ETSI TR 103 527 V1.1.1 (2018-07)

TECHNICAL REPORT

SmartM2M;
Virtualized IoT Architectures with Cloud Back-ends

Reference

DTR/SmartM2M-103527

Keywords

cloud, IoT, virtualisation

Contents

List of figures

Intellectual Property Rights

Foreword

This Technical Report (TR) has been produced by ETSI Technical Committee Smart Machine-to-Machine communications (SmartM2M).

Modal verbs terminology

In the present document "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

Introduction

In addition to interoperability and security that are two recognized key enablers to the development of large IoT systems, a new one is emerging as another key condition of success: virtualization. The deployment of IoT systems will occur not just within closed and secure administrative domains but also over architectures that support the dynamic usage of resources that are provided by virtualization techniques over cloud back-ends.

This new challenge for IoT requires that the elements of an IoT system can work in a fully interoperable, secure and dynamically configurable manner with other elements (devices, gateways, storage, etc.) that are deployed in different operational and contractual conditions. To this extent, the current architectures of IoT will have to be aligned with those that support the deployment of cloud-based systems (private, public, etc.).

Moreover, these architectures will have to support very diverse and often stringent non-functional requirements such as scalability, reliability, fault tolerance, massive data, security. This will require very flexible architectures for the elements (e.g. the application servers) that will support the virtualized IoT services, as well as very efficient and highly modular implementations that will make a massive usage of Open Source components.

These architectures and these implementations form a new approach to IoT systems and the solutions that the present document investigates also should be validated: to this extent, a Proof-of-Concept implementation involving a massive number of virtualized elements has been made.

The present document is one of three Technical Reports addressing this issue:

- ETSI TR 103 527 (the present document): "Virtualized IoT Architectures with Cloud Back-ends" (the present document);
- ETSI TR 103 528 [i.1]: "Landscape for open source and standards for cloud native software for a Virtualized IoT service layer";
- ETSI TR 103 529 [i.2]: "Virtualized IoT over Cloud back-ends: A Proof of Concept".

1 Scope

The present document:

- makes a description of some use cases that benefit from virtualization and outlines which one will be used for the Proof-of-Concept that is described in depth in ETSI TR 103 529 [i.2];
- addresses the rationale and requirements for the use of virtualization - and of the cloud in general - in support of IoT systems. It also introduces some features that will be key for the definition and further implementation of virtualized IoT systems such as microservices;
- provides the identification of new architectural elements (components, mappings, Application Programming Interfaces (API), etc.) that are required to address IoT on a cloud back-end. In particular, one objective of the present document is to describe how current IoT nodes e.g. the oneM2M CSE, can be modified and improved by the introduction of micro-services.

2 References

2.1 Normative references

Normative references are not applicable in the present document.

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long-term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] ETSI TR 103 528: "SmartM2M; Landscape for open source and standards for cloud native software applicable for a Virtualized IoT service layer", 2018.

[i.2] ETSI TR 103 529: "SmartM2M; IoT over Cloud back-ends: a Proof of Concept", 2018.

[i.3] ITU-T News: "What is 'cloud-native IoT' and why does it matter?", October 2017.

NOTE: Available at http://news.itu.int/what-is-cloud-native-iot-why-does-it-matter/.

[i.4] Amazon Web Services: "What is Auto-scaling".

NOTE: Available at http://docs.aws.amazon.com/autoscaling/latest/userguide/WhatIsAutoScaling.html.

[i.5] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation).

NOTE: Available at https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=celex%3A32016R0679.

[i.6] Deloitte: "Data Privacy in the cloud", 2016.

NOTE: Available at https://www2.deloitte.com/content/dam/Deloitte/ca/Documents/risk/ca-en-risk-privacy-in-the-cloud-pov.PDF.

[i.7] ETSI TS 118 101 (V2.10.0): "oneM2M; Functional Architecture (oneM2M TS-0001 version 2.10.0 Release 2)".

[i.8] Recommendation ITU-T Y.3600: "Big data - Cloud computing-based requirements and capabilities", 2015.

[i.9] ETSI GS NFV 002: "Network Functions Virtualisation (NFV); Architectural Framework".

[i.10] ETSI GS NFV-INF 001: "Network Functions Virtualisation (NFV); Infrastructure Overview".

3 Definitions and abbreviations

3.1 Definitions

For the purposes of the present document, the following terms and definitions apply:

Open Source Software (OSS): computer software that is available in source code form

NOTE: The source code and certain other rights normally reserved for copyright holders are provided under an open-source license that permits users to study, change, improve and at times also to distribute the software.

source code: any collection of computer instructions written using some human-readable computer language, usually as text

standard: output from an SSO

Standards Setting Organization (SSO): any entity whose primary activities are developing, coordinating, promulgating, revising, amending, reissuing, interpreting or otherwise maintaining standards that address the interests of a wide base of users outside the standards development organization

NOTE: In the present document, SSO is used equally for both Standards Setting Organization or Standards Developing Organization (SDO).

3.2 Abbreviations

For the purposes of the present document, the following abbreviations apply:

AE	Application Entity (in oneM2M)
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
ARM	Acorn RISC Machine architecture
BCP	Best Common Practices
CAPEX	Capital Expenditure
CEP	Complex Event Processing
CoAP	Constrained Application Protocol
CPU	Central Processing Unit
CSC	Cloud Service Customer
CSE	Common Services Entity (in oneM2M)
CSF	Common Service Function
CSP	Cloud Service Provider
DDoS	Distributed Denial of Service
EU	European Union
GDPR	Global Data Protection Regulation
HLA	High Level Architecture
HTTP	HyperText Transfer Protocol
IaaS	Infrastructure as a Service
IAM	Identity and Access Management
ICT	Information and Communication Technology
IoT	Internet of Things
IP	Internet Protocol
IPC	Inter-Process Communication
IPE	Interworking Proxy Entity (in oneM2M)

ISG	Industry Specification Group
IT	Information Technology
MANO	MANagement and Organization (in NFV)
MQTT	Message Queuing Telemetry Transport
NFV	Network Function Virtualisation
NFVI	NFV Infrastructure
ONAP	Open Network Automation Platform
OSM	Open Source Mano (in ETSI)
OSS	Open Source Software
PaaS	Platform as a Service
PoC	Proof-of-Concept
PoP	Point of Presence
SaaS	Software as a Service
SDO	Standards Development Organization
SE	Service Entity (in oneM2M)
SPOF	Single Point Of Failure
SSO	Standards Setting Organization
UC	Use Case
URI	Uniform Resource Identifier
VM	Virtual Machine
VNF	Virtualized Network Function

4 Rationale for IoT Virtualization

4.1 IoT: towards massive deployments

The focus of IoT in the recent years has been on connecting devices and applications. To this extent, a number of standards, frameworks, solutions have been developed. Now that the maturation of the industry is progressing rapidly, IoT is facing to major challenges.

On the one hand, connected devices as well as applications have to be integrated with existing, evolving or entirely new business processes: this creates the need for very adaptive frameworks that offer the possibility to easily introduce new applications and to ensure that they are properly connected to the existing enterprise systems, and to process enormous quantity of data.

One the other hand, IoT systems are transitioning from proof-of-concept deployments or new projects with limited size and scope towards full-fledge systems. These new systems may require extremely high numbers of connected devices (thus generating needs for scalability or deployment automation) as well as stringent non-functional requirements (such as low latency).

In both cases, new IoT systems will require a high degree of availability, adaptability and flexibility. In particular, the resources used by those systems may have to be very dynamic, both in terms of configuration and run-time flexibility. The models provided by Cloud Computing, which have been designed upfront with these two requirements in mind, seem very attractive in this context.

4.2 Cloud Computing and Virtualization

Cloud computing is allowing the provision of very sophisticated capabilities; for computing, storage, analytics, etc.; to very dynamic and potentially massive number of users. Those capabilities are provided as services (Platform-as-a-Service, Infrastructure-as-a-Service; Software-as-a-Service; etc.) that provides functional and also non-functional support (e.g. low latency fault-tolerance, horizontal scalability, cost-optimization, or geo-optimization together with Service Level Agreements (SLAs), and security.

The technical capabilities of cloud computing technology made it possible to provide the most demanding information and communication technology (ICT) infrastructures, such as communication networks, from specialized hardware and software to new software paradigms, referred to as 'cloud-native'.

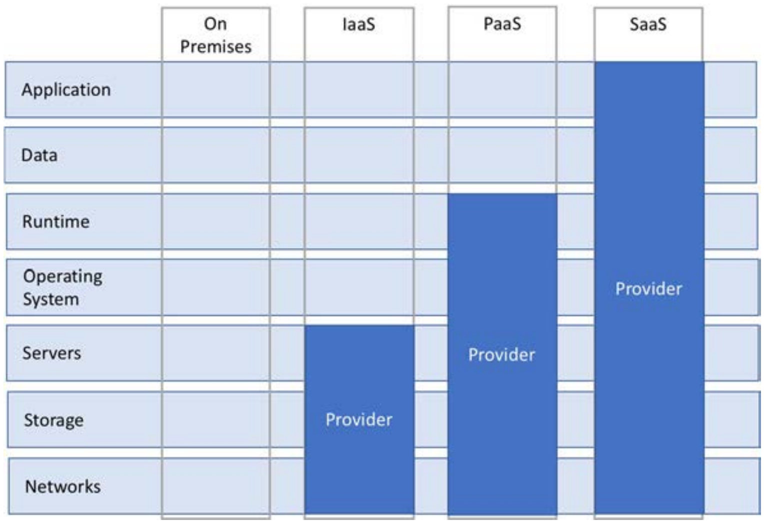

Figure 1: Options for adoption of Cloud Native solutions

The expectation of Cloud-Native applications is to benefit from offerings from Cloud Service Providers (CSP) that may cover parts or all of the layers of Virtualized application, via Infrastructure as a Service (IaaS), Platform as a Service (PaaS) or Software as a Service (SaaS). Figure 1 presents the possible usages of such offerings in delegating more and more important parts of the underlying layers to a third-party in charge of hiding complexity, resource usage, etc.

4.3 The new challenge: combining IoT and Cloud Computing

The IoT industry starts to understand the potential benefits of combining the strengths of both IoT and Cloud industries in a new value proposition (see [i.3] for example). IoT virtualization - i.e. IoT built on cloud-native principles - is to IoT platforms as what Network Function Virtualisation (NFV) is to communication networks.

When applied to IoT, virtualization is expected to provide technical benefits such as more flexibility on assigning IoT virtualized objects and functions to physical resources. Moreover, virtualization should bring as well financial benefits (e.g. greater CAPEX efficiency) or operational benefits (e.g. improvement of automation and operating procedures) altogether resulting in boosted service innovation.

The scope of IoT standards and protocols has so far focused on interface specifications and related data models. Developing IoT platforms that use cloud-native principles will benefit from guidelines and Best Common Practices (BCP) in building operational grade IT applications using cloud technologies.

The convergence of cloud and IoT is of major important to those (e.g. architects) aiming at building IoT solutions that can dynamically reach massive scale in support of large IoT deployments, e.g. in Smart Cities. It is of great importance for technical actors of IoT to benefit from guidelines for IoT virtualization, in particular regarding the 'containerisation' of IoT applications.

4.4 Content of the report

Clause 5 provides a number of Use Cases (UC) that could benefit from virtualization of IoT. Each UC is described from a functional standpoint, together with the expectations towards virtualization. Finally, one UC is highlighted since it is the one that will be selected for implementation (as it is described in ETSI TR 103 529 [i.2]).

Clause 6 of the present document presents the Cloud Computing features that are relevant in the context of IoT Virtualization. First, some functional requirements are introduced that correspond to specific functionalities that are (better) supported by Virtualization. Similarly, some non-functional requirements are presented that are expected to be specially supported through Virtualization. Finally, two key features that will play a key role in architectures and implementations: microservices and inter-process communications.

Clause 7 investigates the main dimensions that will be addressed in order to define layered architectures supported by microservices. A reference model for such an architecture is introduced that also serves as a basis for the description of the "Landscape of Open Source and Standards" that is developed in ETSI TR 103 528 [i.1].

Clause 8 summarizes the main finding of the present document and provides a set of recommendations for architects and developers in charge of potential IoT virtualization projects.

Annex A is addressing the relationship of IoT with Big Data and, in particular, regarding the question of Data Quality and some potential solutions.

5 Some use cases for IoT Virtualization

5.1 Introduction

This clause introduces a (limited) number of generic Use Cases (UCs) that are illustrative of the expected benefits and potential challenges of IoT virtualization. There is probably a large number of Use Cases for which a "traditional" (i.e. non-virtualized) approach can and will apply. However, the introduction of IoT Virtualization is expected to make some UCs more effective: it would generally improve the efficiency of their implementation or support interoperability at a more fine-grained level (or both).

For the presentation of UCs, rather than present them based on the needs of a given business domain (aka a "vertical"), the approach taken is to present the major features outlined by a class of applications in different "verticals". The name of the UC will refer to the major underlying feature involved (e.g. fault-tolerance, data privacy, etc.).

5.2 Horizontal up and down Auto-Scaling

The amount and type of data transmitted by IoT devices may vary drastically in time depending on some events that can be internal or external to the virtualized IoT system (e.g. road traffic increase during holiday departure).

A cloud-native IoT platform shall be able to continuously monitor its resources, scale-up its capabilities when needed, then scale-down to an optimized state to avoid wasting resources. This capability is referred to as "Auto-Scaling" (see [i.4] for example).

