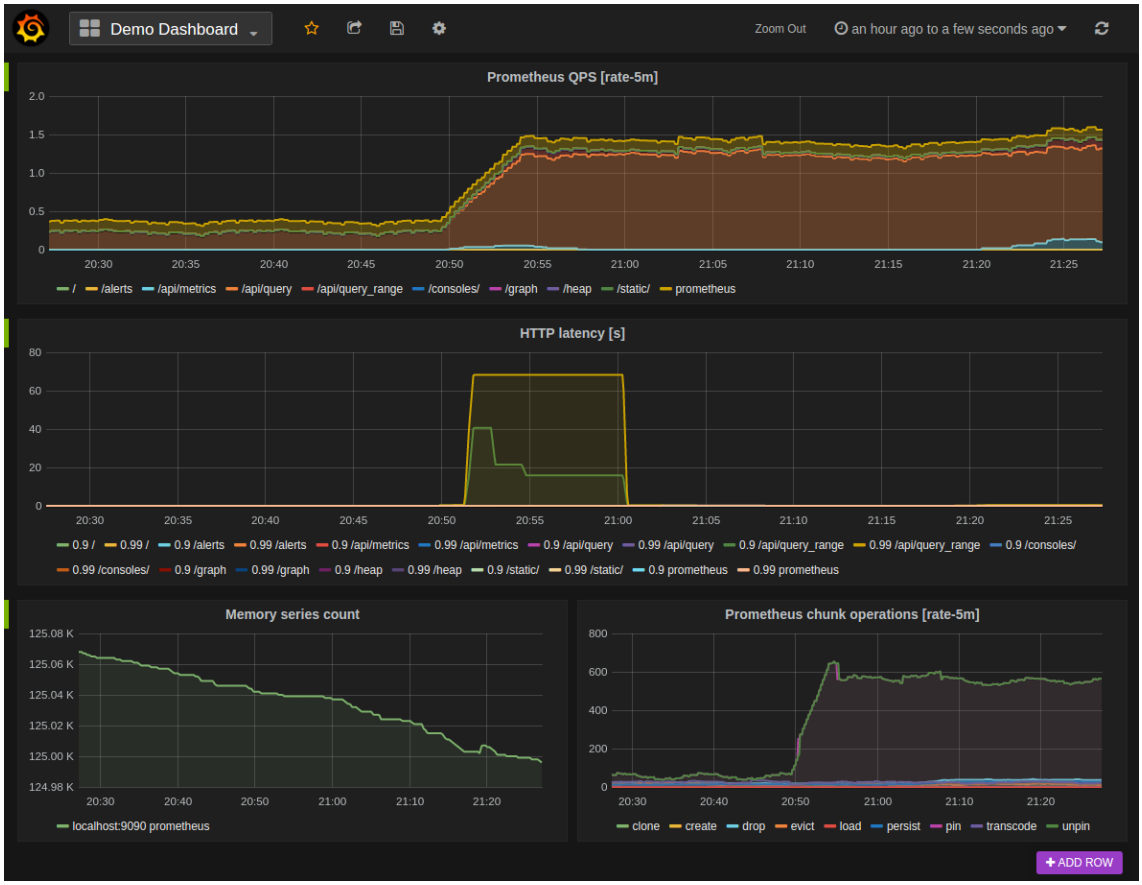


FIGURE 5-5: GRAFANA DASHBOARD EXAMPLE

5.3 EFS monitoring experimentation with Prometheus

Due to its widespread use both in academia and industry, Prometheus was adopted by 5G-CORAL as a monitoring solution. WP2 integrated Prometheus into the testbed setup of Figure 5-6, in order to measure the impact monitoring on the EFS resources. The Prometheus node-exporter provides a mechanism of exposing hardware and operating system metrics to the Prometheus platform by pulling data and metrics from the endpoints defined on each device.

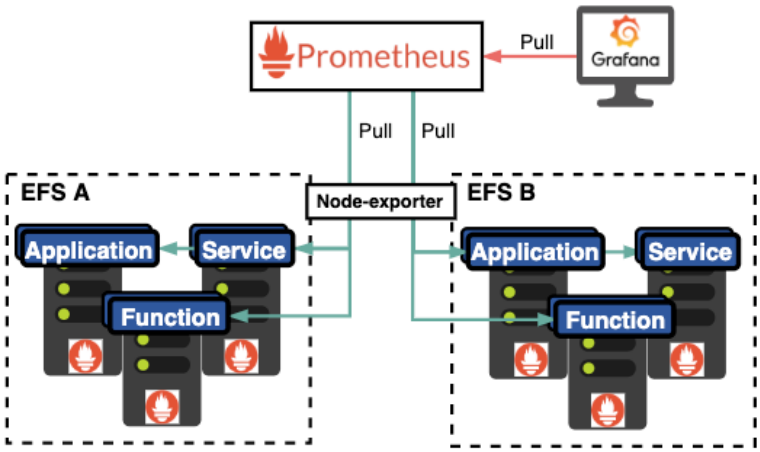


FIGURE 5-6: PROMETHEUS AND 5G-CORAL

5.3.1 Experiment I: EFS resource as virtual machine

To execute the first experiment, a virtual machine with 2GB of RAM and 25GB of disk memory was deployed, which represents a virtual EFS resource that could be instantiated at the cloud or edge. In that virtual machine, a group of seven LXC containers are deployed, six of them were used to extract monitoring data and metrics while the seventh was used to run the Prometheus monitoring platform. Additionally, a Grafana instance was run natively in the virtual machine hosting the LXC containers.

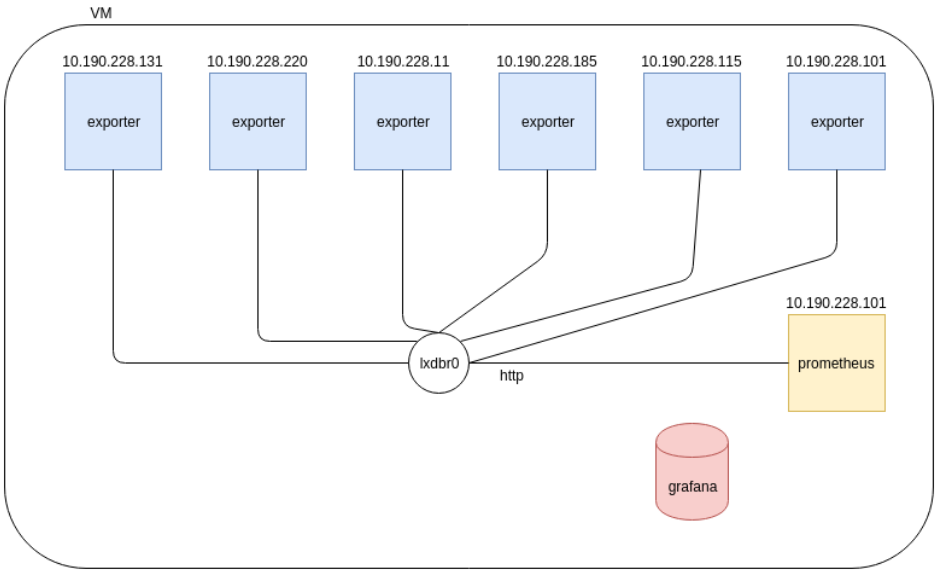


FIGURE 5-7: FOG NODE AS VIRTUAL MACHINE EXPERIMENT

During the experiment, Prometheus collected metrics from each LXC container, by running a node exporter process within the containers; that gathered and exposed CPU, RAM and bandwidth metrics. Grafana was used as the analytics and monitoring interface for the collected metrics. Prometheus was configured to collect metrics at a frequency of 15 seconds, i.e. the default configuration value.

During the experiment, CPU, RAM and bandwidth metrics of the virtual machine were measured with mpstat, free and ifstat commands, integrated in a bash script to automate the process. The experiment started by measuring these metrics without running the node exporter(s), Prometheus server or the Grafana service stopped. After the first minute, Prometheus, Grafana, and the first node exporter were instantiated. Then subsequently, every minute a new node exporter process was started in a new container, until all containers were running one instance of the node exporter.

Figure 5-8 presents the user (USR) and system (SYS) spaces CPU percentage of time spent. From the results it was noted that the highest peak of CPU consumption was located during the instantiation process of Prometheus, Grafana and the first container. After the first peak there were small peaks representing the instantiation of the monitored containers one per minute. Finally, the lowest blue peaks, in between the medium blue peaks, represent the node exporter sending the metrics that Prometheus was polling.

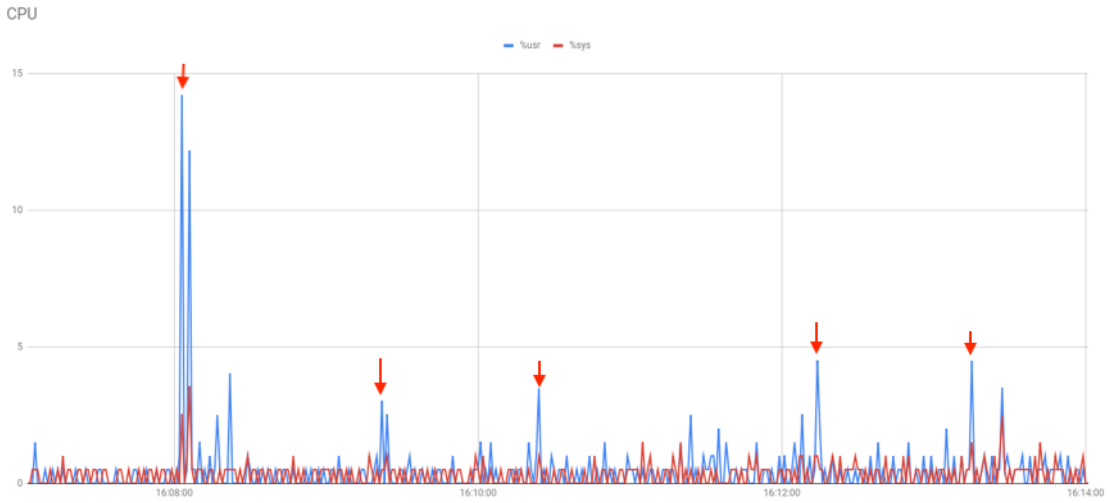


FIGURE 5-8: CPU % OF TIME SPENT IN USR(USER) AND SYS(SYSTEM) SPACES

Figure 5-9 presents the RAM consumption of the machine during the whole experiment, we noticed that RAM usage increased smoothly during the experiment; which leads us to interpret that Prometheus was storing collected data samples in RAM memory. The impact of Prometheus server and the node exporter(s) was minimal, i.e. RAM consumption increased by 150MB during the whole experiment.

