Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
PERCCOM Master Program

Master's Thesis in
Pervasive Computing & COMmunications
for sustainable development

Victor Estuardo Araujo Soto

PERFORMANCE EVALUATION OF SCALABLE AND DISTRIBUTED IOT PLATFORMS FOR SMART REGIONS

2017

Supervisors: *Dr. Karan Mitra* (Luleå University of Technology)
Dr. Saguna Saguna (Luleå University of Technology)
Professor Christer Åhlund (Luleå University of Technology)

Examiners: *Professor Eric Rondeau* (University of Lorraine)
Professor Jari Porras (Lappeenranta University of Technology)
Associate Professor Karl Andersson (Luleå University of Technology)

This thesis is prepared as part of an European Erasmus Mundus programme PERCCOM - Pervasive Computing & COMmunications for sustainable development.

Co-funded by the Erasmus+ Programme of the European Union

This thesis has been accepted by partner institutions of the consortium (cf. UDL-DAJ, n°1524, 2012 PERCCOM agreement).

Successful defense of this thesis is obligatory for graduation with the following national diplomas:

- Master in Complex Systems Engineering (University of Lorraine)
- Master of Science in Technology (Lappeenranta University of Technology)
- Degree of Master of Science (120 credits) –Major: Computer Science and Engineering, Specialisation: Pervasive Computing and Communications for Sustainable Development (Luleå University of Technology)

ABSTRACT

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
PERCCOM Master Program

Victor Estuardo Araujo Soto

Performance Evaluation of Scalable and Distributed IoT Platforms for Smart Regions

Master's Thesis

2017

103 pages, 56 figures, 18 tables.

Examiners: *Professor Eric Rondeau* (University of Lorraine)
Professor Jari Porras (Lappeenranta University of Technology)
Associate Professor Karl Andersson (Luleå University of Technology)

Keywords: IoT; performance; smart cities; device management; FIWARE;

As the vision of the Internet of Things (IoT) becomes a reality, thousands of devices will be connected to IoT platforms in smart cities and regions. These devices will actively send data updates to cloud-based platforms, as part of smart applications in domains like healthcare, traffic and pollution monitoring. Therefore, it is important to study the ability of modern IoT systems to handle high rates of data updates coming from devices. In this work we evaluated the performance of components of the Internet of Things Services Enablement Architecture of the European initiative FIWARE. We developed a testbed that is able to inject data updates using MQTT and the CoAP-based Lightweight M2M protocols, simulating large scale IoT deployments. Our extensive tests considered the vertical and horizontal scalability of the components of the platform. Our results found the limits of the components when handling the load, and the scaling strategies that should be targeted by implementers. We found that vertical scaling is not an effective strategy in comparison to the gains achieved by horizontally scaling the database layer. We reflect about the load testing methodology for IoT systems, the scalability needs of different layers and conclude with future challenges in this topic.

ACKNOWLEDGEMENT

I dedicate this thesis to my family. My mother Mirsa, my father Victor and my sister Gaby are the greatest influence in my life. Your love is always present no matter where I am. Thanks for loving me unconditionally, supporting me when I felt lost and believing in me always. As a 5-year-old, I used to cry and complain about having to repeatedly write the alphabet to learn it. My mother patiently guided me through that first educational challenge. Between that and this thesis, there were many more hurdles, but my family was always there. All the members of my family were my first teachers and I will never stop learning from them.

I want to express my heartfelt gratitude to my supervisors. I had a great introduction to research. Thank you for sharing your knowledge, experience and showing me the way to improve my work. From their encouragement to begin writing early to their dedication to track and polish the small details, their guidance was vital for this work. Professor Karan Mitra closely followed my technical progress and constantly helped me improve its robustness. Professor Saguna Saguna constantly taught me about the way to present my ideas more clearly and develop the written parts. Professor Christer Åhlund always had great questions and insights about the big picture. The ski and research trip to Hemavan (great combination!) that he organized is a memory of Sweden that I will never forget.

Thanks to the PERCCOM Consortium for this opportunity. Professors Eric Rondeau, Jean-Philippe Georges, Jari Porras and Karl Andersson taught us so much about academics and also about their respective countries and cities. The academic and cultural activities were very enriching. Many thanks too to professors Andrei Rybin, Oleg Sadov, Alexandra Klimova, Ah-Lian Kor and Colin Pattinson, who taught seminars and hosted the Summer Schools.

Last but not least, thanks to the professors in every course and all the wonderful friends I met during this journey.

Skellefteå, June 1, 2017

Victor Estuardo Araujo Soto

CONTENTS

List of Figures

List of Tables

ABBREVIATIONS AND SYMBOLS

API	Application Program Interface
AWS	Amazon Web Services
CB	Context Broker
CoAP	Constrained Application Protocol
CPU	Central Processing Unit
EC2	Elastic Compute Cloud
ELB	Elastic Load Balancing
EU	European Union
HTTP	Hypertext Transfer Protocol
ICT	Information and communications technology
IP	Internet Protocol
IETF	Internet Engineering Task Force
IO	Input/Output
JSON	Javascript Object Notation
IoT	Internet of Things
M2M	Machine to Machine
NGSI	Next Generation Services Interface
LWM2M	Lightweight M2M
LB	Load Balancer
OMA	Open Mobile Alliance
REST	Representational State Transfer
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

1 Introduction

This chapter develops the context in which this thesis takes place, describing the background around the Internet of Things, smart cities and its potential for sustainability. The research questions, aims and contributions are described, and an outline for the whole content of this thesis is presented at the end of this chapter.

1.1 Background

The digital revolution has brought deep changes to our economies, societies and lifestyles. Digital technology has permeated our lives and transformed our world in ways that few would have imagined a century ago. Active development and research continues in areas of computing as diverse as artificial intelligence, distributed systems and human-computer interaction. Many of the upcoming applications and visions for digital technology have the potential to create transformations as disruptive as we have seen during the last few decades with the personal computer, the World Wide Web and smartphones.

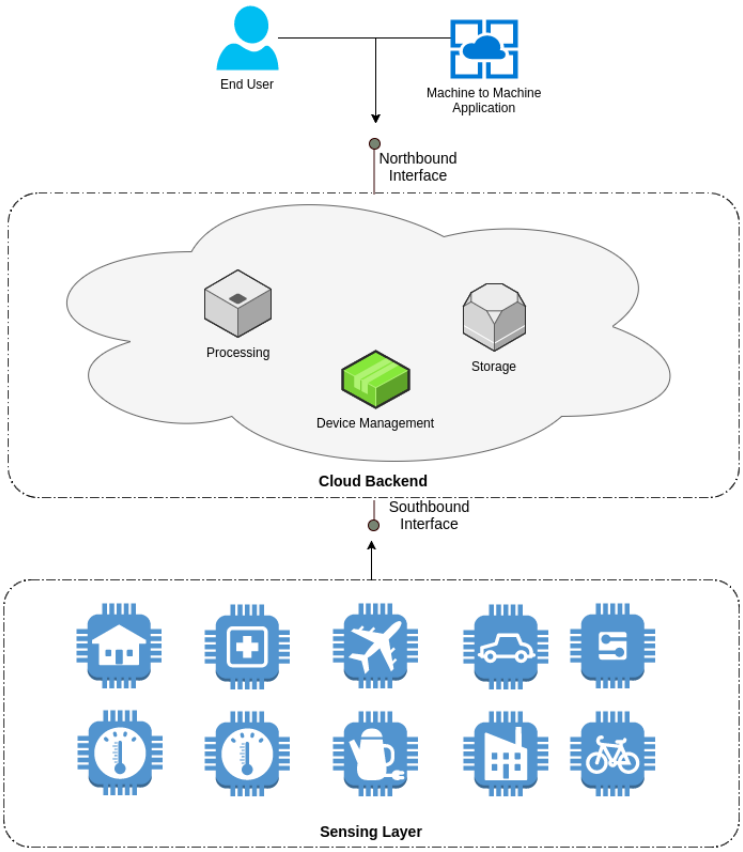
One of these upcoming technologies is the vision of the Internet of Things (IoT) [1]. In this vision, billions of sensors, actuators and everyday objects are expected be connected to the Internet, thus being able to report about the conditions in their surroundings and to act on their environments. These interactions will mostly happen without human intervention, paving the way for smart objects and applications that are able to measure, regulate and optimize their environments or their own operation [2].

The potential of this vision has been explored in several domains, including healthcare, logistics, transportation and smart cities [3]. The concept of "smart cities" envisions safe, efficient and environmentally green urban centers for the future. It is a concept that has received widespread attention in recent years from policymakers, sociologists, economists and researchers in Information and Communication Technologies (ICT). The concept references a wide spectrum of solutions with many different goals. Nonetheless, the use of digital technologies to tackle urban problems like traffic, pollution and city crowding are a common subject. We adopt the definition from [4], which states that *"a smart city is a well defined geographical area, in which high technologies such as ICT, logistic, energy production, and so on, cooperate to create benefits for citizens in terms of well being, inclusion and participation, environmental quality, intelligent development; it is governed by a well defined pool of subjects, able to state the rules and policy for the city government and development"*.

The pervasive monitoring and actuation capabilities of the Internet of Things can fit to monitor and manage the resources, the living spaces and the infrastructure of cities. According to this vision, data about a city's critical infrastructure can be collected and analyzed constantly to guide decision-making, efficient management, optimal allocation of resources and timely responses to accidents or disasters. This type of monitoring could extend to vehicles in the transportation system, roads, bridges, railways, airports, seaports, communications facilities, buildings, households, water distribution systems and power installations. Furthermore, the quality of water, air and lighting can be monitored and adjusted by sensors and actuators in the Internet of Things to provide better living conditions for citizens [5]. Examples of possible applications that have been implemented in smart cities [3] are shown in figure 1. These applications don't have to be confined to the administrative or geographical area of a city, prompting the discussion and development of smart regions [6] [7] that extend these ideas to whole regions.

Figure 1. Examples of applications in smart cities.

Turning this into a reality involves deploying numerous devices that are able to gather data and possibly act on their environments. These devices, tipically sensors and actuators that are constrained in terms of energy and computing capabilities, form a sensing layer that will be ubiquituosly deployed [8]. Storage and processing of the data from devices, as well as management of the devices, needs to be done in a suitable computing environment that provides desirable characteristics like scalability and security. The characteristics of Cloud Computing [9] make it a suitable choice for this function, with many IoT-related functionalities and applications already deployed in cloud systems [10]. Thus, backends for IoT are often deployed in the Cloud. IoT platforms in these cloud backends may provide a southbound interface that interacts with the sensing layer and a Northbound interface for interaction with higher-level services, Machine-to-Machine (M2M) applications or end users (figure 2).



Diverse sensors and smart devices send data updates for storage and processing to IoT platforms in cloud backends through a southbound interface. These platforms provide services to end-users and M2M applications through Northbound interfaces. Tipically, sensors and smart devices are physically located in the same city, while cloud backends might be in another city or country.

Figure 2. Common Cloud-Based IoT High-Level Architecture

1.2 Motivation

Components of the Internet of Things (IoT) will need to be able to scale to handle the hundreds of millions of devices that are expected to be connected to the Internet in upcoming years [1]. The vision for the Internet of Things and smart cities hinges on the capacity to handle thousands of devices pushing data continuously from the sensing layer (or southbound interfaces) to cloud-based device management components and M2M applications. The key research challenge addressed in this thesis is the evaluation of the performance and load scalability of a cloud-based IoT platform for smart cities. This evaluation of performance and scalability considers the capacity of the platform to handle data update loads, which mimic the thousands of connnected data points and devices that will be sending information as part of future smart regions. In this context, performance is defined using metrics related to latency, throughput and resource

utilization of components of the platform. Scalability is the characteristic of the system that allows it to improve its load-handling capacity [11].

We consider our scenario within the context of the Sense Smart Region project [7], a collaboration between companies, local governments and a university to develop a smart region in Northern Sweden. The Sense Smart Region project seeks to combine data from sensors and open data into useful information, in line with many smart city initiatives around the globe. Another of its stated goals is to present this information with augmented reality to the end users. For example, consider a healthcare monitoring application for elderly patients. It might be used by a city or region to improve healthcare services to remote and vulnerable populations and to reduce costs. Such a system can consist of different sensors that transmit information of motion, temperature and vibrations to a cloud platform using IoT protocols [12]. Wearable devices that monitor hearth rate, blood pressure, and blood sugar levels [13] can continuosly send data to the cloud platform for storage or analysis. Applications of this type are currently being implemented [12], and the IoT platforms and applications that handle this data need to be prepared to deal with large-scale deployments of devices in the following years.

In this context, we consider a scenario where sensors connect to a cloud-based IoT environment based on the FIWARE platform [14]. In our scenario, the interaction with sensors and their data is handled by a cloud backend based on FIWARE components, which continuously receives data from the sensing layer to its southbound interface (please refer to figure 2). FIWARE seeks to be an enabler of Future Internet technologies, including particular domains like smart cities (and smart regions by extension). We evaluate its performance when handling large rates of data updates reaching the southbound interface.

We aim to go beyond previous smaller-scale tests [15] and study the performance of these components when handling larger levels of load. We seek to evaluate the robustness of the components and the underlying architecture, by verifying that the system behaves as expected without errors or service degradations. This can uncover potential scalability issues of the platform, which might compromise its ability to deal with the large-scale IoT deployments that are expected in the following years.

1.3 Thesis Aim

This thesis aims to understand the performance and scalability of a cloud-based IoT platform based on FIWARE, a European initiative for the Future Internet [16]. Performance of the platform is evaluated in the context of a smart region, considering the load coming from devices

that actively push their data to the cloud. In particular, it aims to ascertain whether the platform is well-suited to handle large rates of data coming from devices, a key requirement for the future Internet of Things [17]. The performance of the platform under comprehensive vertical and horizontal scalability configurations is analyzed, using metrics related to latency, throughput and resource utilization of the platform's components.

1.4 Delimitations

The Internet of Things will cover a wide range of applications. Applications for smart cities in the literature range from healthcare monitoring of citizens to traffic management [3] . Many of these applications are either only a concept or have only been deployed on a small scale. From the large space of IoT protocols and traffic characteristics, we have to limit our tests to a few choices that are the most representative according to our analysis.

The data path from the devices to the final storage crosses many networks, operating systems and applications. The focus of our analysis is in the application components of the platform in our scenario, and their deployment in a cloud environment. In particular, we limit ourselves to study the FIWARE platform and the particular components of its Internet of Things Services Enablement Architecture that are relevant for our scenario.

The scenario that we test focuses on the performance of the updates that are coming from the southbound interfaces. Particular deployments can have different read and write loads coming from southbound and northbound interfaces, where users or M2M applications can interact with either the devices or the data store in patterns that are application-dependant.

1.5 Research Questions and Methods

To achieve the aim stated above, this thesis raises the following research questions:

1. What is the performance of the FIWARE-based platform of our scenario, when handling a large rate of data updates from the southbound interface; in terms of throughput, computational resources and latencies?
2. How does the platform perform when following the vertical and horizontal scaling strategies for cloud systems, to serve the scale of current and future IoT deployments?

To answer the research questions, we followed an experimental methodology based on the development of a testbed, which allows the user to deploy the components of the platform using multiple configurations. The testbed was built according to FIWARE's Internet of Things Services Enablement Architecture [18]. Thus, the testbed is well aligned with FIWARE specifications and performance evaluations on it reflect the characteristics of the FIWARE platform. It allows us to answer the research questions by observing the actual behavior of the FIWARE components, avoiding the limitations of analytical models and simulations to evaluate its performance [19]. Experiments in the testbed are run using load tests, to evaluate the performance of the different configurations of the platform. These load tests are based on the generation of workloads to stress the system while its performance is monitored, providing the data and observations needed to answer the research questions.

1.6 Thesis Contributions

The contribution of this thesis is an evaluation of device managament and M2M components of a cloud-based IoT platform, regarding their scalability and their capacity to handle the data that is pushed from southbound interfaces. In particular, the following outcomes were achieved and are included in this work:

1. A cloud-based testbed based on the Internet of Things (IoT) Services Enablement Architecture of FIWARE. Our testbed can generate MQTT and LWM2M traffic, enabling large-scale testing without the need to use thousands of physical devices.
2. A comprehensive analysis of the current capabilities and limitations of the platform, when handling large rates of data updates from devices using different scaling strategies in the cloud.

The research questions were answered with a comprehensive set of experiments, which provide valuable information about the performance of the platform, its architectural limitations, and possible strategies to deploy it in cloud environments. FIWARE aims to be an enabler for smart cities and other future technologies [16]. As such, our performance evaluation can help smart city implementers to know its capabilities and limitations. This evaluation arose from the context of the Sense Smart Region project for smart regions in Northern Sweden. However, our evaluation is general enough to be applicable to other use cases.

1.7 Sustainability Aspects

Sustainability first emerged in environmental, economic and social discussions during the late 1970s, in a time when environmental concerns had begun to grow. During the previous decade, books like Rachel Carson's The Silent Spring had exposed environmental issues and concerns to the public, making interest and consciousness around the topic of the environment rise. Sustainability has gained considerable prominence in recent years due to the discussions about climate change and the threats posed by it to our societies and economies [20].

One approach that has found an echo within business ethics and corporate social responsibility is the triple bottom line, which is used to conceptualize sustainability along three axes involving "people, planet and profit". It is based on stakeholder theory, a management and ethical approach for conducting businesses that takes into account the different stakeholders affected by the organization's actions. In contrast, the "shareholder view" has as the major or only goal to maximize the profit of the organization's shareholders [21].

In the triple bottom line, the "people" axis measures the benefits and fairness to the population of a community where an organization operates. The "planet" axis deals with the environmental impacts, where an ecological footprint is analyzed. Techniques such as life cycle analyses are used to estimate ecological footprints of products and services. Finally, the "profit" axis is concerned with the financial sustainability of an operation [22] [21].

The main impacts are indirect, since the concrete sustainability effects will depend mostly on the success of the implementation of smart applications and smart cities. Table 1 shows potential benefits to sustainability from common applications in smart cities [3]. Performance testing is supplementary for these goals, to allow the implementations to scale and the resource usage to be optimized. Complex second-order and third-order effects [23] can also arise from IoT technologies and are yet to be determined. This work can have an indirect role in these effects, by acting as an enabler for increasing the scale of smart systems.