The main objective of Auto Scaling is to ensure that the number of Virtual Machine (VM) instances available for and used by the virtualized application are optimal at a given time. Practically, a minimum number of VM instances is defined (lower threshold for the auto-scaling down) as well as a maximum number (upper threshold for the auto-scaling up). When needed, additional VMs are added (with an increment that can be predefined), used as long as needed and released when the usage is no longer needed.

This UC can be illustrated by a number of examples taken from various verticals:

- Intelligent Transport Systems with a sudden increase in traffic (e.g. vacations).
- Electrical (Smart) Grids with burst reconnection of IoT devices after a power cut.
- Smart Metering with burst transmission of intelligent meters data at given time slots.
- And many more, etc.

The benefits of Auto Scaling are largely related to the non-functional support it provides to the virtualized applications:

- Improved availability: the virtualized application has, at any time, the best adjusted capacity to deal with the most complex and hard to predict traffic patterns.
- Improved fault tolerance: when an instance (or a group of) VM(s) does not function properly, Auto-Scaling allows to quickly terminate it and launch an adequate replacement.
- More effective cost management: thanks to the dynamic increase and decrease of the needed capacity, the usage is constantly adjusted to reduce the consumption, hence the cost, of the computing resources.

5.3 No single point of failure

Like every other ICT systems, IoT systems can be built with one or more Single Point(s) Of Failure (SPOF). A SPOF is the results of a flaw in the design, implementation or configuration of the system. It introduces a potentially very high risk since one fault or malfunction may cause the entire system to stop operating. As an example, IoT servers and gateways can be a SPOF in an IoT system: in a poorly designed architecture, when a server or a gateway goes down, critical functions may stop.

Preventing that catastrophic outcome is possible, as long as the architects and designers can identify the SPOFs that appear in the system's design and implement corrective measures - as long as this is both feasible and cost-effective. The techniques that make this possible are largely based on clustering and replication. In principle, such techniques are supported by a cloud-native IoT architecture and supported by the major cloud infrastructure providers.

Consequently, the possibility of providing this kind of support will be a major support for a large number of Use Cases in a very large array of vertical domains. Some examples of such verticals and related UCs and related are shown below:

- Health Care: remote patient examination, monitoring and surgery.
- Emergency Services: allowing Emergency Management Teams to access patient records while on the road.
- Railway systems: improved signalling with reliable communications and better integration with system.
- And many more, etc.

Cloud Computing is often seen as a way to avoid the single points of failure with built-in redundancy or clustering services. It is important to note that Auto-Scaling is adding to the effectiveness of redundancy (by allowing to quickly terminate malfunctioning VMs and launch an adequate replacement, as noted above) and is an important element in the process of fixing the SPOFs in the cloud.

However, IoT Virtualization is not an absolute panacea, in the sense that there are still SPOFs that may be associated to the use of Cloud Computing resources. An example of such SPOF is a centralized (and not replicated) monitoring function that may be compromised in case of malicious attack (e.g. DDoS). Similarly, the possibility of the failure of one cloud provider is another example of SPOF. So, architects and developers of Virtualized IoT systems will still have to ensure that all SPOFs are identified and corrected, but it is expected that their number will be lower than in "traditional" (non-virtualized) IoT systems.

5.4 Data privacy

Potentially, a wide range of IoT applications are impacted by data privacy and the need for Data Protection. The amount of data that IoT devices can generate is enormous (with billions of data points created every day) and is leaving a lot of this information vulnerable and susceptible to malicious usage (by hackers in the extreme case, but also by unauthorized competitors).

IoT virtualization is also impacted by the question of data privacy. For instance, when a Cloud Service Provider (CSP) is used, it is important to ensure that the data belonging to the Cloud Service Customer (CSC) is well protected by the CSP that provides the Cloud service used by the CSC. In general, the inability to address these problems may create a lack of trust that, in turn, may reduce the consumers' appetite for purchasing connected products, and prevent the IoT from fulfilling its true potential.

The data privacy Use Case can be illustrated by a number of examples taken from various verticals:

- Electrical Smart Grids: collection of user data and anonymization.
- Agriculture: exchange of data between equipment, private storage of field information (crop, etc.).
- Connected cars: security coming together with guaranteed privacy policies.
- And many more, etc.

The reliance of IoT systems on the use of data and the potential issues associated to its management (and protection) have increased the need for ensuring that information is well protected by guaranteed safeguards with a good level of transparency. The question of privacy is going to be shaped by the General Data Protection Regulation (GDPR [i.5]) that is about to enter in force in Europe. GDPR is harmonising the current data protection regulations across EU member states, with strict data compliance stipulations and potential huge financial penalties for those when the rules are breached.

GDPR is not dealing specifically with Cloud Service Providers, but it has implications for organizations that use cloud services to store data. As stated in [i.6], "*[...], the adoption of cloud computing raises challenges in the face of new, and often competing, privacy regulations across various jurisdictions, as well as evolving cybersecurity threats. For example, organizations that rely on multiple cloud service providers may have little or no control over the movement of their data through different data centres around the world. Similarly, it is not always clear whether the data custodian or the third-party service provider is accountable to protect the data, or which sets of data protection laws apply*".

GDPR will have a significant impact on the Cloud Computing industry, especially on Cloud Service Providers who will have to provide a significantly higher level of transparency about the elements of their internal sub-structures that have legal significance (e.g. vis-à-vis Third-Party subcontractors).

As such, the consequences of the introduction of GDPR [i.5] on the architectures of IoT systems are not yet fully understood. At this stage, it is not clear yet whether or not virtualization (and the use of microservices in particular) will improve the way privacy is currently handled. An example of such questions is multi-tenancy: on the one hand, virtualization can help by providing different IP addresses to access the data of different tenants (thus better protecting one tenant's data against - potentially malevolent - access to data from other tenants) whereas, on the other hand, multi-tenancy can be provided without separating the databases of two different tenants. Overall, the answer to the question will depend on the architecture choices and, even more importantly, on the implementation. Altogether, though the impact of GDPR is expected to be significant on IoT virtualization, it is probably too early to assess it, specially via a PoC.

5.5 The use case selected as a proof-of-concept

A Proof-of-concept (PoC) of IoT Virtualization is developed in ETSI TR 103 529 [i.2]. This PoC is an implementation of the "Horizontal Up and Down Auto-Scaling" Use Case described above.

The main reason for the choice of this UC is that it demonstrates the feasibility of IoT Virtualization on a "real-life" Use Case applicable to a large number of "verticals". Auto-Scaling has been seen, in the examples above, as a very critical feature in virtualized IoT: the PoC is also a way to validate its applicability to IoT systems. In addition, it also makes use of a great number of the Open Source Software components that are described in ETSI TR 103 528 [i.1].

It can be noted that the use of Auto-Scaling may not always be necessary in dealing with burst situations. However, indications to developers on the concrete use of this technique in a real implementation is useful, and it is an additional rational for its selection in the Proof-of-Concept (see ETSI TR 103 529 [i.2]).

Beyond the validation of the basic concepts that are implemented in the PoC (e.g. microservices based architecture, layered architectural model), it is important to consider that - in a "real-life" implementation context - the auto-scaling decision mechanism itself may be a differentiating factor, and therefore the possibility to implement Artificial Intelligence-based "auto-scaling". To this extent, though the "auto-scaling" features in the PoC are implemented with a "traditional" algorithmic approach where decisions are calculated on the basis of (more or less) hard-coded rules, the description of the PoC architecture makes room for the possible usage of:

- Monitoring functions that may allow human to use additional rules; or
- Artificial Intelligence systems (e.g. Q-learning) with more refined sets of rules.

6 Cloud Computing features for IoT Virtualization

6.1 Introduction

There are many requirements to be filled to get a full-fledge IoT system. The IoT community has developed a very large number of architectures, platforms, solutions and standards to deal with these requirements. The current clause is not going to address them extensively: it will rather focus on specific IoT requirements that can be effectively addressed by Virtualization and on specific features of Cloud Computing that can benefit to IoT systems.

Some of the requirements identified are regarding the functionalities that the Virtualization of IoT is expected to bring. In addition, a number of non-functional requirements (e.g. high availability) are described in details since they may be those for which Virtualization may bring the most effective solutions.

The Virtualization of IoT will also make use of **microservices**, a key feature of Cloud Computing. Microservices have been massively used in Cloud Computing in support of the implementation of open Cloud architectures. This clause will introduce them as an element for the definition of virtualized IoT systems.

6.2 Functional requirements

6.2.1 Introduction

The functional requirements for IoT systems are extremely varied and most of them will apply to IoT Virtualization unchanged. The purpose of this clause is to address only a few of the functional requirements that are specific to IoT Virtualization.

6.2.2 Multi-tenancy

6.2.2.1 Definition

Multi-tenancy is an architecture principle that allows a single instance of a software application to serve multiple customers. Within a multitenant architecture, a software application is designed to provide every customer with a dedicated portion of the software instance including data, configuration, user management, etc. All multi-tenant architectures work on the same principle: all customers will benefit from a given solution through a common infrastructure. When all of customer data is handled using the same software resources, the architecture design should be extremely rigorous and prevent from data leakage: client A will have no access to the data of the client B (and vice-versa). This is essential requirement of multi-tenancy: the data models should be designed in order to be filterable by a client identifier.

A multitenant application architecture helps optimize the use of hardware, software, and human capital. Its value is related to resource usage optimization and therefore to cost reduction. An additional advantage of multitenant applications is in software lifecycle management: the software upgrade of a given software instance can be done for any customer of the tenant instead of being applied to all software for all customers.

6.2.2.2 Comparison with multi-instance architectures

The choice of multi tenancy vs. multi instance architecture depends on a number of criteria such as: cost, customer expectations, extensibility needs, security challenges, etc. With the provision of multi-tenancy, configuration and management of data is more complex since it has to apply a specific method to prevent the leakage of data (using a single database) between the tenants. A Multitenant application should be put in place carefully to avoid major problems. For instance, it is very important to:

- separate the persistence of data from each tenant and its users;
- separate the configuration of each tenant;
- do not let the data pass between the tenants.

6.2.3 Massive Data processing

In IoT systems, high-volume and high-velocity data is produced, analysed, and used to trigger action. There are two ways to process data: streaming data processing and batch data processing.

Under the streaming model, the processing is usually done in real time. By building data streams, one can feed data into analytics tools as soon as it is generated and get near-instant analytics results. Complex event processing (CEP) is used for streaming data processing with the goal to identify significant events and respond fast.

Under the batch processing model, a set of data is collected over time, then fed into an analytics system. Batch processing is most often used when dealing with extremely large amounts of data, and/or when data sources are legacy systems that are not capable of delivering data in streams. Batch processing is adequate in situations where non-real-time analytics results is needed, and when it is more important to process large volumes of information than to get fast analytics results.

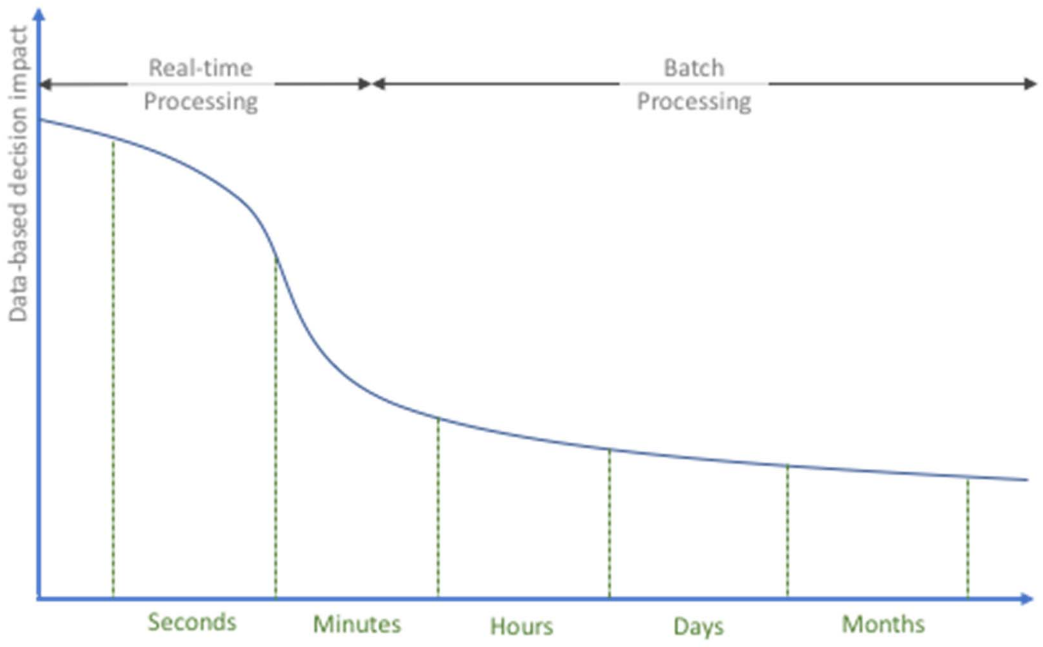

Figure 2: Batch and Streaming data processing

The management of huge amount of data is one of the challenges of Big Data, and the necessity to address simultaneously the "4V" properties: Volume, Velocity, Variety and Veracity. IoT is one of the many domains that is making use of Big Data is solutions, in particular when it comes to three first "V"s. The question of Veracity is key in IoT and some solutions need to be provided to address specific issues of Data Quality. This point is addressed in Annex A where some potential solutions are outlined regarding Fault Detection and Isolation.

6.3 Non-functional requirements

6.3.1 High-throughput

High throughput is in general associated with the use of many and parallel computing capabilities to accomplish a computational task. An efficient use of all available computing resources is the key to achieving high throughput. However, the quest for high throughput is not concerned about the number of operations per second, but rather by the number of operations over a longer period of time, typically days or months. In essence, high throughput is more interested in how many jobs can be completed over a long period of time instead of how fast.