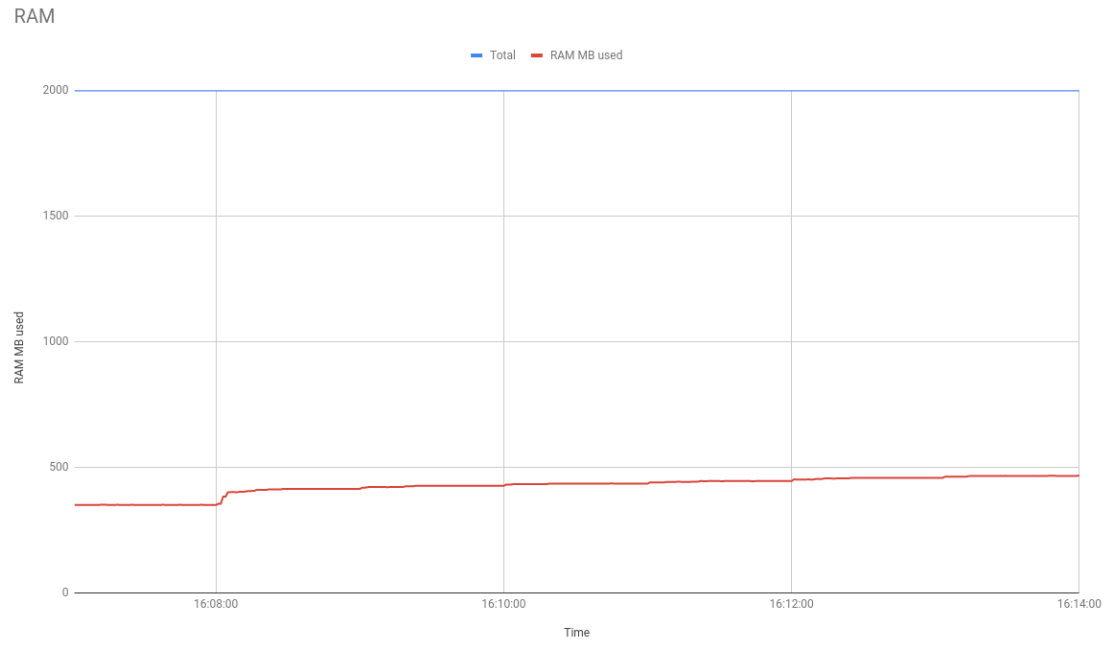


FIGURE 5-9: RAM CONSUMPTION

Finally, Figure 5-10 presents the bandwidth consumption in KB/s at the aggregation bridge interface (lxdbr0). From the results we observed six clear peaks with more or less the same height of 110KB/s. These bandwidth peaks correspond to the metrics extraction procedure of Prometheus to each of the node-exporters in every container. These results indicate that the impact of this monitoring system is quite low in terms of bandwidth consumption, as only some few hundreds of KBs where transferred through the network.

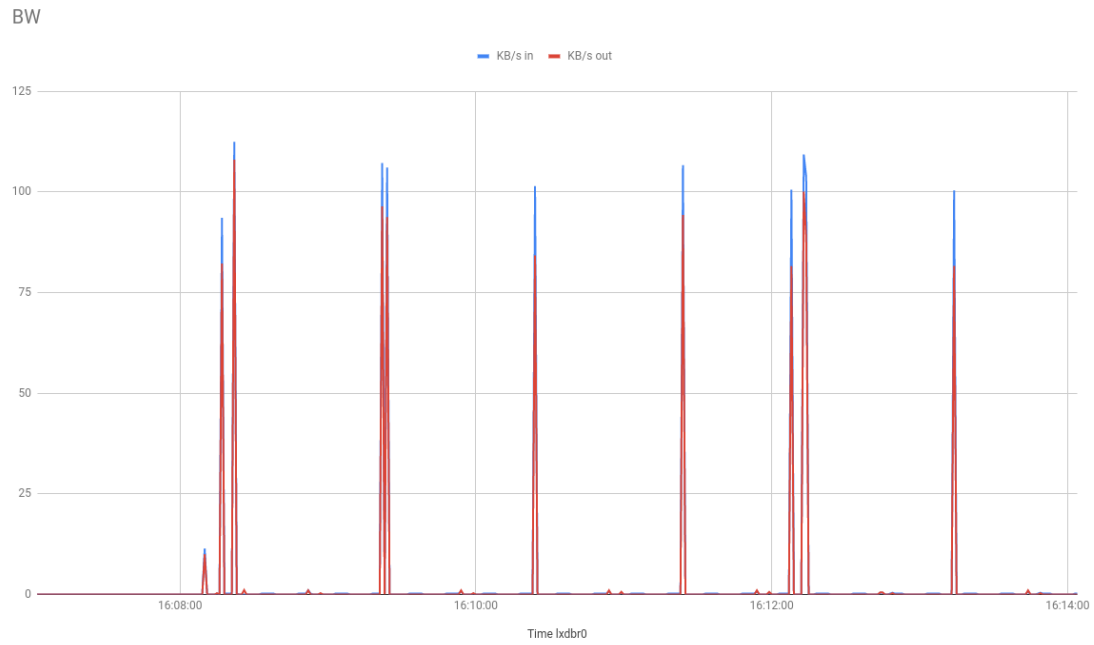


FIGURE 5-10: BANDWIDTH CONSUMPTION IN LXDBR0 INTERFACE

5.3.2 Experiment II: EFS resource as a real physical fog node device

This experiment replicated the experiment described in section 5.3.1 while substituting the host virtual machine physical fog node. Figure 5-11 shows the selected fog node that has the following specification: OS: Ubuntu Server 16.04 LTS; CPU: Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz; RAM: 4GB; and Disk: 8GB SSD.

FIGURE 5-11: QOTOM MINI PC

From Figure 5-12, we observed the biggest CPU peak at the start of the experiment, which represents the deployment and instantiation of the containers, Prometheus server and Grafana. Additionally, we noticed some more peaks (> 2% CPU User usage) that represent the starting of the node exporter process, and sequential polling of Prometheus to every deployed container.

Furthermore, in Figure 5-12 we present the CPU's percentage idle time that allows for a better comparison and contrasting of the impact of Prometheus monitoring on CPU usage. From the results we observed that CPU utilization was highest during the instantiation phase (~10% of CPU time), and dropped to less than 5% over the duration of the experiment.

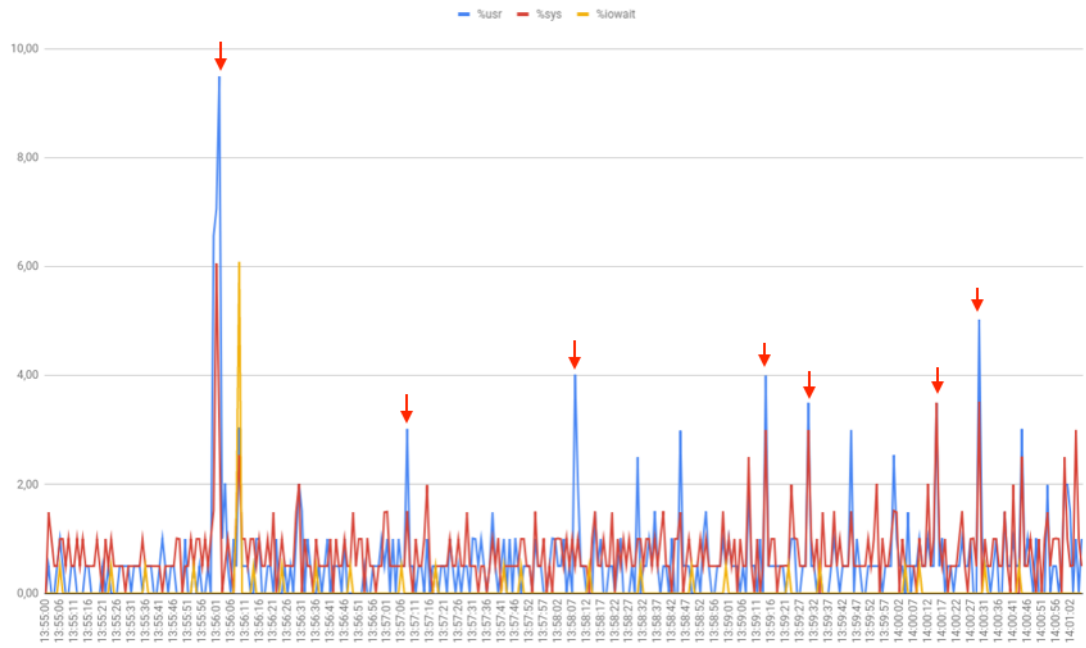


FIGURE 5-12: CPU USAGE FOR USR, SYS AND IOWAIT

Figure 5-13 presents the RAM usage during the experiment. From the results we observed that the RAM usage was less than 100MB, over the duration of the experiment. The minor fluctuations in RAM usage were attributed to RAM being consumed and subsequently released during the experiment.

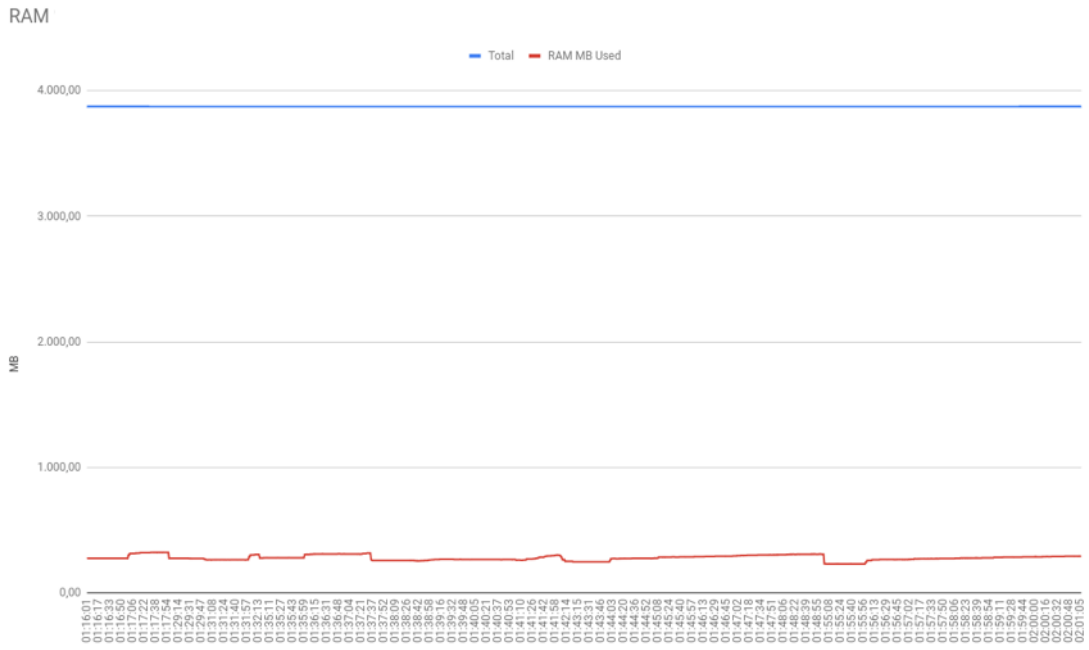


FIGURE 5-13: RAM USAGE

Figure 5-14 shows the results obtained from the disk usage during the experiment. Two snapshots were taken, one at the start of the experiment and another one at the end. We can see a consumption of 2% in /dev/sda2 partition, which totals 120MB of disk usage throughout the experiment.

```
sdn@fog:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            1,9G     0  1,9G   0% /dev
tmpfs           388M  6,0M  382M   2% /run
/dev/sda2       5,8G  4,4G  1,1G  81% /
tmpfs           1,9G     0  1,9G   0% /dev/shm
tmpfs           5,0M     0  5,0M   0% /run/lock
tmpfs           1,9G     0  1,9G   0% /sys/fs/cgroup
/dev/sda1       511M  3,6M  508M   1% /boot/efi
tmpfs           388M     0  388M   0% /run/user/1000
tmpfs           100K     0  100K   0% /var/lib/lxd/shmounts
tmpfs           100K     0  100K   0% /var/lib/lxd/devlxd
sdn@fog:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            1,9G     0  1,9G   0% /dev
tmpfs           388M  6,0M  382M   2% /run
/dev/sda2       5,8G  4,6G  982M  83% /
tmpfs           1,9G     0  1,9G   0% /dev/shm
tmpfs           5,0M     0  5,0M   0% /run/lock
tmpfs           1,9G     0  1,9G   0% /sys/fs/cgroup
/dev/sda1       511M  3,6M  508M   1% /boot/efi
tmpfs           388M     0  388M   0% /run/user/1000
tmpfs           100K     0  100K   0% /var/lib/lxd/shmounts
tmpfs           100K     0  100K   0% /var/lib/lxd/devlxd
```

