



TECHNICAL REPORT

**SmartM2M;
IoT over Cloud back-ends:
A Proof of Concept**

Reference

DTR/SmartM2M-103529

Keywords

cloud, IoT, open source, proof of concept,
virtualisation

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Important notice

The present document can be downloaded from:

<http://www.etsi.org/standards-search>

The present document may be made available in electronic versions and/or in print. The content of any electronic and/or print versions of the present document shall not be modified without the prior written authorization of ETSI. In case of any existing or perceived difference in contents between such versions and/or in print, the only prevailing document is the print of the Portable Document Format (PDF) version kept on a specific network drive within ETSI Secretariat.

Users of the present document should be aware that the document may be subject to revision or change of status.

Information on the current status of this and other ETSI documents is available at

<https://portal.etsi.org/TB/ETSIDeliverableStatus.aspx>

If you find errors in the present document, please send your comment to one of the following services:

<https://portal.etsi.org/People/CommiteeSupportStaff.aspx>

Copyright Notification

No part may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm except as authorized by written permission of ETSI.

The content of the PDF version shall not be modified without the written authorization of ETSI.

The copyright and the foregoing restriction extend to reproduction in all media.

© ETSI 2018.

All rights reserved.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members.

3GPP™ and **LTE™** are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners.

oneM2M logo is protected for the benefit of its Members.

GSM® and the GSM logo are trademarks registered and owned by the GSM Association.

Contents

Intellectual Property Rights	5
Foreword.....	5
Modal verbs terminology.....	5
Introduction	5
1 Scope	7
2 References	7
2.1 Normative references	7
2.2 Informative references.....	7
3 Definitions and abbreviations.....	7
3.1 Definitions.....	7
3.2 Abbreviations	8
4 Virtualization of IoT: A Proof-of-Concept.....	8
4.1 Virtualized IoT and Cloud-Native Computing.....	8
4.2 The rationale for a Proof-of-Concept	8
4.3 Content of the report.....	9
5 Main elements of the Proof-of-Concept.....	9
5.1 The Use Case.....	9
5.2 High-Level Architecture.....	10
5.3 Technical choices	10
5.4 Message flow.....	11
5.5 Auto scaling up and down	12
6 Implementation.....	12
6.1 Initial Deployment architecture	12
6.2 Set-up of the PoC Deployment Infrastructure	13
6.2.1 Installation of the Kubernetes cluster.....	13
6.2.1.1 Introduction.....	13
6.2.1.2 Utilization of Google™ Cloud Kubernetes Engine.....	13
6.2.1.3 Installation from scratch.....	14
6.2.2 Installation of the IoT Components	17
6.2.3 Horizontal autoscaling	18
7 Conclusions	19
7.1 Introduction	19
7.2 Lessons Learned and Recommendations.....	19
Annex A: Change History	20
History	21

List of figures

Figure 1: PoC High level architecture	10
Figure 2: Open source components in the Proof-of-Concept	11
Figure 3: Message Flow in the Proof-of-Concept	11
Figure 4: Kubernetes Horizontal Pod Autoscaler (HPA) architecture	12
Figure 5: PoC initial deployment architecture.....	13

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: "*Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards*", which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (<https://ipr.etsi.org/>).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

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 this STF will investigate will also have to be validated: to this extent, a Proof-of-Concept implementation involving a massive number of virtualized elements will be made.

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

- ETSI TR 103 527 [i.1]: "SmartM2M; Virtualized IoT Architectures with Cloud Back-ends".
- ETSI TR 103 528 [i.2]: "Landscape for open source and standards for cloud native software for a Virtualized IoT service layer".
- ETSI TR 103 529 (the present document): "IoT over Cloud back-ends: A Proof of Concept".

1 Scope

The present document:

- Recalls the main elements of the Proof-of-Concept (PoC) in support of IoT Virtualization: use case description, high-level architecture of the application developed, main technical choices.
- Presents the main implementation choices.
- Outlines the lessons learned and the possible impact of future IoT Virtualization implementations.

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 103 527: "SmartM2M; Virtualized IoT Architectures with Cloud Back-ends".
- [i.2] ETSI TR 103 528: "SmartM2M; Landscape for open source and standards for cloud native software applicable for a Virtualized IoT service layer".

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 Organizations (SDO).