Some IoT applications require processing large data-sets and need to use cloud computing capabilities to be able to process these data sets with a high throughput. For instance, in Industrial IoT, time series are generated very frequently by a substantially large number of devices and processed using high-throughput capabilities. In other scenarios, gateways generate a large quantity of logs (not IoT data per se) which need to be processed at high throughput for the purpose of e.g. preventive maintenance.

There is no single technique to achieve high throughput. Messaging (as described in clause 6.4.2) combined with parallelism provides an important enabler to high throughput.

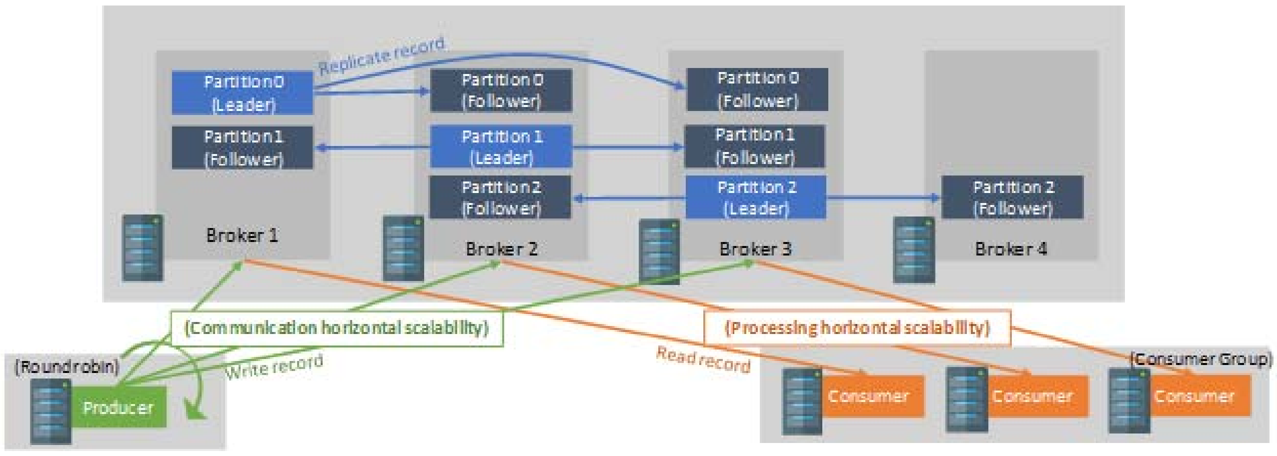

Figure 3: Achieving high throughput processing of data sets

Figure 3 depicts a producer of data, that needs processing which sends data sets using round robin strategy to multiple broker instances. The processing servers are consumers of the brokers, they subscribe to a specific topic they need to process. This figure shows how both processing and communication can be scaled horizontally in order to cope with high data throughput.

6.3.2 High-availability

High-availability embodies the idea of access to services, tools and data anywhere and at any time.

High-availability combines software with industry-standard hardware to minimize downtime by quickly restoring essential services when a system, component, or application fails. While not instantaneously, services are restored rapidly, often (and preferably) in less than a minute.

For highly-available applications, a service needs to be resilient to failures and able to restart on another machine. The problems of failures (and the challenge for resiliency support) are observed during various scenarios, such as:

- Failures occurring when the machine where the service is running fails.
- Failures occurrence due to a service-internal hard problem.
- Failures occurring during an application upgrade. The running service should determine whether it can continue to move forward to the newer version or to roll back to a previous stable version to maintain a consistent state. In this case, one has to consider the following points: are enough machines available to keep moving service upgrade, and how to recover previous versions of the service.

The service should be designed so that the process can be restarted at any time with no data loss.

The easiest way to make a service resilient is to have multiple hosts running the service instance and managed by a load balancer. The service consumers have no knowledge of whether there is one or multiple service instances. The load balancer is capable of:

- Distributing requests between service instances based on algorithms (e.g. round-robin).
- Shutting down remote service instances when failures are detected.
- Adding service instances.

Running multiple service instances gives the capability to handle higher load and to avoid single point of failure.

6.3.3 Low latency

6.3.3.1 Requirements

Low latency is essential to time-critical applications such as autonomous driving or industrial automation. This feature should be supported by system architecture and design. However, there is no single mechanism to achieve low latency - programmers and system architects need a toolbox so that they can mix-and-match tools to meet the different requirements and traffic patterns associated with applications.

Examples of such tools include brokers capable of routing IoT application messages in (near) real-time. Another example is MapReduce which may make use of massive in-memory databases capable of meeting low-latency requirements.

The use of Edge Computing allows to reduce network latency by moving computing resources closer to the field domain where an action takes place.

6.3.3.2 MapReduce

MapReduce is a computer development architecture, invented by Google, in which parallel processing of large data sets is performed over distributed computing resources with an objective to provide low latency to answer requests.

MapReduce allows to manipulate large amounts of data by distributing them in a cluster of machines to be processed. The terms "map" and "reduce", and the underlying concepts, are borrowed from the functional programming languages used for their construction (map and reduction of functional programming and table programming languages).

As the name MapReduce suggests, the "reduce" phase takes place after the completion of the "mapper" phase:

- The first step is the mapping, where a block of data is read and processed to produce key-value pairs as intermediate outputs. The node analyses a problem, splits it into sub-problems, and delegates them to other (children) nodes (which can do the same recursively). The sub-problems are then processed by the different nodes using the Map function.
- The output of the Mapper or map job (key-value pairs) is an input to the Reducer.
- The Reducer receives the key-value pair from multiple map jobs. The children nodes return their results to the parent node that solicited them. This calculates a partial result using the Reduce function which associates all the corresponding values to the same key.
- Then, the Reducer aggregates those intermediate data tuples (intermediate key-value pair) into a smaller set of tuples or key-value pairs which is the final output.

An important concept of MapReduce is that, instead of transferring data to the processing resource, the processing is moved close to the data and only results are returned back.

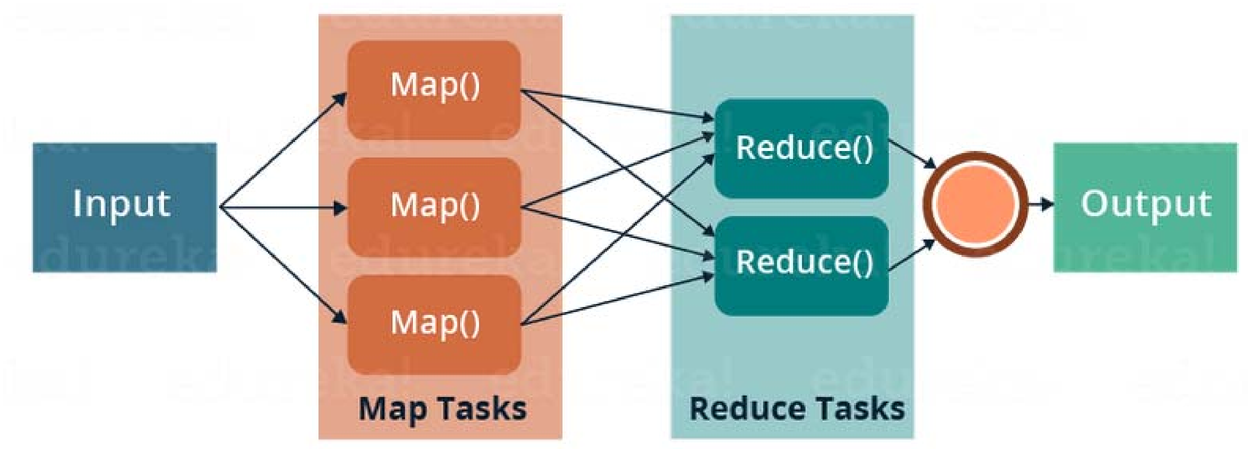

Figure 4: The MapReduce Concept

Several frameworks have emerged to implement the MapReduce concept. The most famous one is Hadoop, which is developed by the Apache Software Foundation.

6.3.3.3 In Memory Databases

In-Memory Databases use (in-)memory for data storage. This is in contrast to databases that use hard disks to store data. Memory databases are faster than disk-based ones because disk access is substantially faster, and internal optimization algorithms are simpler to execute and require fewer CPU instructions.

In-Memory databases are gaining traction as memory costs are reducing. Application that increasingly require in-memory databases are in particular those for which low latency is required. Examples include real-time data analytics.

6.3.3.4 Edge Computing

IoT Virtualization is expected to benefit from Cloud Computing technologies and solutions. However, there are some limitations of Cloud Computing that hinder the fully effective implementation of some of the promising IoT virtualization use cases (e.g. the "Horizontal up and down auto-scaling" use case discussed in clause 6.2).

Recently, Edge Computing has started to emerge as a new approach that may complement Cloud Computing in some cases where the availability of computing resources closer to the devices is required. The support of Edge Computing can be seen as a way to support requirements related to low latency (faster computing), massive handling of data (reduced amount of data transfer toward the core) or fault tolerance (replication of computing resources at the edge).

With Edge Computing, Cloud Computing is going through a fundamental shift in which the traditional model of accessing highly centralized resources is replaced by a distributed, decentralized architecture. This new computing paradigm brings the core building blocks of cloud (computing, storage and networking) closer to the consumers (devices).

Though Edge Computing is still in the maturing phase, the support of Edge Computing will be a key requirement for a very large range of IoT Virtualization use case and for the associated platforms. There are hundreds of use cases where reaction time is the key value of the IoT system. The main goal of Edge Computing is to minimize latency by bringing the public cloud capabilities to the edge, in contrast with "traditional" Cloud Computing where constantly sending the data back to a centralized cloud increases latency.

The implementation of Edge Computing can be achieved by two approaches:

- Device Edge: custom software stack emulating the cloud services running on existing hardware.
- Cloud Edge: the public cloud seamlessly extended to multiple point-of-presence (PoP) locations.

Device Edge: Customers install and run Edge Computing software in existing environments. The hardware can be dedicated or shared with other services. In many scenarios, the edge stack is run on low-powered devices running low consumption processors (e.g. ARM). All the sensors talk to the local edge device, which manages the connectivity with the cloud.

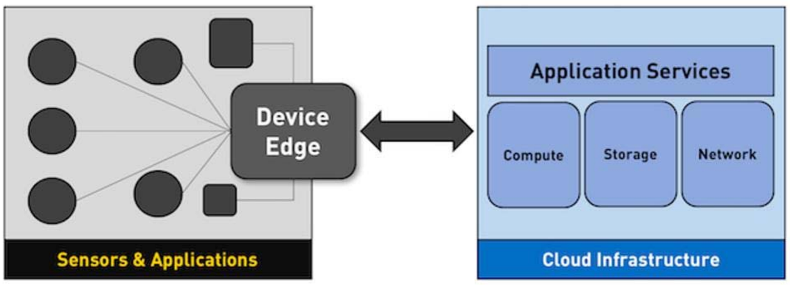

Figure 5: Device Edge

Figure 5 illustrates a device edge architecture. A specialized device is acting as the local IoT Gateway that mimics the public cloud capabilities.

Microsoft Azure IoT Edge is an example of device edge software. It attempts to bring device registry, device twins, device communication, local storage and synchronization capabilities.

Cloud Edge: It is an extension of the public cloud in a highly distributed form. Unlike device edge, cloud edge is owned and maintained by the public Cloud Service Provider. The cloud edge becomes a micro-zone, a logical extension to the existing hierarchy of regions and zones. Micro-zones may extend public clouds to thousands of new locations, enabling developers to keep applications as close as possible to consumers. With just a single hop to the data centre, the latency involved in accessing traditional cloud platforms is dramatically reduced.

The concept of Cloud Edge is illustrated in Figure 6.

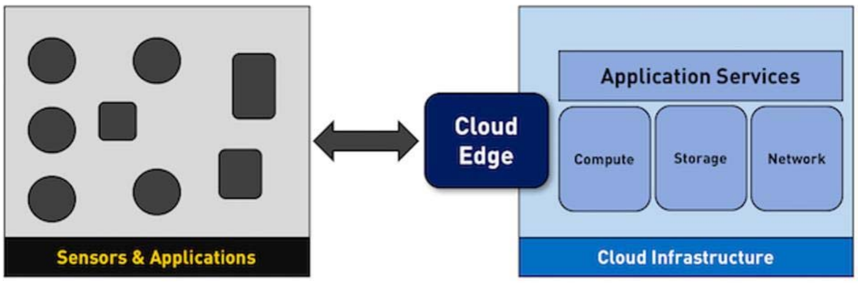

Figure 6: Cloud Edge

6.3.4 Security

Security is a key enabler (or, when not properly taken into account, a roadblock) of Cloud Computing. As stated in [i.6], "*Cloud computing systems can address security requirements such as authentication, authorization, availability, confidentiality, non-repudiation, identity management, integrity, audit, security monitoring, incident response, and security policy management*".

Security is a global requirement for every IoT system that impacts every layer of the system architecture (making it a sort of "vertical" sub-system as it will be depicted below in clause 7) and every component of the system design, in particular when security by design techniques are used, which is expected to be the case for emerging IoT systems that are developed concurrently with the maturation of security-by-design.

IoT virtualization does not reduce the importance of security requirements, nor the complexity of the implementation of security across the components of the virtualized IoT system. In particular, Information Security (which broadly covers the protection of the confidentiality, integrity and availability of information assets) is a major challenge where Identity and Access Management (IAM) is a key building block in any solution. IAM involves the management of individuals and ICT resources in an organization and the definition and enforcement of policies for the authentication and resource authorization. It involves a number of techniques amongst which the support for access control (including role-based access control) is required for any effective and trustable implementation.

