POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Development of a scalable architecture for Industrial Internet of Things

Supervisor
prof. Andrea Acquaviva

Candidate
Daniele Magurano

A.A. 17/18

Contents

Listings

List of Figures

Chapter 1

Introduction

Over the last decades, the Internet has seen an extensive and rapid growth, starting from serving hundreds of hosts, to interconnecting billions of computers and mobile devices. The next step of this evolution is the Internet of Things (IoT) vision, a radical extension from the current Internet, to a future of interconnected physical *things*, allowing their transition from traditional to *smart*, revolutionizing their utility and application. This revolution is already happening around us: smart cities, smart homes and smart cars are just few of the possible applications of the IoT technology.

The increasing trend of IoT implementation is also happening in the industrial environment, driven by the countless number of potential benefits offered, ranging from the creation of novel business models, to the reduction of costs thanks to predictive maintenance. The adoption of IoT technologies into the industry is often referred to as Industrial Internet of Things (IIoT) or *Industry 4.0*, a concept introduced in 2011 by the German government in the strategic planning for their manufacturing industries [1], reflecting the idea that a new industrial revolution is happening right now. Despite the good potential, several issues are holding back the progress of the Industrial IoT.

It is envisioned that more than 20 billions of devices will be connected by the end of 2020 [2], thanks to the steady advances in electronic witnessed

in the last decades, which helped to reduce costs. This unprecedented number of devices resulted in a substantial fragmentation on many levels, with the current IoT scenario that consists in a plethora of different vendors and standards. Moreover, most large corporations have developed their own proprietary solutions in an independent way, often resulting in systems which are incompatible with each other.

According to a survey conducted in 2015 [3], the lack of interoperability standards is one of the greatest barriers inhibiting business from adopting the Industrial Internet, and many experts agree that solving the interoperability problem is the only way to achieve a true Internet of Things [4].

As an attempt to solve this interoperability issue, many standardization entities have proposed their solutions. Most notable ones are OPC Unified Architecture (OPC UA), oneM2M and LightWeight M2M (LWM2M) by Open Mobile Alliance (OMA). The last one represents our choice for this work and it will be described thoroughly in Chapter 2.

With the ever increasing number of devices and the enormous amount of data generated by those, concerns regarding scalability are arising. While the processing power offered by the Cloud Computing paradigm could potentially solve this issue, sending all the data to a remote endpoint may not always be the best solution. In fact, many IoT applications, especially in industrial automation, are often latency critical. A new paradigm, called Edge or Fog Computing is emerging, in which data is initially processed and filtered in gateways, which are located closer to the devices, and only eventually sent to the Cloud for further storage/analysis, thus improving real-time responsiveness and system scalability. The proposed solution is meant to be deployed in such a scenario, addressing these issues regarding scalability and latency.

Security is also a major concern to be considered in an Industrial IoT solution, given the fact that an intrusion in a factory monitoring system could lead to life threatening situations or severe system shutdowns.

1.1 Thesis Objectives

Based on the above mentioned challenges, we aim to design a scalable Industrial IoT solution using the most suitable interoperability standard available as today. To this end, a systematic literature review has been conducted, following the principle in [5]. The purpose of this literature study is to gain a better and complete understanding of the current IoT protocols, standards and to clarify the requirements of an Industrial IoT solution, in order to make an informed decision on the technologies used in our architecture.

The major objectives of this thesis work can be summarized as follows:

- Establish the "state of the art" for IoT protocol stack and application layer standardization activities.
- Design a scalable Industrial IoT architecture making use of the most suitable identified standard and open-source technologies.
- Implement a prototype of the proposed solution and build a testbed for its evaluation.
- Evaluate the performance of the prototype with a scalability analysis on a typical industrial use case.

1.2 Thesis Structure

The work of this Thesis is divided into five main chapters. Chapter 1 introduced the background and the motivation behind this thesis, and presented its main goals. Chapter 2 provides the state of the art regarding the IoT protocols and standards, as well as an introduction to other important topics related to this work, such as virtualization and cloud computing. Chapter 3 presents a general overview of the proposed solution and provides specific details about the prototype implementation. Chapter 4 describes the set-up for the experimental tests that have been conducted on the prototype and

presents the obtained results. Finally, Chapter 5 concludes this work, summarizing the main findings, and giving directions for future works on this topic.

Chapter 2

Background

This chapter will firstly introduce the concept of Industrial IoT, outlining its benefits and the main open issues. Then, an overview of the current protocol stack for the IoT is provided, followed by a description of the most notable standardization initiatives for the interoperability. Subsequently, I will describe the concepts of cloud computing and container virtualization, which are relevant to the understanding of the proposed solution. Lastly, several related research works are presented and discussed.

2.1 Internet of Things and Industrial Internet

2.1.1 Introduction

The Internet of Things has rapidly evolved in the last years as an umbrella term broadly used to refer to both the global network interconnecting a variety of *things* or *objects* around us by means of extended Internet technologies and the set of supporting technologies necessary to realize such a vision (including, e.g., RFIDs, sensor/actuators, physical and network layer protocols, etc.) [6].

The main goal of IoT is a radical evolution from the current Internet,

to a future of interconnected physical *things*, allowing their transition from traditional to *smart*, revolutionizing their utility and application. Those "things" could be any object around us, as long as they are augmented by network capabilities. They could be in the form of small gadgets like key chain, watches, eyeglasses and in the form of big items like cars, industrial robots and buildings. Smart objects are able not only to harvest information from the environment through the means of sensors, but also to interact with the physical world (actuators).

From Human-to-Machine to Machine-to-Machine interaction

However, one of the most interesting feature, is that the electronic interconnection between all the smart objects will enable them to interact with each other, paving the way to a new class of applications. In the IoT vision, the conventional concept of the Internet as an infrastructure network reaching out to end-users' terminals will fade, switching from an Human-to-Machine (H2M) communication, to a Machine-to-Machine (M2M) interaction.

As an example, this complex inter-networking and consequent exchange of data could be used to sense or control object remotely. Home automation has been proven as one successful application, improving the level of ease and convenience for residents to access and control their home appliances. In [7], Zaslavsky et al. depict a potential application in a smart home scenario, in which data coming from domestic sensors are shared with interested companies, in order to save on food expenses, producing a benefit for both parties.

Future Trends

In the near future, it is expected that IoT technologies will have a significant impact on home and business applications, contributing to the quality of life and to the growth of the world's economy [8]. Examples of some of the main domain areas in which IoT technologies are finding a significant adoption are: connected wearable devices, connected cars, connected homes,

connected cities, and, last but not least, the Industrial Internet, which is the major focus of this thesis.

The steady advances in electronics, in terms of size, energy-consumption and price reduction witnessed in the recent years, has led to the increasing integration of processors, communications modules and other electronic components into everyday objects, and this process can be expected to continue and accelerate. Devices are maintaining their physical size, while continuing to gain more and more capabilities [9].

In 2011, the number of interconnected devices on the planet overtook the actual number of people [2]. Many marketing consulting and technology companies have made forecast on the adoption of IoT. In 2016, IHS made a forecast that the installed base of IoT devices will grow from 15.4 billion devices in 2015 to 30.7 billion devices in 2020 and 75.4 billion in 2025. Cisco in 2011 IBSG predicted there will be 25 billion devices connected to the Internet by 2015 and 50 billion by 2020.

Retrospectively, we can state that IoT adoption is definitely growing, but its progress is slower than expected and the current numbers are not meeting the predictions made some years ago [10]. The reasons for that are different, among the others: high operational costs, technology immaturity, lack of interoperability and security and privacy concerns [3]. These issues will be analysed more in detail later. Such an unprecedented number of networked devices leads to new challenges and problems that need to be handled in order to facilitate and speed-up the adoption of IoT solutions in the near future [11].

2.1.2 Industrial Internet of Things

Unquestionably, the main strength of the IoT idea is the large set of opportunities that it will provide to users, manufacturers and companies. From the perspective of a private user, a countless number of novel services are made possible by the means of IoT technology, which are able to support the users in their everyday activities and answer to their needs. Some examples of

possible application scenarios are e-health, smart homes, enhanced learning and assisted living.

Similarly, from a business users point of view, Industrial IoT, focuses on how smart machines, data analytics and networked sensors can improve services in business-to-business domain.

Industrial Internet is often referred to as *Industry 4.0*, a concept introduced in 2011 by the German government in the strategic planning for their manufacturing industries [1]. The term reflects the idea that a new industrial revolution is happening right now. The first three industrial revolutions have seen the introduction respectively of steam engines, electricity and digitalization into the industrial production processes. The fourth industrial revolution is now possible thanks to the exponential growth that information and communication technologies have undergone from the end of the 20th century to the beginning of the 21st century, resulting in a spectrum of new technologies, namely Radio Frequency Identification (RFID) (1940s), Artificial Intelligence (AI) (1950s), Sensor Networks (1970s), 3D Printing (1980s), IoT (1990s), Cyber-Physical Systems (2005), Cloud Computing (2006), Big Data (2008) [12].

2.1.3 Benefits and Potential

The IIoT has the potential to improve connectivity, increase efficiency and push further scalability for various industrial organizations. Companies are already benefiting from the IIoT through time, resources and costs savings due to predictive maintenance, improved safety, and other operational efficiencies [13].

[14] and [15] provide a general introduction to IoT applications in various industry domains, such as automation and industrial manufacturing, logistics, business/process management, intelligent transportation of people and goods, as well as healthcare, security and surveillance.

As an example, thanks to IoT's ubiquitous identification, sensing and communication capabilities, it would be possible to track, collect and share all

healthcare-related information. For instance, it would possible to periodically collect a patient's heart rate and send it to the doctor's office.

Food supply chain would greatly benefit from IoT ubiquitous connectivity, addressing some challenges related to traceability, visibility, and controllability for its extremely distributed and complex operation processes. Collected data could then be analyzed in order to support and improve business decisions.

Pereira et al. [16], developed an efficient IoT Framework to prevent and reduce accidents in the mining industry. They make use of the latest available IoT technology in order to establish an effective communication channel between the surface and the underground. In this way, it is possible to dramatically improve the safety of the working environment, thank to the constant monitoring of the mining activities.

However, according to Accenture [17], operational efficiency is one of the main benefits that has attracted the first IIoT adopters.

A German study [18] states that productivity could be boosted by 30% by introducing automation and more flexible production techniques in the manufacturing industry.

Predictive maintenance

In this regard, predictive maintenance plays a key role and it certainly is one of the major area of focus. The aim of predictive maintenance is to recognize the upcoming equipment failure so that the maintenance can be scheduled only when it is needed, unlike in traditional planned maintenance, where it is done based on predefined scheduled intervals, leading to potentially unnecessary costs. Even worse is the case when maintenance is done only after the component failure, forcing to inconvenient plant and facilities shut-downs, with subsequent reduced throughput and decreased revenues. According to recent estimates [17], predictive maintenance can produce savings up to 12 percent over scheduled repairs, reduce overall maintenance costs up to 30 percent and eliminate breakdowns up to 70 percent.

Today, IIoT technology provides a more versatile way to conduct predictive maintenance: wireless connectivity, big data processing tools and lower cost sensors make it easier and cheaper to collect performance data and monitor the equipment health. For example, Thames Water, the largest provider of water services in the UK, and Apache Corporation, an oil and gas exploration and production company, are using this approach to anticipate equipment failures and respond more quickly to critical situations, such as leaks or adverse weather events.

Novel business models

According to a survey in [3], early adopters identify IIoT as a potential opportunity to create new revenue streams through innovative products and services. Accenture has reported several cases of business, which has been implementing a novel business model based on IIoT technology [17]. Michelin has developed a service to reduce fuel costs in truck fleets. Sensors inside vehicles collect data on fuel consumption, tire pressure, temperature, speed and location, which is then analyzed, with the purpose of making recommendations saving up to 2 liters of fuel for every 100 kilometers driven. CLAAS, a company developing equipment for the agriculture industry, create a new business model by allowing farmers to operate their machines on autopilot, giving also advice on how to improve their crop productivity.

2.1.4 Challenges and open issues

When comparing IIoT to more consumer-based application scenarios such as home automation, security and sports monitoring, there are many requirements that exist in the industrial domain that is non-existent or at least not a primary concern in the consumer world. Generally, IIoT has stricter requirements regarding delay, security and general robustness compared to consumer IoT. In Industrial IoT applications, flawless operation is expected due to the huge capital of investment and because failures in these devices can have consequences on safety of people and environment.

In the consumer market, devices tend to be replaced after a few years, especially if the price of new devices is low. On the contrary, in industrial applications, there is a clear need of long system lifetimes due to high installation costs. Therefore, it is important that the devices, systems and technologies that are installed will be able to operate for many years.

According to a survey conducted in [3], 65% of the respondents agree on the fact that the lack of interoperability standards and security concerns are the greatest barriers inhibiting business from adopting the industrial Internet.

In the following sections, I will describe more in details some of the challenges that IIoT solutions need to address.

Interoperability

The increased diversity and large numbers of devices from different vendors and providers in industrial systems requires standards and documented best practices in order to express the potential of IIoT, otherwise resulting in a massive set of protocols, different formats and interpretations of data, large numbers of APIs.

These issues are holding back the progress of IIoT, forcing companies to create their own proprietary systems to fit their needs, resulting in a large number of systems designed for similar purposes, but incompatible with each other. The rapid growth of IoT makes standardization difficult. However, standardization plays an important role for the further development and spread of IoT. Standardization in IoT aims to lower the entry barriers for the new service providers and users, to improve the interoperability of different applications/systems and to allow products or services to better perform at a higher level [15].

The use of standardized protocols always come with a higher overhead than using customized and fine-tuned proprietary protocols. However, by basing the architecture exclusively on open standards and protocols a high level of interoperability is made possible. An interesting standardization effort has been done by several standardization organizations, among which:

ETSI (the European Telecommunications Standards Institute) and OMA (Open Mobile Alliance). Those solutions will be described more in details later.

Security and Privacy

One of the most critical challenges is security. Having more and more devices networked opens up to more vulnerabilites and more decentralized entry points for remote based attacks. If an intrusion in a home automatic scenario can be annoying for the owner, in the industrial context, an intrusion in a factory monitoring system could potentially lead to life-threatening situations, physical damage, or service unavailability with severe consequences in terms of economical loss.

Privacy represents a serious concern in Industrial IoT. It is necessary to protect user resources and data from unauthorized access that may compromise their integrity along the whole chain from devices, the edge of the network, and to the cloud.

Due to the fact that IoT devices generate, process and exchange vast amounts of safety-critical and often privacy-sensitive information, they have been an appealing target for several attacks [19]. The recent Mirai botnet DDos (Denial of Service) [20] and its numerous variants should represent a wake-up call for industries to better secure IoT devices.

In their study, Sadeghi et al. [19] conclude that existing security solutions are inappropriate since they do not scale to large networks of heterogeneous devices, thus protecting IoT requires a holistic cybersecurity framework covering all abstraction layers.

Device Management

As industrial installations will be of considerable size, with several thousands of sensors and actuators, concerns regarding scalability are arising. In industrial applications, deployment and configuration costs are even higher than the cost of device itself [16]. It is clear that there is a need for methods and

tools for large scale maintenance and configuration in order to minimize the human interaction during their life cycle.

Managing such a number of devices is not an easy task especially in the presence of diverse hardware platforms and communications protocols. Scalable, interoperable and lightweight solutions are needed for the growth of IoT deployment and in order to avoid the management nightmare that will potentially stem in the coming years.

Latency

Industrial automation systems have often stringent requirements regarding to latency and determinism in terms of temporal behaviour. Variations of time an operation takes to perform is critical in control loops for industrial robots, where the signals from the sensors need to be processed quickly to in order to be able to send commands to actuators in time. A potential solution is the deployment of data and service intelligence at different levels in the systems, from the network edge to the cloud. Several paradigms such as Fog and Edge Computing are arising, that are potentially able address those issues. They will be analyzed in more detail later.

