

TECHNICAL UNIVERSITY OF CRETE  
SCHOOL OF ELECTRICAL & COMPUTER ENGINEERING  
INTELLIGENT SYSTEMS LABORATORY



THESIS

LoRaWare: A service oriented architecture for  
interconnecting LoRa devices with the Cloud

Tsakos Konstantinos

COMMITTEE:

Petrakis G.M. Euripides Professor, ECE, TUC (supervisor)

Deligiannakis Antonios Assoc. Professor, ECE, TUC

Sotiriadis Stelios Assist. Professor, Birkberk, UL, UK

CHANIA 2018

# ABSTRACT

In this work, we show how the advantages of a Low Power Wide Area Network (LPWAN) protocol can be exploited to support greater availability and usability of Internet of Things (IoT) applications. The main idea is to show how LPWAN networks can be interconnected with the Cloud where IoT data can be transferred securely for persistent storage and further processing. To show proof of concept, we experimented with LoRa technology and LoRaWAN, the latest successful representative of LPWAN protocols. The LoRaWAN protocol is characterized by long range, low power and low data rate transmission. We applied a typical experimental setup with LoRa environmental sensors transmitting measurements over long distances using LoRa protocol to gateways and from there to the cloud. Our scenario is application agnostic (as it is independent of sensor types and need not be aware of the actual IoT measurements). The advantage of this scenario is that whole cities can be covered with a small number of gateways where, each gateway is capable of dealing with even thousands of sensors.

The LoRa Nodes transmit RF packets with LoRa modulation which are captured by one or more Gateways. The Gateway receives LoRa packets from sensors in range and re-transmits them to the cloud over internet using an IP protocol (e.g. a basic one such as UDP). In this work we opt for MQTT a more elaborate lightweight publish-subscribe IP protocol offering advance security, better routing control and visibility of the communication (i.e. easier handling and control of data packets).

The focus of this work is on interconnecting the gateways with the cloud. We develop the Network Server, a solution that runs as a service on the cloud and whose purpose is to

(a) receive LoRa packets from gateways (b) decode their payload from ASCII characters to bits (base 64 encoding) (c) deduplicate packets received from more than one gateways (d) decrypt the payload (AES 128 bit encryptions is applied by LoRA) and (e) make data available to the cloud services in NGSI - JSON format. For outgoing packets the same solution is applied in reverse order: packets are encrypted, encoded and transmitted to target gateways (is supported by the LoRaWare with some additional configurations-haven't been tested). The service is developed for FIWARE cloud, a pan-European cloud infrastructure which is supported by the EU. NGSI is the protocol which is used by every Generic Enabler of Fiware ecosystem as a data exchange model. One of them is the Publish/Subscribe Context Broker which mediates between devices and applications. Our architecture, referred to as LoRaWare, allows IoT developers to enhance the capabilities of LoRa enabled applications using advanced FIWARE services supporting persistent storage and data analytics, service synthesis using Mashups etc. In our example implementation, humidity and temperature measurements are monitored in real - time on the cloud while historical values are stored in MySQL database.

We run an exhaustive set of experiments using real and simulated (but realistic) data in order to study the system response time and system scalability.

We report average end-to-end processing times (i.e. from the moment IoT data are received by the network server to the time they are stored in the database) and also average time spent on each service in the processing sequence.

To study system scalability we stressed the system with a large synthetic (but realistic) payload simulating up to 2.000 requests (i.e. data packets) received by the Network server

and processed on the cloud. Our experimental results demonstrate that our system is still capable of performing real - time or close to real time for many thousands of concurrent requests.

In a different experiment, we study the practical range of LoRa transmission in a real urban environment (in the city of Chania) with two gateways placed apart from each other. The experimental results reveal that the rate of packages captured by any of the two gateways decreases drastically with the distance from the sensors in all cases.

## ΠΕΡΙΛΗΨΗ

Σε αυτή την εργασία, παρουσιάζουμε πως τα πλεονεκτήματα ενός χαμηλής κατανάλωσης και ευρείας περιοχής δικτύου(LPWAN) πρωτοκόλλου μπορούν να εκμεταλλευθούν για να υποστηρίξουν μεγαλύτερη διαθεσιμότητα και χρησιμότητα εφαρμογών του διαδικτύου των πραγμάτων (IoT). Η βασική ιδέα είναι να δείξουμε πως τα LPWAN δίκτυα μπορούν να διασυνδεθούν με το Νέφος όπου τα IoT δεδομένα μπορούν να μεταφερθούν με ασφάλεια για μόνιμη αποθήκευση και επιπλέον επεξεργασία. Για να αποδείξουμε τη γενική ιδέα, πειραματιστήκαμε με την τεχνολογία LoRa και το LoRaWAN, το τελευταίο αντιπροσωπευτικό από τα LPWAN πρωτόκολλα. Το LoRaWAN πρωτόκολλο χαρακτηρίζεται από μεγάλη εμβέλεια, χαμηλή ισχύ και χαμηλό ρυθμό μετάδοσης των δεδομένων. Εφαρμόσαμε μία τυπική πειραματική εγκατάσταση με LoRa περιβαλλοντικούς αισθητήρες που μεταδίδουν μετρήσεις σε μία μεγάλη απόσταση χρησιμοποιώντας το LoRa πρωτόκολλο μέχρι τα gateways και από εκεί στο Νέφος. Το σενάριό μας είναι ανεξάρτητο από τους τύπους των αισθητήρων και δεν χρειάζεται να γνωρίζει το είδος των πραγματικών IoT μετρήσεων. Το πλεονέκτημα αυτού του σεναρίου είναι ότι ολόκληρες πόλεις μπορούν να καλυφθούν με ένα μικρό αριθμό πυλών δικτύου(gateways) όπου κάθε μία είναι ικανή να εξυπηρετήσει μέχρι και χιλιάδες αισθητήρες.

Οι LoRa κόμβοι μεταδίδουν πακέτα ραδιοσυχνοτήτων με διαμόρφωση LoRa τα οποία συλλαμβάνονται από μία ή περισσότερες πύλες. Κάθε πύλη λαμβάνει τα LoRa πακέτα από εξ αποστάσεως αισθητήρες και αναμεταδίδει αυτά στο Νέφος μέσω του διαδικτύου χρησιμοποιώντας ένα IP πρωτόκολλο (δηλ. κάποιο συνηθισμένο όπως το UDP). Σε αυτή την εργασία χρησιμοποιήσαμε το MQTT ένα πιο λεπτομερές ελαφρύ publish-subscribe IP πρωτόκολλο που

προσφέρει προχωρημένη ασφάλεια, καλύτερο έλεγχο δρομολόγησης και παρακολούθησης της επικοινωνίας (δηλαδή καλύτερο χειρισμό και έλεγχο των πακέτων).

Το επίκεντρο αυτής της εργασίας είναι η διασύνδεση των πυλών δικτύου με το Νέφος. Αναπτύξαμε έναν εξυπηρετητή δικτύου , μία λύση η οποία τρέχει ως υπηρεσία στο Νέφος και που σκοπός της είναι (α) να λαμβάνει LoRa πακέτα από τις πύλες δικτύου (b) να αποκωδικοποιεί το περιεχόμενό τους από χαρακτήρες ASCII σε bits (base64 κωδικοποίηση) (c) να κρατάει ένα μοναδικό πακέτο αν λαμβάνεται από περισσότερες από μία πύλες δικτύου (d) να αποκρυπτογραφεί το περιεχόμενο (AES 128 bit κρυπτογράφηση εφαρμόζεται από το LoRa) και (e) να κάνει τα δεδομένα διαθέσιμα στις υπηρεσίες του Νέφους σε NGSI-JSON μορφή. Για τα πακέτα που βγαίνουν έξω από το Νέφος η ίδια λύση εφαρμόζεται σε αντίστροφη σειρά: τα πακέτα κρυπτογραφούνται, κωδικοποιούνται και μεταδίδονται σε στοχευόμενες πύλες (υποστηρίζεται από το LoRaWare με κάποιες πρόσθετες τροποποιήσεις-δεν έχει δοκιμαστεί). Η υπηρεσία έχει αναπτυχθεί για το Fiware Cloud , μία πανευρωπαϊκή υποδομή νέφους που υποστηρίζεται από την Ευρωπαϊκή Ένωση. Το NGSI πρωτόκολλο χρησιμοποιείται από κάθε Generic Enabler του Fiware οικοσυστήματος ως ένα μοντέλο ανταλλαγής δεδομένων. Ένας από αυτούς είναι και ο Publish/Subscribe Context Broker ο οποίος μεσολαβεί μεταξύ συσκευών και εφαρμογών. Η αρχιτεκτονική μας, αναφέρεται ως LoRaWare, επιτρέπει στους IoT developers να βελτιώσουν τις δυνατότητες των εφαρμογών LoRa χρησιμοποιώντας προχωρημένες υπηρεσίες του Fiware που υποστηρίζουν μόνιμη αποθήκευση, ανάλυση δεδομένων, σύνθεση δεδομένων και υπηρεσιών με Mashups κτλπ. Στο παράδειγμα της υλοποίησής μας, παρακολουθούνται μετρήσεις υγρασίας και θερμοκρασίας σε πραγματικό χρόνο στο Νέφος ενώ

ταυτόχρονα αποθηκεύονται οι τιμές στο ιστορικό σε μία MySQL βάση δεδομένων.

Εκτελούμε ένα εξαντλητικό σύνολο πειραμάτων χρησιμοποιώντας πραγματικά δεδομένα προσομοίωσης ώστε να μελετήσουμε τον χρόνο απόκρισης του συστήματος και την επεκτασιμότητα του.

Αναφέρουμε το μέσο χρόνο επεξεργασίας από άκρο σε άκρο (δηλαδή από τη στιγμή που τα IoT δεδομένα λαμβάνονται από τον εξυπηρετητή δικτύου μέχρι τη στιγμή που αποθηκεύονται στη βάση δεδομένων) και επίσης τον μέσο χρόνο που δαπανάται σε κάθε υπηρεσία στην ακολουθία της επεξεργασίας.

Για να μελετήσουμε την επεκτασιμότητα του συστήματος στρεσάραμε το σύστημα με ένα τεράστιο (πραγματικό) φορτίο προσομοιωμένο με 2000 αιτήματα (πακέτα δεδομένων) που λαμβάνονται από τον εξυπηρετητή δικτύου και επεξεργάζονται στο Νέφος. Τα πειραματικά μας αποτελέσματα δείχνουν ότι το σύστημα μας είναι ικανό να αποδίδει σε πραγματικό χρόνο ή κοντά σε πραγματικό χρόνο για πολλά χιλιάδες ταυτόχρονα αιτήματα.

Σε ένα διαφορετικό πείραμα, μελετήσαμε το πρακτικό εύρος της LoRa μετάδοσης σε ένα πραγματικό αστικό περιβάλλον (στην πόλη των Χανίων) με δύο πύλες δικτύου που τοποθετήθηκαν μακριά η μία από την άλλη. Τα πειραματικά αποτελέσματα που προέκυψαν αποκαλύπτουν ότι ο ρυθμός των λαμβανόμενων πακέτων για κάθε μία από τις πύλες δικτύου μειώνεται δραστικά σε σχέση με την απόσταση των αισθητήρων σε όλες τις περιπτώσεις.

# ACKNOWLEDGMENTS

I would like to thank professor Euripides Petrakis for his valuable help and insightful comments. I would like also to thank the members of the laboratory for the excellent communication and collaboration.



# Contents

1. Introduction.....	11
1.1 PROBLEM DEFINITION AND CONTRIBUTIONS.....	12
2. Background-Related Work .....	14
2.1 Cloud Computing .....	14
2.1.1 Definition .....	14
2.1.2 Advantages of Cloud.....	14
2.1.3 Cloud service models.....	16
2.1.4 Cloud deployment models .....	18
2.1.5 Virtualization .....	19
2.1.6 Openstack.....	23
2.1.7 Fiware .....	25
2.1.8 The NGSI Information Model.....	26
2.1.9 IDAS GE .....	28
2.1.10 Orion Context Broker GE .....	29
2.1.11 Keyrock Identity Manager GE.....	29
2.2 Internet of Things (IoT) .....	30
2.2.1 Devices.....	30
2.2.2 IoT protocols.....	32
2.2.3 IoT platforms .....	36
2.3 LoRa Technology.....	37
2.3.1 Characteristics of LoRa Technology.....	38
2.3.2 LoRaWAN protocol .....	38
2.3.3 Classes of LoRa devices.....	39
2.3.4 LoRa Data Rates.....	41
2.3.5 Security .....	41
2.3.6 LoRa Network Architecture .....	44
3. LoRaWare Reference Architecture.....	44
3.1 LoRa Network .....	46
3.2 LoRa Backend .....	47
4. Implementation of LoRaWare Architecture.....	50
4.1 Architectural Diagrams of LoRaWare .....	51
4.2 LoRa Devices.....	55
4.2.1 Nexus Board .....	56

4.2.2 Nexus Demoboard.....	57
4.2.3 PCB Antenna 868 MHz UFL.....	57
4.2.4 Arduino Sketches.....	58
4.3 Lorank 8 Gateway.....	60
4.3.1 The Lora-Gateway-Bridge Service .....	61
4.4 The Cloud Services of the LoRaWare Architecture .....	63
4.4.1 The Mosquitto MQTT Broker.....	63
4.4.2 LoRa Server .....	65
4.4.3 LoRa App Server .....	67
4.4.4 JSON FILTERING .....	75
4.4.5 JSON/MQTT IoT Agent.....	76
4.4.6 Orion Context Broker .....	79
4.4.7 Keyrock Identity Management .....	81
4.5 Application Logic & Smart Home Web Application.....	84
5. Performance Evaluation .....	88
5.1 Evaluation of the Cloud Infrastructure.....	88
5.2 LoRa Network Evaluation .....	95
5.2.1 First Experiment .....	97
5.2.2 Second Experiment.....	101
6. Conclusion – Future Work.....	103
6.1 Conclusions.....	104
6.2 Future Work .....	105
7. References.....	106
8. Image References .....	107

# 1.Introduction

The idea of the Internet of Things (IoT) combined with cloud computing, opens new horizons in the field of real time data collection and analysis. Due its scalability, modularity and affordability (no up-front investment, low maintenance cost) Cloud is the ideal deployment environment of IoT applications.

**Cloud Computing** makes computing resources accessible over the network, allows high degree of resource sharing (as many user can be accessing the same infrastructure or service at the same time). In the cloud, resources are provisioned and released on-demand allowing users to use the cloud resources based on their actual needs and be charged for this. Finally, a scalable infrastructure is scalable to accommodate demands of the ever increasing number of users and applications. These operations are enabled by monitoring the actual resource usage at all times using appropriate monitoring solutions.

**Internet of Things** is coming into the scene to allow interconnection of user devices and enable the processing of the huge amounts of information that are routinely acquired by the millions of devices connected to the internet. The use of wearable sensors and mobile devices and their capability for Internet connectivity provides significant benefits in applications areas that require fast and continuous monitoring of user data from anywhere (e.g. activity, health monitoring, smart cities etc.).In real-life applications, huge amounts of data are collected and analyzed (e.g. for scientific or business purposes).

Furthermore, A Low-Power Wide-Area Network (**LPWAN**)<sup>1</sup> or LPWA network or Low-Power Network(LPN) is a type of wireless telecommunication wide area network designed to allow long range communications at a low bit rate among things (connected objects) , such as sensors operated on a battery. The **low power, low bit rate** and **intended use** distinguish this type of network from a wireless WAN that is designed to connect users or businesses, and carry more data, using more power. The LPWAN data rate ranges from 0,3 kbit/s to 50 kbit/s

---

<sup>1</sup> <https://en.wikipedia.org/wiki/LPWAN>

per channel. A LPWAN may be used to create a private wireless sensor network, but may also be a service or infrastructure offered by a third party, allowing the owners of sensors to deploy them in the field without investing in gateway technology.

Due to its long range, high mobility, security and low power consumption, **LoRa** is considered to be a promising technology and is projected to support billions of IoT devices which can be connected to internet. Public and private networks using this technology can provide coverage that is greater in range compared to that of existing cellular networks.

## 1.1 PROBLEM DEFINITION AND CONTRIBUTIONS

The present work attempts to become the technological bridge of the three important technologies referred to above namely, IoT, Cloud computing and LPWAN. We are motivated by the need to support interconnection of large numbers of devices to the cloud, where monitoring of the IoT network, persistent storage and analysis of the big amounts of IoT data can take place taking advantage of the scalability, modularity and low cost maintenance of cloud services. In particular, taking advantage of the modularity and extendibility of cloud services, new applications can be designed and deployed on the cloud capable of serving the needs of diverse application domains and of large numbers of users.

The need to interconnect networks of LoRa devices and the cloud has been acknowledged many times in the past by many investigators and practitioners. The appropriate technological bridge (referred to as Network server) would run on the cloud and its purpose would be to receive LoRa encoded data packets from gateways, decode the packets and forward the transmitted payloads in a format that is commonly understandable by services running on the cloud. An obvious disadvantage of such a solution is the lack of standardization in this area mainly due to the heterogeneity of the cloud providers. However, most providers adopt the REST standardization for their services and a JSON

format for data transmission rather than the more general XML format (which would incur an additional overhead for encoding and decoding). Besides, the scalability of such a solution has not been studied elsewhere.

In this work we design and develop LoRaWare a Network server as a service on FIWARE cloud infrastructure. An important advantage of the proposed solution is adoption of the NGSI framework and protocol for subscribing LoRa networks to the cloud and the use of the Publish-Subscribe context broker service of Fiware for making their data available to the users. The service allows LoRa Devices to subscribe their information to the cloud and at the same time, user to subscribe to this information and get notified every time new information becomes available.

To show proof of concept we propose an event based architecture on FIWARE which is capable of using information generated by LoRa networks of devices. This is a generic architecture that shows how data can be stored, analyzed and used by other services in a form that is network agnostic (ie. an application or user need not to be aware of the peculiarities of LoRa in order to use information from LoRa networks or in order to communicate with the LoRa devices).

In order to study scalability of the solution we stress the LoRaWare (the network service and all services subscribing to LoRa devices) with large synthetic (but realistic) payloads produced by many concurrent users and we report total response times of the Network Services as well response times consumed by each individual LoRa Service. The experimental results demonstrate the LoRaWare (and the Network Server) scales-up very well responding in real or in close to real-time in all cases.

In a different experiment, we study the practical range of LoRa transmission in a real urban environment (in the city of Chania) with two gateways placed apart from each other. The experimental results reveal that the rate of packages captured by any of the two gateways decreases drastically with the distance from the sensors in all cases.

## 2. Background-Related Work

### 2.1 Cloud Computing

#### 2.1.1 Definition

According to National Institute of Standards and Technology (NIST) definition<sup>2</sup>, Cloud Computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of three service models and four deployment models.

#### 2.1.2 Advantages of Cloud<sup>3</sup>

Cloud computing has 3 main benefits:

- **Flexibility** which means that users can scale services to fit their needs, customize applications and access cloud services from anywhere with an internet connection.
- **Efficiency** as enterprise users can get applications to market quickly, without worrying about underlying infrastructure costs or maintenance.
- **Strategic value** because cloud services give enterprises a competitive advantage by providing the most innovative technology available.

Every advantage of the above hides more specific benefits.

To begin with, flexibility means **scalability**, as cloud infrastructure can be scaled on demand to support fluctuating workloads. In addition, users can have **storage options** as they can choose public, private or hybrid storage offerings, depending on security needs and other considerations. Furthermore, it provides **control choices**, which means

---

<sup>2</sup> <https://www.nist.gov/sites/default/files/documents/itl/cloud/cloud-def-v15.pdf>

<sup>3</sup> <https://www.ibm.com/cloud/learn/benefits-of-cloud-computing>

that organizations can determine their level of control as-a-service options. These include software as a service (SaaS), platform as a service (PaaS) and infrastructure as a service (IaaS). Also, **tool selection** gives the opportunity to users to select from a menu of prebuilt tools and features a solution that fits their specific need. Finally, **security features** such as virtual private cloud, encryption and API keys help keep data secure.

On the other hand, efficiency means **accessibility** because cloud-based applications and data are virtually accessible from any internet-connected device. Furthermore, developing in the cloud enables users to get their applications to market quickly (**speed to market**). Also, it provides **security to data**, as hardware failures do not result in data loss because of networked backups. In addition, efficiency brings **savings on equipment**. Cloud computing uses remote sources, saving organizations the cost of servers and other equipment. Finally, a “utility” **pay structure** means that users only pay for the resources they use.

Finally, strategic value means **streamlined work** as cloud service providers (CSPs) manage underlying infrastructure, enabling organizations to focus on application development and other priorities. In addition, service providers regularly update offerings to give users the most up-to-date technology (**Regular updates**). Also, worldwide access means teams can collaborate from widespread locations (**collaboration**). Finally, organizations can move more nimbly than competitors who must devote IT resources to managing infrastructure (**Competitive edge**).

### 2.1.3 Cloud service models<sup>4</sup>

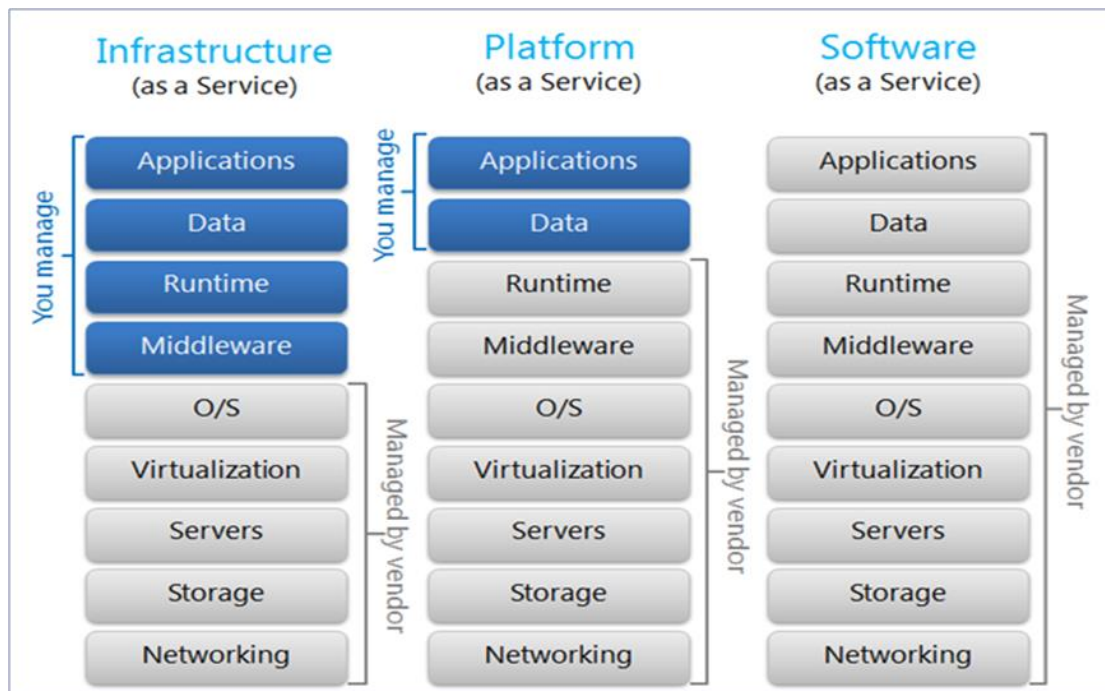


Figure 1: Cloud Service Models - Differences between IaaS, PaaS and SaaS

There are 3 cloud service models (Figure 1): Infrastructure as a service (**IaaS**), Platform as a service (**PaaS**) and Software as a service (**SaaS**). At IaaS, a vendor provides clients pay-as-you-go access to storage, networking, servers and other computing resources in the cloud. At PaaS, a service provider offers access to a cloud-based environment in which users can build and deliver applications. The provider supplies underlying infrastructure. Finally, at SaaS, a service provider delivers software and applications through the internet. Users subscribe to the software and access it via the web or vendor APIs.

To become more specific, **IaaS** is a cloud computing offering in which a vendor provides users access to computing resources such as servers, storage, and networking. Organizations use their own platforms and applications within a service provider's infrastructure. Instead of purchasing hardware outright, users pay for IaaS on demand. Infrastructure is scalable depending on processing and storage needs. In addition, IaaS saves enterprises the costs of buying and maintaining their own hardware. Because data is on the cloud, there can be no single

<sup>4</sup> <https://www.ibm.com/cloud/learn/iaas-paas-saas>



point of failure. Finally, this model enables the virtualization of administrative tasks, freeing up time for other work.

**PaaS** is a cloud computing offering that provides users with a cloud environment in which they can develop, manage and deliver applications. In addition to storage and other computing resources, users are able to use a suite of prebuilt tools to develop, customize and test their own applications. PaaS provides a platform with tools to test, develop and host applications in the same environment. Furthermore, it enables organizations to focus on development without having to worry about underlying infrastructure. This model gives the opportunity to providers to manage security, operating systems, server software and backups and it also facilitates collaborative work even if teams work remotely.

Finally, **SaaS** is a cloud computing offering that provides users with access to a vendor's cloud-based software. Users do not install applications on their local devices. Instead, the applications reside on a remote cloud network accessed through the web or an API. Through the application, users can store and analyze data and collaborate on projects. SaaS vendors provide users with software and applications via subscription model. Users do not have to manage, install or upgrade software because providers manage this. Data is secure in the cloud, as equipment failure does not result in loss of data. Also, use of resources can be scaled depending on service needs and applications are accessible from almost any internet-connected device, from virtually anywhere in the world.