6.4 Features in support of virtualized IoT implementations

6.4.1 Microservices

6.4.1.1 Definition

In order to get the most of the integration of IoT and Cloud Computing, the use of microservices should be considered. Microservices are an architectural approach to developing applications as a set of small services, where each service is running as a separate process, communicating through simple mechanisms.

The advantages of a microservice-based architecture stem from its main feature, the decomposition of a service or an application into many independent units (i.e. smaller component). As a result, one can **develop, deploy, upgrade** and **scale** every microservice independently of the others. This enables to use an optimal amount of resources and to make microservice-based architectures a natural fit for achieving both scalability and elasticity. Developing microservices separately enables the use of different technologies for each microservice. Possible example of microservices in the context of IoT include: communication with IoT devices, protocol adaptation, data processing, communication with databases and visualization.

6.4.1.2 Comparison to monolithic architectures

The advantages of microservices architecture are best identified when compared to the traditional "monolithic" architectures. A monolithic application has all of its components packed together. For instance, monolithic IoT applications may include the whole computing logic for communication with IoT devices; processing of devices data; potential communication with databases and visualization, packaged in a single logical executable. Monolithic IoT applications may become so large that managing them - and moreover updating them - can become extremely complex.

While errors in monolithic IoT applications may be really expensive as they cause the whole application to crash, errors in microservices applications cause only the corresponding microservice to collapse. This means that the microservices-based application is still running, and only the specific functionality implemented by the microservice is unavailable. The importance of this behaviour is even more apparent in IoT applications. For instance, if a microservice which communicates with a certain group of sensors crashes, that will not affect or stop the processing of the data provided by microservices which communicate with other sensors. The other components of the application will still be up and running.

6.4.1.3 Impact on IoT solutions

Microservices architecture have emerged in the recent years and consequently are not fully generalized with universally adopted development principles. However, most of the microservices applications share the same characteristics. In general, the microservices architecture is adaptable to the requirements of IoT applications.

When developing applications, it is a good practice to break down the application into several components. The microservices architecture tends to componentize a project into services, where each service is running in its own separate process. Thereby, each microservice can be deployed and scaled independently. The componentization into microservices allows to address the problem of the vast heterogeneity of IoT devices. It is possible to have distinctive microservices for devices that communicate using different protocols. These microservices might act as a proxy. The problem of adding new devices which communicate using a non-supported protocol is usually resolved by adding a microservice acting as a proxy between protocols.

Another advantage of microservices is regarding the programming languages. Monolithic applications are usually written in one programming language, and the use of different languages is possible but often brings additional problems. On the other hand, microservices enable a decentralized approach that encourages developers to write different microservices using different technologies. This characteristic is especially helpful to IoT applications. For instance, it enables the use of technology for communication with devices different from the one used for data processing, or the one used for visualization as well. Microservices enable all of the technologies to be integrated without having to worry about compatibility issues.

6.4.1.4 Scaling microservices

The microservices architecture brings more freedom for deployment and management of applications in cloud infrastructures. In order for an application to be elastic, one will ensure that every microservice uses an optimal amount of resources. For scaling of microservices, both Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) can be used. In particular, PaaS environments are very suitable for microservice-based applications: they offer a platform, which is responsible for low-level operations like management of virtual machines, application deployment, load balancing, etc. Altogether, PaaS supports an easier management of applications by enabling the designers and developers to focus on the IoT features and not on the low-level part of the application. On the contrary, PaaS is not supporting monolithic applications where developers also have to take care of the configuration and management of the application.

6.4.1.5 Providing persistency for microservices

Microservices persistence means persistence database tables and event streams, in addition to traditional files. In microservice-based architectures, the approach to persistence is different:

- The operational state is not stored in files but in a database table, typically one that is dedicated to the microservice (to accomplish the isolation of duties). The complete state of an application at a given time is distributed across its stores: it is the collective state of what has been read from the input stream, what has been stored in the operational database, and what has been sent to the output stream.
- The units of work are no longer transactions but events, typically received via a publish/subscribe channel by upstream microservices.
- The application history (logs, metrics) is still produced and has to be persisted somewhere for monitoring. Logs are usually saved to a file, and metrics to a stream.

Using a database per service has the following benefits:

- it helps ensure that the services are loosely coupled. Changes to one service's database does not impact any other services;
- each service can use the type of database that is best suited to its needs. For example, a service that does text searches could use ElasticSearch. A service that manipulates a social graph could use Neo4j;

and the following drawbacks:

- Implementing business transactions that span multiple services is not straightforward.
- Implementing queries that join data that is now in multiple databases is challenging.

In the following paragraphs, some current approaches to sharing data in microservices architecture is presented together with their advantages and disadvantages: Shared database; Dedicated microservice; and Event/subscription.

Shared database

To avoid concurrency and inconsistency problem of shared data across databases, there are essentially two approaches: transactions and eventual consistency.

Transactions are mechanisms that allow database clients to make sure a series of changes either happen or not. In other words, transactions allow us to guarantee consistency. In the world of distributed systems, there are distributed transactions. There are different ways of implementing distributed transactions, but in general, there is a transaction manager that will be notified when a client wants to start a transaction. The downside to this approach is that scaling is usually harder. Transactions are useful in the context of small or quick changes.

Eventual consistency deals with the problem of distributed data by allowing inconsistencies for a time. In other words, systems that rely on eventual consistency assume the data will be in an inconsistent state at some point and handle the situation by postponing the operation, using the data as-is, or ignoring certain pieces of data. Eventual consistency systems are easier to reason about but not all data models or operations fit its semantics. Eventual consistency is useful in the context of big volumes of data.

Dedicated microservice

In this approach, a new microservice is developed to manage a shared database rather than allowing the microservices to access the database directly. This microservice manages all access to the shared data by the other services. By having a common entry point it is easier to reason about changes in various places. For small volumes of data, this can be a good option as long as the new microservice is the only one managing the data.

Event/Subscription model

In this approach, rather than allowing each service to fetch directly the data, services that make changes to data or that generate data allow other services to subscribe to events. When these events take place, the services that have subscribed receive the notification and make use of the information contained in the event. This means that, at no point, a microservice may be reading data that has been modified by other microservices. The simplicity of this approach makes it a powerful solution to many use cases. However, there are downsides: a set of events will be integrated into the data generating microservice and losing events becomes a possibility.

6.4.1.6 Security for microservices

The introduction of microservices in virtualized IoT architectures has also an impact on their security architecture. When an architecture is implemented as a number of interacting microservices, the protection of the microservices themselves and of the inter-services communication mechanisms is an essential requirement.

One effective way to secure a microservice is to secure its API. A secure API to a microservice can guarantee the confidentiality of the information it processes, by making it visible only to the users, applications and servers that are authorized to consume it. It should be able to guarantee the integrity of the information it receives from the clients and servers it collaborates with, so that it will only process such information if it knows that it has not been modified by a third party. The ability to identify the calling systems and their end-users is a prerequisite to guarantee those security qualities.

6.4.2 Inter-Process Communication (IPC) in microservices architecture

6.4.2.1 Communication Mechanisms

When selecting an IPC mechanism for a service, it is useful to think first about how services interact. There are a variety of interaction styles which can be categorized along two dimensions. The first dimension is whether the interaction is **one-to-one** where each client request is processed by exactly one service instance or **one-to-many** where each request is processed by several service instances. The second dimension is whether the interaction is **synchronous** where the client expects a timely response from the service, or **asynchronous** where the client does not block while waiting for a response, that is not necessarily sent immediately.

6.4.2.2 Synchronous IPC communications: RESTful communication

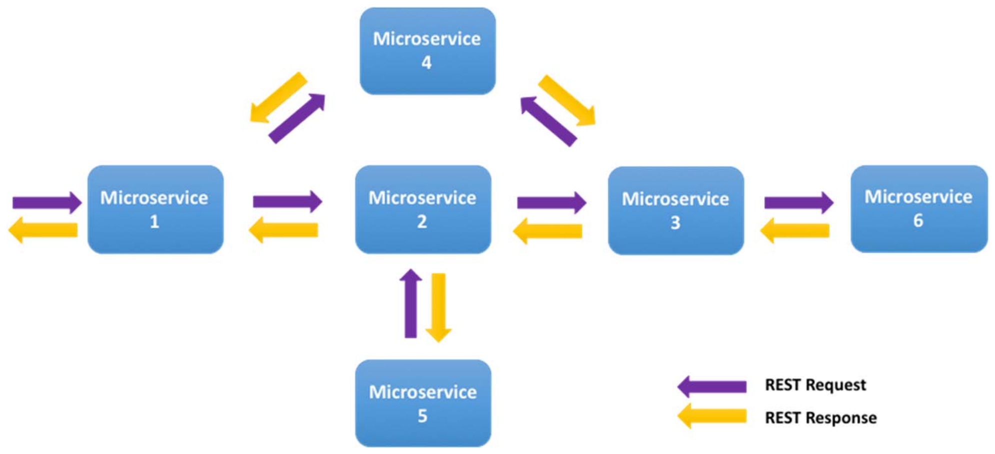

Figure 7: RESTful IPC

In synchronous communications, a call is made to a remote service which blocks until the operation is achieved. With synchronous communication, one knows when requests have been fulfilled successfully or not. The synchronous communication mode enables request/ response collaboration style, where a client initiates a request and waits for the response.

A popular architectural style for request/response communication is REST. RESTful microservices, as shown in Figure 7, communicate directly and synchronously with each other, without the need for any additional infrastructure. RESTful communications are based on HTTP verbs like GET, POST and PUT. An important concept for RESTful collaboration is resources. A resource can be thought of as a thing that the service knows about, for example an order. A service manages the representation of this order on requests (creating different representation, updating, deleting, etc.). It should be noted that REST is independent of the underlying protocol and can be implemented using several ones such as HTTP, CoAP, etc.

6.4.2.3 Asynchronous IPC communications: Messaging

Asynchronous communication is suitable for long-running jobs, where constantly maintaining the connection between the service and the client is not feasible. It is also useful when low latency is required. Asynchronous communications invert the approach: a client does not request for something to be done, but instead says that something happened and waits for other services to react.

In an asynchronous messaging-based system (as depicted in Figure 8), both input and output from services are defined as events. Each service subscribes to the events that it is interested in consuming, and then receives these events reliably when the events are placed on the queue by other services.

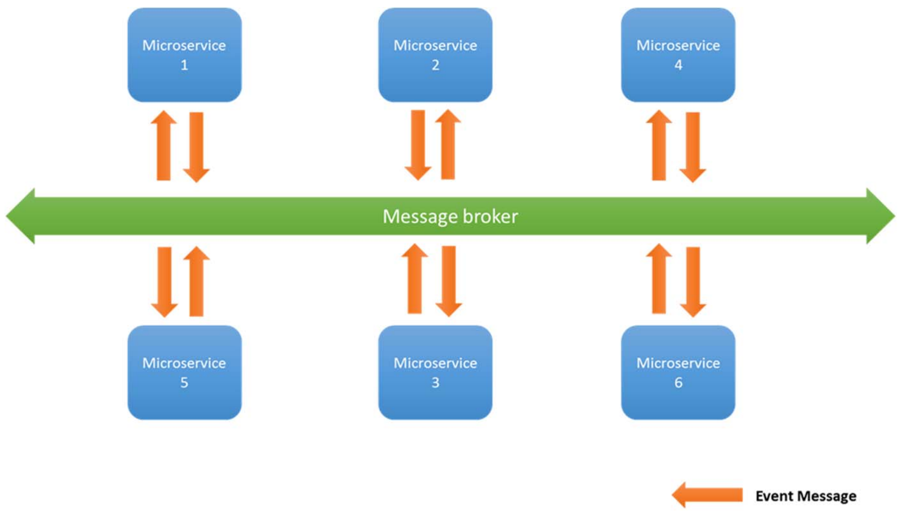

Figure 8: Asynchronous Messaging IPC

This publish/subscribe system is implemented by a message bus (in a message broker). The message bus can be designed as a middleware, with the API needed to subscribe or unsubscribe to events and to publish events. Different message bus implementations are available and each implementation will determine which protocol to use for event-driven, message-based communications. For instance, the AMQP protocol has proven that it can achieve reliable queued communication.

6.4.2.4 Hybrid IPC communications

With hybrid IPC style, each service typically uses a combination of synchronous and asynchronous interaction styles (as shown in Figure 9). For some services, a single IPC mechanism is sufficient. Other services might need to use a combination of IPC mechanisms.

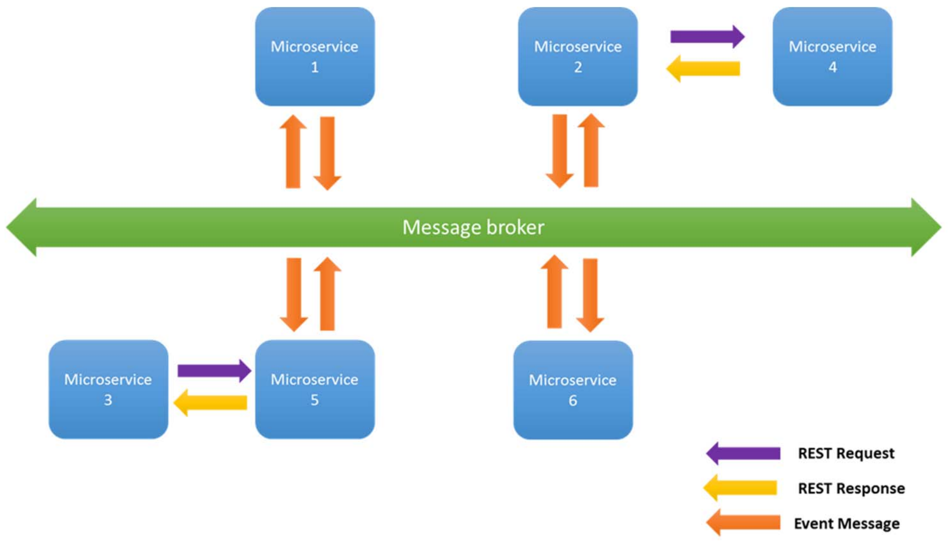

Figure 9: Hybrid IPC communications

7 Implications of IoT virtualization

7.1 Introduction

This clause is addressing the consequences of the introduction of virtualization in IoT systems on their architecture and design. Mainly two major impacts have to be noticed:

- On the one hand, IoT system architectures based on microservices should be able to support the split of monolithic services into a (potentially significant) number of microservices that are able to evolve relatively independently from each other and to communicate in a safe, secure and efficient manner. To this extent, a microservices architecture is a key element. One such architecture is described below.