FIGURE 5-14: SNAPSHOT OF DISK USAGE BEFORE AND AFTER THE EXPERIMENT

Finally, we performed bandwidth measurements at all network interfaces of the fog node. According to Figure 5-15, the highest bandwidth measurement was observed when Grafana, Prometheus and all monitoring containers were started. During the course of the experiment it was observed that Prometheus monitoring utilized a bandwidth of approximately 60 Kbps.

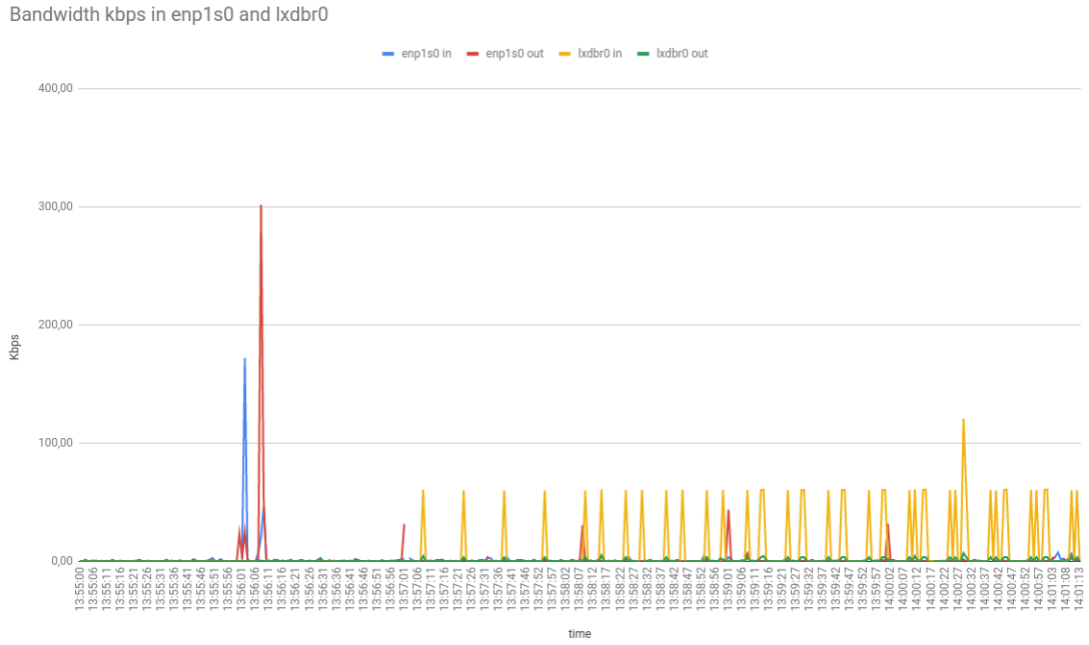


FIGURE 5-15: BANDWIDTH MEASUREMENT IN ALL INTERFACES OF THE PHYSICAL FOG NODE

In conclusion the experiments and results presented in section 5.3.1 and section 5.3.2 revealed that the integration of resource monitoring framework(s), such as Prometheus, into the EFS; had minimal impact on the EFS resources (compute, storage and networking).

6 Conclusions and Future Work

This deliverable, D2.2, is the final deliverable of WP2 with focus on the refined EFS design of 5G-CORAL, following on the baseline design reported in the first deliverable D2.1.

First, Section 2 and 3 address some key aspects in the EFS architecture and its reference design. Section 2 mainly addresses the EFS interfaces, data models and the EFS workflow, while Section 3 provides an extensive analysis of EFS service messaging protocols. The main contributions of Section 2 and 3 are listed as follows:

- Provided the definitions of the EFS internal and external interfaces: Ex interfaces (E1-E4) are the EFS internal interfaces connecting different EFS elements defined in the architecture. Especially E2 interface provides the connectivity enabling distributing and sharing service data between EFS functions and EFS applications via EFS service platform. Ox interfaces (O1, O5 and O6) connect EFS to OCS. Tx interfaces (T1, T3 and T8) connect EFS to OSS/BSS and non-EFS resources. To maximize the architectural compatibility to ETSI MEC and ETSI NFV, E2, E3, E4 and T1 interfaces are defined to be compatible with some reference interfaces defined in ETSI MEC while E1, O1, O5 and O6 are defined to be compatible with some reference interfaces defined in ETSI NFV. For example, for the E2 interface, we define how the ETSI MEC Mp1 interface is used and extended by the EFS. The T3 and T8 interfaces are the new interfaces not scoped in ETSI MEC and ETSI NFV.
- Refined the EFS service platform reference design from D2.1: Considering the needs of 5G-CORAL and compatibility to ETSI MEC, we adopted the ETSI MEC approach using both REST-based API and MQTT brokers. The REST-based API is used for registering and finding services, while MQTT is used for E2 interface as the transport for EFS services. It should be noted that MQTT is given as one example for a reference design. Any other systems like Zenoh, NATS, DDS, etc. can be supported. In addition, data structures associated with EFS service operations are provided. As examples, we also presented the JSON-based data models used in some 5G-CORAL PoC testbeds (i.e. connected cars and robotics).
- Investigated and benchmarked additional EFS service messaging protocols: Extensive experimental investigations have been performed to compare different alternatives, namely Zenoh, NATS, DDS, MQTT, and Kafka REST. The results showed that Zenoh and NATS outperform other protocols. These two are recommended to consider where high performance is needed.

To investigate the feasibility of EFS implementation in real scenarios, verify the EFS reference design and showcase the benefits of EFS, PoC prototypes have been implemented for seven different use cases. In Section 4, use-case specific implementations are described to verify the EFS functionality, and performance is evaluated in each use case through experiments. By adopting the 5G-CORAL design, the experiment results showed clearly the benefits in service delivery, computation offload, bandwidth reduction, and improved multi-RAT support. The following summarizes the implementation insights and findings from each use case:

- Robotics: This use case focuses on moving the robot control to the network side (locating it at the Fog, close to the robots) in the form of EFS applications, which would reduce the robot costs and enable the possibility to coordinate multiple robots. The Edge/Fog assisted robotics system has been designed blending together the Robot Operating System (ROS) that offers a common development framework for robotics applications and the 5G-CORAL EFS platform. In addition, close-loop control of the robot with low latency bounds can be achieved by moving the control along the trajectory of the robot and considering the signal level of the wireless network connecting the robot and the infrastructure. The

experimental results showed that a smart navigation application leveraging the EFS service of Wi-Fi RSSI can smoothen the robot movement at a high speed.

- Virtual Reality: In this use case, the 5G-CORAL solution decomposes the end-to-end 360° video streaming service into micro-services which are then distributed across three computing tiers, namely cloud, edge, and fog. All three compute tiers, composed of heterogeneous computing resources, are orchestrated and controlled using a unified orchestration and control system (OCS) based on Fog05. Experimental results showed that our approach can alleviate the footprint of the 360° video delivery service on a cloud data centre by reducing the GPU load, the consumed power and the memory usage, as well as saving the transport bandwidth by viewport adaptation.
- Augmented Reality: In this use case, EFS architecture has been adopted to realize AR navigation application where the image recognition application is executed on the fog nodes on the network side. The processing was distributed dynamically among multiple fog nodes. iBeacon-based localization was used to reduce the image recognition processing requirement. The experimental results proved that the edge and fog computing architecture is possible to meet the requirements of AR-based navigation application, in terms of latency and processing offload requirements.
- Multi-RAT IoT: The focus of this use case is on cloudifying the IoT communication stacks, like IEEE 802.15.4, NB-IoT and LoRa, in the EFS. We have addressed two key issues regarding efficient RH-Edge interface design and multi-channel transceiver implementation, which are crucial for cloudifying multi-RAT communication stacks. The experimental results regarding latency and network capacity required for fronthauling with 802.15.4 and NB-IoT proved the feasibility of implementing this use case following the 5G-CORAL concept and architecture.
- Connected Car: In this use case, the application is divided into two main components. The first part is responsible for generating the telemetry messages (CAMs) and publishing them to the EFS service platform. The second part takes care of receiving the telemetry messages from the other vehicles nearby and performs the processing needed for collision detection and, in that case, generate and publish a warning (DENMs) message for the position where the car is located. Both Wi-Fi and LTE can be used simultaneously. The experimental results demonstrated achievable low latency figures meeting the requirements set for this use case.
- SD-WAN: This use case focuses on EFS federation. It analyses how two domains can be federated to locate/offload functions, and applications close to the end user. The experimental results showed the feasibility of the proposed EFS federation.
- High-Speed Train: In moving infrastructure scenarios such as train network, the proposed two-hop architecture is adopted to improve on-board user experience by reducing the interaction with on-land base stations. Furthermore, the EFS implementation leveraging edge clouds and virtualization technologies can be utilized to bring services closer to the traveling users. The experimental results showed that the proposed schemes can reduce the control signaling by 50% when compared to the state-of-the-art.

Furthermore, EFS resource monitoring is another important topic addressed in Section 5. An open-source monitoring tool called Prometheus is proposed to fit to the EFS design. Two emulation tests have been done with virtualized fog nodes and real fog nodes, respectively. The experimental results showed that the CPU and memory usages are sufficiently low to be used for constrained devices, i.e. fog nodes.