More concrete sustainability impacts are possible from this line of work for optimizing the operation of related software and systems. Our performance characterization provides an account of the current usage of computational resources in the platform under the conditios of our tests, which can suggest ways to perform optimizations. If the results can be used to improve efficiency of the system, costs of operation and environmental impacts can be reduced.

As discussed previously in this section, this work contributes to the larger research efforts in the Internet of Things and smart cities. We have outlined some of the benefits that can be gained

Table 1. Sustainability potential of smart city applications

Application	Main benefits	Intended effects
Air quality monitoring	People	Warn citizens about hazardous conditions
Emergency response	People	Timely response to disasters to save lives
Remote healthcare	People, profit	Improve health coverage and reduce costs
Structural health	Profit	Improve maintenance of city infrastructure and reduce costs
Smart parking	People	Better experience for drivers
Smart lighting	Planet, profit	Reduce costs and energy consumption
Traffic congestion	Planet, people	Reduce carbon emissions and time in traffic

directly by optimizing existing software systems and indirectly through the role of performance evaluations as enablers for smart applications.

1.8 Thesis outline

We now present an outline of the content of the next chapters in this thesis.

Chapter 2 contains the theoretical background and the literature review, which explain the technical context of the work in this thesis. We cover the Internet of Things, the usage of cloud computing to host components of the Internet of Things, the FIWARE platform and computer performance analysis. We include a review of previous related work at the end of the chapter.

Chapter 3 describes the testbed that we implemented, as well as the procedure that we followed for the load tests and the configurations used for all the components in the testbed.

Chapter 4 presents the performance evaluation results. We describe the results obtained for each configuration, and discuss the outcomes in light of the practical considerations to serve future smart applications. We also compare our results to previous testing efforts.

In Chapter 5 we finish by presenting our conclusions and recommendations for future research.

2 Background and Related Work

This chapter describes the technical background and the state-of-the-art which sustains the work in this thesis. The topics covered begin with the Internet of Things, its current architectural trends and the IoT protocols that are used as part of our performance evaluation of the FIWARE platform. The necessary context about cloud computing, performance analyses and the architecture of FIWARE for IoT is provided to act as a reference for the interested reader. Finally, the related work to performance evaluations of the FIWARE platform in the context of the IoT is examined.

2.1 The Internet of Things

The Internet of Things (IoT; also known as Internet of Objects) is a concept that was first developed by Kevin Ashton in 1999 in the context of supply chain management, where he described a system in which the physical world is connected to the Internet through ubiqituous sensors. This concept has evolved and extended to several industries and applications over the years, encompassing a vision where networked everyday objects are able to push information, change their state, interact with each other without human intervention and perform actions that affect the physical world [2].

Research and development of this vision is active in several industries. Applications have been developed in industries ranging from healthcare to transportation, and in recent years the integration of objects, sensors and cloud computing has been an active area of research [8]. However, the development of these technologies is still in its infancy and many challenges lie ahead. Research continues on standardization, security, infrastructure, interfaces and communication protocols, with several competing implementations and organizations.

The vision of the Internet of Things has several deep implications for our societies. As a market, the whole annual economic impact caused by the IoT could have a value of up to 2-6 trillion USD by 2025. More than 200 billion smart IoT objects are expected to be deployed by the beginning of the next decade and their machine-to-machine (M2M) traffic flows are expected to make up around 45% of Internet traffic. Moreover, the benefits are expected to be not only economic, with several applications geared towards improving the quality of life in aspects such as healthcare or life in cities [1].

Some of the visions for the Internet of Things also include sustainability aspects. The ICT

sector in general and the Internet of Things in particular have the potential to drive efficiency gains across the economy, by allowing consumers and businesses to measure and optimize their actions. It has been estimated that 15% of global carbon emissions could be reduced through the usage of smart technologies by 2020 [24].

In this chapter we begin with a discussion about the architectural elements of the Internet of Things in its current state. Next, we describe the role of middleware and frameworks in the IoT landscape. Finally, we introduce the FIWARE platform [16] that is used in our scenario, which is an enabler for the deployment of architectural elements and middleware components to build smart applications that are IoT-enabled.

2.2 Internet of Things Architecture

The current state of the Internet of Things is characterized by a diverse set of initiatives, standards and implementations. Standardization and interoperability remain a challenge. Initial applications have been developed in vertical domains like logistics or energy, with their own protocols and architectures. Current efforts are converging into standard specifications like OMA LWM2M and reference architectures like the European Internet of Things-Architecture (IoT-A). These efforts are bringing together key players and enabling implementations that are reusable accross different application domains.

2.2.1 Internet of Things Elements

The elements that make up the Internet of Things in current and future applications have been discussed extensively throughout the literature and have been categorized by several authors. We describe next the most important components and architectural elements in the Internet of Things, as described and categorized in the extensive surveys performed by Gubbi *et al* [2], Li *et al* [25] and Al-Fuqaha *et al* [1].

Sensing elements exist at the lowest level, which begins with the "things" themselves: the sensors, actuators and other devices that make part of the Internet of Things. Hundreds of millions of such devices are expected to make part of the Internet of Things, and special identification and addressing schemes are necessary due to the large scale of their deployments and their often constrained nature in terms of energy and computational resources.

Communication elements are needed to connect the heterogenous devices that provide smart

services. Some of the technologies that are used for communication include Radio Frequency Identification (RFID), wireless sensor networks (WSNs) and protocols like IEEE 802.11 (Wi-Fi), IEEE 802.15 (Bluetooth) and IETF Low power Wireless Personal Area Networks (6LoWPAN).

Data storage and analytics are increasingly needed to deal with the generated data. However, more than raw data is needed for smart applications. Data with meaning (knowledge) is produced through analytics, machine learning and context processing elements in many applications, often through cloud deployments [2].

Services are built on top of the previous elements. Al-Fuqaha *et al* [1] make an overall classification of services into Identity-related Services, Information Aggregation services, Collaboration-Aware Services and Ubiquitous Services. Under this classification, the function of Identity-related Services is to identify deployed objects. Information Aggregation Services are the ones responsible to collect and summarize data from sensors. Collaborative-Aware Services use data from Information Aggregation Services to make decisions, implementing the smartness in applications. Ubiquitous Services are an extension of Collaborative-Aware Services, and will realize the vision of the Internet of Things, by providing the previous services anywhere to any user that might need them [1].

Li et al [25] group these elements into a generic service-oriented architecture, consisting of a sensing layer, a network layer, a service layer and interfaces layer. The sensing layer brings together physical objects to sense the status of things, while the network layer is the infrastructure that supports communication among things. Next, the service layer provides the services required by applications and users. Finally, the interfaces layer defines the interaction methods with users and applications.

2.3 Middleware and Frameworks for IoT

Middleware enables interactions between heterogeneous devices, by providing a connectivity layer that is used by connected objects and by the application layers that provide services in the Internet of Things. Middleware supports protocols, services and Application Program Interfaces (APIs), providing a standardized environment that bridges the differences in devices, networks and applications that need to interact in the Internet of Things. The functionalities provided by middleware can span a wide range of the elements of the Internet of Things, from simple device connectivity to advanced messaging and data analytics models.

The landscape for middleware is evolving at a fast pace. Several challenges are being addressed, related to requirements like scalability, security and interoperability. Existing solutions have focused on different technical approaches that include publish/subscribe models, service oriented architectures, semantic models and context-awareness [26]. Moreover, several commercial and open source implementations for real-world deployments have surfaced in recent years. These projects are not mere research prototypes and aim to be widely adopted. Frameworks and technology ecosystems are built around these initiatives, with the support of large companies and non-commercial organizations that include standards bodies, government organizations and industrial alliances. Some examples are Alphabet's (Google) Thread, the Linux Foundation's IoTivity and the Open Mobile Alliance's Lightweight M2M [27]. More recent examples include WSO2's open-source IoT Server middleware, and the European initiative FIWARE which is part of our scenario for smart cities.

2.4 The FIWARE Platform

FIWARE is a project sponsored by the European Commission to develop technologies for the Internet of Things and the future Internet, in collaboration with participants from the ICT sector [16]. FIWARE's mission statement is "to build an open sustainable ecosystem around public, royalty-free and implementation-driven software platform standards that will ease the development of new Smart Applications in multiple sectors".

The FIWARE initiative provides a set of APIs to develop smart applications, based on open and royalty-free specifications. Open source reference implementations of FIWARE components are provided, though the licensing schemes are not homogeneous accross all products because of the large amount of contributing organizations.

FIWARE is based on the concepts from the previous SENSEI project and the IoT-A architecture, a European initiative to define an architecture for the Future Internet that is reusable across different domains[28]. FIWARE specifies components called Generic Enablers that constitute the building blocks for smart applications in different areas. These Generic Enablers cover aspects that include but are not limited to security, data management, cloud hosting and device management. We provide next a description of the components that are used in our scenario.

2.4.1 Internet of Things (IoT) Services Enablement Architecture

FIWARE defines an Internet of Things Services Enablement Architecture for the scenarios that need to connect devices or other data points to other components in the FIWARE platform [18]. This reference architecture is currently on revision 5. It identifies several scenarios with devices or gateways that might use different protocols. In order to connect devices that use different common IoT protocols to the platform, it specifies two mandatory generic enablers which we detail next in sections 2.4.2 and 2.4.3.

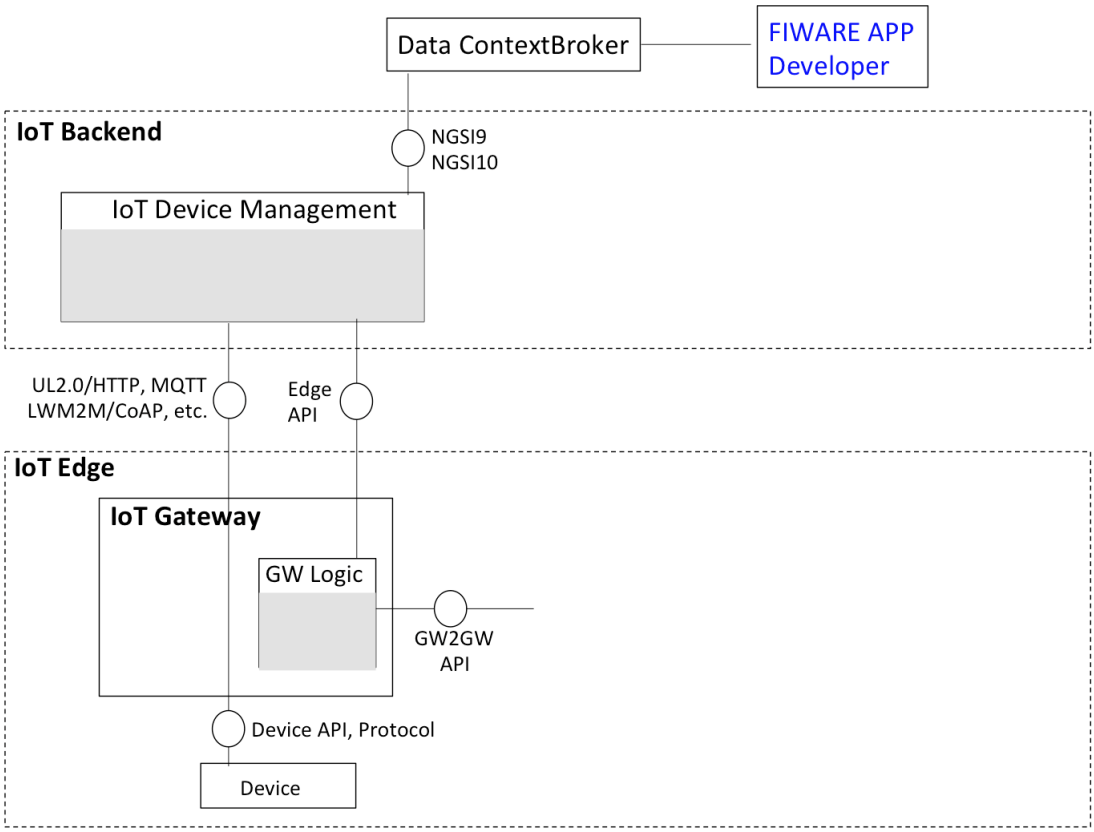


Figure 3. FIWARE's IoT Services Enablement Architecture [18].

2.4.2 Backend Device Management

The first mandatory Generic Enabler is composed of several agents that are able to receive data from devices and gateways that use common IoT protocols like MQTT or CoAP, and communicate this data to the Orion Context Broker using the NGSI API [29]. The implementations of this Generic Enabler are called IoT agents. This component is necessary for deployments

Table 2. Example FIWARE IoT Agents

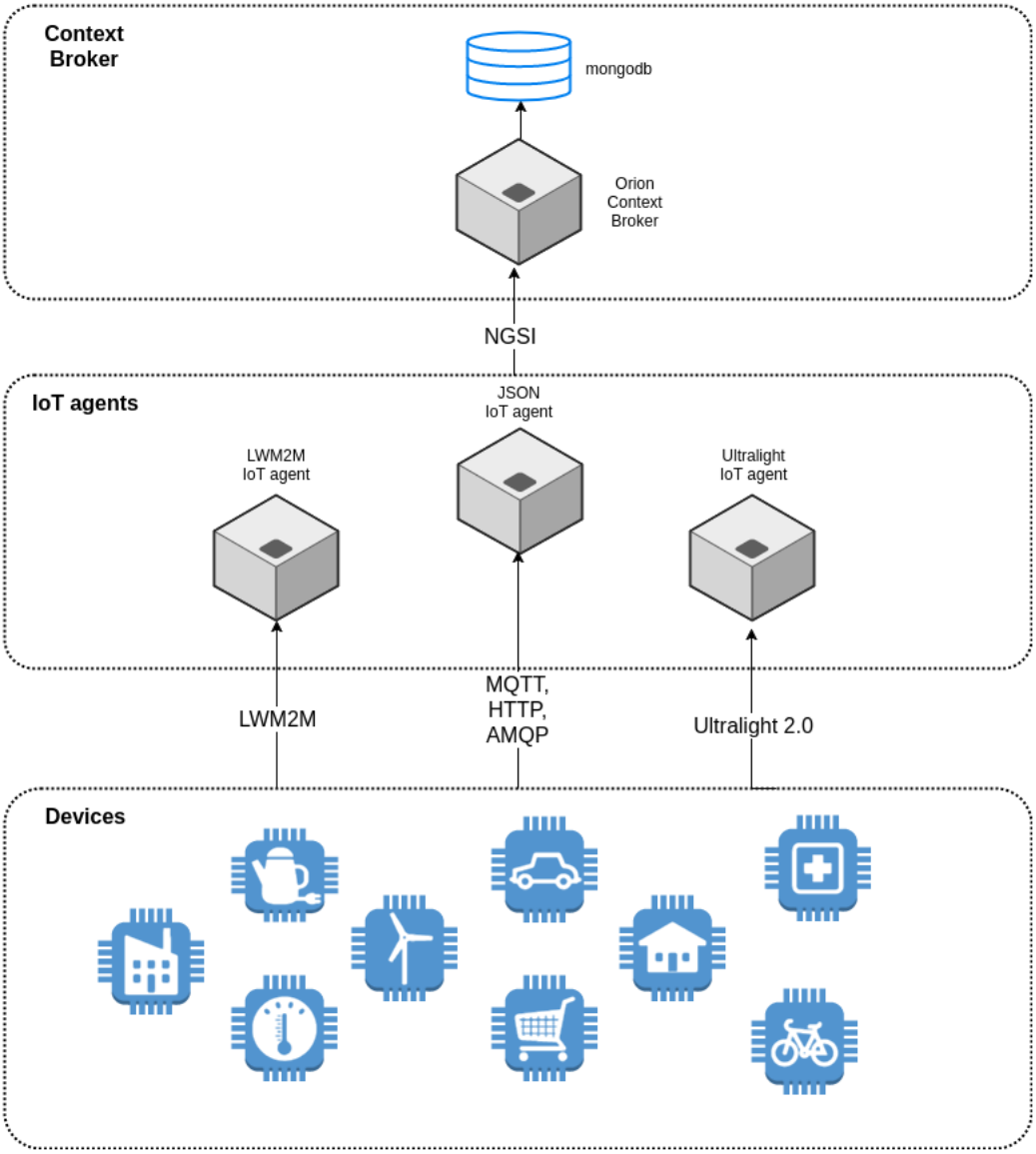
Agent	Semantics	Transport
OMA Lightweight M2M IoT Agent	LWM2M specification developed by the Open Mobile Alliance	CoAP
IoT Agent JSON	JSON-encoded data	AMQP, HTTP, MQTT
Ultralight 2.0 IoT agent	Ultralight protocol 2.0, a text-based protocol developed in FIWARE	AMQP, HTTP, MQTT

where the devices and applications are not able to communicate with the Context Broker using NGSI. This is often the case due to the constrained nature of devices in the field and due to the widespread adoption of protocols like MQTT or CoAP, which makes it inconvenient to have to support NGSI in the devices just to communicate with the context broker. The current generation of IoT agents in FIWARE is written in Node.js, and agents are classified by the top level (device semantics) protocol used [30].

2.4.3 Data Chapter ContextBroker

This Generic Enabler has a central role in FIWARE's model. It is treated as an interface for FIWARE App Developers for getting high-level context information. In IoT scenarios, it handles all the context entities that represent the IoT devices and their information.

The Orion Context Broker is an implementation of the Publish/Subscribe Context Broker Generic Enabler. It decouples the consumers of data, like end users and M2M applications, from the devices, objects and resources that produce the data. The Context Broker provides an API that implements the NGSI-9 and NGSI-10 standards. The API allows the registration of context producers (i.e. a temperature and humidity sensor), updates over existing contexts (updated sensor values, for instance) and queries and notifications that allow context consumers to access available information.



FIWARE defines IoT agents that mediate between diverse devices and the Context Broker. This allows devices that use common IoT protocols to connect to the NGSI-based FIWARE platform.