- On the other hand, the possibility to split an architecture into microservices that can be implemented by separate components (and in particular by Open Source Software components) does not mean that the resulting architecture be largely unstructured. Actually, the possibility to define architectural layers and group them in a High-Level Architecture (HLA) for IoT virtualization may allow for the most effective selection and combination of such components.

7.2 Microservices for IoT Virtualization

7.2.1 Microservices Architecture

Figure 10 describes a Microservices Architecture for IoT systems. Each microservice handles an IoT service logic.

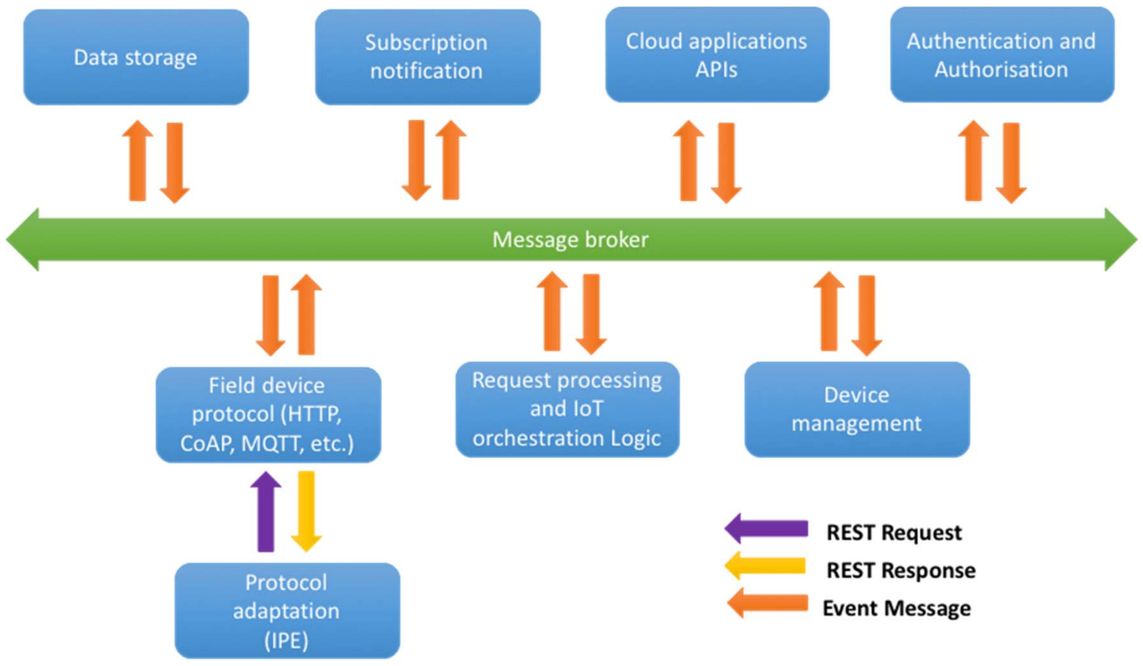
Data storage
Subscription notification
Cloud applications APIs
Authentication and Authorisation
Message broker
Field device protocol (HTTP, CoAP, MQTT, etc.)
Request processing and IoT orchestration Logic
Device management
Protocol adaptation (IPE)
REST Request
REST Response
Event Message

Figure 10: Microservices Architecture for IoT Virtualization

The following microservices are considered:

- Protocol adaptation

  This entity plays the role of an Interworking Proxy Entity which enable seamless communication for legacy devices such as Zigbee, Phidgets, and many other technologies with by performing mapping/converting operations.

- Field device protocol

  This service provides a point of contact for communication between users and the system using several applications protocols. It makes use of existing network connectivity and manages all security aspects for secure session establishment and teardown. For each protocol, a field device protocol microservice is provided.

- Request processing and IoT orchestration logic

  This service provides a protocol independent service for handling users requests. It is also responsible of the coordination between the microservices.

- Device Management

  This service provides functions pertaining to device/gateway life cycle management, such as software and firmware upgrade and provides mechanisms for fault and performance management.

- Data Storage

  This service ensures the persistence of the system by storing information related to IoT applications and microservices states.

- Authentication and Authorization

  This service implements authentication, authorization, and key management functions to establish secure communication between cloud applications and the system.

- Subscription notification

  This service defines the set of procedures allowing an application to subscribe and be notified when specific subscription criteria are matched.

- Cloud application APIs

  This service implements bootstrapping, authentication, authorization, and key management functions to establish secure communication between cloud applications and the system.

As shown in Figure 10, a hybrid Inter-Process Communication (IPC) architecture is deployed: a message broker is in charge of handling communication between microservices: high decoupling and RESTful communication between microservices is supported, in order to minimize microservices communication processing when needed.

7.2.2 The Microservices Architecture in practice: an example

Figure 11 provides an example of message flow explaining how microservices could be used for processing incoming requests from field devices or gateways. For simplicity, the message broker is not shown in this figure, but it is assumed all flows apart from 1 and 6 would take place using the broker.

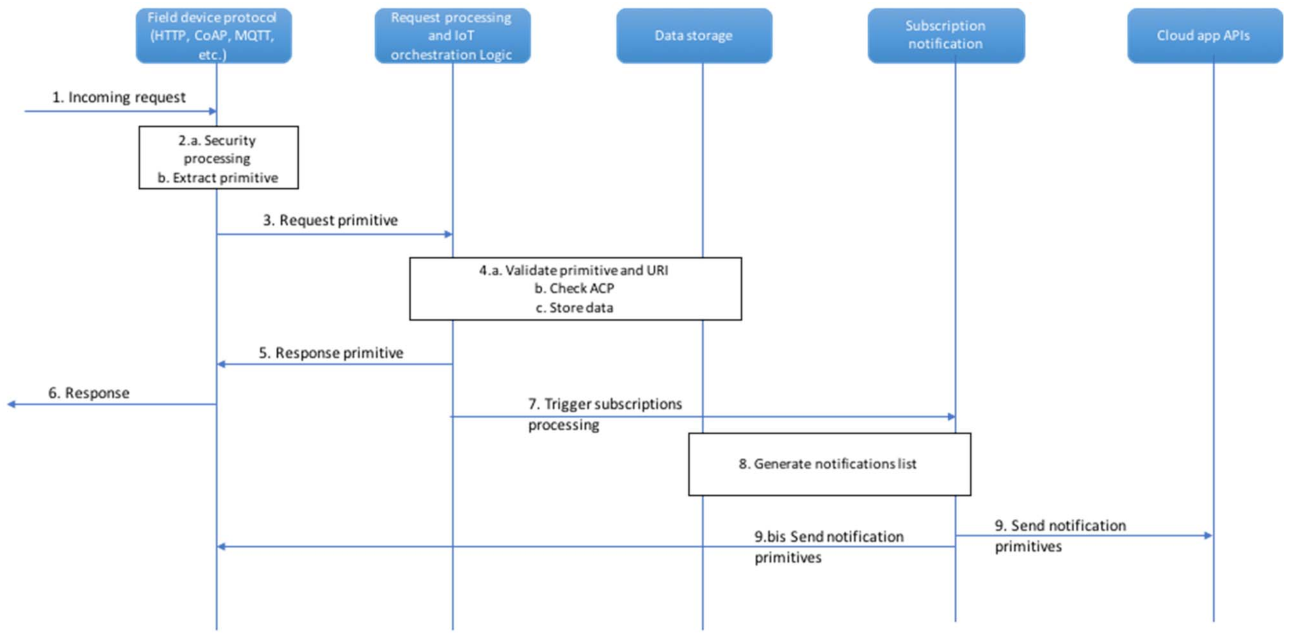
Field device protocol (HTTP, CoAP, MQTT, etc.)
Request processing and IoT orchestration Logic
Data storage
Subscription notification
Cloud app APIs
1. Incoming request
2.a. Security processing
b. Extract primitive
3. Request primitive
4.a. Validate primitive and URI
b. Check ACP
c. Store data
5. Response primitive
6. Response
7. Trigger subscriptions processing
8. Generate notifications list
9.bis Send notification primitives
9. Send notification primitives

Figure 11: Message Flow Example

Step 1: An incoming request is received by the **Field device protocol** microservice. This request would typically be using a binding protocol such as HTTP, CoAP or MQTT.

Step 2: A first security processing takes place, that includes message decryption, authentication headers and integrity validation. Then a Request primitive is extracted and syntactically validated: this Request primitive is now protocol independent.

Step 3: The Request primitive is sent to the **Request processing and IoT orchestration logic** microservice.

Step 4: This microservice would need several interactions with **Data storage**. First the primitive is validated beyond basic syntactic processing. The URI validity is checked using the stored data. Then the Access Control Policies are checked to validate the request is valid from an authorization perspective. Finally (and optionally), data pertaining to the Request primitive is stored using the Data storage microservice.

Step 5: A Response primitive is sent back to **Field device protocol** microservice.

Step 6: A Response is sent back to the originator in step 1. It will use the same protocol binding (HTTP, MQTT, CoAP, etc.) as in step 1.

Step 7: Based on the existence of subscriptions impacted by the Request primitive, subscriptions and notifications processing is triggered.

Step 8: The **Subscription notification** microservice interacts with the **Data storage** microservice to generate notifications primitives to be sent to subscribers as shown in Step 9 and Step 9.bis.

Step 9 and Step 9.bis:
The notification primitives are sent to the subscriber entities in the field or cloud domain.

7.2.3 Relationship of the microservice service HLA to oneM2M

oneM2M defines a list of Common Service Functions (CSFs) as an "informative architectural construct which conceptually groups together a number of sub-functions" (see ETSI TS 118 101 [i.7]). Figure 12, extracted from this document, provides the list of these CSFs.

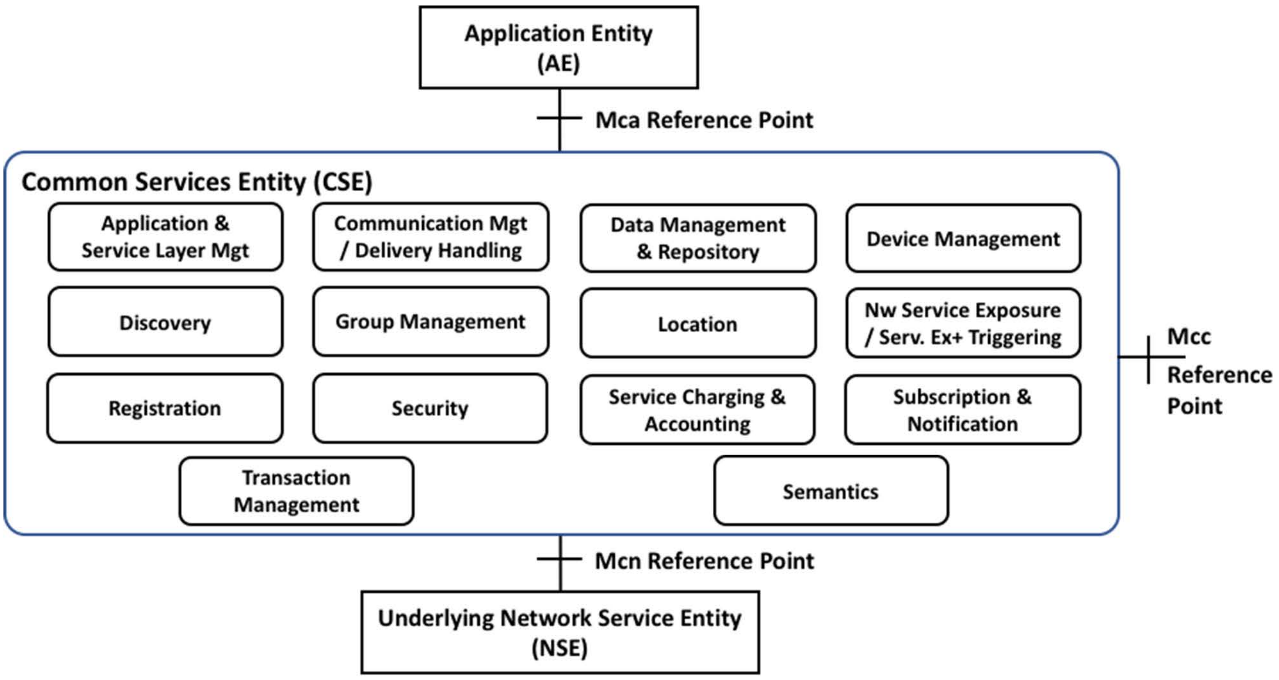
Application Entity (AE)
Mca Reference Point
Common Services Entity (CSE)
Application & Service Layer Mgt
Communication Mgt / Delivery Handling
Data Management & Repository
Device Management
Discovery
Group Management
Location
Nw Service Exposure / Serv. Ex+ Triggering
Registration
Security
Service Charging & Accounting
Subscription & Notification
Transaction Management
Semantics
Mcc Reference Point
Mcn Reference Point
Underlying Network Service Entity (NSE)

Figure 12: Common Services Functions defined by oneM2M

The CSF descriptions are provided for the purpose of understanding of the oneM2M Architecture functionalities and are informative. The CSFs contained inside the Common Services Entity (CSE) can interact with each other but ETSI TS 118 101 [i.7] does not specify how these interactions take place. CSFs are actually defined in order to provide guidelines for implementers. Only the interactions on the reference points between CSEs and between applications (AEs) and CSEs are mandated in oneM2M.