3.2 Abbreviations

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

API	Application Programming Interface
CPU	Central Processing Unit
CSE	Common Service Entity (in oneM2M)
CSP	Cloud Service Provider
EOF	End Of File
HLA	High-Level Architecture
HPA	Horizontal Pod Autoscaler
IaaS	Infrastructure as a Service
NFS	Network File System
OSS	Open Source Software
PaaS	Platform as a Service
PoC	Proof of Concept
RAM	Random Access Memory
RC	Replication Controller
REST	REpresentational State Transfer
SaaS	Software as a Service
SDO	Standards Development Organization
SSH	Secure SHell
SSO	Standards Setting Organization
STF	Specialist Task Force
VM	Virtual Machine
YAML	YAML Ain't Markup Language

4 Virtualization of IoT: A Proof-of-Concept

4.1 Virtualized IoT and Cloud-Native Computing

The IoT industry has started to understand and evaluate the potential benefits of Cloud-Native Computing for the fast, effective and future-safe development of IoT systems combining the strengths of both IoT and Cloud industries in a new value proposition. The expectation of Cloud-Native applications is to benefit from offerings from Cloud Service Providers (CSP) that may cover all or part of the layers of Virtualized application, via Infrastructure as a Service (IaaS), Platform as a Service (PaaS) or Software as a Service (SaaS).

In the case of IoT applications, the trade-off between what is delegated to the Cloud Service Provider and what is kept in the hands of the application developers may vary depending a large number of potential factors and will finally materialize into different architecture, design and implementation choices.

The approach of Cloud-Native Computing is now widely supported by a large set of technologies embedded in Cloud-Native Infrastructures in support of Cloud-Native Applications. These technologies are now very diverse, technology-ready (as shown in the landscape of Open Source components described in ETSI TR 103 528 [i.2]) and support all the layers of a Micro-Service Architecture (such as the one described in ETSI TR 103 527 [i.1]).

4.2 The rationale for a Proof-of-Concept

As pointed out in ETSI TR 103 527 [i.1], "there is probably a large number of [IoT] 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 Use Cases more effective: it would generally improve the efficiency of their implementation or support interoperability at a more fine-grained level (or both)".

The Cloud-Native Computing technologies have been made much easier to apprehend, to master and to package into more and more complex systems. However, the effective usage of the vast catalogue of components potentially applicable for IoT Virtualization is still under evaluation in the IoT community.

The Proof-of-concept (PoC) of IoT Virtualization exposed in the present report is an implementation of the "Horizontal Up and Down Auto-Scaling" Use Case which has been selected in ETSI TR 103 527 [i.1] as a well-adapted example for the following reasons:

- It demonstrates the feasibility of IoT Virtualization on a "real-life" Use Case applicable to a large number of sectors (aka "verticals").
- It addresses a feature (auto-scaling) that is being deemed as very critical in virtualized IoT and for which an implementation via the use of "off-the-shelf" Open Source Software components requires some validation.
- It makes use of a great number of the Open Source Software components described in ETSI TR 103 528 [i.2].

4.3 Content of the report

Clause 5 outlines the main elements for the definition, design and implementation of the selected Use Case, mostly the High-Level Architecture, the main Open Source components selected and the message flow.

Clause 6 describes the initial deployment architecture and explains the steps to be followed for the set-up of the deployment infrastructure.

Clause 7 outlines the main lessons learned from the implementation and provides a few basic recommendations.

5 Main elements of the Proof-of-Concept

5.1 The Use Case

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 will 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".

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.

5.2 High-Level Architecture

The main goal of this PoC is to experiment resource monitoring and autoscaling up and down in terms of communications, processing, storage and data analytics. A typical IoT architecture is considered composed of IoT devices, Message Brokers, Data Processors, Databases and Search Engines, and Dashboards. The monitoring and scaling service monitors resources consumed by each micro service and scale up and down the instances accordingly to optimize system performance. The PoC high level architecture is described in Figure 1.

