H2020 5G-Coral Project

Grant No. 761586

D2.1: Initial design of 5G-CORAL Edge and Fog computing system

Abstract

This deliverable provides the first release of the 5G-CORAL Edge and Fog Computing System (EFS) architecture and design. It addresses the following aspects of the 5G-CORAL EFS: the EFS requirements; the EFS architecture including internal and external interfaces; a comprehensive survey, analysis and selection of the EFS Service platform messaging/communication protocols; and a baseline EFS design for the 5G-CORAL use cases.

Document properties

Document number	**D2.1**
Document title	D2.1: Initial design of 5G-CORAL Edge and Fog computing system
Document responsible	Simon Duqennoy (SICS)
Document editor	Charles Turyagyenda (IDCC)
Editorial team	Simon Duqennoy (SICS), Alain Mourad (IDCC), Charles Turyagyenda (IDCC), Ping- Heng Kuo (IDCC), Giovanni Rigazzi (IDCC), Chenguang Lu (EAB), Shahzoob Bilal Chundrigar (ITRI), María Felisa Sedano Ruíz (TELCA), José María Roldán Gil (TELCA)
Target dissemination level	Public
Status of the document	Final
Version	1.0

List of contributors

Partner	Contributors
ADLINK	Gabriele Baldoni
IDCC	Charles Turyagyenda, Ping-Heng Kuo, Giovanni Rigazzi
ITRI	Robert Gdowski, Samer Talat, Shahzoob Bilal Chundrigar, Gary Huang, Ibrahiem Osamah
NCTU	Li-Hsing Yen, Hojjat Baghban, Hsu-Tung Chien
TELCA	María Felisa Sedano Ruiz, José María Roldán Gil
UC3M	Luca Cominardi
SICS	Simon Duquennoy, Niklas Wirström
EAB	Chenguang Lu, Daniel Cederholm, Miguel Berg
AZCOM	Riccardo Ferrari and Alessandro Colazzo

Production properties

Reviewers	Alain Mourad, Shahzoob Bilal Chundrigar, Chenguang Lu

Document history

Revision	Date	Issued by	Description
1.0	1 June 2018	SICS	D2.1 ready for publication

Disclaimer

Table of Contents

List of Figures

List of Tables

List of Acronyms

3GPP	3rd Generation Partnership Project
5G-PPP	5G Private Public Partnership
AAA	Authentication, Authorization and Accounting
AMQP	Advanced Message Queuing Protocol
AP	Access Point
API	Application Programming Interface
APIaaS	API as a service
AR	Augmented Reality
BLE	Bluetooth Low Energy
BSS	Business Support System
BSSID	Basic Service Set Identifier
CD	Computing Devices
CoAP	Constrained Application Protocol
CPU	Central Processing Unit
D2D	Device-to-Device
DAS	Direct Attached Storage
DASH	Dynamic Adaptive Streaming over HTTP
DDS	Data Distribution Service
DTOA	Differential Time of Arrival
EFS	Edge and Fog Computing System
EFS-VI	EFS Virtualisation Infrastructure
EI	Event Information
EPC	Evolved Packet Core
ETSI	European Telecommunications Standards Institute
GNSS	Global Navigation Satellite System
GUI	Graphical User Interface
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
IR	Image Recognition
ISO	International Organization for Standardization
ISG	Industry Specification Group
LAN	Local Address Network
LTE	Long Term Evolution
MAC	Media Access Control
MEC	Mobile Edge Computing
MME	Mobility Management Entity
MQTT	Message Queue Telemetry Transport
NAS	Network Attached Storage
NB-IoT	Narrow-Band IoT
NFV	Network Functions Virtualisation
OCS	Orchestration and Control System
OSS	Operation Support System
P2P	Peer-to-Peer
PEF	Performance Enhancing Function
PHP	Hypertext Preprocessor
PNF	Physical Network Functions
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
SaaS	Software as a Service
SASL	Simple Authentication and Security Layer
SC	Safety Core
SDR	Software-Defined Radio
S-GW	Serving Gateway
SSID	Service Set IDentifier
STOMP	Simple (or Streaming) Text Oriented Message Protocol
TAU	Tracking Area Update
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TSCH	Time-Slotted Channel Hopping
UC	Use Case
UDP	User Datagram Protocol
UE	User Equipment
URL	Uniform Resource Locator
vAP	Virtual Access Point
VIM	Virtualisation Infrastructure Manager
vMME	Virtual MME
VNF	Virtual Network Functions
VR	Virtual Reality
XMP	Extensible Metadata Platform
XMPP	Extensible Messaging and Presence Protocol

Executive Summary

The 5G-CORAL system aims to address the ultra-low latency requirements of emerging 5G applications 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 virtualised networking and computing solution where virtualised functions, user and third-party applications, and context-aware services are blended together to offer enhanced connectivity and better quality of experience. An integral part of the 5G-CORAL system is the distributed Edge and Fog Computing System (EFS) that offers a shared hosting environment for virtualised functions, services and applications.

This deliverable provides the first release of the 5G-CORAL Edge and Fog Computing System (EFS) architecture and design. The deliverable addresses the following aspects of the 5G-CORAL EFS: the EFS requirements; the EFS architecture including internal and external interfaces; a survey, analysis and selection of the EFS Service platform messaging/communication protocols; and a baseline EFS design for the 5G-CORAL use cases. The following highlights the key achievements in this deliverable:

- A comprehensive description of the 5G-CORAL Edge and Fog Computing System (EFS) requirements and architectural design, including: the EFS virtualisation infrastructure (i.e. physical/virtual compute, storage and networking), the EFS entities (i.e. EFS applications, EFS functions, EFS service platform and the respective entity managers), the EFS internal and external interfaces.
- A detailed description of the microservice-based design principle for EFS atomic entities, i.e. the building blocks of EFS entities.
- A detailed description of the publish/subscribe communication framework between the EFS service platform and the EFS/non-EFS applications and functions.
- The selection of messaging protocols (DDS and MQTT), as the reference/baseline messaging protocols for the EFS following an in-depth analysis of state of the art protocols.
- A baseline EFS design for 5G-CORAL use cases, namely: Robotics, Virtual Reality (VR), Augmented Reality (AR), High-speed Train, IoT Multi-RAT Gateway and Connected Cars. The baseline design decomposed each use case into the constituent EFS entities and described their respective interworking(s).

Future work is anticipated to expand and refine these results by filling gaps identified, such as: the design of the EFS service platform data storage engine, a study of RESTful publish/subscribe messaging, a refined description of EFS internal and external interfaces, EFS workflows for resource/service discovery and integration, data models for the EFS APIs and EFS implementation.

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. As the technical requirements of these 5G applications materialize [18], it is evident that certain applications will require very low end-to-end latency (~0.1-20 milliseconds). This ultra-low latency requirement is extremely challenging and stressing for the network 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 virtualised networking and computing solution where virtualised 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 virtualised 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.).

Against this backdrop, the objective of this deliverable is to provide the first release of the 5G-CORAL EFS architecture and design developed over the first nine-month period of the project. The deliverable addresses several aspects of the 5G-CORAL EFS, namely: the EFS requirements; the EFS architecture including internal and external interfaces; a comprehensive survey, analysis and selection of the EFS Service platform messaging/communication protocols; and a baseline EFS design for the 5G-CORAL use cases. The deliverable is structured as follows.

Section 2 presents a description of the 5G-CORAL EFS architecture and requirements highlighting the following key aspects: the overall 5G-CORAL system architecture, the EFS virtualisation infrastructure requirements and the EFS entities' requirements, the EFS architectural design and the corresponding EFS internal and external interfaces.

Section 3 describes the EFS service platform and presents a comprehensive survey, analysis and selection of messaging/communication protocols suitable for the EFS Service platform. The protocols studied include: DDS, MQTT, AMQP, RabbitMQ, XMPP, Kafka, NATS, and Confluent.

Section 4 presents the EFS baseline design for 5G-CORAL use cases, namely: Robotics, Virtual Reality (VR), Augmented Reality (AR), High-speed Train, IoT Multi-RAT Gateway and Connected Cars. The baseline design decomposes each use case into the constituent EFS entities and describes their respective interworking(s).

Finally, in **Section 5**, a conclusion is presented to summarize the findings of this deliverable, as well as setting the prospects for future work.

2 EFS Requirements and Architecture

2.1 5G-CORAL Refined Architecture

The 5G-CORAL architecture was specified in 5G-CORAL deliverable D1.1 [1]**Error! Reference s ource not found.**. It is based on the ETSI MEC and ETSI NFV frameworks and composed of the following two sub-systems:

- **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 EFS domains.
- **Orchestration and Control System (OCS):** an OCS is a logical system in charge of composing, controlling, managing, orchestrating and federating one or more EFS(s). An OCS comprises Virtualisation Infrastructure Managers (VIMs), EFS managers, and EFS orchestrators. An OCS may interact with other OCS domains.

The EFS incorporates a mix of physical and virtualised resources available in the fog and edge tiers. As described in D1.1 [1]**Error! Reference source not found.**, the main differences between a n edge resource and a fog resource lie in their i) computing capabilities (a fog resource is more limited than an edge resource), ii) mobility (a fog resource may be mobile whereas an edge resource is assumed stationary), and iii) availability (a fog resource may have intermittent availability compared to an edge resource which is always available). These differences influence design aspects such as dominance of wireless communication, battery support and low resource extensibility for the fog resource as compared to an edge resource.

The EFS is a logical system composed of functions/applications and a service platform. An atomic entity is an unpartitionable computing task executed in the EFS. An EFS function is defined as a software entity comprised by at least one atomic entity deployed in EFS for network infrastructure. An EFS application is defined however as a software entity comprised by at least one atomic entity deployed in EFS for end users and third parties.

The EFS Service Platform is a logical data exchange platform constituting of:

- Data storage to keep the collected information from applications/functions and edge/fog resources.
- Messaging/communication protocols to gather/provide information from/to applications/functions and edge/fog resources.

EFS applications, functions, and Service Platform are also referred to as EFS entities. Figure 1 shows the refined 5G-CORAL architecture developed in collaboration between WP2 and WP3. The main changes for the EFS subsystem architecture compared to the initial version presented in D1.1 [1] are the following:

- NFVI has been renamed as EFS-VI as to encompass the capability of the EFS Virtualisation Infrastructure to also host applications and service platform;
- The Element Managers in the EFS have been renamed as Entity Managers to reflect the definition of EFS application, function, and Service Platform. Nevertheless, the Entity Managers play the same role of ETSI NFV Element Managers as initially stated;
- T8 interface has been extended as to also support non-EFS Resources;
- O5 and O6 interfaces connect also the EFS App/Func Manager to the corresponding EFS application/function Entity Manager;
- VNF Descriptor has been changed to Entity Descriptor to cover EFS application, EFS function, and EFS Service Platform Descriptor.