## 2.1.4 Cloud deployment models<sup>5</sup>

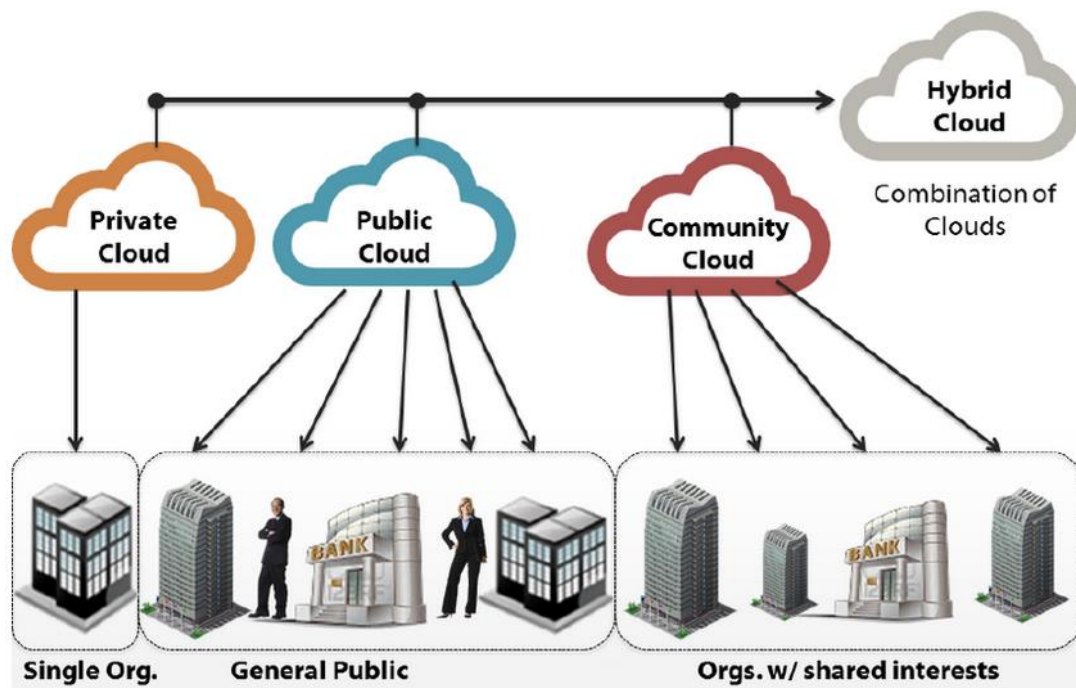


Figure 2: Cloud deployment models

There are 4 cloud deployment models (Figure 2):

- The most common and well-known deployment model is **Public Cloud**. A Public Cloud is a huge data center that offers the same services to all its users. The services are accessible for everyone and much used for the consumer segment. Examples of public services are Facebook, Google and LinkedIn. For consumers, Public Cloud offerings are usually free of charge, for professionals there is usually a per-per-use (or user) pricing model. The Public Cloud is always hosted by a professional Cloud supplier.
- The other commonly used deployment model is **Private Clouds**. There are lots of discussions for how strict the definition of Private Clouds should be. In general a customer's internally hosted data center is regarded as a Private Cloud. If we add virtualization and automation, such a setup may very well be regarded as a Private Cloud. A professional Cloud vendor may also offer a Private Cloud to their customers by supporting a separate hardware environment in the data center. A Private Cloud is therefore

<sup>5</sup> <https://www.visma.com/blog/cloud-basics-deployment-models/>

mostly suited for sensitive data, where the customer is dependent on a certain degree of security. Private Clouds, to a certain degree, lose the economy of scale compared to a Public Cloud.

- A way to preserve the benefits of economy of scales with the Private Cloud is a **Community Cloud**. This is cooperation between users who share some concerns like security, application types, legislative issues and efficiency demands. In other words, a Community Cloud is a closed Private Cloud for a group of users. For governments this is called Government Cloud and is a type of Cloud that is more and more adapted. Due to legislative issues, a Government Cloud may be the answer to country specific judicial concerns.
- The **Hybrid Cloud** is a combination of both Private and Public. This is a setup that is much used for large companies. Vital data is usually preferred in a Private Cloud and supporting services in Public, for instance search, email, blogs, CRM etc. In other words strategic applications are run separately.

### 2.1.5 Virtualization

**Virtualization**<sup>6</sup> refers to the creation of a virtual resource such as a server, desktop, operating system, file, storage or network.

It is the key to cloud computing<sup>7</sup>, since it is the enabling technology allowing the creation of an intelligent abstraction layer which hides the complexity of underlying hardware or software.

The main goal of virtualization is to manage workloads by radically transforming traditional computing to make it more scalable. Virtualization has been a part of the IT landscape for decades now, and today it can be applied to a wide range of system layers, including operating system-level virtualization, hardware-level virtualization and

---

<sup>6</sup> <https://www.techopedia.com/definition/719/virtualization>

<sup>7</sup> <https://www.computerworld.com/article/2468246/cloud-computing/why-virtualization-is-the-foundation-of-cloud-computing.html>

server

virtualization.

## TRADITIONAL AND VIRTUAL ARCHITECTURE

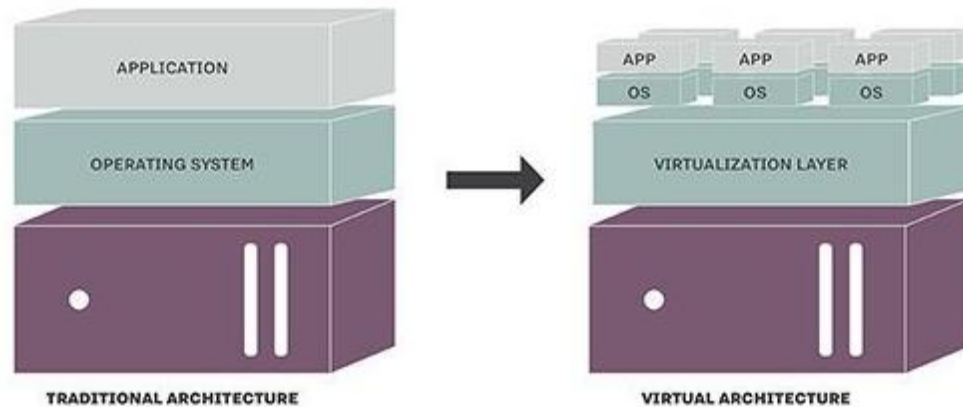


Figure 3: Difference between traditional and virtual architecture

It is commonly hypervisor-based<sup>8</sup>. The hypervisor isolates operating systems and applications from the underlying computer hardware so the host machine can run multiple virtual machines (VM) as guests that share the system's physical compute resources, such as processor cycles, memory space, network bandwidth and so on. (Figure 3)

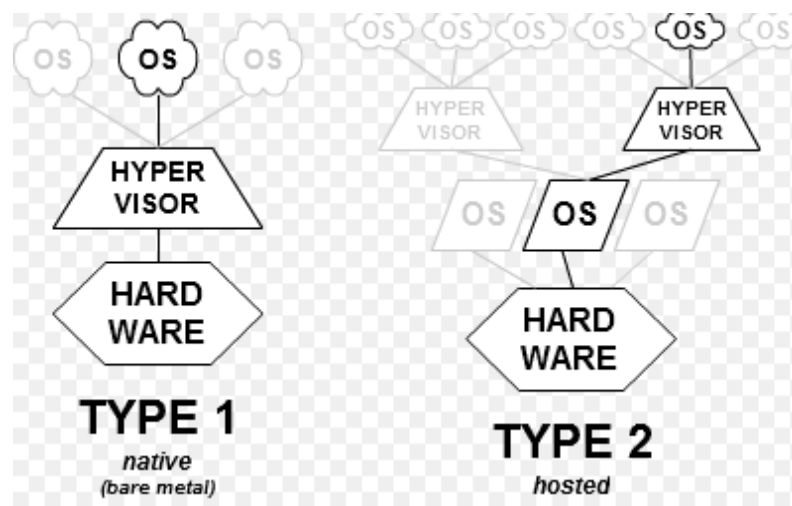


Figure 4: Hypervisor type 1 & type 2

There are two hypervisor types (Figure 4):

**Type 1 hypervisors**, sometimes called bare-metal hypervisors, run directly on top of the host system hardware. Bare-metal hypervisors

<sup>8</sup> <https://whatis.techtarget.com/definition/virtualization-architecture>

offer high availability and resource management. Their direct access to system hardware enables better performance, scalability and stability. Examples of type 1 hypervisors include Microsoft Hyper-V, Citrix XenServer and VMware ESXi.

A **type 2 hypervisor**, also known as a hosted hypervisor, is installed on top of the host operating system, rather than sitting directly on top of the hardware as the type 1 hypervisor does. Each guest OS or VM runs above the hypervisor. The convenience of a known host OS can ease system configuration and management tasks. However, the addition of a host OS layer can potentially limit performance and expose possible OS security flaws. Examples of type 2 hypervisors include VMware Workstation, Virtual PC and Oracle VM VirtualBox.

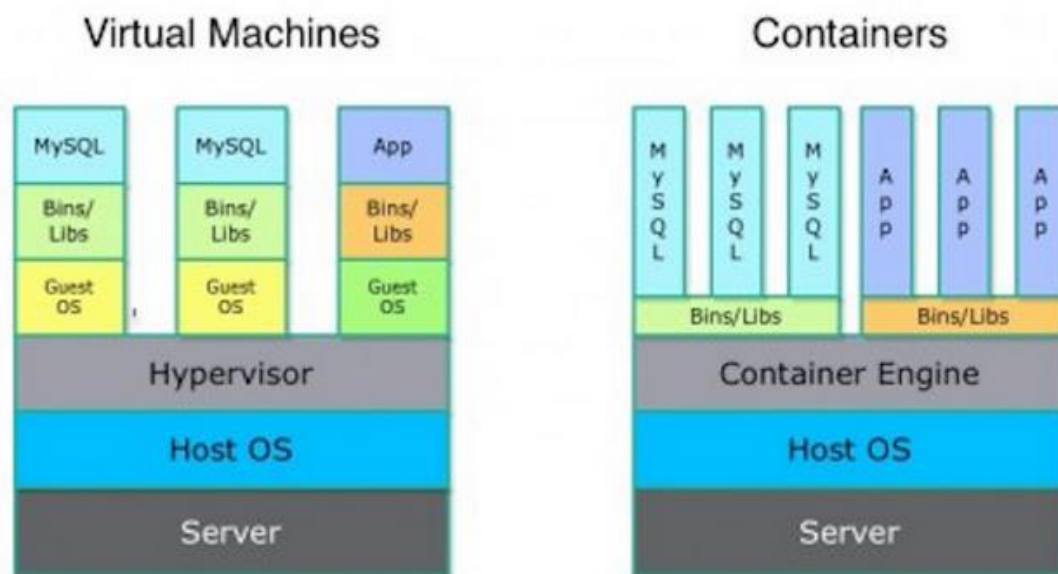


Figure 5: Virtual Machines stuck vs Containers stuck

The main alternative to hypervisor-based virtualization is **containerization** (Figure 5). A container<sup>9</sup> image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system

<sup>9</sup> <https://www.docker.com/what-container>

libraries, settings. Available for both Linux and Windows based apps, containerized software will always run the same, regardless of the environment. Containers isolate software from its surroundings, for example differences between development and staging environments and help reduce conflicts between teams running different software on the same infrastructure.



Figure 6: Docker logo

**Docker** is a computer program that performs operating-system-level virtualization also known as containerization.

Docker is a tool that can package an application and its dependencies in a virtual container that can run on any Linux server. This helps enable flexibility and portability on where the application can run.

## 2.1.6 Openstack



Figure 7: Openstack logo

**OpenStack**<sup>10</sup> is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed through a dashboard that gives administrators control while empowering their users to provision resources through a web interface.

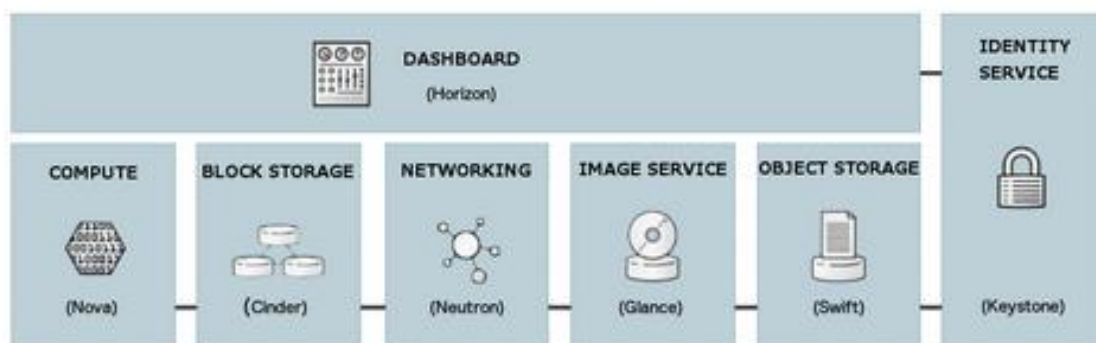


Figure 8: OpenStack embraces a modular architecture to provide a set of core services that facilitates scalability and elasticity as core design tenets.

OpenStack<sup>11</sup> embraces a modular architecture to provide a set of core services that facilitates scalability and elasticity as core design tenets (Figure 8).

OpenStack **Compute service (nova)** provides services to support the management of virtual machine instances at scale, instances that host multi-tiered applications, dev or test environments, “Big Data” crunching Hadoop clusters, or high-performance computing.

<sup>10</sup> <https://www.openstack.org/software/>

<sup>11</sup> <https://docs.openstack.org/security-guide/introduction/introduction-to-openstack.html>

The OpenStack **Object Storage service (swift)** provides support for storing and retrieving arbitrary data in the cloud. The Object Storage service provides both a native API and an Amazon Web Services S3-compatible API. The service provides a high degree of resiliency through data replication and can handle petabytes of data.

The OpenStack **Block Storage service (cinder)** provides persistent block storage for compute instances. The Block Storage service is responsible for managing the life-cycle of block devices, from the creation and attachment of volumes to instances, to their release.

The OpenStack **Networking service (neutron, previously called quantum)** provides various networking services to cloud users (tenants) such as IP address management, DNS, DHCP, load balancing, and security groups (network access rules, like firewall policies). This service provides a framework for software defined networking (SDN) that allows for pluggable integration with various networking solutions.

The OpenStack **Dashboard (horizon)** provides a web-based interface for both cloud administrators and cloud tenants. Using this interface, administrators and tenants can provision, manage, and monitor cloud resources. The dashboard is commonly deployed in a public-facing manner with all the usual security concerns of public web portals.

The OpenStack **Identity service (keystone)** is a shared service that provides authentication and authorization services throughout the entire cloud infrastructure. The Identity service has pluggable support for multiple forms of authentication.

The OpenStack **Image service (glance)** provides disk-image management services, including image discovery, registration, and delivery services to the Compute service, as needed.



### 2.1.7 Fiware<sup>12</sup>



Figure 9: Fiware logo

The **FIWARE** middleware platform provides a rather simple yet powerful set of APIs (Application Programming Interfaces) that ease the development of Smart Applications in multiple vertical sectors. The specifications of these APIs are public and royalty-free. Besides, an open source reference implementation of each of the FIWARE components is publicly available so that multiple FIWARE providers can emerge faster in the market with a low-cost proposition.

FIWARE provides an enhanced OpenStack-based cloud environment plus a rich set of open standard APIs that make it easier to connect to the Internet of Things, process and analyse Big data and real-time media or incorporate advanced features for user interaction.

The **FIWARE Community** is an independent Open Community whose members are committed to materialize the FIWARE mission, that is: “to build an open sustainable ecosystem around public, royalty-free and implementation-driven software platform standards that will ease the development of new Smart Applications in multiple sectors”. The FIWARE Community is not only formed by contributors to the technology (the FIWARE platform) but also those who contribute in building the FIWARE ecosystem and making it sustainable over time. As such, individuals and organizations committing relevant resources in FIWARE Lab activities or activities of the FIWARE Accelerator, FIWARE Mundus or FIWARE iHubs programmes are also considered members of the FIWARE community.

**FIWARE Lab** is a non-commercial sandbox environment where innovation and experimentation based on FIWARE technologies take place. Entrepreneurs and individuals can test the technology as well as

---

<sup>12</sup> <https://www.fiware.org/>

their applications on FIWARE Lab, exploiting Open Data published by cities and other organizations. FIWARE Lab is deployed over a geographically distributed network of federated nodes leveraging on a wide range of experimental infrastructures.

**The FIWARE Catalogue** contains a rich library of components (Generic Enablers) with reference implementations that allow developers to put into effect functionalities such as the connection to the Internet of Things or Big Data analysis, making programming much easier. All of them are public, royalty-free and open source!

**Generic Enablers (GEs)** offer a number of general-purpose functions, offered through well-defined APIs, easing development of smart applications in multiple sectors. They will set the foundations of the architecture associated to our application.

The FIWARE Catalogue includes links to other catalogues bringing information about **domain-specific enablers (DSEs)** to be combined with those serving general purposes (Generic Enablers - GE). They may be helpful for those who plan to develop applications in the domains of energy, creative media, smart manufacturing, health and wellbeing and the agrifood sector.

### 2.1.8 The NGSI Information Model<sup>13</sup>

OMA<sup>14</sup> NGSI defines two interfaces for exchanging information based on the information model. The interface OMA NGSI-10 is used for exchanging information about entities and their attributes, i.e., attribute values and metadata. The interface OMA NGSI-9 is used for availability information about entities and their attributes. Here, instead of exchanging attribute values, information about which provider can provide certain attribute values is exchanged.

The central aspect of the NGSI-9/10 information model is the concept of **entities**. Entities are the virtual representation of all kinds of physical objects in the real world. Examples for physical entities are tables,

---

<sup>13</sup> [https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/NGSI-9/NGSI-10\\_information\\_model](https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/NGSI-9/NGSI-10_information_model)

<sup>14</sup> [https://en.wikipedia.org/wiki/Open\\_Mobile\\_Alliance](https://en.wikipedia.org/wiki/Open_Mobile_Alliance)

rooms, or persons. Virtual entities have an identifier and a type. For example, a virtual entity representing a person named “John” could have the identifier “John” and the type “person”.

Any available information about physical entities is expressed in the form of **attributes** of virtual entities. Attributes have a name and a type as well. For example, the body temperature of John would be represented as an attribute having the name “body\_temperature” and the type “temperature”. Values of such attributes are contained by value containers. This kind of container does not only consist of the actual attribute value, but also contains a set of metadata. Metadata is data about data; in our body temperature example this metadata could represent the time of measurement, the measurement unit, and other information about the attribute value.

There is also a concept of **attribute domains** in OMA NGSI 9/10. An attribute domain logically groups together a set of attributes. For example, the attribute domain "health\_status" could comprise of the attributes "body\_temperature" and "blood\_pressure".

The data structure used for exchanging information about entities is **context element**. A context element contains information about multiple attributes of one entity. The domain of these attributes can also be specified inside the context element; in this case all provided attribute values have to belong to that domain.

Formally, a context element contains the following information

- an entity id and type
- a list of triplets <attribute name, attribute type, attribute value> holding information about attributes of the entity
- (optionally) the name of an attribute domain
- (optionally) a list of triplets <metadata name, metadata type, metadata value> that apply to all attribute values of the given domain

### 2.1.9 IDAS GE<sup>15</sup>

The IDAS component is an implementation of the Backend Device Management GE, according to the FIWARE reference architecture. We need this component if we plan to connect IoT devices/gateways to FIWARE-based ecosystems.

IoT Agents translate IoT-specific protocols into the NGSI context information protocol, that is the FIWARE standard data exchange model. We do not need this component if our devices or gateways natively support the NGSI API.

By using an IoT Agent , our devices will be represented in a FIWARE platform as NGSI entities in a ContextBroker. This means that we can query or subscribe to changes of device parameters status by querying or subscribing to the corresponding NGSI entity attributes at the ContextBroker.

Additionally, we may trigger commands to our actuation devices just by updating specific command-related attributes in their NGSI entities representation at the Context Broker. This way, all developers interactions with devices are handled at a Context Broker, providing an homogeneous API and interface as for all other non-IoT data in a FIWARE ecosystem.

Currently there are four supported IoT Agents by Fiware which have been implemented with node.js:

- IoTAgent-JSON 1.6.2 (HTTP/MQTT transport):

This IoT Agent is designed to be a bridge between an HTTP/MQTT+JSON based protocol and the FIWARE NGSI standard used in FIWARE, like the Orion Context Broker.

- IoTAgent-LWM2M 0.4.0 (CoaP transport)

This IoT Agent is designed to be a bridge between a Lightweight M2M protocol and the FIWARE NGSI standard.

---

<sup>15</sup> <https://catalogue-server.fiware.org/enablers/backend-device-management-idas>

➤ IoTAgent-UL 1.5.2 (HTTP/MQTT transport)

This IoT Agent is designed to be a bridge between an UltraLight2.0 protocol and the FIWARE NGSI standard.

➤ IoTAgent-node-lib 2.5.1

This repository does not belong to an executable agent, but it is a library to create new agents. This core library allows developing new agents for specific southbound protocols/standards/messages.

### 2.1.10 Orion Context Broker GE<sup>16</sup>

Orion Context Broker is able to mediate between consumer producers (e.g. sensors) and every context consumer application. It is an implementation of the Publish/Subscribe Context Broker GE, providing the NGSI9 and NGSI10 interfaces. Using these interfaces, clients can do several operations:

- Register context producer applications, e.g. a temperature sensor within a room
- Update context information, e.g. send updates of temperature
- Being notified when changes on context information take place (e.g. the temperature has changed) or with a given frequency (e.g. get the temperature each minute)
- Query context information. The Orion Context Broker stores context information updated from applications, so queries are resolved based on that information.

Orion is a C++ implementation of the NGSIv2 REST API binding developed as a part of the FIWARE platform.

### 2.1.11 Keyrock Identity Manager GE<sup>17</sup>

Identity Management covers a number of aspects involving users' access to networks, services and applications, including secure and private authentication from users to devices, networks and services, authorization & trust management, user profile management, privacy-

---

<sup>16</sup> <https://catalogue-server.fiware.org/enablers/publishsubscribe-context-broker-orion-context-broker>

<sup>17</sup> <https://catalogue-server.fiware.org/enablers/identity-management-keyrock>

preserving disposition of personal data, Single Sign-On (SSO) to service domains and Identity Federation towards applications. The Identity Manager is the central component that provides a bridge between IdM systems at connectivity-level and application-level. Furthermore, Identity Management is used for authorizing foreign services to access personal data stored in a secure environment. Hereby usually the owner of the data must give consent to access the data.

## 2.2 Internet of Things (IoT)

The **Internet of Things (IoT)**<sup>18</sup> is the network of physical devices, vehicles, home appliances and other items embedded with electronics, software, sensors, actuators, and connectivity which enables these objects to connect and exchange data. Each thing is uniquely identifiable through its embedded computing system but is able to inter-operate within the existing Internet infrastructure.

### 2.2.1 Devices

**IoT devices**, or any of the many things in the internet of things, are nonstandard computing devices that connect wirelessly to a network and have the ability to transmit data.

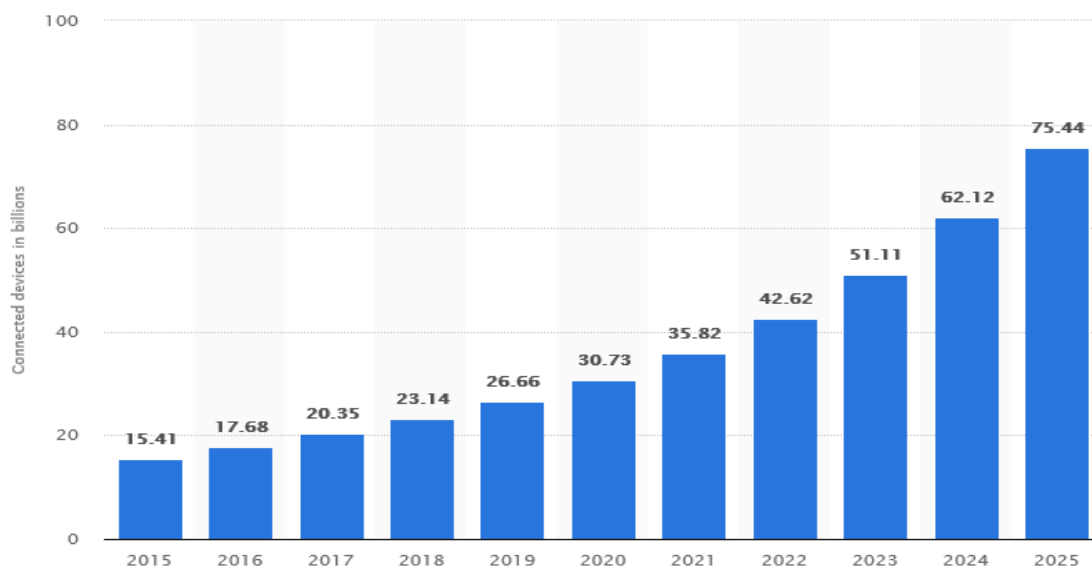


Figure 10: Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions)

<sup>18</sup> [https://en.wikipedia.org/wiki/Internet\\_of\\_things](https://en.wikipedia.org/wiki/Internet_of_things)

This statistic<sup>19</sup> (Figure 10) shows the number of connected devices (Internet of Things; IoT) worldwide from 2015 to 2025. For 2020, the installed base of Internet of Things devices is forecast to grow to almost 31 billion worldwide. The overall Internet of Things market is projected to be worth more than one billion U.S. dollars annually from 2017 onwards.

So we could understand how crucial the role of Cloud Computing is in order to deal with this rapid increase.

IoT devices have some fundamental characteristics. First of all, **everything communicates** which means that smart things have the ability to wirelessly communicate among themselves, and form adhoc networks of interconnected objects. In addition, **everything is identified**. Each thing has a unique identifier (e.g., IP address if IPv4 – 32bit address space, but due to limited address space IoT will have to use IPv6 - 128 bit address space). Finally, **everything interacts** which means that smart things can interact with their environment through sensing and actuation capabilities.

IoT devices may be sensors, actuators, microcontrollers or shields.

**Sensors** are devices that detect and respond to some type of input from the environment (temperature, motion, humidity, pressure etc.)

**Actuators** are systems which convert electrical signals to physical actions (for interacting with environment).

**Microcontrollers** are small computers on a single board containing processor, memory and i/o peripherals. They are embedded with low power consumption and small size.

**IoT Node shields** are integrated solutions on a board for secure connectivity (e.g., AES encryption) along with programmable board (m2m).