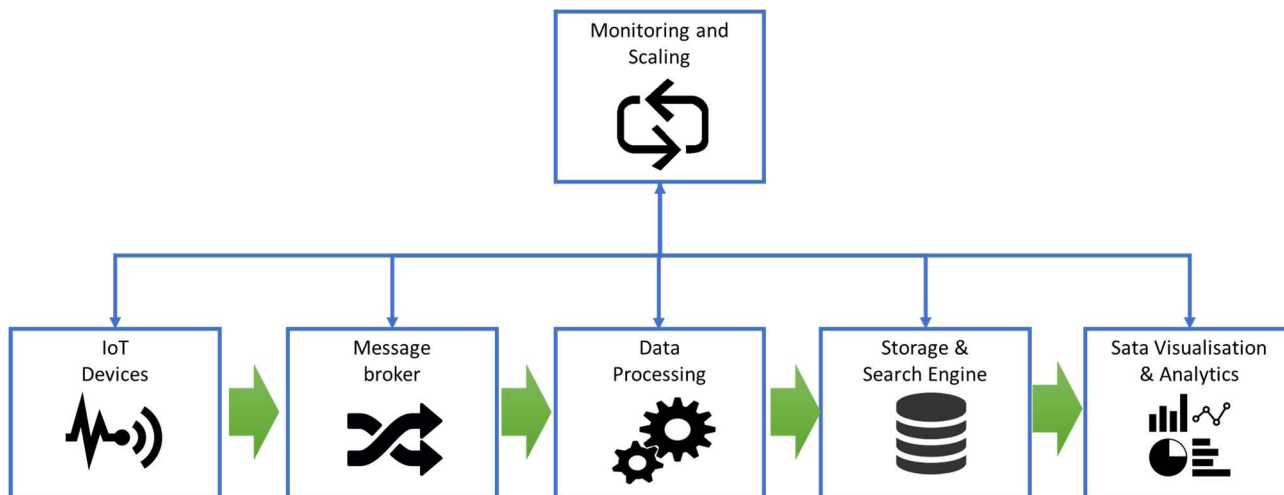


Figure 1: PoC High level architecture

5.3 Technical choices

The following open source components are considered to build the IoT system architecture as describe in Figure 2:

- Docker as Container solution: Docker is an open-source solution that automates the deployment of applications inside software containers. Docker implements a high-level API to provide lightweight containers that run processes in isolation. Building on top of facilities provided by the Linux kernel, a Docker container, unlike a virtual machine, does not require or include a separate operating system. Instead, it relies on the kernel's functionality and uses resource isolation.
- Kafka as Message Broker: Kafka is an open source a distributed streaming platform that follows a publish/subscribe architecture to manage data streams. It is used for two broad classes of application:
 - 1) Building real-time streaming data pipelines that reliably get data between systems or applications; and
 - 2) Building real-time streaming applications that transform or react to the streams of data. Kafka runs as a cluster on one or more servers. The Kafka cluster stores streams of records in categories called "topics".
- Logstash as data processor: Logstash is an open source, server-side data processing pipeline that ingests data from a multitude of sources simultaneously, transforms it, and then sends it to elastic search. It supports a variety of inputs that pull in events from a multitude of common sources, all at the same time. Easily ingest from logs, metrics, web applications, data stores, all in continuous, streaming fashion.
- Elasticsearch as database and search engine: Elasticsearch is an open source storage and real-time search and analytics engine capable of searching and analysing big volumes of data in near real time. It is open-source and is built to be used by distributed systems.
- Kibana as Visualization interface: Kibana is an open source web-based graphical interface for searching, analysing, and visualizing data stored in Elasticsearch. It uses the REST interface of Elasticsearch to retrieve the data and enables users to create customized dashboard views of their data and allows them to query and filter data. It allows to visualize data in a variety of charts, tables, lines, circles and maps.

- **Kubernetes as Container Orchestrator:** Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery. Heapster, combined with Grafana and InfluxDB, enables resource monitoring and performance analysis based on Kubernetes.