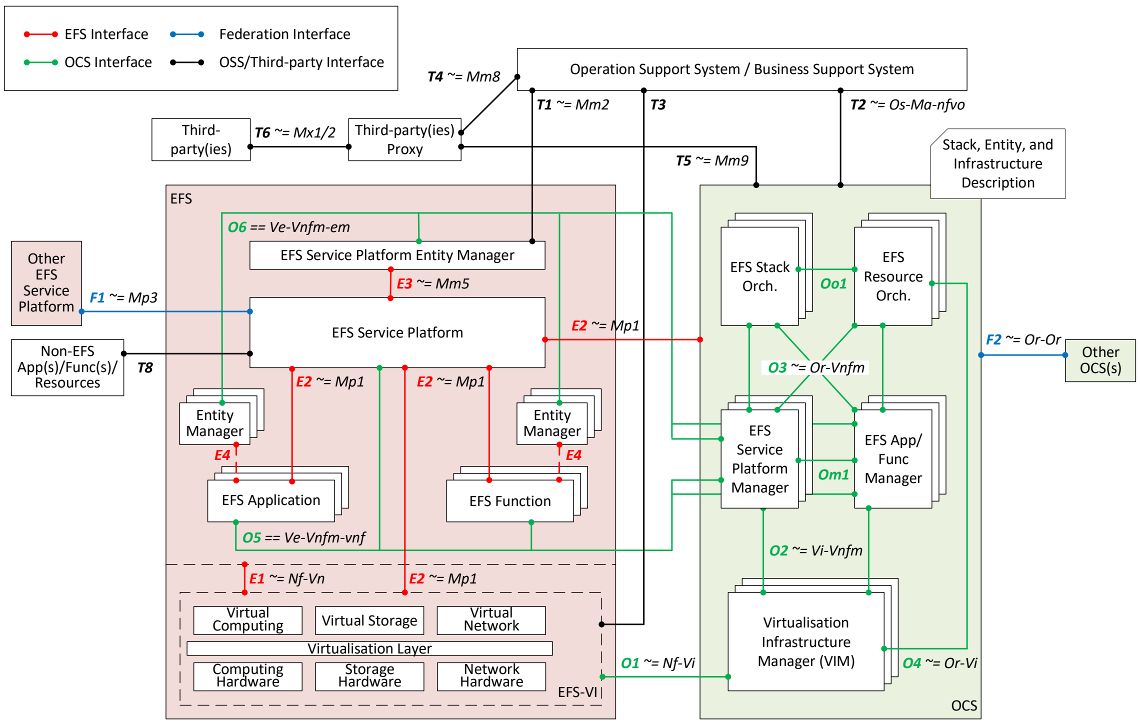


Figure 1 5G-CORAL Refined System Architecture

2.2 EFS Requirements

This section presents an overview of the EFS requirements derived from the 5G-CORAL system requirements presented in D1.1 [1]. The EFS requirements are grouped into two categories, namely 1) requirements pertaining to the EFS virtualisation infrastructure (EFS-VI) and 2) requirements pertaining to the EFS entities, i.e. EFS applications, EFS functions and EFS service platform. Table 1 and Table 2 present the mapping of EFS requirements against the system-level requirements defined in D1.1 [1].

Table 1 Requirements for EFS virtualisation infrastructure (EFS-VI)

ID	Requirement	System-level requirement
EI-01	The EFS-VI shall support various categories of EFS resources, e.g. Fog and Edge resources.	FT-01
EI-02	The EFS-VI shall support the abstraction and virtualisation of the EFS resources.	FT-02
EI-03	The EFS-VI compute[1], storage and network nodes shall incorporate one or more Ethernet network ports for interconnection with other devices, e.g. Physical Network functions (PNFs).	FT-05, FT-08
EI-04	The EFS-VI network node northbound links (i.e. towards the VIM) shall incorporate a variety of Multi-RAT links including Ethernet ports based on industry standards.	FT-05, NF-08
EI-05	The EFS-VI network node east/west ports should exist for each additional compute/storage/network node.	FT-05, FT-08

[1] Compute, Storage and Network nodes are defined as EFS-VI nodes that provide compute, storage networking functions, respectively, to the NFVI.

EI-06	The EFS-VI compute/storage/network nodes may support east/west Multi-RAT links	FT-05, FT-08
EI-08	The EFS-VI compute, storage and network nodes should be able to physically be interconnected with the minimum hops in order to cope with latency.	FT-05, FT-08, NF-04
EI-09	The EFS-VI compute nodes shall incorporate support of host platforms e.g., virtual machines or containers.	FT-02
EI-10	The EFS-VI compute nodes shall incorporate minimum hardware requirements to run the virtualised hosts.	FT-02
EI-11	The EFS-VI shall support occasional addition and removal of EFS resources that maybe mobile and volatile.	FT-03, FT-09, FT-10
EI-12	The EFS-VI shall support localization of EFS resources.	FT-11
EI-13	The EFS-VI shall support synchronization across the EFS resources that maybe distributed or co-located.	FT-12

TABLE 2 REQUIREMENTS FOR EFS ENTITIES

ID	Requirement	System-level requirement
EE-01	Support for discovery of EFS entities, i.e. EFS applications, EFS functions and the EFS service platform.	FT-03, FT-07, FT-13, NF-02
EE-02	EFS entities shall expose APIs through which other EFS and Non-EFS entities may communicate.	FT-04, FT-07, FT-11, FT-12, FT-15, NF-01, NF-03, NF-05, NF-06, NF-08, NF-12, NF-13
EE-03	Support for placement, instantiation, migration, monitoring, configuration and termination of EFS entities.	FT-06
EE-04	Support for synchronization across the distributed EFS entities.	FT-11
EE-05	Support the subscription, authentication, registration, and admission to EFS entities from both inside and outside of the EFS.	FT-04, FT-07, NF-12
EE-06	Support for the independent deployment of EFS entities, i.e. the deployment of any EFS entity does not depend on the existence of other EFS entities.	FT-07, NF-05, NF-11
EE-07	Support for multiple technologies in the software stack of EFS entities, i.e. the use of different programing languages within each EFS application, EFS function and the EFS service platform.	FT-07, NF-05
EE-08	Support for decoupling of the implementation logic of EFS entities from APIs exposed by the EFS entities.	FT-07, NF-05, NF-12
EE-09	Support for loose coupling among the EFS atomic entities that make up the EFS entities. Loose coupling among atomic entities facilitates refactoring, upgrading and deleting modules without unnecessarily affecting other parts of the EFS entity.	FT-07, NF-05, NF-10, NF-12
EE-10	Support for scaling of EFS entities.	NF-11
EE-11	Support for isolation of EFS entities.	NF-05
EE-12	Support for integration of 3rd party EFS and Non-EFS entities.	FT-05, FT-08, NF-11
EE-13	Support for multiple RATs via EFS functions.	FT-08
EE-14	Support for the extraction and distribution of context information from RATs using the EFS service platform.	FT-07, FT-08,
EE-14	Support for localization of EFS entities.	FT-11

2.3 EFS Architecture

This section presents the EFS design philosophy addressing: (i) the virtualisation infrastructure, (ii) the EFS entities hosted therein and (iii) the interconnecting logical interfaces. The EFS design leverages virtualisation technologies that decouple the EFS entities from the underlying EFS resources, i.e. compute, storage and network. Some of the benefits derived from virtualisation include: resource consolidation, isolation, faster provisioning, disaster discovery, dynamic load balancing, faster development and test environment, reduced hardware vendor lock-in, improved system reliability, and security. While the EFS architectural design is compliant with ETSI MEC and ETSI NFV frameworks, the EFS provides two notable extensions:

1. First, the EFS virtualisation infrastructure (EFS-VI) extends the ETSI Network Functions Virtualisation (ETSI- NFV) 0 reference architecture to incorporate mobile and volatile resources that have different levels of availability, mobility, storage, computing, networking and power capabilities.
2. Second, the EFS entities extend the ETSI-NFV network functions to include EFS applications and an EFS service platform.

Table 3 presents an overview of the EFS internal and external interfaces (also illustrated in Figure 1).

TABLE 3 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”
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.

ID	ETSI NFV/MEC ref. point	Description
F1	ETSI MEC: Mp3	This is the reference point between the EFS service platform and other EFS Service platform(s).

2.3.1 EFS Virtualisation Infrastructure (EFS-VI)

The EFS virtualisation 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. Table 1, presents the requirements for the EFS-VI that are largely adopted from the NFV Infrastructure (NFVI) requirements of [3]. To fulfil the EFS-VI requirements, the EFS-VI is designed to comprise three domains (i.e. compute, storage, and network) and a virtualisation layer.

- **Compute domain:** provides the processing capability and comprises both virtual and physical compute resources. The EFS compute domain consists of: inexpensive compute nodes placed very close to the end user (i.e. Fog nodes); and high-end servers that normally reside in Edge data centers. D1.1 [1] provides a detailed classification of Edge and Fog computing devices.
- **Storage domain:** responsible for storing, porting and extracting data files and objects. Storage resources can either be virtual or physical; and may be classified as either shared network attach storage (NAS) or direct attached storage (DAS). The EFS-VI storage domain primarily refers to shared network attach storage (NAS).
- **Network domain:** responsible for moving data between the storage domain and the compute domain. Network resources can either be virtual or physical constituting: network interface cards (wired and wireless), switches and routers.
- **Virtualisation layer:** abstracts the hardware resources and decouples the EFS entities from the underlying hardware/resource i.e. hardware/resource abstraction. Additionally, the virtualisation layer is responsible for enabling the EFS entities to utilize the virtualised infrastructure. The EFS-VI design considers both hypervisor-based and container-based virtualisation:
    - Hypervisor-based virtualisation: software-based emulation of hardware resources such as: compute, storage and network. Therefore, it is possible for the host to emulate other types of devices, CPU architectures and operating systems.
    - Container-based virtualisation: utilizes kernel features to create an isolated environment for processes. In contrast to hypervisor-based virtualisation, containers do not get their own virtualised hardware but use the hardware of the host system. Therefore, software running in containers does directly communicate with the host kernel and must be able to run on the operating system and CPU architecture the host is running on.

2.3.2 EFS Entities

The EFS entities comprise 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.

The EFS entities are designed following the microservice-based [19] design paradigm as opposed to monolithic software design principles, i.e. the EFS atomic entities are independent of the microservices. Software programs designed using the monolithic design principles are characterised as being self-contained with tightly coupled software modules/components.

Consequently, updates to any software module/component necessitate re-writing and re-deploying the entire software program.

The microservice-based design, adopted by 5G-CORAL, structures the EFS entities as a collection of loosely coupled autonomous software modules working together, i.e., the EFS atomic entities that make up the EFS applications, EFS functions and the EFS service platform are implemented as microservices. Each microservice is a separate entity with no dependency on other microservices. These microservices implement the logic of the EFS atomic entities and interact with each other and communicate via network calls to enforce separation and avoid tight coupling. The network calls may be implemented using either a request/response model or a publish/subscribe model. Consequently, each EFS atomic entity exposes an application programming interface (API[2]) for other EFS atomic entities to communicate in collaborative way.

The benefits of a microservice- based EFS design include:

- A single EFS entity (i.e. EFS application, EFS function and the EFS service platform) can be deployed independently of the rest of the system. This allows to deploy some of the EFS entities at the edge or fog while keeping others at distinct locations.
- Microservices permit the use of different technologies (e.g. programming languages) inside each EFS atomic entity; additionally, the software stack within each EFS atomic entity can be freely replaced while keeping the API towards the other EFS atomic entities the same.
- Adapting existing EFS entities (i.e. EFS application, EFS function and the EFS service platform) does not require refactoring of the whole EFS entity.
- Microservices permit scaling of only those EFS entities (i.e. EFS application, EFS function and the EFS service platform) that need scaling while keeping the rest untouched.
- Microservices provide better fault isolation, i.e. if one EFS entity or EFS atomic entity fails, the others continue to function.
- With microservices, it is easy to integrate with 3rd party EFS entities (i.e. EFS application, EFS function and the EFS service platform).
- Microservice-based EFS atomic entities and EFS entities can easily be implemented using containers or virtual machines.
- Microservice-based design of EFS atomic entities inherently provides isolation of the individual EFS atomic entities thus simplifying the security-related design.

In 5G-CORAL, we take the same design principle of microservices to design the EFS atomic entities. Additionally, because of the distributed nature of the 5G-CORAL EFS and the mobility and volatility of EFS resources, the EFS entities' API design adopted publish/subscribe messaging as the communication mechanism among the EFS applications, EFS functions, and the EFS service platform, as depicted in Figure 2. The benefits of the "publish/subscribe" EFS service platform include:

- Loose coupling: Publishers are loosely coupled to subscribers, and don't need to know about their existence or physical location.
- System topology agnostic: Both publishers and subscribers focus on the topic and can be ignorant of the underlying system topology.
- Robustness: Each publisher and subscriber can continue to operate normally regardless of the other unlike the traditional client–server paradigm; where the client cannot post messages to the server while the server process is not running, nor can the server receive messages unless the client is running. This quality is particularly relevant to out-of-coverage or offline scenarios where the EFS service platform may retain some data regarding mobility and volatility.