---

<sup>19</sup> <https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/>

### 2.2.2 IoT protocols

As we have already mentioned, IoT devices communicate among themselves, but in order to succeed this task they have to use a non-ip communication protocol. There are several non-ip protocols which have both advantages and disadvantages relatively with the domain of their usage. The most popular IoT **non-ip protocols** are presented below:

**Radio-frequency identification (RFID)**<sup>20</sup> uses electromagnetic fields to automatically identify and track tags attached to objects. The tags contain electronically-stored information. Passive tags collect energy from a nearby RFID reader's interrogating radio waves. Active tags have a local power source (such as a battery) and may operate hundreds of meters from the RFID reader. Unlike a barcode, the tag need not be within the line of sight of the reader, so it may be embedded in the tracked object.

**Near-field communication (NFC)**<sup>21</sup> is a set of communication protocols that enable two electronic devices, one of which is usually a portable device such as a smartphone, to establish communication by bringing them within 4 cm (1.6 in) of each other. NFC devices are used in contactless payment systems, similar to those used in credit cards and electronic ticket smartcards and allow mobile payment to replace/supplement these systems. This is sometimes referred to as NFC/CTLS (Contactless) or CTLS NFC. NFC is used for social networking, for sharing contacts, photos, videos or files. NFC-enabled devices can act as electronic identity documents and keycards. NFC offers a low-speed connection with simple setup that can be used to bootstrap more capable wireless connections.

**Zigbee**<sup>22</sup> is an IEEE 802.15.4-based specification for a suite of high-level communication protocols used to create personal area networks with small, low-power digital radios, such as for home automation, medical device data collection, and other low-power low-bandwidth needs, designed for small scale projects which need wireless connection.

---

<sup>20</sup> [https://en.wikipedia.org/wiki/Radio-frequency\\_identification](https://en.wikipedia.org/wiki/Radio-frequency_identification)

<sup>21</sup> [https://en.wikipedia.org/wiki/Near-field\\_communication](https://en.wikipedia.org/wiki/Near-field_communication)

<sup>22</sup> <https://en.wikipedia.org/wiki/Zigbee>



Hence, Zigbee is a low-power, low data rate, and close proximity (i.e., personal area) wireless ad hoc network. Its low power consumption limits transmission distances to 10–100 meters line-of-sight, depending on power output and environmental characteristics.[1] Zigbee devices can transmit data over long distances by passing data through a mesh network of intermediate devices to reach more distant ones. Zigbee is typically used in low data rate applications that require long battery life and secure networking (Zigbee networks are secured by 128 bit symmetric encryption keys.) Zigbee has a defined rate of 250 kbit/s, best suited for intermittent data transmissions from a sensor or input device.

**Bluetooth**<sup>23</sup> is a wireless technology standard for exchanging data over short distances (using short-wavelength UHF radio waves in the ISM band from 2.4 to 2.485 GHz) from fixed and mobile devices, and building personal area networks (PANs).

**Bluetooth Low Energy**<sup>24</sup> (Bluetooth LE, colloquially BLE, formerly marketed as Bluetooth Smart) is a wireless personal area network technology designed and marketed by the Bluetooth Special Interest Group (Bluetooth SIG) aimed at novel applications in the healthcare, fitness, beacons, security, and home entertainment industries. Compared to Classic Bluetooth, Bluetooth Low Energy is intended to provide considerably reduced power consumption and cost while maintaining a similar communication range.

**Wi-Fi or WiFi**<sup>25</sup> is a technology for wireless local area networking with devices based on the IEEE 802.11 standards. Devices that can use Wi-Fi technology include personal computers, video-game consoles, smartphones and tablets, digital cameras, smart TVs, digital audio players and modern printers. Wi-Fi compatible devices can connect to the Internet via a WLAN and a wireless access point. Such an access point (or hotspot) has a range of about 20 meters (66 feet) indoors and a greater range outdoors. Hotspot coverage can be as small as a single room with walls that block radio waves, or as large as many square

---

<sup>23</sup> <https://en.wikipedia.org/wiki/Bluetooth>

<sup>24</sup> [https://en.wikipedia.org/wiki/Bluetooth\\_Low\\_Energy](https://en.wikipedia.org/wiki/Bluetooth_Low_Energy)

<sup>25</sup> <https://en.wikipedia.org/wiki/Wi-Fi>

kilometres achieved by using multiple overlapping access points. Wi-Fi most commonly uses the 2.4 gigahertz (12 cm) UHF and 5.8 gigahertz (5 cm) SHF ISM radio bands. Anyone within range with a wireless modem can attempt to access the network; because of this, Wi-Fi is more vulnerable to attack (called eavesdropping) than wired networks. Wi-Fi Protected Access is a family of technologies created to protect information moving across Wi-Fi networks and includes solutions for personal and enterprise networks. Security features of Wi-Fi Protected Access constantly evolve to include stronger protections and new security practices as the security landscape changes.

**DASH7**<sup>26</sup> Alliance Protocol (D7A) is an open source Wireless Sensor and Actuator Network protocol, which operates in the 433 MHz, 868 MHz and 915 MHz unlicensed ISM band/SRD band. DASH7 provides multi-year battery life, range of up to 2 km, low latency for connecting with moving things, a very small open source protocol stack, AES 128-bit shared key encryption support, and data transfer of up to 167 kbit/s.

**LoRa**<sup>27</sup> is a patented wireless data communication technology developed by Cycleo of Grenoble, France, and acquired by Semtech in 2012. LoRa uses license-free sub-gigahertz radio frequency bands like 169 MHz, 433 MHz, 868 MHz (Europe) and 915 MHz (North America). It enables very-long-range transmissions (more than 10 km in rural areas) with low power consumption. The technology is presented in two parts — LoRa, the physical layer and LoRaWAN, the upper layers. LoRa is described analytically in section 2.3

On the other hand, **ip protocols** are needed in order to collect data from the IoT and interconnect it on the Cloud or other computing systems. Some of the main ip protocols are presented below:

**The Transmission Control Protocol (TCP)**<sup>28</sup> is one of the main protocols of the Internet protocol suite. It belongs to the Transport Layer of The Open Systems Interconnection model (OSI model). TCP originated in the initial network implementation in which it complemented the Internet

---

<sup>26</sup> <https://en.wikipedia.org/wiki/DASH7>

<sup>27</sup> <https://en.wikipedia.org/wiki/LoRa>

<sup>28</sup> [https://el.wikipedia.org/wiki/Transmission\\_Control\\_Protocol](https://el.wikipedia.org/wiki/Transmission_Control_Protocol)

Protocol (IP). Therefore, the entire suite is commonly referred to as TCP/IP. TCP provides reliable, ordered, and error-checked delivery of a stream of octets (bytes) between applications running on hosts communicating via an IP network. Major Internet applications such as the World Wide Web, email, remote administration, and file transfer rely on TCP.

Applications that do not require reliable data stream service may use the **User Datagram Protocol (UDP)**<sup>29</sup>, which also belongs to Transport Layer of OSI and provides a connectionless datagram service that emphasizes reduced latency over reliability. UDP is suitable for purposes where error checking and correction are either not necessary or are performed in the application. It avoids the overhead of such processing in the protocol stack. Time-sensitive applications often use UDP because dropping packets is preferable to waiting for packets delayed due to retransmission, which may not be an option in a real-time system.

**The Hypertext Transfer Protocol (HTTP)**<sup>30</sup> is an application protocol for distributed, collaborative, and hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web. Hypertext is structured text that uses logical links (hyperlinks) between nodes containing text. HTTP is the protocol to exchange or transfer hypertext. HTTP is not ideal for many of the special IoT needs. For example, it is unsuitable for emitting information from one to many, listening for events whenever may happen and pushing information over unreliable networks. Also, HTTP is slow (not ideal for real-time processing), uses more battery and is less reliable.

**MQTT (Message Queuing Telemetry Transport)**<sup>31</sup> is an ISO standard (ISO/IEC PRF 20922) publish-subscribe-based messaging protocol. It works on top of the TCP/IP protocol. It is designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited. It is ideal for long battery-life of devices, fast responses, one to many communication due to the

---

<sup>29</sup> [https://en.wikipedia.org/wiki/User\\_Datagram\\_Protocol](https://en.wikipedia.org/wiki/User_Datagram_Protocol)

<sup>30</sup> [https://en.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol)

<sup>31</sup> <https://en.wikipedia.org/wiki/MQTT>

publish/subscribe mechanism and reliable data transmissions. The publish-subscribe messaging pattern requires a message broker. From the broker we can monitor and handle the routing of the mqtt packets. Relying on TCP, MQTT allows also using TLS (Transport Layer Security) in order to encrypt the data and have a secure communication.

**Constrained Application Protocol (CoAP)**<sup>32</sup> is a specialized Internet Application Protocol for constrained devices. It enables those constrained devices called "nodes" to communicate with the wider Internet using similar protocols. CoAP is designed for use between devices on the same constrained network (e.g., low-power, lossy networks), between devices and general nodes on the Internet, and between devices on different constrained networks both joined by an internet. This protocol is also being used via other mechanisms, such as SMS on mobile communication networks. Essentially, it is a service layer protocol that is intended for use in resource-constrained internet devices, such as wireless sensor network nodes. It is designed to easily translate to HTTP for simplified integration with the web, while also meeting specialized requirements such as multicast support, very low overhead, and simplicity. Multicast, low overhead, and simplicity are extremely important for Internet of Things (IoT) and Machine-to-Machine (M2M) devices, which tend to be deeply embedded and have much less memory and power supply than traditional internet devices have. Therefore, efficiency is very important. CoAP can run on most devices that support UDP or a UDP analogue.

### 2.2.3 IoT platforms

IoT platforms are the support software that connects everything in an IoT system. An IoT platform facilitates communication, data flow, device management, and the functionality of applications. Essentially, an IoT platform helps to connect new hardware and handle different communication protocols. Furthermore, it helps to provide security and authentication for devices and users and to collect, visualize and analyze data. Finally, it gives the ability to be integrated with other web services.

---

<sup>32</sup> [https://en.wikipedia.org/wiki/Constrained\\_Application\\_Protocol](https://en.wikipedia.org/wiki/Constrained_Application_Protocol)

## 2.3 LoRa Technology<sup>33</sup>



Figure 11: LoRa logo

As we have already mentioned, LoRa technology constitutes a Semtech innovation which is easy to plug into the existing infrastructure and offers a solution to serve battery-operated IoT applications. Semtech builds LoRa technology into its chipsets. These chipsets are then built into the products offered by a vast network of IoT partners and integrated into LPWANs from mobile network operators worldwide.

LoRa is the physical layer or the wireless modulation utilized to create the long range communication link. Many legacy wireless systems use frequency shifting keying (FSK) modulation as the physical layer because it is a very efficient modulation for achieving low power. LoRa is based on chirp spread spectrum modulation, which maintains the same low power characteristics as FSK modulation but significantly increases the communication range. Chirp spread spectrum has been used in military and space communication for decades due to the long communication distances that can be achieved and robustness to interference, but LoRa is the first low cost implementation for commercial usage.

One technology cannot serve all of the projected applications and volumes for IoT. WiFi and BLE are widely adopted standards and serve

<sup>33</sup> <https://loro-alliance.org/about-lorawan>

the applications related to communicating personal devices quite well. Cellular technology is a great fit for applications that need high data throughput and have a power source. LPWAN offers multi-year battery lifetime and is designed for sensors and applications that need to send small amounts of data over long distances a few times per hour from varying environments

### 2.3.1 Characteristics of LoRa Technology<sup>34</sup>

LoRa technology has some fundamental characteristics. First of all, it enables GPS-free (**geolocation**), which gives the opportunity to create low power tracking applications. In addition, infrastructure investment, operating expenses and end-node sensors are at a **low cost** in contrast with other technologies. Furthermore, LoRa technology is **standardized**, which means that improved global interoperability speeds adoption and roll out of LoRaWAN-based networks and IoT applications. Speaking about LoRa protocol, it is designed specifically for **low power** consumption extending battery lifetime up to 20 years. On the other hand, single base station provides deep penetration in dense urban/indoor regions and it also connects rural areas up to 30 miles(48 km) away (**long range**). Finally, LoRa provides **secure** communication, as AES128 encryption is embedded into the end-to-end nodes, and it supports millions of messages per base station, ideal for public network operators serving many customers (**high capacity**).

### 2.3.2 LoRaWAN protocol

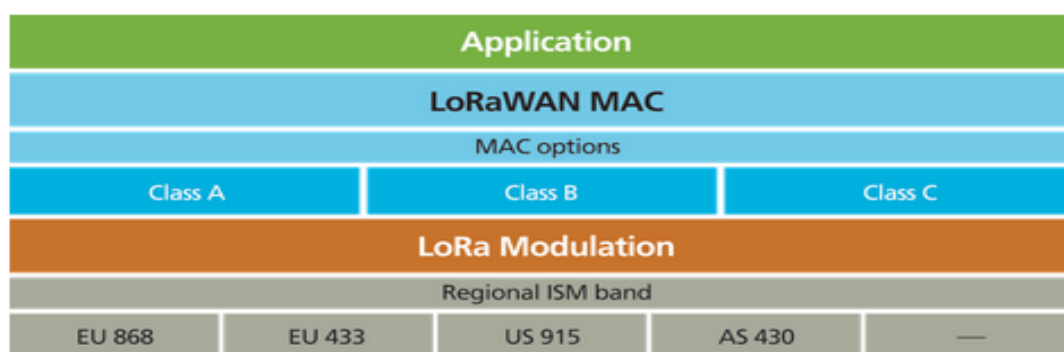


Figure 12: LoRa Stack

<sup>34</sup> <https://www.semtech.com/technology/lora/what-is-lora>

LoRaWAN is a protocol specification built on top of the LoRa technology developed by the LoRa Alliance (Figure 12). It uses unlicensed radio spectrum in the Industrial, Scientific and Medical (ISM) bands to enable low power, wide area communication between remote sensors and gateways connected to the network. This standards-based approach to building a LPWAN allows for quick set up of public or private IoT networks anywhere using hardware and software that is bi-directionally secure, interoperable and mobile, provides accurate localization, and works the way you expect.

LoRaWAN™ defines the communication protocol and system architecture for the network while the LoRa® physical layer enables the long-range communication link. The protocol and network architecture have the most influence in determining the battery lifetime of a node, the network capacity, the quality of service, the security, and the variety of applications served by the network.

### **2.3.3 Classes of LoRa devices**

End-devices serve different applications and have different requirements. In order to optimize a variety of end application profiles, LoRaWAN™ utilizes different device classes. The device classes trade off network downlink communication latency versus battery lifetime (Figure 13). In a control or actuator-type application, the downlink communication latency is an important factor.

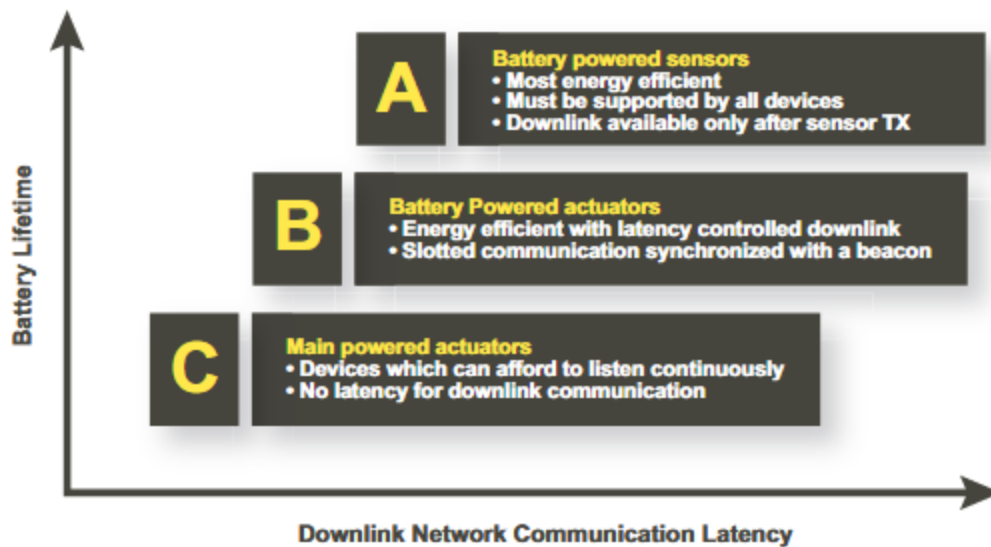


Figure 13: LoRa Class of Device- Downlink Network Communication Latency in comparison with battery lifetime

**Bi-directional end-devices (Class A):** End-devices of Class A allow for bi-directional communications whereby each end-device's uplink transmission is followed by two short downlink receive windows. The transmission slot scheduled by the end-device is based on its own communication needs with a small variation based on a random time basis (ALOHA-type of protocol). This Class A operation is the lowest power end-device system for applications that only require downlink communication from the server shortly after the end-device has sent an uplink transmission. Downlink communications from the server at any other time will have to wait until the next scheduled uplink.

**Bi-directional end-devices with scheduled receive slots (Class B):** In addition to the Class A random receive windows, Class B devices open extra receive windows at scheduled times. In order for the end-device to open its receive window at the scheduled time, it receives a time-synchronized beacon from the gateway. This allows the server to know when the end-device is listening.

**Bi-directional end-devices with maximal receive slots (Class C):** End-devices of Class C have almost continuously open receive windows, only closed when transmitting. However, this class of devices reduce the battery lifetime.



In addition to the class A structure of uplink followed by two downlink windows, class C further reduces latency on the downlink by keeping the receiver of the end-device open at all times that the device is not transmitting (half duplex). Based on this, the network server can initiate a downlink transmission at any time on the assumption that the end-device receiver is open, so no latency. The compromise is the power drain of the receiver (up to ~50mW) and so class C is suitable for applications where continuous power is available. For battery powered devices, temporary mode switching between classes A & C is possible, and is useful for intermittent tasks such as firmware over-the-air updates.

### 2.3.4 LoRa Data Rates

In addition to frequency hopping, all communication packets between end-devices and gateways also include a variable 'Data rate' (DR) setting. The selection of the DR allows a dynamic trade-off between communication range and message duration. Also, due to the spread spectrum technology, communications with different DRs do not interfere with each other and create a set of virtual 'code' channels increasing the capacity of the gateway. To maximize both battery life of the end-devices and overall network capacity, the LoRaWAN network server manages the DR setting and RF output power for each end-device individually by means of an Adaptive Data Rate (ADR) scheme. LoRaWAN baud rates range from 0.3 kbps to 50 kbps.

### 2.3.5 Security

Security is a primary concern for any mass IoT deployment and the LoRaWAN specification defines two layers of cryptography:

A unique 128-bit **Network Session Key (NewSKey)** shared between the end-device and network server. It is used for interaction between the Node and the Network. This key is used to check the validity of messages (Message Integrity Code-MIC).

A unique 128-bit **Application Session Key (AppSKey)** shared end-to-end at the application level. It is used for encryption and decryption of the payload. The payload is fully encrypted between the Node and the

Network Server which means that nobody in the middle of the communication is able to read the contents of the messages which are sent or received.

AES algorithms are used to provide authentication and integrity of packets to the network server and end-to-end encryption to the application server. By providing these two levels, it becomes possible to implement 'multi-tenant' shared networks without the network operator having visibility of the users payload data.

To participate in a LoRaWAN network, each end-device has to be personalized and activated. Activation of an end-device can be achieved in two ways, either via **Over-The-Air Activation (OTAA)** when an end-device is deployed or reset, or via **Activation By Personalization (ABP)** in which the two steps of end-device personalization and activation are done as one step.

After Activation By Personalisation(ABP) the following information is stored in the end-device:

- DevAddr: Consists of 32 bits and identifies the end-device within the current network
- AppEUI: Is a global application ID in IEEE EUI64 address space that uniquely identifies the entity able to process the JoinReq frame
- NwkSKey
- AppSKey

Under certain circumstances, end-devices can be activated by personalization. Activation by personalization directly ties an end-device to a specific network. Activating an end-device by personalization means that the DevAddr and the two session keys NwkSKey and AppSKey are directly stored into the end-device. The end-device is equipped with the required information for participating in a specific LoRa network when started. Each device should have a unique set of NwkSKey and AppSKey.

For over-the-air activation, end-devices must follow a join procedure prior to participating in data exchanges with the network server. An end-

device has to go through a new join procedure every time it has lost the session context information (the information that server sends into the join accept message after the join request by the device). The join procedure requires the end-device to be personalized with the following information (be stored into the device) before it starts the join procedure:

- DevEUI: Is a global end-device ID in IEEE EUI64 address space that uniquely identifies the end-device
- AppEUI: Is a global application ID in IEEE EUI64 address space that uniquely identifies the entity able to process the JoinReq frame
- AppKey: is an AES-128 root key specific to the end-device. Whenever an end-device joins a network via over-the-air activation, the AppKey is used to derive the session keys NwkSKey and AppSKey specific for that end-device to encrypt and verify network communication and application data.

For over-the-air-activation, end-devices are not personalized with any kind of network key. Instead, whenever an end-device joins a network, a network session key specific for that end-device is derived to encrypt and verify transmissions at the network level. This way, roaming of end-devices between networks of different providers is facilitated. Using both a network session key and an application session key further allows federated network servers in which application data cannot be read or tampered by the network provider.

## 2.3.6 LoRa Network Architecture

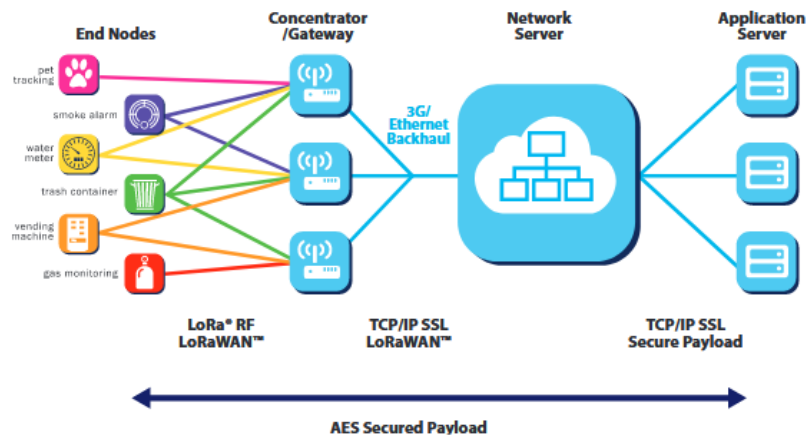


Figure 14: LoRa Network Architecture

LoRaWAN network architecture (Figure 14) is deployed in a star-of-stars topology in which gateways relay messages between end-devices and a central network server. The gateways are connected to the network server via standard IP connections and act as a transparent bridge, simply converting RF packets to IP packets and vice versa. The wireless communication takes advantage of the Long Range characteristics of the LoRa physical layer, allowing a single-hop link (network coverage area is equal with the radio range of a single node) between the end-device and one or many gateways. All nodes are capable of bi-directional communication, and there is support for multicast addressing groups to make efficient use of spectrum during tasks such as Firmware Over-The-Air (FOTA) upgrades or other mass distribution messages.

The intelligence and complexity is pushed to the network server, which manages the network and will filter redundant received packets, perform security checks, schedule acknowledgments through the optimal gateway, and perform adaptive data rate, etc.

## 3. LoRaWare Reference Architecture

The Reference Architecture (RA) (Figure 15) is a generic high-level conceptual model which highlights the LoRaWare Ecosystem consisting of three interconnected parts namely (a) Sensing platform implemented as network of Things connected to the cloud through an internet

Backhaul, (b) the Back-End Cloud implementing a Network Server to support connectivity with the Sensing Platform, a Web Proxy, and a context server providing application-specific services (context services) and (c) the consumer (end-user) applications which connect with the cloud to receive services and through it connect to the sensing platform and the end-users (e.g., individuals who receive coaching instructions). Each consumer subscribes to particular services and IoT devices (through “Publish/Subscribe” context services running on the cloud) that publish information from the sensing environment (e.g., activity, health information).