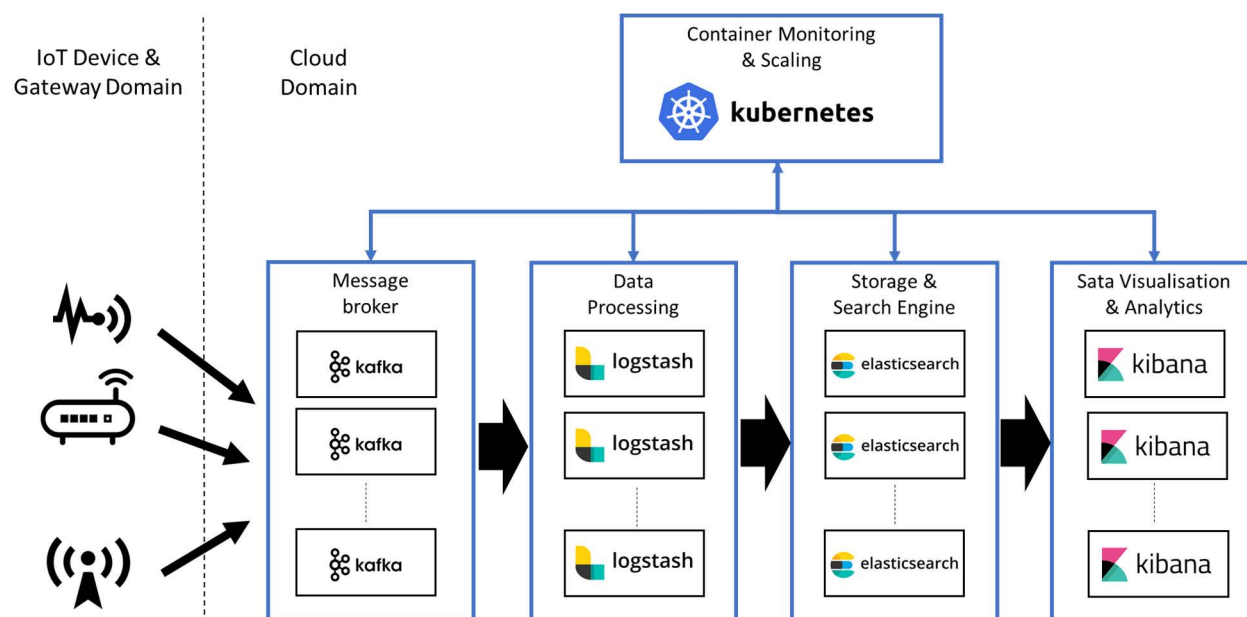


Figure 2: Open source components in the Proof-of-Concept

5.4 Message flow

IoT devices push data to message brokers into specific "topics". Data processors subscribe to the relevant topic, process the received data and store it into a database. The data visualization and analytics interacts with the search engine API to display data and perform analytics operation.