[2] An Application Programming Interface (API) is defined as the means by which one software program provides access of its data or processes to another software program(s).

- Highly suited for ubiquitous computing and distributed embedded systems, i.e. Fog and Edge environments.
- Adaptability: Can be varied to suit different environments, e.g. error-prone environments as opposed to the cloud/edge only data centres where the environment is very well controlled

The EFS service platform provides the API framework through which both EFS and non-EFS applications and functions can publish and subscribe to various services e.g. localisation service, radio network information service (RNIS), etc. EFS and non-EFS applications and functions may publish and/or subscribe to one or more topics, maintained by EFS service(s). The EFS service platform performs Authentication, Authorisation and Accounting (AAA) of EFS and non-EFS applications and functions that publish and/or subscribe to the EFS service(s). The EFS service platform also maintains a service registry to track all of the services consumed by the applications and functions. Each EFS service is essentially an API with multiple topics that authorised applications/functions can publish to and/or subscribe to receive notifications from. Therefore, the EFS service platform is an API as a service (APIaaS), a subset of Software as a Service (SaaS).

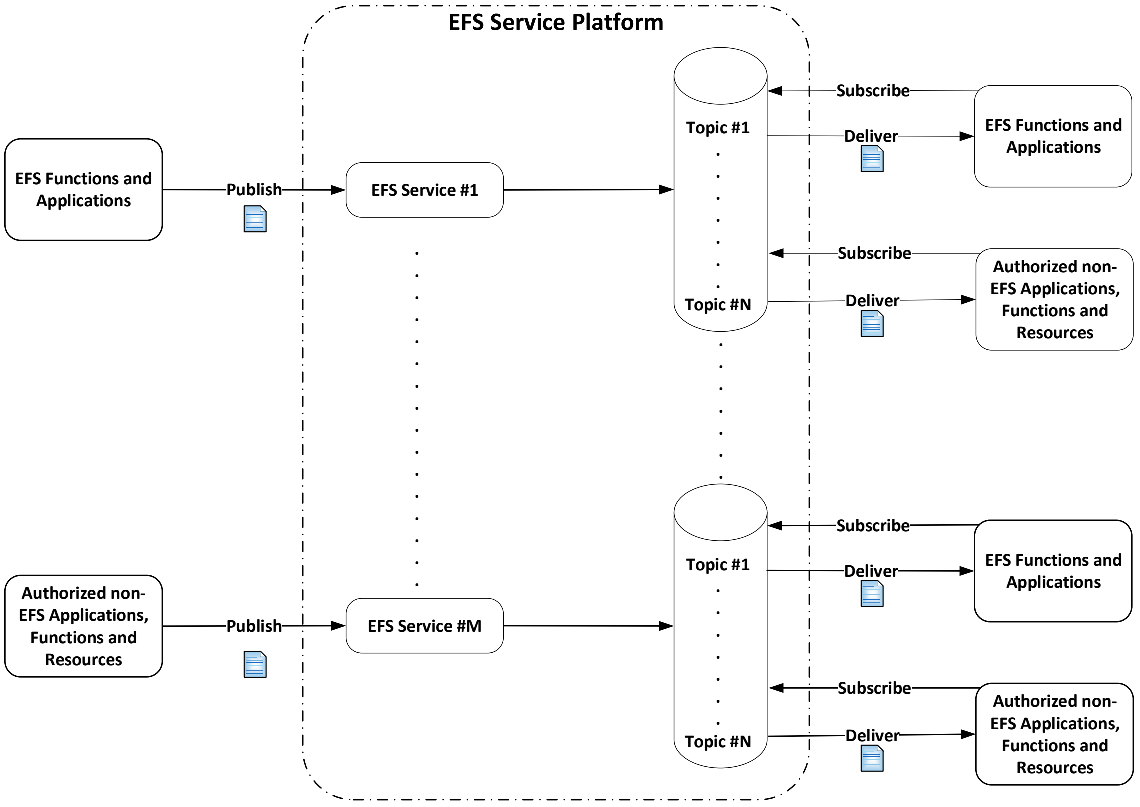

EFS Service Platform
EFS Functions and Applications
Publish
EFS Service #1
Topic #1
Topic #N
Subscribe
Deliver
EFS Functions and Applications
Authorized non-EFS Applications, Functions and Resources
Authorized non-EFS Applications, Functions and Resources
Publish
EFS Service #M
Topic #1
Topic #N
Subscribe
Deliver
EFS Functions and Applications
Authorized non-EFS Applications, Functions and Resources

FIGURE 2 PUBLISH/SUBSCRIBE MESSAGING AMONG EFS ENTITIES

3 EFS service platform and messaging protocols

3.1 EFS service platform

5G-CORAL aims to deliver a multi-RATs convergence mechanism based on information sharing. By allowing different RATs to share their context information, potentially coordination among multiple RATs can be carried out for purposes of performance enhancement. In addition to RATs convergence, such an information sharing framework can be further generalized for the applications and functions running in EFS to obtain and exploit the knowledge on the status of each other, so these entities can be configured in a more judicious manner. Essentially, the functions and applications can share a global view comprising other entities co-existing in the same EFS, so the user experience can improve via tighter coordination in the network side. All these objectives can be realized with EFS Service platform, which is the anchor entity of great importance that defines the EFS solution in 5G-CORAL.

Basically, 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. 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 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 analysed several existing messaging protocols, as detailed in the section 3.2. Section 3.2 presents an analysis of potential EFS messaging/communication protocols. The analysis was conducted by undertaking a comprehensive survey of existing messaging/communication protocols and assessing their feasibility to fulfill the EFS messaging requirements. The analysis also presents a selection of reference messaging/communication protocol(s) for the EFS in section 3.3.

3.2 Survey of messaging/communication protocols

The protocols that were studied by 5G-CORAL WP2 include: **DDS, MQTT, AMQP, RabbitMQ, XMPP, Kafka, NATS, and Confluent.** It is worth noting that WP2 also surveyed enterprise platform(s) such as Solace, however, the scope of this deliverable will be focused on open-source protocols.

3.2.1 DDS

The OMG Data Distribution Service (DDS) is a standardized set of APIs, behaviour and protocols for building real-time distributed applications. The programming model of DDS is an eventually-consistent shared data space, but it also gives direct access to the underlying publish-subscribe messaging system. It originated from the defence and aerospace industry but is now also one of the leading protocols for enabling the Internet of Things. There are many independent implementations.

The mapping to UDP/IP is standardized, but other protocols can (and are) supported as well. Various vendors currently providing their own mapping to TCP/IP, and there are also implementations that map it to shared memory transports or allow operating over layer-2 Ethernet without an IP stack.

3.2.1.1 Architecture of DDS

A DDS-based network is a dynamically discovered abstract “Global Data Space or Domain” that “DomainParticipants” interact with. Typically, these DomainParticipants correspond one-to-one to application processes. The Domain is partitioned into named *Topics*, where each Topic has a type, a QoS and a *key*, a designated subset of the type that is used to distinguish different *instances* of the topic.

DomainParticipants gain access to this data space by creating “DataReaders” and “DataWriters”. The former give read access to the data of a specific Topic — hence creating one is often called “taking a subscription”; and the latter allow the application to update data of a specific Topic in the data space.

The QoS govern many different aspects of the behavior with the most important one being whether or not data once written is retained for future subscriptions. The other key QoS allows further partitioning of the data space based on tagging data with abstract *partitions*, which are arbitrary names and/or wildcards, that are taken into account in the publish-subscribe mechanism. Beyond these, many details are covered, such as: reliability, the size of the history to retain, whether updates to an instance may be dropped if a more recent value is available, rate limiting, content filtering, etc.

3.2.1.2 Security

The DDS Security standard defines a fine-grained mechanism for securing the data. The basic model is that only authorized DomainParticipants may interact with the data space, with optional access control at the level of DataReaders and DataWriters. The encryption and authentication of data is done application-to-application, so that there is no need to trust the middleware.

DDS implementations that support operating over TCP/IP also offer the possibility of securing these connections with TLS. Applications therefore have the option of using the more deeply integrated but less common DDS Security or to use completely standard internet technology for providing secure communications.

3.2.1.3 System requirements

The type of systems supported by different DDS implementations varies considerably, but in general ranges from high-end microcontrollers with real-time operating systems (e.g., ARM Cortex-M4 with FreeRTOS) to many-core workstations running Linux or Windows. Network overhead tends to be significantly lower than most other middleware, thanks to the use of UDP, efficient encodings and multicast.

Language support varies from vendor to vendor, but typically covers mainstream programming languages, such as C, C++, Java and C#, Python and JavaScript. There also exist bindings for Haskell, Lua, Matlab, and several others, some as officially supported language bindings, some as community projects.

3.2.2 MQTT

MQTT (Message Queue Telemetry Transport) has been identified as an enabling connectivity protocol for IoT (Internet of Things) scenarios involving telemetry devices with low computing power and limited battery. Thanks to the extremely lightweight protocol of MQTT, low power consumption and traffic overhead over transport can be anticipated when it is used. MQTT is an ISO standard (ISO/IEC PRF 20922) publish-subscribe-based messaging protocol operating above TCP/IP protocol. Its extended version, namely MQTT-SN, has been specified in 2013 to support operation with UDP, ZigBee and other transport protocols.

3.2.2.1 Architecture of MQTT

An MQTT-protocol-based network comprises two types of entities, namely MQTT Broker (or Server) and MQTT Client. The MQTT broker is responsible for managing subscription of clients and their connections, as well as delivering messages from a publisher to its corresponding subscribers. It receives identification, e.g. IP address and device identifier, and topics from a MQTT client, by which the broker can make a routing table to address the MQTT client. The MQTT broker receives messages from publishers and multicasts them to subscribers that are interested in the same topics. A secure connection between the broker and a client can be established with TLS security method.

A MQTT client can be either a subscriber or a publisher for a topic. As a subscriber, the client subscribes topics via the MQTT broker, which may multicast messages associated to its subscribed topics. On the other hand, the client transmits messages of one or more topics to the MQTT broker when it is acting as a publisher. Note that an MQTT broker can play the roles of publisher and subscriber at the same time. The broker-based publication and subscription concept is illustrated in Figure 3.

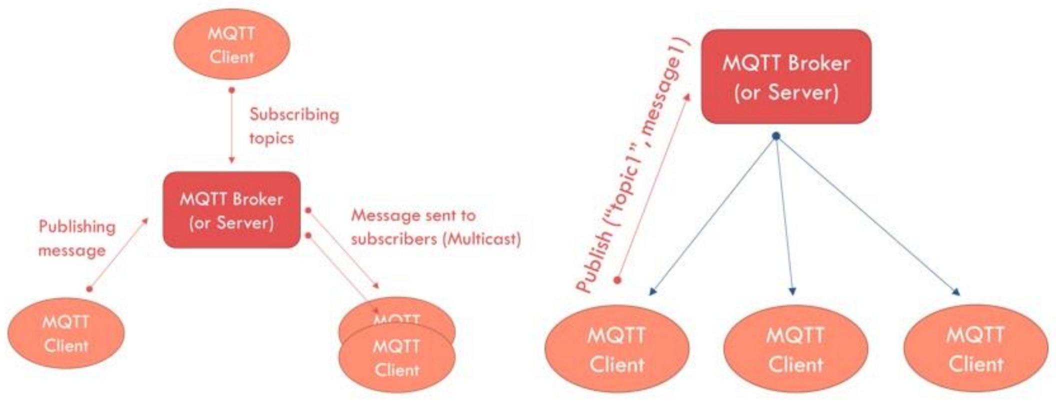


FIGURE 3 MQTT PUBLICATION AND SUBSCRIPTION CONCEPT BASED ON BROKER

The MQTT messages can be classified into two types, namely a command message or a data message. The two types of messages are distinguished by the control header field in the message format. The control messages may include connection establishment message, (un)subscribe message, ACK message, and so on. A data message, on the other hand, includes a list of topic names in a header field with a variable length, along with the data payload. Both topic name and data payload are encoded by UTF-8. The maximum length of topic name and data payload is 32,767 characters and 256 Mbytes respectively. Figure 4 shows the message format of MQTT.