Figure 4. FIWARE IoT Services Enablement

2.5 Standards and Protocols in the Internet of Things

Several competing standards and protocols have been developed to enable the vision of the Internet of Things. In this section we describe prominent standardized protocols that are used in the general IoT environment and within FIWARE. The protocols described in this section are all supported in our testbed.

2.5.1 CoAP

The Constrained Application Protocol (CoAP) is a web transfer protocol specifically designed for use with constrained nodes and constrained networks. It was designed taking into account the requirements for low-powered devices with possibly small amounts of memory, in network environments that might be lossy and low-powered as well. The specification considers nodes with capabilities as low as 8-bit microcontrollers, and lossy networks with a typical throughput of tens of kbit/s like IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs). It was designed with the goal of supporting machine-to-machine (M2M) applications, running on top of existing infrastructures and interfacing with existing web standards and paradigms [31].

CoAP is an Internet Engineering Task Force (IETF) standard. The stable specification is defined in RFC 7252. CoAP aligns itself with the web paradigm by providing URIs to locate resources, Internet media types to describe them and a stateless mapping to HTTP verbs. CoAP has low overhead and is compatible with existing IP infrastructure with an UDP binding. Other bindings for SMS, TCP, TLS and Websockets exist as IETF drafts [32], with some of them being offered in implementations.

In CoAP, data and information are modeled as resources. Similarly to HTTP, resources are located and identified with URIs and can be accessed, created, modified or deleted with semantic methods. Resources are accessed using a request/response model as in HTTP. However, requests and responses are tipically sent using UDP instead of using a TCP connection like in HTTP. CoAP messages can be confirmable for reliability or non-confirmable for applications where best-effort delivery is sufficient.

CoAP requests act on resources, and are built by specifying the method to be applied to the resource, its identifier, a payload, an Internet media type and optional metadata about the request. After receiving and interpreting a request, the server sends a CoAP response with a Response Code that might indicate success, client error or server error. The response is matched to the request using the client-generated token in the request.

CoAP endpoints are able to actively send data when an observe relationship is established. This mechanism is an implementation of the observer design pattern. A client that is interested in receiving updates of a piece of data can issue a GET request to the relevant resource, setting the observe option to the value of 0. If possible, the server will add the client to the list of observers of the resource. By setting the observe option to 1, the client can request to be derregistered. When the resource changes, additional responses called notifications will be sent to the client with the updated data. Each notification includes the token that the client submitted in the

Table 3. Verbs in CoAP according to RFC 7252 [31]

Verb	Description
GET	Retrieves a representation of the resource identified by the URI.
POST	Requests that the representation enclosed in the request be processed. It usually results in a new resource being created or the target resource being updated.
PUT	Requests that the resource identified by the request URI be updated or created with the enclosed representation.
DELETE	Requests that the resource identified by the request URI be deleted.

request. A notification and a normal response are otherwise normal, with the only difference being the presence of the Observe Option.

2.5.2 OMA Lightweight M2M

Lightweight M2M (also known as LWM2M) is a device management protocol developed by the Open Mobile Alliance, an open standards body in the mobile industry whose members include manufacturers, mobile operators and software vendors. Members include Ericsson, Samsung, Philips, Motorola, Qualcomm, Vodafone, Orange, Microsoft, Sun, Oracle, IBM and others.

Lightweight M2M implements the interface between M2M devices and M2M servers. The motivation of LightweightM2M is to develop a fast deployable client-server specification to provide M2M services, defining device management and transmission of service and application data that meet the requirements of modern applications (including the Internet of Things) [33]. The requirements of the protocol were made with resource constrained devices in mind. Therefore, the protocol was designed to be light and compact and the resource data model tries to be efficient.

Lightweight M2M uses CoAP as its transport. The current specification is version 1.0, and it defines a client-server interaction model through the application layer communication protocol between a LWM2M Server and a LWM2M Client in a LWM2M Device.

LWM2M defines a resource model on top of CoAP, to model the capabilities, services and data

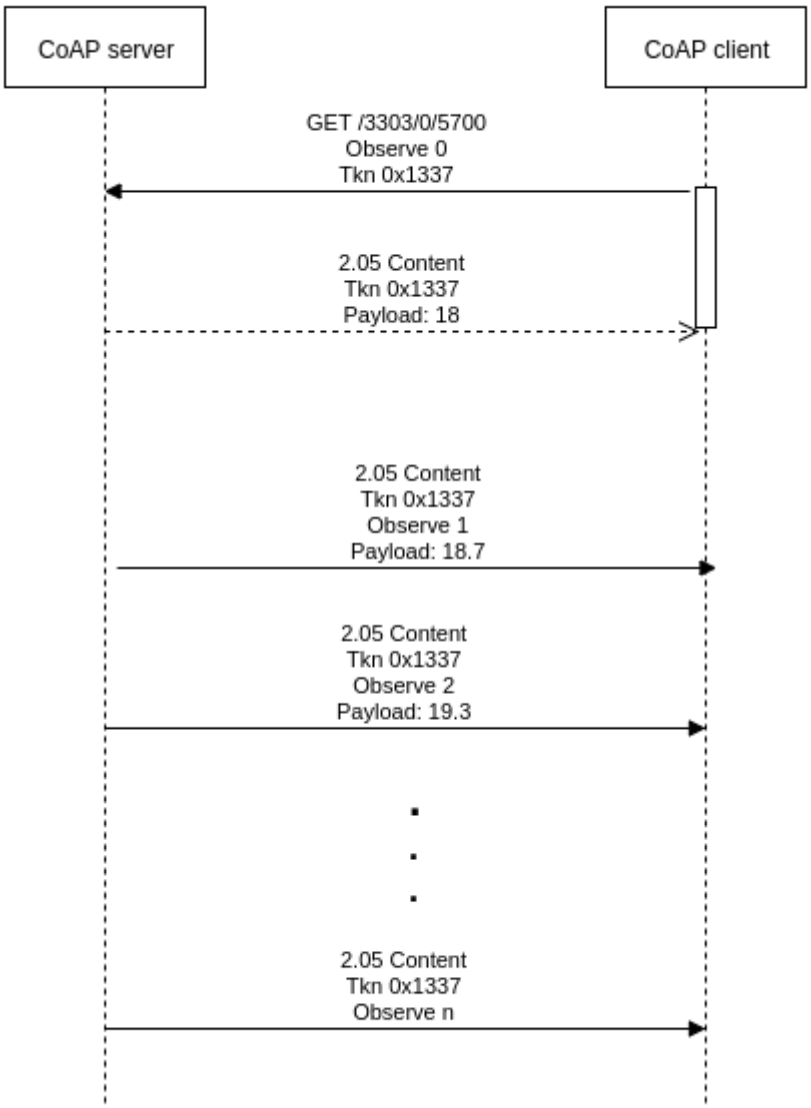


Figure 5. The observe mechanism, used by devices to actively send data updates.

in a device. At the top level, a LWM2M device can have objects that are characterized by a name, an object ID, the ability to be instantiated multiple times, being mandatory or optional and finally an object URN.

Resources exist within objects (or within object instances if these are supported), and they are defined similarly through the use of an ID, a name, the ability to support multiple instances and additionally by supported operations, units, data types, a range or enumeration that can limit the values and a resource description.

The Lightweight M2M version 1.0 specification defines 8 basic objects and their related re-

Table 4. Object Definition in LWM2M

Name	Object ID	Instances	Mandatory	Object URN
Name of the object	16-bit unsigned integer	Multiple/ Single	Mandatory/ Optional	Uniform Resource Name

Table 5. Resource Definition in LWM2M

Name	ID	Operations	Instances	Mandatory	Type	Range	Units	Description
Name	ID	R (Read), W (Write), E (Execute)	Multiple/ Single	Mandatory/ Optional	String, Integer, Float, Boolean, Opaque, Time, Objlnk	If any	If any	Description

sources. Each of these objects covers a specific aspect of device management like security, access control, device information, connectivity and location. The object and resource model specified in LWM2M is extensible, allowing organizations and developers to create their own objects and resources. An important standardization effort that is based on the LWM2M object model is the IPSO Smart Objects initiative. IPSO smart objects provide interoperability through a standard high-level object model based on LWM2M's object model specification. The model defines both generic and specialized devices. They are represented by a path URI that includes the object, object instance and resource ID separated by the "/" character.

Four interfaces exist between LWM2M client and servers [33]:

Table 6. Definition of the IPSO Humidity Sensor.

Name	Object ID	Instances	Object URN	Description
IPSO Humidity	3304	Multiple	urn:oma:lwm2m:ext:3304	Relative humidity sensor

Table 7. Example Definition of the Resources of an IPSO Humidity Sensor

Name	ID	Ops.	Single/ Mult.	Opt./ Mand.	Type	Description
Sensor value	5700	R	Single	M	Float	Last or current measured value
Units	5701	R	Single	O	String	Measurement units definition
Min measured value	5601	R	Single	O	Float	Mininimum value measured since power ON or reset
Max measured value	5602	R	Single	O	Float	Maximum value measured since power ON or reset
Reset min and max measured values	5605	E	Single	O	Opaque	Reset min and max to current value

1. Bootstrap interface, used to provision the initial configuration that a LWM2M device needs to register with one or more LWM2M servers.

2. Registration interface, used by a client to register to the server with an endpoint name and the objects and objects instances that it supports. Optional parameters include a registration lifetime, a binding mode and SMS number. This interface also allows a client to periodically update its registration to the server.

3. Device management and service enablement interfaces, which is used by the LWM2M server to execute "Create", "Read", "Write", "Delete", "Execute", "Write attributes" and "Discover" operations in the LWM2M device.

4. Information reporting interface, used by a LWM2M server to observe changes in resources of LWM2M clients. After establishing an observe relationship through an "Observe" operation, the LWM2M client sends "Notify" messages with updated values of an updated resource.

2.5.3 MQTT

MQTT is a Client Server publish/subscribe messaging transport protocol. It was originally developed by Andy Stanford-Clark (IBM) and Arlen Nipper (Arcom) in 1999. After being used internally in IBM, it was made publicly available in 2010. MQTT was designed to be lightweight and simple to implement in constrained devices, characterized by scarce processing power and network environments with limited bandwidth. Its characteristics make it a strong choice for use in constrained environments and communications in Machine to Machine (M2M) and Internet of Things (IoT) contexts [34].

MQTT tipically runs on top of TCP/IP, in contrast to CoAP's standard UDP binding. MQTT supports three QoS levels. QoS 0, or "at most once" has the lowest guarantees. A message can be delivered or not, according to the capabilities of the underlying network (a "best effort" service with no attempted retries). QoS 1, or "at least once", guarantees that the message arrives at the receiver at least once but with the possibility of duplicate message. QoS 2, or "exactly once", has the strongest guarantees for use cases where both message loss and message duplication are unacceptable.

In MQTT, data producers are able to publish messages to a topic, and data consumers can subscribe to receive messages published to this topic. These interactions are mediated and decoupled through a broker, so the publishers and subscribers only need to know about the broker and not about the other producers and consumers. Topics are defined using UTF-8 strings, that have a tree structure in which the levels are separated by a forward slash ('/').

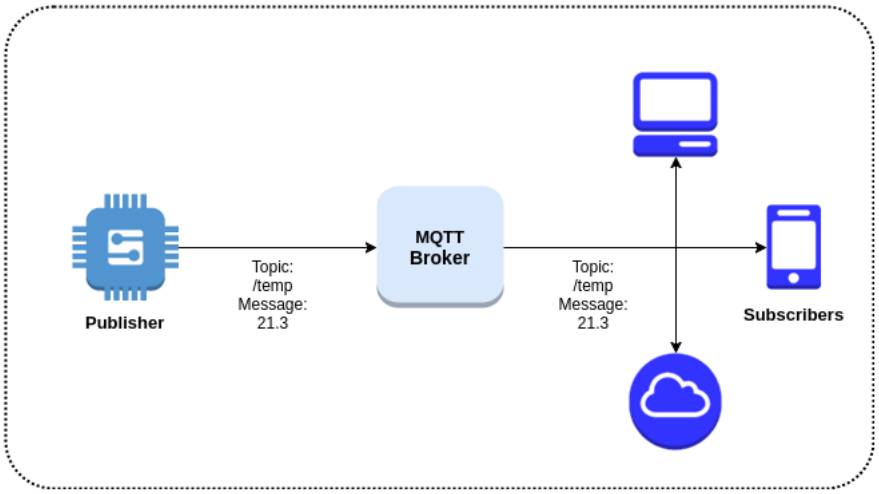


Figure 6. Decoupled Publish-Subscribe in MQTT through an MQTT broker.

For example, a light sensor in the living room of a house could be modeled using the topic "house/firstfloor/livingRoom/lightSensor". A temperature sensor in the same living room could be assigned to the "house/firstfloor/livingRoom/tempSensor" topic. An application interested in the data produced by these sensors can subscribe to those topics. MQTT defines wildcards to subscribe to multiple topics in levels of the topic tree. The multi-level wildcard *"#"* can be used to match any number of levels within a topic, while the wildcard *"+"* matches only one topic level. In our example, we could subscribe to the topic *"house/firstfloor/#"* to subscribe to all topics that include the *"house/firstfloor"* subtree. This would include topics like *"house/firstfloor/kitchen/tempSensor"* and also a topic like *"house/firstfloor/livingRoom/wall/sensor"*. To illustrate the usage of the single-level wildcard, we can consider a *"house/firstfloor/+/tempSensor"* that would match the *"house/firstfloor/kitchen/tempSensor"* topic but not a topic with additional levels like *"house/firstfloor/livingRoom/northwall/motionSensor"*.

MQTT defines control packets that consist of a fixed header, an optional variable header and a payload. In control packets, data can be represented using bits, 16-bit big-endian integer values, or UTF-8 encoded strings. The payloads are sent in byte format and it is up to the application to perform the encoding and decoding.

2.5.4 NGSI

NGSI stands for Next Generation Services Interface. It is a set of standards specified by the Open Mobile Alliance, focused on personal communication services and advanced composition of services. The scope includes mobile networks but is not limited to it. NGSI defines the following functionalities:

- Data Configuration and Management
- Call Control and Configuration
- Multimedia List Handling Extensions
- Context Management
- Registration and Discovery
- Identity Control functionalities

NGSI Context Management is used in FIWARE to define a context model to build smart applications. It defines the NGSI-9 Context Entity Discovery Interface and the NGSI-10 Context Information Interface.

The standard supports context aware computing [35], with a Context Information model that defines Context Entities for modeling elements at the highest-level. Context Entities can have several attributes, and these can in turn be described through metadata.

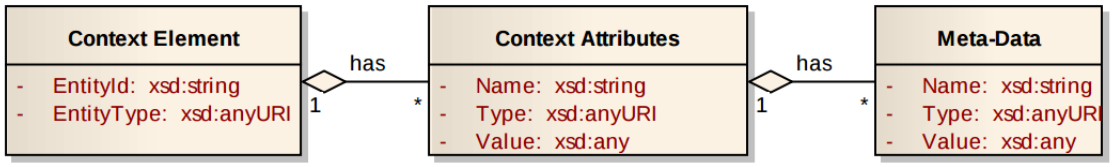


Figure 7. Context Information Model in NGSI.

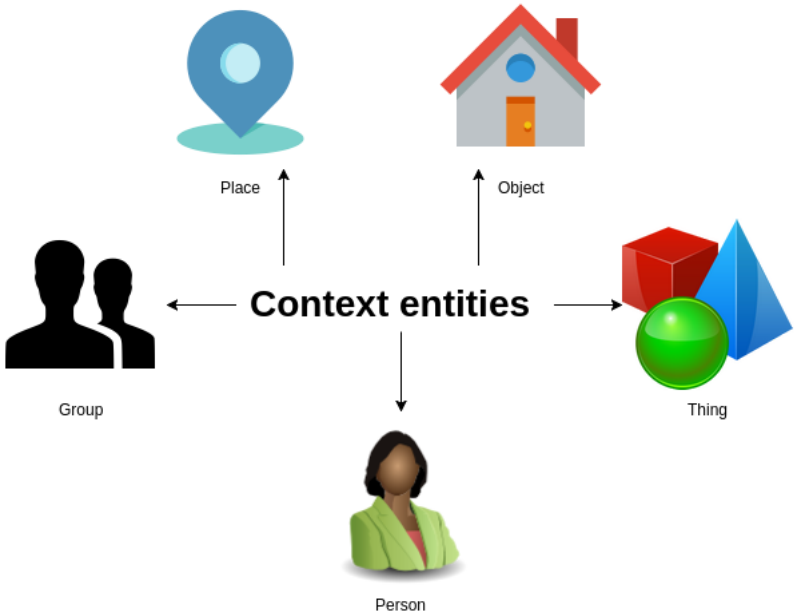


Figure 8. Examples of Context Entities.

The NGSI-9 Context Entity Discovery Interface defines three operations:

1. Register Context Entity, which is used to register and update Context Entities, their attributes and availability. It can supply metadata, a Context Entity ID and the context attributes.

2. Discover Context Entity, for discovery of Context Entities and their related attributes.

3. Subscribe and Notify based Context Entity Discovery, which allows applications to register to receive notifications about context changes or about the availability of new Context Entities.

NGSI-10 defines the Context Information Interface to handle context queries, updates and notifications. The three operations it defines are the following:

1. Update Context, which lets a Context producer provide or update Context Information in the Context Management component.
2. Query Operation, used by context consumers to query Context Information using Context Entitiy IDs or patterns of IDs and attributes.
3. Subscription and Notification Operation, used by applications to register themselves or other applications to receive notifications for context changes.

2.6 Cloud Computing

The National Institute of Standards and Technology of the United States of America (2010) provides the following widely cited definition of cloud computing[9]:

> Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.

The five essential characteristics of cloud computing are as follows:

1. On-demand self-service, meaning a consumer can get computing resources and services without the need of human interaction with the service provider.
2. Broad network access to computing resources, through thick or thin clients.
3. Resource pooling, meaning that consumers have access to virtual resources that lie in a common pool, following a multi-tenant model.

4. Rapid elasticity, characterized by the ability to acquire and release resources according to demand, often automatically.