In WP2, we provided an EFS reference design following the 5G-CORAL architecture defined in WP1. The design enables the virtualization of EFS functions and applications on EFS resources and facilitate sharing the context information as EFS services. PoC testbeds for different use cases have

been developed and small-scale measurements have been done, which have verified the feasibility of the EFS design developed in this project. In the future, more EFS related studies and research works will be performed based on the results achieved in 5G-CORAL. Hereby, we conclude this deliverable by highlighting two future research directions in particular:

1. Investigation of a large-scale EFS deployment integrating multiple use cases running on the same EFS, which is closer to real business deployment.
2. Incorporating the capabilities of machine learning, AI techniques and data handling into the EFS, as well as the interactions and extensions with Cloud. This would require a further extension of the EFS design and make the EFS more intelligent and optimized.

Last but not least, there are also a lot of possibilities to further improve the implementation of each use case presented in Section 4 in this deliverable. More details regarding each use case has been provided in Section 4, respectively.

Bibliography

[1] D2.1: Initial design of 5G-CORAL Edge and Fog computing system, 5G CORAL Project, June 2018.
[2] ETSI, "Mobile Edge Computing (MEC); Framework and Reference Architecture," European Telecommunications Standards Institute, GS MEC 003, Mar 2016.
[3] ETSI, "Multi-access Edge Computing (MEC); General principles for MEC Service APIs," European Telecommunications Standards Institute, GS MEC 009, Jan 2019.
[4] ETSI, "Mobile Edge Computing (MEC); Mobile Edge Platform Application Enablement," European Telecommunications Standards Institute, GS MEC 011, Jul 2017.
[5] D3.1: Initial design of 5G-CORAL orchestration and control system, 5G CORAL Project, June 2018.
[6] ETSI GS NFV-IFA 008 V3.2.1, April 2019
[7] https://psutil.readthedocs.io/en/latest/ *(last accessed 2019/05/10)*
[8] https://github.com/prometheus/node_exporter *(last accessed 2019/05/10)*
[9] https://docs.docker.com/engine/api/v1.37/#operation/ContainerInspect *(last accessed 2019/05/10)*
[10] https://docs.docker.com/engine/api/v1.37/#operation/ContainerStats *(last accessed 2019/05/10)*
[11] https://prometheus.io/docs/instrumenting/exporters/ *(last accessed 2019/05/10)*
[12] http://cassandra.apache.org/ *(last accessed 2019/05/06)*
[13] https://cloud.google.com/bigtable/ (last accessed 2019/05/06)
[14] http://druid.io/ (last accessed 2019/05/06)
[15] https://www.mongodb.com/ (last accessed 2019/05/06)
[16] https://www.project-voldemort.com/voldemort/ (last accessed 2019/05/06)
[17] http://zenoh.io/download/pdf/zenoh.pdf (last accessed 2019/05/10)
[18] https://tools.ietf.org/html/rfc3986 (last accessed 2019/05/10)
[19] https://github.com/confluentinc/kafka-rest (last accessed 2019/05/10)
[20] https://docs.confluent.io/current/connect/references/restapi.html *(last accessed 2019/05/10)*
[21] https://hackernoon.com/supercharging-kafka-enable-realtime-web-streaming-by-adding-pushpin-fd62a9809d94 *(last accessed 2019/05/10)*
[22] http://www.ros.org/ *(last accessed 2019/05/10)*
[23] https://www.wowza.com/products/streaming-engine
[24] O. Liberg, M. Sundberg, E. Wang, J. Bergman and J. Sachs, *Cellular Internet of Things: Technologies, Standards, and Performance*, Academic Press, 2018
[25] 3GPP, "Study on new radio access technology; Radio access architecture and interfaces (Release 14)", TR 38.801, March 2017.
[26] D4.1 5G-CORAL testbed definition, integration and demonstration plans, 5G CORAL Project.
[27] ETSI EN 302 637-2 v1.3.1, September 2014
[28] ETSI EN 302 637-3 v1.2.1, September 2014
[29] https://azure.microsoft.com/en-us/
[30] https://www.consul.io/
[31] https://aws.amazon.com/ec2/?nc1=f_ls
[32] https://www.openstack.org/
[33] https://cloud.google.com/container-engine/?hl=es
[34] https://kubernetes.io/
[35] https://mesosphere.github.io/marathon/
[36] https://www.joyent.com/
[37] https://kafka.apache.org/documentation/
[38] https://softwaremill.com/kafka-with-selective-acknowledgments-performance/
[39] https://snapshot.raintank.io/dashboard/snapshot/ZNHUfxvjLWF1tG8KR4vFsHttBZF037o0?orgId=2

[40] https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines
[41] D3.2: Refined design of 5G-CORAL orchestration and control system and future directions, 5G CORAL Project, June 2019
[42] D1.1 5G CORAL Initial system design, use cases, and requirements. 5G CORAL Project.
[43] What is the Confluent Platform?: https://docs.confluent.io/1.0/platform.html *(last accessed 2019/05/30)*
[44] https://softwaremill.com/mqperf/
[45] https://snapshot.raintank.io/dashboard/snapshot/WKBeTn44tM8J71GCJW4KGSfv7bMASMLR?orgId=2

7 Appendix: PoC service data models

This appendix contains data models specified in JSON schemas for the messages used in the EFS services provided by some of the PoCs.

7.1 Connected cars

The Connected cars PoC make use of the CAM and DENM messages defined in ETSI EN 302 637-2 and ETSI EN 302 637-3.

7.1.1 CAM – Cooperative Awareness Message

```
{
  "$schema": "http://json-schema.org/schema#",
  "id": "jsonschema://it.azcom.RoadSafetyClient.schemas.CAM.json",
  "title": "CAM",
  "description": "CAM JSON representation",
  "version": "1",
  "type": "object",
  "properties": {
    "header": {
      "type": "object",
      "properties": {
        "protocolVersion": {
          "type": "number",
          "enum": [1]
        },
        "messageID": {
          "type": "number",
          "enum": [2],
          "description": "Accept ONLY 2, i.e. CAM"
        },
        "stationID": {
          "type": "number",
          "minimum": 0,
          "maximum": 4294967295
        }
      },
      "required": ["protocolVersion", "messageID", "stationID"]
    },
    "cam": {
      "type": "object",
      "properties": {
        "generationDeltaTime": {
          "type": "number",
          "minimum": 0,
          "maximum": 65535,
          "description": "Time corresponding to the time of the reference position in
the CAM, considered as time of the CAM generation. The value of the DE shall be wrapped
to 65 536. This value shall be set as the remainder of the corresponding value of
TimestampIts divided by 65 536 as below:\ngenerationDeltaTime = TimestampIts mod 65
536\nTimestampIts represents an integer value in milliseconds since 2004-01-
01T00:00:00:000Z as defined in ETSI TS 102 894-2 [2]"
        },
        "camParameters": {
          "type": "object",
          "properties": {
            "basicContainer": {
              "type": "object",
              "properties": {
                "stationType": {
                  "type": "number",
                  "enum": [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 15],
                  "description":
"0=Unknown\n1=Pedestrian\n2=Cyclist\n3=Moped\n4=Motorcycle\n5=Passenger
car\n6=Bus\n7=Light truck\n8=Heavy truck\n9=Trailer\n10=Special
vehicles\n11=Tram\n15=Road Side Unit"
                },
                "referencePosition": {
                  "type": "object",
                  "properties": {
```

```
                    "latitude": {
                      "type": "number",
                      "description": "Unit: 0,1 microdegree\n900000001 if unavailable",
                      "minimum": -900000000,
                      "maximum": 900000001
                    },
                    "longitude": {
                      "type": "number",
                      "description": "Unit: 0,1 microdegree\n1800000001 if unavailable",
                      "minimum": -1800000000,
                      "maximum": 1800000001
                    },
                    "positionConfidenceEllipse": {
                      "type": "object",
                      "properties": {
                        "semiMajorConfidence": {
                          "type": "number",
                          "description": "Unit: 1 centimetre\n4095 if unavailable",
                          "minimum": 0,
                          "maximum": 4095
                        },
                        "semiMinorConfidence": {
                          "type": "number",
                          "description": "Unit: 1 centimetre\n4095 if unavailable",
                          "minimum": 0,
                          "maximum": 4095
                        },
                        "semiMajorOrientation": {
                          "type": "number",
                          "description": "Unit: 0,1 degree\n3601 if unavailable",
                          "minimum": 0,
                          "maximum": 3601
                        }
                      }
                    },
                    "altitude": {
                      "type": "object",
                      "properties": {
                        "altitudeValue": {
                          "type": "number",
                          "minimum": -100000,
                          "maximum": 800001
                        },
                        "altitudeConfidence": {
                          "type": "string",
                          "enum": ["alt-000-01", "alt-000-02", "alt-000-05", "alt-000-
10", "alt-000-20", "alt-000-50", "alt-001-00", "alt-002-00", "alt-005-00", "alt-010-00",
"alt-020-00", "alt-050-00", "alt-100-00", "alt-200-00", "outOfRange", "unavailable"]
                        }
                      }
                    }
                  },
                            "required": ["latitude", "longitude"]
                }
              },
                        "required": ["stationType", "referencePosition"]
            },
            "highFrequencyContainer": {
              "type": "object",
              "properties": {
                "basicVehicleContainerHighFrequency": {
                  "type": "object",
                  "properties": {
                    "heading": {
                      "type": "object",
                      "properties": {
                        "headingValue": {
                          "type": "number",
                          "minimum": 0,
                          "maximum": 3601,
                          "description": "Unit: 0,1 degree\n3601 if unavailable"
                        },
                        "headingConfidence": {
                          "type": "number",
```

```
                        "minimum": 1,
                        "maximum": 127,
                        "description": "Unit: 0,1 degree\n126 if out of range\n127 if
unavailable"
                      }
                    }
                  },
                  "speed": {
                    "type": "object",
                    "properties": {
                      "speedValue": {
                        "type": "number",
                        "minimum": 0,
                        "maximum": 16383,
                        "description": "Unit: 0,01 m/s\n16383 if unavailable"
                      },
                      "speedConfidence": {
                        "type": "number",
                        "minimum": 1,
                        "maximum": 127,
                        "description": "1 cm/s\n126 if out of range\n127 if
unavailable"
                      }
                    }
                  },
                  "driveDirection": {
                    "type": "string",
                    "enum": ["forward", "backward", "unavailable"]
                  },
                  "vehicleLength": {
                    "type": "object",
                    "properties": {
                      "vehicleLengthValue": {
                        "type": "number",
                        "minimum": 1,
                        "maximum": 1023,
                        "description": "0,1 metres\n1023 if unavailable"
                      },
                      "vehicleLengthConfidenceIndication": {
                        "type": "string",
                        "enum": ["noTrailerPresent", "trailerPresentWithKnownLength",
"trailerPresentWithUnknownLength", "trailerPresenceIsUnknown", "unavailable"]
                      }
                    }
                  },
                  "vehicleWidth": {
                    "type": "number",
                    "minimum": 1,
                    "maximum": 62,
                    "description": "0,1 metres\n62 if unavailable"
                  },
                  "longitudinalAcceleration": {
                    "type": "object",
                    "properties": {
                      "longitudinalAccelerationValue": {
                        "type": "number",
                        "minimum": -160,
                        "maximum": 161,
                        "description": "Unit: 0,1 m/s2\n161 if unavailable"
                      },
                      "longitudinalAccelerationConfidence": {
                        "type": "number",
                        "minimum": 0,
                        "maximum": 102,
                        "description": "Unit: 0,1 m/s2\n101 if out of range\n102 if
not available"
                      }
                    }
                  },
                  "curvature": {
                    "type": "object",
                    "properties": {
                      "curvatureValue": {
                        "type": "number",
```

```
                        "minimum": -30000,
                        "maximum": 30001,
                        "description": "1 over 30 000 metres\n30001 if unavailable"
                      },
                      "curvatureConfidence": {
                        "type": "string",
                        "enum": ["onePerMeter-0-00002", "onePerMeter-0-0001",
"onePerMeter-0-0005", "onePerMeter-0-002", "onePerMeter-0-01", "onePerMeter-0-1",
"outOfRange", "unavailable"]
                      }
                    }
                  },
                  "curvatureCalculationMode": {
                    "type": "string",
                    "enum": ["yawRateUsed", "yawRateNotUsed", "unavailable"]
                  },
                  "yawRate": {
                    "type": "object",
                    "properties": {
                      "yawRateValue": {
                        "type": "number",
                        "minimum": -32766,
                        "maximum": 32767,
                        "description": "0,01 degree per second\n32767 if unavailable"
                      },
                      "yawRateConfidence": {
                        "type": "string",
                        "enum": ["degSec-000-01", "degSec-000-05", "degSec-000-10",
"degSec-001-00", "degSec-005-00", "degSec-010-00", "degSec-100-00", "outOfRange",
"unavailable"]
                      }
                    }
                  },
                  "lateralAcceleration": {
                    "type": "object",
                    "properties": {
                      "lateralAccelerationValue": {
                        "type": "number",
                        "minimum": -160,
                        "maximum": 161,
                        "description": "Unit: 0,1 m/s2\n161 if unavailable"
                      },
                      "lateralAccelerationConfidence": {
                        "type": "number",
                        "minimum": 0,
                        "maximum": 102,
                        "description": "Unit: 0,1 m/s2\n101 if out of range\n102 if
not available"
                      }
                    }
                  },
                  "verticalAcceleration": {
                    "type": "object",
                    "properties": {
                      "verticalAccelerationValue": {
                        "type": "number",
                        "minimum": -160,
                        "maximum": 161,
                        "description": "Unit: 0,1 m/s2\n161 if unavailable"
                      },
                      "verticalAccelerationConfidence": {
                        "type": "number",
                        "minimum": 0,
                        "maximum": 102,
                        "description": "Unit: 0,1 m/s2\n101 if out of range\n102 if
not available"
                      }
                    }
                  }
                }
              }
            },
                      "required": ["basicVehicleContainerHighFrequency"]
          }
```

```
          },
                    "required": ["basicContainer", "highFrequencyContainer"]
        }
      },
            "required": ["generationDeltaTime", "camParameters"]
    }
  },
  "required": ["header", "cam"]
}
```