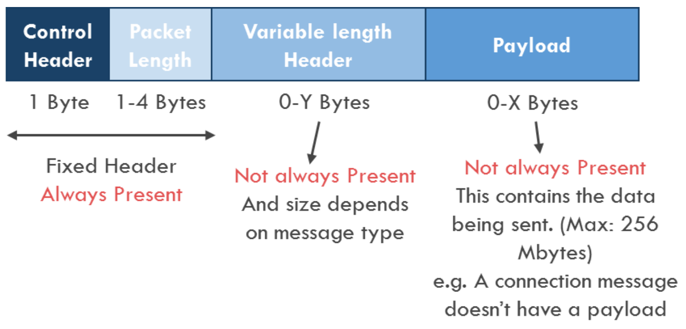


FIGURE 4 MESSAGE FORMAT OF MQTT PROTOCOL

3.2.2.2 MQTT Characteristics

As mentioned above, MQTT is a lightweight messaging protocol comprising MQTT client and server. For instance, paho mqtt-client library and mosquito mqtt-server application only require 100kbytes and 300kbytes respectively. Header overhead is as small as 1~10 bytes. Regarding computational complexity, the CPU usage is negligible at the client side, and not so heavy at the server side. For example, only 50% of the computing power of the CPU (Intel Core 2 Duo CPU E8400 3.00GHz) is needed by a MQTT server to serve more than 60,000 clients.

The MQTT Broker shall provide stable and reliable connectivity. This is achieved with three levels of QoS on logical connection between MQTT client and server. The lowest QoS level does not require acknowledgement and delete message immediately after sending and, hence, successful reception of a message cannot be guaranteed. Meanwhile, acknowledgement is needed for the highest QoS level, and the message can only be deleted from the buffer after a release message is received. Additionally, broker can retain a message in cases of instantaneous connection loss via re-transmission.

The trust of connection for MQTT protocol can be handled through authentication and TLS-based security. A MQTT broker assesses whether a connection is allowed or not with the client ID, username/password, and the client certificate. TLS-based security algorithm is employed to encrypt the pipe over the connection.

In general, MQTT protocol is a mature technology, which has libraries supporting most of the prevalent programming languages including Actionscript, Bash, C / C++, Clojure, Dart, Delphi, Erlang, Elixir, Go, Haskell, Java, Javascript / Node.js, LotusScript, Lua, .NET, Objective-C, OCaml, Perl, PHP, Python, REXX, Prolog, Ruby, Qt, Shell Script, Smalltalk, Swift, and Tcl.

3.2.3 AMQP

AMQP is an open standard dated back to 2003. It is a message-oriented middleware which defines wire protocol to ensure interoperability despite the vendor specific implementation. It supports:

- Queuing – allows to establish a message broker between producer and consumer of the message.
- Filtering – a broker can route messages to appropriate queue (consumer) basing on routing key.
- Reliability levels – various reliability guarantees supported: at-most-once, at-least-once and exactly-once.
- Security – TLS (Transport Layer Security) and SASL (Simple Authentication and Security Layer.

3.2.3.1 Architecture

The specification of AMQP is divided into five documents: Types, Transport, Messaging, Transactions, Security. All constitute a base for reliable, secure exchange of messages between two parties.

3.2.3.1.1 Types

AMQP [4] defines basic data representation types which can be incorporated into the AMQP message. Data type definition allow interoperable exchange of messages between AMQP clients developed by different companies. In addition, data types can be annotated with information relevant to correctly interpret the data sent between parties. Type specification also defines how untyped stream of data can be interpreted by receiving entity. To this purpose AMQP introduces a constructor which defines how to interpret untyped values and eventually transform it into typed ones. Constructor, by means of embedded descriptor, indicates how to produce domain specific type out of primitive typed values sent over the wire.

3.2.3.1.2 Transport

Transport specification defines basic actors and their relationships within AMQP [5]. It recognizes a node as a basic AMQP actor. Nodes are named entities responsible for the safe storage and/or delivery of *messages*. Examples of nodes are: producer, consumer, queue. Nodes exist within a *container*. Each container may hold many nodes. Examples of containers are: client, broker. *Connection* allow communication between containers and consists of a full-duplex, reliably ordered sequence of *frames*. A frame is the unit of work carried on the wire. An AMQP connection is divided into a negotiated number of independent unidirectional *channels*. An AMQP *session* correlates two unidirectional channels to form a bidirectional, sequential conversation between two containers. AMQP actors and their relations are depicted in Figure 5.

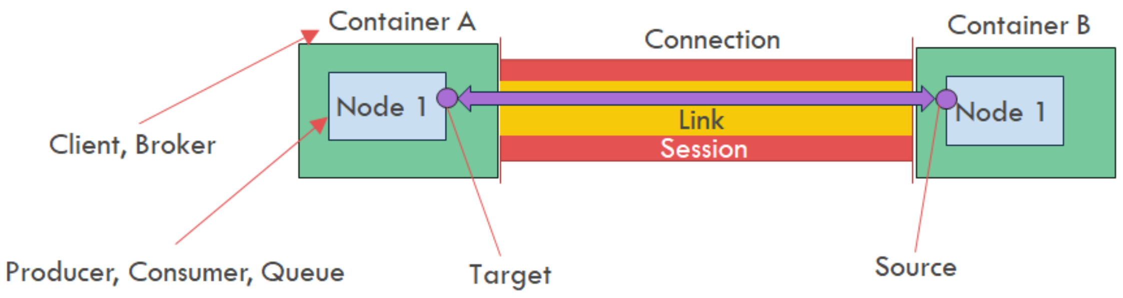


FIGURE 5 AMQP ACTORS AND THEIR RELATIONS

3.2.3.1.3 Messaging

The messaging layer relies heavily on concepts defined in Types and Transport documents [6]. It defines, among others, message format (including headers, footers and properties of the message) and delivery states for messages traveling between nodes. AMQP provides a guarantee that messages are not altered along their end-to-end path. This feature is important since AMQP supports P2P deployment where messages are disseminated by means of middle nodes. In such environment immutability of the message becomes highly desired. For this purpose, AMQP introduces *bare message* which describes message provided by the sender, which may not be changed on the route, and *annotated message* which includes bare message but also information provided by the messaging infrastructure.

3.2.3.1.4 Transactions

Transactions document defines the requirements for transactional messaging which allows a coordinated outcome of many independent transfers [7]. Transactional messaging requires existence of two actors: *transactional resource* and *transactional controller*. Both are containers which communicate over a control link leading a transactional resource to perform a *transactional work*.

3.2.3.1.5 Security

AMQP enables establishment of a secure link between sender and receiver which tunnels messages between two parties [8]. Security aspects include authentication and encryption and are arranged into so called security layers which cover simple or more complex security aspects. Security layers can be tunneled through one another e.g. security layer tackling authentication problem can be tunneled through the encryption security layer. Protocols used for the security purposes by AMQP are TLS and SASL.

3.2.3.2 RabbitMQ – An implementation of AMQP

Developed in 2013 by Pivotal Software, RabbitMQ is an open source messaging broker which is an implementation of AMQP but also supports other messaging protocols such as MQTT or STOMP. The server software (broker) is developed in the Erlang programming language. The client software is available in many popular programing languages.

RabbitMQ provides broker based messaging exchange between parties. Producer Applications create messages and deliver it to a AMQP server called Broker. Inside the broker the messages are routed and filtered by Exchange until they arrive to Queues where Consumer Application are connected and get the messages (see Figure 6).

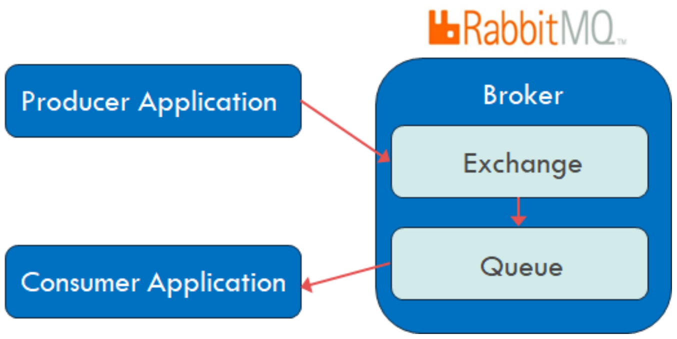


FIGURE 6 RABBITMQ CONCEPT

3.2.3.3 Characteristics

There is little available resources on the performance of AMQP, however, several metrics can be inferred from the specification. These are computing resource requirement, network/connectivity, latency, throughput and refer to RabbitMQ implementation of AMQP:

Computing resource – disk space required for the server is relatively heavy: around 11MB of installation and additionally 50MB for of free space. For client side, the deployment of 500KB is enough. 128MB [9] of available memory should be available at all times in case of a server. High CPU usage was reported for AMQP [10]. With a load of 4000 messages per second, CPU consumption reaches 80%.

Networking/connectivity - Packet size is limited only to the disk space. The connection is done over reliable TCP protocol with the support of three QoS levels.

Latency – the performance in terms of the delay depends on particular settings and is in a range between 0.5ms and 400ms [11]. Therefore, latency can be a concern for particular use cases.

Throughput – Maximum publishing of the RabbitMQ implementation of AMQP were estimated to ~150,000 msg/s [12], while maximum messaging consumption was estimated to be ~65,000 msg/s (few byte message).

3.2.4 Kafka

When Kafka was designed, the purpose was to develop a distributed messaging system to collect and develop high volumes of log data. Nowadays, Kafka has evolved to a distributed streaming platform with three key capabilities: publish/subscribe, process and store. It is mainly used to build two types of applications: real-time streaming data pipelines that reliably get data between systems or applications, and real-time streaming applications that transform or react to the streams of data.

3.2.4.1 Architecture

One can identify four basic concepts in Kafka: producer, consumer, topic and broker.

A topic is a category to which records are published. Kafka is usually deployed as a cluster of brokers. A topic is divided in multiple partitions, and each broker will store one or more of these partitions. A partition is defined as an ordered, immutable sequence of records that is continually appended with new ones. The partitions are distributed over the brokers, with each server handling data and request for a share of the partitions. Each partition is replicated in multiple servers for fault tolerance. So, each partition has one server acting as leader and handling requests, and others as followers who can replace the leader if it fails.

The producers publish data to the topics of their choice. They are responsible for choosing which data to assign to which partition within the topic.

A topic can have many subscribers simultaneously consuming the data that is published. A consumer will read a selected partition. In addition, it can belong to a consumer group, that will be explained later. Kafka uses a "pull model" where the consumers retrieve the messages at the rate they can sustain, instead of a "push model" where the broker forwards the messages.