5. Measured service, where resource usage can be monitored, controlled, and reported.

The service models define three broad categories that classify the offered services according to the distribution model of the offered resources.

1. Infrastructure as a Service (IAAS) is characterized by the offering of processing, storage, networks, and other fundamental infrastructure resources. The consumer cannot manage the underlying hardware but is able to deploy arbitrary software and services on top of the provisioned resources.

2. Platform as a Service (PAAS) enables the deployment of applications that run in a platform that offers a runtime of programming languages, libraries and tools supported by the provider. The consumer does not have control over the underlying infrastructure, only over the deployed application and limited configuration settings. This service can be useful for use cases where a managed runtime is needed to deploy applications, and specifiying and maintaining the underlying infrastructure is not desirable or essential.

3. Software as a Service (SAAS) provides consumers with applications running on cloud infrastructures, that are accessible through diverse clients. Consumers do not manage or control the underlying infrastructure or the application itself, they are just consumers of a service and are limited to the configuration options allowed by the provider. This model is useful for users that do not need or want to manage an application and only care about getting a service.

The deployment models describe the type of organization that deploys the cloud and the intended consumers.

1. Private clouds are provisioned by a single organization, and different units of that organization are the intended consumers.

2. Community clouds are for exclusive use by a specific community. The provider can be a set of members of the community or a third party.

3. Public clouds are open to the general public, and might be run by business, academic or government providers.

4. Hybrid clouds are a composition of two or more distinct cloud infrastructures (private, community, or public), that are bound together by technology that enables data and application portability.

The applications envisioned for the Internet of Things require computing, storage and capabilities for accessing raw data and information. The projected scale of the Internet of Things could imply a gigantic demand of computing resources. On the other hand, many of the devices that make part of the Internet of Things would be constrained in terms of processing power, memory, energy and network capabilities. The on-demand and virtually unlimited capabilities for storage and processing of the cloud (from the perspective of the user), have made it a strong candidate for deploying IoT components. Cloud computing provides a suitable environment for data access, computing, data analysis and new opportunities to support the Internet of Things. The cloud layer deals with resource provisioning for storage and processing the captured data [10].

Sensors, actuators and other devices can be viewed as part of a sensing layer. Between the cloud and the devices, there is often a connecting layer made up of gateways and embedded devices. These devices can collect data from many sensors, and often translate the wireless protocols used by constrained devices to protocols tipically found in Internet communications. Gateways can also perform some processing tasks in order to reduce latencies, or need to offload them to the cloud since gateways can also be mobile constrained devices [36].

Numerous cloud platforms provide scalable storage and processing capabilities. Some of the techniques used to deal with big data in the cloud include batch processing, distributed databases, time series databases and stream processing systems. Cloud-based middleware and integration platforms from a number of governmental initiatives and commercial ventures exist [10].

Research and development of cloud components, platforms and solutions continues in the context of the Internet of Things. The concept of fog computing is a possible future extension to be added between devices and the cloud, by performing certain storage and computation tasks in systems residing at the edge, closer to the devices. This could result in more efficient computing and reduced latencies for certain tasks [1]. Considering the possibility of having millions of interconnected things, the dynamic deployment, ideal resource provisioning and auto-scaling of computational resources are some of the main research challenges for the cloud layer in the Internet of Things [36].

2.7 Smart Cities and Regions

The transformations brought by digital technology have reached countless aspects of our societies, including politics and policy. Many discussions have centered over the usage of technology by government bodies. While some are critical and privacy concerns have been often raised, the potential to provide better services to citizens has also been highlighted [37].

A definition coined by Dameri (2013) summarizes the characteristics envisioned for smart cities in the following way:

> A smart city is a well defined geographical area, in which high technologies such as ICT, logistic, energy production, and so on, cooperate to create benefits for citizens in terms of well being, inclusion and participation, environmental quality, intelligent development; it is governed by a well defined pool of subjects, able to state the rules and policy for the city government and development [4]

The concept of the smart city has been developed over several years through the confluence of socioeconomic and technological viewpoints. Perspectives on the subject and the main focus of research and development vary across the different fields and the different initiatives in the subject, which can involve government bodies, policymakers, economists and researchers in ICT. While the concept has been celebrated enthusiastically by many, it has also been criticized as ambiguous, empty and discursive. The idea of using ICT to enable low-carbon development has often been adopted uncritically by academics and policymakers, and the discussions about smart city development might need a more critical eye to be refocused and to steer away from empty narratives or from discourses that sound like utopias but result in commodization, control, exclusion and injustice [38]

Within the ICT research for smart cities, the potential of using Internet of Things to achieve the smart city goals has been often enunciated. The applicability of the Internet of Things to further the vision of smart cities extends to many areas. Previous efforts have focused on diverse solutions including renewable energy generation, traffic management, public safety, communications infrastructure and infrastructure monitoring. Some cities, like Santander in Spain, have built large-scale testbeds and proofs of concept that deal with data collection and visualization. [39] [40].

To attain the goals of citizen well-being and economic efficiency, the Internet of Things will incorporate thousands of heterogeneous devices that are able to collect data or act on the surrounding environment. Some initiatives are also looking to offer some collected data as open

data. This seamless infrastructure of devices and the collected data can be used by institutions and by the public to develop smarter services, guide decision-making and open up new service ecosystems and economic opportunities.

In Northern Sweden, the Sense Smart Region project is currently building a smart city platform through the collaboration of government, academic and business organizations. The project is funded by the Vasterbotten region, the Skelleftea municipality and the European Regional Development fund. Lulea University of Technology is the academic partner, while the government participants are the local municipalities of Umea and Skelleftea. Companies working directly on the platform include Sokigo, North Kingdom and DataDuctus [7].

The platform of the Sense Smart Region will enable other companies to build smart applications. It will facilitate information access and understanding of information in everyday environments, by providing means for the consumption of public data from the regional authorities and tools to connect and authenticate sensors to the platform. Combining data from sensors and open data into useful information would ideally create smartness in regional products and services.

2.8 Computer Performance Analysis

Performance analysis is performed within experimental computer science to express notions about the capacity and efficiency of a computer system. It has elements of measurement, interpretation and communication of metrics that are captured while trying to minimize the perturbances to the system [19].

Lilja (2005) lists three fundamental techniques that are usable for performance analysis:

1. Analytical modeling is based on mathematical models of a system. It is useful to provide insight about the behavior of real systems, but it is prone to oversimplifications that don't capture the complex interactions that happen in hardware and the different abtraction levels in software.

2. Simulations use computer programs that model the most important features of the system that is being analyzed. It has the benefit of being easily modifiable, being often cheaper than performing measurements on real systems, and having the potential to be accurate given that the assumptions and models are realistic enough. However, reaching a sufficient level of realism for having accurate results can be overly complex and time consuming.

3. Measurements on real systems often provide the best results when a specific system needs

to be analyzed, since no simplifying assumptions need to be made given the right measurement tools. However, it can be costly and also time consuming. Special measurement techniques have to be used to minimize the impact on the performance of the studied system.

This work is based on the approach of performing measurements in a real system. Measurements in real systems can take place in production environments, evaluating the actual conditions that take place in an actual deployment. In certain systems this is not possible, or it might be that the purpose of the performance analysis is to evaluate untested or experimental features in a real system that hasn't been deployed in a production environment. For these cases, the peformance evaluation relies on the generation of workloads as input to the system, in order to observe their performance when handling these workloads. This approach has been extensively used for benchmarks [19], and is necessary for large-scale evaluations of IoT systems performance since large deployments are currently scarce. Load testing is a particular set of techniques that relies on workload generation to test the capacity of systems to handle given loads. It has been often used for web systems, and many tools exist for this purpose.

Performance analysis is concerned with the definition and measurement of performance metrics. Performance metrics express numeric rates, ratios and amounts that describe an aspect of the system's performance. Common performance metrics include throughputs, to describe operations or volumes per units of time; utilization as the ratio of used capacity versus total capacity; and latencies as a measure of time spent waiting [41].

Scalability is another dimension of performance that concerns performance analysis. Scalability is a desirable attribute of a network, system or process. A distinction can be made between structural scalability and load scalability. Structural scalability is the property which allows a system to expand without needing extensive modifications to its architecture. In contrast, load scalability allows a system to perform gracefully when the offered load is increased [11]. Load scalability is the focus of this work, since we are interested in observing how well the system can handle a large amount of load coming from the southbound interfaces.

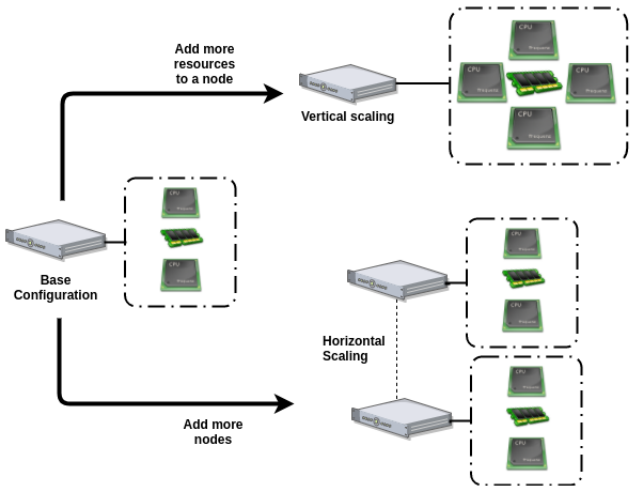
Bondi (2000) defines load scalability as follows [11]:

"We say that a system has load scalability if it has the ability to function gracefully, i.e., without undue delay and without unproductive resource consumption or resource contention at light, moderate, or heavy loads while making good use of available resources."

Load scalability can be negatively affected by factors like the scheduling of a shared resource

and inadequate exploitation of parallelism. Properties like computational time and space might be affected by implementation decisions. Thus, scalability gains can be achieved by fixing inefficient or imperfect implementations. When these changes are not possible or have already been performed, it is often possible to add more resources to a system to handle more load.

The two approaches to increase the load-handling capacity of contemporary systems are the vertical scalability (or scale-up) approach and the horizontal scalability (scale-out) approach. Scale-up solutions are characterized by large symmetric multiprocessing systems that share memory in the same machine. More load is handled by using a more powerful computer. In contrast, solutions that employ horizontal scalability are characterized by the usage of clusters of smaller machines. Each machine is a node with its own operating system. The vertical approach was dominant during the previous decades, while the adoption of horizontal scalability has been widespread for high-throughput web applications [42].



Scalability approaches can seek to upgrade the capacity of a machine or divide the work among several smaller machines. It is also possible to combine both approaches for high-end systems.

Figure 9. Scalability Strategies

Adding resources to improve scalability either through the vertical or horizontal approach comes with a cost. Gene Amdahl created a formula to predict the theoretical speedup when using multiple processors, known as Amdahl's law or Amdahl's argument. The speedup gained on n processors is limited by the fraction f of execution time that can be parallelized [43].

$$speedup_{parallel}(f, n) = \frac{1}{1 - f + \frac{f}{n}}$$

Gunther (1993) derived a scalability model that combines the level of parallelism N, the contention α for waiting or queueing for shared resources and the level of coherency β due to the effect of waiting for consistency [44]:

$$C(N) = \frac{N}{1 + \alpha(N-1) + \beta N(N-1)}$$

Different levels of contention and coherency exhibit four different behaviors. When contention and coherency are not significant, the system can scale linearly with load. After a certain point, contention causes the throughput to grow sublinearly as load is increased, until incoherency causes the throughput to decrease.

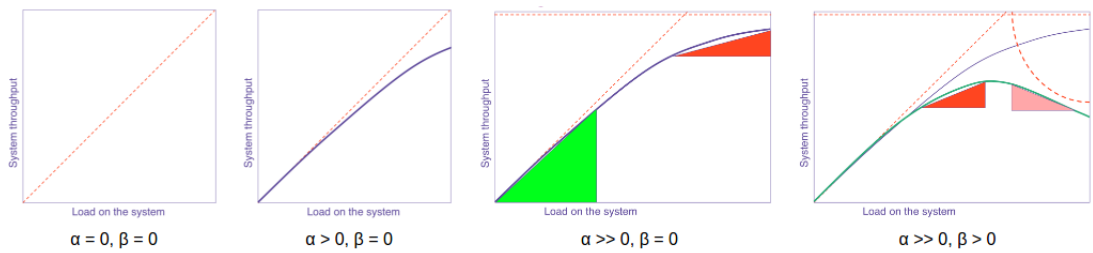


Figure 10. Gunther's scalability model and its four behaviors.

While these models don't take into account other complex factors that limit performance, like memory bandwidth or I/O latencies, they provide bounds on scalability gains and a general picture of behaviors observed when scaling the parallelism and load of a system.

2.9 Related Work

The projected scale of the Internet of Things has been identified as a challenge, and the evaluation of IoT solutions in real-world experimental deployments impacts the maturity and adoption of technologies. Key requirements for IoT testing include building a scale representative of projections, reflecting the heterogeneity of devices, protocols and solutions and achieving repeatibility of specifications, traces and results [45].

Early testbeds focused on wireless sensor networks and ranged from a few dozens to several hundred nodes. An example of a large-scale testbed with real devices is the SmartSantander smart city project, which set the objective to deploy up to 20,000 heterogeneous IoT devices

and connect them in an IoT infrastructure, in order to support research and experimentation [46].

Testing with real devices is cumbersome and expensive. Several initiatives have focused on emulating or simulating the IoT devices that connect to a system or platform. Some of these approaches focus on connectivity and network-level emulation, running on top of simulation frameworks like Contiki's Cooja or being deployable in academic testbeds like Emulab or Planetlab [47].

Solutions that focus on the application-level view are closest to the work in this thesis. Some commercial offerings are able to simulate devices or gateways connecting to an IoT platform, like in IBM's Bluemix and Eclipse Kapua's Kura simulation framework. These virtual devices can serve for demonstration purposes and functional testing, but it is not clear if they can easily scale to test scenarios in the Internet of Things. Google describes using containers and Kubernetes to scale load testing tools that can be used for websites and IoT scenarios [48]. An approach using containers was able to create a few hundred emulated LWM2M devices for use in a testbed at Ericsson [49]. However, a larger scale is needed to test certain scenarios. An earlier project had targeted 20 million CoAP nodes, but could only implement around ten thousand per machine due to resource constraints [50].

The performance of several FIWARE Generic Enablers has been analyzed within their organization as part of a quality assurance process. An independent evaluation of performance for usage in the precision agriculture domain was also carried out [51].

The performance studies done by FIWARE that are relevant for our scenario focus on the Context Broker. Performance studies for the IoT Backend Device Management IoT agents only covered the Ultralight 2.0 agent using the HTTP protocol. Internet of Things protocols like CoAP and MQTT were not tested. Furthermore, the generated requests to the Context Broker and to the only IoT agent tested are synchronous, while many load patterns coming from devices and also from the IoT agents are asynchronous due to protocols like CoAP or the choice of using Node.js by FIWARE for the IoT agents.

The tests for the Context Broker use Apache JMeter for generating requests to Orion's API. JMeter generates synchronous requests for every thread. The usage of synchronous or asynchronous requests can generate different behaviors in the System Under Test. Using synchronous requests is related to a closed queueing model, in which there is a fixed number of load generators in the system. These load generators can be either waiting to issue a new request following a think time or being serviced in the system:

$$N = N_{think} + N_{system}$$

The server load ρ, which denotes the fraction of time that the server is busy, is given by the mean throughput *X* and the mean service demand *E[S]*

$$\rho = XE[S]$$

A closed system can have a maximum of *N* requests in the system at any time, since the load generators block and can't issue a new request after the previous request has finished. The system is self-throttling and the mean arrival rate λ depends on the *N* users.

In open systems, there is a stream of requests arriving with an average rate λ. Workload generators often generate the average requests following a statistical distribution. The server load is given by:

$$\rho = \lambda E[S]$$

The usage of particular load testing tools assumes a particular model, and measured latencies can be different depending on the chosen model. Some researchers have found that this choice is often overlooked by practitioners and that in some applications a partly open model might be more realistic [52]. The behavior of an open model can be approximated with a closed model when the concurrency or multiprogramming level (the number of threads) is large. Moreover, concrete implementation details of the load testing tools can introduce inaccuracies that are outside the control of the testers. Issues related to the Java Virtual Machine and networking libraries have been singled out for some tools [53].

3 Testbed Design

This chapter describes the testbed that we have built to deploy and test our scenario. Based on the current research challenge, the objectives of our work, our review of the state-of-the-art and the particular context of the Sense Smart Region platform we begin by elucidating the objectives that we set to achieve with the testbed.

Next, we give a rundown of the components of our platform and how they fit in our scenario. Testing the performance of the FIWARE components that handle the data from devices in the Internet of Things required the development of new tools to generate load of IoT protocols, deploying and configuring the standard architectural components of the platform and installing a set of supporting applications for performance measurements and administration of the testbed. In this chapter we describe the challenges that we faced when building the testbed, and we list its detailed configurations and technical characteristics for repeatability of our results.

3.1 Objectives of the Testbed

We established a number of objectives to guide the development of the testbed. These objectives are are based on our stated goal to evaluate the capacity of the platform in our scenario, to handle the load of hundreds or thousands of sensors that will make part of future smart cities. Considering this goal within the context of the state-of-the-art of cloud computing, the Internet of Things and the architectural choices of the FIWARE platform we formulated the goals that we detail next.

1. Cloud-based deployment. While the cloud is less of a controllable environment in comparison with using fully-owned hardware, the cloud is the final target environment for many real-world deployments for the Internet of Things. It is widely used throughout the literature and in industry to deploy IoT components and we seek to build our testbed in an environment that is as close as possible to actual deployments.

2. Performance observability. Performance observability is the ability to accurately capture, analyze, and present (collectively observe) information about the performance of a computer system [54]. Several components interact in IoT scenarios and it is necessary to balance the need for performance data against the cost of obtaining it. We desire to collect resource usage and application-level metrics to be able to identify behaviors and bottlenecks, while minimizing the perturbances to the system.