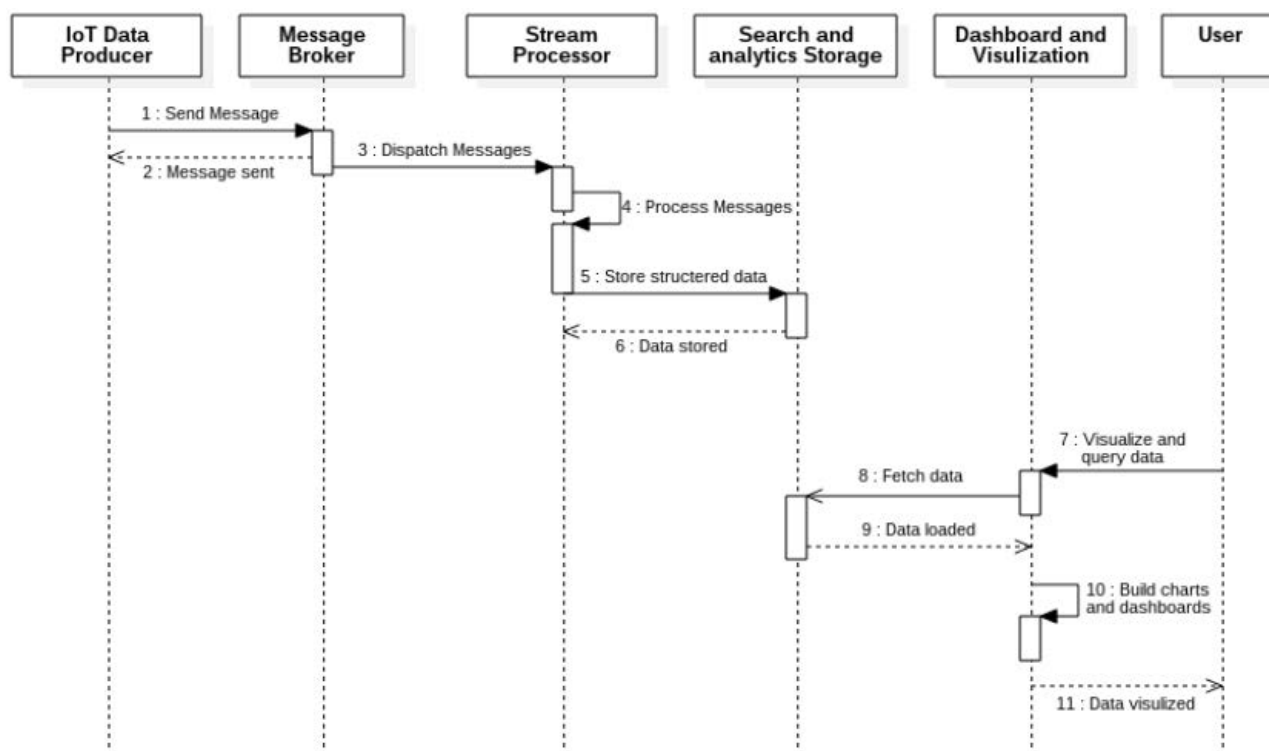


Figure 3: Message Flow in the Proof-of-Concept

5.5 Auto scaling up and down

In Kubernetes, the Horizontal Pod Autoscaler automatically scales the number of pods in a replication controller (RC), deployment based on the observed CPU/RAM utilization or on custom metrics. The Horizontal Pod Autoscaler controller can fetch metrics in two different ways: direct Heapster access, and REST client access (as shown in Figure 4).

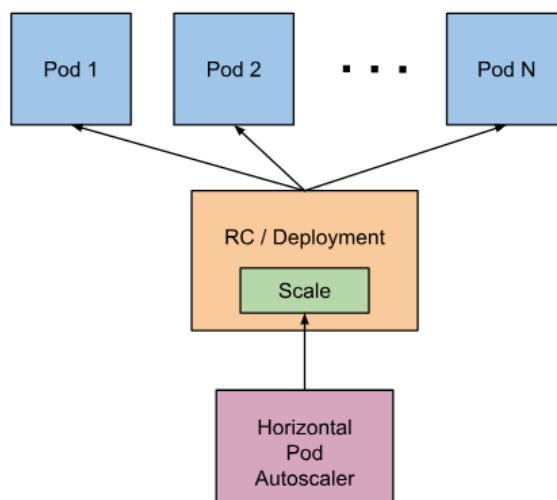


Figure 4: Kubernetes Horizontal Pod Autoscaler (HPA) architecture

6 Implementation

6.1 Initial Deployment architecture

Figure 5 shows the initial deployment architecture considered for implementing PoC. Each microservice is replicated in 3 replicas and is deployed as a Kubernetes service to ensure horizontal scalability and fault tolerance. For the sake of simplicity, the IoT devices are simulated using Apache Kafka Producers. The number of Kafka producers is fully managed by Kubernetes in order to control data throughput and easily test system scalability. Apache Kafka requires Zookeeper for broker coordination: Zookeeper is added to the deployment architecture as a Kubernetes service. Elasticsearch engine is composed of two types of nodes: Master and Data that are deployed also as Kubernetes services. The proposed architecture offers full flexibility in terms of scalability. The replicas of every service could be scaled up and down in a seamless way to meet requirements in terms of high throughput and low latency. The usage of cloud infrastructure resources is continuously optimized.