3.2.4.2 Characteristics

- Simple storage: partitions are implemented as a set of segment files. The broker appends a new message when it is published. A record stored in Kafka is identified by an offset within the partition it belongs, what simplifies the storage. For better performance, the segment files are flushed to disk after a expected number of messages have been published or certain amount of time has elapsed.
- Efficient transfer: a producer can submit a set of messages in a single send request. Also a consumer can pull multiple messages at the same time. Moreover Kafka relies on the underlying filesystem page cache for caching retaining warm cache even when a broker process is restarted. The performance is better than maintaining an in-memory cache or other structure. Finally, the network access for consumers is optimized by using the Unix sendfile API to efficiently deliver bytes in a log segment file from a broker to consumer.
- Stateless broker: The complexity of the broker is reduced by letting the consumer keep track of the information consumed. Since the broker does not know which messages have been read, it follows a time policy to delete them, i.e., a message is deleted when a period of time expires. As a consequence, a consumer can consume records in any order it takes. It could reset to an older offset to reprocess data or skip ahead to the most recent record.
- Distributed coordination: as commented before, in Kafka some consumers can form a consumer group to jointly consume topics. It is possible to have many consumer groups subscribed to the same topic. Thus, each record published to a concrete topic will be received by one consumer instance within a consumer group. Please note that load-

balancing (many consumers in a consumer group) or broadcast (one consumer per group) can be achieved with this design. Consumption is implemented such that each subscriber of a group consumes a share of the total partitions of a topic. Hence consumers within the group does not have to coordinate to consume the partition, only in the event of load rebalance (e.g. when a consumer leaves the group). There is not a master node, consumers coordinate among themselves in a decentralized fashion. Zookeeper helps to achieve it by detecting the addition and removal of brokers and consumers; triggering a rebalance process in consumers maintaining the consumption relationship and consumed offset of a partition. Zookeeper uses four registries to handle different information: the consumer registry (consumer group of a consumer and topics subscribed); the broker registry (broker information and topics and partitions stored in the broker); the ownership registry (stores the partition and the consumer consuming it) and the offset registry (offset of the last consumed message in a partition). When a consumer starts or a consumer is notified of a broker or consumer change, a rebalance process start to determine the new subset of partitions that it will consume within the group. After the partitions are assigned, it will pull data from them and update the offset registry.

- Guarantees: Messages sent by a producer to a particular topic partition will be appended in the order they are sent, but there is no guarantee on the order coming from different partitions. A consumer sees records in the order they are stored in the log. A topic with replication factor N tolerates N-1 server failures without losing records. At-least-once delivery. In some cases, a consumer may receive duplicates messages (for instance, a consumer reads a partition left by a crashed consumer).
- Messaging patterns: Kafka combines publish-subscribe and queuing when using the consumer groups. With a queue, the consumer group allows you to divide up processing over a collection of consumers withing the group. With publish subscribe, it allows you to broadcast to multiple consumer groups.

3.2.4.3 Integrations and clients

Kafka has defined a protocol that defines the available requests and responses and how to use them to implement a client. It is a binary protocol over TCP. A client might have multiple connections to different servers, but for a single broker it is ok to maintain just one connection. Ordering of request is guaranteed in a single TCP connection. When the client request to publish or consume data, the requests must be sent to the broker serving as a leader of the partition otherwise an error will be triggered. Any broker can provide metadata in order to know topics, partitions, leaders etc. and thus the client knows how to proceed. The client has to set the procedure that wants to follow to select the partition. A client can use both the API to send or fetch data in a efficiently way sending some messages in a row. Finally, the client has to agree with the server on the version of protocol used. The client should use the highest commonly supported version, and indicate it in the messages.

There are many clients already implemented in multiple languages for the Kafka platform. This is the list published in their web [20]: C, C++, Python, Go, Erlang, .NET, Clojure, Ruby, Node.js, Proxy (HTTP REST, etc.), Perl, stdin/stdout, PHP, Rust, Storm, Scala DSL, Clojure and Swift. These clients are maintained by Kafka. In addition, Kafka has a built-in framework called Kafka Connect that allows writing sources and sinks to generate data into Kafka or extract it to external systems. So, it makes it simple to define connectors who can collect metrics from the application servers or deliver data from topics to secondary storage. Connectors are maintained outside the main code base.

3.2.5 NATS

NATS is defined as a simple, high performance open source messaging system for cloud native applications, IoT messaging and microservices architectures.

3.2.5.1 Architecture

The architecture of NATS is based on a client-server scheme. Provides a lightweight server maintained by Apcera. There are many client libraries too, some of them maintained by Apcera like Go, Node, Ruby, Java, C, C#, and NGINX C.

3.2.5.2 Protocol

NATS provides a simple, text based, publish/subscribe protocol that makes easy to develop clients. Clients communicate with the server through a regular TCP/IP socket using a small set of protocol operations.

Messages sent by the server:

- INFO. Sent to client after initial TCP/IP connection.
- MSG. Sent to server to specify connection information.
- +OK. Acknowledges well-formed protocol message in verbose mode.
- -ERR. Indicates a protocol error. May cause client disconnection.

Messages sent by the client:

- CONNECT. Sent to server to specify connection information.
- PUB. Publish a message to a subject, with optional reply subject.
- SUB. Subscribe to a subject.
- UNSUB. Unsubscribe from subject.

Messages sent by both of them:

- PING. Ping keep-alive message.
- PONG. Pong keep-alive response.

3.2.5.3 Characteristics

NATS is designed around the following core features: high performance, always-on and available, extremely lightweight and small footprint, support for multiple qualities of service, support for various messaging models and use cases.

Then, we can gather other features:

- Message patterns: pub/sub, queueing and request reply.
- Clustered mode server.
- Auto-pruning of servers. This supports scaling. The server can cut off the connection if the client is slow or does not respond the ping-pong messages. Even disconnects clients that send bad protocol messages.
- Multiple quality of services. It is possible to have at-most-once delivery (TCP) or at-least-once (via NATS Streaming).
- Durable subscriptions. The subscription delivery state is maintained so that durable subscriptions may pick up where they left (NATS Streaming).
- Event streaming service. Messages may be persisted for later replay (NATS Streaming).
- Last/Initial value caching. Subscription delivery can begin with the most recently published message for a subscription (NATS Streaming).

NATS Streaming is a separate service for data streaming that provides additional features built on top of NATS, as seen above. It was added to provide persistence.

3.2.6 Confluent

Confluent Platform is a streaming platform that enables organization and management of data from many different sources with one reliable, high performance system. Provides not only the system to transport data but all the tools needed to connect data sources, applications and data sinks to the platform. Confluent was built by a team that used to work in Kafka before. Confluent uses Kafka as the platform's core and it is supposed to add new features to improve Kafka system. Thus, especially Confluent simplifies connecting data sources to Kafka, building applications with Kafka, securing, monitoring and managing Kafka.

There is a Confluent Open Source platform available. Any new release of Confluent Platform includes the latest release of Apache Kafka.

3.2.6.1 Connectors

Confluent and partner vendors offer tested and secure connectors for many popular systems. Kafka includes a framework called Kafka Connect whose purpose is to make easy to add new systems to Kafka.

To import or export data from Kafka, Kafka connectors need to be instantiated. They are classified as sink or source connectors.

There are available connectors developed and fully supported by Confluent (for instance, Amazon S3 or JDBC). Then, other connectors have been certified since its vendors have met the criteria established (for instance, Azure IoT Hub, IBM Data Replication, Kinetica or SAP HANA). Finally, there are other notable connectors that have been developed utilizing the Kafka Connect framework (for instance, Apache Ignite, Blockchain, Cassandra, DynamoDB, Github, IBM MQ, MongoDB, RabbitMQ, NATS, MQTT).

3.2.6.2 Multi-language support

Confluent Open Source includes clients that allow your Kafka cluster to talk to applications written in many languages: C/C++, Go, Java, JVM, .NET, Python.

3.2.6.3 Schema registry

In a decoupled system like Confluent, the services that interact must agree on a common format for messages. This is called schema. In many systems, these formats are ad-hoc. With the current requirements changes, these formats have to evolve. The Confluent Schema Registry enables safe, zero downtime evolution of schemas by centralizing the management of schemas written for the Avro serialization system. Tracks all versions of schemas used for every topic in Kafka and only allows evolution of schemas according to user-defined compatibility settings.

3.2.6.4 REST Proxy

Confluent provides a RESTful interface to the Kafka interface, making easy to produce and consume messages, view the state of the cluster (set of topics, mapping of partitions to brokers…) or perform administrative actions. The REST API provides more freedom to select languages beyond those for which stable clients exists today. At its core, the REST Proxy wraps the existing libraries provided with Apache Kafka. Confluent ensures that the REST Proxy uses a compatible version.

Requests will include embedded data, using vendor specific content types headers to make the format of the data explicit. Also schemas need to be included in every request to be registered and validated against the schema registry.

The consumers are stateful, as we know from Kafka, and tied to a Proxy instance. This is a violation of the REST principles, although Confluent argues that this is the right design since the consumer group protects the consumers against fails and the added complexity of stateless consumers was not worth it.

Finally, the REST Proxy is designed to be accessed via any load balancing mechanism while maintaining consumer support by letting address an individual instance.

Although using the REST proxy provides a simple HTTP-based interface that should be easily accessible from any language, it adds complexity to the system besides the cost in performance.

3.3 Analysis and selection of EFS messaging/communication protocol

The objective of this survey was to identify the messaging protocols that can be appropriately used for 5G-CORAL research and platform development. The selection was made by taking various factors into account. In particular, the completeness and maturity of the technology was considered. Also, as fog computing resources (the resources pertaining to constrained devices such as a smartphone) are involved in 5G-CORAL solutions, protocols that require high complexity are deemed unsuitable. From these perspectives, the hardware requirement of more sophisticated technologies such as Kafka and Confluent may be too high for certain Fog CDs (for instance, at least 32 or 64 GB of RAM), in spite of the promising performance they potentially can deliver. Although AMQP is relatively lightweight, other technologies such as DDS and MQTT are still superior in terms of hardware requirements.

To this end, we have narrowed the scope down to three candidate protocols, namely DDS, MQTT, and NATS. These technologies are more complete technologies that are able to deliver better performance with reasonable hardware such as Fog resources on constrained devices, and, in the case of DDS and MQTT, more mature.

The NATS system is very similar to Kafka, however NATS broker is really lightweight (the server files are 6.6MB) and it seems that it has the best throughput capacity while keeping the latency below 1ms. It could be deployed in the edge or in the fog using constrained devices. For the users and developers, there are many clients and connectors and even a client protocol to interact with the server. Thus, developing heterogeneous EFS functions or applications would be easier. To provide even more flexibility there are three messaging patterns supported hence no additional constraints to the EFS. It is mostly used for internal communication among services. While the NATS system has suitable characteristics and some implementations [21], the 5G-CORAL consortium deferred NATS for future research because none of the partners has practical experience with NATS.

The low latency performance of DDS (in specific cases, 60μs) makes it very appealing for EFS implementation. The real time publish/subscribe messaging is completed with persistence features for fault tolerance and late joiners. The main difference with other messaging systems is that DDS does not deploy a centralized node or broker to handle the communications. Instead, broadcast is used. Note that 5G-CORAL partner ADLINK has substantial experiences with DDS and hence able to provide in-depth technical support when it comes to implementation.

Unlike DDS, MQTT requires a broker to conduct data publish/subscribe. However, the extremely lightweight makes MQTT a very attractive alternative for EFS. Moreover, 5G-CORAL partner IDCC has already implemented MQTT to transfer information in the robotic use case, and partner AZCOM has already implemented MQTT in the connected cars use case.

A comparison between DDS and MQTT can be found in Table 4. Since 5G-CORAL consortium partners already have experience of implementing DDS and MQTT, and the characteristics of both technologies are suitable for EFS implementation.

TABLE 4 COMPARISON BETWEEN DDS AND MQTT

Protocol	**MQTT**	**DDS**
Deployment	No limitation (LAN, WAN)	DDS routing entity is needed when deployed in WAN
Client Complexity	Light node Client does not need cache Broker can handle reliability	Medium node Reliability is maintained with sufficient cache
Broker	Required	Not required (optional)
Latency	Medium level latency	Low latency (<1ms)
Data Size	<256 Mbytes	<100 Kbytes
QoS	Support 3 QoS levels	Support many more QoS levels
Network Overhead	Low	Medium
Security	Managed by the broker	Key Sharing among clients
Web Performance	Similar to non-web performance	Latency increase in Web Applications
Implementation	Plenty of libraries	Support Fog O5

It is also worth noting that RESTful APIs have been adopted in the 5G Core network defined by 3GPP. This is a clear indication that RESTful protocols should not be precluded due to potential alignment with 3GPP framework. Future activities will investigate the feasibility of RESTful publish/subscribe as a potential messaging/communication protocol for the EFS.