3. Massive load generation. An important goal of our testbed is to enable testing of southbound data being pushed into the platform without the need to provision and configure thousands of physical sensors. As the Internet of Things matures and increasingly connects more components in more layers, new challenges arise for testing scenarios. A major goal of IoT research is to connect millions of devices in a global infrastructure, which is a sizeable jump in terms of scale and heterogeneity for testbeds [45]. Some approaches for emulation of IoT devices for use in testbeds have been able to generate a few hundred emulated devices. Here we aim to go beyond and be able to simulate tens of thousands of devices.

4. Adherence to standards. Our testbed should implement a standard architecture within the specifications of FIWARE and also strive to reflect the broad trends in the state of the art of popular architectures and standardization efforts.

3.2 Parameter Identification

In this section we make a list of system, load, and environment characteristics that can have an effect on the performance of the studied system. Therefore they should be considered during the design of the tests and the testbed.

1. Average requests arrival per second: The average requests arrival is a measure of the load that is coming into a component of the system, by expressing the amount of incoming requests per unit of time (tipically per second). The components of the system need to be able to support sustained loads during their operation. Load testing increases this average load in controlled ways to study the response of the system and determine if it is fit to support scenarios with the studied load characteristics. Increasing the average requests arrival can cause an increase in the computational resources needed to process them, cause queuing, increase response times and even overload the system to the point where it will crash and stop functioning.

2. Arrival distribution: While the average requests arrival describes the amount of requests per unit of time, it doesn't specify the way in which the requests are tipically distributed within that period. The same amount of incoming requests could, for instance, be equally spaced within the time period or occur mostly around a specific offset within each period. These distributions are often described using statistical distributions, including the exponential distribution of Poisson processes, the normal distribution, uniform distributions, the Gamma distribution and others.

3. Transmission unit size: Network transmission times, processing overheads and disk I/O can be affected by the size of transmission units.

4. Transport protocols: Transport protocols have different overheads in terms of processing, memory and bandwidth. Protocols geared for the Internet of Things seek to reduce these overheads.

5. Instance Type: Instances in the cloud have different performance characteristics. Amazon Web Services describes their instances by the number of virtual CPUs, the amount of RAM memory and network capacity classified as "low", "moderate", "high" and "10 Gbps". Amazon provides tiers grouping instances into general purpose, compute-optimized, memory-optimized and I/O optimized instances. The increased or decreased computational resources of an instance in relation to another can affect the performance of applications.

6. Operating system: Operating systems have different performance characteristics across versions within the same distribution, across distributions with the same kernel, and across different kernels or brands.

7. Operating system limits: Operating systems might impose limits on the number of file descriptors, CPU time, virtual memory and number of threads usable by an application. These settings should be controlled to avoid hindering the execution of applications, especially for high-performance and high load use cases.

8. Application version: Software releases often implement new features, bug fixes and performance improvements after existing issues have been identified and fixed. Similarly to operating systems, the most recent versions of applications are recommended for security, performance, stability and extended features.

9. Application parameters: Many applications provide configuration parameters to tune their runtime characteristics for specific use cases. Documented and undocumented parameters can have large effects on application performance.

3.3 Parameter Selection

After the identification of the parameters for our tests, we now establish the choices to control the parameters. We make our selections aiming to choose representative parameters, that allow us to observe performance changes that can be of interest when implementing a deployment for smart city scenarios.

1. Average requests arrival per second: We use average request arrivals that are multiples of 100. After starting the system, it runs for a period of 120 seconds with no load. From our observations, the system is clearly in a stable state after just a few seconds. We proceed to gradually increase the average requests arrival by 100, and after reaching a steady state each level runs for 120 seconds. We continue this procedure until reaching 1500 average requests per second. This upper limit is chosen based on our initial observations of the capabilities of the platform's components.

2. Arrival distribution: We choose an exponential distribution to generate our average requests arrival. This distribution has been widely used to model network arrivals and also to model scenarios in the Internet of Things [55], [56]. This distribution is the default for load generation tools like Tsung and it is also supported by Apache JMeter. More complex distributions with different burstiness characteristics might be used, but their properties need to be known first (and very large deployments don't exist yet to determine them). Moreover, the technical challenges to generate load with these characteristics have to be solved and incorporated into the testing tools. Exponential traffic has the best fit when considering its availability in current tools and its usage in previous research.

3. Transmission unit size: Few datasets of IoT applications are publicly available due to commercial and privacy concerns [57], and a broad characterization of applications in smart cities is lacking. Furthermore, the payload sizes can change across application domains and across implementations within the same domain. For choosing our message sizes, we consider the sizes in the IPSO standard, which is often used along with LightweightM2M [58]. We chose on the low end a payload of 10 bytes, which covers the boolean, integer and float data types and can also cover strings. On the higher end, we consider a payload size of 1000 bytes, which covers the largest value in the evaluation of real world datasets performed in [59], and the average value of updates reported in [46]. Both of these values are small compared to web applications, but they reflect the constrained nature of IoT devices, the interest in low-overhead protocols and economic transmissions, and the sizes reported in the literature.

4. Transport protocols: Within the available transport procotols for the IoT agents in Fiware, we choose MQTT and CoAP based on their popularity in research and in real-world applications, their maturity as standards, their support from technical bodies and the adoption they have within FIWARE.

5. Instance Type: Based on the system requirements stated in Orion's documentation, which recommnend a baseline system with two cores and 4 GB of RAM, we choose an m4.large AWS instance with 2 cores and 8 GB of RAM. Instances of type m4 are the latest generation of General Purpose Instances in Amazon Web Services. This family provides a

balance of compute, memory, and network resources, making it a good general purpose choice for applications. For this reason we also choose it for the IoT agents, which have no recommended system requirements in their documentation

6. Operating system: From Orion's documentation, we choose Red Hat Enterprise Linux 6.5 for the context broker. We choose the latest version of Ubuntu, Ubuntu 16 Xenial Xerus, for the rest of components, due to its wide availability of packets and ease of deployment.

7. Operating system limits: We have performed tuning of the per-user and root soft and hard limits in /etc/security/limits.conf, /etc/pam.d/common-session*, systemd and System V initialization.

8. Application version For the database we have installed the recommended version of mongodb, mongodb 3.2. This version has extensive modifications to the storage engine and several performance improvements compared to previous versions, specially for scenarios with frequent updates. It is one of the recommended versions in Orion's documentation.

9. Application parameters: Following Orion's documentation, we have enabled the requests pool in the Context Broker, setting it equal to the number of available cores in the system.

3.4 Design of the Testbed

In this section we describe the cloud-based testbed that we have created to deploy the necessary FIWARE components for our scenario and to evaluate their performance.

3.4.1 FIWARE Components

The components that we deployed for our scenario, simulating thousands of devices pushing data to the platform come from the Common Simple Scenario in FIWARE's Internet of Things (IoT) Services Enablement Architecture Reference 5.

This architecture defines generic enablers that connect Things to Fiware, making them observable, representable and searchable as high-level entities according to FIWARE's context model. The reference architecture gives the following definition for an IoT backend:

> This domain comprises the set of functions, logical resources and services that are normally hosted in a Cloud datacenter. Its northbound interface is connected to

Table 8. IoT Agents used in the testbed

Repository name	Semantics	Transport protocol to test
lightweightm2m-iotagent	LWM2M	CoAP
iotagent-json	JSON-encoded data	MQTT

the data chapter ContextBroker, where IoT entites are mapped to NGSI "Context Entities". Its southbound interface is connected to the IoT edge.

Two mandatory generic enablers are defined, to integrate IoT devices into the Data chapter ContextBroker so that FIWARE App developers can interact with them. We describe in the following subsections the mandatory generic enablers that we added to our deployment. Configurations for these applications are detailed in the Configuration section of this chapter.

3.4.2 Backend Device Management

This Generic Enabler interacts with the devices or the IoT edge, translating the protocols used by devices and gateways into NGSI. It is in charge of handling sensor notifications from the devices towards the Context Broker.

Based on our Transport Protocols and Application Version guidelines for testing, we deployed the most up to date versions of the IoT agents for MQTT and LWM2M, from the official repositories of telefonicaid in Github.

3.4.3 Data Chapter ContextBroker

The Context Broker handles all the context entities that represent the information generated by the IoT devices. It is the natural interface for FIWARE App Developers for reading IoT sensing information (Entity attributes in FIWARE's context model).

The context broker used is version 1.7.0. It was downloaded from FIWARE Forge, the cen-

Table 9. Popular Open Source Load Testing Tools

Tool	Lang	Distributed mode	HTTP load	MQTT load	CoAP / LWM2M load	Extensibility
Tsung	Erlang	Yes	Native	Native	No	+
Apache JMeter	Java	Yes	Native	Plugin [60]	No	+++
Gatling	Scala	No	Native	No	No	+
Locust	Python	Yes	Native	No	No	++
httperf	C	No	Native	No	No	++

tral file repository of FIWARE. It was installed using Red Hat Packet Manager on a Red Hat Enterprise Linux 6.5 operating system following the recommendations from the official documentation.

3.4.4 Load Generation

The ability to generate load from IoT device protocols is essential to test our scenario. Load testing is based on the use of specialized software that is able to generate workloads and collect some performance metrics. A number of commercial and open source tools are available. Commercial licenses can cost hundreds of thousands of dollars for simulating a few hundred users. We make a comparison between the capabilities of several open source tools and their applicability for our scenario.

Most load testing tools have a strong focus on HTTP since they were originally created for that purpose. Support for MQTT has been added either natively or through plugins in the most mature tools. Ease of extensibility is a factor to allow us to test the IoT protocols of our scenario. Tsung [61] and JMeter [62] are the most mature tools in our comparison, having been around since the beginning of the 2000s. JMeter is more extensible for our purposes, and along with its usage in previous FIWARE testing makes it our choice for load generation.

For MQTT an existing tool integrated with JMeter existed, however this was not the case for LWM2M. Therefore, we developed such a tool and we describe our approach in this section. We have published it as open source, hoping that it can help further testing efforts. In this section

we begin by describing the load testing tool JMeter. Next, we describe the way in which we generate MQTT load. Finally we detail our approach for generating LWM2M load.

3.4.5 Apache JMeter

JMeter is an open-source tool that is used to test functional behavior and measure some aspects of the performance of systems. JMeter is a load testing tool that is used to model the behavior of entities or users that interact with a given system, to verify that the system behaves as expected in terms of functionality and performance when subject to load.

The JMeter engine is able to generate load using diverse protocols and complex logic that is customizable by the user through the definition of a test plan. A test plan describes the series of steps that JMeter will execute during the test. JMeter provides a set of abstractions called test elements to describe these steps. Test elements describe the protocols, the types of requests to use, their configuration and the logic that guides the execution of the test.

One of the highest-level test elements of JMeter is the thread group. JMeter's architecture is multithreaded, and in a typical scenario a thread will be used to model a user by performing the actions defined in the test plan independently of other threads. Logical controllers and samplers, the elements that send requests, are added to thread groups. Thread groups can be configured with the number of total threads to spawn, the amount of time over which these threads will be spawned, and the number of times to loop over the actions defined in the thread group. When doing functional and non-functional testing, the number of threads and the time it takes to spawn them has to be regulated to match the desired load.

Logical controllers may be added inside a thread group to customize the conditions that are used by JMeter to decide when to execute a sampler or other component. Logical controllers include rules that might execute other components a single time, randomly, a fixed amount of times or when necessary to achieve a particular throughput. This allows the user of JMeter to define complex scenarios.

Samplers are the test elements that send requests to the system that is being tested. JMeter includes samplers for several protocols, including FTP, HTTP, JDBC, TCP, LDAP, JMS and others. It is possible to extend JMeter and define samplers for other protocols that are not part of the upstream distribution. Samplers are configurable according to the protocol, and their behavior can be adjusted by other components.

Timers can be added as children of samplers and controllers to pause, synchronize or adjust their behavior. Thus, timers are used to shape the distribution of requests and their rate of generation. Timers range from simple constant time pauses to behaviors that follow a particular statistical distribution, including Gaussian, Poisson and uniform distributions. Another important use of timers is to synchronize the threads to execute a sampler or a controller at the same time, effectively creating concurrent requests to the system under test.

To parameterize the test scripts, JMeter provides configuration elements and variables. Configuration elements define a set of values that apply to specific test elements in a part of the test plan. For instance, a configuration element can be added that defines the host to target for all HTTP samplers. Variables can also be defined by the user and be accessed in every thread. JMeter exposes a set of predefined variables that are accessible during the test to configure parameters and to provide introspection about the current state of the test.

3.4.6 Generating MQTT Load

To generate MQTT load it is necessary to publish messages to a topic. Subscribers to a topic will receive the generated messages. Devices that are deployed according to FIWARE's IoT Service Enablement Architecture act as publishers when pushing data, with the IoT agent as the subscriber. The structure of the topic begins with the API key of the device, which has to be provisioned in the IoT agent, and is followed by the ID of the device. Next comes the word "attrs" and finally the name of the attribute that is being updated, according to the provisioned configuration. For example, the topic for a temperature attribute named "temp" in a fictional device with ID "sensor1" and API key "abc" would be "/abc/sensor1/attrs/temp".

We configure the JMeter sampler to publish messages to the topics to which the IoT agent is subscribed to receive information for each device. The sampler is configured to use the asynchronous mode of the underlying Paho client.

3.4.7 Generating LWM2M load

Previous approaches for testing LWM2M servers include emulated LWM2M devices [49]. The deployment of such clients using containers is an approach that has been described in technical articles by companies and can also be seen in open source projects.

Scaling these clients beyond hundreds is challenging. Despite the efficiency of existing libraries

in terms of memory, instantiating hundreds of these clients in a host can already consume several GB of memory. Deploying hundreds of a readily available client like Leshan using containers or instantiating them per thread in a tool like JMeter implies a considerable usage of memory, just from the clients themselves. For Java-based clients, deploying them in containers can entail large overheads due to the separate Java Virtual Machines that have to run in each container. After analyzing the memory usage of such clients, we decided to write custom mock LWM2M clients to be able to scale to thousands of clients on each host.

3.4.8 Library for Mock Clients

Mock objects are implementations that are used in computing to mimic the behavior of real objects in controlled ways. To be as economical with computing resources, we implement the basic LWM2M interactions on top of Californium, an efficient implementation of the CoAP protocol [63]. We avoid the overhead of complex objects by manually constructing requests, and we give the user of the library the option to manually specify the LWM2M objects, instances and resources. The interactions with the CoAP resources are handled directly by Californium.

3.4.9 Plugin for JMeter

A way to generate load for LWM2M had to be devised since it is not suppported in testing tools. The library described in the previous subsection was used to create a JMeter plugin, which provides configuration elements and samplers that allow virtual clients to register to LWM2M servers, and to notify observers when the value of a resource of interest changes.

The configuration element of the plugin allows the user to set the hostname and port of the server that will receive the LWM2M load. The user can configure registration options and behaviors. The registration path can be set, for servers that don't use the common *rd* path and the registration duration can be customized. The option to make the virtual client automatically register or derregister is also given to the user, to have more control over the scenario. The resources to add to the mock devices are added as comma-separated text, following the *objectId/instanceId/resourceId* convention in LWM2M. The plugin will take care of constructing the objects, instances and resources as Californium CoAP objects, allowing them to interact through the CoAP verbs and observe mechanisms.

The ability to have arbitrary resources allows the user to build any scenario by specifying the virtual LWM2M objects and resources that will generate load. JMeter acts as the driver, triggering

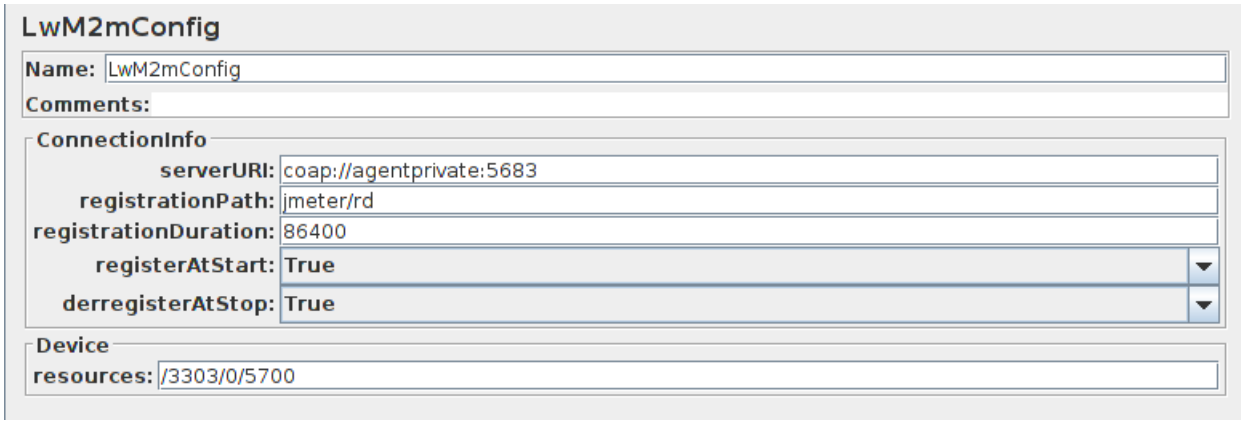
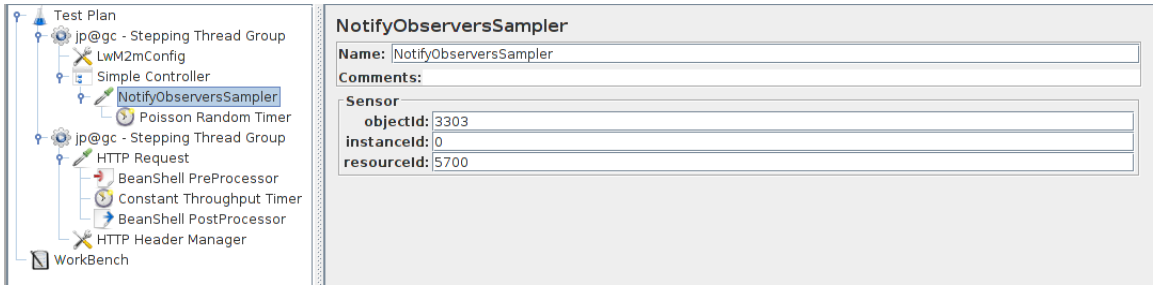


Figure 11. Configuration of our plugin in JMeter's UI.

requests from the mock LWM2M devices in the plugin according to the test script.