2.2 IoT Protocols Landscape

This section will provide an introduction to the currently leading application layer protocols, since they serve as the basis on which the interoperability solutions are laid on. An overview of the current IoT connectivity technologies, as they play an important role in the current and future Industrial IoT development and adoption.

2.2.1 Application Layer

2.2.1.1 CoAP

CoAP is an application layer protocol designed by IETF Constrained RESTful Environment (CoRE) working group, specifically designed for constrained network and devices. The main design objectives were to keep a small overhead, limited fragmentation, multicast support and to create a simplistic protocol for M2M communication [21].

It provides a RESTful API with similar methods as the HTTP protocol, like GET, POST, PUT, DELETE, but offering a significantly smaller message overhead, and extending it with support for specific M2M problems such as resource discovery of the nodes and providing an asynchronous transfer model.

In order to avoid the overhead created by TCP and its connection-oriented mechanism, UDP (User Datagram Protocol) or SMS (Short Message Service) are used by CoAP as its underlying transport protocol. Reliability becomes optional and it is offered at the application level, by providing different CoAP message types: confirmable (CON), non-confirmable (NON), acknowledgement (ACK) and reset (RST).

Built-in resource discovery is supported using the CoRE Link Format standard. CoAP messages are encoded in a simple binary format, allowing this functionality starting with just a 4-byte overhead.

CoAP introduces an asynchronous publish/subscribe mechanism to enable server-initiated communication, which is called *Observe/Notify*. This is due to avoid the polling mechanism of standard HTTP, which require clients to repeatedly perform the same request to know if a server resource has changed, an approach which is absolutely not efficient for power-constrained devices.

To start the observer/notify mechanism, the client indicates in a CoAP request its interest to *observe* the changes in the CoAP server by specifying the "Observe" option in the message options. This way, the client starts

observing the resource on the server and if the resource is updated, the server *notifies* the client with the new information.

2.2.1.2 MQTT

Message Queuing Transport Protocol (MQTT) is a standard developed by Organization for the Advancement of Structured Information Standards (OASIS) [22], designed for low power and constrained devices. Unlike CoAP, MQTT is based on the publish/subscribe paradigm, making use of a *broker* between the publisher and the subscriber. A publisher can simply send new updated data to the broker, which in turn relays the message to the clients who subscribed to that resource. The broker enables flexibility and decoupling between the entities: the subscriber does not need to know who the publisher is, and the message exchange is asynchronous. Since MQTT is based on TCP, all the messages are acknowledged on the transport layer. This causes and increased load on the network compared to CoAP. MQTT provides three levels of QoS (Quality of Service): The first one is best effort and it provides the same guarantees the same as the underlying TCP protocol. The second is called "At least once" and it guarantees that the message will be delivered at least once to the receiver. But the message can be delivered more than once. The third level guarantees that every message is received, and it is provided by two flows between the sender and the receiver, greatly impacting on the load and the latency of the communication.

2.2.2 Connectivity Layer

Traditionally, industrial systems have been using classic wired networking solutions, such as Fieldbus, CAN, PROFIBUS, HART, which have been proven to be reliable and capable to satisfy industrial connectivity requirements [23].

However, wireless communications is quickly making its way into the Industrial Internet of Things. Without the restrictions of cables, developers of industrial systems have the potential to cut costs and make deployment and maintenance easier, improving efficiency and productivity.

A plethora of IoT communication technologies has emerged recently, most notable ones are Bluetooth Low Energy (BLE) and ZigBee that are prevalent in specific scenarios, and Low Power Wide Area Network (LPWA), as well as cellular technologies, i.e., 4G machine-type-communication (MTC), Narrow-Band IoT (NB-IoT), that are meant for a much broader scope [24].

None of the aforementioned technologies has prevailed as a market leader, mainly because of technology shortcomings and business model uncertainties [25]. We can state that we are in a turning point, with many promising radio technologies and the improvement of cellular M2M communication, which is one of the major focus area into the ongoing 5G standardization.

In the following section, I will give a brief review of some of the most popular connectivity standards for IoT, focusing on whether they are suitable or not for industrial applications.

Bluetooth Low Energy. Bluetooth was originally designed by the Bluetooth Special Interest Group (SIG) for short range applications with data rates in the low Mbps. In 2010, Bluetooth Low Energy (BLE) was introduced as a part of the Bluetooth 4.0 specification, nowadays updated to 4.1, in order to provide lower costs, lower power consumption and less complex radio standard. One of the main drawbacks is the restriction to a single-hop topology, with one master device communicating with several slave nodes. In 2015, the Bluetooth SIG proposed the formation of the Bluetooth Smart Mesh working group [26], to develop and standardize an architecture for mesh networking, which will play an important role in the deployment of BLE for IoT solutions.

Given its features, BLE is believed to become the main standard for consumer applications such as smart living, health care and so forth. However, it may not be suitable for Industrial applications, since it lacks delay guarantees and due to the assumptions of a quasi-static star topology [23].

ZigBee. ZigBee is a low power and low cost standard which has found a wide application in Wireless Sensor Networks, which are generally considered

as the pioneering Industrial IoT applications [27]. It is based on the IEEE 802.15.4 Physical and Link standard specifications. The main drawback, especially from an Industrial perspective, is that it uses a single static channel for the communication, using Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) for channel access. This is clearly not ideal in an industrial context, where a massive number of connected devices might attempt to communicate concurrently [26].

In 2008, the IEEE 802.15 Task Group 4e was created, in order to overcome this limitations and develop an new multi hop Medium Access Control better suitable for emerging Industrial Applications.

LPWA. Low Power Wide Area is a recently emerged class of M2M communication technology, operating in the unlicensed spectrum. It focuses on resource and energy constrained devices, which require low power transmission on small amount of data exchanged, an area for which traditional cellular networks have not historically been optimized. It is currently implemented in many different proprietary solutions (LoRa, Sigfox, Coronis, NWave, Amber Wireless, etc.) [27]. The main goals of LPWA are to enable extended coverage (up to some tenths of kms), low device complexity, and long battery lifetimes (up to 10 years with AA batteries).

Despite its appealing and promising features, the use of the unlicensed spectrum for long-range communication represents a major drawback. Several restrictions are imposed on radio transmitters, which can cause asymmetric link budgets between the uplink and downlink directions [26]. Moreover, Palattella et al. [27] state that spectrum congestion could represent a problem to meet the scalability requirements of large scale Indutrial IoT deployments. It is important to say that LPWA is not seen as a replacement for cellular connection, however, it could play an important role to support the IoT market until the standardized cellular solutions will be ready for the adoption.

2.2.2.1 M2M cellular technologies

As I have pointed out in the previous section, the presented technologies can be viable for consumer use cases, but may not be able to meet the robustness and security requirements for civic, industrial and other related IoT applications. I.e., neither ZigBee nor Bluetooth provides a guaranteed wireless communication delay, they are susceptible to interference, and often produce lengthy system outages [12].

According to Andreev et al. [25], one of the biggest early mistakes was to think that low power technologies were the best solution, while high-transmission-power low-energy could be what we actually need. ZigBee only offer low power, which leads to short transmission range and multihop, that in turns yields to poor reliability as described above.

As long as the transmission is done in a very short time, high power cellular technologies can be energy efficient, having also the advantage of a higher communication range.

Moreover, due to the offered benefits in terms of wide coverage, high data rate, low latency, relatively low deployment costs and high spectrum efficiency, long-range cellular networks, are becoming increasingly attractive for many applications such as connected cars, smart cities, industrial internet [28].

4G technologies, i.e., LTE and LTE-A [29], are interesting again since their radio interface, OFDM, allows the scaling of the bandwidth according to needs, although the modem cost is quite high in early releases [27].

3GPP (3rd Generation Partnership Project) have started several initiatives aiming at enhancing LTE to become more suitable for M2M applications. The first proposed solution is called eMTC (enhanced Machine-Type-Communication), also referred to as LTE Cat-M1, an LTE evolution standardized to ensure Massive IoT deployment and coverage [25], that led to the introduction of a further evolution featuring Narrowband operations, that is NB-IoT.

Narrowband IoT (NB-IoT)

NB-IoT was proposed in 2016 by 3GPP in their Release-13 specification as an evolution to LTE Cat-M1, in order to provide a technology better suited for MTC applications needing low data rate, low module cost and and long battery lifetime.

NB-IoT is designed for optimal co-existence with existing legacy GSM and LTE technologies. Due its bandwidth of 180 kHz, it is possible for a GSM operator to deploy NB-IoT application in refarmed GSM carriers. Alternatively, LTE network operators can as well deploy NB-IoT in the guard bands of the LTE spectrum allocations.

Thanks to its 180 Khz bandwidth, NB-IoT can be deployed in re-farmed GSM carrier offering an alternative use of GSM spectrum. Alternatively it can be deployed in the guard bands of LTE spectrum allocations or using part of an operators LTE spectrum [27].

The performance is similar to the previous LTE MTC solution, with the benefit of having greater flexibility in the deployment, only requiring a base-band card update at the base station, enabling the reuse of all existing cell site equipment. NB-IoT is foreseen as a pioneer technology will continue to evolve towards 5G future requirements [28].

Towards 5G standardization

The fifth generation mobile network (5G) has been introduced in 2012 by ITU (Intenational Telecommunication Union), when they decided to develop a new International Mobile Telecommunication (IMT) system for 2020 and beyond. One of the main goals is to include from the beginning support for massive machine-type communications, high reliability and ultra low latency, in order to fulfil industrial IoT requirements.

It aims to achieve hundreds of billions connection, 1 ms end-to-end latency and real, not theoretical peak rates, of 1 to 10 Gbit/s, all features that will significantly benefit the development of industrial IoT applications. The networks should have a perceived availability of 99.999%, and a perceived

coverage of 100%, trading off data rate for range, which will also help with different M2M applications.

Several new technologies are needed to enable 5G communications. Massive MIMO and millimeter wave is required to increase bandwidth and spectral efficiency, supporting more data per each node and making a better use of the 5Ghz unlicensed spectrum. Offloading and extreme densification can improve the area spectral efficiency, enabling more active nodes per unit area and frequency [23].

2.3 Interoperability standards

The rapid growth of IoT makes the standardization difficult. However, as pointed out in the previous sections, standardization plays an important role for the further development and spread of IoT. This section will provide an introduction to the most prominent IoT standards and initiatives, focusing on LwM2M which is the choice of this thesis work, and comparing it with OPC-UA, the successor of the classic OPC standard, which has strong ties with the industrial market.

2.3.1 SmartM2M and oneM2M

An interesting standardization effort has been done by ETSI (the European Telecommunications Standards Institute), which resulted in smartM2M. It is based upon an adaption layer called Service Capability Layer (SCL) that can run on top of constrained devices, IoT gateway and network instances. It exposes a RESTful API enabling generic resource access, addressing, device management, data storage, transaction management and so on [27] .

The standard also defines an M2M security framework, encompassing authentication, key agreement and establishment, service bootstrap, and connection procedures. The most critical downside of this framework is that each transaction is mediated by the M2M network, which can easily become a single point of failure, leading to scalability issues, which makes it a non

ideal solution for future IIoT.

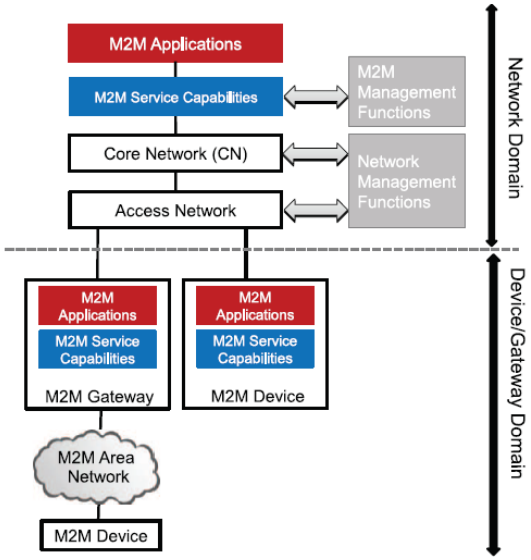


Figure 2.1: SmartM2M high level architecture.

With similar objectives, the oneM2M initiative started as an international project between different standardization entities around the world [30]. Its goal is to define an horizontal service layer interconnecting M2M components on a global scale. oneM2M also tackles the scalability problem by representing its resources in a hierarchical organization called *resource tree*.

oneM2M also targets the definition a horizontal service layer that interconnects heterogeneous M2M hardware and software components on a global scale. Figure **??** illustrates its architecture, which consist of at least one Common Service Entity (CSE), one Application Entity (AE) and a Network Service Entity (NSE). A CSE node is a logical instance taht can provide a set of services called Common Service Functions (CSFs), that can be used by the AE nodes to provide application logic, such as remote monitoring functionalities, for end-to-end M2M solutions.

Being a relatively new standard, oneM2M has still not many mature implementations as of today.

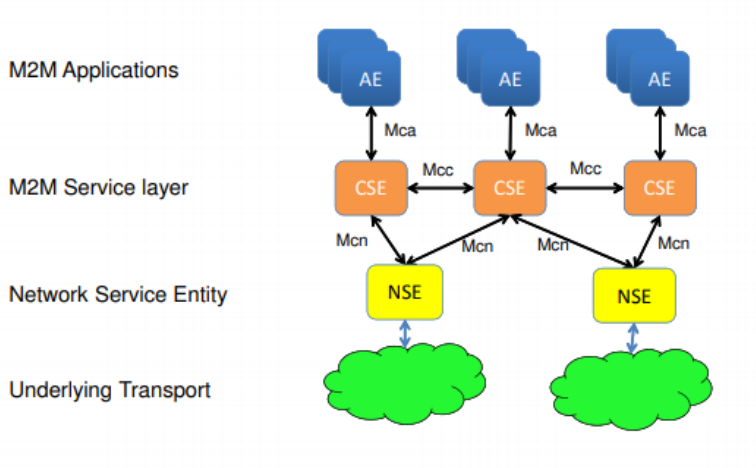


Figure 2.2: oneM2M architecture.

2.3.2 LWM2M

LWM2M is the interoperability solution developed by OMA, on the most prominent standardization bodies in mobile communication. Its main feature is to provide several interface for the purpose of monitoring, provisioning and managing connections of networked devices, enabling remote service and application management for the emerging Internet of Things [31].

It features a modern architectural design based on REST, defines an extensible resource and data model and builds on an efficient secure data transfer standard called the Constrained Application Protocol (CoAP). LWM2M is targeted in particular at constrained devices, such as low-power microcontrollers with small amounts of memory, but it can also find applications with more powerful embedded devices that benefit from efficient communication.

While LWM2M is used for device management operations, its Object Model is being used to provide a resource model for applications. IPSO Alliance is defining IPSO Objects built on the LWM2M Object Model, providing a reusable Object Model so that any party can define their own objects and either suggest them for standardization to OMA, or just use them on their applications without standardisation [32].

OMA has approved first version (V1.0) of LWM2M in February 2017, so

it is a fairly recent standard. Despite that, several implementations already exist, both in C and Java, provided by the Eclipse Foundation [33].

2.3.2.1 Architecture

As depicted in Figure 2.3, LWM2M consists in a client-server architecture. LWM2M architecture defines three logical components:

LWM2M Client: It contains several LWM2M objects with several resources. LWM2M Server can execute commands on these resources to manage the client, commands such as to read, to delete or to update the resources. LWM2M Clients are generally the constrained devices (sensors, actuators, etc.).