WP2 decided that DDS and MQTT will be adopted as the reference/baseline messaging protocols for the EFS, based on the suitability of each and existing implementations for each use case. WP2 also designated NATS and RESTful publish/subscribe as messaging protocols that merit further study.

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

4.1 Robotics

In this use case, the focus is on the cleaning service provided by robot(s) as described in D1.1 [1]. When a dirty area in the shopping mall is detected, a cleaning operation will be triggered, and a robotic cleaner will be called and guided to the area for the cleaning task execution. In the EFS, location service of the robot will be consumed by a robot navigation application in the EFS, so it can work out the route that the robot should take to arrive at the dirty area to be cleaned. Note that the location of the robot can be evaluated via different means. For instance, iBeacons deployed in the shopping mall could be used for 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 output of the robot navigation application allows a robot control application in the EFS to instruct the movement of the robot (via wireless connectivity, the protocol functionalities of which can be hosted in the EFS too). It is assumed that the robot has on-board sensors and cameras, and the sensor readings as well as the images captured by the camera can be further analyzed in the EFS for sake of finer control; this is particularly useful when the robot is already near the area to be cleaned. It is worth noting that, the EFS computing tasks for this use case, such as robot navigation, robot application, 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 are transparently handled by the OCS. The EFS entities involved in the robotics use case, as well as their interconnection, is illustrated in Figure 7. Table 5 presents a description of the robotics use case EFS entities depicted in Figure 7.

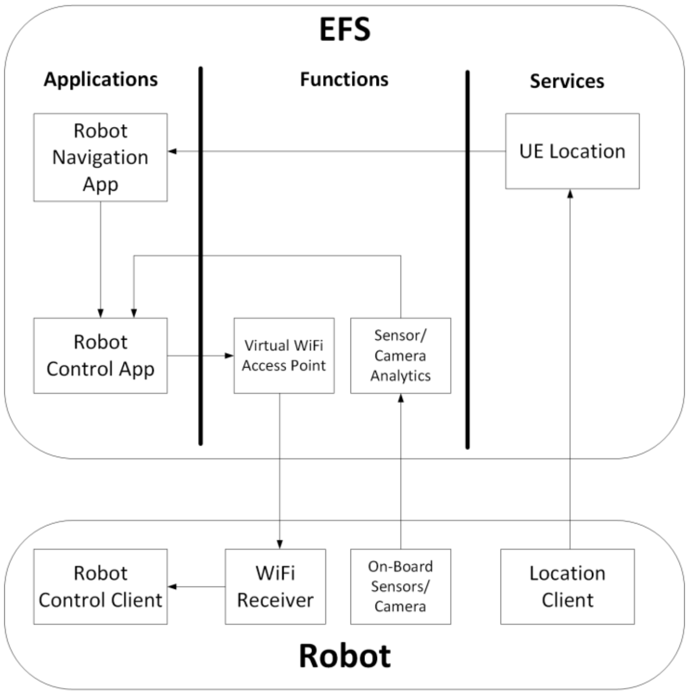


FIGURE 7 EFS ENTITIES INTERCONNECTION FOR THE ROBOTICS USE CASE

TABLE 5 SUMMARY OF EFS ENTITIES FOR ROBOTIC USE CASE

EFS Entity	Description
Robot navigation App	EFS application computing the best route for a robot to reach the area to be cleaned. It consumes data provided by the UE location

	service and communicates with the robot control application taking the necessary actions to guide the robot.
Robot Control App	EFS application in charge of controlling and guiding the robot towards the area to be cleaned. This application provides the robot intelligence which is located inside the EFS platform.
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 avoid connection disruptions. This function is also employed to help robots establish Bluetooth D2D communication in case accurate movement synchronization is required.
Sensor/Camera analytics	EFS function employed for a finer control of the robot. Sensor readings and images coming from on-board sensors and cameras, respectively, are analysed by the EFS platform with the aim of enhancing the control accuracy in the proximity of the target area.
UE location	EFS service which provides the UE geographical position consumed by the robot navigation application to perform the route computation. Robot location can be obtained through different techniques, such as employing iBeacons technology, which allows the UE to estimate and publish its location into the EFS platform.

4.2 Virtual Reality (VR)

VR has been deemed as one of the important use cases of 5G where both ultra-low latency and enhanced broadband communications are needed. Through 5G-CORAL solutions, we may show how VR can be facilitated by the EFS. In particular, we allow the user to expose the information related to its orientation (e.g., the viewing angle) to the EFS service platform, which can be consumed by other VR application-related computing tasks, such as decoding, encoding, composition, and segmentation, that are also hosted in the EFS. Based on the orientation of the UE, the video content captured by the 360-degree camera can be processed in a more appropriate manner prior to being played at the UE side. The EFS entities involved in the VR use case, as well as their interconnection, are illustrated in Figure 8. Table 6 presents a description of the virtual reality use case EFS entities depicted in Figure 8.

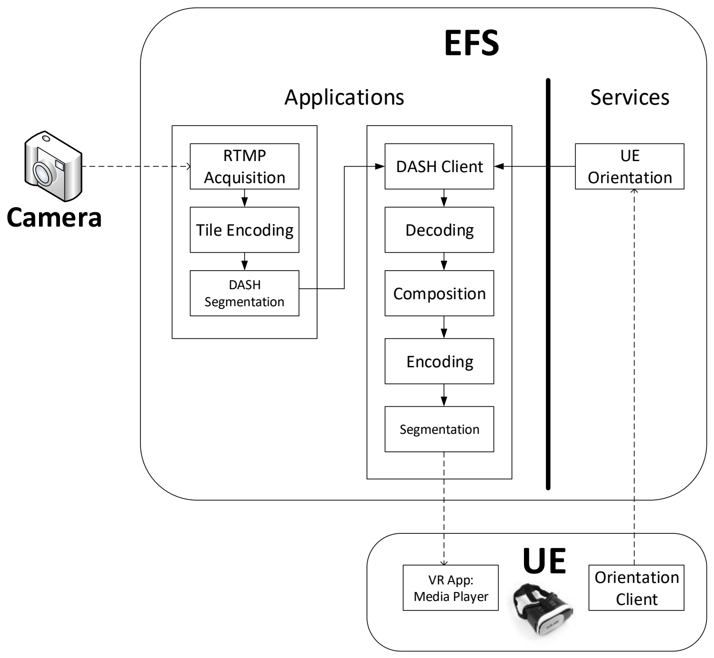


FIGURE 8 EFS ENTITIES INTERCONNECTION FOR THE VR USE CASE

TABLE 6 SUMMARY OF EFS ENTITIES FOR VR USE CASE

EFS Entity	Description
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 service 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.

4.3 Augmented Reality (AR)

The goal of the AR Navigation use case is to provide the assistance to the AR Navigation feature deployed on the UE by nearby Fog CDs running IR processes. The targeted environment for this use case is the shopping mall. Clients of the mall will be equipped with a possibility of using a smartphone app which will provide indoor navigation as well as merchandise promotions form the nearby stores. In order, not to burden their phones with heavy processing and not to deplete the storage of their phones (IR function requires significant disk space), the shopping mall provides an edge network with deployed Fog CDs in the vicinity of the end-users. Through a tight binding with Wi-Fi AP, Fog CDs will be easily accessible by the clients' phones. While Fog CDs host computationally-heavy IR application and Location Estimation function, WiFi AP functionality will be fully virtualised to dynamically control connectivity and as a result decrease communication delay. Functions, services, and applications comprising on AR Navigation use case and the way they are interconnected is depicted in Figure 9. The AR Navigation App helps the user navigate inside the shopping mall. The navigation could be communicated to the user via the users' smartphones, or via screens in the mall which displays guiding information when the user approaches or passes by. The application could run partially on the end devices, and partially in the fog or cloud. The AR Navigation app posts iBeacon localization data that is consumed by Image Recognition (IR) app together with the video frame to be processed. Table 7 presents a description of the augmented reality use case EFS entities depicted in Figure 9.

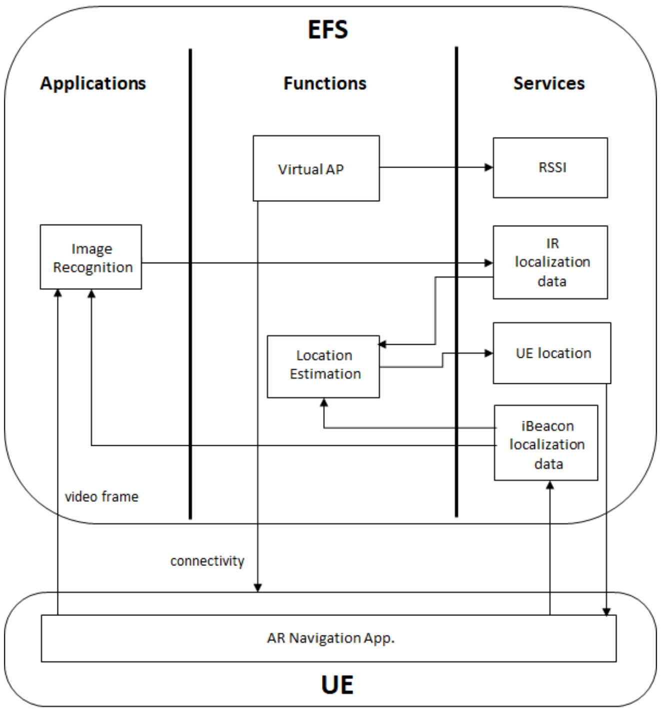
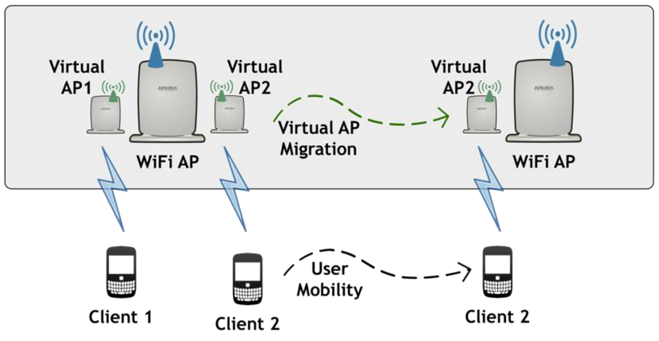


FIGURE 9 EFS ENTITIES INTERCONNECTION FOR THE AR USE CASE

TABLE 7 SUMMARY OF EFS ENTITIES FOR AR USE CASE

EFS Entity	Description
Image recognition	The goal of the Image Recognition (IR) application is to detect objects in an image. To do so, it runs algorithms comparing the picture against a well-known set of images residing in its internal database. In the shopping mall scenario, Image Recognition (IR) processes input video frames sent by the AR navigation application and iBeacon localization data, and communicates with the IR localization data service. After the image recognition process, the application can determine in which zone of the shopping mall a given UE is located. The zone ID is then published as IR localization data.