7.1.2 DENM – Decentralised Environmental Notification Message

```
{
  "$schema": "http://json-schema.org/schema#",
  "id": "jsonschema://it.azcom.RoadSafetyClient.schemas.DENM.json",
  "title": "DENM",
  "description": "DENM JSON representation",
  "version": "1",
  "type": "object",
  "properties": {
    "header": {
      "type": "object",
      "properties": {
        "protocolVersion": {
          "type": "number",
          "enum": [1]
        },
        "messageID": {
          "type": "number",
          "enum": [1],
          "description": "Accept ONLY 1, i.e. DENM"
        },
        "stationID": {
          "type": "number",
          "minimum": 0,
          "maximum": 4294967295
        }
      },
      "required": ["protocolVersion", "messageID", "stationID"]
    },
    "denm": {
      "type": "object",
      "properties":{
        "management": {
          "type": "object",
          "properties": {
            "actionID": {
              "type": "object",
              "properties": {
                "originatingStationID": {
                  "type": "number",
                  "minimum": 0,
                  "maximum": 4294967295
                },
                "sequenceNumber": {
                  "type": "number",
                  "minimum": 0,
                  "maximum": 65535
                }
              }
            },
            "detectionTime": {
              "type": "number",
              "minimum": 0,
              "maximum": 4398046511103
            },
            "referenceTime": {
              "type": "number",
              "minimum": 0,
              "maximum": 4398046511103
            },
            "termination": {
              "type": "string",
              "enum": ["isCancellation", "isNegation"]
```

```
            },
            "eventPosition": {
              "type": "object",
              "properties": {
                "latitude": {
                  "type": "number",
                  "description": "Unit: 0,1 microdegree\n900000001 if unavailable",
                  "minimum": -900000000,
                  "maximum": 900000001
                },
                "longitude": {
                  "type": "number",
                  "description": "Unit: 0,1 microdegree\n1800000001 if unavailable",
                  "minimum": -1800000000,
                  "maximum": 1800000001
                },
                "positionConfidenceEllipse": {
                  "type": "object",
                  "properties": {
                    "semiMajorConfidence": {
                      "type": "number",
                      "description": "Unit: 1 centimetre\n4095 if unavailable",
                      "minimum": 0,
                      "maximum": 4095
                    },
                    "semiMinorConfidence": {
                      "type": "number",
                      "description": "Unit: 1 centimetre\n4095 if unavailable",
                      "minimum": 0,
                      "maximum": 4095
                    },
                    "semiMajorOrientation": {
                      "type": "number",
                      "description": "Unit: 0,1 degree\n3601 if unavailable",
                      "minimum": 0,
                      "maximum": 3601
                    }
                  }
                },
                "altitude": {
                  "type": "object",
                  "properties": {
                    "altitudeValue": {
                      "type": "number",
                      "minimum": -100000,
                      "maximum": 800001
                    },
                    "altitudeConfidence": {
                      "type": "string",
                      "enum": ["alt-000-01", "alt-000-02", "alt-000-05", "alt-000-10",
"alt-000-20", "alt-000-50", "alt-001-00", "alt-002-00", "alt-005-00", "alt-010-00",
"alt-020-00", "alt-050-00", "alt-100-00", "alt-200-00", "outOfRange", "unavailable"]
                    }
                  }
                }
              }
            },
            "relevanceDistance": {
              "type": "string",
              "enum": ["lessThan50m", "lessThan100m", "lessThan200m", "lessThan500m",
"lessThan1000m", "lessThan5km", "lessThan10km", "over10km"]
            },
            "relevanceTrafficDirection": {
              "type": "string",
              "enum": ["allTrafficDirections", "upstreamTraffic", "downstreamTraffic",
"oppositeTraffic"]
            },
            "validityDuration": {
              "type": "number",
              "description": "Unit: 1 second",
              "minimum": 0,
              "maximum": 86400
            },
            "transmissionInterval": {
```

```
              "type": "number",
              "description": "Unit: 1 Millisecond",
              "minimum": 0,
              "maximum": 10000
            },
            "stationType": {
              "type": "number",
              "enum": [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 15],
              "description":
"0=Unknown\n1=Pedestrian\n2=Cyclist\n3=Moped\n4=Motorcycle\n5=Passenger
car\n6=Bus\n7=Light truck\n8=Heavy truck\n9=Trailer\n10=Special
vehicles\n11=Tram\n15=Road Side Unit"
            }
          }
        },
        "situation": {
          "type": "object",
          "properties": {
            "informationQuality": {
              "type": "number",
              "minimum": 0,
              "maximum": 7,
              "description": "0=Unavailable\n1=Lowest\n7=Highest"
            },
            "eventType": {
              "type": "object",
              "properties": {
                "causeCode": {
                  "type": "number",
                  "enum": [0, 1, 2, 3, 6, 9, 10, 11, 12, 14, 15, 17, 18, 19, 26, 27, 91,
92, 93, 94, 95, 96, 97, 98, 99],
                  "description": "0=reserved\n1=traffic
condition\n2=accident\n3=roadworks\n6=Adverse Weather Condition Adhesion\n9=hazardous
Location Surface Condition\n10=hazardous Location Obstacle On The Road\n11=hazardous
Location Animal On The Road\n12=human Presence On The Road\n14=wrong Way
Driving\n15=rescue And Recovery Work In Progress\n17=adverse Weather Condition Extreme
Weather Condition\n18=adverse Weather Condition Visibility\n19=adverse Weather Condition
Precipitation\n26=slow Vehicle\n27=dangerous End Of Queue\n91=vehicle Breakdown\n92=post
Crash\n93=human Problem\n94=stationary Vehicle\n95=emergency Vehicle
Approaching\n96=hazardous Location Dangerous Curve\n97=collision Risk\n98=signal
Violation\n99=dangerous Situation"
                },
                "subCauseCode": {
                  "type": "number",
                  "minimum": 0,
                  "maximum": 255,
                  "description": "Type of sub cause of a detected event as defined in
ETSI EN 302 637-3 [i.3]"
                }
              }
            }
          }
        }
      },
      "required": ["management", "situation"]
    }
  },
  "required": ["header", "denm"]
}
```


7.1.3 OBU Configuration parameters

The connected car PoC defines OBU configuration parameters as shown in Table 7-1.

TABLE 7-1: OBU CONFIGURATION PARAMETER

Parameter name	Description
require_mobile_data_ connection	0 = No mobile data request is performed 1 = A mobile data connection is performed
wifi_enabled	0 = WiFi is disabled; 1 = WiFi is enabled
wifi_ssid	SSID of the WiFi AP to connect to

wifi_password	Password of the WiFi AP
mqtt_broker_host[]	Host of the MQTT Broker
mqtt_broker_port[]	Port of the MQTT Broker
mqtt_broker_username[]	MQTT Broker username
mqtt_broker_password[]	MQTT Broker password
mqtt_broker_use_ssl[]	0 = SSL is not used ; 1 = SSL is used when connecting to the MQTT Broker
mqtt_broker_cert_file[]	Path to the PEM certificate file
mqtt_default_topic_name[]	MQTT topic name
mqtt_default_qos[]	MQTT QOS
mqtt_keep_alive[]	MQTT keep alive
vehicle_length_cm	The vehicle length in centimeters
vehicle_width_cm	The vehicle width in centimeters
its_station_id	The ITS "stationID" field in the ITS PDU Header Min: 0, Max: 4294967295 Set to -1 to randomly generate a stationID
its_station_type	The "stationType" field in the ITS PDU Header 0 = unknown 1 = pedestrian 2 = cyclist 3 = moped 4 = motorcycle 5 = passengerCar 6 = bus 7 = lightTruck 8 = heavyTruck 9 = trailer 10= specialVehicles 11= tram 15= roadSideUnit
its_vehicle_length_confidence_indication	The vehicleLengthConfidenceIndication field inside the ITS VehicleLength 0 = noTrailerPresent 1 = trailerPresentWithKnownLength 2 = trailerPresentWithUnknownLength 3 = trailerPresenceIsUnknown 4 = unavailable
cam_max_generation_period_ms	The maximum time between the generation of two CAM messages (minimum rate)
cam_min_generation_period_ms	The minimum time between the generation of two CAM messages (maximum rate)
bt_enabled	Enable / disable the Bluetooth interface This interface is used to communicate with a BT device used as an HMI (such as a tablet) 1 = enabled 0 = disabled
bt_device_name	Set the name of the Bluetooth device. The following placeholders can be used: {{its_station_id}} - will be replaced by the value of the its_station_id configuration variable.
hmi_http_url	Set the URL where to make POST requests with the HMI JSON RPC data
hmi_http_user_agent	User agent used for the HTTP HMI interface