LWM2M Server: It manages LWM2M Clients by sending management commands to them. The LWM2M Bootstrap Server configures the access control for a specific LWM2M Server on the constrained device.

LWM2M Bootstrap Server: It is used to manage the initial configuration parameters of LWM2M Clients during the bootstrapping process. It is only entitled to configure the device to give access to specific LWM2M Servers, hence, management of the device does not involve the bootstrap server after the bootstrap process.

2.3.2.2 Interfaces

The standard defined four logical interfaces between the components described above, which are the following:

Bootstrap: LWM2M Bootstrap Server sets the initial configuration on LWM2M Client when the client device bootstraps. For this interface, the client sends a “Request Bootstrap” message to the bootstrap server and the server performs “Write” and “Delete” on the client’s access control objects to register one or more LWM2M Servers.

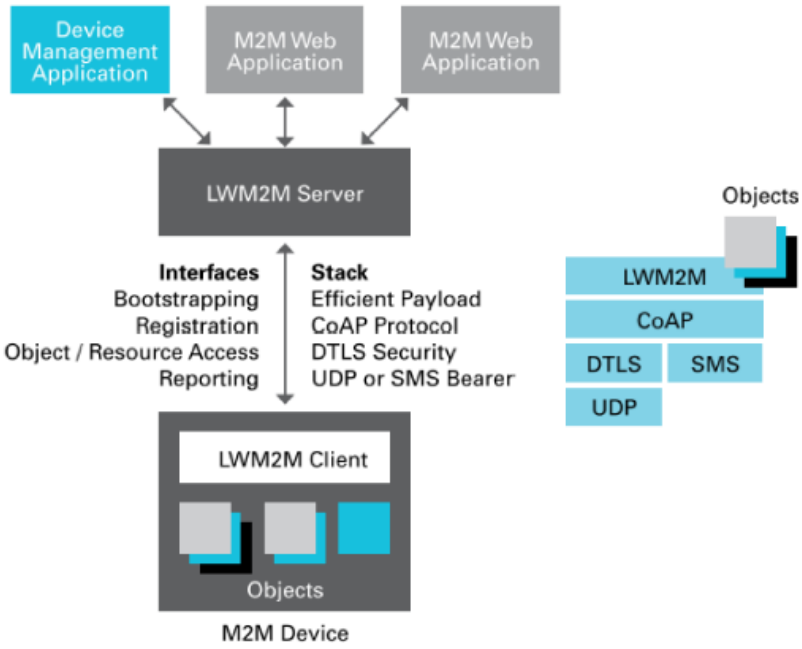


Figure 2.3: LWM2M architecture and protocol stack [31]

Client Registration: LWM2M Client registers to one or more LWM2M Servers when the bootstrapping is completed.

Device Management and Service Enablement: LWM2M Server can send management commands to LWM2M Clients to perform several management actions on LWM2M resources of the client. Access control object of the client specifies the set of actions the server can perform, that can be configured during the bootstrap phase.

Information Reporting: Exploiting the Observe/Notify feature of the underlying CoAP protocol, an LWM2M client is able to directly send a resource update to the LWM2M server in form of notifications, without explicitly waiting for a request.

2.3.2.3 Resource Model

The LWM2M standard defines a simple data model in which each LWM2M Client exposes its accessible information throught the means of *Resources*. Multiple resources are then aggregated together forming an *Object*. The LWM2M Figure 2.4 depicts the relationship between Resources, Objects, and the LWM2M Client.

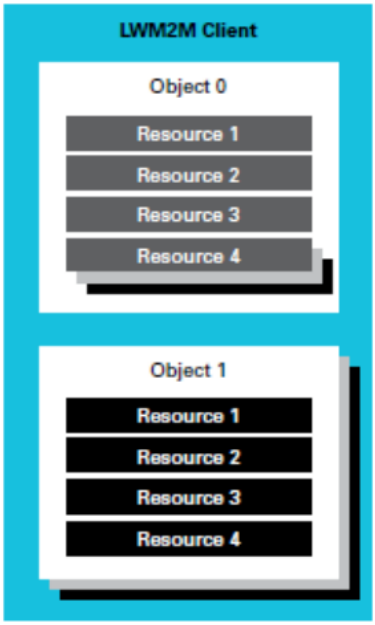


Figure 2.4: LWM2M object and resource model [31]

The LWM2M data model and the open OMA naming authority registry for Objects provide easily accessible and reusable semantics for both device management and application data for the whole Internet of Things industry [34]. OMA is also developing a LwM2M editor tool [35] to safely construct these objects/resources models.

2.3.2.4 Security

In order to ensure a secure communication channel between the LWM2M clients and the LWM2M servers, the standard specifies the use of Datagram

Transport Layer Security (DTLS) protocol [36]. There are several available security modes: pre-shared key for very limited devices and public and private key technology for more capable ones. LWM2M Bootstrap server is used to manage the keying, access control and configuration of a device to authenticate with a LWM2M Server. The endpoint identifiers, security identifiers and keys are used systematically in the LWM2M environment to provide a complete security life-cycle.

2.3.3 OPC-UA

OPC-UA (Unified Architecture) is the standard for industrial automation developed by the Open Process Controlled Foundation. It is an evolution of the classic OPC standard, which dates back in the 90s and it was tightly coupled with Microsoft Windows and its COM/DCOM interface.

The objectives of this new standard are: to be platform independent, dropping the dependency from Microsoft COM, to improve security and improve modeling.

OPC-UA provides a client-server architecture, in which clients and servers are entities interacting that can be combined to create applications. The OPC-UA specification defines abstract services following services oriented architecture (SOA). The Service Mapping part of the specification defines several protocol bindings to map these services to network transport. Currently, there are four bindings: a binary mode through TCP or HTTPS, or an XML encoding which is accessible through SOAP webservices.

The data model is no longer based on folders, items and properties, but it is a Full Mesh Network based on Nodes, that can be compared to a modern object oriented programming language.

OPC-UA security is based on a Public Key Infrastructure (PKI) using industry standard X.509 digital certificates and addresses authentication, authorization, encryption and data integrity.

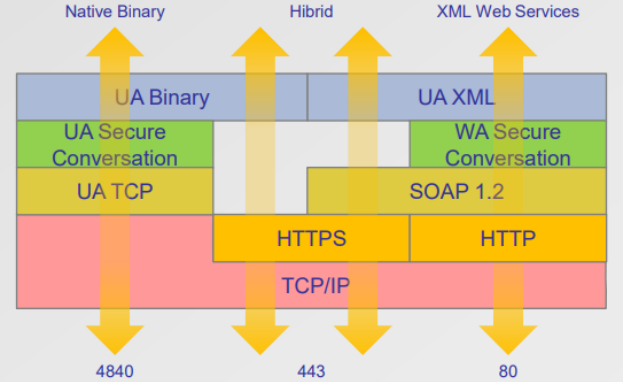


Figure 2.5: OPC-UA protocol bindings [37]

2.3.4 Summary and Comparison

oneM2M and LWM2M are two complementary technologies that are not meant to be competitors. While LWM2M is originally a Device Management technology extended to support generic data exchange, oneM2M tries to provide a full, but complex service layer technology that supports devices as well as cloud servers. Several works already exist that try to combine LWM2M and oneM2M [38] [39]. My work focuses on extending the capabilites of LWM2M adding support for scalability, without the added complexity of oneM2M.

On the other hand, OPC-UA is a widely accepted standard in Industrial Automation. However industrial communication protocols such as OPC-UA sometimes seem a bit expensive for small embedded devices, while LWM2M standardisation is designed to provide efficient communication and device management protocols for resource constrained embedded devices.

Table 2.1 gives a comparison of the most important properties between the two standards.

2.4 Cloud Computing

As pointed out before, IoT usually consists of devices with limited memory and processing power, and the number of these connected *things* is predicted

Table 2.1: Comparison Between OPC-UA and LWM2M

	OPC-UA	**LwM2M**
Protocol State	Stateful	Stateless
Transport Protocol	TCP	UDP and SMS
API	SOAP or Binary	REST
Communication Style	Client-Server, Publish-Subscribe (draft)	Client-Server
Application Protocol	TCP Binary or HTTP	CoAP

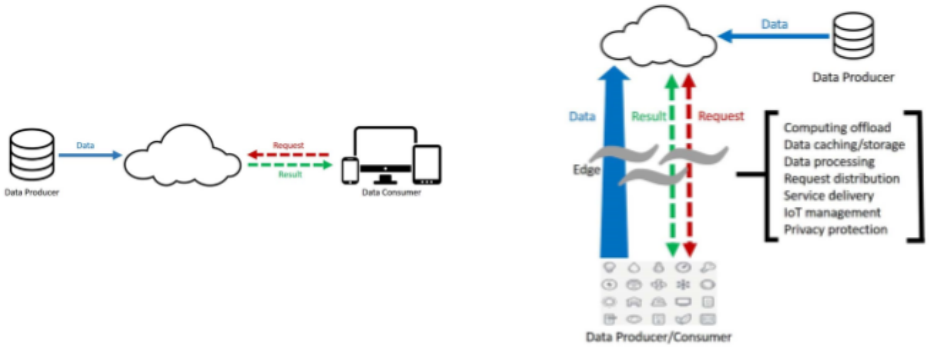
to grow at an unprecedented rate in the next few years. However, IoT applications require huge computing power and massive storage, in order to analyse, collect and manage such a large and heterogeneous volume of data. Cloud Computing has emerged as one of the most promising solution to meet these requirements.

Cloud Computing can be defined as a model for on-demand and ubiquitous access to a shared set of configurable computing resources (e.g., servers, storage, networks, applications, services), which can be dynamically provisioned and released with minimum management effort or interaction with the service provider, thanks to next generation data centers in which computer nodes are virtualized through the means of Virtual Machines (VMs) [40].

Thanks to its features of virtually infinite scalability, fault tolerance, high availability and affordable price due to the economy of scale, Cloud Computing has attracted attention of both academia and industry, leading to the development of several IoT solution based on this technology [11].

2.4.1 Edge and Fog Computing

Despite Cloud Computing can help to overcome most of the IoT limitations, there are various situations in which it could be not the best option. As the number of connected things is expected to grow to more than billions in a few



(a) Cloud computing (b) Edge computing

Figure 2.6: Illustration of the two computing paradigms [41]

years, the enormous data that will be produced at the edge of the network could lead to huge unnecessary bandwidth and computing resource usage. Certain IoT applications may have very strict response time requirements or privacy constraints for which Cloud Computing could represent an unviable option. For instance, an autonomous car will generate around 1 gigabyte of data every second [41], thus sending all the data to the cloud may result in a response time that would be too long to make correct decisions. Moreover, considering that IoT wireless communication module is usually very energy hungry, offloading some computing tasks to the edge could be more energy efficient. Cisco predicted that, by 2019, 45% of IoT-created data will be stored, processed, analyzed, and acted upon close to, or at the edge of, the network [42].

This model where data is analyzed and processed by applications running in devices within the network rather than in a centralized Cloud is referred to as Edge or Fog Computing, a term introduced by Cisco engineers in [43]. Figure **??** show the difference between Edge Computing and the traditional cloud architecture.

It is worth noting that Edge Computing is not intended as a replacement for the cloud. As Bonomi states in [43], Fog computing provide localization,

therefore enabling low latency and context awareness, the Cloud provides global centralization. Fog Computing cannot provide functionalities such as complex analysis, data access to large numbers of users and storing historical data, which is complemented with Cloud Computing. In many application both models will be combined, forming a 3-tier architecture, as shown in Figure 2.6(b). While the first Edge layer is designed for M2M interaction, data processing and filtering, the cloud tier deal with visualization and reporting (human-to-machine [HMI] interactions), as well as historical data storing and analytics. Table 2.2 shows a general overview of what could be the possible roles of the Edge and Cloud layers in different IoT applications.

Table 2.2: Potential roles of edge and cloud layers. Adapted from [44]

Device	**Edge or Fog Layer**	**Cloud Layer**
Provides user interface (I/O, rendering of output)	Hosts as 3rd party apps "network" functions like video acceleration	Provides data storage (long permanence)
Hosts micro-control of actuators	Hosts middleware like registry for edge applications, inter-edge-application communications, services that provide access radio network information.	Provides human to machine interface for overall application management (e.g. dashboard, deployment, provisioning)
Hosts micro-control of onboard sensors	Offers APIs to 3rd party applications executed at the edge	Hosts visualization and reporting for operations
Hosts local compute, storage, network stack	Hosts compute-heavy parts of the overall application (e.g. object recognition, motion classification)	Provides off-line, batch data analytics software
Others	Hosts real-time analytics software	Hosts machine-learning software
	Hosts latency-sensitive control-loop software	Serves queries from device with response time >100ms
	Issues control commands to devices and actuators	Hosts enterprise integration components
	Collects M2M/IoT data incl. sensor data	Others
	Filters data (to consume locally or to send to the main cloud)	
	Serves queries from devices with required response times in a range 1ms to 100ms	

2.5 Virtualization

Due to the difficulty of setting up an environment with a great number real devices for the evaluation of our architecture, a testbed consisting of simulated virtual devices has been implemented. Virtualization provides the flexibility to be deployed in different physical machines in a completely transparent way. For this reason it has also become the standard model used by service providers to enable Cloud Computing.

The classic virtualization technology is called *Hypervisor based Virtualization*. The Hypervisor, also called Virtual Machine Monitor (VMM), is a software that run in the *host* Operating System, and it is used to create and manage virtual machines (VM) called *guests*, each one running in an isolated manner and with its own allocation of resources (CPU, memory, etc.) [45]. The main drawbacks of this approach is the unnecessary resource utilization that causes, since the operating system has to be replicated multiple times on the host machine, and the runtime overhead caused by the hypervisor managing all the privileged instructions executed inside the guest VMs.

For this reason, a new virtualization paradigm has emerged recently, called Container based virtualization. Container based virtualization is a lightweight alternative, that achieves virtualization at the level of the host operating system without the need of an hypervisor [46]. It is based on virtual environment called *containers*, which all share the same operating system of the host, dramatically reducing the occupied storage space. Even though different containers are executed side-by-side on the host operating system, they can still remain isolated thanks to particular features of the operating system kernel, which is able to provide a virtualized view of the resources: network interfaces, process IDs, interprocess communication, and directory tree. Although they could be theoretically implemented on any operating system, most popular solution are based on the Linux Kernel.

Figure 2.7 illustrates the difference between virtual machine and container based virtualization architectures.

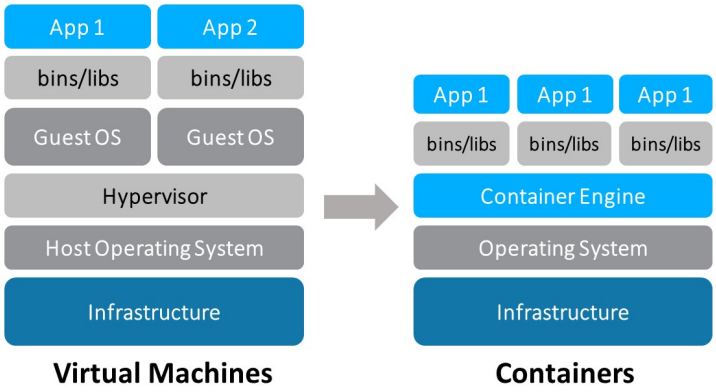


Figure 2.7: Difference between virtual machines and containers

2.5.1 Docker

Docker is arguably the most popular container-based virtualization solution, built on top of Linux kernel feature called *cgroups* and *namespaces* [47]. It provides all the necessary tools to package applications with all the necessary runtime dependencies into so called *images*, enabling it to run the same way regardless of the environment. Docker images are composed by several read-only layers, each one corresponding to an instruction, such as running some specific command, adding a file into a directory, setting the environment variable, and exposing a port for communication. The idea is that in case multiple images have common base layers, it is possible to keep only a single copy of the common layer and share it among all the images, allowing to substantially save the storage space required.