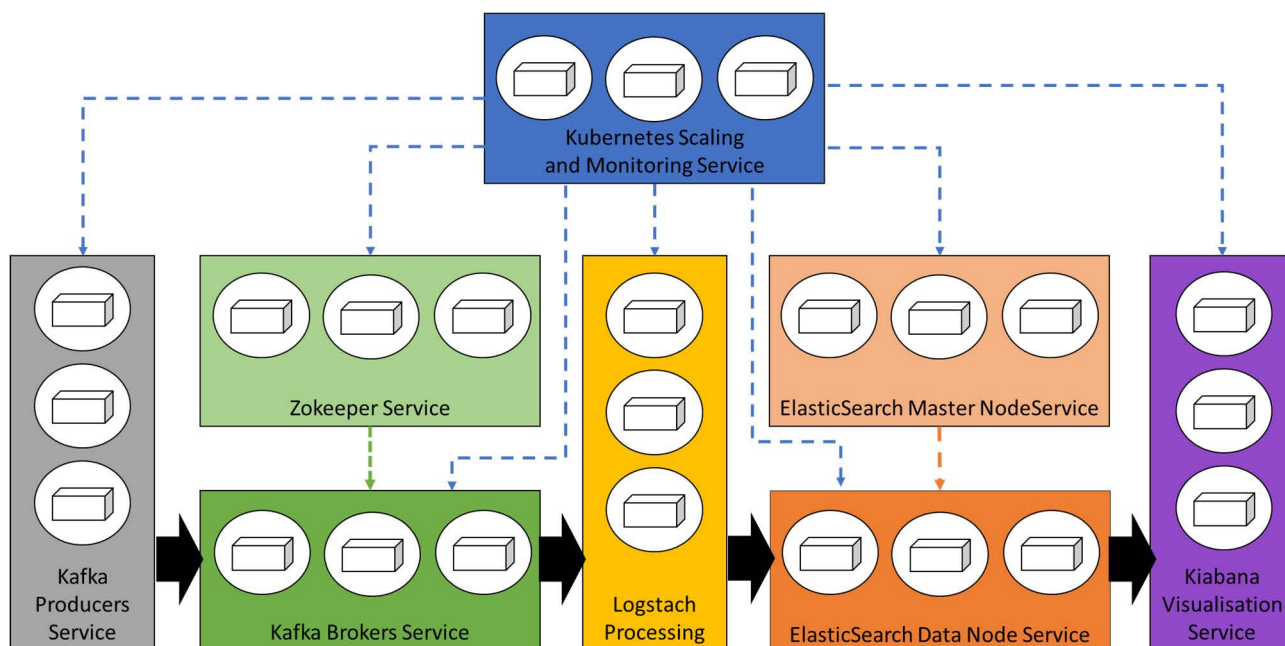


Figure 5: PoC initial deployment architecture

6.2 Set-up of the PoC Deployment Infrastructure

6.2.1 Installation of the Kubernetes cluster

6.2.1.1 Introduction

Two scenarios can be followed depending on the choice regarding the Cloud Service Provider:

- 1) Installation using existing the Cloud Service Provider infrastructure.
- 2) Complete install from scratch using a private infrastructure.

6.2.1.2 Utilization of Google™ Cloud Kubernetes Engine

Step 1 Login to Google™ Kubernetes Engine

Use an already existing Google™ account or create one to login to Google™ Cloud Platform.

- The Google™ Cloud Platform is available on the following link:
 - <https://cloud.google.com/free>

Click on Compute Engine on the left menu to open VMs management dashboard.

- The Compute Engine is directly accessible using the following link:
 - <https://console.cloud.google.com/compute>

Click on Kubernetes Engine on the left menu to open Kubernetes Cluster management dashboard.

- The Kubernetes Engine is directly accessible using the following link:
 - <https://console.cloud.google.com/kubernetes>

Step 2 Create a Kubernetes Cluster

On Google™ Kubernetes Engine dashboard, click on "Create Cluster" button.

Enter your Kubernetes cluster configuration including name, zone, machine type, node image and size then click on create.

The cluster will take few minutes to be created.

Step 3 Cluster nodes swap and virtual memory configuration

Open the Compute Engine dashboard.

The following commands should be repeated for every VM:

- Click on SSH button, wait for the SSH console to open in a separate page and configure the SWAP and Virtual memory using the following command:

- Configure the Swap:

```
$ sudo fallocate -l 1G /swapfile
sudo chmod 600 /swapfile
sudo mkswap /swapfile
sudo swapon /swapfile
echo '/swapfile none swap sw 0 0' | sudo tee -a /etc/fstab
```

- Configure the VM (required for elastichsearch instances)

```
$ sudo sysctl -w vm.max_map_count=512000
```

Step 4 Connect to Kubernetes cluster using cloud shell

Google™ Kubernetes Engine offers a cloud shell to connect to your cluster directly from your browser.

In the Kubernetes clusters page, click on "connect" button available in front of your cluster. Then chose "Run in Cloud Shell".

Wait few second for the console to be started on the same page. Then press Enter to connect.

Step 5 Clone the IoT Prototype project

Clone the virtualized IoT prototype project from the dedicated ETSI git repository on your cluster cloud shell.

```
$ git clone repo-url
```

Jump to clause 6.2.2 to continue the installation of IoT components.

6.2.1.3 Installation from scratch

Step 1 Installation of Docker

Docker is installed on each node of the cluster.

Follow these steps to install Docker:

```
$ sudo apt-get update

$ sudo apt-get install \
apt-transport-https \
ca-certificates \
curl \
software-properties-common
```

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
$ sudo add-apt-repository \
"deb [arch=amd64] https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) \
stable"
$ sudo apt-get update
$ sudo apt-get install docker-ce
```

Step 2 Installation of Kubernetes

Kubernetes is installed in each node of the cluster.

Follow these steps to install Kubernetes:

```
$ apt-get update && apt-get install -y apt-transport-https
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
$ cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
$ apt-get update
$ apt-get install -y kubelet=1.6.4-00 kubeadm=1.6.4-00
```

Step 3 Building the Kubernetes cluster

Select a node to be the master by executing the following command.