Additionally, CSFs have not been defined with a micro service architecture in mind. Indeed, the choice of dividing a CSE into microservices should always be left up to specific implementations, which means that the optimizations made for two different deployment scenarios may result in two different choices of grouping into microservices.

The example provided in clause 7.2.1 is rather generic and has not been provided with oneM2M in mind. Nevertheless, it is possible to reconcile the microservices architecture (presented in clause 7.2.1) and the oneM2M architecture, especially the functional architecture (presented in Figure 12). This is the purpose of Figure 13.

It should be noted that, even if a given oneM2M Service Entity (SE) can be mapped onto a single microservice, this is not meant to be the general case: for implementation efficiency considerations, a oneM2M SE will be mapped onto more than one microservice. This is the main expected benefit of introducing the Microservice Architecture.

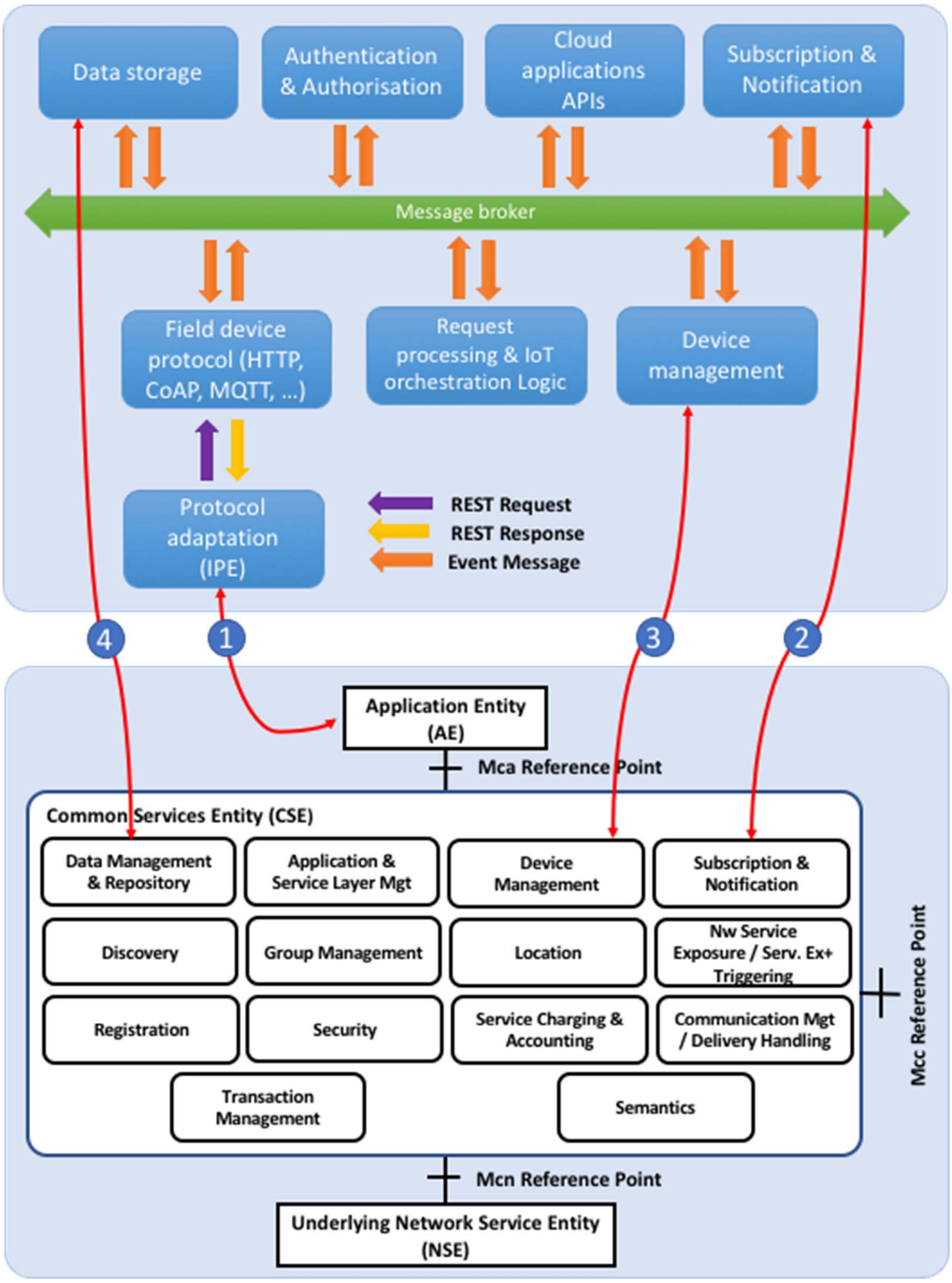
Data storage
Authentication & Authorisation
Cloud applications APIs
Subscription & Notification
Message broker
Field device protocol (HTTP, CoAP, MQTT, ...)
Request processing & IoT orchestration Logic
Device management
Protocol adaptation (IPE)
REST Request
REST Response
Event Message
4
1
3
2
Application Entity (AE)
Mca Reference Point
Common Services Entity (CSE)
Data Management & Repository
Application & Service Layer Mgt
Device Management
Subscription & Notification
Discovery
Group Management
Location
Nw Service Exposure / Serv. Ex+ Triggering
Registration
Security
Service Charging & Accounting
Communication Mgt / Delivery Handling
Transaction Management
Semantics
Mcc Reference Point
Mcn Reference Point
Underlying Network Service Entity (NSE)

Figure 13: Comparison between the microservices architecture and oneM2M CSF

The following mappings between oneM2M CSFs and the Microservice Architecture (developed in clause 7.2.1) can be made:

1) Protocol adaptation is implemented in oneM2M as an Application Entity. It is functionally equivalent to the Protocol Adaptation IPE of the microservice architecture.

2) Subscription management CSF is equivalent to the microservice in the architecture.

3) Device management CSF is equivalent to the microservice in the architecture.

4) Data management and repository CSF is equivalent to Data storage microservice.

As noted above, a given oneM2M CSF will be mapped onto several microservices. In some case, the mapping is less obvious. The Field device protocols microservice as well as the Cloud application API microservice have no equivalent in the oneM2M CSFs. Indeed, in some way, they are part of the reference points Mcc and Mca which are used for communication between oneM2M functional entities. This confirms that microservices as presented in clause 7.2.1 are closer to a real implementation, while oneM2M CSFs are providing guidelines.

7.3 One High-Level Architecture for IoT Virtualization

7.3.1 Functional Architecture for IoT Virtualization

Figure 14 introduces **one example** of a structuration of the functional architecture into layers (and sublayers) with an indication of the main functions that are expected to be provided in each of the layers and sublayers. In addition, two vertical functions are added related to cross-layer functionality: security and management.

The focus in the present document is on the functions. This architecture is also used in ETSI TR 103 528 [i.1] and the functions described in the layers and sublayers are used for the identification of potential Open Source Components that can support the implementation of the IoT microservices.

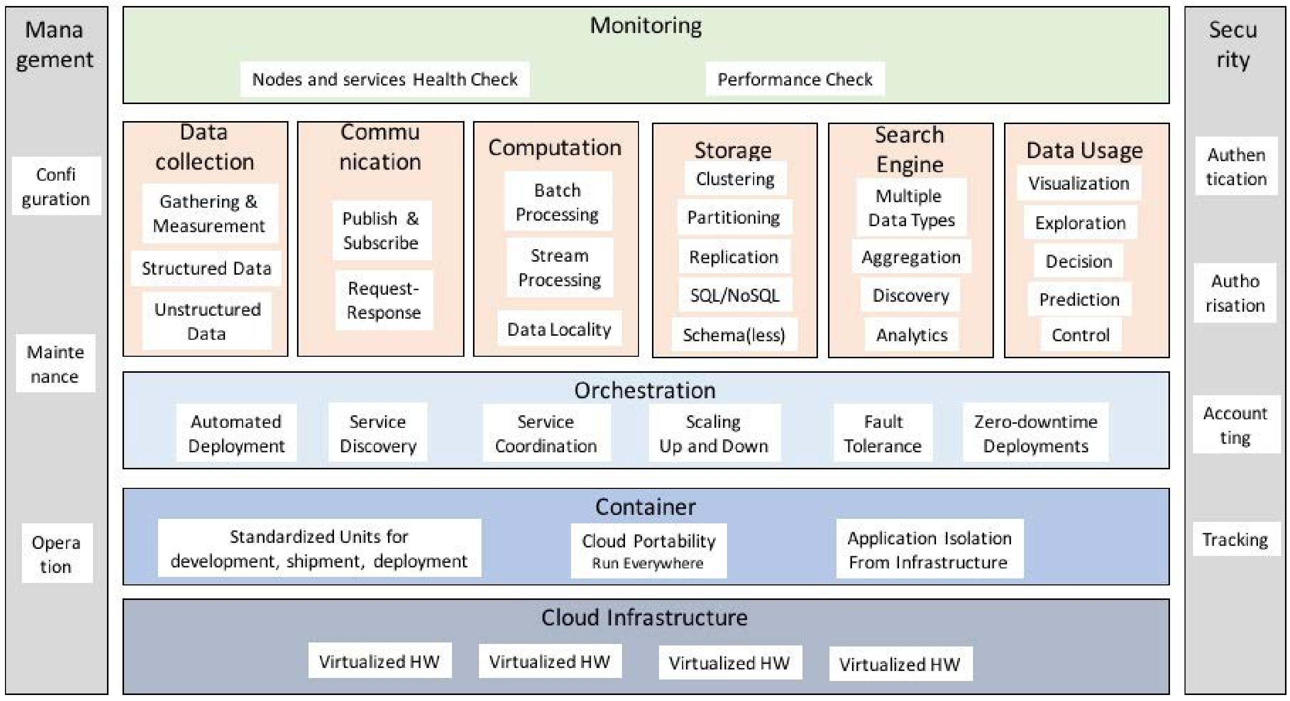

Figure 14: A High-Level Architecture for IoT Virtualization

The above architecture is one example (amongst other possible ones) that is in particular dealing with a structuration of the generic microservices that could be found in an IoT Layer.

7.3.2 HLA for IoT Virtualization and oneM2M HLA

The Microservice Architecture described in Figure 14 is meant to be generic in the sense that it can apply to a large number of systems in various business sectors. It provides a way to structure the provision of functionality with layers that ensure a certain degree of separation that can be supported by APIs and implemented via microservices. The layers described can be refined in order to address specific issues. For example, it could be possibly needed to have an "Edge layer" (not represented in Figure 14) sitting under the "Cloud Infrastructure" and meant to handle specific requirement (e.g. low latency) on virtualized resources.

The respective positioning of oneM2M Common Service Entities (CSE) and the microservices in the Microservice Architecture is shown in Figure 15. To better understand what is represented, the following observations can be made:

- There is a difference between the CSFs (that are specified via a standard) and the microservices that are one possible implementation of (a subset of) a CSF.
- All (as well as only a part of) the microservices described on Figure 14 can be included in a given CSE. The set of implemented microservices and their chosen implementations can (and probably will) be different from one CSF to another. Consequently, there is no standardized mapping of one CSF to microservices.
- Some alternatives for the choice of an implementation of microservice is addressed in ETSI TR 103 528 [i.1].

It has already been noticed (as described in Figure 13) that the mapping of microservices in the Microservice Architecture with oneM2M CSEs can be one-to-one. However, this is not the general case and, moreover, the use of microservices is, in principle, a way to map a given oneM2M CSE on more than one microservice, thus enabling the use of more fine-grain services, evolving separately and eventually developed with different technologies (and different developers).