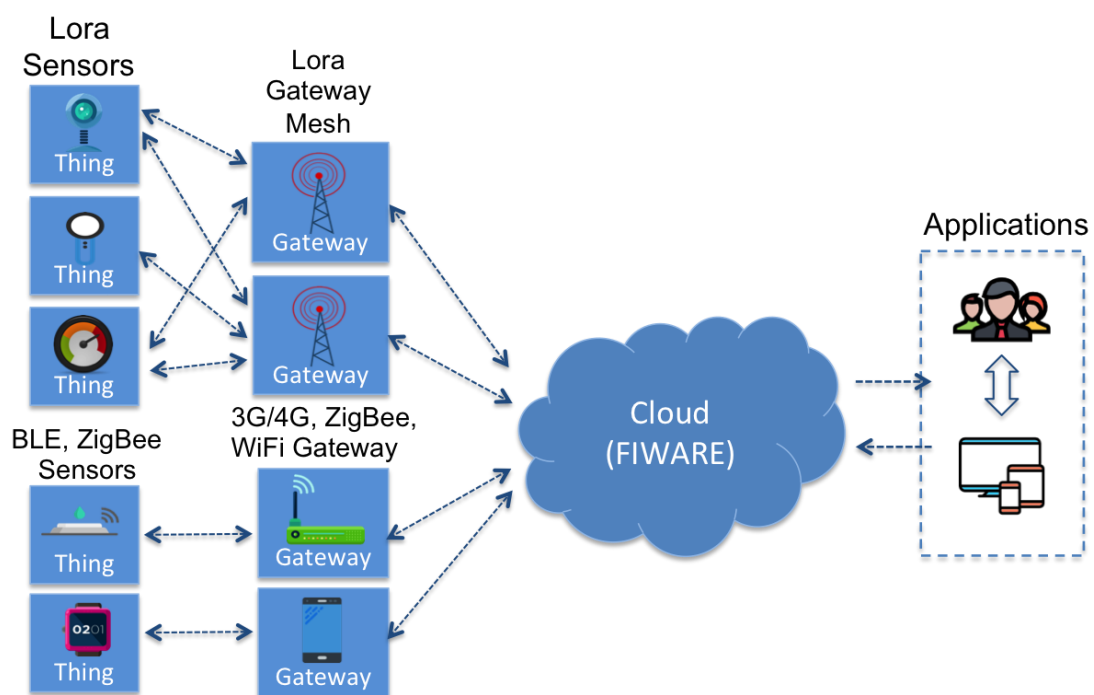


Figure 15: LoRaWare Reference Architecture

### 3.1 LoRa Network

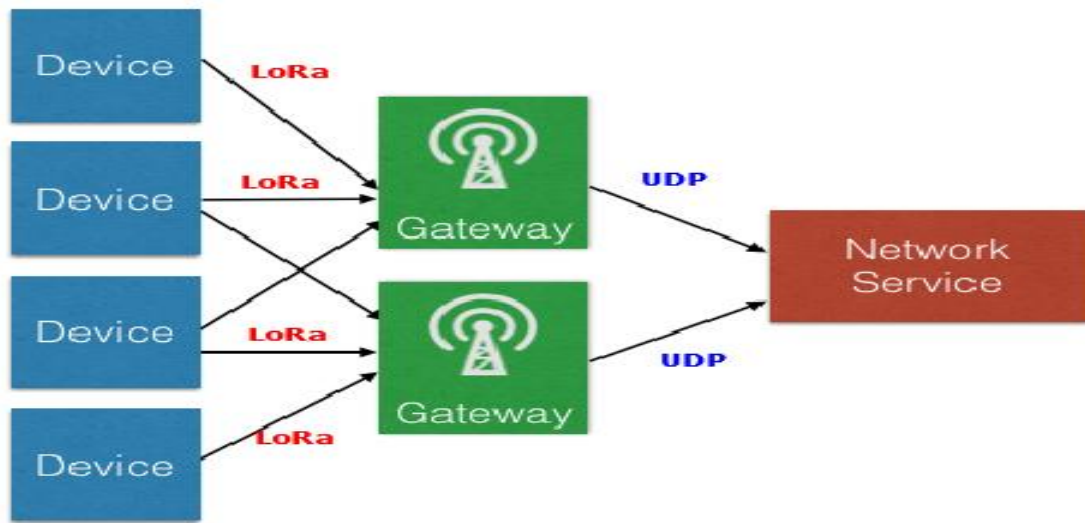


Figure 16: LoRaWare Network (Front end)

The LoRa Network (Figure 16) consists of Lora devices and LoRa Gateways in a star-of-stars topology. Devices transmit RF packets with LoRa modulation which are captured by one or many Lora gateways (uplink transmission).

LoRa gateways convert the LoRa packets to an IP protocol in order to be transmitted over the Internet to the Network Server (Cloud). This operation is done using the packet forwarder, a semtech's open source software, which is running on every gateway. Furthermore, LoRa gateways receive downlink messages (from the Network server to the gateway) in order to convert them from the IP protocol to LoRa and transmit it wirelessly to a specific LoRa device.

The most common IP protocol which is used by the semtech's packet forwarder for the communication between the LoRa Gateways and the Network Server is the UDP protocol. However, we can choose other protocols for this purpose like MQTT over TCP for secure one-to-many communication, easier handling and monitoring of the packets sent or received by the Gateways and Network Server.

## 3.2 LoRa Backend

The LoRa Backend (Figure 17) consists of the Fiware Infrastructure which accommodates all the Cloud Services of the LoRaWare Architecture. In this section we describe the functionality of every service and how it is connected with the other services.

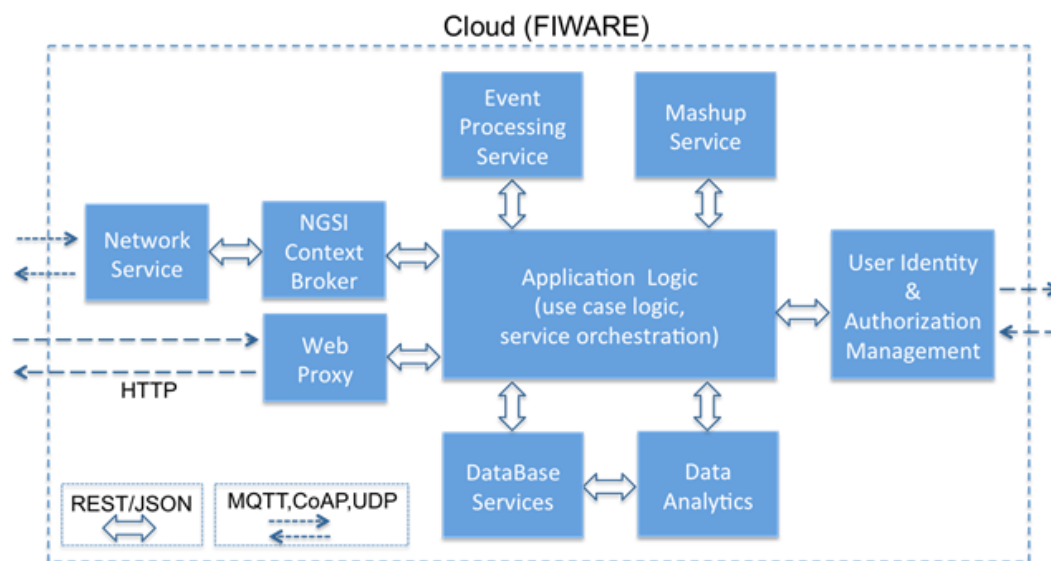


Figure 17: LoRaWare Service Oriented Architecture (Back end)

### ➤ Network Service

The intelligence and complexity of the LoRa sensing network is pushed to the network server on the FIWARE cloud, which manages the network, filters redundant received packets, performs security checks, schedules acknowledgments through the optimal gateway, perform adaptive data rate, etc. The Network Server can be viewed as an enhanced version of IDAS - Device Management GE of FIWARE that is used to assist connection of IoT devices to a FIWARE cloud platform. More specifically, we choose to convert into the gateway the LoRa to MQTT, as it provides some benefits like easy monitoring and routing of the packets, secure and one-to-many communication and easy development of mqtt clients. In addition, there is one IoT Agent of IDAS which supports the MQTT protocol so it is very simple to convert the LoRa to MQTT and the MQTT to the NGSI format in order to publish data

into the NGSI Context Broker from the Network Service. It maps the requests coming from IoT devices to NGSI entities.

The Network Service transforms data to JSON format and then with the suitable asynchronous calls, they are forwarded to an NGSI Compliant Content Broker (i.e., the Publish Subscribe Context Broker GE of FIWARE), Web Proxy and other cloud services.

#### ➤ **Web Proxy**

IoT of today comprises a collection of isolated Internet of Things that can't really interact with each other, nor can they be searched and discovered on the Web and used by applications. Although lightweight Web servers can be embedded in small devices and enable such functionality, they feature limited resources and the solution is not optimal for battery life time, sensor autonomy and cost). The Web Proxy keeps the virtual image of each device or sensor (their descriptions and services) so that Things become part of the Web just like Web sites: they can be published, consumed, aggregated, filtered and searched for by humans and applications. Supporting this functionality in LoRaWare will lead to higher degree of interoperability with other systems as other applications can search and discover Things on the LoRaWare platform to connect to.

#### ➤ **NGSI Context Broker**

The NGSI Context Broker is a Publish/Subscribe service for managing device subscriptions and user subscriptions to data and sensors. It mediates between devices (producers) and applications (consumers).

#### ➤ **Database services**

Database services are used to permanently store information about user (administrator and consumers), sensors and sensor data (history data) and user subscription history. It is implemented as a relational (e.g. MySQL) or NoSQL database (e.g., Casandra, MongoDB).

#### ➤ **Data Analytics**



Data Analytics Service will demonstrate functionality related to uncovering hidden patterns in data, unknown correlations, user preferences and useful business information (e.g. user's data may provide feedback for enhancing system functionality and users acceptance). The project will utilize the COSMOS big data analysis GE or the Data Visualization - SpagoBI GE of FIWARE.

#### ➤ **Event Processing Service**

The Event Processing module handles events (e.g. creates alarm notifications based on end-user conditions and information received from the sensors) and notifies the Publish/Subscribe service, which is responsible for passing this information to the end-users. The Complex Event Processing (CEP) GE of FIWARE is a reference implementation of this service.

#### ➤ **User Identity & Authorization Management**

The cloud platform, provides also mechanisms for user Identity and Authorization Management supporting access control based on user roles and access policies on services and data using FIWARE Keyrock Identify management GE providing Single Sign On (SSO) service of users to services in conjunction with Authorization PDP GE or PEP Proxy GE that manage user permissions and access policies to resources (all services above are protected by an OAuth2.0 mechanism ).

#### ➤ **Mashup Service**

The Mashup Service allows application developers to compose new applications. This will not only take significantly less time to build an application, but also minimize the effort required to maintain the system each time a device or service is added, removed, or updated. Using services such as IFTTT or Node-RED , devices can be integrated with modern Web applications and services with minimal effort (physical mashups). A Mashup Editor with similar capabilities is offered in FIWARE (WireCloud Mashup GE ).

#### ➤ **Application Logic**

The Application Logic implements application or use-case specific (business) services and orchestrate the transferring of the information to the appropriate individual services (storage, identification and information manager). It implements application intelligence for handling context events (e.g., a rule based system) or decides whether the consumers must handle these events

## **4.Implementation of LoRaWare Architecture**

In this section we describe the implementation of the architecture, the operations and the components of the system.

More specifically, we present the architecture of the system as a whole and for ease of presentation we discuss independently the functionality of each component. Then, we describe the sensors we used, the gateways, the services that run on the gateways and on the Cloud. To show proof of concept, at the end of this section we present an application running on LoRaWare.

## 4.1 Architectural Diagrams of LoRaWare

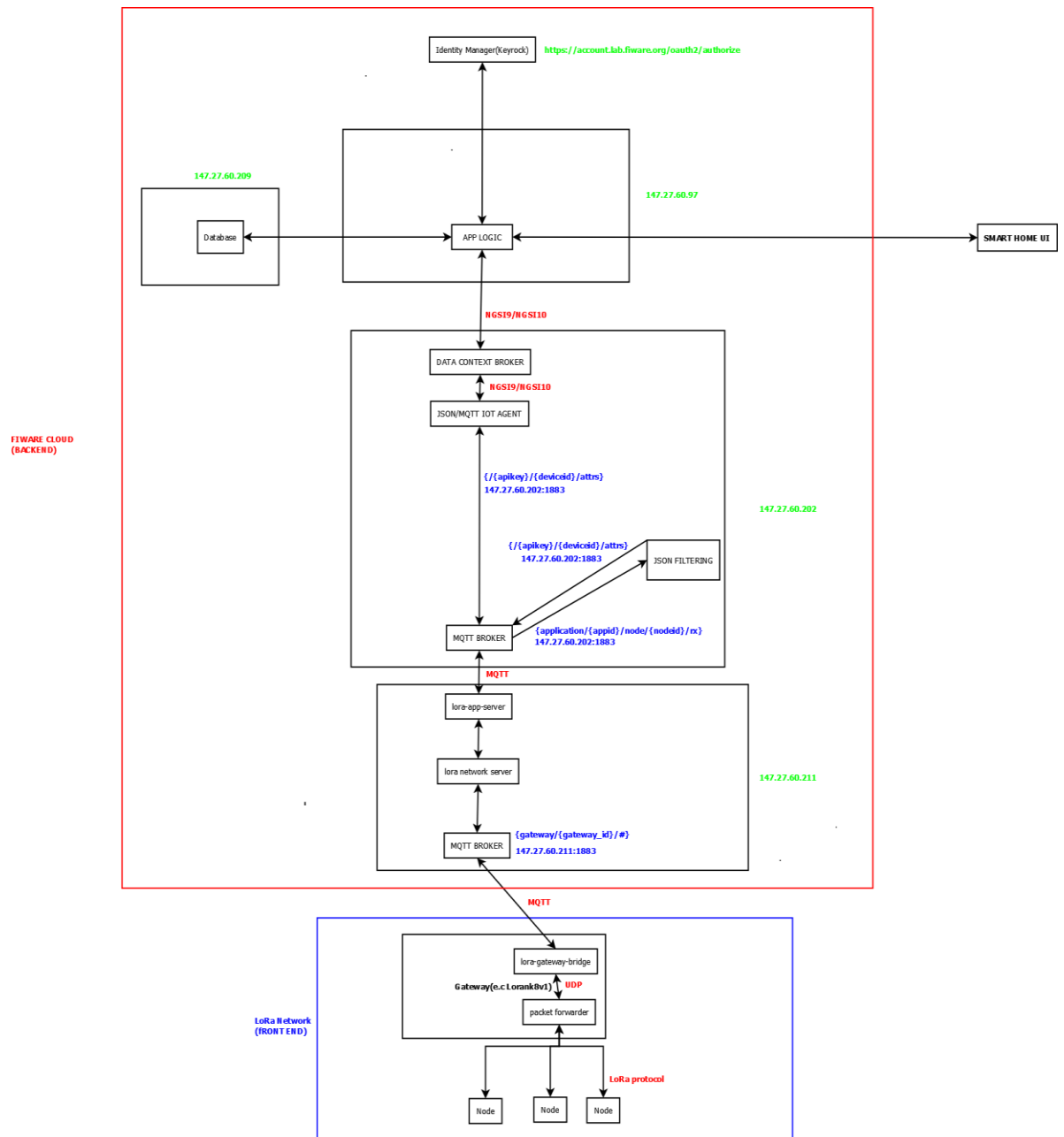


Figure 18: Architectural diagram of LoRaWare (LoRa Network and LoraWare Architecture)

At the above diagram (Figure 18) we present the whole implementation of our system where are shown both the lora network (devices, gateways and services on gateways) and the Cloud Infrastructure with the services running on different virtual machines. It presents also the floating IPs of every virtual machine (green color), the mqtt broker's topics where the data are published during every mqtt communication

(blue color) and the protocol which is used for the service communication at every part of architecture (red color)

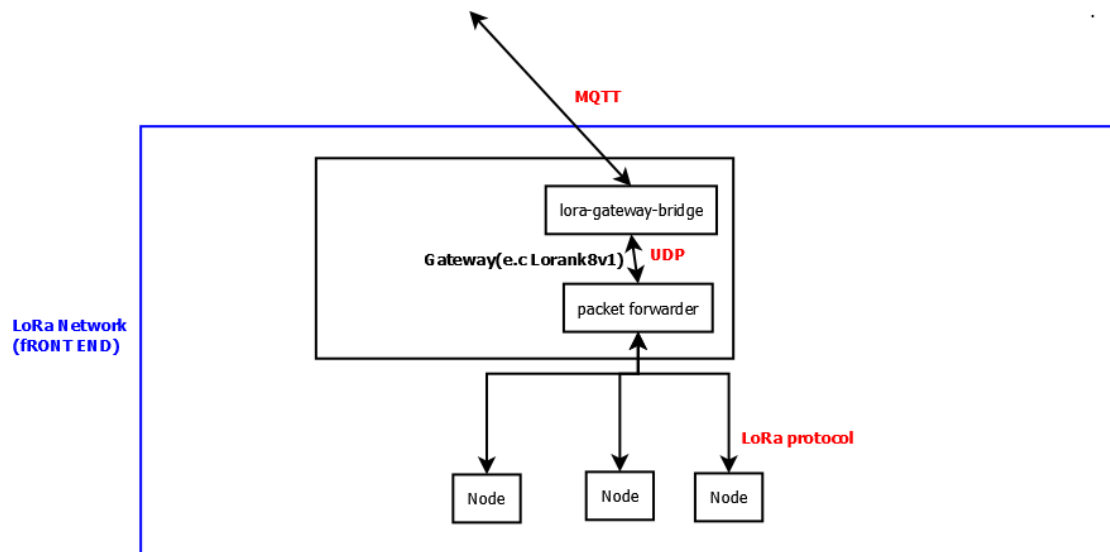


Figure 19: LoRa Network – Front end of LoRaWare

Initially we describe the LoRa Network (Front End-Figure 19). Using the LoRaWAN protocol the Nodes transmit RF packets which are captured by one or more Gateways. After catching a LoRa packet, the Semtech's packet forwarder converts it to UDP. The packet forwarder is the Semtech's open source software which converts the LoRa packets to UDP and vice versa. Our main purpose was to convert the packets to a protocol which would be supported from the Fiware's IoT agents which are responsible for the device management of IoT sector. We deployed an open-source service called lora-gateway-bridge which takes the packets from the Gateway's packet forwarder and converts them from UDP to MQTT/TCP in order to be forwarded to the network server over IP. After the conversion from UDP to MQTT, the packets are forwarded from the lora-gateway-bridge to the Cloud via MQTT which is a publish/subscribe protocol. This means that a MQTT broker is needed for the communication between a server and a client as we have already referred (section 2.2.2). The publish/subscribe mechanism give us the ability for one to many communication. This means that every payload which is published in the mqtt broker could be sent to many mqtt clients. Practically, we could send the same packet simultaneously to

different cloud infrastructures for different processing. As a result, the network server can serve many clients in a short time through a secure communication with an easy packet handling and monitoring. This is the reason why the MQTT protocol is faster and more secure instead of other IP protocols like COAP, HTTP or UDP.

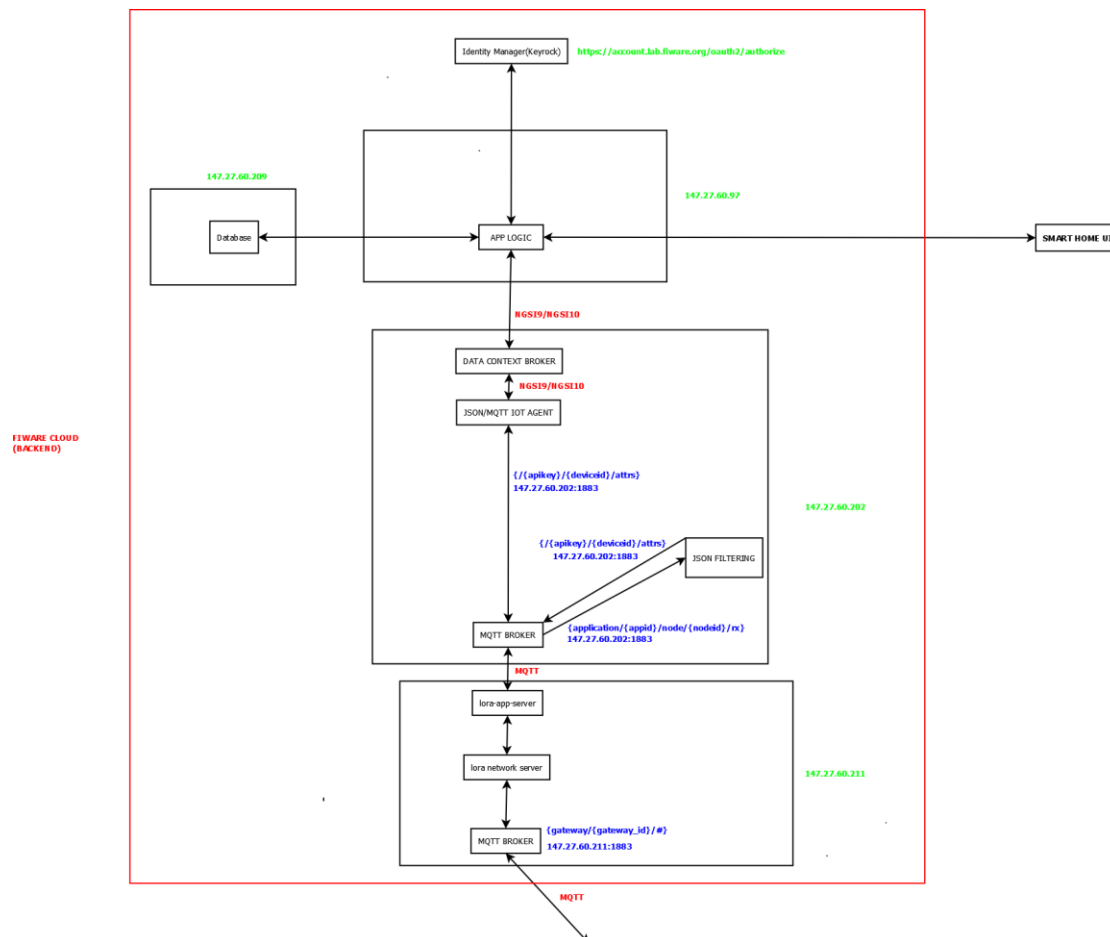


Figure 20: LoRaWare service oriented architecture – Back end of LoRaWare

Figure 20 shows the back end of the LoRaWare architecture. This contains all the Fiware Cloud Infrastructure with the services run on it. The services have been deployed and run on four different virtual machines of the Fiware Lab and we use also a public instance for the Keyrock Identity Management operations. Except of all devices and services, it presents also the floating IPs of every virtual machine (green color), the mqtt broker's topics where the data are published at every level (blue color) and the protocol which is used for the service communication at every part of architecture (red color).

The virtual machine with floating ip 147.27.60.211 hosts the services for the LoRa Network Server. Generally, the network server is responsible to know the active sessions (devices that have joined the network), serve the new nodes when join the network, decode and decrypt the physical payload of the packets, deduplicate the received data (which is potentially captured by multiple gateways) , authenticates this data to make sure that these are not replay attacks. Also, it manages the state of the node through mac-commands (e.g, to change data rate, channels, etc.).

More specifically, the lora-server is subscribed on a specific topic of mqtt broker where the lora-gateway-bridge service publishes the packets which are passed by the gateway's packet forwarder. After the decoding, the decryption and the deduplication of the packets we publish them on a different mqtt broker on another virtual machine with floating ip 147.27.60.202. Practically, we could use the same mqtt broker instance hosted on the virtual machine with floating ip 147.27.60.211 but we deploy a second mqtt broker on the other virtual machine because in the future if we use also devices that support directly the mqtt protocol they won't have to pass from the first virtual machine and the lora network server.

On the virtual machine with floating ip 147.27.60.202 we have develop our own json parser in order to filter the json payloads that the lora server publishes to the mqtt broker. JSON filtering service gives the format of the payload that is needed for the Fiware's JSON/MQTT IoT agent. More specifically it ignores the metadata of the payload and keeps only the values of the sensors as an one level key value pair JSON payload which is republished on a different topic of the mqtt broker.

Finally, we register our devices (sensors) to the JSON/MQTT IoT agent which is subscribed on a specific mqtt broker's topic (different for every device) waiting for json payload to be published. Every time a payload is published at a mqtt broker's topic, the payload is then converted from MQTT to NGSI protocol by the IoT agent and is forwarded at the Context Broker.

The Orion Context Broker is an implementation of the Fiware's Publish/Subscribe Context Broker GE, providing the NGSi9 and NGSi10 interfaces (sections 2.1.8 & 2.1.10). Using these interfaces, clients can do several operations such as register context producer applications, update context information, being notified when changes on context information take place or query context information.

Two LoRa sensors are sending measures about Humidity and Temperature every minute. The application informs us for the current Humidity and Temperature and also for the date and time they were measured. The measures are changed automatically in the User Interface every time a different payload is transmitted. All different measures are stored in a database on a different virtual machine which gives the opportunity to develop other applications using our sensor's data independently. Our application finally provides historical measurements about the last 10 different measures of temperature and humidity for every sensor and these are changed automatically every time a different measure is captured.

Finally, we use the public instance of the Keyrock Identity Manager service of Fiware for authentication during the log in and for controlling users access to services according to their roles. (e.g., administrator, user, etc.).

## 4.2 LoRa Devices<sup>35</sup>

We used two same set of boards from Ideetron company which constitute a LoRa Node.

Every Node has one Nexus Board which constitutes the microcontroller, one Nexus Demoboard where a number of sensors is mounted on it and a PCB antenna for the transmission of RF packets.

---

<sup>35</sup> <https://webshop.ideetron.nl/>

### 4.2.1 Nexus Board

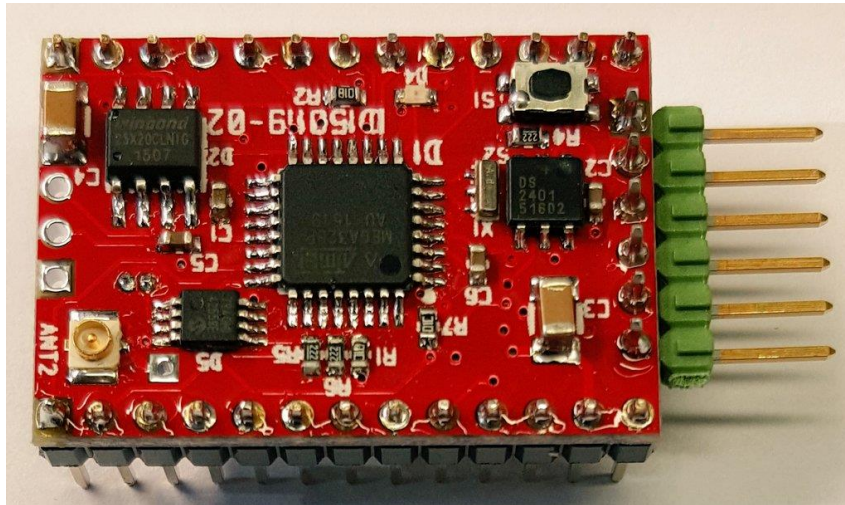


Figure 21: Ideetron's Nexus Board Microcontroller

Nexus Board (Figure 21) is the microcontroller we chose for our LoRa Nodes. It is based on a Arduino Mini shape. It needs power supply of 3,3Vdc and its dimensions are 23x33 mm.