The sampler will send observations of a */3303/0/5700* resource (temperature sensor). Using standard JMeter elements the load can be scheduled to have specific throughputs or follow a statistical distribution with a timer.

Figure 12. Example configuration of a sampler

We were able to create up to 2000 clients per host in machines with modest capabilities, namely 8 GB of RAM. We have published the plugin as an open source contribution, hoping that it can be useful to practitioners. The source code is available on Github [64].

3.4.10 Sending Observations to the Platform

To make the devices actively push data to the platform, the LWM2M server of the platform has to send an Observe request to the resources of interest. LWM2M devices cannot begin sending data before getting an Observe request. LWM2M servers often provide a REST API or static configuration elements to observe resources in registered clients. We configure the LWM2M server in Fiware's IoT agent to observe the virtual resource of a generic sensor after registration.

3.4.11 Collection of Metrics

To collect and monitor resource usage metrics, we use the open source Prometheus project [65]. Prometheus is a monitoring system and time-series database, part of the Cloud Native Computing Foundation projects. It uses software agents called *exporters* to collect metrics of interest from applications or the operating system. The official Node exporter exposes hardware and operating system metrics. In Linux, it is capable of collecting a wide set of metrics coming from kernel counters, procfs and other performance observability tools. The metrics covered by the Node exporter cover CPU, memory and network usage among many others, and the maturity and stability of the project make it a great choice for collecting resource metrics.

3.4.12 User Interface

Performance analysis of the platform involves deploying components that can exceed ten instances per test. These components can be launched into several different instance types and arranged into diverse configurations. Handling this manually would be time consuming and error-prone. Therefore, a Python controller was developed that takes as input the instance types, the configuration of the components that will be tested and the JMeter test script to use. The controller uses the boto3 library to manage the lifecycle of the instances in Amazon Web Services. The controller provisions all the instances automatically and performs any administrative tasks necessary to connect them. It also creates a JMeter cluster for generating the southbound load using the script entered as input. When the script finishes running, the controller collects the results from JMeter and Prometheus for further analysis.

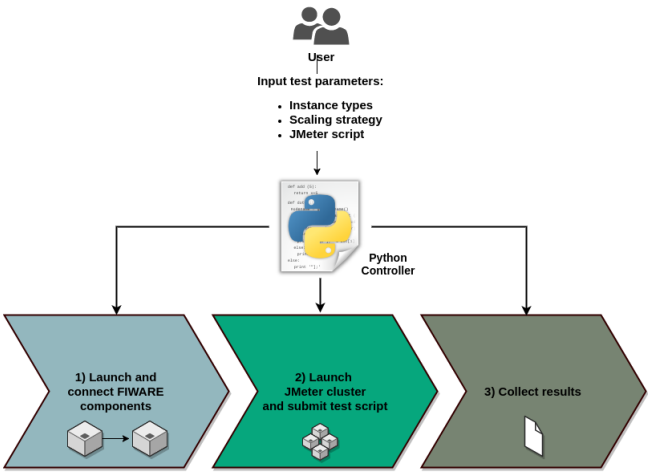


Figure 13. Automated operation of the testbed through a Python controller.

Table 10. Instance Types Used in Amazon Web Services

Instance type	Type	vCPU	Processor	Clock speed (GHz)	Memory (GB)
m4.large	General purpose	2	Intel Xeon E5-2676 v3	2.5	8
c4.xlarge	Compute optimized	4	Intel Xeon E5-2666 v3	2.9	7.5
c4.2xlarge	Compute optimized	8	Intel Xeon E5-2666 v3	2.9	15
i3.large	IO optimized	2	E5-2686 v4	2.9	15.25
i3.xlarge	IO optimized	4	E5-2686 v4	2.9	15.25

3.5 Configuration

3.5.1 Instance Types

We use the Elastic Compute Cloud (EC2) of Amazon Web Services to obtain and configure computing resources. We created virtual machine images and performed all the necessary configuration in the OS and applications for a high load scenario. All the virtual machines we use are hosted in the eu-central-1 region, located in Frankfurt, Germany. Furthermore, they are deployed in the eu-central-1b availability zone. Virtual machines in the same availability zone are located in the same datacentre.

3.5.2 Operating System Configurations

Our testbed supports two operating systems. Red Hat Enterprise Linux 6.5 is used for Orion Context Broker since the packages from FIWARE support this particular version. Ubuntu 16, the latest Ubuntu version with long-term support, is used in all other virtual machines.

Table 11. Operating System Configurations

Attribute	Value
file size	unlimited
max memory size	unlimited
open files	64000
max user processes	64000
cpu time	unlimited

To test the performance of FIWARE components (and other application software in general) under high-load scenarios, it is necessary to tune certain parameters in the operating system that place limitations on the amount of resources that can be used by a process. If this is not performed, results would be inaccurate and any performance measurements would only reflect the limits enforced by the operating system, instead of the capabilities of the applications. We observed that increasing the limit of file descriptors is of particular importance for the IoT agent due to the high number of sockets that it can use when the system is overloaded. These configurations are described in table 11.

3.5.3 Application Versions and Configurations

We list next the versions of all the applications in use in our testbed (tables 12 and 13). As discussed previously, we strived to support the application versions that come from documentation recommendations in FIWARE components and the latest available application versions if there is no particular recommendation.

3.6 Operation of the Testbed

To test the different configurations of our scenario, we follow a set of steps that we describe in this section. The testing process allowed us to observe the performance of the components of FIWARE's IoT Services Enablement Architecture in a cloud deployment, under different configurations of instance types and different load characteristics. The process involves a set of general steps which begin with provisioning fresh virtual machines for each component, to avoid undesirable effects from previous runs. A JMeter cluster is then started and the JMeter

Table 12. FIWARE Component Configuration

Role	**OS**	**Application version**	**Application parameters**
lightweightm2m-iotagent (LWM2M)	Ubuntu 16	Github 9b0cb99	
iotagent-json (MQTT)	Ubuntu 16	Github 06278f9	*ncpu*
Orion Context Broker	Red Hat Enterprise Linux 6.5	1.7.0	-reqMutexPolicy none, -reqPool *ncpu*

Table 13. Supporting Application Configuration

Role	**OS**	**Application version**	**Application parameters**
mongodb	Ubuntu 16	3.2	
Mosquitto MQTT broker	Ubuntu 16	1.4.8	
Prometheus	Ubuntu 16	0.16.2	
node exporter	Ubuntu 16, RHEL 6.5	0.14.0-rc.2	
Apache JMeter	Ubuntu 16	3.1	-Xms2048m, -Xmx2048m
Leshan	Ubuntu 16	0.1.11	-Xms4096m, -Xmx4096m
etcd	Ubuntu 16	2.2.5	
Prometheus	Ubuntu 16	1.5.2	

script that implements the test logic is run on it. When the run finishes, results from JMeter and Prometheus are collected for analysis.

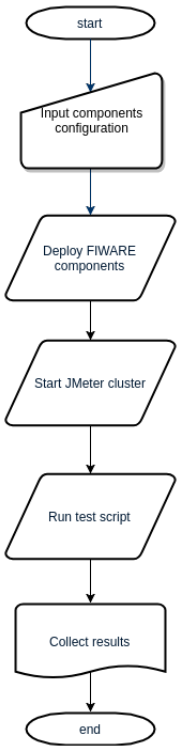


Figure 14. Flowchart of the steps taken to run each test.

3.6.1 Deploying the System Under Test

The System Under Test consists of a deployment of the Common Simple Scenario of FIWARE's Internet of Things Services Enablement Architecture described earlier. This deployment includes one or more IoT agents that receive data from the southbound interfaces, and the Orion Context Broker with its mongodb database. Furthermore, these components can be deployed in cloud instances with different specifications to scale vertically, or deployed with multiple instances of the same component behind a load balancer or in a sharded cluster for the database.

Additional supporting components are needed during each run of a test, including an MQTT broker if the MQTT IoT Agent is used, an etcd server for configuration management, a Prometheus server for monitoring, and the JMeter cluster for generating the load from the southbound interface.

During the development of the testbed, several virtual machine images were created for every FIWARE component and every application necessary for testing. These virtual machines have the application versions and application configurations that support the parameters that we identified for their effect on the system's performance. The application versions and particular configurations that were added to support high levels of load are described in the configuration

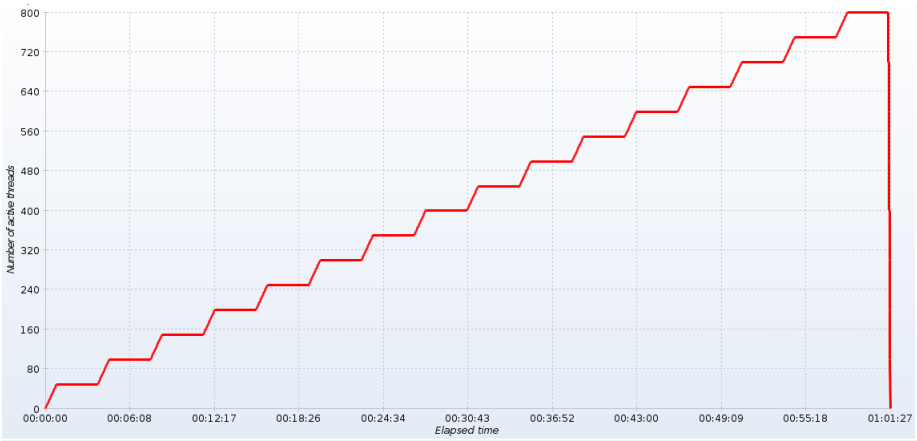


Figure 15. The average load on the system is increased by a predefined step size. Performance measurements are taken for each level of load.

section of this chapter, for repeatability of our results and to provide an account of the details that have to be considered when deploying such a system in similar setups. These finalized snapshots reside in the Amazon Machine Images service.

When a test needs to be run, it is submitted to a Python controller that launches an instance of the specified size using the corresponding image for each component. The controller also creates and operates the JMeter cluster and collects the results for future analysis.

3.6.2 Specifying Load from the Southbound Interface

JMeter scripts were created to specify the type, content and rate of data updates coming from the mock devices that simulate thousands of sensors sending data to the platform. The API of the IoT agents is used to generate the subscription to the resources in the mock devices, which internally handles establishing the observations for LWM2M/CoAP or the topic subscriptions for MQTT. These scripts use the standard JMeter functionalities and the custom plugins described in the previous Design of the Testbed section in this chapter. To control the average requests per second generated and the exponential interarrivals the Poisson random timer of JMeter is used. A stepping thread group is used to progressively add more threads and more load, increasing the average data updates per second by one hundred at each step. For each level of load, 30 measurements of the performance metrics of interest are taken after the steady state has been reached, and then summarized by computing their mean and 95% confidence interval.

3.7 Test Plan

In this section we give a rundown of the tests that were performed, describing the goals that we wanted to achieve, the parameters of the test and the performance metrics to evaluate.

3.7.1 Performance Metrics

The performance metrics were chosen to quantify and compare the performance of the platform in different deployment configurations. We define them next and describe the procedure we used to calculate them.

3.7.1.1 Throughput to the Database

We define this metric as the average rate of bytes per second that reaches a mongodb instance, as reported by Prometheus's node exporter through the instantaneous rate of the counter named *node_network_receive_bytes*.

It is fetched from Prometheus using the following expresssion:

```
irate(node_network_receive_bytes[5m])
```

We chose to measure the throughput to the database instance as a metric for the effectiveness of delivering data through all the components of the platform and finally reaching the database. This metric should ideally scale linearly, since when the rate of data coming from devices is doubled, the rate of data delivered to the database should also double.

3.7.1.2 Average CPU Utilization

The Average CPU Utilization is the proportion of the total available processor cycles that are consumed by each process between consecutive measurements. We configured Prometheus to take measurements every five seconds.

This metric is calculated from the kernel counters that are available through */proc/stat*, describing how many seconds each CPU spent doing each type of work:

- user: Time spent in user space
- system: Time spent in kernel space
- iowait: Time spent waiting for I/O
- idle: Idle time with no tasks to do
- irq&softirq: Time servicing interrupts
- guest: Time used by hosted virtual machines
- steal: Time stolen by other virtual machines for VM guests

These counters always sum to 1 for each CPU. The average utilization for all CPUs is calculated with the following expression:

```
100 - (avg by (instance) (irate(node_cpu{job="node",mode="idle"}[5m]))
```

3.7.1.3 Load Average

The load averages are reported in Unix systems as exponentially damped moving averages of the run queue. Thus, it is a measure of the load on the CPUs that also includes information about the processes that are waiting for CPU. Load averages are often reported in Unix systems for the 1, 5 and 15 minute marks.

We take the 1-minute load average of the instances during our tests, with the following query in Prometheus:

```
node_load1
```

3.7.1.4 Active Memory

Active memory is the total amount of buffer or page cache memory, in kilobytes, that is in active use and is usually not reclaimed for other purposes. Prometheus node exporters use the memory information available in */proc/meminfo*. It can be queried using the following expression:

```
node_memory_Active
```

Table 14. Parameters of the Baseline Scenario

Parameter	Configuration
Average Load from the Southbound	100-1500 requests per second
Payload size	10 bytes, 1000 bytes
IoT Agent Type	LWM2M, MQTT
IoT Agent Instance	m4.large
Orion Context Broker Instance	m4.large
mongoDB Instance	m4.large

3.7.1.5 Orion Query Latency

The generated load targets the southbound interface. This metric can show the latency performance of a query to the northbound under a given load level of data updates flowing into the southbound interface. It provides a measure of the way an entity query to the northbound interface of Orion Context Broker is affected with increasing load.

This measured latency is a value in milliseconds, as reported by a JMeter sampler when issuing a request for an entity attribute to the Context Broker. The target URL for the query is */v2/entities/entityId/attrs/attrName*, in which the entityId is chosen randomly.

3.7.2 Baseline Performance Scenario

The goal of this set of tests is to observe the performance of the platform according to the metrics we defined previously, using the basic configuration proposed in the documentation of FIWARE. This will provide a basis for comparison when scaling vertically or horizontally the components of the platform.

Performance is evaluated for a setup with the LWM2M IoT agent and a different setup with the JSON IoT agent for MQTT. A diagram for both setups is shown in figure 16. The average load per second is increased from 100 data observations per second to 1500 data observations per second. Each level of load is maintained for 180 seconds after reaching a steady state, and the 30 measurements of the performance metrics are taken during this period which are later summarized during post processing. The test parameters are listed in table 15.

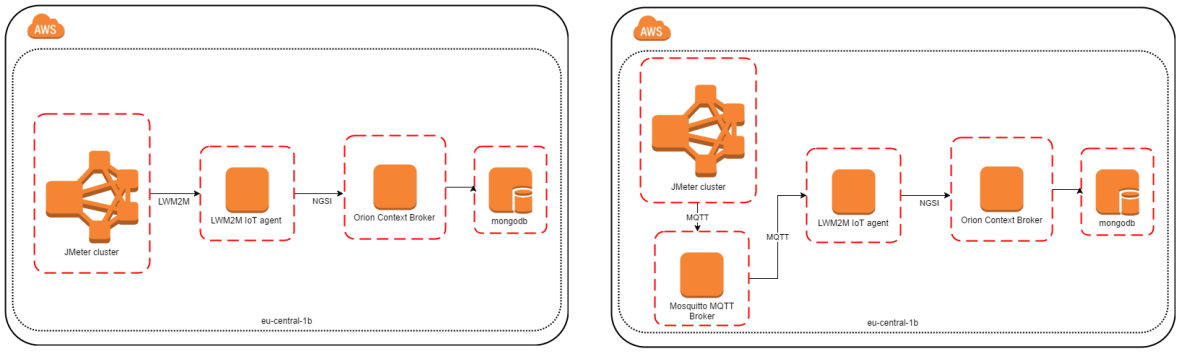


Figure 16. Setup for the test with the agent for LWM2M (left) and for MQTT (right).

Table 15. Parameters of the Vertical Scaling Scenario

Parameter	Configuration
Average Load from the Southbound	100-1500 requests per second
Payload size	10 bytes, 1000 bytes
IoT Agent Type	LWM2M, MQTT
IoT Agent Instance	m4.large
Orion Context Broker Instance	m4.large, c4.xlarge, c4.2xlarge
mongoDB Instance	m4.large, i3.large, i3.xlarge

3.7.3 Vertical Scaling Scenario

The goal of this set of tests is to observe the effect on performance when scaling vertically the components of our scenario. This is important for identifying bottlenecks in the system and scaling strategies, and also for finding the most efficient configurations in terms of throughput and cost for actual deployments.

We follow the connection diagram in figure 16 like in the baseline configuration, but we scale up the components in two dimensions, which can be seen in figure 17. On one axis we have the Context Broker, and on the other we have the instance for mongoDB. We evaluate the performance with all the different configurations for the parameters of our tests.

The IoT agents are not scaled up because we found that they are not able to take advantage of more CPU cores, due to being written in Node.js. Node.js uses an asynchronous processing model based on a single-threaded Event Loop, so a Node.js process cannot use more than one core [66].

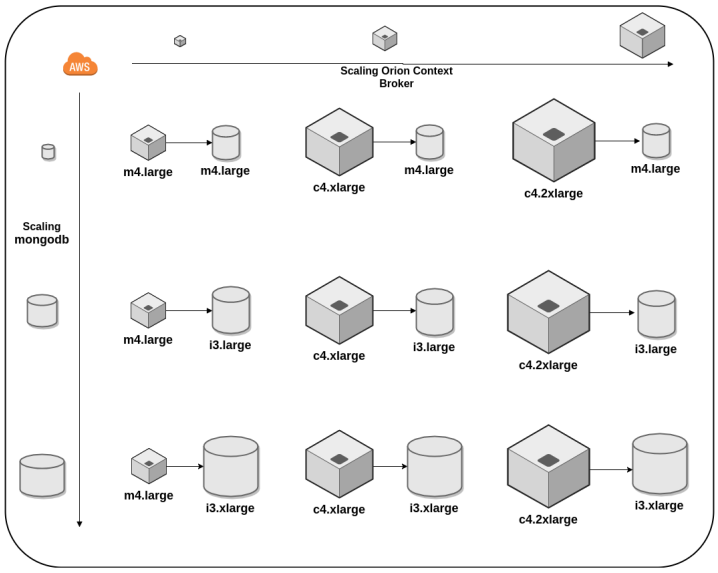


Figure 17. Scaling vertically increasing the instance capacity of Orion Context Broker and the mongodb database.