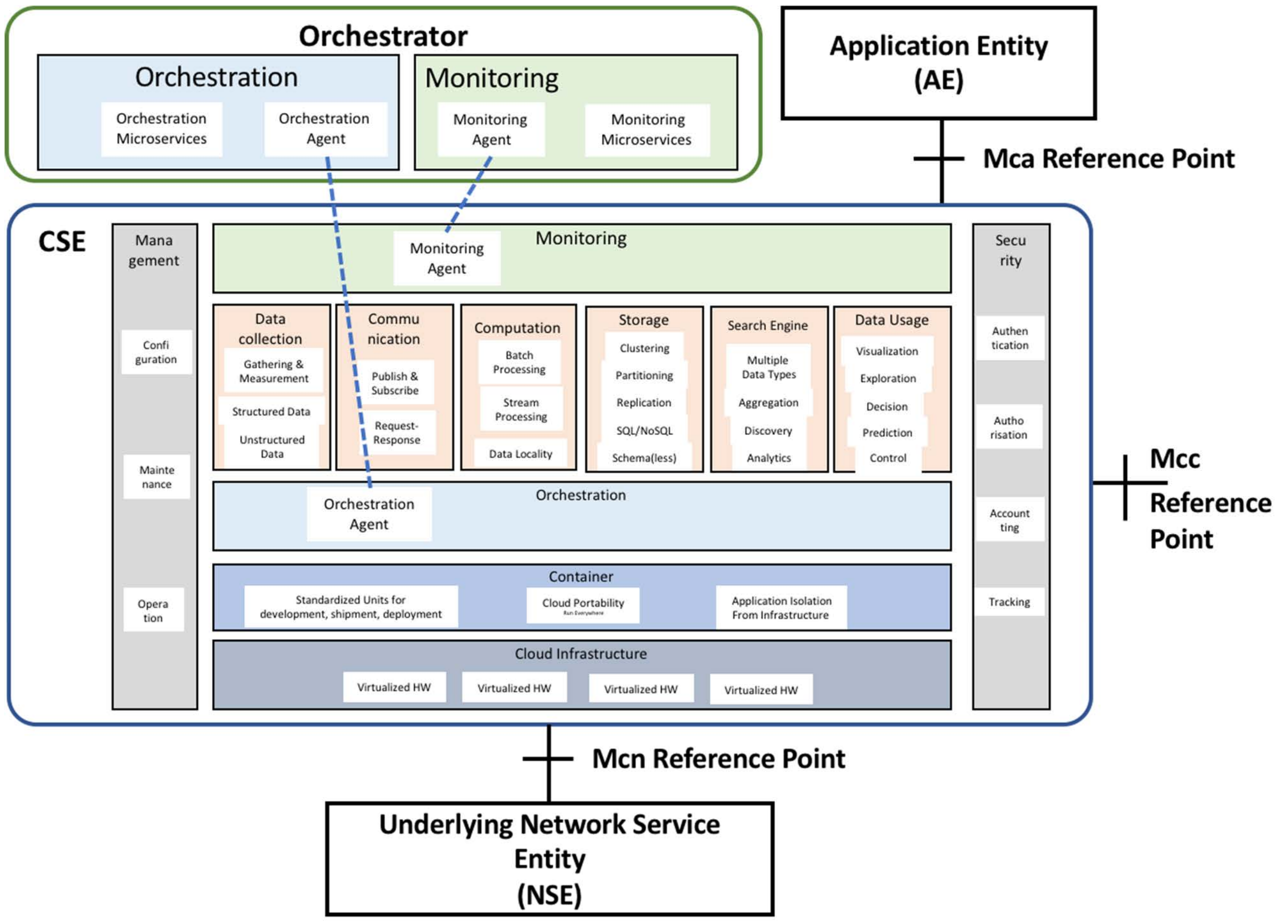

Figure 15: Mapping the Microservice Architecture and oneM2M Common Service Entities

An important point of notice is regarding the role of the orchestrator in the mapping of the microservices HLA. The following remarks can be made:

- No provision for an orchestrator is made in the oneM2M architecture specification.
- The global orchestrator is entirely outside of the CSE. It includes in particular:
  - In the Orchestration function: the microservices of the HLA Orchestration layer and an Orchestration Agent that interacts with an Orchestration Agent in the CSE Orchestration layer.
  - In the Monitoring function: the microservices of the HLA Monitoring layer and an Monitoring Agent that interacts with an Orchestration Agent in the CSE Monitoring layer.

In Figure 15, no assumption is made regarding the Application Entity (AE): the implementation of functionality in the AE can be done in a variety of ways, depending on the needs of the application designers and developers. In some cases, the implementation can be made without using microservices (and an underlying Microservice Architecture).

In case the implementation is done with microservices, some of the functional blocks present in the CSE part of Figure 15 (e.g. data collection, communications, computation, storage, search engine, data usage) can be (fully of partly) present in the AE. However, whereas the microservices in the CSE are expected to be rather generic, those in the AE will probably be much more "domain specific".

The CSFs have not been defined with a micro service architecture in mind. Indeed, the choice of dividing a CSE into microservices should always be left up to specific implementations, which means that the optimizations made for two different deployment scenarios may result in two different choices of grouping into microservices.

The example in Figure 16 is showing a possible implementation where two CSE are involved that may embed different functionalities:

- A CSE with all the microservices described in Figure 15. This could typically correspond to one possible implementation of microservices on a cloud-based platform that will support all the common services offered for Data Collection, Communication, Computation, Storage, Search Engine, Data Usage.
- A CSE with only a part of the microservices described in Figure 15. This could correspond to one possible implementation of microservices on a basic gateway that will not support parts of the common services whose implementation can be hindered by the limitations of the gateway. Services like Storage (if no persistence is required), Search engine or Data usage may not be implemented.

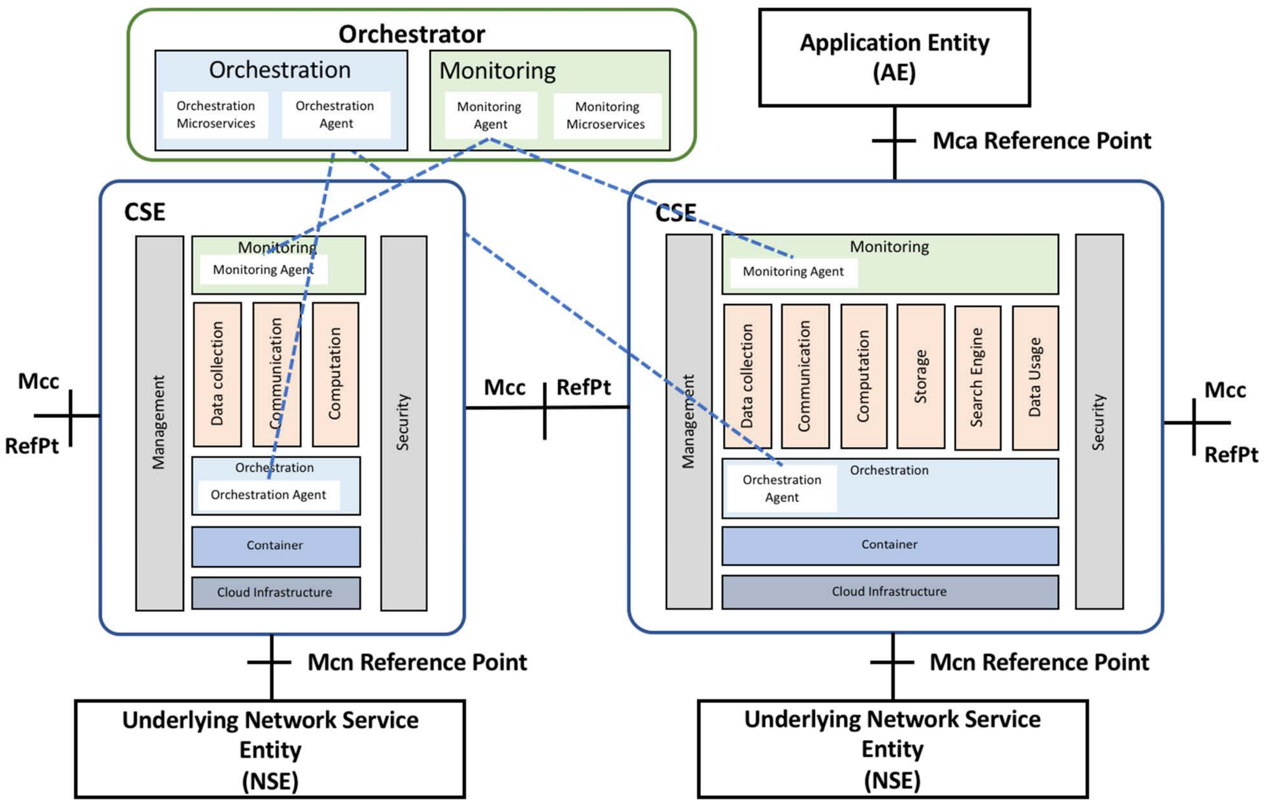

Figure 16: An example of implementation options of the microservices HLA

8 Conclusions

8.1 Implications

The introduction of microservices and the support of Virtualization is expected to be a major improvement factor in the development of IoT systems. It is also important to note that this introduction comes with a number of impacts - and challenges - regarding the current approach to IoT systems. Amongst them:

- Efficient implementations:

  As already pointed out in the previous clauses, the introduction of microservices comes with expectation of improvement in the implementation of IoT systems, in particular in terms of efficiency. In order to ensure that the expected benefits will materialize, microservices will need to be supported by open architectures; a large catalogue of effective, easily available and possibly certified (OSS) components; supporting integration platforms, etc. Moreover, as soon as they will be part of large - and even complex - IoT systems, microservices will not be working in full isolation from others and their development will have to made with this in mind and depart from ad-hoc solutions.

- Deconstruction of monoliths:

  Microservices are specially expected to support the development of fine-grained components that interact with a plurality of others. In the case of existing IoT systems, their introduction may require that some currently used solutions (applications, building blocks) be split in smaller (interacting) units. This "deconstruction" may be quite significant - and require a certain amount of effort - to benefit from microservices. The support of an architecture - like the High-Level Architecture (HLA) mentioned in the previous clause - will be needed, as well as its mapping to other existing HLAs.

- Role and place of legacy:

  Not all IoT systems are greenfield and are incorporating existing (and sometimes long existing) elements. The introduction of microservices may not be possible for the entire system, for cost reasons as well as difficulties related to old or unmaintained technologies. The potential coexistence of old and new parts (the latter based on microservices) will require some adaptations, in particular as long as the communication mechanisms used by microservices may not be supported in the legacy part.

- Federation of systems:

  The question of federation of systems is much debated in the IoT community, in particular by the Research community, and somehow by the Standard community. The need for federation of IoT systems is becoming a credible requirement for some Use Cases like Smart Cities (and even in Industrial IoT). The impact of IoT Virtualization still needs to be assessed: the evaluation of relevant use cases may provide a useful input.

- Security and trust:

  Security is universally pointed out as a key enabler for trusted IoT systems. From this standpoint, there is no silver-bullet solution for security is the current (IoT) systems. The introduction of virtualization (and the widespread use of microservices) may not change this situation drastically without a specific effort. However, given the higher degree of flexibility combined with the possibility to use (trusted) open source components offered by virtualization, there is a possibility that more efficient security solutions be more easily introduced thanks to microservices. But this will come as a miracle, but only if this is taken as a central requirement by the microservices architects and developers.

- Standards:

  The most visible benefits of IoT virtualization (and the introduction of microservices) point to the massive reuse of OSS components. From this standpoint, the traditional role of standards is going to be challenged. On the other hand, the potential "deconstruction" of IoT systems may also require that some existing building blocks (and associated standard) are re-considered with a layering approach that may in turn promote the development of new standards. For instance, where a complex Application Server currently exist, virtualization may introduce the need for new interfaces that may become in turn supported by standards.

- Regulation:

  IoT Virtualization will introduce new requirements on the systems concerned that are currently not apparent but may become significant. In particular, virtualization will introduce more consideration of the role of Cloud Service Providers and some of aspects of their solutions. Such aspects may involve business (such as Service Level Agreements) but also regulation, the example of GDPR being currently the most discussed one.

- Education:

  The technical expertise requirements for developing full-fledged IoT systems are quite high. They span a large range of capabilities such as IoT platforms and protocols, big data or security-by-design. The introduction of microservices is asking even more from the architects and developers: expertise on the development of the microservices themselves, on the integration of open source components, etc. This is going to be a challenge in terms of the education support for such varied and highly demanded profiles.

8.2 Lessons Learned

After the definition of the Microservices HLA and its applications to IoT Virtualization in clause 7, a more practical validation has been made. The experiences gained from the evaluation of the Open Source Components landscape (see ETSI TR 103 528 [i.1]) and from the implementation of the Proof-of-Concept (see ETSI TR 103 529 [i.2]) have produced guidelines for the architects and designers of IoT systems based on microservice architectures that can be found in both Technical Reports.

In addition to the above cited guidelines, some lessons have been drawn regarding the approach taken in the present document. The main lessons learned are the following:

- The approach taken in the present document has been confirmed by the complementary approach of selecting Open Source Software (OSS) components and of using (some of) them for the implementation of a Proof-of-Concept (PoC). The principles of layering exposed in the Microservice HLA have been useful in the selection of the components for the PoC implementation and in the fast deployment of the resulting application.

- One major enabling factor is that there is that there is a large number of OSS components available that have a very high level of technical readiness (TRL-9). The mix of chosen components can be different from one implementation to another one and may dictated by different considerations.

- Taking into account the constraints at the edge requires attention in the architecture in order to fully support the implementation. The orchestrator can consider the resources at the edge in the same manner as those on the cloud but this can create problems if those resources are more constrained (size, computing power, storage capabilities, etc.) and may have less non-functional capabilities (e.g. latency on a Raspberry Pi ™) that may degrade the performance of the system: to alleviate this, deployment rules have to be carefully defined.

- If applications are developed with the microservices approach, they can benefit from the advantages of virtualization (in terms of scalability, reliability, etc.) but also from the possibility to orchestrate the application microservices (via the orchestrator) and still benefit from what is done for the common services.

8.3 Recommendations to oneM2M

Based on the above considerations regarding the mapping of the Microservices HLA on oneM2M and on the feedback provided by the Proof-of-Concept (see ETSI TR 103 529 [i.2]), the following recommendations can be made to oneM2M:

- Integrate the orchestrator in the oneM2M framework as a standardized element, with the corresponding Reference Point(s).

- Add a "high-throughput/big data" binding to the current list, since no existing solution for this is included in oneM2M.

Annex A: Relationship to big data

The boundaries between Cloud, IoT and big data are rapidly blurring, actually they may be seen by young developers as quite artificial. While the present document is focused mostly on IoT virtualization, this clause explores the relationship with big data. Big data is defined by Recommendation ITU-T Y.3600 [i.8] as a "*A paradigm for enabling the collection, storage, management, analysis and visualization, potentially under real-time constraints, of extensive datasets with heterogeneous characteristics.*"

Big data is often linked with the famous "4V" properties:

- Volume;
- Velocity;
- Variety; and
- Veracity.