A Docker container can then be created from an image with minimal effort using the Docker client interface. When a container is created, a writeable layer is created on top of the image. Any change that happens during the execution of the container is written on this layer, while the underlying image remains immutable. This allows for multiple containers to share the same image, confirming the lightweight nature of Docker. Figure 2.8 illustrates the structure of a Docker image and container.

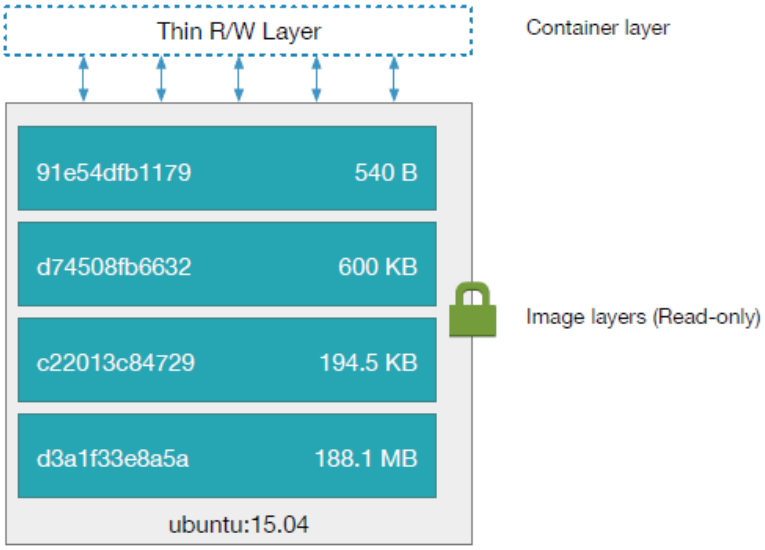


Figure 2.8: Docker images and containers structure

Docker Engine provides an API and command-line tools for starting, stopping, resuming and removing containers. When a container is started, it is possible to specify additional configuration that determine how the container interacts with the host system. For instance, ports can be published by providing a port mapping between an external and an internal port number. Other hosts on the same physical network are able to communicate with the container through the external port, and the connection is mapped to the internal port inside the container. Volumes can also be mounted, which allow the container to read and write directories of the host filesystem. Changes to these files are persistent, even in the event that the container is stopped.

2.6 Related Works

There are several research works in the literature that analyse problems similar to the one considered in this work.

Tanganelli et al. [48], propose an Edge-centric distributed architecture to provide IoT applications with a common service for global discovery and

access of resources. This service is realized by deploying IoT gateways in the Fog layer, collaborating together by the means of a P2P overlay implemented by means of a DHT (Distributed Hash Table). They provide a scalability analysis similar to ours, confirming the benefits of a distributed and federated architecture. However, their implementation makes use of plain CoAP, without any additional data model. Moreover, it does not consider any security mechanism, making it unsuitable for Industrial IoT, as it lacks two major requirements (Interoperability, Scalability).

Robles et al. [49] provided a group management extension for the LwM2M standard. This is achieved by introducing a new device in the architecture, the LwM2M Proxy, which can minimize the messaging to the LwM2M Server by forwarding a request to all the devices belonging to a certain group. The proxy is implemented by behaving both as a LwM2M client for the server and as a LwM2M for the devices. A server then can simply issue an *exec* command to address a certain group. The authors provide an evaluation of the solution, demonstrating that the load on the network is substantially reduced when using the group feature, thus potentially representing an useful device to improve the scalability of an LwM2M solution.

Makinen et al. [50] presented the design of an IoT emulation platform (ELIoT). The proposed solution makes use of Docker containers running LwM2M client instances, in order to provide a flexible and portable environment to test and study interactions between IoT devices. Inspired by this work, we used a similar approach to evaluate our architecture, enabling us to flexibly deploy a variable number of devices for our test cases.

Chapter 3

Design and Implementation

In this chapter, I will present the general design choices adopted to develop our scalable architecture. First of all, I will introduce Eclipse Leshan, an LWM2M library around which the implementation is based. Next, a general overview of the proposed architecture is provided, followed by a detailed description of its components and their implementation.

3.1 Eclipse Leshan

Eclipse Leshan is an implementation of the LWM2M standard written in the Java programming language. The development was originally started by Sierra Wireless[1] in 2014 with the early versions of the standard, then becoming in 2015 an open-source project mantained by the Eclipse Foundation. Leshan does not provide a full LWM2M solution. It is instead a set of libraries, which a developer can use to write its own LWM2M Client and/or Server implementations.

The main reasons for choosing it for this work is that Leshan is considered to be a reference and mature implementation by many researchers [51] and it has a very active development community. Moreover, it has an almost complete coverage of the LWM2M specifications.

[1] https://www.sierrawireless.com

Leshan is based on the Eclipse Californium[2] project, a CoAP framework which has been designed for cloud services, therefore having a focus on scalability and usability instead of resource efficiency for embedded devices. The Scandium[3] sub-project is used as the DTLS implementation, providing LWM2M security with three authentication options: Pre Shared Keys, Raw Public Keys and X.509 certificates. All of these reasons make Leshan an ideal choice for the LWM2M server cluster implementation that will be describer later.

The project also includes a client and server demo to be used as a reference for one's own implementation. The server demo also contains a simple web UI, which manages user commands, visualizes the communication between the client and server and shows CoAP messages, useful for familiarizing with the protocol and debugging clients.

A demo bootstrap server implementation is also provided, which however was not used in this work, in order to reduce complexity during the test phase.

3.2 Architecture Overview

The idea behind the proposed solution is to combine LWM2M together with the *Fog Computing* paradigm, which has already been introduced in Section 2.4.1, combining both the advantages of using a widely adopted and secure standard with the benefits of Fog Computing. Having multiple Fog nodes can help in many situation, especially in an industrial environment. For example, in a factory floor where there could be thousands of sensors and actuators, it would be inconceivable to handle them with a single IoT gateway. Recurring to Cloud Computing is also not always feasible, due to the added latency. Moreover, in future large scale Industrial IoT systems, it is expected to have several different IoT domains deployed in the same factory,

[2]http://www.eclipse.org/californium

[3]https://github.com/eclipse/californium.scandium

collaborating with each other in order to reach a common main goal. In order to integrate different IoT infrastructure and enable seamless discovery and resource access across multiple domains, it is necessary to achieve collaboration and cooperation between the IoT gateways, in this case represented by the LWM2M servers. Figure 3.1 depicts a general overview of the proposed architecture, illustrating its main components.

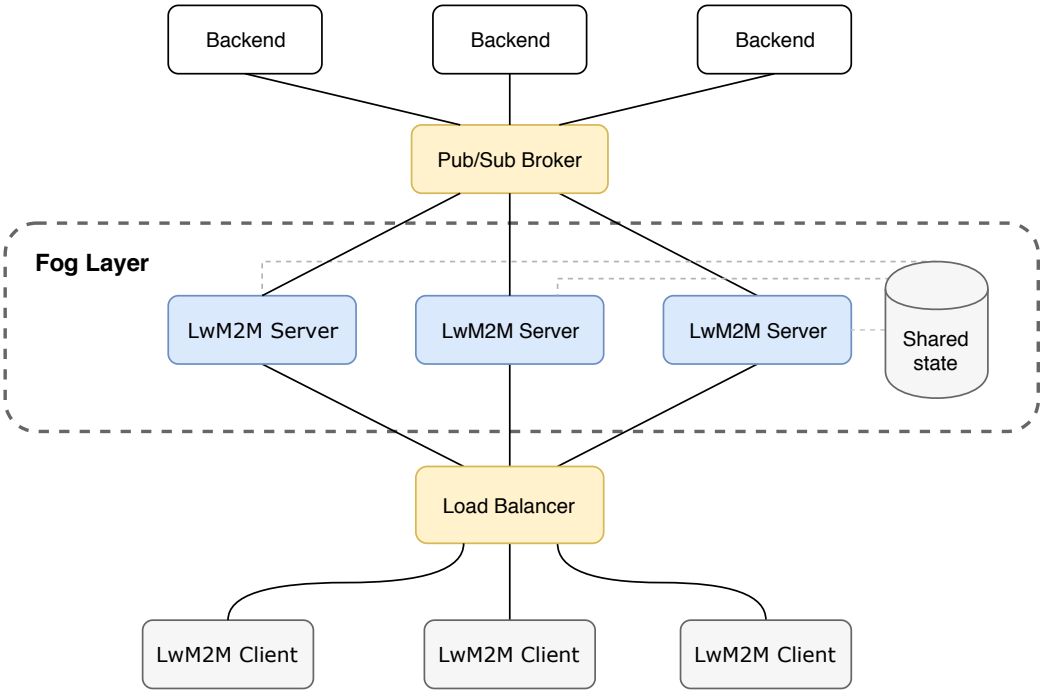


Figure 3.1: Overview of the proposed architecture

3.2.1 Requirements

The main issue is that the LWM2M specification per se does not provide any clustering support or any sort of federation between servers. There are many things to consider in order to implement clustering for LWM2M servers. Several states need to be shared across the instances, as shown in the figure. CoAP uses Message IDs and Tokens for duplicates and reliability.

A DTLS connection needs to maintain several information like handshakes, keys, epochs, fragments. Then, at LWM2M level it is necessary to share data about the registered clients, the ongoing observations and the security keys, in order to achieve a global resource access directory.

Using a Load Balancer that operates at Level 4 of the TCP/IP stack (i.e. forwarding based on the IP/port combination), it is possible to avoid saving CoAP and DTLS states, which would otherwise dramatically impact performance. The Load Balancer, in fact, would forward all the packets coming from one LWM2M client to the same LWM2M server instance.

Regarding the LWM2M registrations and security info, Leshan already provides an implementation of a "Registration Store" and "Security Store", which can easily be adapted for my architecture.

While the Leshan demonstration server integrates a Web server, which exposes a RESTful API to interact with it and access client objects and resources, a specific backend interface needs to be implemented in the cluster case. In order to interact with the cluster and access a specific resource on a device, it necessary to determine, which server instance is responsible for that device. The use of a Publish/Suscribe broker represent a convenient and reliable way to handle requests and responses. The use of a Backend interface could even enable more freedom and flexibility, allowing the decoupling between LWM2M resource management and IoT applications.

Lastly, a frontend in order to interact with the end user need to be implemented. Due the use of a Publish/Subscribe broker, Websockets represent a convenient way for the development of Web based user interface.

All the components presented above will be described in more detail in the following sections.

3.3 LWM2M Server Cluster

This section provides a detailed description of the various components that are necessary to enable collaboration and clustering support for LWM2M

Servers. After this, an illustration of the sequence of interactions when registering a new client will be depicted, followed by a description of the backend interface enabling the end-user to make requests and receive responses.

3.3.1 Load Balancer

In order to enable a seamless clustering support and a global resource access, every LWM2M instance has to know the same amount of information as the others. Sharing all the DTLS and CoAP states would be too expensive, negatively impacting the overall performance. The use of a Layer 4 Load Balancer represents a potential solution to this problem. Packets coming from the same pair of source ip address and port will be forwarded to the same server, thus when a LWM2M client performs a registration, all the subsequent communications will be handled by the same LWM2M server instance, keeping the same DTLS session. In this way, the complexity of the implementation is greatly reduced, at the cost of having a slight reduction in flexibility and parallelism.

3.3.1.1 Linux Virtual Server

One issue is that not so many Level 4 Load Balancers that support UDP protocol exist, as they are often only based on TCP. One of the most prominent UDP Load Balancers is Linux Virtual Server[4] (LVS).

LVS is a free and open-source load balancing solution with "the mission to build a high-performance and highly available server for Linux using clustering technology, which provides good scalability, reliability and serviceability" [52]. The software is built on top of `Netfilter`, a Linux Kernel module present since version 2.3.x. This permits to achieve incredibly fast speeds, often within 5 percent of direct connection [53].

The userland utility program used to configure LVS is called `ipvsadm`, which requires superuser privileges to run. In LVS terminology, the Load

[4]http://www.linuxvirtualserver.org

Balancer is also referred to as *director* or *virtual server*, as it appears as a single server to the clients. The *real servers* are the machines holding the actual services and the ones who serve the request.

The director is basically a router, with routing tables set-up specially for LVS operation. These tables allow requests from clients to services provided by LVS to be redirected to the real servers.

The code snippet illustrated in Listing 3.1 shows an example of LVS configuration for a cluster of LWM2M servers. The first two commands assigns a virtual server service on UDP port 5683 and 5684, which is the IP address LWM2M clients will use for the registration. The chosen scheduling algorithm for load balancing is round-robin `(-s rr)`. The commands in the loop assign real servers to the previous chosen virtual service. The `-m` option specifies that *masquerading*, better known as NAT (Network Address Translation), will be used as the packet forwarding method.

Listing 3.1: LVS configuration for Leshan server cluster.

```
1  # $VIP: virtual server address
2
3  sudo ipvsadm -A -u $VIP:5683 -s rr #CoAP
4  sudo ipvsadm -A -u $VIP:5684 -s rr #CoAP w/ DTLS
5
6  # R_IPS: array containing real server addresses
7
8  for RIP in "${R_IPS[@]}"
9  do
10     sudo ipvsadm -a -u $VIP:5683 -r $RIP:5683 -m
11     sudo ipvsadm -a -u $VIP:5684 -r $RIP:5684 -m
12 done
```

Using the command `ipvsadm -L -n`, it is possible to query the status of the current LVS set-up, as well as some statistics regarding the incoming

connections and assignments. An example of output can be seen in Listing 3.2.

Listing 3.2: LVS set-up summary

```
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port       Forward Weight ActConn InActConn
UDP  130.233.96.206:5683 rr
  -> 172.18.0.2:5683          Masq    1      0       0
  -> 172.18.0.3:5683          Masq    1      0       0
  -> 172.18.0.4:5683          Masq    1      0       0
UDP  130.233.96.206:5684 rr
  -> 172.18.0.2:5684          Masq    1      0       0
  -> 172.18.0.3:5684          Masq    1      0       0
  -> 172.18.0.4:5684          Masq    1      0       0
```

3.3.2 Shared Store

Once having solved the issue regarding CoAP and DTLS states, we need to manage the states related to the LWM2M standard: Registrations, Observations and Security information. It necessary to implement a shared store of these states so end-users can have access to all the registered clients and their resources and LWM2M servers are able to identify, which instance is responsible for a particular device.

Leshan already provides two implementations for the Registration Store: a simple one residing in the Java VM memory, and another one using an external data store called Redis[5].

3.3.2.1 Redis

Redis is an open-source NoSQL in-memory key-value database, which can store data in various useful data structures such as strings, hashes, lists, sets and others. Redis supports many powerful features like built-in persistency,

[5]https://redis.io

pub/sub, limited transaction support with optimistic locking and Lua Scripting. It is ranked as the most popular key-value database as per DB-Engines Ranking [54].

Redis is written in ANSI C and is heavily optimized for performance, working as an in-memory dataset. Persistence can be achieved either by dumping the dataset to disk every once in a while, or by appending each command to a log.

An interesting feature that can be useful for the proposed architecture is its native support for clustering. Redis Cluster provide the ability to automatically split the dataset among multiple nodes. Moreover, thanks to the support for master/slave replication, it is able to continue operations even when a subset of the nodes are experiencing failures.

In my solution, each LWM2M server node has its own Redis instance. Configuring clustering support for Redis requires only modification to the `redis.conf` file, and the use of Ruby utility called `redis-trib`.

Many popular Redis client API provide native support for Redis Cluster, such as Jedis[6], the Java Redis API that is used in my implementation.