3.7.4 Horizontal Scaling Scenario

This scenario tests the scale out configurations of Orion Context Broker and mongoDB. The Context Broker can be scaled horizontally by deploying several instances behind a load balancer. Scaling out mongoDB involves the deployment of a sharded cluster, which divides a database in shards that can run in separate machines [67]. An illustration of both strategies is shown in figure 18

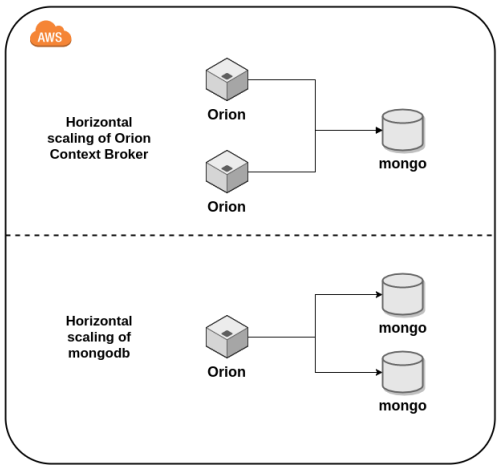


Figure 18. Scaling the Context Broker behind a load balancer versus the database using a sharded cluster

Table 16. Parameters of the Horizontal Scaling Scenario for Orion Context Broker

Parameter	Configuration
Average Load from the Southbound	100-1500 requests per second
Payload size	10 bytes, 1000 bytes
IoT Agent Type	LWM2M, MQTT
IoT Agent Instance	m4.large
Orion Context Broker	2 m4.large instances behind a load balancer
mongoDB	m4.large

3.7.4.1 Scaling Orion Horizontally

The parameters of the test are shown in table 16. The deployed instances differ from the baseline scenario by the addition of a load balancer behind Orion Context Broker. The IoT agent sends NGSI requests to the Context Broker through the load balancer, which alternates between the two Context Brokers. The Load Balancer is the Elastic Load Balancing (ELB) Service of Amazon Web Services [68]. This service distributes the incoming network traffic between different EC2 instances. A Classic Load Balancer was chosen, configured to distribute TCP traffic to Orion Context Broker's port (1026) equally between participating instances. The ELB service uses a round-robin algorithm to distribute incoming traffic [69]. This reduces the load that has to be processed by each Context Broker instance, allowing it to scale horizontally.

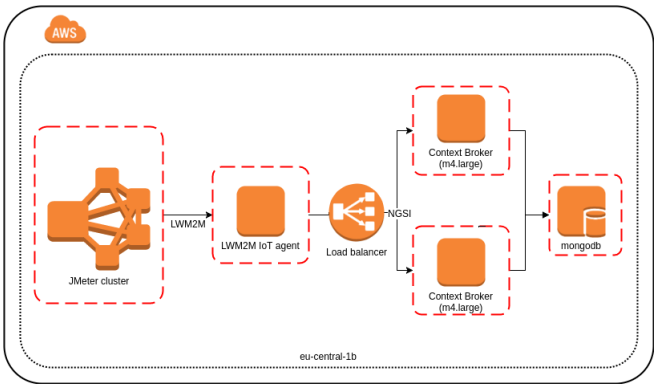


Figure 19. Scaling Orion Context Broker horizontally using a load balancer.

Table 17. Parameters of the Horizontal Scaling Scenario for mongoDB

Parameter	Configuration
Average Load from the Southbound	100-1500 requests per second
Payload size	10 bytes, 1000 bytes
IoT Agent Type	LWM2M, MQTT
IoT Agent Instance	m4.large
Orion Context Broker	m4.large, c4.xlarge, c4.2xlarge
mongos Router	m4.large
mongoDB Shards	m4.large

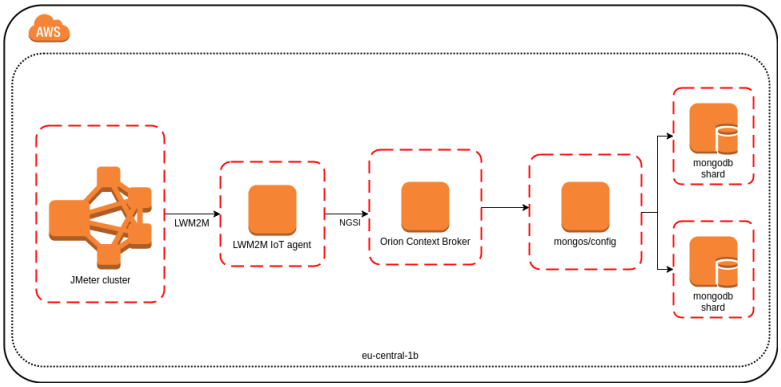


Figure 20. Scaling mongodb horizontally using a sharded cluster.

3.7.4.2 Scaling mongoDB Horizontally

The configurations used are presented in table 17. The schematic of the instances is shown in figure 20. A sharded cluster is the official scale out configuration for a mongoDB database. A sharded cluster places subsets of the data in different machines, instead of a single machine. The performance limitations due to lock contention and limited resource capacity in a single machine can be improved due to the distributed nature of the sharded cluster. A sharded cluster requires configuration servers and a server that runs the *mongos* application, which functions as a router that handles the database operations to the shards. The shards are often implemented as separate machines, and in production systems they run as part of replication groups [67]. Since running a production cluster is out of the scope of this work, the mongos router and the configuration server are run in a single instance, and the shards are run without a replication group. We deploy the minimum of two shards required for this type of cluster. We evaluate the performance with this baseline and then we also try different instance sizes for the Context Broker.

4 Results and Discussion

This chapter presents our performance evaluation of the FIWARE [14] components that allow IoT devices to connect to the FIWARE platform, and send data updates to the cloud backend (see section 3.4.1). The load generator in our testbed creates data updates using the MQTT and LWM2M IoT protocols, as described in section 3.4.4.

Results are presented for every configuration in the test plan (section 3.7). Every test was performed following the procedure previously described in section 3.6. The components are deployed in virtual machines with a baseline capacity, and then scaled vertically by using a larger instance or scaled horizontally through load balancing or database sharding. The performance metrics from the components in the testbed are summarized, presented and analyzed in this section. Every performance metric at every metric of load was summarized using the mean of 30 measurements in a steady state, along with its 95% confidence intervals. In the case of latencies the medians, quartiles and outliers are also included for visualization in boxplots.

The baseline performance is presented in section 4.1. The amount of requests per second at which the platform ceases to be robust and crashes in our configuration was found to be in the order of 1000 requests per second. Next, results obtained by deploying the platform in scaled up instances for Orion Context Broker and mongoDB are described in section 4.2. Slight performance improvements in throughput were achieved, which are not proportional to the increase in cost and CPU capacity of the larger instances. Performance with horizontal scaling is reported in section 4.3. Sharding the database proved to be the most effective strategy for increasing the throughput and keeping latencies consistently low. A discussion about the findings follows, reflecting also on the methodology and future challenges for testing IoT systems.

4.1 Baseline Performance

This section begins with the results obtained from the performance evaluation with a baseline configuration, as described in section 3.7.2, using instance types with the mininum capacity recommended by FIWARE's documentation. It serves as a point of comparison for the vertical and horizontal scaling results in sections 4.2 and 4.3.

4.1.1 Throughput to mongoDB

Throughput to the database increases linearly until the 900 data updates per second mark. There is a slight slowdown in the increase of the throughput when adding load after this point for our small payload of 10 bytes, while for the large payload of 1000 bytes there is a slight decrease (figure 22). Beyond a thousand data updates per second, the throughput sharply falls since the IoT Agent crashes. In our results, we have found that FIWARE's IoT Agents are unstable when the system goes beyond certain level of load. We observed that the agents sharply increase the number of open TCP connections and active memory before crashing due to lack of memory, instead of gracefully handling the load. At this point, the platform is not robust to handle the handle the incoming load.

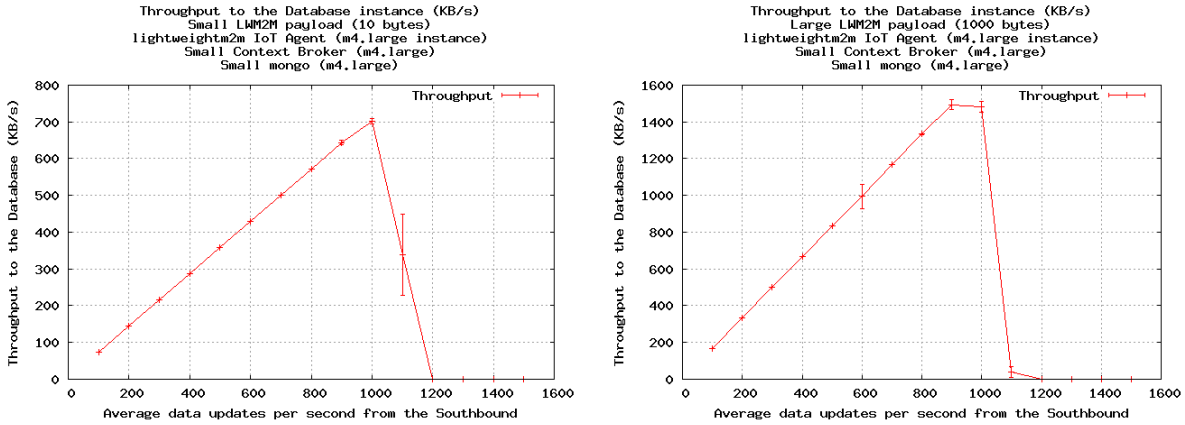


Figure 22. Throughput to the database in the baseline configuration for the LWM2M IoT Agent.

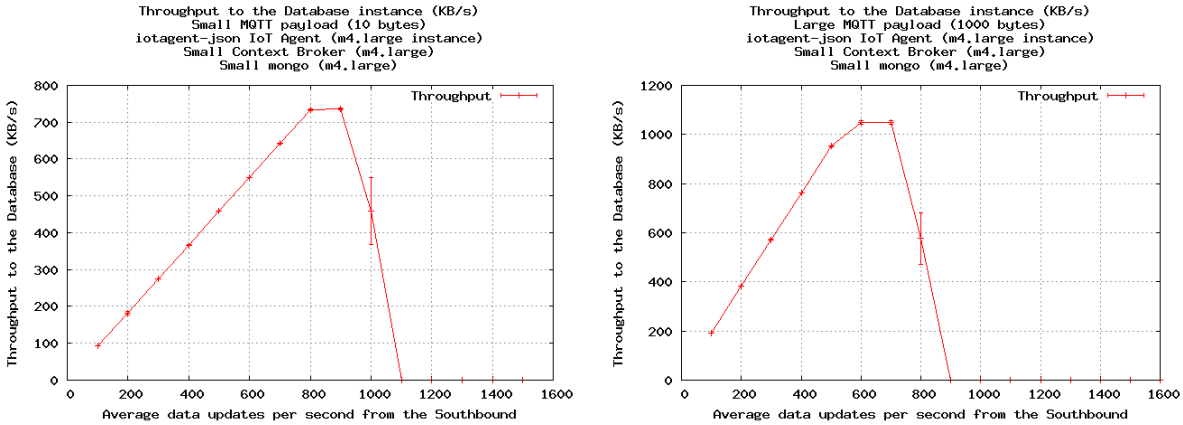
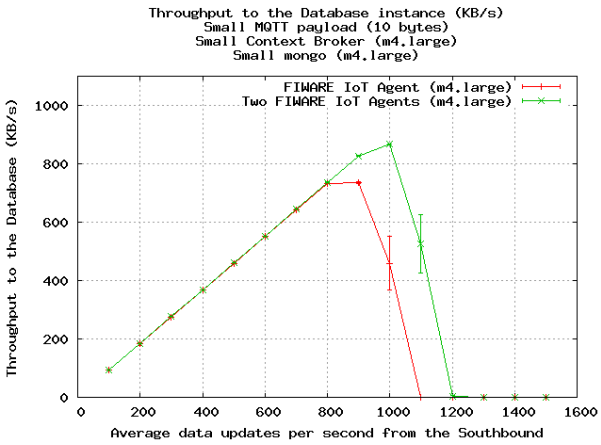


Figure 23. Throughput to the database in the baseline configuration for the IoT Agent for MQTT.

A similar behavior happens when using FIWARE's JSON IoT Agent for MQTT, as can be appreciated in figure 23. Both agents are based on the same underlying Node.js library, with

slight differences in their communications with the Context Broker after receiving data from their respective IoT protocols. The peak throughput with the agent for MQTT is lower than with the LWM2M IoT Agent for both payloads. Both are however close, within the same order of magnitude.

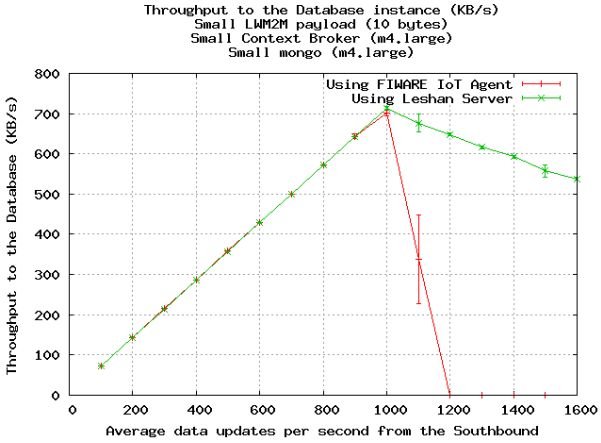
Resource utilization of the IoT Agent (described in the next section) was high. We tried to split the load between two different agents, effectively halving the amount of requests they each have to process while maintaining the same load on the Context Broker and the database. The results (figure 24) indicate that the improvements are slight. An improvement of around 2x should have been possible assuming that only the IoT Agent was the bottleneck, crashing because it couldn't handle the amount of requests. We would expect that two independent IoT Agents could each run until the previous breaking point, with a total throughput close to the double seen from one agent. Instead, both agents crash before that point. This result suggests that the Context Broker and the database also need to be able to scale to improve the amount of requests per second that can be handled. Scaling Orion and the database are the topic of the next sections.



Here, the fact that throughput doesn't double when the load is doubled suggests that Orion Context Broker and mongoDB also need to be scaled.

Figure 24. Throughput to the database using two separate IoT Agents

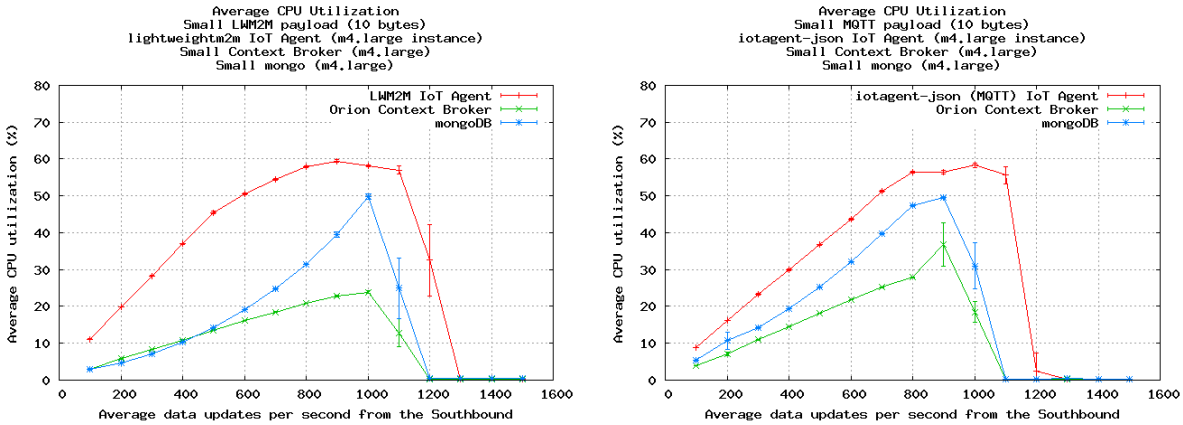
We also ran some of the tests with a customized LWM2M server that uses the open source Eclipse Leshan library. This is done when necessary to illustrate some particular points. We observed that the throughput to the database with the LWM2M server based on Leshan peaks (but does not crash) at the point where FIWARE's IoT Agent crashes (figure 25), which suggests that there are issues with the implementation of the IoT Agents that don't arise with other applications.



FIWARE's agent crashes and is unstable after the system is overloaded. Our custom Leshan LWM2M server doesn't crash, suggesting that FIWARE's implementation of the IoT Agent is faulty.