```
kubeadm init --pod-network-cidr=10.244.0.0/16
```

At the end of the console log, note the token that will be used to add nodes to the cluster.

Configure the KUBECONFIG environment variable.

```
$ cd ~
$ cp /etc/kubernetes/admin.conf .
$ export KUBECONFIG=$HOME/admin.conf
$ echo 'export KUBECONFIG=$HOME/admin.conf' >> .bashrc
```

Install the networking component flannel

The necessary permission for the flannel deployment is added:

```
$ kubectl create -f
https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-
flannel-rbac.yml
```

Deploy the flannel as a Daemonset using the following command:

```
$ kubectl create -f
https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-
flannel.yml
```

Once the networking component is up and running, the worker nodes can be cluster by executing the following command on every node and using the noted token:

```
$ kubectl join --token=XXXXXX 192.168.1.1:6443
```

The Kubernetes cluster is up and running.

Step 4 Install Kubernetes NFS provisioner

NFS provisioner allows dynamic allocation of volumes which are required by statefulsets.

First, nfs-common needs to be install on each machine belonging to the cluster by executing the following command:

```
$ apt-get install nfs-common
```

The NFS provisioner needs to be deployed on a specific machine, otherwise it will result in data loss.

```
$ kubectl label node dev1 storage=nfs
```

NOTE: In the yaml file, the pod has this field:

```
nodeSelector:
  storage=nfs
```

This ensures that the provisioner will only run in that specific machine, but if Kubernetes needs to be restricted from assigning other pods to that machine for performance reasons, you need to execute the following command:

```
$ kubectl edit nodes dev1
```

Now add the taint NoSchedule in the spec section:

```
spec:
  taints:
  - effect: NoSchedule
    key: node-role.kubernetes.io/master
    timeAdded: null
```

But this will restrict all pods from being scheduled on the node, so a toleration is added in the YAML file of the provisioner:

```
tolerations:
- effect: NoSchedule
  key: node-role.kubernetes.io/master
  operator: Exists
```

The NFS provisioner .YML files are available on the folder kube_workspace/nfs.

Go to kube_workspace directory and execute the following command:

```
$ kubectl -n kube-system create -f nfs
```

This will create the necessary permissions, the service, the class storage and the deployment.

Step 5 Installation of Kubernetes Dashboard

Install the Kubernetes Dashboard using the following command:

```
$ kubectl create -f https://git.io/kube-dashboard
```

This will create a dashboard deployment that will not be exposed. If exposure is required, the service can be edited:

```
$ kubectl edit svc kubernetes-dashboard -n kube-system
```


Then add the following line:

```
spec:
  type: NodePort
  ports:
    - nodePort: 32323
```

Note that the port number can also be fixed via the field `nodePort`. Kubernetes requires a port number between 30 000 and 33 000.

6.2.2 Installation of the IoT Components

Step 1 Installation of Zookeeper

Zookeeper .YML files are available on the folder `kube_workspace/zookeeper`.

Go to the `kube_workspace` folder and execute the following command to install zookeeper:

```
$ kubectl create -f zookeeper
```

Step 2 Installation of Kafka Broker

Kafka .YML files are available on the folder `kube_workspace/kafka`.

Go to `kube_workspace` and execute the following command to install Kafka:

```
$ kubectl create -f kafka
```

Step 3 Installation of ElasticSearch

On each worker machine in the kubernetes cluster, the system is configured by adding the following line in the `/etc/sysctl.conf`:

```
vm.max_map_count=512000
```

ElasticSearch .YML files are available on the folder `kube_workspace/elasticsearch`.

Go to `kube_workspace`.

Setup the elasticsearch services using the following command:

```
$ kubectl create -f elasticsearch
```

Deploy elasticsearch master node using the following command:

```
$ kubectl create -f elasticsearch/master
```

Deploy elasticsearch data node using the following command:

```
$ kubectl create -f elasticsearch/data
```

Step 4 Installation of Logstash

Logstash .YML files are available on the folder `kube_workspace/logstash`.

Go to `kube_workspace` folder and execute the following command to install logstash:

```
$ kube_workspacekubectl create -f logstash
```

Step 5 Installation of Kibana

Kibana .YML files are available on the folder kube_workspace/kibana.

Go to kube_workspace folder and execute the following command to install kibana:

```
$ kubectl create -f kibana
```

The Kibana port can be found by executing:

```
$ kubectl get svc
```

Step 6 Installation of Heapster, InfluxDB and Graphana for monitoring

Install Heapster

Heapster read from kubernetes api, store the information in influxdb and grafana visualize data from influxdb.

Give Heapster required permissions to watch the resources:

```
$ kubectl create -f heapster-permissions.yml
```

Deploy influxdb:

```
$ kubectl create -f influxdb.yml
```

Deploy heapster. Two configurations are required, the source and the sink. In our case, heapster should read from kubernetes and store in influxdb:

```
command:  
- /heapster  
- --source=kubernetes:https://kubernetes.default  
- --sink=influxdb:http://monitoring-influxdb.kube-system.svc:8086  
kubectl create -f heapster.yml
```

Deploy Grafana:

```
$ kubectl create -f grafana.yml
```

Step 7 Installation of Kafka Producers

Kafka Producer .YML files are available on the folder kube_workspace/producer.

Go to kube_workspace folder and execute the following command to deploy kafka producers:

```
$ kubectl create -f producer
```

6.2.3 Horizontal autoscaling

Horizontal Pod Autoscaler (HPA), is supported in a standard way by kubectl. There is a special autoscaling command for easy creation of a Horizontal Pod Autoscaler. For instance, executing the following command will create an autoscaler for replication controller kafka, with target CPU utilization set to 80 % and the number of replicas between 2 and 5:

```
$ kubectl autoscale rc kafka --min=2 --max=5 --cpu-percent=80
```

7 Conclusions

7.1 Introduction

The PoC described above is presenting one amongst many architecture and implementation. It is therefore one example and is not to be considered as a reference implementation.

The architecture of the PoC refers to the Microservices-based High-Level Architecture (HLA) as described in ETSI TR 103 527 [i.1] and is one illustration of the proposed mapping on the oneM2M CSE. In particular, it introduces a subset of the potential Microservices presented. This PoC can be extended in order to include more features, in particular support of other non-functional properties: this can be done by the introduction of new Microservices.

A recommendation is made in ETSI TR 103 528 [i.2] regarding the integration of a Kafka binding in oneM2M. With such a binding, the extension of the implementation of the PoC in a oneM2M context becomes more straightforward.

7.2 Lessons Learned and Recommendations

A few early lessons learned regard the approach taken for the set-up and deployment of the supporting infrastructure:

- The choice of the technical components is very open, given the large number of components available that have a very high level of technical readiness (TRL-9). The mix of chosen components can be different in other implementations and may be dictated by different considerations such as the ecosystem in which the IoT system will be deployed.
- The installation and deployment of the infrastructure is quick and efficient.
- The expectations in terms of reliability and performance have been fully met.

Horizontal scalability and fault tolerance of stateful applications require more consideration than for stateless applications. The main reason for this is the need to implement persistence in a stateful application. Scaling requires the orchestrator not just to manage the duplication of applications instances over computing resources but also the distribution of storage on "persistence volumes". It is recommended that applications and storage should not be collocated on the same computing resources because a newly created replica (in case of failure of a running replica) should not start with an empty storage.

Scaling down requires more attention than scaling-up. Stateless as well as stateful applications cannot be stopped before it is sure that they have finished their operation. For stateless applications, any related data can be discarded. In the case of stateful applications, any data on the replica shutting down needs to be redistributed on the other instances before it is actually shut down. It should be noticed that there is built-in support for this in most of the OSS components and data bases: a careful preliminary analysis - before designing the application - of the way this is done will be useful.

A fine-tuning of auto-scaling parameters (e.g. overall infrastructure allocated budget, minimum and maximum number of instances, CPU and RAM thresholds) is necessary before the proper parameters can be validated.

The horizontal scalability and fault tolerance principles applied for the IoT system will also be applied to the orchestrator components in order to improve performance and to avoid Single-Point-Of-Failure.

Annex A: Change History

Date	Version	Information about changes
Mar. 2018	0.0.1	Early version for discussion at March 27 th , 2018 Ad-hoc Meeting with SmartM2M TC
May 2018	0.1.0	First stable draft for discussion at May 7 th , 2018 Ad-hoc Meeting with SmartM2M TC
May 2018	0.5.0	Final version for STF review before upload on the SmartM2M portal for final review
May 2018	0.9.0	Final draft for approval by SmartM2M
July 2018	0.9.1	ETSI Secretariat checks and EditHelp clean-up

History

Document history		
V1.1.1	August 2018	Publication