hmi_http_request_timeout_s	Timeout of the HTTP request
can_enabled	Enable / disable the OBD-II / CAN interface Note: the CAN interface HAS to be enabled even if loading OBD-II samples from the test vector. 1 = enabled 0 = disabled
can_bitrate	Bitrate of the CAN BUS 125000 or 500000
can_obdii_identifiers_bit_size	Size in bits of the CAN identifiers for the OBD The options are the following: 11: 11-bit CAN ID 29: 29-bit CAN ID
can_interface	CAN interface to be used for OBDII
can_high_freq_timer_period_ms	Acquisition rate of high frequency parameters from the OBD-II/CAN. e.g. vehicle speed.
can_read_vehicle_speed	Set to 1 to enable the reading of the vehicle speed. 0 otherwise.
can_read_mil_status	Set to 1 to enable the detection of the MIL (Malfunction Indicator Lamp). 0 otherwise.
messages_exchange_e2e_ latency_measurement_enable	Set to 1 to enable the E2E latency measurement of the message exchange. This will measure the time elapsed between the transmission of a message and its reception back on the same device. This does not include the application overhead, with the exception of the data exchanger (e.g. MQTT client) and a minimum overhead required to perform the measurement itself.
messages_exchange_e2e_ latency_measurement_logfile	The output file where to store the latency measurements The following placeholders can be used: {{its_station_id}} - will be replaced by the value of the its_station_id configuration variable {{current_timestamp}} - will be replaced by the value of the current unix timestamp
app_collision_avoidance_enable	Set to 1 if you want to enable the collision avoidance algorithm
app_collision_avoidance_ detection_radius_m	Radius in meters of the threshold for the collision avoidance. i.e. if two vehicles come this close, a DENM is generated
gnss_position_acquisition_rate_ms	GNSS position acquisition rate in milliseconds
gnss_minimum_h_accuracy_for_fix	Use horizontal accuracy threshold instead of GNSS fix state.

7.2 Edge robotics

The Edge robotics PoC defines one message for WiFi monitoring as follows.

```
{
    "$schema": "http://json-schema.org/schema#",
```

```
    "id": "jsonschema://eu.5g-coral.Robotics.schemas.wifimonitor.json",
    "title": "wifimonitor",
    "description": "5GCORAL Robotics WiFi monitor message",
    "version": "1",
    "type": "object",
    "properties": {
        "host": {
            "type": "string",
            "description": "<node_id> - string? integer? mac address?"
        },
        "timestamp": {
            "type": "number",
            "description": "Absolute time in Unix Epoch time?"
        },
        "station": {
            "type": "string",
            "description": "MAC address of access point (ESSID?)"
        },
        "channel": {
            "type": "integer",
            "description": "WiFi channel number"
        },
        "signal": {
            "type": "number",
            "description": "Signal quality (RSSI?) in dBm"
        },
        "datarate": {
            "type": "number",
            "description": "Connection data rate (Mbps)"
        }
    },
    "required": ["host", "timestamp", "station", "channel", "signal", "datarate"]
}
```

7.3 IBeacon

The AR PoC uses one message for iBeacon coordinates as follows.

```
{
    "$schema": "http://json-schema.org/schema#",
    "id": "jsonschema://eu.5g-coral..schemas.ibeacon.json",
    "title": "ibeacon",
    "description": "5GCORAL  ibeacon message",
    "version": "1",
    "type": "object",
    "properties": {
        "iBeacon_Addr": {
            "type": "string",
            "description": "MAC address"
        },
        "iBeacon_Region": {
            "type": "string",
            "description": "Region..."
        },
        "iBeacon_LiDAR_Coordinates": {
            "type": "object",
            "properties": {
                "X_Cod": {
                    "type": "integer"
                },
                "Y_Cod": {
                    "type": "integer"
                },
                "Z_Cod": {
                    "type": "integer"
                }
            },
            "description": "LiDAR coordinates...",
            "required": ["X_Cod", "Y_Cod", "Z_Cod"]
        },
        "required": ["iBeacon_Addr", "iBeacon_Region", "iBeacon_LiDAR_Coordinates"]
    }
}
```

8 Appendix: Survey and analysis of SoA monitoring frameworks.

In this section, we identify and discuss available monitoring tools that will be adopted in 5G-CORAL. We can classify monitoring in three different categories: host monitoring, virtual machine monitoring and container monitoring. In the following subsections, we describe each of these categories in more details.

8.1 Host Monitoring

Python system and process utilities (*psutil* [7]), is a well-known tool to monitor hosts. It is a cross-platform library for retrieving information on running processes and system utilization in Python. It is useful mainly for system monitoring, profiling, limitation of process resources and the management of running processes. It implements many functionalities offered by UNIX command line tools such as: ps, top, lsof, netstat, ifconfig, who, df, kill, free, nice, ionice, iostat, iotop, uptime, pidof, tty, taskset or pmap.

psutil, supports all well-known operating systems, such as Linux, Windows, macOS, FreeBSD/OpenBSD/NetBSD, Sun Solaris, AIX. Additionally, *psutil* can retrieve information about CPU, memory, disks, network, sensors, processes, users or boot time. Below, several examples to execute this tool in a terminal are shown:

CPU times:

```
>>> import psutil
>>> psutil.cpu_times()
scputimes(user=17411.7, nice=77.99, system=3797.02, idle=51266.57,
iowait=732.58, irq=0.01, softirq=142.43, steal=0.0, guest=0.0, guest_nice=0.0)
```

CPU percentage:

```
>>> import psutil
>>> # blocking
>>> psutil.cpu_percent(interval=1)
2.0
>>> # non-blocking (percentage since last call)
>>> psutil.cpu_percent(interval=None)
2.9
>>> # blocking, per-cpu
>>> psutil.cpu_percent(interval=1, percpu=True)
[2.0, 1.0]
```

Memory:

```
>>> import psutil
>>> mem = psutil.virtual_memory()
>>> mem
svmem(total=10367352832, available=6472179712, percent=37.6, used=8186245120,
free=2181107712, active=4748992512, inactive=2758115328, buffers=790724608,
cached=3500347392, shared=787554304, slab=199348224)
```

Network interfaces:

```
>>> import psutil
>>> psutil.net_if_addrs()
{'lo': [snicaddr(family=<AddressFamily.AF_INET: 2>, address='127.0.0.1',
netmask='255.0.0.0', broadcast='127.0.0.1', ptp=None),
```

```
snicaddr(family=<AddressFamily.AF_INET6: 10>, address='::1',
netmask='ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff', broadcast=None, ptp=None),
snicaddr(family=<AddressFamily.AF_LINK: 17>, address='00:00:00:00:00:00',
netmask=None, broadcast='00:00:00:00:00:00', ptp=None)], 'wlan0':
[snicaddr(family=<AddressFamily.AF_INET: 2>, address='192.168.1.3',
netmask='255.255.255.0', broadcast='192.168.1.255', ptp=None),
snicaddr(family=<AddressFamily.AF_INET6: 10>,
address='fe80::c685:8ff:fe45:641%wlan0', netmask='ffff:ffff:ffff:ffff::',
broadcast=None, ptp=None), snicaddr(family=<AddressFamily.AF_LINK: 17>,
address='c4:85:08:45:06:41', netmask=None, broadcast='ff:ff:ff:ff:ff:ff',
ptp=None)]}
```

8.2 Virtual Machines Monitoring: Prometheus Node Exporter

To monitor a virtual machine, we chose Prometheus system and node exporter tool [8] to export hardware and OS metrics exposed by *NIX kernels, written in GO with pluggable metric collectors, supporting collectors for each operating system. Collectors are enabled by providing a --collector.<name> flag and disabled by providing a --no-collector.<name> flag. Table 8-1, contains the list of collectors available for Linux operating system, including a short description of where the statistics are gathered.

TABLE 8-1: PROMETHEUS COLLECTORS FOR LINUX

Collector	Description
arp	Exposes ARP statistics from /proc/net/arp
bcache	Exposes bcache statistics from /sys/fs/bcache/
bonding	Exposes the number of configured and active slaves of Linux bonding interfaces.
boottime	Exposes system boot time derived from the kern.boottime sysctl.
conntrack	Shows conntrack statistics (does nothing if no /proc/sys/net/netfilter/ present).
cpu	Exposes CPU statistics
diskstats	Exposes disk I/O statistics.
edac	Exposes error detection and correction statistics.
entropy	Exposes available entropy.
exec	Exposes execution statistics.
filefd	Exposes file descriptor statistics from /proc/sys/fs/file-nr.
filesystem	Exposes filesystem statistics, such as disk space used.
hwmon	Expose hardware monitoring and sensor data from /sys/class/hwmon/
infiniband	Exposes network statistics specific to InfiniBand and Intel OmniPath configurations.
ipvs	Exposes IPVS status from /proc/net/ip_vs and stats from /proc/net/ip_vs_stats.
loadavg	Exposes load average.
mdadm	Exposes statistics about devices in /proc/mdstat (does nothing if no /proc/mdstat present).
meminfo	Exposes memory statistics.

netclass	Exposes network interface info from /sys/class/net/
netdev	Exposes network interface statistics such as bytes transferred.
netstat	Exposes network statistics from /proc/net/netstat. This is the same information as netstat -s.
nfs	Exposes NFS client statistics from /proc/net/rpc/nfs. This is the same information as nfsstat -c.
nfsd	Exposes NFS kernel server statistics from /proc/net/rpc/nfsd. This is the same information as nfsstat -s.
sockstat	Exposes various statistics from /proc/net/sockstat.
stat	Exposes various statistics from /proc/stat. This includes boot time, forks and interrupts.
textfile	Exposes statistics read from local disk. The --collector.textfile.directory flag must be set.
time	Exposes the current system time.
timex	Exposes selected adjtimex(2) system call stats.
uname	Exposes system information as provided by the uname system call.
vmstat	Exposes statistics from /proc/vmstat.
wifi	Exposes WiFi device and station statistics.
xfs	Exposes XFS runtime statistics.
zfs	Exposes ZFS performance statistics.

Prometheus WMI exporter is available for Windows machines, which leverages Windows Management Instrumentation. Table 8-2: lists the different collectors supported to gather metrics.