RSSI	This EFS service exposes the received signal strength of a user towards an Access Point. Received signal strength is for the Wi-Fi access points. The data is reported by the virtual AP.
IR localization data	IR localization data is an output of the Image Recognition (IR) application and conveys information about which area of the shopping mall the UE is located. The granularity of the mall to zone division depends on the performance efficiency of the IR application. The zone ID can be expressed simply as an integer.
iBeacon localization data	iBeacon localization data includes iBeacon ID and signal strength with respect to the UE receiving the beacon signal. This service can estimate the vicinity of the UE to the iBeacon, and when the location of the iBeacon is known, the approximate location of the UE. The iBeacon localization data is provided by the UE and consumed by the Image Recognition application and the Location Estimation function. iBeacon data helps the Image Recognition (IR) application to decrease the processing time by selecting only pictures related to the location of the beacon.
UE location	The UE location service describes the position of the UE in the target environment (e.g. shopping mall). The position has common format (e.g. X, Y, Z coordinates) so that it can be consumed and reused by EFS applications and functions. This service is provided by Location Estimation function.
Virtual AP	Virtual access point (vAP) is network function abstraction that was designed to move the client-AP association decision from the client to the infrastructure. In the IEEE 802.11 standard, the association process begins with the discovery phase, where clients actively scan for APs by generating probe requests. During a scan, APs that respond to the probe message become candidate for association. At this point, the association is defined between the client's MAC address and the BSSID of the AP. The downside of this procedure is, the infrastructure has no control over the client association. This results on connectivity interruption or puts requirements on client devices to mitigate this issue. With the introduction of vAP, each client will be associated with a vAP, that consists of four parameters (1) client MAC address, (2) a virtual AP MAC address, (3) client IP address and (4) the service set identifier (SSID) to be used in the communication. 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. The concept of vAP is illustrated below [17].  Virtual AP1 Virtual AP2 WiFi AP Virtual AP Migration Virtual AP2 WiFi AP User Mobility Client 1 Client 2 Client 2
Location estimation	The goal of the Localization estimation function goal is to estimate relative UE location in the indoor environment. It combines location information from multiple localization sources including iBeacon signal parameters, Image Recognition application, Phone's gyroscope data

	and possibly any other location information from other source (applications/functions). The output of the function is a common location description format (e.g. X, Y, Z coordinates) valid for the environment in focus (e.g. shopping mall). To provide the output, the Location Estimation function must employ translation techniques for different localization metrics such as distance from the known iBeacon device or indoor coordinate estimation based on image recognition techniques into common location format and then combine results to obtain more accurate UE location.

4.4 High-Speed Train

The goal of the high-speed train use case is to provide the seamless connection for passengers especially when they transit from train to the train station. To achieve this goal, EFS devices (such as Fog CDs) will be deployed onboard and on land, which will reduce the traffic to the backhaul network. In the high-speed train Use Case, potentially, hundreds of passengers use different applications, such as AR Navigation or video streaming while they are onboard. Meanwhile, Fog CDs will collect context information related to passengers, by utilizing services in which the application type is extracted from onboard applications, while the QoS requirements are extracted from passenger devices. Then, it will report it to user classifier function which will define classified groups. Besides, the train approximate service will report to vMME function. If the train is approaching, then vMME will utilize the classified group to trigger group handover for user from multi-RAT onboard to multi-RAT on land. This part especially included modification for S1/S10 handover process of vMME deployed on Fog CD, and aggregate part of the S10 handover procedures from hundreds of UEs into several messages. In addition, the service migration will occur from onboard unit to on land units. Figure 10 shows the EFS entities involved in the High-Speed Train Use Case. Table 8 presents a description of the high-speed train use case EFS entities depicted in Figure 10.

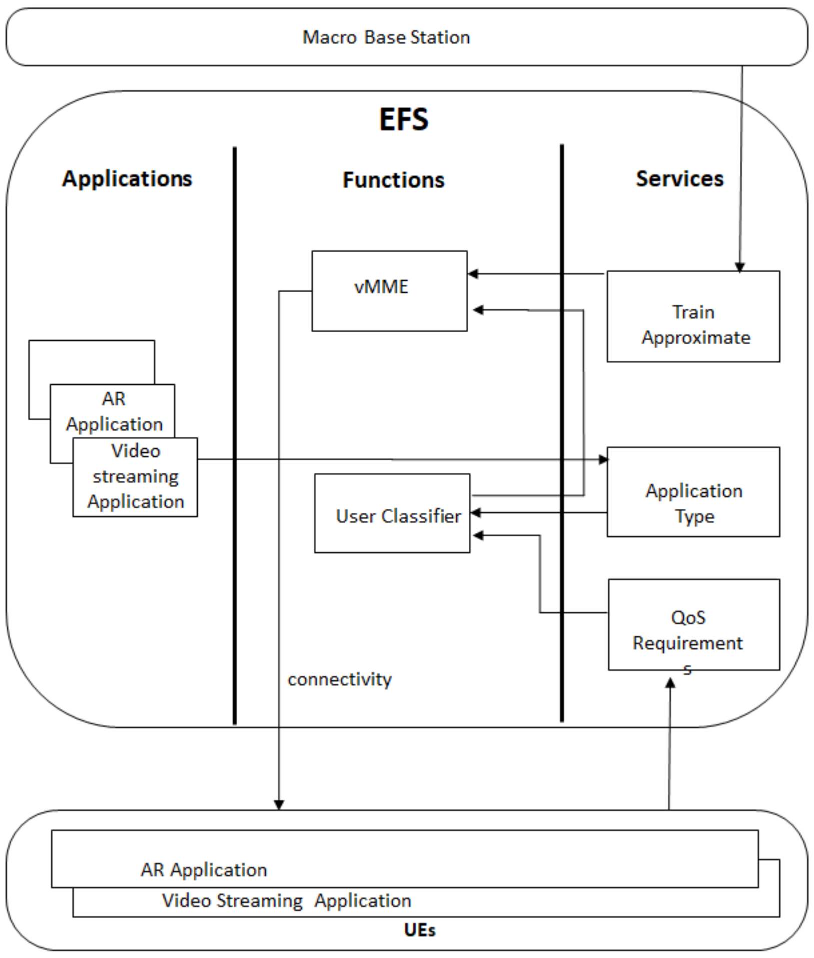


FIGURE 10 EFS ENTITIES INTERCONNECTION FOR THE HIGH-SPEED TRAIN USE CASE

TABLE 8 SUMMARY OF EFS ENTITIES FOR HIGH-SPEED TRAIN USE CASE

EFS Entity	Description
Video streaming app	EFS application for streaming video from the UEs to the EFS.
AR application	EFS application for augmenting user content.
Train approximate	The train approximate position service describes the position of the train on the track of the railway (e.g., Train is near train station co-located with shopping mall). The approximate position is determined based on a comparison of physical cell ID (PCID) mapped in the database with nearby base stations deployed along the railway, and the current obtained PCID. In case the PCID obtained is equal to the one near the train station, then this EFS service will indicate whether the train is near the station or not. Consequently, the EFS can determine if the train is approaching the train station and trigger the functions/applications accordingly.
Application type	This service collects the various application types of end users. The data provided by this service can be utilized to classify users into groups. Application types may include: video streaming, voice call, web browsing, AR, etc.
QoS requirements	The end users on board the train have different QoS requirements This service collects the various QoS requirements of end users. The data provided by this service can be utilized to classify users into groups.

vMME	In LTE EPC architecture, Mobility Management Entity (MME) is the main entity which handles control signaling. MME is responsible for initiating paging and authentication of the mobile device. Also, MME retains location information at the tracking area level for each user and then selects the appropriate gateway during the initial registration process. MME connects to the eNB through the S1-MME interface and connects to S-GW through the S11 interface. MME also plays a vital part in handover signaling. In 5G-CORAL, virtual MME (vMME) is deployed on the Edge/Fog. It has some of the functionalities inherited from the MME and additional functionalities such as: • Initial Attach procedure • Detach procedure • S1-Release • Service Request • Tracking Area Update (TAU) • Handover of a group of users
User classifier	The main goal of this EFS function is to classify users based on context information obtained from multi-RATs (e.g. LTE, Wi-Fi). The user classifier function utilizes the context information such as data session, allocated resources and bearer service to classify end users into several groups. The classification criterion is based on either end user QoS or application types.

4.5 IoT multi-RAT Gateway

In this use-case, technology-agnostic access points are deployed, offering future-proof IoT connectivity. The access point acts as a radio front-end, but signal modulation/demodulation as well as the rest of the communication stack is decoupled and running in the EFS. The AP collects low-level physical-layer information (IQ samples as well as metadata) and makes it available through an EFS service. This is fed to the communication stack, performance enhancing functions, as well as to the localization function, as described in Table 9. Examples of performance enhancement functions include channel blacklisting and scheduling. Localization will exploit rich, low-level radio signals. This location data is used by the User Navigation and Object Localization applications.

In addition, the fact that the architecture enables support for multiple RATs, enables the following:

- Serving multi-RATs in one edge infrastructure.
- Providing EFS services of communication metadata, enabling advanced EFS applications.

The service data flow between the EFS functions, services and applications involved in the IoT multi-RAT gateway are shown in Figure 11.

Note that in this Use-Case, the AP and UE are outside of the EFS: only the communication stack and additional features (localization, interference mitigation) are in the EFS, but the radio head, and IoT devices connected to it, are not.

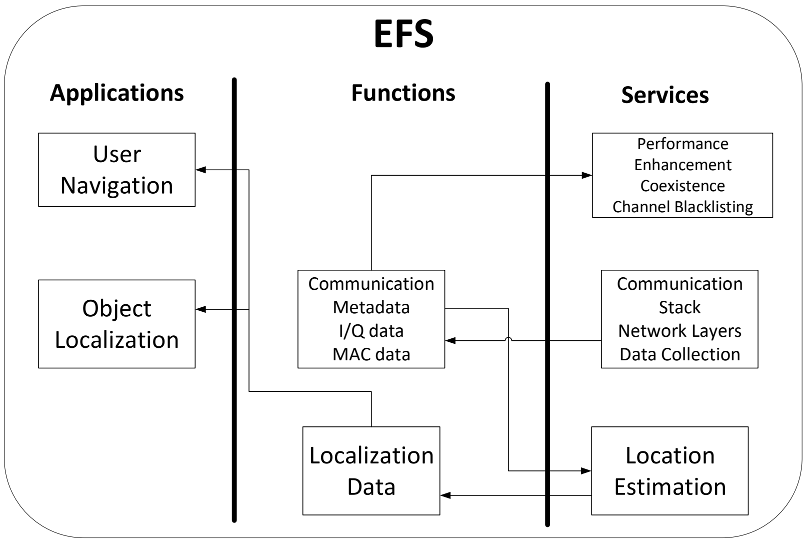

EFS
Applications
Functions
Services
User Navigation
Object Localization
Communication Metadata I/Q data MAC data
Localization Data
Performance Enhancement Coexistence Channel Blacklisting
Communication Stack Network Layers Data Collection
Location Estimation

FIGURE 11 EFS ENTITIES INTERCONNECTION FOR THE IOT MULTI-RAT GATEWAY USE CASE

In a nutshell, the communication stack function gathers low-level radio signals (IQ samples) at the edge. This information is provided as an EFS service, in turn used to provide two functions: (1) enhancing the performance of the communication stack, and (2) computing location estimates. The location data is provided as an EFS service, for use by applications such as User Navigation or Object Localization.

Table 9, presents a description of the IoT Multi-RAT gateway use case EFS entities depicted Figure 11.

TABLE 9 SUMMARY OF EFS ENTITIES FOR IOT MULTI-RAT GATEWAY USE CASE

EFS Entity	Description
User navigation	EFS application deployed to provide users with navigation inside the shopping mall. It requires localization input supplied by the Localization Data service.
Object localization	Potentially hundreds of objects such as trash cans or advertisement boards are deployed in the shopping mall, and need to be localized automatically. For instance, trashcans can report when they are full, and boards can be used for targeted ads. This application relies on the same technology as the End-User Navigation.
Communication metadata	Provides multi-RAT communication metadata for applications. This consists mostly of low-level radio signals, i.e. IQ samples gathered from IoT gateways. The data is provided by the IoT communication stack and used by the IoT performance enhancement and the localization function.
Localization data	This service provides localization data from IQ signals in the multi-RAT IoT gateway context. The location is in the form of coordinates or area identifier, for applications such as User Navigation or Object localization.
Performance enhancement	A set of Performance Enhancing Functions (PEFs) IoT communication technologies, e.g.: IEEE 802.15.4 at both 2.4 Ghz and subGHz, BLE, NB-IoT. These PEFs rely on low-level signal information (IQ signals and metadata, available through the EFS service platform) to learn

	about the radio environment and then configure devices for improved co-existence, throughput, latency, reliability etc.
Communication stack	The Multi-RAT communication stack implements the IoT communication layers (L1, L2 etc.) and related functions (e.g. management and control functions). Each of them acts as a virtualised modem for one IoT technology, same as the cloud-RAN concept (cloudifying the baseband functions). Example IoT technology under considerations are IEEE 802.15.4 (both non-beacon-enabled and TSCH), NarrowBand-IoT, and Bluetooth Low-Energy (BLE), as well as LoRa.
Location estimation	Exploits low-level IQ samples for location estimation, particularly, IQs as well as communication stack metadata available through the EFS service platform. Potentially leverages phase difference between multiple antennas at each anchor, and uses DTOA (differential time of arrival) between different anchors to infer location. This function publishes its data through the *Localization Data* EFS service.