On the nexus board are placed the following components:

ATMEGA328P-AU MCU, 8BIT, ATMEGA, 20MHZ, TQFP-32

DS2401P+ SILICON SERIAL NUMBER

MCP7940M-I/MS RTC, I2C, 64BYTES SRAM

AZ1117CR-3.3TRG1 LDO VOLT REG, 0.5A, 3.3V (normal mode; select)

W25X40CLSNIIG-ND FLASH 4MBIT

U.FL antenna connector

RFM95W or RFM98W (select)



## 4.2.2 Nexus Demoboard



Figure 22: Ideetron's Nexus Demoboard

We used Ideetron's Nexus Demoboard (Figure 22) where are mounted the following components:

Header for LoRa Nexus Board

PIR, Panasonic EKMB110111

Temp. & RH% sensor Si7021-A20

LDR: NSL 19M51

Potentiometer 10kA 4-turn

Movement sensor MVS0608.02

2x LED

2x Push button

## 4.2.3 PCB Antenna 868 MHz UFL

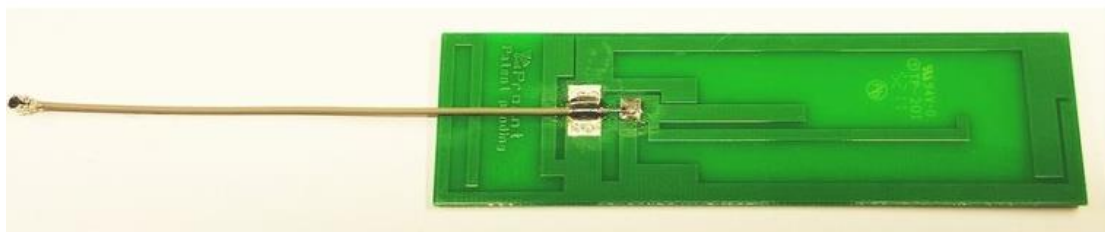


Figure 23: Ideetron's PCB Antenna 868MHz UFL

We connected the PCB antenna (Figure 23) with the nexus board antenna UFL connector.

The Antenna has the following features:

Cable Length: 10 cm (65 mm outside PCB)

PCB dimensions: 100x28 mm

Weight: 4 gram

#### 4.2.4 Arduino Sketches

Combining the three above components e.g., the nexus board, the nexus demoboard and the PCB antenna we have a complete LoRa Node.

A code or program written for Arduino called Sketch. Ideetron provides many Arduino sketches which can be loaded on the nexus board using a TTL cable and the Arduino IDE in order to make the Node functional. The

For our own application, we use the TH06 arduino sketch of Ideetron for low power LoRaWan. This sketch wakes up the Node each minute, measures the temperature, humidity from the Demoboard and transmits the results in LPP format. This sketch makes use of the Nexus Demoboard and defines the basic characteristics of our Node such as data rate, class of device, frequency channel and the type of the activation. We chose the Class A device and the Activation By Personalization. Every time we cut the power supply of node, we have to activate it again every time we turn it on. This can be done from the lora-app-server UI which we will describe below.

Relatively with the node's payload, we use the Cayenne<sup>36</sup> LPP format. The Cayenne Low Power Payload (LPP) provides a convenient and easy way to send data over LPWAN networks such as LoRaWAN. The Cayenne LPP is compliant with the payload size restriction, which can be lowered down to 11 bytes, and allows the device to send multiple sensor data at one time.

Additionally, the Cayenne LPP allows the device to send different sensor data in different frames. In order to do that, each sensor data must be prefixed with two bytes:

- **Data Channel:** Uniquely identifies each sensor in the device across frames, eg. "indoor sensor"

---

<sup>36</sup> <https://github.com/myDevicesIoT/cayenne-docs/blob/master/docs/LORA.md>

- **Data Type:** Identifies the data type in the frame, eg. “temperature”

The payload structure is a sequence of bytes as we can see below:

1 Byte	1 Byte	N Bytes	1 Byte	1 Byte	M Bytes	...
Data1 Ch.	Data1 Type	Data1	Data2 Ch.	Data2 Type	Data2	...

Finally, Data Types conform to the IPSO Alliance Smart Objects Guidelines, which identifies each data type with an “Object ID”. However, as shown below, a conversion is made to fit the Object ID into a single byte.

```
LPP_DATA_TYPE = IPSO_OBJECT_ID - 3200
```

Each data type can use 1 or more bytes to send the data according to the following table (Figure 24).

Type	IPSO	LPP	Hex	Data Size	Data Resolution per bit
Digital Input	3200	0	0	1	1
Digital Output	3201	1	1	1	1
Analog Input	3202	2	2	2	0.01 Signed
Analog Output	3203	3	3	2	0.01 Signed
Illuminance Sensor	3301	101	65	2	1 Lux Unsigned MSB
Presence Sensor	3302	102	66	1	1
Temperature Sensor	3303	103	67	2	0.1 °C Signed MSB
Humidity Sensor	3304	104	68	1	0.5 % Unsigned
Accelerometer	3313	113	71	6	0.001 G Signed MSB per axis
Barometer	3315	115	73	2	0.1 hPa Unsigned MSB
Gyrometer	3334	134	86	6	0.01 °/s Signed MSB per axis
GPS Location	3336	136	88	9	Latitude : 0.0001 ° Signed MSB
					Longitude : 0.0001 ° Signed MSB
					Altitude : 0.01 meter Signed MSB

Figure 24: Cayenne LPP format – Table of Data representation

The whole logic of the Cayenne LPP format has been implemented in Ideetron's Arduino sketches we used in order to be possible to project our payloads clearly on a dashboard. We take the temperature measures in Celsius degrees and the humidity measures in percentage.

### 4.3 Lorank 8 Gateway



Figure 25: Ideetron's Lorank8v1 Gateway

The Lorank 8 (Figure 25) is the first LoRa Gateway with professional specifications which constitutes an Ideetron's product. With almost 50 DSP pipes on board it processes 8 LoRa transmissions simultaneously. This enables the connection with several tens of thousands end nodes around the gateway. And, with a sensitivity of -138 dBm and a maximum power of 500 mW we can easily reach the most distant nodes. According to Ideetron, although the maximum connection distance is ~25km in open terrain, buildings and metal structures do hinder the transmission. Experience learns that distances of 5km are realistic if the gateway is mounted on a (high) point with free sight.

The hardware is based on the high quality radio board of IMST(tm) and the open source Beagle Board. Also, the software is completely open source and may be changed to our liking.

Lorank Gateway has the following characteristics:

## Hardware:

- Frequency band : 868 MHz
- Sensitivity: -138 dBm
- Maximum power: 27 dBm (500mW)
- LoRa demodulators: 49
- Simultaneous channel : 8
- Max connected nodes: ~60 thousand (\*)
- Processor: 1GHz, ARM Cortex A8
- OS: Debian / Angstrom Linux
- Wifi: Optional (via USB, not implemented yet)
- Current : 1A
- Max Current USB : 500mA
- Power Adapter : 5Volt= , 2Amp

(\*) This is a theoretical maximum, under the assumption that nodes only send once per hour. Due to collisions, resend packets, packet loss etc., the number of nodes that can effectively be handled is lower, typically 10..20 thousand.

## Software:

- Lora libraries : Semtech, with modifications from Beta Research BV
- basic packet forwarder : Semtech,
- poly packet forwarder : Beta Research BV, based on code from Semtech
- Installation scripts : Beta Research BV
- Beagle Bone : Various versions

### 4.3.1 The Lora-Gateway-Bridge Service<sup>37</sup>

As we have already described, the gateway uses the UDP protocol in order to send the data to the network server. However, we have deployed on the gateway an open-source-service called **lora-gateway-**

---

<sup>37</sup> <https://github.com/brocaar/lora-gateway-bridge>

**bridge.** LoRa Gateway Bridge is a service which abstracts the packet-forwarder UDP protocol into JSON over MQTT.

There are three basic reasons why we chose the MQTT protocol for the communication between the Gateway and the Cloud:

- **Visibility:** Because LoRa Gateway Bridge publishes the content of the UDP packets as JSON over MQTT, it becomes trivial to monitor the data that is sent and received by each gateway just by subscribing at all topics of the mqtt broker. This wouldn't be easy with UDP.
- **Routing:** The MQTT broker will handle the routing of which (downlink) frame must be sent to which LoRa Gateway Bridge instance. This could be easy just by publishing the downlink payload into different topics (different for every gateway).
- **Security:** By running the LoRa Gateway Bridge on the gateway itself, it is possible to use MQTT over Transport Layer Security (TLS), meaning the transport between the gateway and server(s) is secure.
- **One to many Communication:** The publish/subscribe mechanism means that every payload which is published in the mqtt broker could be sent to many mqtt clients (they subscribe to the topic where the server publishes data). Practically, we could send the same packet simultaneously to different cloud infrastructures for different processing just by one transmission. As a result, the network server can serve many clients in a short time simultaneously.

The only configuration we had to do, was to modify the packet-forwarder of the gateway so that it would send its data to the LoRa Gateway Bridge. We did that only by changing the following configuration keys in the file "global\_conf.json" via ssh connection to the gateway:

- `server_address` to the IP address / hostname(0.0.0.0 for localhost) of the LoRa Gateway Bridge

- serv\_port\_up to 1700 (the default port that LoRa Gateway Bridge is using)
- serv\_port\_down to 1700 (same)

After that we run the packet forwarder and the lora-gateway-bridge as services in the Ubuntu Operating System of the Gateway. We should follow this procedure for every gateway we would like to add in our LoRaWare Architecture. Finally, lora-gateway-bridge publishes its data into the mqtt broker. We describe this operation in the next section.

## 4.4 The Cloud Services of the LoRaWare Architecture

All Cloud Services of our architecture have been deployed in several Virtual Machines which we created using the Dashboard of the Fiware lab. Only for the Keyrock Identity Manager we used a public instance that Fiware provides.

### 4.4.1 The Mosquitto MQTT Broker<sup>38</sup>

Eclipse Mosquitto is an open source (EPL/EDL licensed) message broker that implements the MQTT protocol versions 3.1 and 3.1.1. Mosquitto is lightweight and is suitable for use on all devices from low power single board computers to full servers.

The MQTT protocol provides a lightweight method of carrying out messaging using a publish/subscribe model. This makes it suitable for Internet of Things messaging such as with low power sensors or mobile devices such as phones, embedded computers or microcontrollers.

The lora-gateway-bridge we described in the previous section is publishing LoRa frames to the MQTT broker. Every client which is subscribed to the MQTT topic “gateway/+/rx”, could monitor and receive those frames. The “+” on a topic represents a single-level wildcard. Any topic matches to a topic including the single level wildcard if it contains an arbitrary string (the Gateway ID in our case) instead of the wildcard. For example, we can monitor all the received data from all gateways just by subscribing to the topic “gateway/+/rx”, but we can

---

<sup>38</sup> <https://mosquitto.org/>

monitor only the received data from the gateway with ID: 1dee18c14948a955 just by subscribing to the topic "gateway/1dee18c14948a955/rx".

While the single level wildcard only covers one topic level, the multi level wildcard "#" covers an arbitrary number of topic levels. In order to determine the matching topics it is required that the multi level wildcard is always the last character in the topic and it is preceded by a forward slash. A client subscribing to a topic with a multi level wildcard is receiving all messages, which start with the pattern before the wildcard character, no matter how long or deep the topics will get. If we only specify the multilevel wildcard as a topic (#), it means that we will get every message sent over the MQTT broker. For example, if a client subscribes to the topic "gateway/1dee18c14948a955/#" , can monitor topics like "gateway/1dee18c14948a955/rx" (received data from the gateway with the specific ID), "gateway/1dee18c14948a955/tx" (transmitted data to the gateway with the specific ID" or "gateway/1dee18c14948a955/stats" (statistics about the gateway with specific ID).

For debugging purposes we can use in the terminal the command `mosquitto_sub` in order to subscribe to a specific topic and see the data.

When we are subscribed to the `gateway/+ /rx`, we see the data received from every gateway. If we don't see anything, it means either that the LoRa Gateway Bridge does not receive data from the packet forwarder or that the MQTT credentials/authorizations are invalid (the user is not authorized to subscribe to the MQTT topic).

In addition, there is also the `mosquitto_pub` command, which can be used for publishing data into the mqtt broker. However, for both publishing and subscribing into the mqtt broker there are specific libraries for the most common programming languages in order to achieve a communication via the MQTT protocol. We used one of them for our json filtering which we are going to describe in a section below. Such libraries are also used for the Go programming language into the lora server service as and for the Node JS into the MQTT IoT Agent .



We deployed two MQTT brokers in different Virtual Machines whose flowting IPs are presented in the architectural diagram of LoRaWare(Figure ?).

#### 4.4.2 LoRa Server<sup>39</sup>

**LoRa Server** is an open-source LoRaWAN network-server, part of the open-source LoRa Server project. The responsibility of the network server component is the de-duplication and handling of received uplink frames received by the gateway(s), handling of the LoRaWAN mac-layer and scheduling of downlink data transmissions.

More specifically, the LoRa Server component is responsible for the network. It knows about active node sessions (nodes which have join the network) and when a new node joins the network, it will ask the application-server if the node is allowed to join the network and if so, which settings to use for this node.

For the active node-sessions, it de-duplicates the received data (which is potentially received by multiple gateways), it authenticates this data (to make sure that these are not replay-attacks), it forwards this (encrypted) data to the application-server and it will ask the application-server if it should send anything back.

Besides managing the data-flows, it also manages the state of the node through so called mac-commands (e.g. to change the data-rate, channels, ...).

LoRa Server implements a gRPC API so that we could easily build our own application-server.

The following table represents the features of the lora server (Figure 26):

---

<sup>39</sup> <https://www.loraserver.io/loraserver/overview/>

<b>Device classes</b>	Class-A, Class-B & Class-C
<b>Message types</b>	(Un)confirmed uplink and downlink, Proprietary uplink and downlink
<b>Device activation</b>	Over-the-air (OTAA) and activation by personalization
<b>Adaptive data-rate</b>	Supported for all regions
<b>Regions supported</b>	AS 923 AU 915-928 CN 470-510 CN 779-787 EU 433 EU 863-870 IN 865-867 KR 920-923 US 902-928
<b>Frame-counter validation</b>	Strict (default) Skip frame-counter mode (for debugging <b>only</b> )
<b>Statistics</b>	Per gateway received / transmitted (configurable aggregation levels)
<b>Mac-layer handling</b>	Channel (re)configuration Adaptive data-rate Device-status Link check (initiated by the device) Ping-slot channel configuration Device time RX parameter (re)configuration
<b>Integration</b>	gRPC based API to an external application-server and external network-controller (optional)

Figure 26: LoRa Server's specifications

We deployed the lora server on a virtual machine of fiware lab with Ubuntu 14.04 Operating System running on it.

In addition, for the lora server operation two databases are required. We install a PostgreSQL for the LoRa server to persist the gateway data and a Redis datastore to store all non-persistent data.

PostgreSQL<sup>40</sup> is a powerful, open source object-relational database system. Its main characteristics are reliability, feature robustness, and performance.

Redis<sup>41</sup> is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperlogs and geospatial indexes with radius queries. Redis has built-in replication, Lua scripting, LRU eviction, transactions and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster.

<sup>40</sup> <https://www.postgresql.org/>

<sup>41</sup> <https://redis.io/>

### 4.4.3 LoRa App Server<sup>42</sup>

**LoRa App Server** is an open-source LoRaWAN application-server, part of the LoRa Server project. It is responsible for the device “inventory” part of a LoRaWAN infrastructure, handling of join-request and the handling and encryption of application payloads.

It offers a web-interface where users, organizations, applications and devices can be managed. For integration with external services, it offers a RESTful and gRPC API. Device data can be sent and / or received over MQTT, HTTP and be written directly into InfluxDB.

More specifically, it is compatible with the LoRa Server component and offers node management per application, per organization and gateway management per organization. It also offers user management and the possibility to assign users to organizations and / or applications. Communication with the application is using JSON over MQTT and using the exposed APIs. Finally it provides a web interface and an API which can be used for all the above operations.

LoRa App Server uses, as LoRa Server does, a PostgreSQL database to persist the gateway data and stores all non-persistent data into a Redis datastore.

We have deployed the LoRa App Server on the same VM with the LoRa Server.

Binding the url: <https://147.27.60.211:8080> with any browser we enter the User Interface of LoRa App Server (Figure 27):

---

<sup>42</sup> <https://www.loraserver.io/lora-app-server/overview/>

Figure 27: LoRa App Server UI – Login Page

For authentication and authorization, **users** can be created (Figure 28) in LoRa App Server. A user itself can be a global admin or a regular user. A **global admin user** is authorized to perform any action. It can for example manage gateways, users, create organizations, applications and nodes. A regular user has no permissions by default. However, it can be assigned to one or multiple organizations.

Figure 28: LoRa App Server UI – User Creation

LoRa App Server is able to connect to one or multiple LoRa Server network-server instances. Global admin users are able to add new network-servers to the LoRa App Server installation (Figure 29). Once a network-server is assigned to a service-profile or device-profile, a network-server can't be removed before deleting these entities, it will return an error. Depending the configuration of LoRa Server and LoRa

App Server, we must enter the CA and client certificates in order to let LoRa App Server connect to LoRa Server and in order to let LoRa Server connect to LoRa App Server.

**LoRa Server** Organizations Users Network servers **itsakos**

**Network-server name**  
e.g. EU868 network-server  
A memorable name of the network-server.

**Network-server server**  
e.g. localhost:8000  
The hostname/IP of the network-server.

**Certificates for LoRa App Server to LoRa Server connection**

**CA certificate**

Paste the content of the CA certificate (PEM) file in the above textbox. Leave blank to disable TLS.

**TLS certificate**

Paste the content of the TLS certificate (PEM) file in the above textbox. Leave blank to disable TLS.

**TLS key**

Paste the content of the TLS key (PEM) file in the above textbox. Leave blank to disable TLS. Note: for security reasons, the TLS key can't be retrieved after being submitted (the field is left blank). When re-submitting the form with an empty TLS key field (but populated TLS certificate field), the key won't be overwritten.

**Certificates for LoRa Server to LoRa App Server connection**

**CA certificate**

Paste the content of the CA certificate (PEM) file in the above textbox. Leave blank to disable TLS.

**TLS certificate**

Paste the content of the TLS certificate (PEM) file in the above textbox. Leave blank to disable TLS.

**TLS key**

Paste the content of the TLS key (PEM) file in the above textbox. Leave blank to disable TLS. Note: for security reasons, the TLS key can't be retrieved after being submitted (the field is left blank). When re-submitting the form with an empty TLS key field (but populated TLS certificate field), the key won't be overwritten.

Figure 29: LoRa App Server UI – Network server assignment

From the UI we can manage also the organizations.

LoRa Server

Organizations Users Network servers ktsakos

Organizations / Create organization

Organization name  
e.g. my-organization  
The name may only contain words, numbers and dashes.

Display name  
My Organization

Can have gateways  
☐ Can have gateways  
When checked, it means that organization administrators are able to add their own gateways to the network. Note that the usage of the gateways is not limited to this organization.

GO BACK SUBMIT

Figure 30: LoRa App Server UI – Organization creation

An **organization** can be used (Figure 30) to let organizations or teams manage their own applications and optionally their own gateways. An organization can have service-profiles, device-profiles, gateways (when allowed), applications and users (Figure 31).

LoRa Server

Organizations Users Network servers ktsakos

Organizations / ktsakos

DELETE ORGANIZATION

Applications Gateways Organization configuration Organization users Service profiles Device profiles

CREATE APPLICATION

ID	Name	Service-profile	Description
1	Temperature-Humidity	private service-profile	Measures by sensors

Figure 31: LoRa App Server UI – List of Applications for specific organization

The **service-profile** can be seen as the “contract” between an user and the network (Figure 32). It describes the features that are enabled for the user(s) of the service-profile and the rate of messages that can be sent over the network. When creating a service-profile, LoRa App Server will create the actual profile on the selected network-server, and will keep a reference record so it knows to which organization it belongs.

The screenshot shows the 'Create service-profile' form in the LoRa App Server UI. The form is titled 'Create service-profile' and contains the following fields and options:

- Service-profile name:** A text input field with a placeholder 'e.g. my-service-profile' and a note 'A memorable name for the service-profile.'
- Network-server:** A dropdown menu with a 'Select...' placeholder. A note below states: 'The network-server on which this service-profile will be provisioned. After creating the service-profile, this value can't be changed.'
- Add gateway meta-data:** A checkbox labeled 'Add gateway meta-data'. A note below states: 'GW metadata (RSSI, SNR, GW geoLoc, etc.) are added to the packet sent to the application-server.'
- Device-status request frequency:** A text input field with a value of '0'. A note below states: 'Frequency to initiate an End-Device status request (request/day). Set to 0 to disable.'
- Report battery level:** A checkbox labeled 'Report battery level'. A note below states: 'Report End-Device battery level to application-server.'
- Report margin:** A checkbox labeled 'Report margin'. A note below states: 'Report End-Device margin to application-server.'
- Minimum allowed data-rate:** A text input field with a value of '0'. A note below states: 'Minimum allowed data rate. Used for ADR.'
- Maximum allowed data-rate:** A text input field with a value of '0'. A note below states: 'Maximum allowed data rate. Used for ADR.'

At the bottom right of the form, there are two buttons: 'GO BACK' and 'SUBMIT'.

Figure 32: LoRa App Server UI – Service profile creation

A **device-profile** defines the device capabilities and boot parameters that are needed by the network-server for setting the LoRaWAN radio access service. These information elements shall be provided by the end-device manufacturer. When creating a device-profile (Figure 33), LoRa App Server will create the actual profile on the selected network-server, and will keep a reference record so it knows to which organization it belongs.

The screenshot shows the 'Create device-profile' form in the LoRa App Server UI. The form is titled 'Create device-profile' and has three tabs: 'General', 'Join (OTAA / ABP)', and 'Class-C'. The 'General' tab is selected. The form contains the following fields:

- Device-profile name:** A text input field with a placeholder 'e.g. my device-profile' and a note 'A memorable name for the device-profile.'
- Network-server:** A dropdown menu with a 'Select...' option and a note 'The network-server on which this device-profile will be provisioned. After creating the device-profile, this value can't be changed.'
- LoRaWAN MAC version:** A dropdown menu with a 'Select...' option and a note 'Version of the LoRaWAN supported by the End-Device.'
- LoRaWAN Regional Parameters revision:** A dropdown menu with a 'Select...' option and a note 'Revision of the Regional Parameters document supported by the End-Device.'
- Max ERP:** A text input field with a value of '0' and a note 'Maximum ERP supported by the End-Device.'

At the bottom right of the form, there are two buttons: 'GO BACK' and 'SUBMIT'.

Figure 33: LoRa App Server UI – Device profile creation

An **application** is a collection of devices with the same purpose / of the same type. Think of a weather station collecting data at different locations for example. When creating an application (Figure 34), we need to select the Service-profile which will be used for the devices created under this application. Note that once a service-profile has been selected, it can't be changed. An application can be configured to decode the received uplink payloads from bytes to a meaningful data object, and to encode downlink objects to bytes. The Cayenne LPP codec, which is used in the Arduino sketches we load on our Nodes, is supported from the LoRa App Server. However, we can write our own javascript codec functions.



The screenshot shows the 'Create application' form in the LoRa App Server UI. The form has the following fields:

- Application name:** A text input field with a placeholder 'e.g. "temperature-sensor"'. Below it, a note states: 'The name may only contain words, numbers and dashes.'
- Application description:** A text input field with a placeholder 'a short description of your application'.
- Service profile:** A dropdown menu with 'Select...' as the current selection. Below it, a note states: 'The service-profile to which this application will be attached. Note that you can't change this value after the application has been created.'
- Payload codec:** A dropdown menu with 'Join...' as the current selection. Below it, a note states: 'By defining a payload codec, LoRa App Server can encode and decode the binary device payload for you.'

At the bottom right of the form are two buttons: 'GO BACK' and 'SUBMIT'. In the top right corner of the page, there is a red button labeled 'DELETE ORGANIZATION'.

Figure 34: : LoRa App Server UI – Application creation

A **device** is the end-device connecting to, and communicating over the LoRaWAN network. LoRa App Server supports both OTAA (over the air activation) and ABP (activation by personalization) type devices (configured by the selected device-profile). When creating or updating a device (Figure 35), we need to select the device-profile matching the device capabilities. E.g. the device-profile defines if the device is of type OTAA or ABP.

The screenshot shows the 'Create device' form in the LoRa App Server UI. The form has the following fields:

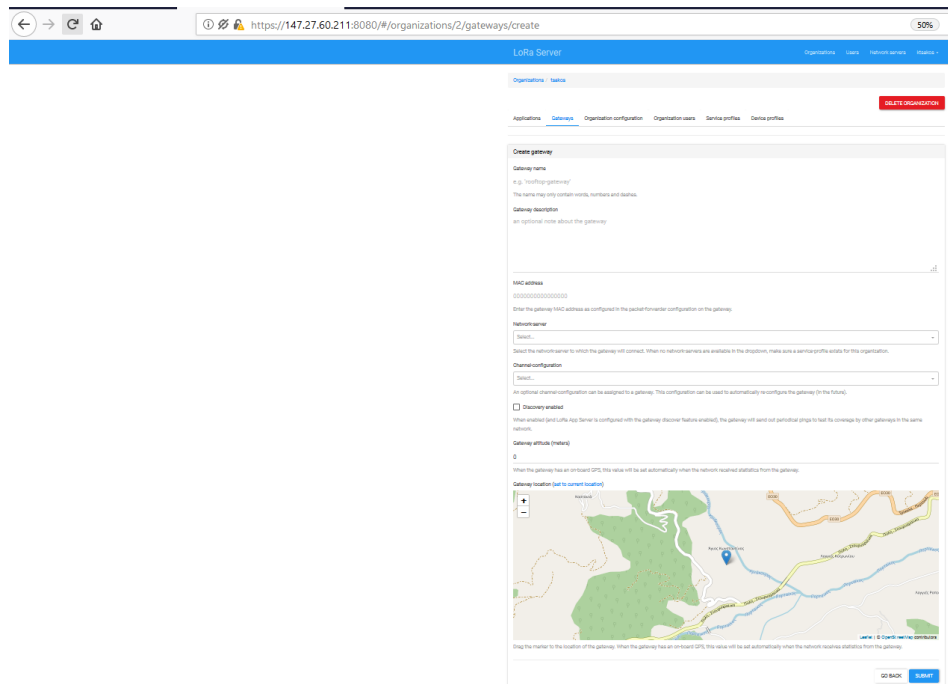
- Device name:** A text input field with a placeholder 'e.g. "garden-sensor"'. Below it, a note states: 'The name may only contain words, numbers and dashes.'
- Device description:** A text input field with a placeholder 'a short description of your node'.
- Device EUI:** A text input field with a placeholder '0000000000000000'.
- Device-profile:** A dropdown menu with 'Select...' as the current selection.