TABLE 8-2: PROMETHEUS COLLECTORS FOR WINDOWS

Collector	Description
ad	Win32_PerfRawData_DirectoryServices_DirectoryServices Active Directory
cpu	Win32_PerfRawData_PerfOS_Processor metrics (cpu usage)
cs	Win32_ComputerSystem metrics (system properties, num cpus/total memory)
dns	Win32_PerfRawData_DNS_DNS metrics (DNS Server)
hyperv	Performance counters for Hyper-V hosts
iis	Win32_PerfRawData_W3SVC_WebService IIS metrics
logical_disk	Win32_PerfRawData_PerfDisk_LogicalDisk metrics (disk I/O)
net	Win32_PerfRawData_Tcpip_NetworkInterface metrics (network interface I/O)
msmq	Win32_PerfRawData_MSMQ_MSMQQueue metrics (MSMQ/journal count)
mssql	various SQL Server Performance Objects metrics
os	Win32_OperatingSystem metrics (memory, processes, users)

process	Win32_PerfRawData_PerfProc_Process metrics (per-process stats)
service	Win32_Service metrics (service states)
system	Win32_PerfRawData_PerfOS_System metrics (system calls)
tcp	Win32_PerfRawData_Tcpip_TCPv4 metrics (tcp connections)
textfile	Read prometheus metrics from a text file
vmware	Performance counters installed by the Vmware Guest agent

8.3 Containers Monitoring

This subsection analyses how different container platforms expose metrics that can be later extracted. Docker, LXD and LXC container platforms/technologies will be further analyzed in the following subsections.

8.3.1 Docker Monitoring

Docker platform is the first section to be analysed. Docker exposes mainly two commands and an API that additional monitoring metrics.

The first command is **docker inspect** which exposes low-level information on Docker Objects. Some of the commands main capabilities can be found in Table 8-3: And Table 8-4:. Docker inspect command is capable of exposing low-level information from instantiated docker containers and docker images.

TABLE 8-3: DOCKER INSPECT LOW-LEVEL CONTAINER IMAGE PROPERTIES

Low-level Container Image properties	Description
D	This is the unique identifier of the image.
Parent	Represents the link to its parent image identifier. It is very common for an image to have a defined parent.
Container	Represents the container identifier stored in the image metadata. The container identifier is a temporary container created when the image was built. Docker will create a container during the image construction process, and this identifier is stored in the image metadata.
ContainerConfig	Represents the temporary container configuration created when the image is built
DockerVersion	Describes the version of Docker used to create the image. This value is specially useful to check backwards compatibility between Docker versions.
VirtualSize	Describes the container image size reported in bytes.

TABLE 8-4: DOCKER INSPECT LOW-LEVEL CONTAINER INSTANCE PROPERTIES

Low-level Container Instance properties	Description
D	Describes the container unique identifier.
State	Represents the container state, which can be further described with multiple status flags and the process id of the container.
Image	Describes the image from which this container was instantiated.
NetworkSettings	The network environment for the container and therefore for the application(s) within the image.
LogPath	Represents the system path to the container's log file.
RestartCount	Value that keeps track of the number of times a container has been restarted. This value is the key value used when defining a container's restart policy.
Name	Represents the name defined by the user to the container.
Volumes	Defines the volume mapping between the host system and the container.
HostConfig	Set of configuration parameters which describe how the container will interact with the host system. These parameters include CPU and memory limits, networking parameters, and/or device driver paths.
Config	Represents the runtime configuration options set when the docker run command is executed.

The second command is **docker stats,** which displays live stream resource usage statistics of a set of containers. The next figure shows an example of the output of the command.

```
$ docker stats redis1 redis2

CONTAINER           CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O
redis1              0.07%               796 KB / 64 MB        1.21%               788 B / 648 B       3.568 MB / 512 KB
redis2              0.07%               2.746 MB / 64 MB      4.29%               1.266 KB / 648 B    12.4 MB / 0 B
```

FIGURE 8-1: DOCKER STAT OUTPUT

The data and metrics which can be retrieved by the docker stats command is explained in Table 8-5.

TABLE 8-5: DOCKER STATS DATA AND METRIC FIELDS

Placeholder	Description
.Container	Container name or ID (user input)
.Name	Container name
.ID	Container ID
.CPUPerc	CPU percentage
.MemUsage	Memory usage
.NetIO	Network IO

.BlockIO	Block IO
.MemPerc	Memory percentage (Not available on Windows)
.PIDs	Number of PIDs (Not available on Windows)

Finally, docker provides and API to interact with the running Docker daemon (called the Docker Engine API) (as well as SDKs for Go and Python). To inspect a container (docker inspect equivalent) [9]: GET /containers/{id}json. Also, to retrieve stats [10]: GET /container/{id}/stats

8.3.2 LXD Monitoring

LXD provides monitoring for the containers using a REST API running on the host node (GET /1.0/containers/<name>/state. This API exposes useful information about and instantiated container. The information which this API allows to gather is the following:

- Status of the container
- CPU usage (unclear the unit used)
- Disk usage for each mount (in bytes)
- RAM usage [peak, current, spaw, peak swap] (in bytes)
- Network interfaces: addresses, counters, MAC address, MTU, hostname, state, type.
- PID
- Number of processes running in the container.

Figure 8-2 showcases how the LXD API can further be accessed using Python (currently using pylxd in fog05):

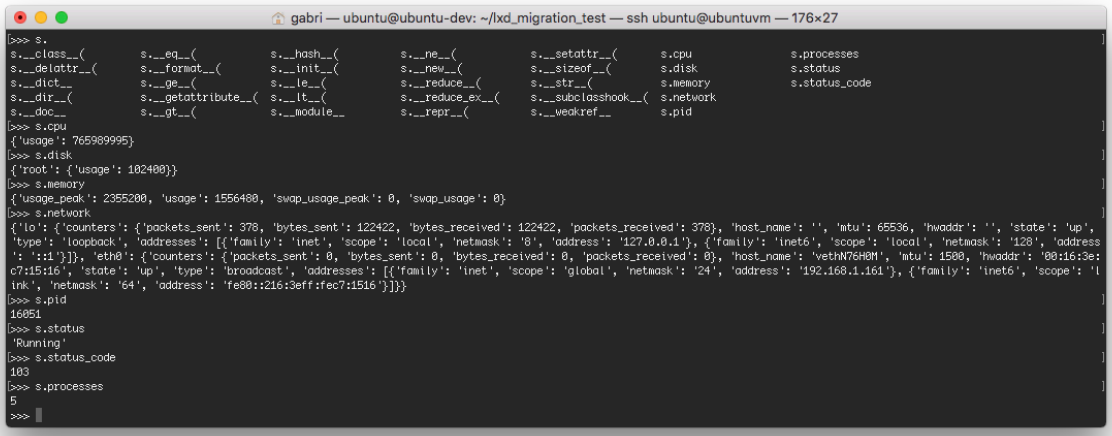


FIGURE 8-2: LXD API CONSUMPTION WITH PYTHON

8.3.3 LXC Monitoring

There are third party exporters and integration tools for LXC. The LXC exporter, located in https://github.com/SebastianCzoch/lxc-exporter, is able to monitor the following information described in Figure 8-3.

Metric name	Description	Enabled by default
lxc_cpu	Seconds the cpus spent in each mode. For all containers	yes
lxc_cpu_precentage	Precentage of usage processor	yes
lxc_cpu_physical_real	Seconds the real physical cpu spent in each mode. (minus containers usage)	yes
lxc_cpu_physical_real_precentage	Precentage of usage processor (minus containers usage)	yes
lxc_memory_usage	Memory usage in each container in bytes	yes

FIGURE 8-3: LXC MONITORING EXPORTER

In addition, LXC default API "lxc info NAME" exposes: total used CPU time; disk usage (for root device); memory usage (current and peak); swap usage (current and peak); network usage (bytes/packets sent/received)

Data gathered from LXC default API can easily be polled and sent to Prometheus platform. However, according to LXC developers, it is quite expensive to extract data from LXC default API consequently it is not advised to fetch data very often as it could cause additional load in the system.

9 Appendix: Zenoh and NATS comparison

TABLE 9-1: ZENOH AND NATS COMPARISON

Protocol	Synch/Async	Pub/Sub reliability	Request Reply	Load Balancing	Topic structure	APIs
NATS	Yes	At most once, at least once	Yes	Queue subscription that balances over subscribers	Tree URI bases with wildcards: • * single token match • > multiple tokent match at end of topic name	Go, Nodejs, Ruby, Java, C, C#
Zenoh	Yes	Netx-hop, First-to-last broker, End-to-En All of this at most once	Yes	Load balancing between brokers	Tree URI based with wildcards • ? single char • * single token • ** multiple tokens	Java, OCaml, Pyhton, C

9.1 Kafka Brokered Performance

In Figure 9-1, RabbitMQ's latency is constant, while ActiveMQ and Kafka are linear. What's unclear is the apparent disconnect between their throughput and mean latencies.

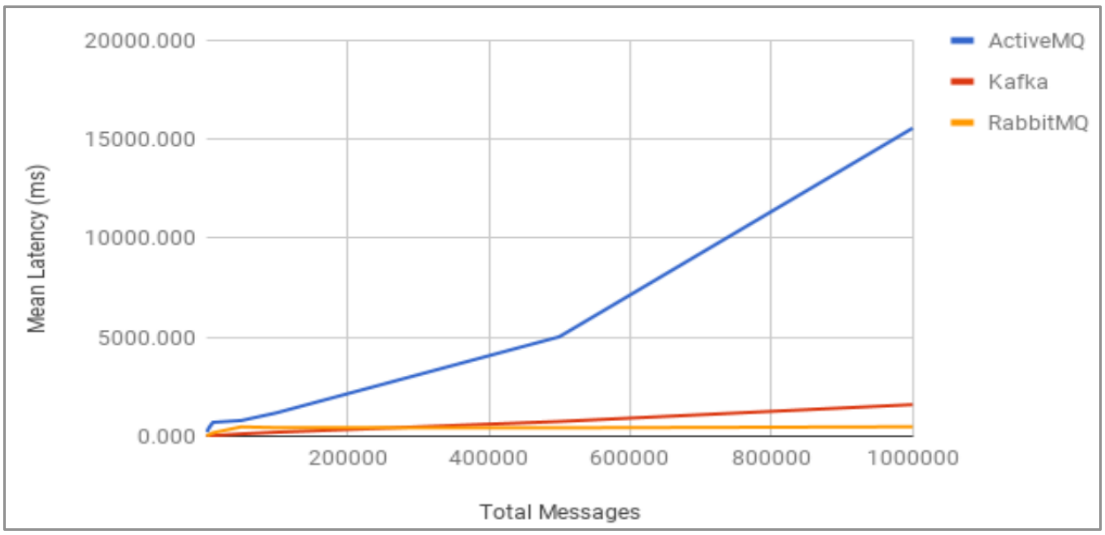


FIGURE 9-1: LATENCY VS MESSAGES ON RESTFUL PROTOCOLS

9.2 Apache Kafka

With synchronous replication in Figure 9-2 [38], a single-node single-thread achieves about 2391 msgs/s, and the best result is 54494 msgs/s with 25 sending and receiving threads and 6 client sender/receiver nodes.