4.6 Connected Cars

This section is focused on the Connected Cars Use Cases, which is composed of safety and infotainment entities. Safety entities manages functions that warn the driver of external hazards and internal responses of the vehicle to hazards (e.g. vehicle condition and service reminders, remote operation, transfer of usage data) and Infotainment entities manages functions involving the entertainment for the driver and passengers (e.g. smartphone interface, WLAN hotspot, music, video, internet, social media, mobile office).

4.6.1 Safety

For this application, the solution offers a supported driving after processing context data from the traffic environment. The driver receives signals and indications to help the driving. The EFS entities used in the safety application are shown in Figure 12.

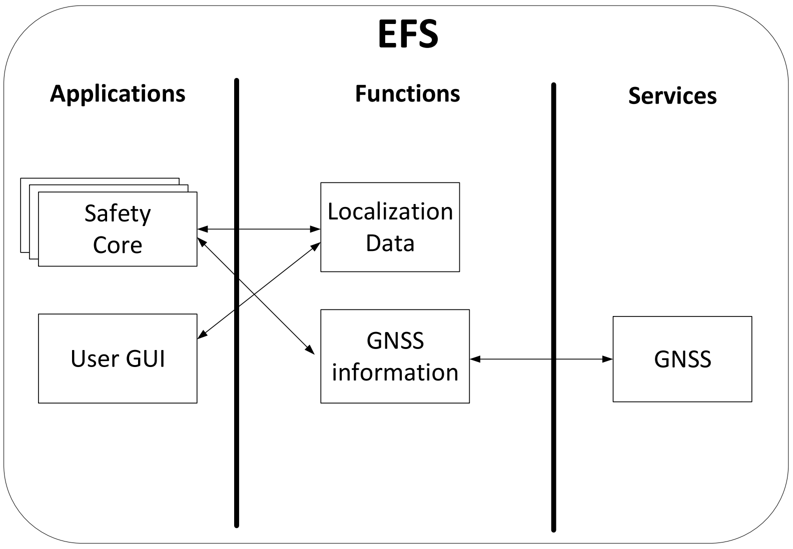


FIGURE 12 EFS ENTITIES INTERCONNECTION FOR THE SAFETY CONNECTED CAR USE CASE.

The safety User GUI application provides the user graphic interface for this scenario. It receives all the data needed to warn the driver. It could also control, for instance, the braking system to automatically perform safety actions. The application is executed on top of the vehicle software platform with access to, e.g., a screen of lights panel. Moreover, Safety Core can collect the position information provided by the surrounding vehicles, through the GNSS Information EFS Service, and collected by on-board modules. This information can be processed by the application, which can send alerts regarding the existence of a risk of collision with other vehicles, for instance. These operations can be done not only at the edge cloud, but also at the vehicle or at the RSU i.e. on a traffic light. These are all fog nodes, as proposed in 5G CORAL.

The information sent to the User Interface application (GUI) is generated by the Safety Core application (SC) which coordinates the User GUI applications and GNSS functions that forms this whole application. It stores context data and generates alarms based on the processed information to later send actions to the drivers. Event Information service (EI) interconnects the two applications. EI is used to handle context events. Some information that can be exchanged is **event name, context data, priority** and **action.**

The context information regarding location is received by the service GNSS Information with origin in the GNSS function. Data from multiple vehicles will be stored. Also, since the Safety Core application (SC) is a distributed application, Event Information service (EI) will be used to communicate different instances of the core.

In addition, the context information generated formed by events and location could be the basis for other future applications, for instance a real-time route generator, where a vehicle navigator could leverage the information when selecting the best path to arrive the destination. The exchange and process of the data can be used as inputs for generating a road map and therefore select the best route.

4.6.2 Infotainment

Infotainment is an EFS Application which aims to enhance the experience and entertainment of the passengers through multimedia (e.g., 4K videos) and gaming contents especially considering the slow-moving vehicles (or the parked ones) situation as could be a traffic jam[13], [14]. This EFS Application collects the position and velocity information from the GNSS Information EFS Service to understand if there is a traffic jam like condition enabling contents local caching in Road Side Unit (RSU) or even the vehicles. The infotainment EFS application allows the users to exchange local cached content leveraging on the 5G-CORAL network, improving the user experience but without requesting the content from a distant data cloud and so overloading the network backhauling. In Figure 13 the EFS entities involved in the infotainment application are depicted.

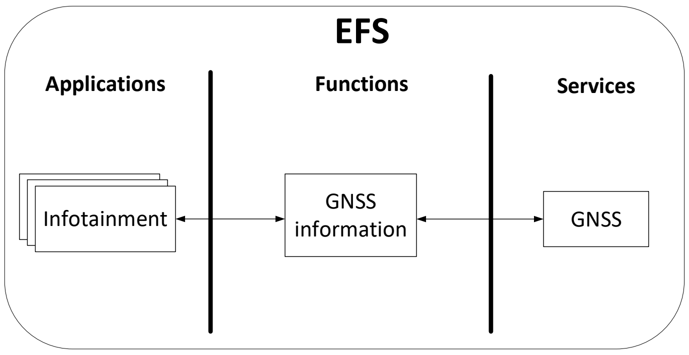


FIGURE 13 EFS ENTITIES INTERCONNECTION FOR THE INFOTAINMENT CONNECTED CAR USE CASE.

In a traffic jam like condition the vehicle or RSU can start local caching [15], [16] and so as gateway send information, i.e. video gaming, web browsing and so on, to the other vehicles based on the information received. This infotainment application has not critical requirements in term of latency, but requires a large data throughput when it works as gateway since several other vehicles/users may use the applications simultaneously. The information sent to the user is generated by the application when a traffic jam like situation happens. The context information regarding location is received by the service with origin in the GNSS function as described in Table 10.

Table 10, presents a description of the connected car use case EFS entities depicted in Figure 12 and Figure 13.

TABLE 10 SUMMARY OF EFS ENTITIES FOR CONNECTED CARS USE CASE

EFS Entity	Description
Safety core	EFS application providing road safety warning messages. It communicates with the Event Service and the GNSS information service. It can collect location information provided by surrounding vehicles and generated by on-board modules.
User GUI	EFS application providing User Graphical Interface.
Infotainment	Infotainment application aims to enhance the experience and entertainment of the passengers through multimedia (e.g., 4K videos) and gaming contents. It receives the data from the GNSS applications and if a traffic jam like situation is recognised. It also enables multimedia content local caching between RSUs and vehicles.
Event service	EFS service used to handle context events. It exchanges data with User GUI and Safety Core applications, such as event name, context data, priority and action.
GNSS information	A GNSS device will collect information about the location of a vehicle. Some of the data that can be gathered is latitude, longitude, speed or altitude. This data can be used by applications like infotainment.
GNSS	The aim of the Global Navigation Satellite System (GNSS) function is to process and transform location data received from a satellite to device in a vehicle. It exposes the data through the EFS localization service. It requires access to the signals from a GNSS antenna. The GNSS function can be either installed close to an antenna or at a different location given it is provided the antenna data.

5 Conclusions and Future Work

Nine months after the start of the 5G-CORAL project, WP2 has reached a preliminary design of the Edge and Fog Computing System (EFS) as reported in this deliverable. This initial release addressed the following aspects of the 5G-CORAL EFS, namely: the EFS requirements; the EFS architecture including internal and external interfaces; a survey, analysis and selection of the EFS Service platform messaging/communication protocols; and a baseline EFS design for the 5G-CORAL use cases.

In section 2, we presented a comprehensive description of the 5G-CORAL Edge and Fog Computing System (EFS) requirements and architectural design including: the EFS virtualisation infrastructure (i.e. physical/virtual compute, storage and networking), the EFS entities (i.e. EFS applications, EFS functions, EFS service platform and the respective entity managers), the EFS internal and external interfaces. Section 2 also provided a detailed description of the microservices based design principle for EFS atomic entities, i.e. the building blocks of EFS entities. Finally, section 2, provided a detailed description of the publish/subscribe communication framework between the EFS service platform and the EFS/non-EFS applications and functions.

In section 3, we presented an in-depth analysis of state of the art messaging/communication protocols with the objective of identifying candidate protocols for the 5G-CORAL EFS service platform. DDS and MQTT will be adopted as the reference/baseline messaging protocols for the EFS, while NATS and RESTful publish/subscribe were earmarked for further study.

Finally, in section 4 a baseline EFS design for each 5G-CORAL use case was presented, namely: Robotics, Virtual Reality (VR), Augmented Reality (AR), High-speed Train, IoT Multi-RAT Gateway and Connected Cars. The baseline design decomposed each use case into the constituent EFS entities and described their respective interworking(s).

The next deliverable from WP2 (D2.2, due Month 21) will extend the present 5G-CORAL EFS design to include:

- Association/pairing between the 5G-CORAL EFS and 5G-CORAL OCS.
- The design of the EFS service platform data storage engine.
- A study of NATS and RESTful publish/subscribe messaging.
- A refined description of EFS internal and external interfaces.
- EFS workflows for resource/service discovery and integration.
- Data models for the EFS APIs.
- EFS implementations for the 5G-CORAL use cases.

Bibliography

[1] D1.1 5G CORAL Initial system design, use cases, and requirements. 5G CORAL Project.

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

[3] ETSI, "Network Functions Virtualisation (NFV); NFV Evolution and Ecosystem: Hardware Interoperability Requirements Specification," ETSI GS NFV-EVE 007 V3.1.1, March 2017.

[4] http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-types-v1.0-os.html#toc *(last accessed 2018/05/30)*

[5] http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-transport-v1.0-os.html#toc *(last accessed 2018/05/30)*

[6] http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-messaging-v1.0-os.html#toc *(last accessed 2018/05/30)*

[7] http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-transactions-v1.0-os.html#toc *(last accessed 2018/05/30)*

[8] http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-security-v1.0-os.html#toc *(last accessed 2018/05/30)*

[9] https://www.rabbitmq.com/production-checklist.html *(last accessed 2018/05/30)*

[10] https://groups.google.com/forum/#!topic/rabbitmq-users/RcSja7QZLtc *(last accessed 2018/05/30)*

[11] https://bravenewgeek.com/benchmarking-message-queue-latency/ *(last accessed 2018/05/30)*

[12] https://www.rabbitmq.com/blog/2012/04/25/rabbitmq-performance-measurements-part-2/ *(last accessed 2018/05/30)*

[13] EU H2020 5G NORMA, “D2.1: Use cases, scenarios, and requirements”, September 2015

[14] EU H2020 ERTICO, “5G Automotive Vision”, October 2015

[15] Xiaohu Ge, Zipeng Li, Shikuan Li, “5G Software Defined Vehicular Networks”, IEEE Communications Magazine,July 2017

[16] M. Gregory, J. Gomez-Vilardebo, J. Matamoros and D. Gunduz, “Wireless content caching for small cell and D2D networks”, IEEE Journal vol. 34, 5, May, 2016.

[17] Y. Grunenberger and F. Rousseau, “Virtual access points for transparent mobility in wireless LANs,” in IEEE Wireless Communications and Networking Conference (WCNC), 2010, pp. 1–6.

[18] 3GPP, “Study on Architecture for Next Generation System,” TR 23.799, v0.8.0, September 2016.

[19] Sam Newman, “Building Microservices: Designing Fine-Grained Systems”, first edition, O’Reilly, February 2015.

[20] https://kafka.apache.org/ *(last accessed 2018/05/30)*

[21] https://nats.io/ *(last accessed 2018/05/30)*