3.3.2.2 Registration Store

As pointed out before, Leshan already provides an external `RegistrationStore` using Redis. This implementation has been extended for the use with the Redis Cluster, by simply changing some Jedis function calls.

The Leshan implementation also stores information about observations. Whenever a client performs or update a registration, the information are serialized as JSON bytes and stored under a key prefixed `REG:EP:` plus the endpoint name. A secondary key with the registration Id and the corresponding endpoint is also kept under the prefix `EP:REGID`. Listing 3.3 provides an example of a registration record.

An end-user interested in discovering the registered client resources, or an intermediary backend wanting to provide such a service, simply has to

[6]https://github.com/xetorthio/jedis

query the Redis database directly. In this way, it is possible to obtain all the clients' registration information, including their connected objects. Then, in order to know the available resources for each object, it is necessary to download the objects specifications from the IPSO Object Registry. This procedure will be detailed later in the frontend implementation.

Listing 3.3: Client Registration entry

```
{
    "regDate": 1529700131323,
    "identity": {
        "address": "172.17.0.2",
        "port": 38055
    },
    "regAddr": "0.0.0.0",
    "regPort": 5683,
    "lt": 30,
    "ver": "1.0",
    "bnd": "U",
    "ep": "test1",
    "regId": "dA3f5jm7at",
    "objLink": [{
        "url": "/", "at": { "rt": "oma.lwm2m"}
    }, {
        "url": "/1/0", "at": {}
    }, {
        "url": "/3/0", "at": {}
    }, {
        "url": "/6/0", "at": {}
    }, {
        "url": "/3303/0", "at": {}
    }],
    "addAttr": {},
    "root": "/",
    "lastUp": 1529700185460
}
```

For observations, only a token is stored under the key prefixed `OBS:TKN:` for managing its existence; data for values in the observations notifications is not

kept in the registration store. This token is managed by the underlying Californium layer, in order to correlate the notification with the corresponding observation. A list that associates an endpoint with its current observations is also kept. Listing 3.4 illustrates an example of observation record, which shows all the necessary fields to re-correlate a received notification.

Listing 3.4: Observation entry

```
{
    "request": "4801ffffd7ca10784d92d74b605433333033
                01300435373030622d17",
    "peer": {
        "address": "172.17.0.8",
        "port": 35691
    },
    "context": {
        "leshan-endpoint": "test7",
        "leshan-path": "/3303/0/5700",
        "leshan-regId": "oxytptIZV7"
    }
}
```

3.3.2.3 Security Store

The security store is used to hold security information for clients in order to authenticate with LWM2M servers, like identity and password for the pre-shared secret method, as well as raw public keys or X.509 certificates. With X.509 common name of the certificate presented by the client is used as the endpoint name. However, despite the Security Store is fully implemented, it lacks an interface in order to add/or remove information on it. This has been done as part of the Backend Interface as it will be described later.

3.3.3 Pub/Sub Broker

Due to the introduced architectural constraints, whenever an end user wants to perform a request, this must be handled by the LWM2M server instance in charge of the target device. A Publish/Subscribe broker comes handy for this purpose and can serve as an interface with an IoT backend.

Redis has already an integrated Pub/Sub module providing this feature. Thanks to this, it not necessary to install any additional piece of software and it is possible to use the same Redis Cluster instance used for the Shared Store.

With the Publish/Subscribe paradigm, senders (publishers) do not send their messages directly to receivers (subscribers). Rather, messages are sent onto *channels*, without knowing who are the receivers, if there are any. An entity can express interest in one or more channels by subscribing to them, receiving then only messages of interest. This decoupling of publishers and subscribers can allow for greater scalability and a more dynamic network topology [55].

There are many client libraries supporting Redis Pub/Sub, like the Jedis Java library used in this implementation. A revised infrastructure of the architecture including the more specific components introduced so far is depicted in Figure 3.2.

3.3.4 Clients Registration

This section will describe the series of actions and interactions that happen when an LWM2M client wants to register with the system. Each server in the cluster is given an instance Id during its initialization. This Id will be used later to identify the responsible instance for a specific client.

Figure 3.3 provides an illustration of the steps. First of all, the client issues a Registration Request to the Load Balancer IP address. LVS will then choose a real server instance based on the scheduling policy and create a mapping. All the subsequent communications will happen through the

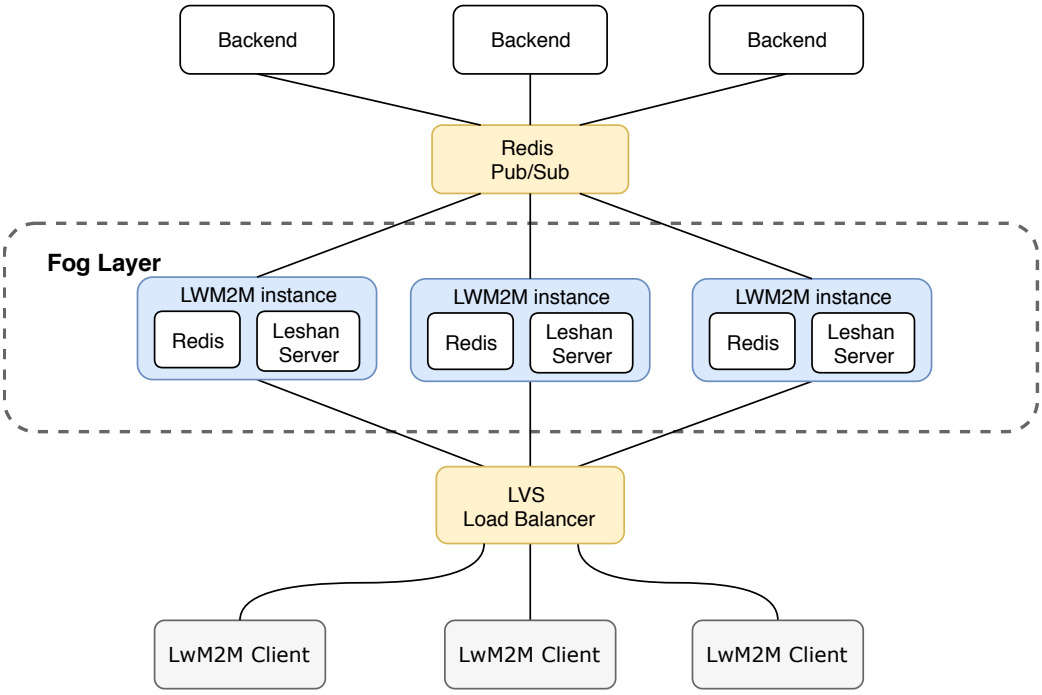


Figure 3.2: Final architecture with components

same instance, unless no message exchange happens over the duration of a configurable timeout. The default setting of 300 seconds was considered reasonable for the purposes of this work.

On the the LWM2M server, a `RegistrationListener` will intercept the request and save the client/server mapping on Redis, under `EP#UID`.

When the registration is ultimated, a registration JSON record, similar to the one shown in Listing 3.3, will be sent over the channel `LESHAN_REG_NEW` on the Redis Pub/Sub broker. In the same way, a registration update or de-registration event will be notified by sending a message on the channels `LESHAN_REG_UPD` and `LESHAN_REG_DEL` respectively. In this way, it is possible to easily create a real-time responsive frontend, using, e.g., WebSocket technologies or similars.

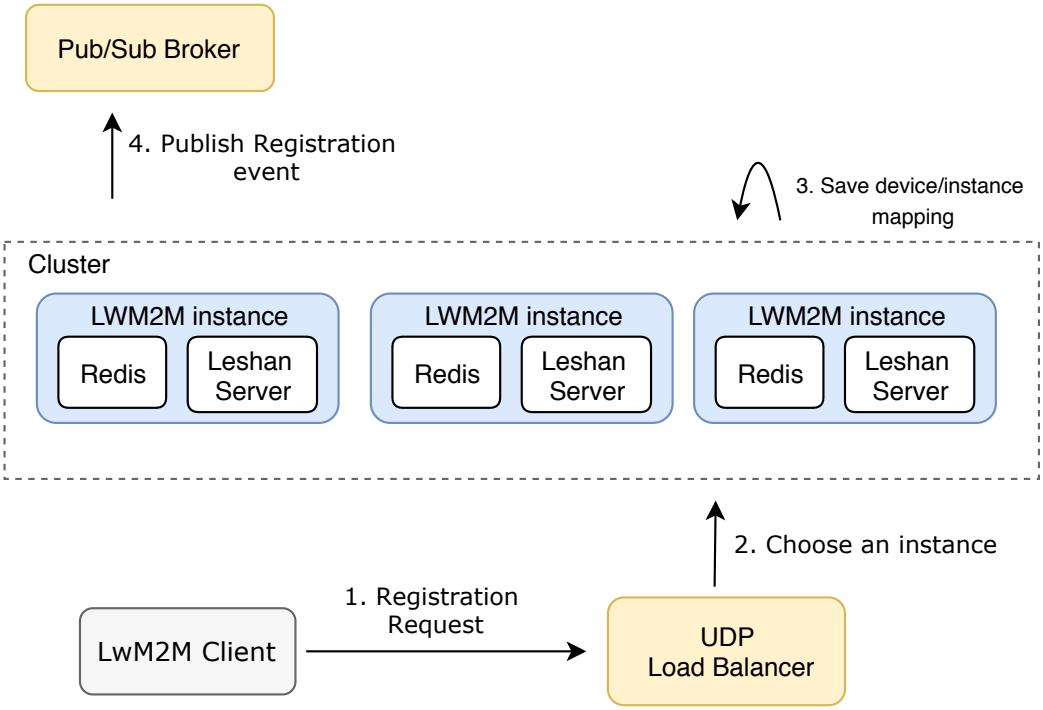


Figure 3.3: Overview of the registration process

3.3.5 Backend Interface

As explained before, an end-user does not send a request directly to a specific LWM2M server, instead the request will be directed to the Pub/Sub broker, which will then forward the request to all the instances. Two Pub/Sub channels are used for this purpose: `LWM2M_REQ` and `LWM2M_RESP`. Basic support for LWM2M read requests was already provided as a proof of concept API by Leshan. This API was extended in order to provide support for LWM2M Observations and Notification forwarding, as well as to enable storing and retrieval of the Security information for DTLS authentication.

3.3.5.1 Requests API

Figure 3.4 illustrates the complete flow of a request and the following resulting response.

Initially, a request in JSON format is sent over the `LWM2M_REQ`. A snippet of a Python script used for testing purposes is shown in Listing 3.5, illustrating its format. A token is attached by the entity making the request, that can be used to recorrelate the response once it will be sent back over the response channel. This is necessary due to the asynchronous and loosely coupled nature of the Pub/Sub paradigm. I decided to use an universally unique identifier (UUID) library to generate the code, in order to guarantee uniqueness (at least locally to the test environment).

Listing 3.5: Example of Observation Request

```
TEST_PATH = "/3303/0/5700"
r = redis.Redis(host=REDIS_URL)
p = r.pubsub()

for endpoint in endpoints:
    uid = uuid.uuid4()
    req = json.dumps({
        "ep": endpoint,
        "ticket": str(uid),
        "req":{
            "kind": "observe",
            "path": TEST_PATH,
            "contentFormat": "JSON",
        }
    })
    r.publish(REQ_CHANNEL, req)
```

Once the request reach the servers, they will do a look-up on Redis to identify if they are responsible for the targeted client or not. If the answer is yes, an acknowledgment message is sent over the Response channel and the request is processed, otherwise the request is simply ignored.

Then, the Registration Store is queried to access the endpoint address and information. The request is then forwarded to the client using the CoAP protocol as required by the standard. Once a response or notification is received by the server, it is sent back on the Response channel along with

the corresponding ticket, using a JSON encoding similar to the one used for the request. In case of an unexpected response, e.g. a resource not existing on the client, a detailed error message is returned instead.

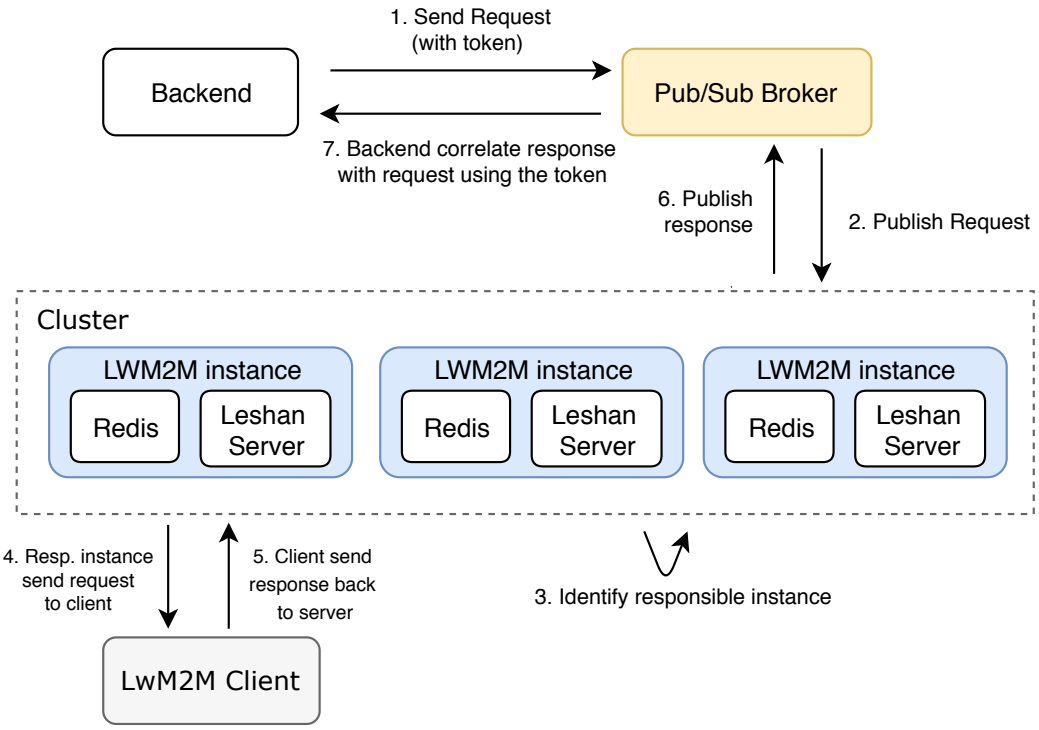


Figure 3.4: Overview of the request/response flow

3.3.5.2 Security Info API

As pointed out before, Leshan provides an implementation for a Security Store, but no end-user interface to manage it. To achieve this, a new kind of request, apart from the ones envisaged by the LWM2M standard, called `sec_info`. Only Pre-Shared-Key type of security has been implemented, as it was sufficient for the evaluation purposes of this work.

3.4 Backend Implementation

In order to test and evaluate the Cluster architecture, a demonstration backend was implemented. Since the LWM2M server cluster exposes its capabilities through a Publish/Subscribe broker, the WebSocket protocol [56] represent a natural and convenient way for the implementation of a Web application backend. Unlike regular HTTP, where a new connection has to be made for every new request, WebSocket provides full-duplex communications channel over a single TCP connection.

It is designed to be used in a regular Web environment and can be used through the JavaScript WebSocket API [57], standardized by the *World Wide Web Consortium* (W3C), that is nowadays implemented on all the major browsers. To maintain compatibility with HTTP, it is designed to work on the same ports (80 and 443), and the handshake uses a request with a special *HTTP Upgrade header* in order to switch protocol.

Once the connection is established, there is a permanent two-way communication channel throughout the session, allowing the server to push data to the client when necessary. This comes handy in this case if we want to receive data and refresh the UI in real-time each time a new registration happens or an observation is sent to the server. Figure 3.5 illustrates the differences between WebSocket and the standard HTTP polling.