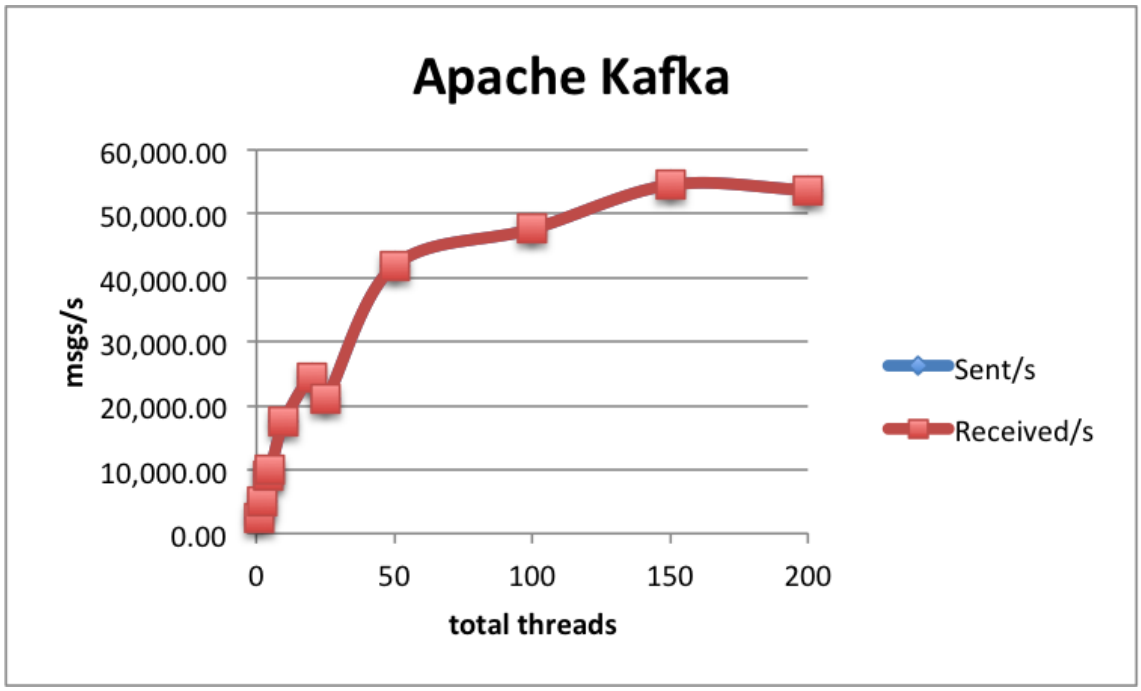


FIGURE 9-2: MESSAGES PER SECONDS VS THREADS ON APACHE KAFKA

In Figure 9-3 [39], the receive rates are very stable. The 95th percentile of the processing latency is also a stable 47 ms. Also, the send latencies are around 48 ms.

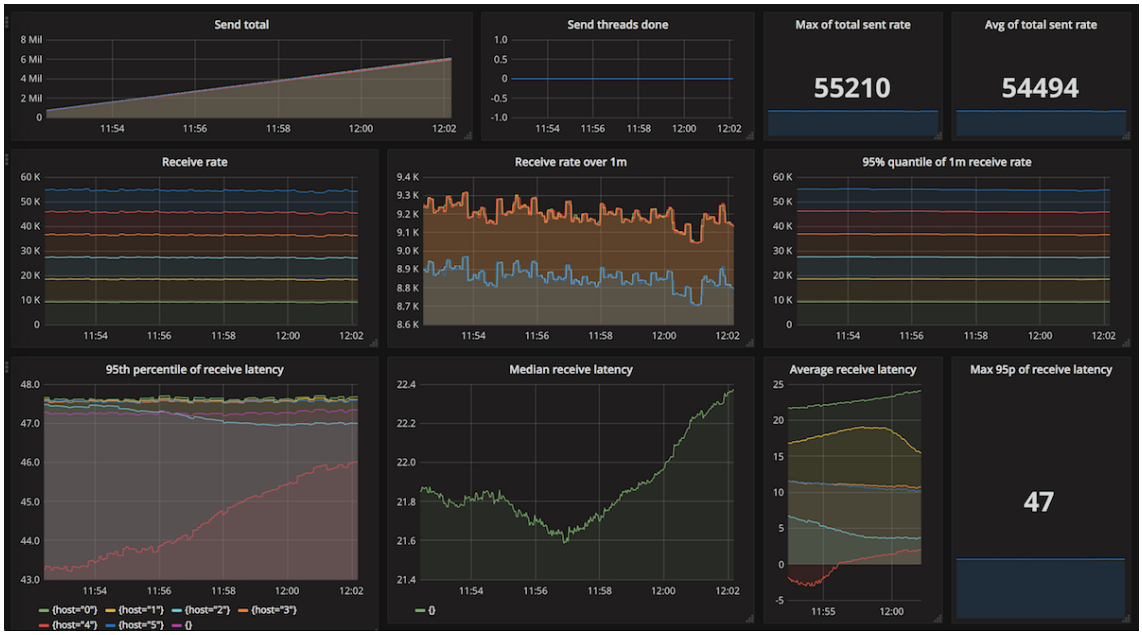


FIGURE 9-3: MAIN MEASUREMENTS ABOUT APACHE KAFKA

Above 6 nodes adding more client threads doesn't increase performance; that's possibly the most we can get out of a 3-node Kafka cluster. However, Kafka has a big scalability potential, by

adding nodes and increasing the number of partitions. We could also scale up the batches: by using batches of up to 100, we can achieve 102170 msgs/s with 4 client nodes, and with batches of up to 1000, a whopping 141250 msgs/s. However, the processing latency then increases to 443 ms.

According to analysis of protocol and comparison with others[40], it has been added two more main results which are show in Figure 9-4 and Figure 9-5.

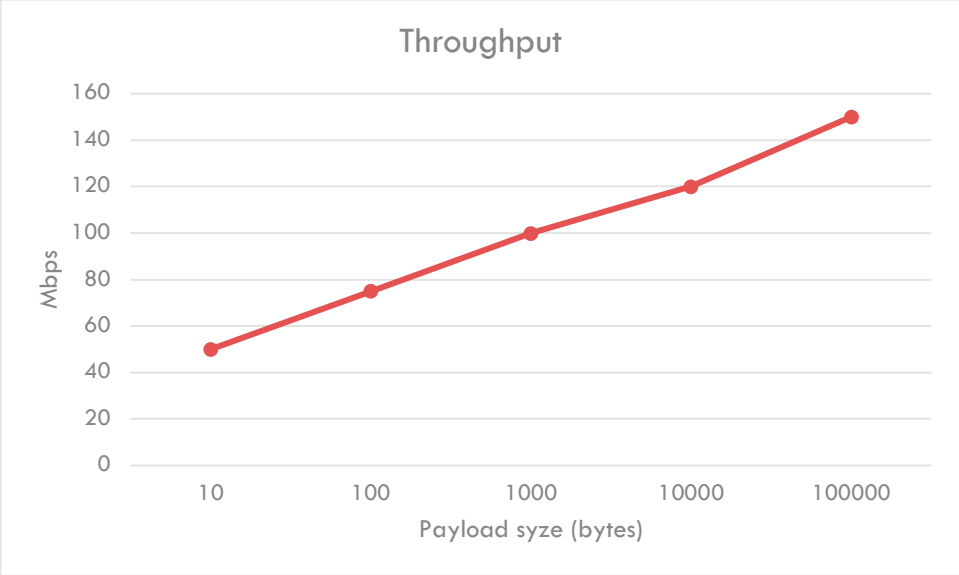


FIGURE 9-4: MBPS VS PAYLOAD ON APACHE KAFKA

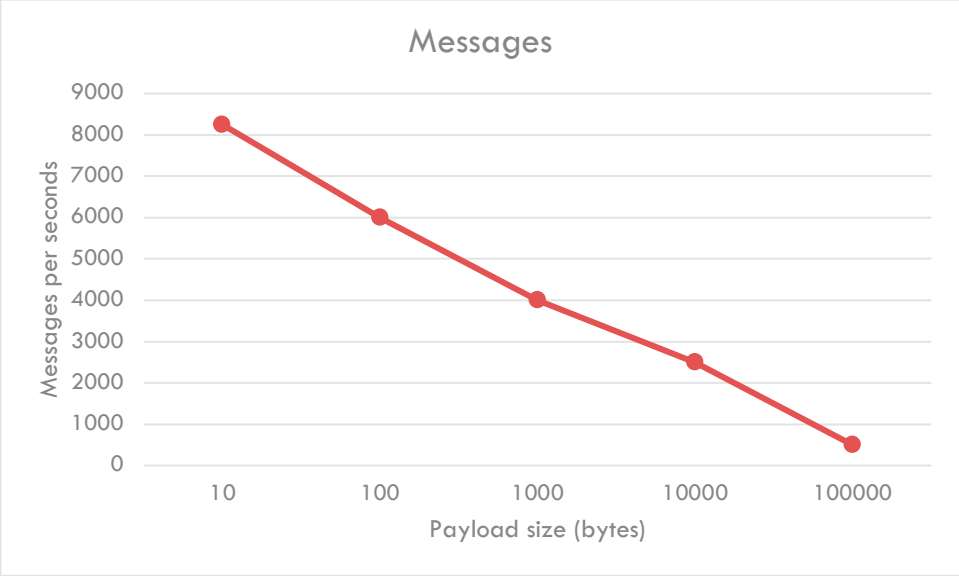


FIGURE 9-5: MESSAGES PER SECONDS VS PAYLOAD ON APACHE KAFKA

9.3 EFS service platform data storage engine

The EFS service platform provides two main tasks: data storage to collect information from applications, functions and edge and fog resources; and the communication protocol to provide or gather this information. First, this subsection outlines options for the EFS service platform data

storage engine. Next, it complements the analysis of messaging protocols provided in deliverable D2.1 [1] with two additional protocols namely Zenoh and RESTful publish-subscribe. This is then followed with 5G-CORAL final conclusions regarding the messaging protocols of choice.

After having described the characteristics of the EFS platform, where edge and fog devices form the substrate of the system, the data storage system that suits within the EFS service platform is a distributed one. In this kind of storage systems, the information is stored in more than one node, probably having replicas of the information spread over some nodes. One of the most common distributed databases are non-relational ones. Depending on the implementation of the distributed database, it may expose from key-value schema to more complex queries.

Regarding the design of this type of storage systems, it is relevant to point out the CAP theorem, which states that a distributed data store will approve two out of three features, named consistency, availability and partition tolerance. The last one means that the system will continue running correctly even when there are isolated network failures. In a distributed system, this is a key feature. Regarding consistency, all nodes have the same view of the data at the same time. Finally, availability, aiming to keep the system operational all the time, regardless the state of any node of the cluster.

Examples of distributed non-relational databases: Apache Cassandra [12]; Bigtable [13]; Druid [14]; MongoDB [15]; Voldemort [16].