At the bottom right of the form are two buttons: 'GO BACK' and 'SUBMIT'. In the top right corner of the page, there is a red button labeled 'DELETE APPLICATION'.

Figure 35: LoRa App Server UI – Node (device) registration to specific application

An organization is able to manage its own set of **gateways** (Figure 36). This feature might be unavailable when the organization is configured without gateway support. That a gateway belongs to a given organization does not mean that the usage of a gateway is limited to the organization. Every node in the whole network will be able to communicate using the gateway. The organization will be responsible

however for managing the gateway details (e.g. name, location) and will be able to see its statistics. Gateway statistics are based on the aggregated values sent by the gateway / packet-forwarder. In case no statistics are visible, it could mean that the gateway is incorrectly configured.



The screenshot shows the 'Create gateway' form in the LoRa App Server UI. The browser address bar shows the URL: <https://147.27.60.211:8080/#/organizations/2/gateways/create>. The page has a blue header with 'LoRa Server' and navigation links for 'Organizations', 'Users', 'Network server', and 'Statistics'. The main content area is titled 'Create gateway' and includes the following fields and options:

- Gateway name:** A text input field with a placeholder 'e.g. "foo@bar-gateway"'. A note below states: 'The name may only contain words, numbers and dashes.'
- Gateway description:** A text area for an optional note about the gateway.
- MAC address:** A text input field with a placeholder '0000000000000000'. A note below states: 'Enter the gateway MAC address as configured in the packet-forwarder configuration on the gateway.'
- Network server:** A dropdown menu with a 'Select...' button. A note below states: 'Select the network server to which the gateway will connect. When no network servers are available in the dropdown, make sure a service profile exists for this organization.'
- Channel configuration:** A dropdown menu with a 'Select...' button. A note below states: 'An optional channel configuration can be assigned to a gateway. This configuration can be used to automatically reconfigure the gateway (in the future).'
- Discovery enabled:** A checkbox. A note below states: 'When enabled (and LoRa App Server is configured with this gateway discovery feature enabled), the gateway will send out periodic ping to test its coverage by other gateways in the same network.'
- Gateway altitude (meters):** A text input field with a placeholder '0'. A note below states: 'When the gateway has an on-board GPS, this value will be set automatically when the network requests statistics from the gateway.'
- Gateway location (set to current location):** A map showing a geographical area with a blue location pin. A note below states: 'Drag the marker to the location of the gateway. When the gateway has an on-board GPS, this value will be set automatically when the network requests statistics from the gateway.'

At the bottom right of the form are 'GO BACK' and 'SUBMIT' buttons.

Figure 36: LoRa App Server UI – Gateway registration

LoRa App Server makes it possible to **log frames** sent and received by a gateway or device in realtime. The frame logs view on the device detail page (Figure 37) will display only the frames that could be related to a device.

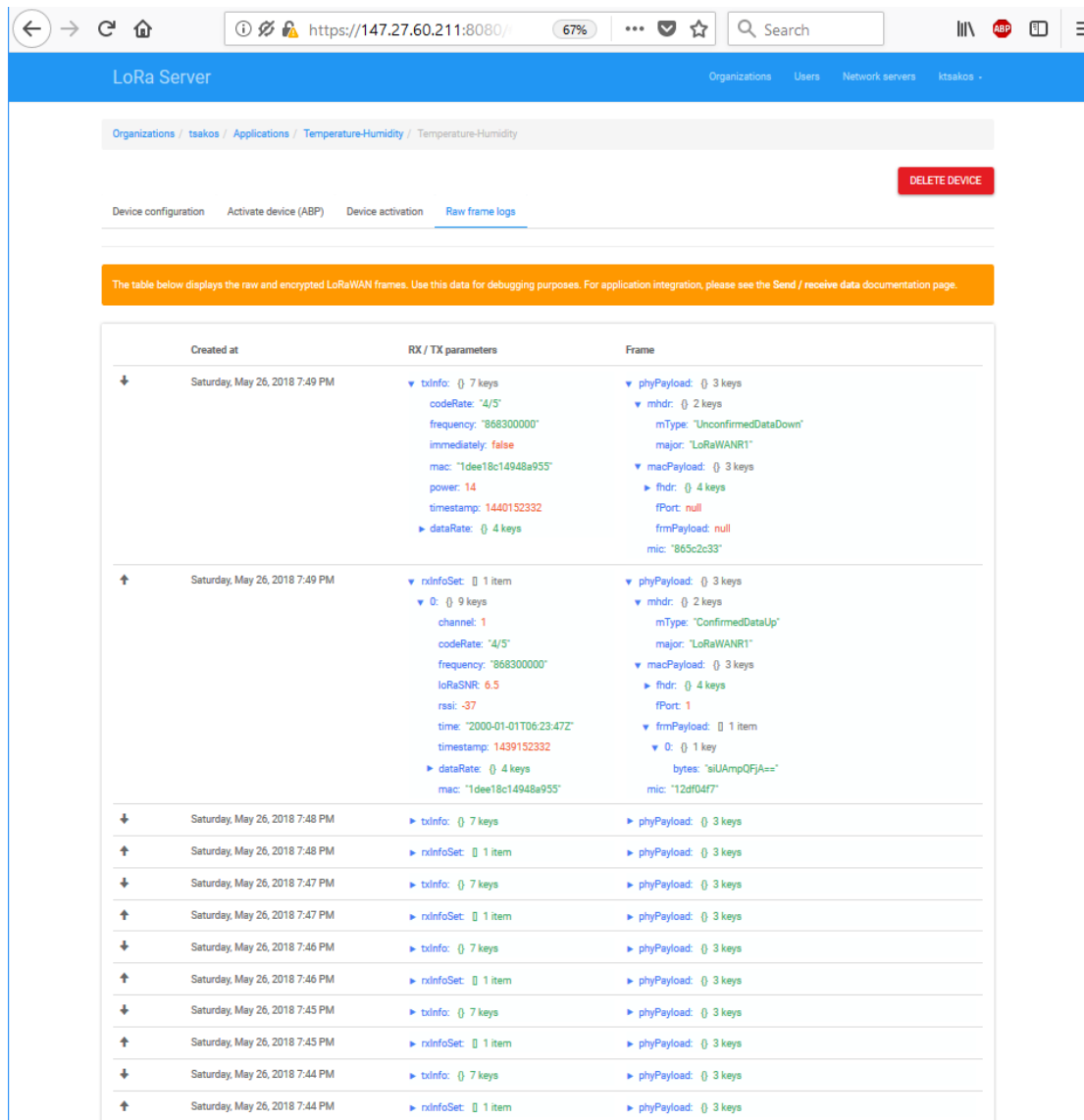


Figure 37: LoRa App Server UI – Log frames received by devices in real time

#### 4.4.4 JSON FILTERING

As we have already referred for the lora app server, it uses the MQTT to publish or receive data. With the MQTT integration lora app server publishes all the data it receives from the devices as JSON over MQTT. To receive data from our nodes, we therefore need to subscribe to its MQTT topic.

As the MQTT IoT agent of fiware requires a different payload format than this one the lora app server publishes, we have developed our own JSON parser(**JSON Filtering component**) using the python programming language.

More specifically, we used the paho.mqtt library to subscribe to a topic or publish data to the mqtt broker. Our script runs always as a daemon on our virtual machine and it is initially connected to the topic where the lora app server publishes its data. After this it parses the json payload, keeping only the information that the MQTT IoT agent needs. Finally it publishes the new JSON payload on a different topic where the IoT agent is subscribed to receive the data from our devices.

The topic where the LoRa App Server publishes its data and the JSON Filtering service is subscribed to is: “application/+ /node/#” and the topic where the JSON Filtering service publishes the data and the JSON/MQTT IoT agent is subscribed to is: “/{apikey}/{devEUI}/attrs”.

#### 4.4.5 JSON/MQTT IoT Agent<sup>43</sup>

By using an IoT Agent , our devices are represented in a FIWARE platform as NGSI entities in the ContextBroker. This means that we can query or subscribe to changes of device parameters status by querying or subscribing to the corresponding NGSI entity attributes at the ContextBroker.

Additionally, we could trigger commands if we used actuation devices just by updating specific command-related attributes in their NGSI entities representation at the Context Broker. This way, all developers interactions with devices are handled at a ContextBroker, providing an homogeneous API and interface as for all other non-IoT data in a FIWARE ecosystem.

We deployed on a virtual machine the JSON/MQTT IoT Agent in order to translate the MQTT protocol to NGSI. After parsing the decrypted data that LoRa-App-Server publishes into the MQTT broker, we republish them into a new topic:“/{apikey}/{devEUI}/attrs”.

Essentially, we exploit the Agent’s MQTT binding, which is based on the existence of a MQTT broker and the usage of different topics to separate the different destinations and types of the messages.

---

<sup>43</sup> <https://catalogue-server.fiware.org/enablers/backend-device-management-idas>

All the topics used in the protocol are prefixed with the APIKey of the device group and the Device ID of the device involved in the interaction. The API Key is a secret identifier shared among all the devices of a service, and the DeviceID is an ID that uniquely identifies the device in a service. API Keys can be configured with the IoT Configuration API or the public default API Key of the IoT Agent can be used in its instead. The Device ID must be provisioned in advance in the IoT Agent before the information is sent.

In order to send multiple measures our devices can publish a JSON payload to an MQTT topic with the following structure: `/api-key/{device-id}/attrs`

The message in this case must contain a valid JSON object of a single level; for each key/value pair, the key represents the attribute name and the value the attribute value. Attribute type will be taken from the device provision information.

In our case, we use the `mosquitto mqtt` broker as we have already mentioned. If a device with id: `000000000000000000`, API Key: `1234` and attribute IDs `temperatureSensor` and `humiditySensor` then all measures (temperature and humidity) are reported this way:

```
$ mosquitto_pub -t /1234/000000000000000000/attrs -m '{"TemperatureSensor": 22.3, "HumiditySensor": 70}' -h <mosquitto_broker> -p <mosquitto_port> -u <user> -P <password>
```

After publishing the data at the above format and deploying the JSON/MQTT IoT Agent we make a simple HTTP POST to create an IDAS Service:

```
POST http://147.27.60.202:5351/iot/services
```

Headers:

```
{
  'Content-Type':      'application/json',
  'X-Auth-Token' :     '[TOKEN]',
  'Fiware-Service':    'openiot',
  'Fiware-ServicePath': '/'
}
```

Payload:

```
{
  "services": [
    {
      "apikey": "1234",
      "cbroker": "http://0.0.0.0:1026",
      "entity_type": "thing",
      "resource": "/iot/d"
    }
  ]
}
```

The Context Broker has been deployed at the same VM, so we use the url: *0.0.0.0:1026*. Furthermore, we need also an OAuth token. In order to obtain it, we need our Fiware account username and password after running the *token\_script.sh* script that FIWARE provides. Essentially, it makes a rest call to obtain an OAuth token which expires after 1 hour.

At the last step, before our devices send observations or receive commands a register operation is needed. It is a HTTP Post to the IoT Agent's endpoint:

POST *http://147.27.60.202:4041/iot/devices/*

Headers:

```
{
  'Content-Type': 'application/json',
  'Fiware-Service': 'tourguide',
  'Fiware-ServicePath': '/'
}
```

Payload:

```
{
  "devices": [
    {
      "device_id": "0000000000000000",
      "protocol": "MQTT",
      "entity_name": "Multisensor1",
      "entity_type": "sensor",
      "attributes": [
        {
          "object_id": "temperatureSensor",
          "name": "temperature",
          "type": "number"
        },
        {
          "object_id": "humiditySensor",
          "name": "humidity",
          "type": "number"
        }
      ]
    }
  ]
}
```

After register our devices into the IoT Agent, every time a new payload is published into the MQTT broker, the IoT agent translates it to NGSI format. Then we can make HTTP GET requests to the Context Broker to query the values of our sensors (entities). We describe the Context Broker in the following section.

#### 4.4.6 Orion Context Broker<sup>44</sup>

We follow the Common Simple Scenario that Fiware suggests for connection with IoT (Figure 38). We can access IoT data as attributes of entities representing devices and we can also send commands to devices by updating command-related attributes, providing we have access rights for that operation. The IoT agents (as the MQTT IoT agent we use) stay at the southbound of the Orion Context Broker and they are used by IoT integrators to connect devices in this scenario. IoT Agents support several IoT protocols with a modular architecture. Therefore, we had to determine first which protocol we will use to connect our devices. In our case we need to translate the MQTT protocol to the NGSI format, so we chose the JSON/MQTT IoT Agent which is described in the previous section.

---

<sup>44</sup> <https://catalogue-server.fiware.org/enablers/publishsubscribe-context-broker-orion-context-broker>

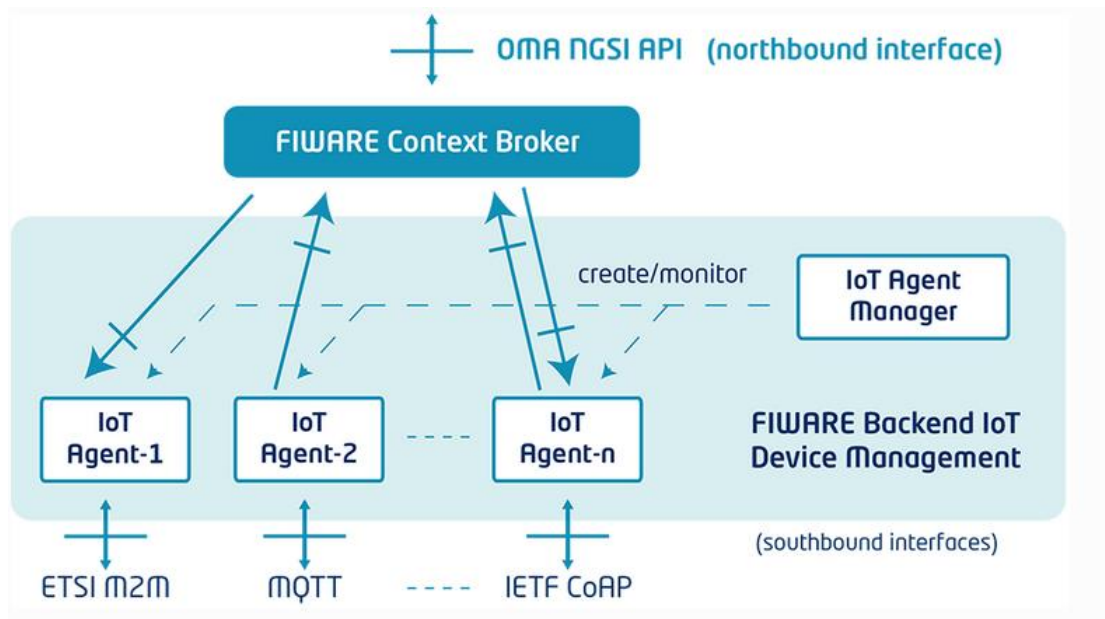


Figure 38: Fiware Reference Architecture for interconnecting IoT devices with Context broker

When we register a device to the IoT agent, a new entity is created into the Context Broker. Then, we can make a simple HTTP GET request to the context broker in order to read measures captured from our IoT devices. Here is an example of a GET request and response by the Context Broker in order to read the temperature value from the Multisensor1:

#### REQUEST:

```
GET
'http://147.27.60.202:1026/v2/entities/Multisensor1?attrs=temperature
'
```

#### RESPONSE:

```
{
  "id": "Multisensor1",
  "type": "sensor",
  "temperature": {
    "type": "number",
    "value": "27.4",
    "metadata": {
      "TimeInstant": {
        "type": "ISO8601",
```



```

"value":"2018-05-20T20:52:25.768Z"
    }
  }
}

```

We deployed the Context Broker on the same VM with JSON/MQTT IoT Agent using docker. We downloaded images for Orion Context Broker and MongoDB from the public repository of images called Docker Hub. Then we have created two containers based on both images.

#### 4.4.7 Keyrock Identity Management<sup>45</sup>

In order to implement OAuth2 authentication in our application we use our account on Fiware Lab. In order to create an account we visit the url: <https://account.lab.fiware.org/> (Figure 39)

Figure 39: Fiware lab welcome page

After following the instructions filling our data, we receive a confirmation mail. Once we have an account we can start creating organizations and applications.

The first step to start managing authorization in our application is to register the application in FIWARE Account. In order to do that we have to click on “Register” option of the Account Portal (Figure 40). The

<sup>45</sup> <https://catalogue-server.fiware.org/enablers/identity-management-keyrock>

“Home Sensors Monitoring” application is the app we have developed for demonstration purposes of our LoRaWare Architecture.

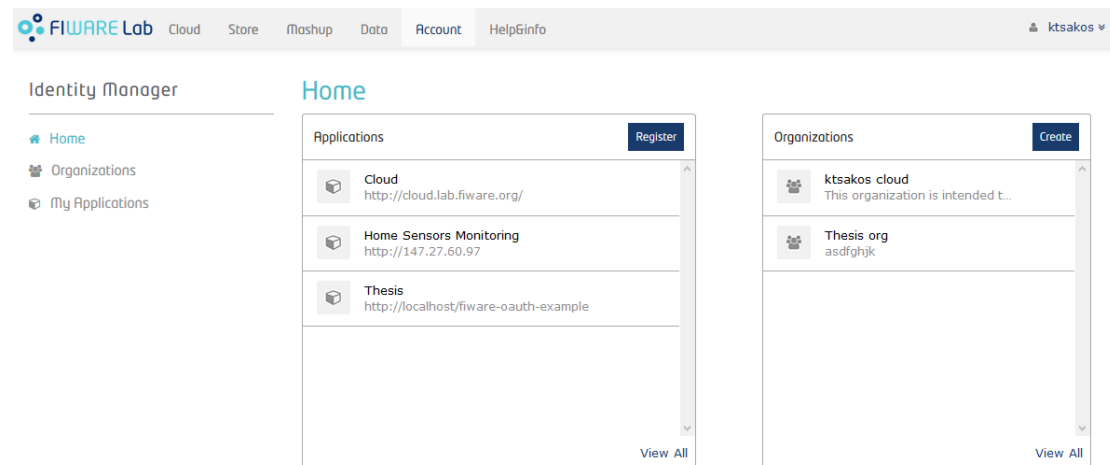


Figure 40: Fiware lab – Account Portal

Then we follow the steps with the data of our application. Once registered, we have to implement OAuth2 protocol in our application. The message flow between our web application and IDM account server is represented below (Figure 41):

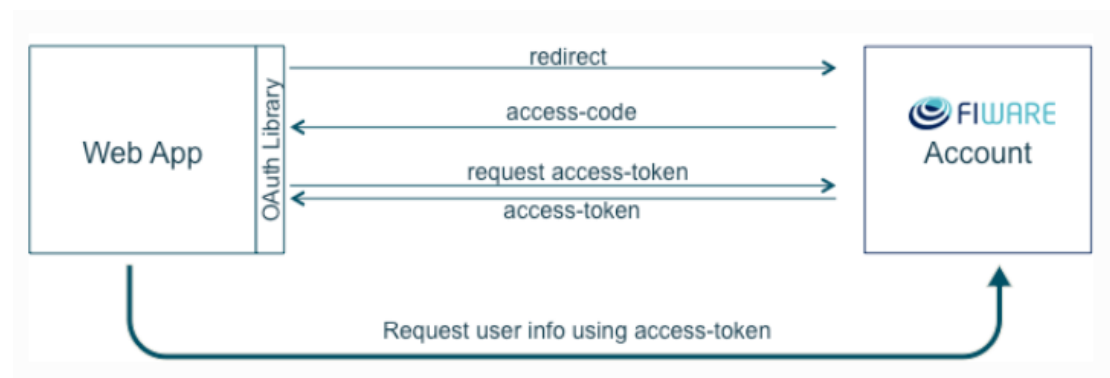


Figure 41: Message flow for OAUTH2 authentication using Keyrock Identity Management

In order to implement this flow we used the curl library of PHP.

We redirect to the fiware portal <https://account.lab.fiware.org/oauth2/authorize> in order to sign in and after the authentication an access code is returned.

After the callback, we make a HTTP GET request with the access code in its payload at the <https://account.lab.fiware.org/oauth2/token> url. This request requires also the OAuth2 credentials (ClientID and Client Secret)

which we can find in our Fiware Account Portal during the registration procedure of our application (Figure 42):

The screenshot shows the 'FiWARE Lab' account portal. The 'Account' tab is selected, displaying the 'Home Sensors Monitoring' application details. The 'OAuth2 Credentials' section is expanded, showing the following information:

- Description:** An application which provides statistical and current measures of temperature and humidity.
- URL:** http://147.27.60.97
- Callback URL:** http://147.27.60.97/callback.php
- Client ID:** 9b849804e7b9420db52ae5d0ff7b76e0
- Client Secret:** 16f8b24b663d4cbfb96d2f044f440890

Below the credentials, there are sections for 'PEP Proxy', 'IoT Sensors', 'Authorized Users', and 'Authorized Organizations', each with a filter input and an 'Authorize' button.

Figure 42: OAuth2 credentials can be found in the Fiware Account Portal after register an application

The response of our request is an access token. We make another request at the url: [http://account.lab.fiware.org/user?access\\_token=](http://account.lab.fiware.org/user?access_token=) using the access token as an argument to request the user's data (Figure 43).

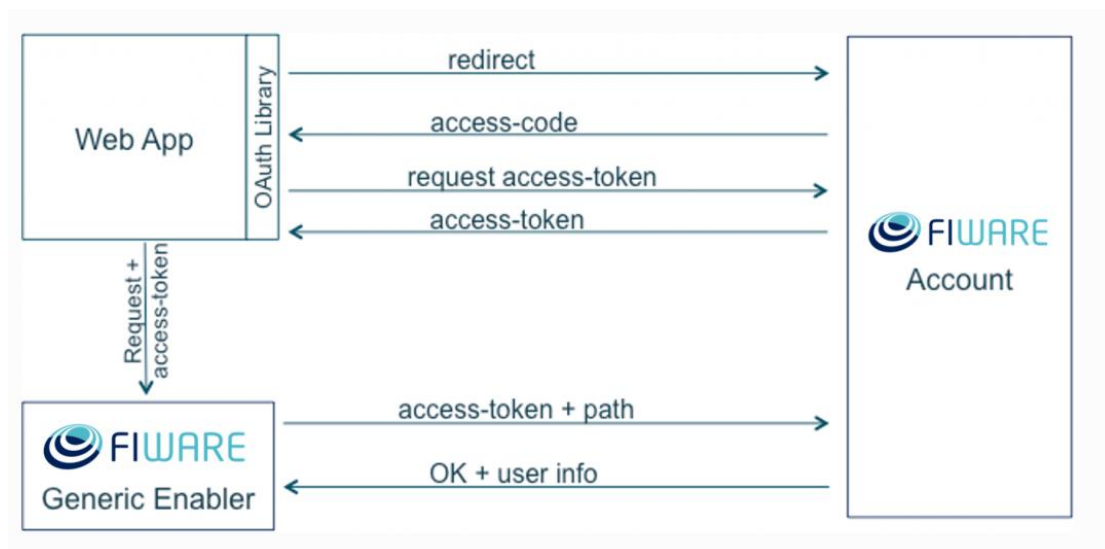


Figure 43: Message flow for OAuth2 authentication and user info request after successful authentication

The whole procedure we described above is part of our app logic which we are going to describe further in the next section.

## 4.5 Application Logic & Smart Home Web Application

We have developed a web application using the LoRaWare Architecture. Our system is distributed on four virtual machines with 2GB RAM, 1VCPU and 20GB storage (LoRa Server, Fiware GEs, App Logic and Storage) and it uses also a public instance of the Keyrock Identity Manager GE. With the Web app we can monitor in real time the humidity and temperature measures from two sensors. Furthermore, we can be informed about the last 10 different temperature and humidity measures by each sensor which are updated without refreshing the web page.

The application logic is the central component of our architecture. It is the part of our own code where happens all the communication between the services and their orchestration. We have used the PHP server-side programming languages in order to develop the operations of our application and the REST calls using a PHP library called "CURL".

The first operation in the app logic is the repeated REST calls to the Context Broker in order to be informed about the values of temperature and humidity that our sensors measure. Then every time a value changes than the previous one (the last one we stored) , it is stored on a database. We have created a MySQL database which is distributed on a different VM. The database contains two tables, the one for the Temperature measures and the other for the Humidity measures. Every time a new different measure is published in the Context Broker, we store it in the database with device id, value and date as columns. We have two php scripts which are running repeatedly as daemons on our VM, because it is important for this procedure to happen either we access the Web App or Not.

Then we can make every query we want to the database in order to come to statistical conclusions or process the data. We created a

distributed database on a different VM, as we can give privileges to other applications or users to have access to our sensors measures.

Furthermore, in the app logic we make the calls to the Keyrock Identity Manager public instance in order to implement the OAuth2 authentication protocol into our application as we described in section 3.4.7.

Finally, we developed a web application (part of the app logic into the same virtual machine) for demonstration purposes. It constitutes an example about how the LoRaWare architecture could be useful. Our web application's User Interface created using HTML and CSS technologies. Furthermore, it is running on the Apache server which supports the development with PHP.