A plethora of different framework to implement WebSocket servers exist, for example based on NodeJS (Socket.IO), Ruby (FireHose.IO), Perl (Mojolicio.us, Web::Hippie), and so on.

For my implementation, I decided to use the Java based Spring Framework[7]. This choice was due to several reasons. First of all, the thesis author already had some familiarity with the framework. Plus, the fact that Java is also used in the rest of the platform allows us to reuse some of the code already written to manage the Redis datastore and Pub/Sub module using the Jedis library.

[7] http://spring.io

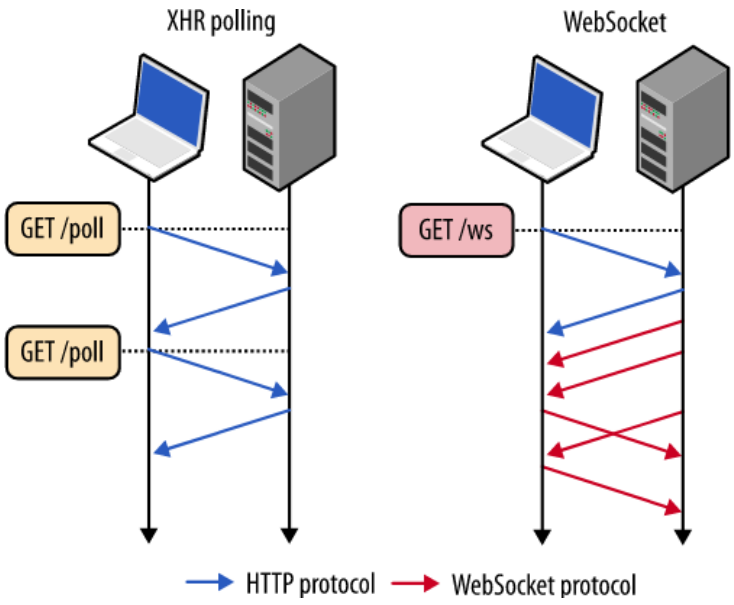


Figure 3.5: Comparison of Websocket vs HTTP polling

But most importantly, Spring already provides an integrated STOMP broker, which comes handy in managing multiple WebSocket connections in an efficient way. Before going into the implementation details, a brief introduction to the Spring Framework is provided.

3.4.1 Spring Framework

Spring is an open source framework designed to reduce the complexity of J2EE development and to promote good programming practices [58]. It can be used to develop any Java application, but it is mostly for building web applications on top of the Java EE (Enterprise Edition) platform. It includes several modules that provide a wide range of features. One of the most distinctive one is the *Dependency Injection*.

The concept of Dependency Injection is also referred to as Inversion of Control. In traditional architectures, every objects that needs some external service to achieve its goal is responsible for obtaining its own dependencies. This strong coupling leads to code that is difficult to test, reuse and maintain.

With dependency injection, this work is delegated to Spring container, a core component that manages the life cycle of objects. Dependencies can be configured explicitly using XML files or implicitly using the `@Autowired` Java Annotation, in which case dependencies are searched using the object type.

The Spring Framework provides a module with special support for web applications, called Spring MVC (Model-View-Controller). It is designed around a *Dispatcher Servlet*, that is responsible to route the requests to specific handlers. These handlers are classes declared using the `@Controller` annotations and their methods are annotated with `@Request mapping`, in order to serve specific URIs. They can either return a JSP/HTML view, or they can be used to create RESTful web services.

From version 4, Spring includes a `spring-websocket` module with comprehensive WebSocket support. It is compatible with the Java WebSocket API standard and also provides additional features such as a fallback option for non-compatible browsers, a messaging architecture and sub-protocol support [59].

3.4.2 STOMP Broker

The main issue with WebSocket is that it only offers a very thin layer over TCP, without offering any information about how to route or process a message. This is different from the standard REST architecture, that relies on having many URLs and several HTTP methods for the routing.

For this reason, a higher application layer protocol is generally used on top of WebSocket. The standard also defines the use of a `Sec-WebSocket-Protocol` in order to agree on a sub-protocol.

The Spring Framework WebSocket module already provides an integrated STOMP broker (Simple Text Oriented Messaging Protocol). STOMP provides "an interoperable wire format so that STOMP clients can communicate with any STOMP message broker to provide easy and widespread messaging interoperability" [60].

It is a frame based protocol, each frame consisting of a command, a set of optional headers and an optional body. A STOMP server is basically a message broker with a series of destinations to which messages can be sent. A STOMP client can act both as a producer, sending messages to a server destination via a `SEND` frame, or as a consumer, sending a `SUBSCRIBE` frame for a given destination and receiving message from the server.

In this way, its nature similar to the Redis Pub/Sub protocol, thus representing a convenient way to implement a Web server backend and tunnel message to the LWM2M cluster backend interface described in the previous section.

Using Spring, setting up a STOMP server and its destination requires few lines of code. Listing 3.6 provides an extract from the configuration class used in this work.

The URL `/ws` is the endpoint to which the browser will send the HTTP Upgrade request for the protocol switch. The option `withSockJS()` is used to enable the fallback option for browsers without WebSocket support. The `/queue` destination is used for one-to-one communication, messages sent to this channel are sent at most to one single client that subscribed to it. Messages sent to the `/topic` destination are instead forwarded to all subscribed clients. The `/app` endpoint is used to send requests that are processed by specific mapped controllers like in Spring MVC.

For the purposes of this implementation, a `RequestController` was created, handling LWM2M requests sent by the end-user in the same JSON format specified in Listing 3.5. The `RequestService` is then in charge of actually processing the request. First of all, the WebSocket session ID is saved together with the request token into a HashMap object. This will be used later to send the response back to the correct client. The request is then serialized and published on the Redis request channel, where it will be then processed by the cluster and sent to the target client.

An end-user has to subscribe to the `/queue/responses` destination in order to receive responses coming back from the client. For this purpose, a

Listing 3.6: STOMP broker configuration in Spring

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements
WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(
    StompEndpointRegistry registry) {
        registry.addEndpoint("/ws").withSockJS();
    }