Volume denotes the dimension that gives the Big Data field its name. Velocity describes the pace of data generation from a diversity of data sources such as physical sensors. Variety describes different aspects of data sources, including structured and unstructured, multimedia, languages, etc. Veracity addresses both the natural and artificially injected noises and miss-information in many (open) data sources.

There are several ways to qualify the veracity of collected data. Several studies have for example dealt with fault detection and isolation of IoT datasets or time series using rule-based fault detection as well as self-learning fault detection algorithms based on e.g. an approach of statistics sliding window. The rule-based approach would be based on available manufacturer datasheets including rules when available. The self-learning algorithm is based on determining trend vectors and comparing such vectors with longer term historic data.

One of the challenges in IoT fault detection and isolation (as a possible scenario for data veracity) is to perform data processing in a passive manner, that is without impacting the normal operations of an IoT system. The advantage of passive monitoring is two-fold:

1) no additional traffic to the devices and gateways, and
2) runs without any disruption to the ongoing data collection mechanisms.

Using a micro services architecture (as proposed in clause 7.2.1) with a message broker that allows a microservice to subscribe to receive specific data sets provides a perfect fit for a microservice performing IoT fault detection in a passive manner. The microservice would typically run using its own computing and memory resources while the broker would simply perform additional replication of collected data sets when such data is exchanged using the broker. This is explained in Figure A.1.

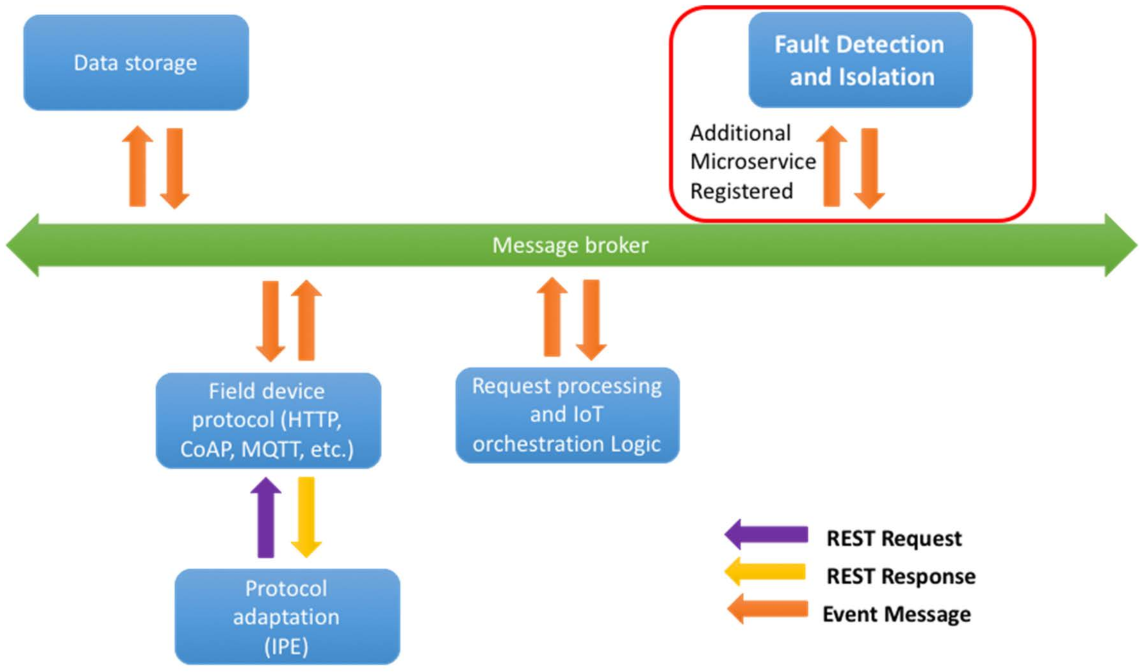
Data storage
Fault Detection and Isolation
Additional Microservice Registered
Message broker
Field device protocol (HTTP, CoAP, MQTT, etc.)
Request processing and IoT orchestration Logic
Protocol adaptation (IPE)
REST Request
REST Response
Event Message

Figure A.1: Passive IoT fault detection and isolation module

In Figure A.1, the Fault detection and isolation microservice can subscribe (from the message broker) to received datasets or time series data in order to perform the detection of one of the following faulty scenarios:

- **Outliers**: A single isolated event that is outside the expected range of values to be returned. An example of outlier is provided in Figure A.2.

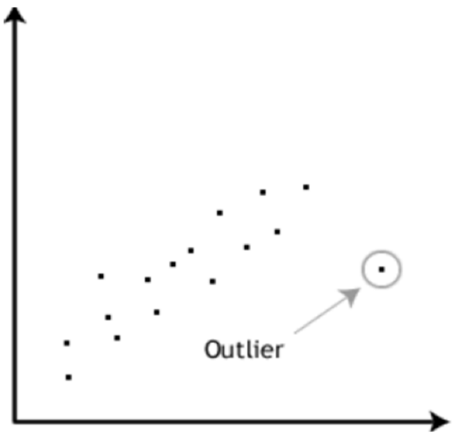

Figure A.2: Fault detection: Outlier data-point

- **Stuck-at faults**: A series of data values with little or no variation for a period of time longer than expected.
- **Spikes**: A change in gradient over a period of time much greater than expected. The biggest difficulty with spikes is the issue of determining if a gradient change is part of normal behaviour such as alarm or a faulty behaviour of a sensor. A spike example is depicted in Figure A.3.

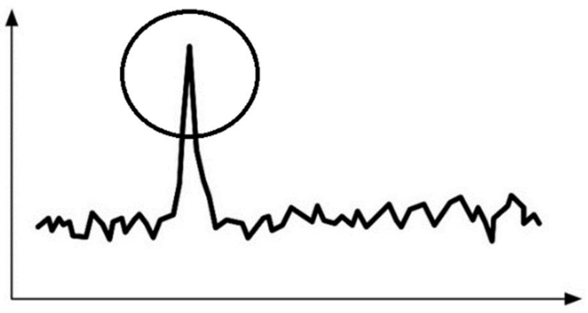

Figure A.3: Fault detection: Spike behaviour

Annex B: Relationship with NFV

B.0 Introduction

The Network Functions Virtualisation (NFV) Industry Specification Group (ISG) has been created in ETSI in 2012 with the overall vision that the application of Virtualization technology may help address many of the evolution challenges that new usages (such as IoT and M2M communications) pose to existing networks and, in turn, simplify the roll-out of network services, reduce deployment and operational costs and improve network management automation.

As stated by ETSI: "*With NFV, standard IT virtualization technology is adapted to consolidate many network equipment types onto industry-standard high-volume servers, switches and storage. This involves implementing network functions in software which can run on a range of industry-standard server hardware. This software can then be moved to, or introduced in, various locations in the network as required.*" (see http://www.etsi.org/technologies-clusters/clusters/networks).

B.1 Virtualization in the NFV Architecture

The NFV ISG has initially worked on the identification of use cases for virtualization and their implication on the virtualization of traditional network functions. Based on this, the ISG has defined the NFV Architectural Framework, its main components and reference points (see ETSI GS NFV 002 [i.9]).

More specifically, the ISG has defined the "NFV Infrastructure" (NFVI): "The NFVI is the totality of the hardware and software components which build up the environment in which VNFs are deployed. The NFVI is deployed as a distributed set of NFVI-nodes in various locations to support the locality and latency requirements of the different use cases and the NFVI provide the physical platform on which the diverse set of VNFs are executed; enabling the flexible deployment of network functions envisaged by the NFV Architectural Framework." (see ETSI GS NFV-INF 001 [i.10]).

The high level NFV framework (see ETSI GS NFV 002 [i.9]), can be seen in Figure B.1 and consists of three main domains:

- **Virtualized Network Function (VNF)**: the software implementation of a network function which is capable of running over the NFVI.
- **NFV Infrastructure (NFVI)**: includes the diversity of physical resources and how they can be virtualized. The NFVI supports the execution of the VNFs.
- **NFV management and orchestration (MANO)**: covers the orchestration and lifecycle management of physical and/or software resources that support the infrastructure virtualization and the lifecycle management of VNFs. NFV Management and Orchestration focuses on all virtualization-specific management tasks necessary in the NFV framework.

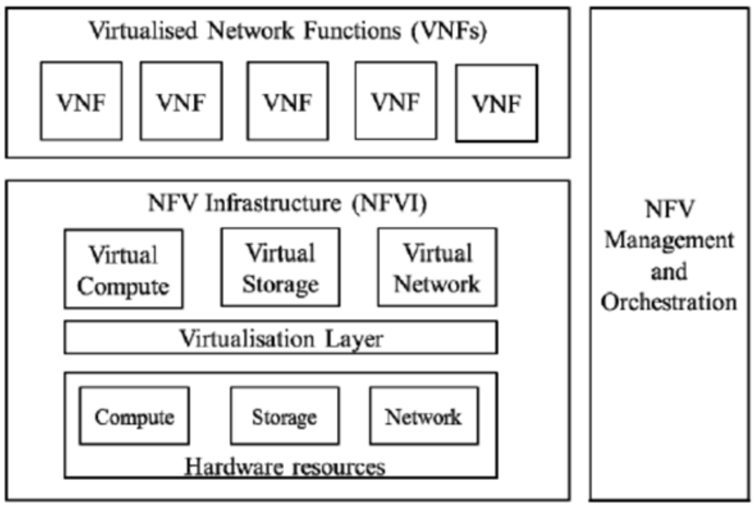

Figure B.1: High Level NFV Framework

B.2 The NFV architecture and the Microservice-based HLA

The Microservices-based architecture presented in clause 7.3 and the NFV Architectural Framework have been developed in different contexts. In particular, NFV in addressing primarily the "traditional" networks (e.g. those operated by Telecom Service Providers) and focuses on their major Network Functions. In contrast, the "Microservices-based HLA" is spanning across high-layers of the "IoT Stack" and potentially address a larger set of "IoT functions". Consequently, both approach share common objectives and similarities, as well as differences that are not fully visible today given the lack of maturity of the corresponding deployments. Some preliminary remarks are outlined below:

- The potentially critical importance of the support from standards:

  The NFV Architectural framework has been defined with the expectation that its approach to virtualization should be supported by a very precise set of standards (developed by NFV or not) supporting Reference Points. A similar approach has been taken by oneM2M for the development of IoT systems using its architectural framework. In both cases, the challenge posed to virtualization is to make sure that the support of standards will not be compromised.

- Specialization:

  An important difference between the NFV architectural approach and the IoT virtualization approach described in clause 7.3 is that NFV is more focused on the functions related to the network and does not systematically take into account higher-layer functions.

- NFV as an IoT Virtualization Framework:

  As long as the IoT functions that are targeted for virtualization are matching the ones defined in the NFV Architectural Framework, the latter can be used as an IoT virtualization framework where a VNF is replaced by an "IoT Virtualized Function". The main advantage of this approach is that the Reference Points defined by the NFV Architectural Framework can be used by the virtualized IoT system.

- The role of the network: 5G, orchestration:

  The deployment of 5G networks involves the redesign of the Core Network where the NFV architecture will play a central part, in particular the MANO component of the NFVI. The approach taken by the industry for the implementation of MANO has been to use Open Source both as a methodology and as a technology. Two major Open Source projects are under development: Open Source MANO (OSM) and Open Network Automation Platform (ONAP).

- Microservices as a new programming paradigm:

  The technologies available for the implementation of microservices-based applications have reached a level of maturity and effectiveness that has made their usage become mainstream in software engineering. The development of the Virtualized Network Functions of NFV is largely based on this approach. This is a strong enabler to the adoption of microservice-based architectures like the one described in clause 7.3.

- The role of Open Source components in the procurement of Telecom Operators:

  Telecom Service Providers are increasingly using (and sometimes requesting) Open Source components in their procurement. The deployment of NFV systems will also include Open Source components (and not just for the MANO part). This can be seen as another enabler to the adoption of microservices-based architectures and should improve the effectiveness of NFV as an IoT virtualization framework.

Despite the differences outlined above, the two approaches are not mutually exclusive and microservices (and microservices-based architectures) can be used in the NFV context, for example for the implementation of Virtualized Network Functions.

The coming years are going to see the concurrent deployment of IoT systems using microservices-based HLA and of 5G systems with IoT sub-systems. They will provide feedback and more lessons learned in terms of e.g. which use cases and business models are better supported by which approach; how far the microservice software development model will become the de-facto approach; which will be the relative part for the support of standards (and frameworks such as NFV and oneM2M and their related reference points) versus the use of OSS components, etc.

Annex C: Change History

Date	Version	Information about changes
November 2017	0.1.0	Early version for discussion at SmartM2M TC Meeting #44
December 2017	0.2.0	First stable version for discussion at meeting with SmartM2M on January 9th, 2018
January 2018	0.2.1	Revised stable version for discussion at meeting with SmartM2M on January 9th, 2018
February 2018	0.3.0	Revised version for presentation at SmartM2M #45
February 2018	0.3.1	Slightly revised version for SmartM2M #45 (document date; additional paragrah in clause 7.3.1)
May 2018	0.4.0	Revised version for presentation at meeting with SmartM2M on May 7th
May 2018	0.9.0	Final stable draft for review by SmartM2M participants
July 2018	0.9.1	ETSI Secretariat check, EditHelp Clean-up
July 2018	1.0.0	Final version for publication

History

Document history		
V1.1.1	July 2018	Publication