Binding the 147.27.60.97 endpoint from a browser we visit our application's welcome page (Figure 44):

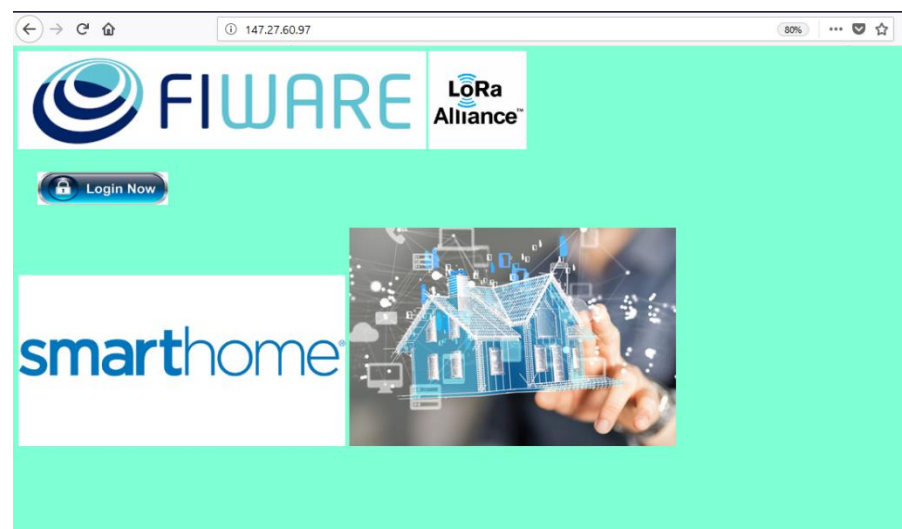


Figure 44: Application's welcome page for real time measures monitoring

Then we click on the "login now" image. We are redirected to the Fiware Lab in order to give our Fiware's account email and password (Figure 45):

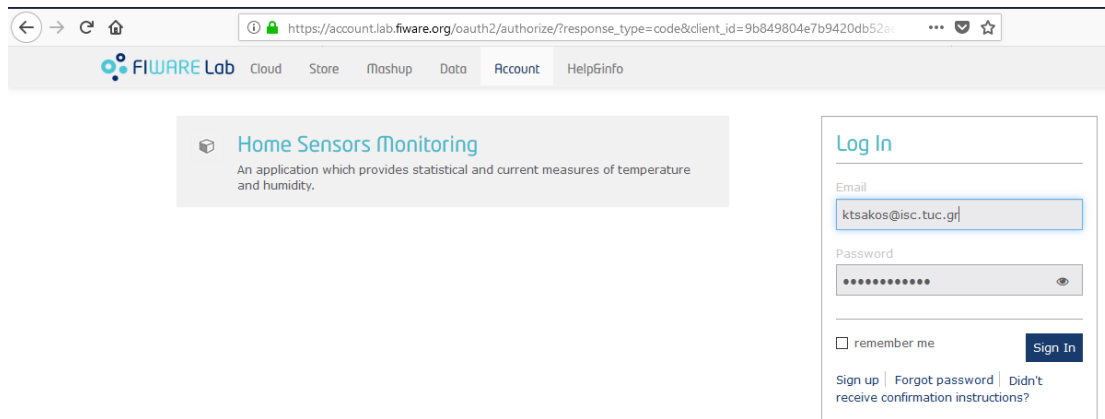


Figure 45: Redirection to Fiware lab page for OAuth2 authentication and access to the application

After the successful authentication we log in the user.php page (Figure 46) where we can monitor our two sensors current measures. Every time a new measure is stored in the database, the UI is updated with AJAX calls. In this page we see the last new different measure and the date and time it was published into the Context Broker:



Figure 46: Web page of the application – User info, log out link, Temperature & Humidity statistics links and last measured values with data and time of temperature and humidity by each sensor.

We have also the option to log out or be informed about the Temperature and Humidity statistics.

Clicking on the Temperature statistics we visit a different page of our application (Figure 47). There we can see the last 10 different Temperature statistics of the two sensors and also relative charts about them sorted by date:

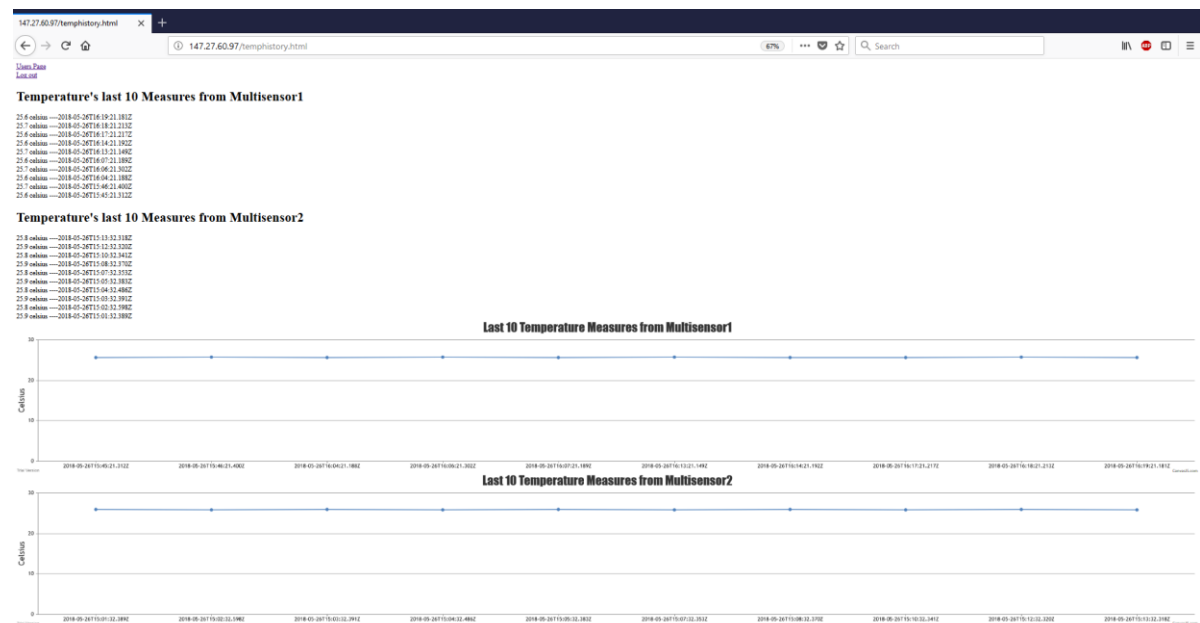


Figure 47: Temperature statistics – Last 10 different temperature measures of each sensor represented on a list and on charts descending sorted by date

The Humidity Statistics link represents the same information but for the Humidity measures of our two sensors (Figure 48):

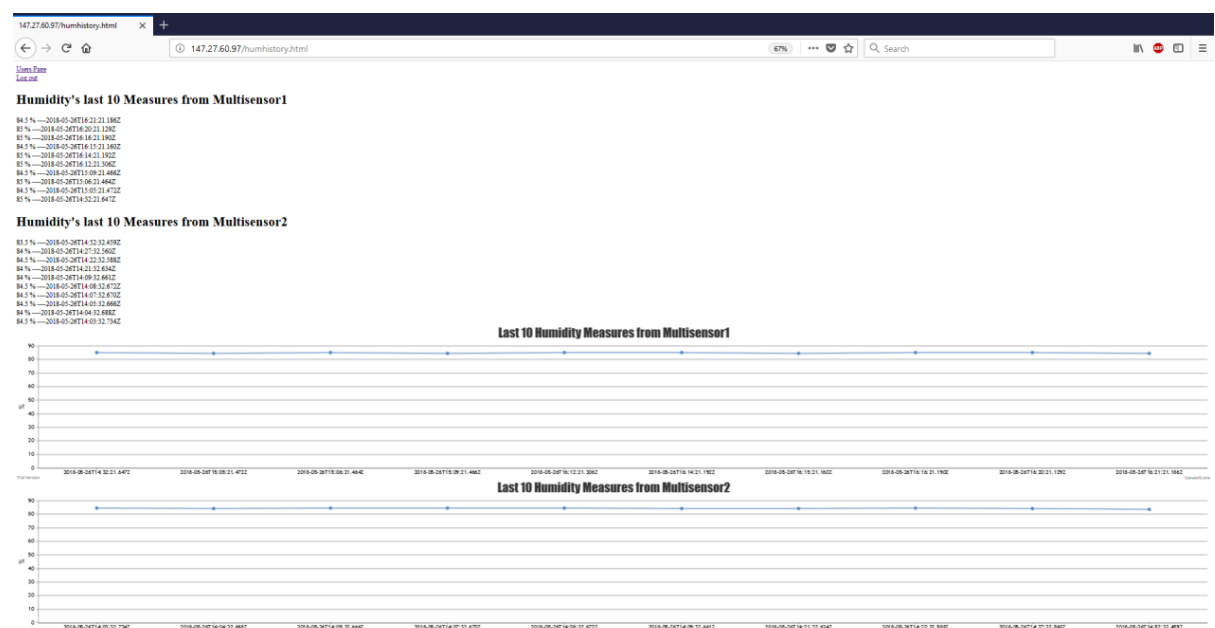


Figure 48: Humidity statistics – Last 10 different humidity measures of each sensor represented on a list and on charts descending sorted by date

Both the statistical measures and the charts are updated automatically with AJAX calls every time a new different measure is stored in the database. In this way, we can monitor the Temperature & Humidity measures without refreshing the web pages of our application.

## **5. Performance Evaluation**

In the following section we run an exhaustive set of experiments whose purpose is to evaluate the scalability of the back-end system and also the LoRa applicability in a complicated environmental terrain in the city of Chania. In the first part we stress the system by issuing many concurrent requests simulating the workload of a large IoT network with thousands of LoRa sensors. We measure the overall system response time (i.e. from the time a measurement is received by the cloud to the time it is stored in the database), the average response time of each service in the service sequence and also, the average workload (i.e. CPU, memory usage) of each running VM. In the second part, we attempt to study the long range characteristics of the LoRa network and how the error rate (percentage of transmitted but not captured by a gateway LoRa packages ) depends with the distance of the LoRa node from a gateway.

### **5.1 Evaluation of the Cloud Infrastructure**

For the implementation of our architecture we use four VMs which are running on the Fiware Lab Infrastructure. The first VM (Floating IP:147.27.60.211) contains a Mosquitto MQTT Broker, the Lora Server and the LoRa App Server services. The second VM (Floating IP:147.27.60.202) contains another Mosquitto MQTT Broker, the JSON filtering component, the MQTT IoT Agent and the Context Broker. The third VM (Floating IP:147.27.60.97) contains the App Logic and the Web Application which is served by the Apache Server. Finally we use a fourth VM (Floating IP:147.27.60.202), where we store all the data of our devices into a MySQL database. Every instance has a 20GB disk, 2GB



RAM and 1 VCPU. On the three instances is running an Ubuntu 16.04 operating system, except of the second VM(147.27.60.202) where a Centos 7 operating system is running due to the Context Broker's requirements.

In the first experiment, we compute an average time in which the packet payload is published into the mqtt broker in the first VM until it is stored in the database. Via ssh, we ran the command "tcpdump" into the first VM which give us the ability to check the packet flows to the port 1883, which is the port of the MQTT Broker. Then every time a payload is published there we can take the timestamp information. Simultaneously, we ran the same command in the second VM to check the packet flow to the port 1026 where the Context Broker sends its payload. The time after the subtraction of the two timestamps was 47,3344ms. This is the average time of the payload transmission from the MQTT broker to the Context Broker. We also calculated the time from the Context Broker response until the values were stored into the MySQL database. This time was 140ms(10ms for the REST calls to the Context Broker,30 ms to read from the database and 100 ms to write in the database). This is the response time of the App Logic and the insertion into the database. So the average time of a packet from the time it is sent to the Cloud until it is stored to the database is  $47,3344+140=187,3344$  ms.

After this calculation, we calculated also the average response time of every component of our architecture in order to come to other conclusions.

The services which have an endpoint were tested with the Apache Bench tool. We made 2000 request at the endpoint of every service and we took the average time of every response to be answered. The JSON Filtering component was measured with the "time" library of python and the difference between the times in the start and in the end of the script. In a same way, we calculate the App Logic response time using the "microtime" method of PHP.

The following diagram represents the average time of every service to answer a request (Figure 49).

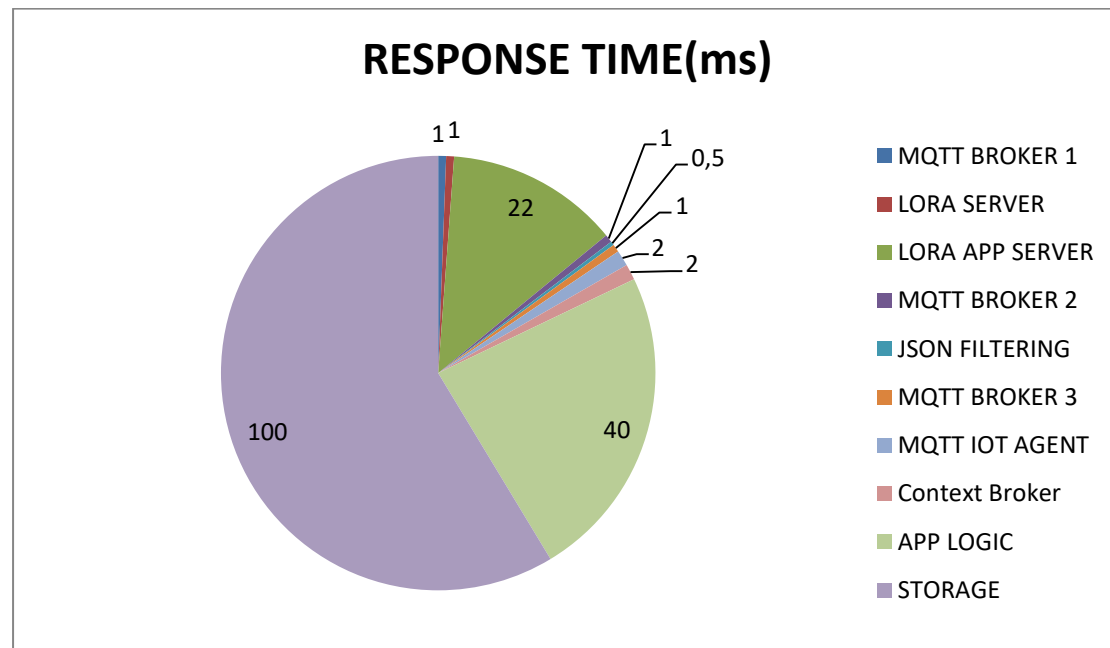


Figure 49: Pie representing the average response time in milliseconds of each LoRaWare's service to serve one request

Essentially, the lora-gateway-bridge running on the gateway publishes the data into the MQTT Broker 1. The LoRa Server is subscribed to the specific topic and takes the data(1 ms – 1%). After the decoding and deduplication of the packet (1 ms – 1 %) the payload is forwarded to the LoRa App Server. After the decryption, the payload is published to the MQTT Broker2 (22ms – 13%). Then the JSON Filtering is parsing the json payload and republish the data to the mqtt broker but on a different topic (0,5ms-0%). Then the IoT Agent is subscribed to the MQTT Broker and takes the data (1ms-1%). The IoT Agent converts the data from MQTT to the NGSI protocol and forwards them to the Context Broker (2ms-1%). The Context Broker takes the NGSI payload and updates the value of an entity (2ms-1%). Finally, in the app logic a get request is done to the Context Broker. If the measure is different from the last received, then this payload is stored to the database (40ms+100ms-82%).

Adding all the above times of every operation the sum is 170,5 ms.

The difference between the whole average time and the sum of every response time independently is the network's latency which is 187,3344-

170,5=16,8344ms. We expected such a short time as the VMs are running on the same Cloud Infrastructure.

As a second experiment for the back end evaluation, we used the Apache Bench tool in order to stress the services which have an endpoint to be accessed. Apache Bench gives us the ability to make a massive number of requests and declare both the number of them and the number of the requests which must be served concurrently. Then it returns the average response time for the requests to be served. We make 2000 request and we increase the c parameter which represents the number of concurrent requests which must be served. Simultaneously, we check the CPU and the Memory usage of the VM on which every service is deployed using the HTOP tool. The HTOP is a lightweight program for resources management and is executed from the command line. It gives us the opportunity to monitor in real time the resources consumption per process or in total.

Below there are indicative tables about the above experiments. For every service there is one table for the resources usage relatively with the “c” parameter and one table for the response time (in milliseconds) relatively with the “c” parameter and the percentage of requests which were served. Time per request (mean) tells us the average amount of time it took for a concurrent group of requests to process. Time per request (mean, across all concurrent requests) tells us the average amount of time it took for a single request to process by itself.

- 2000 requests to the MQTT Broker(147.27.60.211:1883)

	C=1	C=50	C=100	C=150	C=300
CPU(%)	35,1	42,4	45,3	52	56,7
MEM(MB)	228	232	239	241	253

	C=1	C=50	C=100	C=150	C=300
50%	1	16	34	50	110
66%	1	16	35	52	117
75%	1	16	36	52	121
80%	1	16	36	53	122

90%	1	17	37	54	122
95%	1	17	38	55	123
98%	1	18	39	55	124
99%	1	20	40	56	127
100%	5	21	40	57	129
Time per request	0,939	16,026	35,413	53,751	124,049
Time per request (across all concurrent requests)	0,939	0,321	0,354	0,358	0,413

We consider that the results of the above table would be the same for the MQTT broker of the second VM.

- 2000 requests to the LoRa App Server (147.27.60.211:8080)

	C=1	C=50	C=100	C=150	C=300
CPU(%)	22,7	100	100	100	100
MEM(MB)	247	252	257	263	277

	C=1	C=50	C=100	C=150	C=300
50%	22	1002	2020	2942	5866
66%	22	1094	2216	3299	6797
75%	22	1146	2325	3428	6910
80%	23	1170	2349	3542	7094
90%	24	1282	2596	3958	7812
95%	25	1373	2822	4120	7949
98%	26	1460	2922	4336	8240
99%	28	1513	2990	4572	8860
100%	34	1610	3375	5263	10261
Time per request	21,838	1018,013	2065,020	3119,216	6522,003
Time per request (across all concurrent requests)	21,838	20,360	20,650	20,795	21,740
Transfer Rate (Kbytes/sec)	23,21	24,89	24,54	24,37	23,31

- 2000 requests to the JSON/MQTT IoT Agent (147.27.60.202:4041)

	C=1	C=50	C=100	C=150	C=300
CPU(%)	45,2	55,3	57,7	66,4	100

MEM(MB)	570	585	593	596	604
---------	-----	-----	-----	-----	-----

	C=1	C=50	C=100	C=150	C=300
50%	4	20	7	16	120
66%	4	52	42	84	226
75%	4	74	76	176	357
80%	5	81	102	207	401
90%	5	105	1003	1004	1008
95%	7	130	1011	1047	1140
98%	11	1010	3010	3013	1236
99%	1007	2013	3013	3016	2402
100%	3013	3017	5374	9132	3017
Time per request	16,024	122,149	397,436	699,308	562,916
Time per request (across all concurrent requests)	16,024	2,443	3,974	4,662	1,876
Transfer Rate (Kbytes/sec)	26,75	175,89	108,11	92,17	229

- 2000 requests to the Context Broker (147.27.60.202:1026)

	C=1	C=50	C=100	C=150	C=300
CPU(%)	30,9	31,9	34,3	39,1	58,1
MEM(MB)	551	558	560	574	584

	C=1	C=50	C=100	C=150	C=300
50%	2	2	4	10	60
66%	3	3	21	42	77
75%	3	4	37	62	90
80%	3	6	47	70	106
90%	4	20	67	1003	1005
95%	5	1004	1004	1015	1070
98%	7	2329	3006	3008	1089
99%	1003	3009	3008	3009	1099
100%	1008	7022	6892	4413	6643
Time per request	13,024	201,628	513,774	341,110	1038,385
Time per request (across all concurrent requests)	13,024	4,033	5,138	2,274	3,461

Transfer Rate (Kbytes/sec)	19,64	63,45	49,80	112,51	73,92
-------------------------------	-------	-------	-------	--------	-------

- 2000 requests to the Apache Server of the Web App(147.27.60.97)

	C=1	C=50	C=100	C=150	C=300
CPU(%)	22,1	29,8	37,1	33,3	44,8
MEM(MB)	220	228	234	237	252

	C=1	C=50	C=100	C=150	C=300
50%	2	2	6	21	56
66%	2	3	15	30	70
75%	2	6	30	43	79
80%	2	9	39	51	111
90%	3	23	69	69	316
95%	3	1002	1008	1013	1003
98%	4	3004	3004	1022	5601
99%	999	3099	3012	1030	5605
100%	3032	7017	12417	21626	23822
Time per request	14,528	235,633	630,252	1622,485	3574,786
Time per request (across all concurrent requests)	14,528	4,713	6,303	10,817	11,916
Transfer Rate (Kbytes/sec)	61,57	189,81	141,93	82,70	75,07

The following tables conclude the results for the three VMS (147.27.60.211, 147.27.60.202, 147.27.60.97). More specifically, we represent the usage of the computing resources and the average time for 1 request to be served from all services of every VM in comparison with the number of the requests which must be served concurrently. We added the response times per request across all concurrent requests for every service per VM and we show the ranges of the demanding computing resources. The fourth VM contains only the MySQL database which practically cannot be stressed with apache bench. If the time per request (across all concurrent requests) is increased when we increase the concurrent requests, it means that the VM scales up in order to serve them.

➤ VM 147.27.60.211 (MQTT BROKER-LORA APP SERVER)

	C=1	C=50	C=100	C=150	C=300
Time per request to be served(across all concurrent requests) (ms)	22,777	20,681	21,004	21,153	22,153
MEM(MB)	228-247	232-252	239-257	242-263	253-277
CPU(%)	22,7-35,1	42,4-100	45,3-100	52-100	56,7-100

➤ VM 147.27.60.202 (MQTT BROKER-JSON/MQTT IoT Agent – Context Broker)

	C=1	C=50	C=100	C=150	C=300
Time per request to be served(across all concurrent requests) (ms)	29,987	6,797	9,466	7,294	5,75
MEM(MB)	228-570	232-585	239-593	241-596	253-604
CPU(%)	30,9-45,2	31,9-55,3	34,3-57,7	39,1-66,4	56,7-100

➤ VM 147.27.60.97 (Apache server for the Web Page of the Web App)

	C=1	C=50	C=100	C=150	C=300
Time per request to be served(across all concurrent requests) (ms)	14,528	4,713	6,303	10,817	11,916
MEM(MB)	247	252	257	263	273
CPU(%)	22,7	100	100	100	100

## 5.2 LoRa Network Evaluation

According to LoRa specifications, there are transmission parameters which could influence the performance of the network. These include parameters such as bitrate, resistance to interference noise, ease of decoding or energy consumption and can be set at system set-up.

In order to be understood, **chirp** is a sinusoidal signal in which the frequency increases or decreases over time. LoRa is a chirp spread spectrum technique, so chirp pulses are used to encode information (symbols). Chirp spread spectrum uses its entire allocated bandwidth to broadcast a signal, making it robust to channel noise.

**Transmission Power** on a LoRa radio can be adjusted from -4 dBm to 20 dBm, in 1 dB steps, but because of hardware implementation limits, the range is often limited to 2 dBm to 20 dBm. In addition, because of hardware limitations, power levels higher than 17 dBm can only be used on a 1% duty cycle.

**Carrier Frequency** is the center frequency that can be programmed in steps of 61 Hz between 137 MHz to 1020 MHz. Depending on the particular LoRa chirp, this range may be limited between 860 MHz and 1020 MHz.

**Spreading Factor (SF)** is the ratio between the symbol rate ( $BW/2^{SF}$ ) and chirp rate (BW). A higher spreading factor increases the Signal to Noise Ratio (SNR), and thus sensitivity and range, but also increases the airtime of the packet (the LoRa radio module needs more time to send the same amount of data). **Sensitivity** is the minimum magnitude of input signal required to produce a specified output signal having a specified signal-to-noise ratio. The number of chirps per symbol is calculated as  $2^{SF}$ . For example, with an SF of 12 (SF12) 4096 chirps/symbol are used. Each increase in SF halves the transmission rate and, hence, doubles transmission duration and ultimately energy consumption. Spreading factor can be selected from 6 to 12.

**Bandwidth (BW)** is the width of frequencies in the transmission band. Higher BW gives a higher data rate (thus shorter time on air), but a lower sensitivity (because of integration of additional noise). A lower BW gives a higher sensitivity, but a lower data rate. Lower BW also requires more accurate crystals (less ppm). Data is sent out at a chirp rate equal to the bandwidth; a bandwidth of 125 kHz corresponds to a chirp rate of 125 kcps. Although the bandwidth can be selected in a range of 7.8 kHz to



500 kHz, a typical LoRa network operates at 500 kHz, 250 kHz or 125 kHz (resp. BW500, BW250 and BW125)

Finally, LoRa includes a forward error correction code. The **Code Rate (CR)** equals  $4/(4 + n)$ , with  $n \in \{1, 2, 3, 4\}$ . In telecommunication and information theory, the code rate (or information rate) of a forward error correction code is the proportion of the data-stream that is useful (non-redundant). That is, if the code rate is  $k/n$ , for every  $k$  bits of useful information, the coder generates a total of  $n$  bits of data, of which  $n-k$  are redundant.

Taking this into account, as well as the fact that  $SF$  bits of information are transmitted per symbol, the following equation allows one to compute the useful bit rate ( $R_b$ ).

$$R_b = SF \times \frac{BW}{2^{SF}} \times CR$$

For example, a setting with  $BW = 125$  kHz,  $SF = 7$ ,  $CR = 4/5$  gives a bit rate of  $R_b = 5.5$  kbps.

Generally speaking, an increase of bandwidth lowers the receiver sensitivity, whereas an increase of the spreading factor increases the receiver sensitivity. Decreasing the code rate helps reduce the Packet Error Rate (PER) in the presence of short bursts of interference, i.e., a packet transmitted with a code rate of  $4/8$  will be more tolerant to interference than a signal transmitted with a code rate of  $4/5$ .

In order to evaluate our LoRa Network we used two Lorank8v1 Gateways of Ideetron and two LoRa Nodes (the combination of Ideetron's Nexus Board and Nexus Demoboard). We have already described these devices in section 3.2.

### 5.2.1 First Experiment

As a first experiment we had two Lorank8v1 gateways located in two different areas in Chania Crete and a LoRa Node moving between them.

From the Google Maps we see the two points on the Map where every Gateway was placed.