    @Override
    public void configureMessageBroker(
    MessageBrokerRegistry r) {
        r.enableSimpleBroker("/topic", "/queue");
        r.setApplicationDestinationPrefixes("/app");
    }
}
```

`ResponseService` has been implemented. This is launched at the start of the program and it listens to messages over the Redis response channel. Once a message is received, the corresponding LWM2M request is de-serialized and the HashMap is looked up to identify the session ID of the user who made the request. The response is then sent back to the corresponding user by publishing a message to the respective queue. Figure 3.6 shows a sequence diagram outlining the flow of a request and the different protocols used for the transmission to the LWM2M and backwards.

In the same way, a `RegistrationService` has been implemented to send real-time messages about new registrations, updates or de-registrations. This time the `/topic/registration` destination was used, in order to send the messages to all the connected WebSocket sessions.

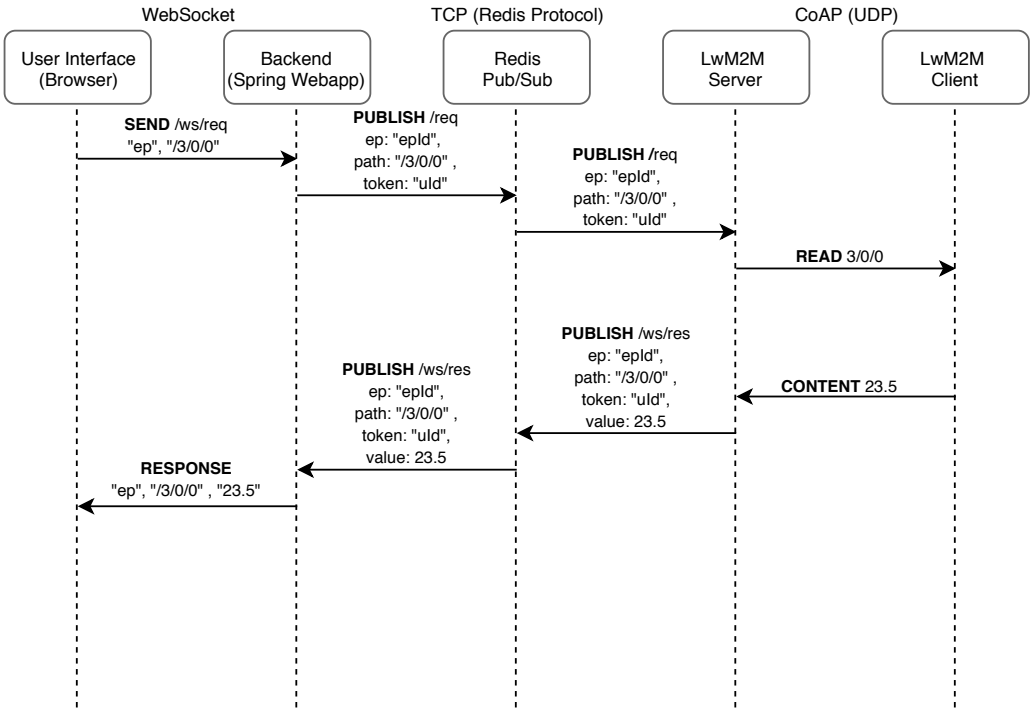


Figure 3.6: Sequence diagram showing the involved protocols

3.4.3 REST API

A REST API was also part of this implementation. Its main function is to provide end-users with the set of registered clients and their details about connected objects and resources. A `RestController` was created using the standard Spring MVC module, that is responsible of retrieving the data from the Redis datastore and sending them to the user in the required JSON format.

An endpoint to serve the set of IPSO standard objects specification has been also implemented. This will be used by the UI to obtain readable information about the resources.

3.5 Frontend Implementation

The implementation of a web based frontend was not strictly required, since, for the performance and scalability evaluation, a specific application was created in order to systematically send request to the backend.

Nevertheless, a simple web user interface has been created as a proof-of-concept and convenient testing purposes. The web interface allows an user to visualize the list of clients registered to the cluster, to obtain detailed information about the contained objects and resource and to make every kind of LWM2M request to them. The implementation is heavily based on the Leshan Demonstration server Web UI and it has been adapted in order to use WebSocket and the STOMP protocol.

The frontend logic is written using AngularJS [61], a model-view-viewmodel (MVVM) JavaScript framework for implementing Single-page applications. The main idea behind the framework is to add custom tag attributes to the HTML page, that are then interpreted as directives to bind input or output parts of the page to a model, also referred to as *scope*. Then special functions called controllers are responsible for defining methods and properties that the view can bind to and interact with. Data is usually retrieved from RESTful web services using singleton "service" objects that are called by the controllers. This helps to decouple user interaction from data management.

For the integration with the WebSocket, the `SockJS` [62] library was used. It provides a WebSocket abstraction object in order to be used with all web browser and also in environments without WebSocket support, in which case technologies like long polling are used.

A new angular service object was implemented to interact with the STOMP broker, using the `stomp-websocket` library [63].

The `ClientList` controller at first subscribes to the registration destinations of the STOMP broker for real-time updates of the interface, then gets the list of the currently connected clients from the REST API. The list saved into the *scope* and automatically bound to the view by Angular. When an user clicks on the name of a client endpoint, the view is changed

and a new controller, `ClientDetail`, is instantiated. A resource tree of the client is built at first, by retrieving the IPSO object specifications from the REST API. This is saved into the model and bound to the view as well. It is then possible to issue LWM2M requests by pressing the respective buttons on the user interface which are mapped to functions sending messages to the STOMP destinations described in the previous section.

Figure 3.7 shows a screen-shot of the client details page.

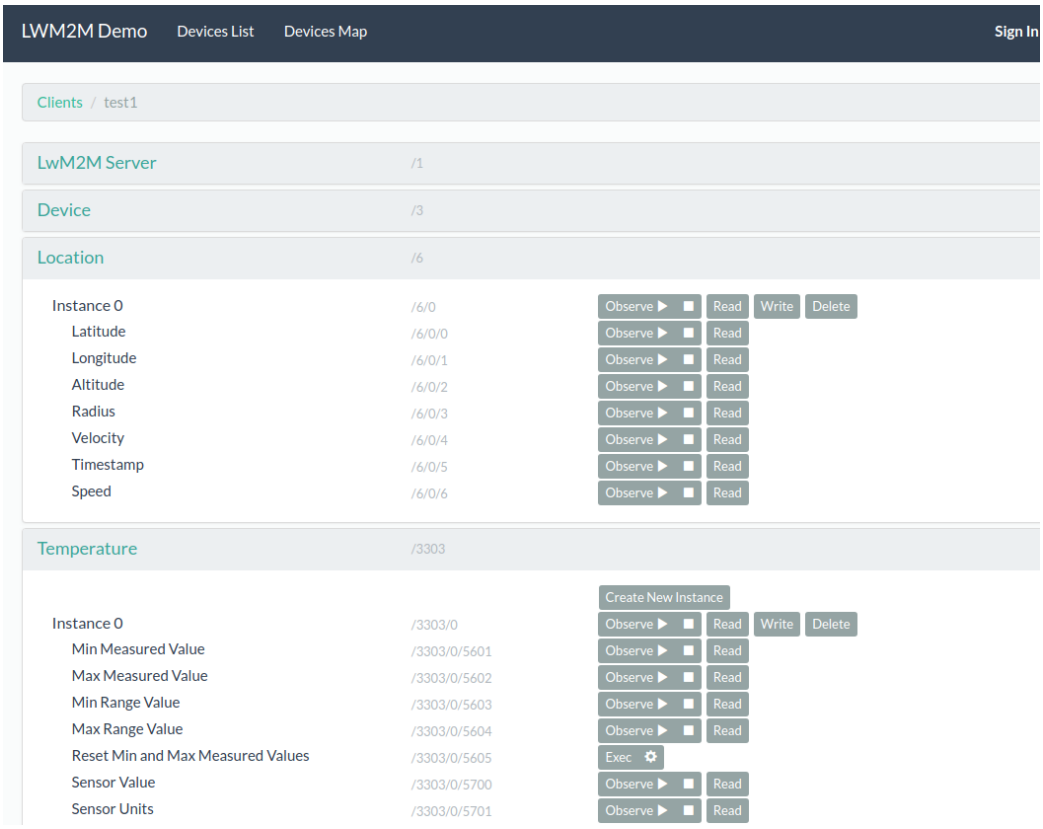


Figure 3.7: Client Details user interface

Chapter 4

Testbed and Evaluation

In this Chapter, I will present the testbed setup and environment, as well as the test application which has been used to evaluate the proposed architecture. In addition, the results gained from the tests are presented and discussed.

4.1 Testbed Implementation

In order to evaluate the performance of the proposed architecture and properly stress the cluster, a great number of devices needs to be deployed. However, carrying out such an analysis using real devices is highly non practical, other than expensive. Managing such a large amount of devices, as well as testing the different IoT scenarios, may be cumbersome with physical testbeds, requiring extensive and time-consuming configurations. Following the work done in [50], I decided to use virtualized Docker containers for the testbed implementation. This allows to have much greater flexibility and better control over the environment. Deploying different test configurations can be done by using few lines of code, leading to considerable savings in time.

The test environment consist of two separate machines, one for the clients emulation and one for the server cluster. The two machines are connected

through a Gigabit Ethernet switch, in order to keep delays due to the network as low as possible and thus obtaining consistent results even with different test configurations. The resulting setup can be observed in Figure 4.1.

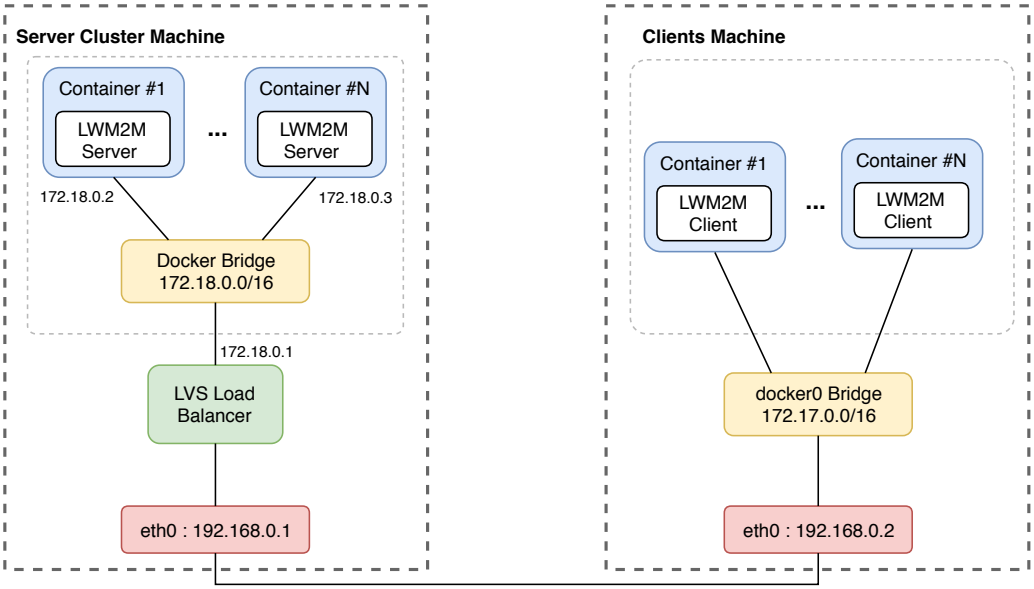


Figure 4.1: Overview of the testbed network infrastructure

4.1.1 Clients Emulation

The machine used for the LWM2M clients emulation is a workstation with 64-bit Ubuntu 16.04 LTS operating system, Intel® Xeon E3-1230 (3.40 GHz x 8) processor and 15.6 GiB of memory. In order to optimize memory consumption for the containers, `alpine-java` was used as a base image in the `Dockerfile`, which is a configuration file containing a set of instructions for building a docker image. This image is based on the lightweight Alpine Linux distribution, with the added minimal requirements to run Java based applications. In this way the result image size was only 103 MB, compared to the 501 MB needed using Ubuntu as the base image.

The LWM2M client implementation follows the guidelines provided by

the Eclipse Leshan demonstration client. The emulated device is a 3-axis accelerometer, with data generated randomly every 50 ms. The device is compliant to the IPSO 3313 Accelerometer[1] model, which is the standard LWM2M object, designed by IPSO Alliance, used to represent an accelerometer sensor and its accessible resources.

With this setup I was able to flawlessly run up to 250 containers, before hitting the memory limit of the machine with the system becoming slow and unstable.

4.1.2 Cluster Setup

The LWM2M server cluster was deployed on a Dell laptop machine, running 64-bit Ubuntu 16.04 LTS, with an Intel® Core™i7-7700HQ CPU (2.80 GHz x8) and 15.5 GiB of RAM. Two different Docker images were created, one for the LVS Load Balancer, and one for the LWM2M servers. In both cases, Alpine Linux was again used as the base image. LVS configuration was already shown in section 3.3.1.1. Since, for LVS to work properly, virtual and real servers have to be on different networks, an user-defined Docker bridge network was created, named `cluster`. Containers can be attached to the network by using the `--network` flag of the `docker run` command. The load balancer belongs both to the default `docker0` bridge and to the cluster network, while the servers are isolated. Ports 5683 and 5684, which are used by CoAP and DTLS respectively, are opened to the external network by using the `EXPOSE` command in the `Dockerfile`.

In order to better represent the behaviour of a Fog node, which is usually a constrained device, I decided to limit containers' resources. This is done thanks to *cgroups* and Linux Completely Fair Scheduler (CFS) Bandwidth Control features. The bandwidth allowed for a group is specified using a quota and period. Within each given "period" (microseconds), a group is allowed to consume only up to "quota" microseconds of CPU time. When the CPU bandwidth consumption of a group exceeds this limit (for that period),

[1] www.openmobilealliance.org/tech/profiles/lwm2m/3313.xml

the tasks belonging to its hierarchy will be throttled and are not allowed to run again until the next period. The `docker run` command allows to set two flags for this purpose: `--cpu-period` and `--cpu-quota`. In this case, the quota was set to 25ms over a period of 100ms, that means a single container will access 25% of the CPU, practically limiting the frequency at 700 MHz. Memory was limited to 512 Mb for each container using cgroups' memory controller capability.

4.2 Test Application

The test application consist of two parts, a Python application and a Bash shell script. The Python application is responsible for formatting and sending the request to the backend, as specified in Section 3.4. `websockets` and `stompy` Python libraries were used for this purpose. Requests are LWM2M read type sent to randomly chosen connected devices and their resources. It is possible to specify the number of requests which are sent per second. During my tests, the application was able to consistently forward up to 500 requests per second (RPS), as confirmed by using *Wireshark*[2] network packet analyzer. However, I decided to limit the maximum number of RPS to 300, since it was already possible to show the desired behaviour. Latency measurements were done saving the timestamp before sending the request in a hashmap, with the relative request token as the key. Once the response is received back, the hashmap is looked-up for the starting time and the difference with the current timestamp is saved onto a file.

The Bash script is used to automatically setup the test environment with different configurations. The number of clients is set to vary from 25 to 225, with a step of 25. The number of server instances in the cluster ranges from 1 to 5, while the RPS sent by the application go from 25 to 300, totaling a number of 540 different combinations. Firstly, the server containers are deployed, and the LVS Load Balancer is setup. Next, the client containers are launched

[2]https://www.wireshark.org/

on the other machine, using `ssh` connections. In order to simulate an heavier load on the cluster, an observation request is sent to every registered client. The test application is then started with a specific RPS parameter and runs for 60 seconds. Once the measurements are done and the results are saved, all the containers are destroyed and a new configuration is established. The same configurations are re-run again with DTLS enabled, in order to evaluate the impact of the security protocol on the cluster performance.

4.2.1 Metrics Collection

Latency

Collection of latency measurements was already described above. In order to improve consistency of the results and reduce fluctuations due to the network jitter and other phenomena, tests are run for 60 seconds and only values below the 95th percentile are kept.

CPU Usage

Since cgroups are used to limit containers' resources, collecting CPU usage metrics is not as straightforward as, for instance, simply using the Linux `top` utility. Cgroups statistics are usually exposed by the Linux kernel by the means of *pseudo-files*, which are usually located under the directory `/sys/fs/cgroup`. In order to identify the cgroup of a specific container, its long ID is used, which can be found using the `docker ps` utility. The pseudo-file exposing CPU metrics is the following:

`/sys/fs/cgroup/cpu/docker/$CONTAINER_ID/cpu.stat`.

`cpu.stat` contains 3 fields: `nr_periods`, which is the number of period intervals (as specified in `--cpu-period`) that have elapsed, `nr_throttled`, counting the number of times tasks in a cgroup have been throttled (that is, not allowed to run because they have exhausted all of the available time as

specified by their quota), and `throttled_time`, showing the total time duration (in nanoseconds) for which tasks in a cgroup have been throttled. The ratio between `nr_throttled` and `nr_periods` during the test execution is collected, which is arguably a close representation of traditional CPU usage.

Memory Consumption

In a way similar to CPU statistics, control groups memory metrics are exposed in the `/sys/fs/cgroup/memory/docker/$CONTAINER_ID/` directory. The `memory.stat` pseudo-file contains a great amount of information. In our case, we will simply collect information contained in the `memory.usage_in_bytes` file, which contains the total amount of memory used by the specified container.

Network Throughput

Unfortunately, network metrics are not exposed directly by control groups. However, since network interfaces exist within the context of network *namespaces*, metrics can be retrieved from within the containers. To accomplish this, it is possible to run an executable from the host environment within the network namespace of a container using the `ip-netns` utility.

For our evaluation purposes, it was sufficient to run the `ip netns exec $CONTAINER_ID netstat -i` command, collecting the amount of traffic transmitted and received by the virtual container interface, at the beginning and at the end of the test. By doing the difference, we obtain the amount of traffic exchanged during the test execution for each server instance, which can then be stored for further analysis.

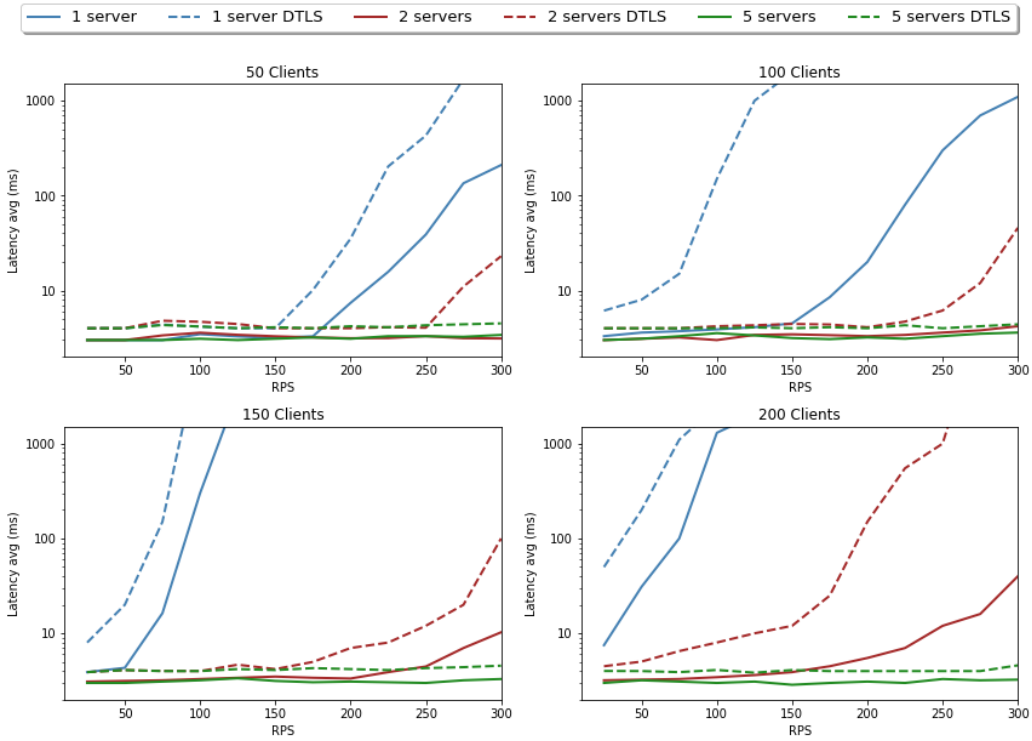


Figure 4.2: Average Response time against Requests per second for 50, 100, 150 and 200 clients configurations

4.3 Results

4.3.1 Latency

The results of the latency evaluation for 4 significant client configurations are depicted in Figure 4.2. They clearly show the scalability of the proposed solution. With a single instance and 50 clients, the response time starts to increase after 175 requests per second, while there are no problems with 2 instances, with an average latency that stands at around 3 ms. Increasing the number of clients, the limit is reached at lower rates, starting to become practically unusable with 200 clients even at 50 RPS. This is due to the fact that the single server is already overloaded with all the automatically sent notifications, from the observation subscription made before the test.

The configuration with 2 server instances, starts to have some latency increase with 150 clients and 300 RPS, reaching a maximum of 34 ms with 200 clients / 300 RPS. With 5 server instances, I was able to conduct all the tests flawlessly, with a stable average latency of 3.5 ms with every configuration.

The use of DTLS had a significant impact over the cluster performance. The limit is reached much earlier with the single instance configuration, generally with 50 RPS less than the plain CoAP counterpart. The same behaviour is observed with 2 servers. It is interesting to notice how the average latency is about 2 ms higher with every configuration, that is a sign of the of the overhead due to the DTLS encryption.

4.3.2 CPU usage

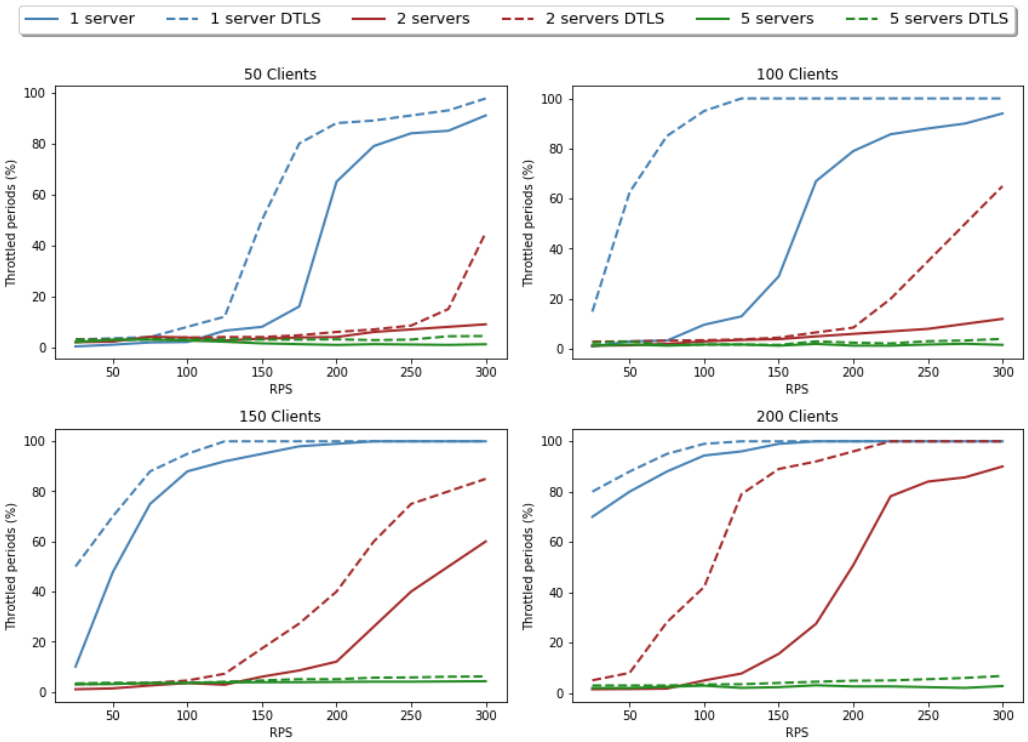


Figure 4.3: Average CPU usage per instance against requests per second for 50, 100, 150 and 200 clients configurations

Figure 4.3 illustrates the obtained results concerning the average CPU usage per each container, as the percentage of throttled periods, like described above. The CPU usage basically reflects the trend observed in the latency evaluation, indicating that the increased response time is due to a lack of processing power. An increase in the number of throttled periods corresponds to higher latency times. Whenever the CPU usage starts to get higher than 80%, requests start to get queued and the response time can become as high as 1500 ms. The results show how the cluster architecture manage to distribute the load between the instances, with the 5 servers configuration keeping very low value even with the heaviest settings.

4.3.3 Memory and Network

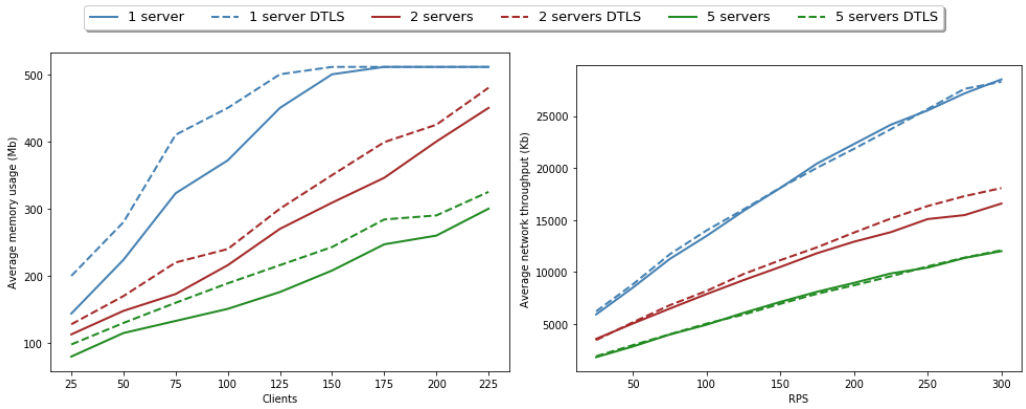


Figure 4.4: Memory usage per container against number of clients (left) and Average network throughput per container (right) with 100 clients

Figure 4.4 illustrates the collected values regarding memory consumption and network traffic. It was observed that memory usage was not affected by the number of requests per second, therefore the plot reports an average between the different rate settings per container. It can be seen that the amount of allocated memory grows quite linearly as the number of clients

increases. Adding more server instances helps in distributing the memory consumption, however, doubling the number of servers does not lead to a reduction in half of the used memory space. That may be due to the Java Virtual machine, that is notably memory hungry and it could be possibly mitigated with a more careful software implementation. It can be noticed that with a single server instance, after 175 clients the memory limit of 512 MB is hit. That may explain the abnormally high latency values obtained in these cases, as the garbage collector could kick in, further slowing down the entire processing.

Regarding the network traffic, it can be seen that we have an almost linear increase as the number of requests per second grows. A very similar behaviour was observed when changing the number of clients, for this reason the figure depicts only the configuration with 100 clients. It is interesting to observe that DTLS had almost no impact on network throughput. An hypothesis could be that DTLS network overhead is outweighed by the upper layer protocol such as LWM2M and WebSockets.

4.3.4 Summary

The obtained results clearly confirm that the proposed architecture is indeed scalable. Despite the solution can be certainly improved, the results are quite promising. Adding more server containers helps to distribute CPU load, memory consumption and network traffic between all the instances. However, it has be noted that the tests were conducted in a controlled environment, where the only additional load was represented by the notifications regularly sent by the devices. In a real-world environment, an edge/fog device could have supplementary duties like filtering and light processing, thus scalability of such a platform will be even more relevant.

Chapter 5

Conclusions

The main goal of this work was to develop an architecture that would be able to solve two main Industrial IoT issues, scalability and interoperability, in order to handle the ever growing number of heterogeneous IoT devices. As a first step, in order to gain a better understanding of the topic, an extensive literature research about the current state of the art of IoT technologies and standardization activities has been conducted. As a result, the LWM2M interoperability standard was identified as the most suitable for its simplicity and maturity, to be used in conjunction with the Edge/Fog Computing paradigm, which aim to solve scalability and latency requirements of industrial applications.

Next, the proposed architecture is presented, describing the idea of combining multiple LWM2M servers forming a cluster, with the aim to be deployed in a Fog scenario. Details about the prototype implementation are then presented. The software applications for LWM2M clients and servers were implemented using the Java based Eclipse Leshan library. One issue was the sharing of information between the instances, to allow global resource discovery and access. This was achieved using Redis, an efficient in-memory NoSQL datastore. An appropriate backend interface had to be implemented as well, in order to correctly manage the requests/responses flow. A Publish/Subscribe broker was implemented for this purpose.

As an interface for end-users, a Web based backend and frontend were implemented, using cutting-edge technologies such as Spring Framework and

AngularJS. WebSocket and STOMP protocols were used for full-duplex communication and for tunnelling requests and responses towards the cluster's backend interface.

As an important part of this work's contributions, a virtualized testbed platform was implemented, in order to evaluate the scalability of the prototype. A large-scale deployment scenario was emulated using Docker containerization and networking capabilities. Several scenarios and configurations were tested, collecting metrics about latency, CPU usage, memory consumption and network throughput. Security was also taken into consideration and the impact of using the DTLS security protocol has been evaluated. The obtained results confirm the scalability of the proposed solution, which is able to efficiently distribute the load among the instances in the cluster.

Overall, it is possible to state that the objectives set at the beginning are met, providing a working prototype of the proposed architecture, which could be revised and extended, in order to be possibly deployed in a real-world environment.

5.1 Future Work

Despite the promising results, this work still has some limitations and there are several possibilities to make improvements. One limitation is that the number of server instances in the cluster is fixed and can be only manually changed. In order to realize a true Fog Computing paradigm, it would be more meaningful to have an architecture that is able to dynamically scale serving instances based on the cluster load. One possible approach, as proposed in [64], could be to implement an additional node, namely, a Fog Manager, that would be able to carry out the scaling process.

Another possible enhancement could be the implementation of the LWM2M Proxy devices proposed in [49]. The proxy enables device group management, which can help in reducing messaging to the LWM2M servers and it could further improve the overall scalability of the solution.

Bibliography

[1] R. Drath and A. Horch. "Industrie 4.0: Hit or Hype? [Industry Forum]". In: *IEEE Industrial Electronics Magazine* 8.2 (June 2014), pp. 56–58.

[2] Dave Evans. "How the Next Evolution of the Internet Is Changing Everything". In: *Cisco White Paper* (2011), p. 11.

[3] World Economic Forum. *Industrial Internet of Things: Unleashing the Potential of Connected Products and Services*. 2015.

[4] Lu Tan and Neng Wang. "Future Internet: The Internet of Things". In: 2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE). Vol. 5. Aug. 2010, pp. V5-376-V5-380.

[5] Barbara Kitchenham. "Guidelines for Performing Systematic Literature Reviews in Software Engineering". In: (), p. 44.

[6] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. "Internet of Things: Vision, Applications and Research Challenges". In: *Ad Hoc Networks* 10.7 (Sept. 1, 2012), pp. 1497–1516.

[7] Arkady Zaslavsky, Charith Perera, and Dimitrios Georgakopoulos. "Sensing as a Service and Big Data". In: *CoRR, Abs/1301.0159*. July 1, 2012.

[8] Ala Al-Fuqaha et al. "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications". In: *IEEE Communications Surveys & Tutorials* 17.4 (24–2015), pp. 2347–2376.

[9] Friedemann Mattern and Christian Floerkemeier. "From the Internet of Computers to the Internet of Things". In: *From Active Data Management to Event-Based Systems and More*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2010, pp. 242–259.

[10] Rick Merritt. *IoT Growth Slower Than Expected*. URL: `https://www.eetimes.com/document.asp?doc_id=1332061` (visited on 04/23/2018).

[11] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. "Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions". In: *Future Generation Computer Systems* 29.7 (Sept. 2013), pp. 1645–1660.

[12] J. Q. Li et al. "Industrial Internet: A Survey on the Enabling Technologies, Applications, and Challenges". In: *IEEE Communications Surveys Tutorials* 19.3 (thirdquarter 2017), pp. 1504–1526.

[13] C. Perera, C. H. Liu, S. Jayawardena, and M. Chen. "A Survey on Internet of Things From Industrial Market Perspective". In: *IEEE Access* 2 (2014), pp. 1660–1679.

[14] Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The Internet of Things: A Survey". In: *Computer Networks* 54.15 (Oct. 2010), pp. 2787–2805.

[15] L. D. Xu, W. He, and S. Li. "Internet of Things in Industries: A Survey". In: *IEEE Transactions on Industrial Informatics* 10.4 (Nov. 2014), pp. 2233–2243.

[16] Pablo Puñal Pereira et al. *Efficient IoT Framework for Industrial Applications*. OCLC: 1026792671. Lulea, 2016.

[17] Accenture. *Driving Unconventional Growth through the Industrial Internet of Things*. 2015.

[18] Deutsche Bank Research. *Industry 4.0: Huge Potential for Value Creation Waiting to Be Tapped*. 2014.

[19] Ahmad-Reza Sadeghi, Christian Wachsmann, and Michael Waidner. "Security and Privacy Challenges in Industrial Internet of Things". In: ACM Press, 2015, pp. 1–6.

[20] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas. "DDoS in the IoT: Mirai and Other Botnets". In: *Computer* 50.7 (2017), pp. 80–84.

[21] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. URL: `https://tools.ietf.org/html/rfc7252` (visited on 05/05/2018).

[22] *Documentation — MQTT*. URL: `http://mqtt.org/documentation` (visited on 06/18/2018).

[23] Shancang Li, Li Da Xu, and Shanshan Zhao. "5G Internet of Things: A Survey". In: *Journal of Industrial Information Integration* (Feb. 2018).

[24] S. Mumtaz et al. "Massive Internet of Things for Industrial Applications: Addressing Wireless IIoT Connectivity Challenges and Ecosystem Fragmentation". In: *IEEE Industrial Electronics Magazine* 11.1 (Mar. 2017), pp. 28–33.

[25] S. Andreev et al. "Understanding the IoT Connectivity Landscape: A Contemporary M2M Radio Technology Roadmap". In: *IEEE Communications Magazine* 53.9 (Sept. 2015), pp. 32–40.

[26] G. A. Akpakwu, B. J. Silva, G. P. Hancke, and A. M. Abu-Mahfouz. "A Survey on 5G Networks for the Internet of Things: Communication Technologies and Challenges". In: *IEEE Access* 6 (2018), pp. 3619–3647.

[27] Maria Rita Palattella et al. "Internet of Things in the 5G Era: Enablers, Architecture, and Business Models". In: *IEEE Journal on Selected Areas in Communications* 34.3 (Mar. 2016), pp. 510–527.

[28] Leonardo Militano et al. "Device-to-Device Communications for 5G Internet of Things". In: *EAI Endorsed Transactions on Internet of Things* 15 (Oct. 26, 2015).

[29] *4G: LTE/LTE-Advanced for Mobile Broadband - 2nd Edition*. URL: https://www.elsevier.com/books/4g-lte-lte-advanced-for-mobile-broadband/dahlman/978-0-12-419985-9 (visited on 05/04/2018).

[30] J. Swetina et al. "Toward a Standardized Common M2M Service Layer Platform: Introduction to oneM2M". In: *IEEE Wireless Communications* 21.3 (June 2014), pp. 20–26.

[31] *Lightweight M2M (LwM2M)*. URL: http://openmobilealliance.org/iot/lightweight-m2m-lwm2m (visited on 02/21/2018).

[32] Jaime Jimenez, Michael Koster, and Hannes Tschofenig. "IPSO Smart Objects". In: *Position paper for the IOT Semantic Interoperability Workshop* (Mar. 2016).

[33] *Eclipse Leshan*. URL: https://www.eclipse.org/leshan/ (visited on 02/22/2018).

[34] *Open Mobile Alliance - Registries*. URL: http://www.openmobilealliance.org/wp/registries.html (visited on 05/05/2018).

[35] *OMA_LwM2M_for_Developers: Public - OMA LightweightM2M Specifications and LwM2M Developer Tool Kit for Public Review*. Apr. 18, 2018.

[36] Eric Rescorla and Nagendra Modadugu. *Datagram Transport Layer Security Version 1.2*. URL: https://tools.ietf.org/html/rfc6347 (visited on 05/05/2018).

[37] *Unified Architecture*. URL: https://opcfoundation.org/about/opc-technologies/opc-ua/ (visited on 06/18/2018).

[38] H. C. Chen and F. J. Lin. "Efficient Device Group Management in oneM2M". In: *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*. 2018 IEEE 4th World Forum on Internet of Things (WF-IoT). Feb. 2018, pp. 439–444.

[39] Nicolas Damour. "Combining LwM2M and oneM2M". In: (), p. 33.

[40] Peter Mell and Tim Grance. "The NIST Definition of Cloud Computing". In: (2011).

[41] W. Shi et al. "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* 3.5 (Oct. 2016), pp. 637–646.

[42] Dave Evans. "The Internet of Things: How the Next Evolution of the Internet Is Changing Everything". In: (Apr. 2011).

[43] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. "Fog Computing and Its Role in the Internet of Things". In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. MCC '12. New York, NY, USA: ACM, 2012, pp. 13–16.

[44] Guenter I. Klas. "Fog Computing and Mobile Edge Cloud Gain Momentum Open Fog Consortium, Etsi Mec and Cloudlets". In: *Google Scholar* (2015).

[45] J. Huai, Q. Li, and C. Hu. "CIVIC: A Hypervisor Based Virtual Computing Environment". In: *2007 International Conference on Parallel Processing Workshops (ICPPW 2007)*. 2007 International Conference on Parallel Processing Workshops (ICPPW 2007). Sept. 2007, pp. 51–51.

[46] Stephen Soltesz, Herbert Pötzl, and Marc E Fiuczynski. "Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors". In: (), p. 13.

[47] *Docker Overview*. June 16, 2018. URL: `https://docs.docker.com/engine/docker-overview/` (visited on 06/19/2018).

[48] G. Tanganelli, C. Vallati, and E. Mingozzi. "Edge-Centric Distributed Discovery and Access in the Internet of Things". In: *IEEE Internet of Things Journal* 5.1 (Feb. 2018), pp. 425–438.

[49] M. I. Robles, D. D'Ambrosio, J. J. Bolonio, and M. Komu. "Device Group Management in Constrained Networks". In: *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*. 2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops). Mar. 2016, pp. 1–6.

[50] A. Mäkinen, J. Jiménez, and R. Morabito. "ELIoT: Design of an Emulated IoT Platform". In: *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*. 2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC). Oct. 2017, pp. 1–7.

[51] C. A. L. Putera and F. J. Lin. "Incorporating OMA Lightweight M2M Protocol in IoT/M2M Standard Architecture". In: *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT). Dec. 2015, pp. 559–564.

[52] *LVS Introduction - Load Balancing Server Cluster*. URL: `http://www.linuxvirtualserver.org/whatis.html` (visited on 06/27/2018).

[53] *Kernel Load Balancing for Docker Containers Using IPVS*. Nov. 17, 2015. URL: `https://blog.codeship.com/kernel-load-balancing-for-docker-containers-using-ipvs/` (visited on 06/22/2018).

[54] *DB-Engines Ranking - Popularity Ranking of Database Management Systems*. URL: `https://db-engines.com/en/ranking` (visited on 06/22/2018).

[55] *Pub/Sub – Redis*. URL: `https://redis.io/topics/pubsub` (visited on 06/23/2018).

[56] Ian Fette <ifette+ietf@google.com>. *The WebSocket Protocol*. URL: `https://tools.ietf.org/html/rfc6455` (visited on 06/25/2018).

[57] *The WebSocket API*. URL: `https://www.w3.org/TR/websockets/` (visited on 06/25/2018).

[58] *Spring Framework Documentation*. URL: https://docs.spring.io/spring/docs/5.0.7.RELEASE/spring-framework-reference/ (visited on 06/25/2018).

[59] *22. WebSocket Support*. URL: https://docs.spring.io/spring/docs/5.0.0.BUILD-SNAPSHOT/spring-framework-reference/html/websocket.html (visited on 06/25/2018).

[60] *STOMP*. URL: http://stomp.github.io/ (visited on 06/26/2018).

[61] *AngularJS — Superheroic JavaScript MVW Framework*. URL: https://angularjs.org/ (visited on 06/26/2018).

[62] *Sockjs-Client: WebSocket Emulation - Javascript Client*. June 26, 2018.

[63] *STOMP Over WebSocket*. URL: http://jmesnil.net/stomp-websocket/doc/ (visited on 06/26/2018).

[64] C. L. Tseng and F. J. Lin. "Extending Scalability of IoT/M2M Platforms with Fog Computing". In: *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*. 2018 IEEE 4th World Forum on Internet of Things (WF-IoT). Feb. 2018, pp. 825–830.