The one Gateway was placed on a urban area (Figure 50):

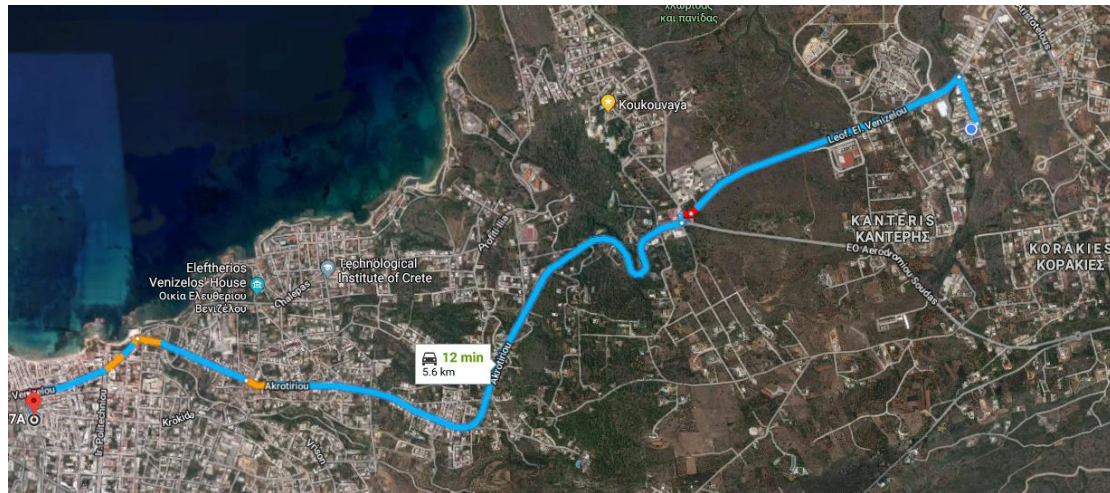


Figure 50: Route of the experiment and the red point where the one gateway was placed (urban area)

The second Gateway was placed on a semi urban area (Figure 51):

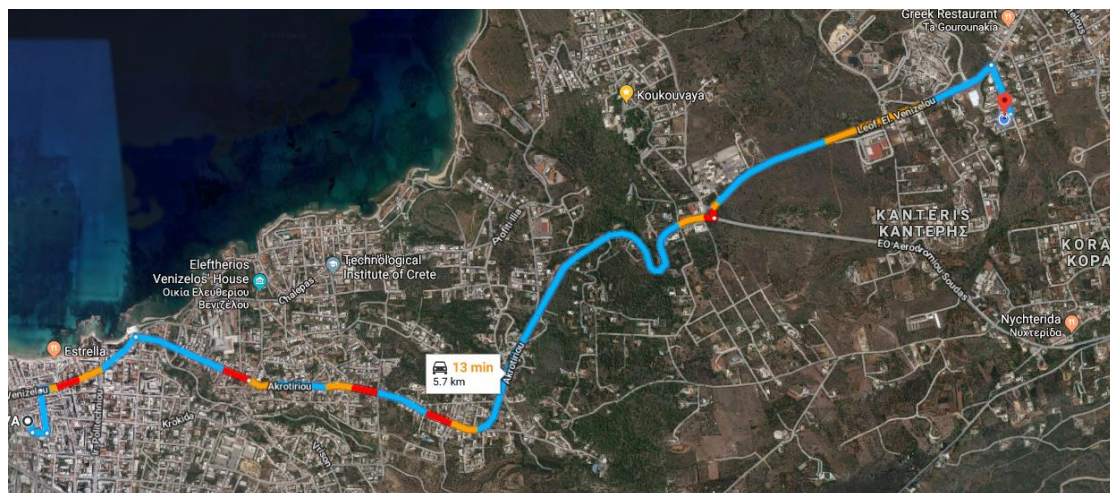


Figure 51: Route of the experiment and the red point where the one gateway was places (semi urban area)

We configure one of our LoRa Nodes suitably in order to succeed the best range we could, ignoring the power consumption. More specifically, we declare from the Arduino sketches the Spread factor as 12, the bandwidth as 125kHz and the Transmission Power at 17 dbm. These are the maximum values in order to succeed the best range we can. However the Code Rate was by default set at 4/5. Theoretically, we

wanted to configure it at 4/8, but we couldn't change this parameter in the code as it is fixed.

We drove in a route from the one Gateway to the other and back. The sensor was sending one package per minute (Class A device) and the gateways were writing the metadata and the physical payload of every received packet into a .csv file. This procedure was run by executing an Ideetron's test script on every gateway called "util\_pkt\_logger".

After processing the data of the two .csv files (one for every gateway), two diagrams (one for every gateway) were exported:

The following diagrams show the SNR (Sound to Noise Ratio) and the RSSI (Received Signal Strength Indicator) values of the successfully captured packets in comparison with the distance between the transmission point and the gateway (base station)

The first diagram shows the results taken from the Gateway on the Semi Urban Area (Figure 52):

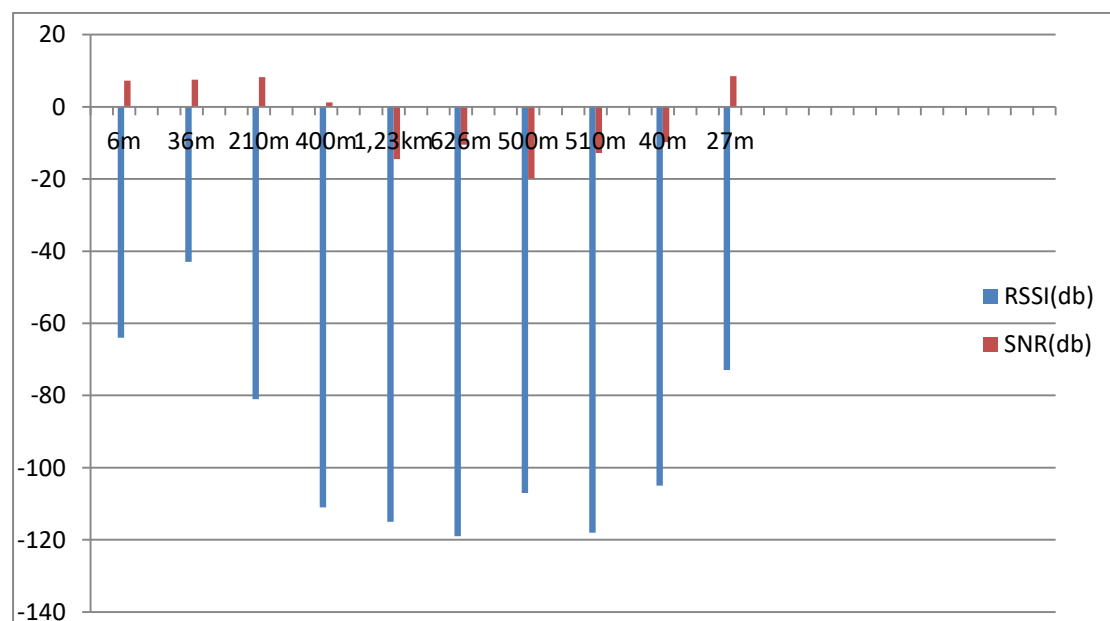


Figure 52: RSSI & SNR values in db of the successfully captured packets by the Gateway on the Semi Urban Area

Totally, 50 packets were transmitted. From them only 10 were captured. The longest transmission was happened 4,37 km far from the gateway. However, the longest transmission which successfully captured from the gateway was 1,23 km far from the gateway. Generally, 20% of the

transmissions were captured successfully. We can also notice that the nearest we are to the Gateway, the highest RSSI and SNR we take in db.

Speaking about the Gateway on the Urban Area the results were relatively worse and are presented in the following diagram (Figure 53):

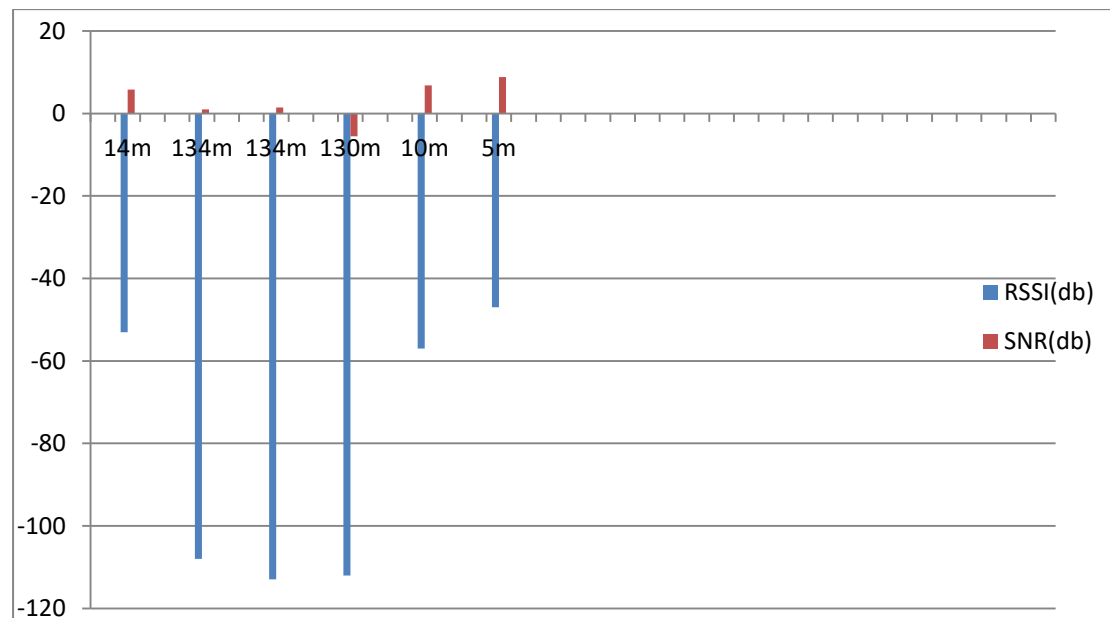


Figure 53: RSSI & SNR values in db of the successfully captured packets by the Gateway on the Urban Area

Totally, 27 packets were transmitted. From them only 6 were captured. The longest transmission was happened 4,37 km far from the gateway. However, the longest transmission which successfully captured from the gateway was only 134m far from the gateway. Generally, 22,22% of the transmission were captured successfully.

In conclusion, the results from the first experiment were relatively different from the LoRa's expectations. This could be due to the noise interference of the area. Furthermore, the elevation of the gateways plays a great role to the expected range. We could have taken better results if we have placed the gateways on a high point with free sight similar to this where cellular antennas' are placed. In addition the environment of route was not an open terrain, but there were buildings, metal structures and a non-uniform ground with many obstacles. Finally, we could expect better results, in case we could also decrease the Code Rate parameter and succeed a smaller bit rate.



### 5.2.2 Second Experiment

In the second experiment we used one gateway of the same model as in the first experiment. We also used two LoRa Nodes of the same type. This time we conduct the experiment by walking in case the speed of the car influenced the performance in the first experiment. The nodes were configured similarly with Spread Factor equal to 12, Bandwidth equal to 125KHz, the Transmission Power equal to 17dbm and the Code Rate equal to 4/5. Every node was sending again one packet per minute (Class A device).

Below is the route we walked with the two sensors and the gateway's location (Figure 54):

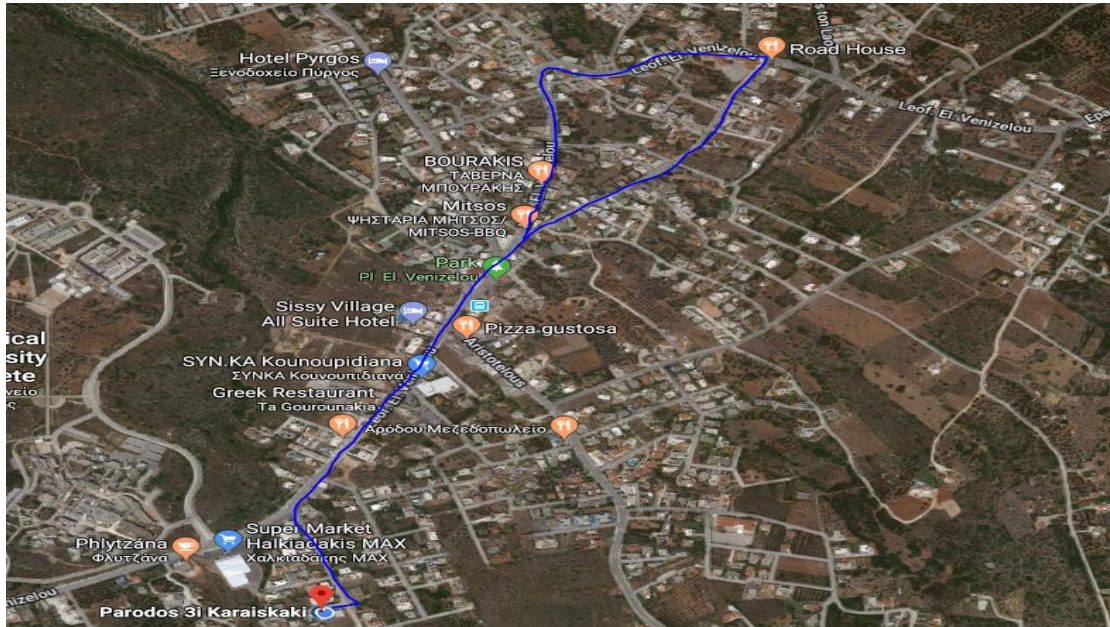


Figure 54: Route of the second experiment and the red point where the Gateway was placed

We executed again the “util\_pkt\_logger” script on the gateway and from the .csv file we took the following diagram (Figure 55) which presents the RSSI and SNR values in comparison with the distance from the point of successfully captured transmission to the gateway's location:

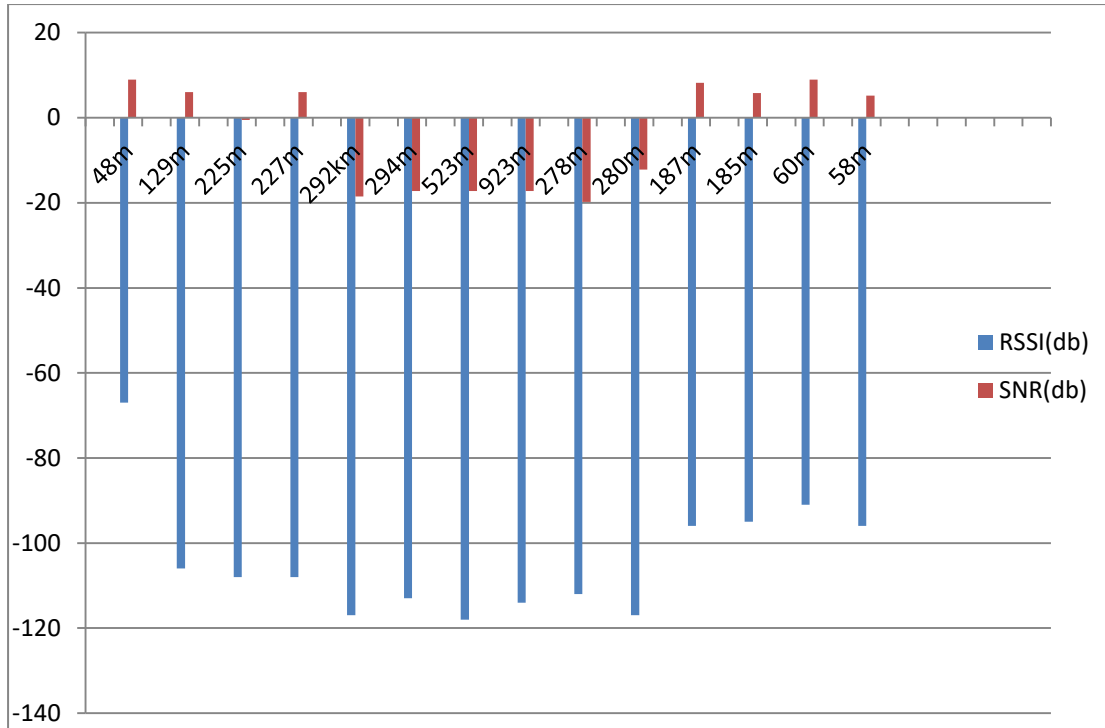


Figure 55: RSSI & SNR values in db of the successfully received packets by the Gateway

Totally, 74 packets were transmitted (37 per node). From them only 14 were captured. The longest transmission was happened 1,36 km far from the gateway. However, the longest transmission which successfully captured from the gateway was 923m far from the gateway. Generally, 18,92% of the transmission were captured successfully. We conclude that there are no differences in the results from the previous experiment although we changed the rate we moved. However besides all the reasons of the first experiment, we had also placed the gateway inside a building which may also have influenced the performance.

In conclusion, the results from the second experiment were also relatively different from the LoRa's expectations. This could be due to the noise interference of the area. Furthermore, the placement of the gateway inside a building with a very small elevation had also a great impact on the performance. In addition the environment of route was not an open terrain, but there were buildings, metal structures and a non-uniform ground with many obstacles. Finally, we could expect better results, in case we could also decrease the Code Rate parameter and succeed a smaller bit rate.

Concluding the results from the two experiments about the LoRa Network we can present the estimated percentage of the successfully received packets in comparison with the distance from the base station (Figure 56). We notice that only in the first 150m we have a 100% success of transmission in the Urban Area and in the first 300m in the Semi Urban Area. The percentage is reduced with a bigger rate in the Urban Area. After 1km we have a 0% success in the urban Area and after 1,5 km a 0% success in the Semi Urban Area. We could improve these results for both areas if we placed the Gateways on a higher elevation with a uniform environment with an open terrain and free sight without obstacles in order to restrict the noise interferences.

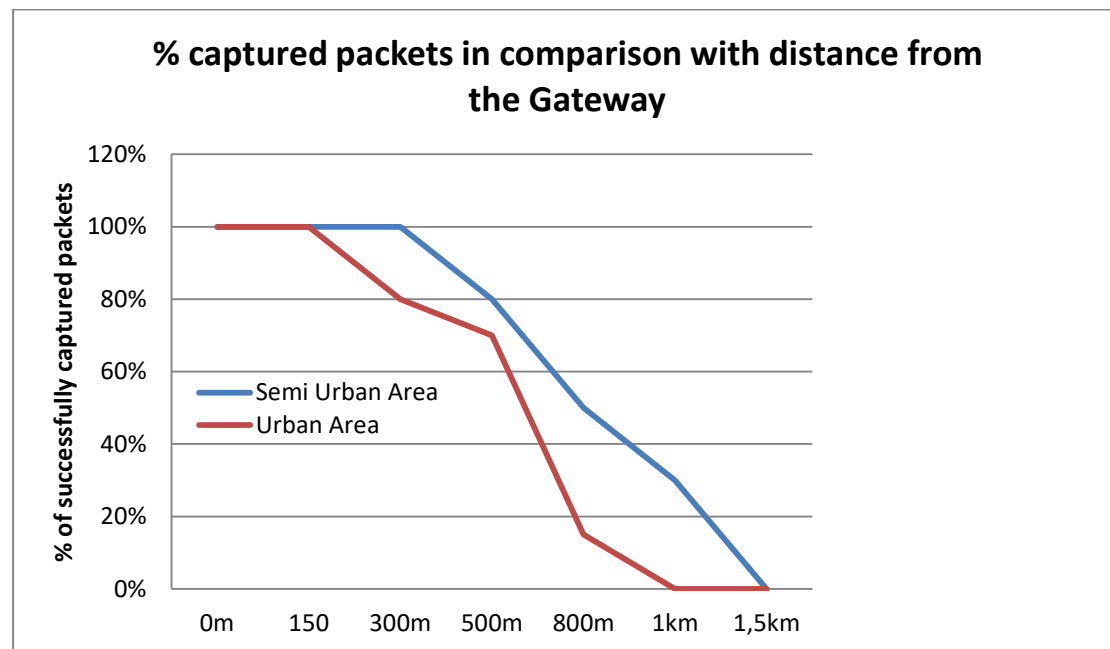


Figure 56: Estimated percentage of the successfully captured packets in comparison with the distance from the base station places on Urban and Semi Urban Area

## 6. Conclusion – Future Work

In this section we are going to present our conclusions about this thesis and suggest further expansions in order to improve the system's functionality and performance.

## 6.1 Conclusions

The main goal of this thesis was to design and develop a service oriented architecture in order to interconnect LoRa devices with the Cloud. During this project we came to the following conclusions:

- Cloud computing provides a big variety of services which can be very useful for the easy and fast development of applications. For example, the device management service (IoT Agent), the Context Broker and the Keyrock Identity Manager of Fiware constitute required services for the most applications of the IoT sector. Relative services are provided also from other Cloud providers.
- The usage of service oriented architectures and the RESTful Web Services facilitate the communication among services even if they are deployed on different cloud environments.
- With the virtualization we can easily start the development of our application and avoid incompatibility problems due to the hardware architecture or the physical resources.
- The MQTT is an open protocol which is ideal for constrained networks with low bandwidth, high latency, data limits and fragile connections. It is a publish/subscribe protocol which gives us the opportunity to check the packets which are published. Furthermore, it is secured as it runs over TCP and there is a variety of MQTT client libraries which are available at the most programming languages and could simplify the job of a developer.
- LoRa constitutes a constantly evolving technology. We can cover whole cities with LoRa devices using just a few gateways with low cost. The LoRa network performance depends on the morphology of the environment and the obstacles between the nodes and the base stations. Furthermore, the location and the elevation of the gateways can play a great role to deal with the interferences. Finally, transmission parameters of the nodes such as the spreading factor and the bandwidth have significant impact on the network coverage, as they affect the data rate. Generally, there is a trade-off between the power consumption and the long range of LoRa protocol. We need to find the suitable point in order to



consume the less power and have the range we want for our network's requirements. Finally, the small bit rate makes LoRa unsuitable for applications which require continuous and massive data transmissions.

## 6.2 Future Work

Below we describe a few plans for the future in order to improve the performance of our system, the coverage of the LoRa Network and to expand the functionality of the LoRaWare Architecture.

- As we have already noticed in the backend evaluation of our system, the applogic and the storage in the database covers most of the response time. In order to reduce this latency we would like to use a NoSQL database like the mongodb to store the persist data instead of MySQL database we used.
- Also, we would use better sensors and gateways in order to succeed a better range of the network. Furthermore, we would like to do more experiments about the network performance after placing the gateways at a high point (high elevation) with free sight (no obstacles) in order to approach the maximum range that the protocol could practically succeed.
- Our architecture contains the basic services for interconnecting LoRa devices to the Cloud. However our future goal is to expand this architecture with other services. For example, data analytics service will demonstrate functionality related to uncovering hidden patterns in data, unknown correlations, user preferences and useful business information (e.g. user's data may provide feedback for enhancing system functionality and users acceptance). COSMOS big data analysis GE or the Data Visualization –SpagoBI GE of Fiware could be used for such operations. In addition, an Event Processing module could be added to handle events (e.g. creates alarm notifications based on end-user conditions and information received from the sensors) and notify the Publish/Subscribe service, which is responsible for passing the information to the end-user. The Complex Event Processing (CEP) GE of Fiware is a reference implementation of

this service. Finally, a Mashup Service would allow application developers to compose new applications. This would not only take significantly less time to build an application, but also to minimize the effort required to maintain the system each time a device or service was added, removed or updated. Using services as IFTTT or Node-RED, devices can be integrated with modern Web applications and services with minimal effort (physical mashups). A Mashup Editor with similar capabilities is offered in Fiware, the WireCloud Mashup GE.

## 7. References

- [1] <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5038744/> A Study of LoRa: Long Range & Low Power Networks for the Internet of Things
- [2] LoRa Alliance - LoRaWAN Specification Authors: N. Sornin (Semtech), M. Luis (Semtech), T. Eirich (IBM), T. Kramp (IBM), O.Hersent (Actility)
- [3] LoRaWAN Network Server Demonstration: Gateway to Server Interface Definition
- [4] Ideetron Manual Nexus LoRaWAN low power
- [5] <https://pdfs.semanticscholar.org/d77f/f1693e1482e05e9c12df768a28b4e7fef759.pdf> LoRa Transmission Parameter Selection Author: Martin Bor, Utz Roedig
- [6] <https://www.fiware.org/developers/catalogue/>
- [7] <https://www.loraserver.io/>
- [8] <https://forum.loraserver.io/>
- [9] RIOT Reference Architecture

## 8. Image References

- [1][Figure 1] <https://rajivramachandran.wordpress.com/2012/06/19/cloud-service-models-iaas-vs-paas-vs-saas/>
- [2][Figure 2] [https://www.researchgate.net/figure/Cloud-deployment-model-fig2\\_260192916](https://www.researchgate.net/figure/Cloud-deployment-model-fig2_260192916)
- [3][Figure 3] <https://whatis.techtarget.com/definition/virtualization-architecture>
- [4][Figure 4] <https://en.wikipedia.org/wiki/Hypervisor>
- [5][Figure 5] <https://yourdailytech.com/storage-architecture/containers-dont-contain-the-whole-future/>
- [6][Figure 6] <https://www.theverge.com/circuitbreaker/2018/5/25/17386716/docker-kubernetes-containers-explained>
- [7][Figure 7] <https://www.openstack.org/>
- [8][Figure 8] <https://docs.openstack.org/security-guide/introduction/introduction-to-openstack.html>
- [9][Figure 9] <https://www.fiware.org/>
- [10][Figure 10] <https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/>
- [11][Figure 11,12,13,14] [https://www.mouser.com/pdfdocs/LoRaWAN101\\_final.pdf](https://www.mouser.com/pdfdocs/LoRaWAN101_final.pdf)
- [12] [Figure 15,17] RIOT Reference Architecture
- [13][Figure 16] <http://blog.janjongboom.com/2016/04/04/intro-to-lora.html>
- [14] [Figure 21,22,23,25] <https://webshop.ideetron.nl/>
- [15] [Figure 24] <https://github.com/myDevicesIoT/cayenne-docs/blob/master/docs/LORA.md>
- [16] [Figure 26] <https://www.loraserver.io/loraserver/overview/>
- [17][Figure 38] <https://fiwaretourguide.readthedocs.io/en/latest/connection-to-the-internet-of-things/introduction/>
- [18][Figure 41,43] <https://www.slideshare.net/alvaroalonsogonzalez/lesson-3-applications-how-to-create-oauth2-tokens>