Figure 25. Throughput to the database with FIWARE's LWM2M IoT Agent and a custom Leshan server

4.1.2 Resource utilization

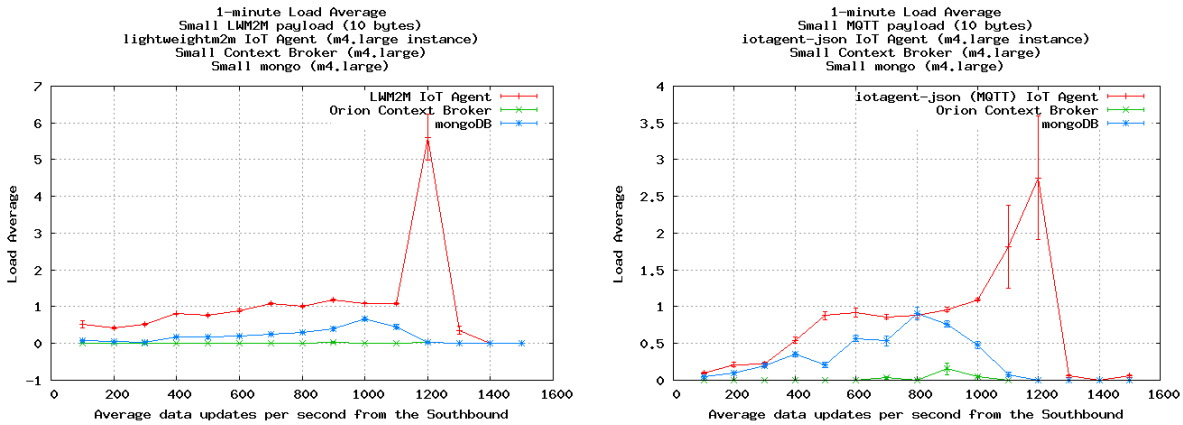


CPU utilization of the IoT Agents is high, but the capacity in other vCPUs cannot be used due to the limitations of Node.js

Figure 26. CPU utilization of the components of the platform with the agents for LWM2M (left) and MQTT (right)

Even though it is expected that different applications will have different usage patterns of computational resources, comparing them visually can highlight trends, provide insight into possible issues. CPU utilization of the LWM2M IoT Agent steadily increases linearly when handling loads ranging from 100 or 500 average sensor observations per second. After that point, CPU utilization increases more slowly and then sharply falls after 1000 observations per second since the application crashes (figure 26). The IoT Agent for MQTT shows a similar trend.

It should be noted that FIWARE's IoT Agents are written in Node.js. Node.js uses a single-threaded model and can achieve high utilization efficiency on one core. However, this model prevents the application from taking advantage of the multicore capabilities of modern hardware. As the load increases, the application is not able to use the other core in the *m4.large instance*. This limits the scalability of the application since it is only able to use one core while the capacity of the rest is wasted.



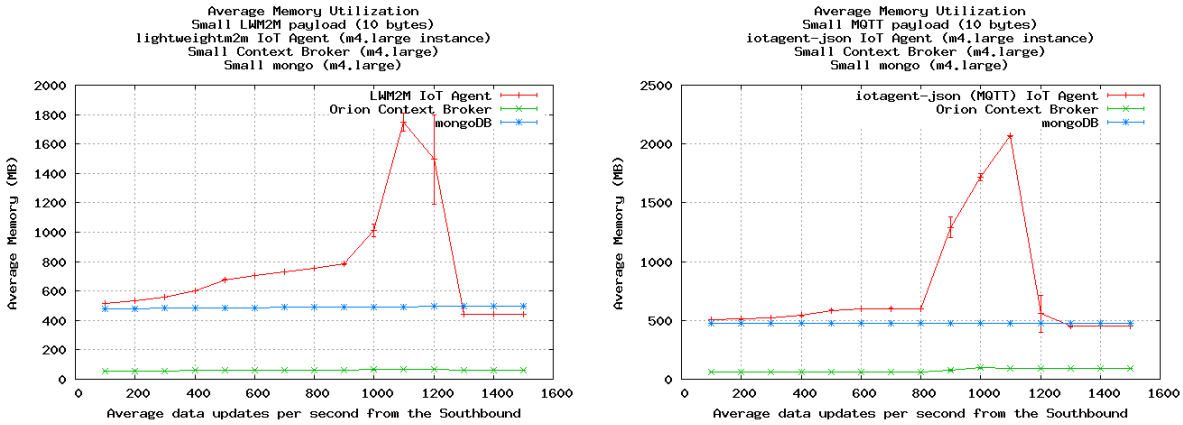
The load for both IoT Agents is high and spikes before they crash.

Figure 27. Load average of the components of the platform, using the agent for LWM2M (left) and MQTT (right).

In Unix systems, load averages are provided as a CPU utilization or saturation metric. These numbers are exponentially damped moving averages, which reflect load in the last 1, 5 and 15 minute marks. If the load average is higher than the CPU count, this means there are not enough CPUs to service the threads. A load average over the number of CPUs for an extended period of time is an indicator of saturation [41]. For our IoT Agent, which can only run in one core, a load average over one indicates that there is saturation and processing is queueing. Just before it crashes the load average increases sharply, going from 1 to over 5. In contrast, the load average does not increase considerably for the Context Broker, while it does increase for the database albeit gradually (see figure 27).

Figure 28 shows the memory utilization of the different instances in our scenario. We can see that memory usage is stable for Orion Context Broker and for the database. In contrast, the IoT Agent shows a very sharp spike in memory usage before crashing. Memory utilization is highest for the IoT Agent, followed by mongo and finally by Orion.

Resource utilization of the configuration that uses the MQTT IoT Agent shows the same pattern and is omitted here for brevity. The spikes in load and memory usage also correspond to the point where the IoT Agent for MQTT crashes.



The memory used by the IoT Agents spikes before they crash, when the system is overloaded. Memory usage of the other components remains stable.

Figure 28. Average memory utilization for all components

4.1.3 Latencies

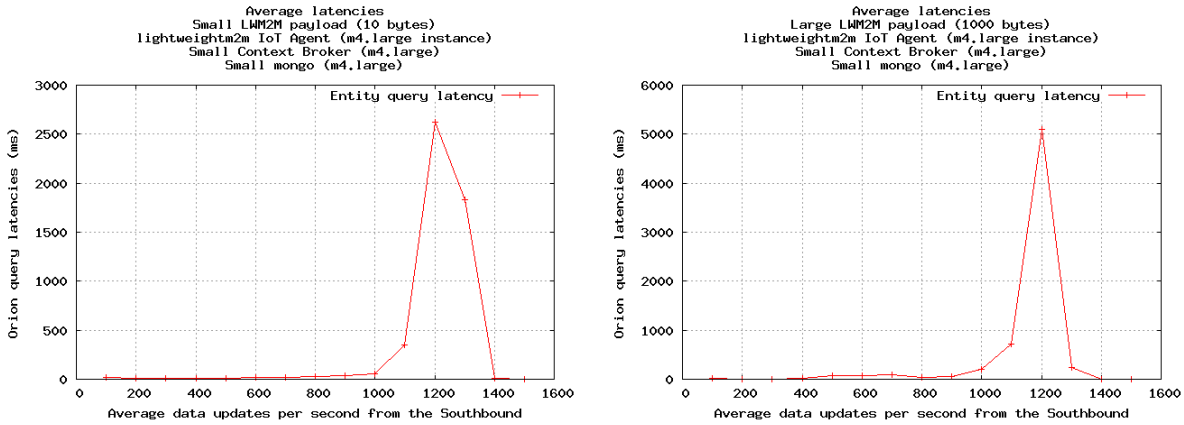
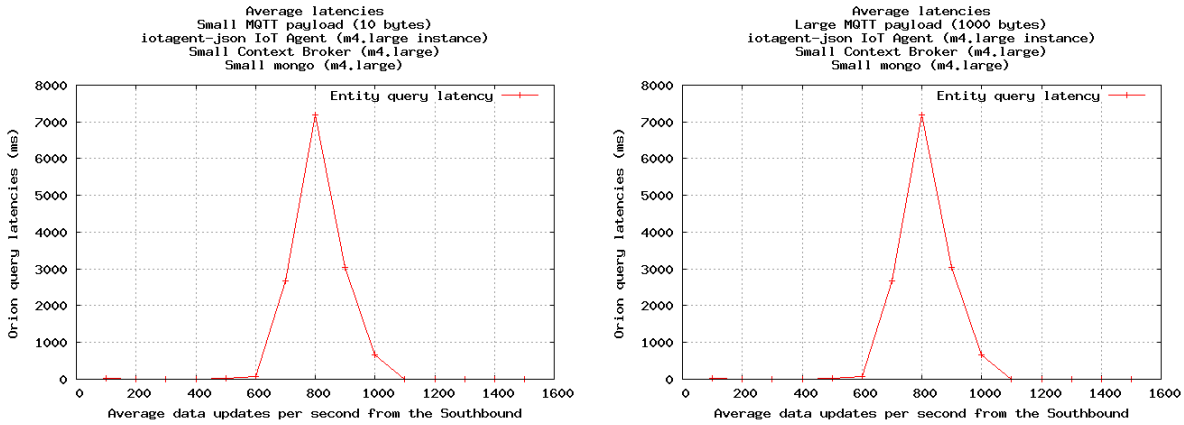


Figure 29. Growth of average latencies of query entities to the Context Broker under load, with the baseline configuration with the LWM2M IoT Agent and for both payload sizes.

For the latencies we computed the average latencies, to show the overall trend of the growth of latencies under increasing load (figure 29 and 30). Next, we also show the median, quartiles and outliers (figures 31 and 32) to give a more accurate representation of the performance of the queries, since the presence of large outliers in the queries hides the behavior of the more average cases. Outliers larger than 100 ms are not shown, since these can have values of several seconds and would prevent the visualization of the lower values. Initial connection establishment causes a latency over 100 ms for the first measurement (seen at the 100 requests per second mark).

Average latencies for entity queries to the Context Broker increase sharply for both payload

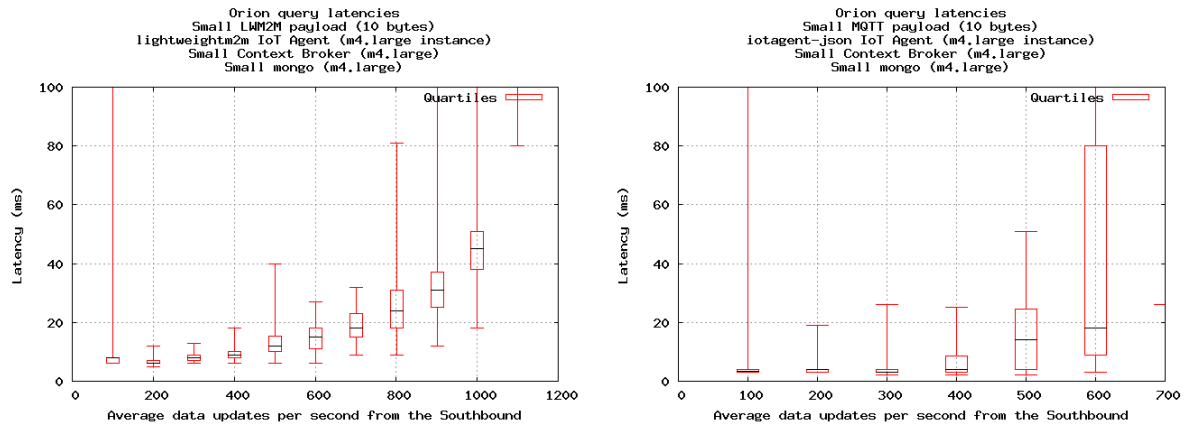


Since the configuration with the IoT Agent for MQTT crashes earlier, the spike happens earlier.

Figure 30. Growth of average latencies of query entities to the Context Broker under load, for the baseline configuration with the IoT Agent for MQTT and both evaluated payload sizes

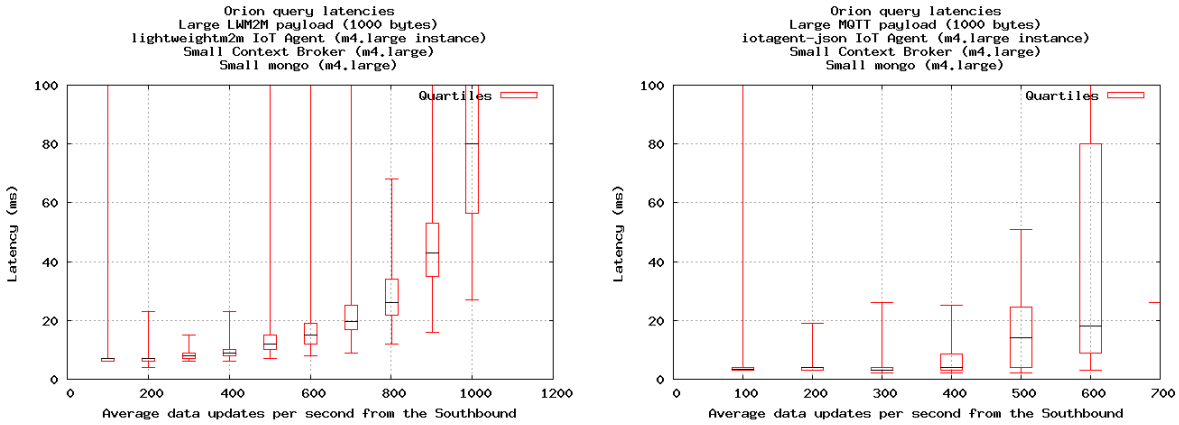
sizes after one thousand data updates per second, when the system is overloaded. These queries target the northbound interface of the Context Broker, for getting context information of entities and their attributes. Under one thousand updates per second, the median query latency and its variability increase with load, but stay under 50 milliseconds when measured from the JMeter cluster in our setup as can be seen from the quartiles in the plots.

The sharp increase in latencies can affect user-facing apps and M2M applications that interact with the Context Broker. Some applications are tolerant to delays, while others have tight latency constraints.



The left image shows the trend when using the system with the LWM2M IoT Agent and right with the MQTT IoT Agent. Outliers are due to the overloaded system taking long times to serve some queries (compare to the best-performing configuration in figure 53).

Figure 31. Performance of entity queries to the Context Broker when the system is not overloaded



Compare the trend with the large payload to the behavior in figure 31. Differences between both payloads are not considerable, except that the LWM2M IoT Agent crashes earlier with the 1000 bytes payload.

Figure 32. Latencies for the configurations with both agents using a large payload

The agent for MQTT crashes earlier than the LWM2M IoT Agent and the growth in the latencies reflects this behavior. The latencies for the LWM2M IoT Agent with the 10 bytes payload (figure 31) have less outliers over 100 ms in comparison to the 1000 bytes payload. The medians are similar, however. Even though the difference for IoT transmissions between these two payloads is large, both are small in comparison to the service demand of web applications, with images, stylesheets, scripts, hypertext and JSON data that can have a size of several KB.

As IoT applications grow and more context information is stored, the size of datasets will also have an impact on the database. The database in these tests holds entities in the order of magnitude of thousands. When the entities number millions and datasets are large enough to not fit into memory and caches, performance variations from indexes, caching, algorithmic complexity and IO will affect the performance of the overall system.

4.1.4 Summary

For all the IoT protocols and payload sizes evaluated, the system can handle a few hundred requests per second (depending on the particular IoT Agent used). When the load is increased beyond 700 or 1000 requests per second (depending on the IoT Agent, see section 4.1.1) the IoT Agents crash.

CPU and memory utilization of Orion Context Broker are stable and relatively low. CPU demand in the database grows fast to handle the updates, while memory is stable and also relatively low. CPU utilization is high for the IoT Agents and memory is unstable when the system

is challenged.

When data of entities in Orion Context Broker is queried, latencies spike from less than 100 ms to several seconds as the rate of updates from the southbound approaches the point where the IoT Agent crashes.

4.2 Scaling Vertically

Vertical scalability was tested by using more powerful instances for the Context Broker and the database, as described in section 3.7.3. In theory, the Context Broker is able to scale vertically by using its multicore capabilities to process more requests simultaneously. It is necessary to enable the application parameters for the requests thread pool and to set the mutex policy to *none* to enable the parallel processing of API requests, as described in section 3.5. On the other hand, scaling mongodb vertically is done by using the instance type optimized for IO and increasing the available cores and memory.

4.2.1 Scaling Orion Context Broker

4.2.1.1 Throughput to mongoDB

Figures 33, 34, 35 and 4.2.1.1 show the increase in the throughput to the database when increasing the instance capacity of Orion Context Broker, for a given database instance. For all three tests with different mongo instances, the peak throughput shows a larger increase when going from a *m4.large* with 2 virtual CPUs to a *c4.xlarge* instance with 4 virtual CPUs. The increase is larger for the payload of 1000 bytes compared to the 10-byte payload. However, going from a *c4.xlarge* instance with 4 vCPUs to a *c4.2xlarge* instance with 8 vCPUs doesn't show a comparable increase in the throughput to the database.

The gains in throughput are not very promising, specially considering the cost increase for operating larger instances. The underlying reason can be related to the Context Broker not being able to use all cores efficiently or the capacity of the database instance being already saturated. In section 4.3.2 we found that the latter is the main cause, and with a sharded cluster for the database scaling up the Context Broker can increase the throughput.

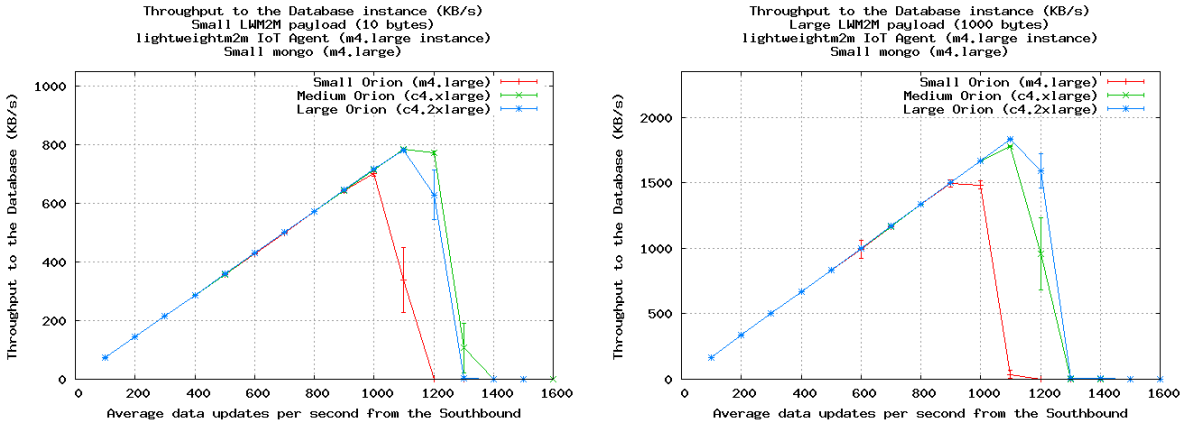


Figure 33. Effect of using increasingly larger instances for Orion Context Broker, keeping the database instance (m4.large) fixed.

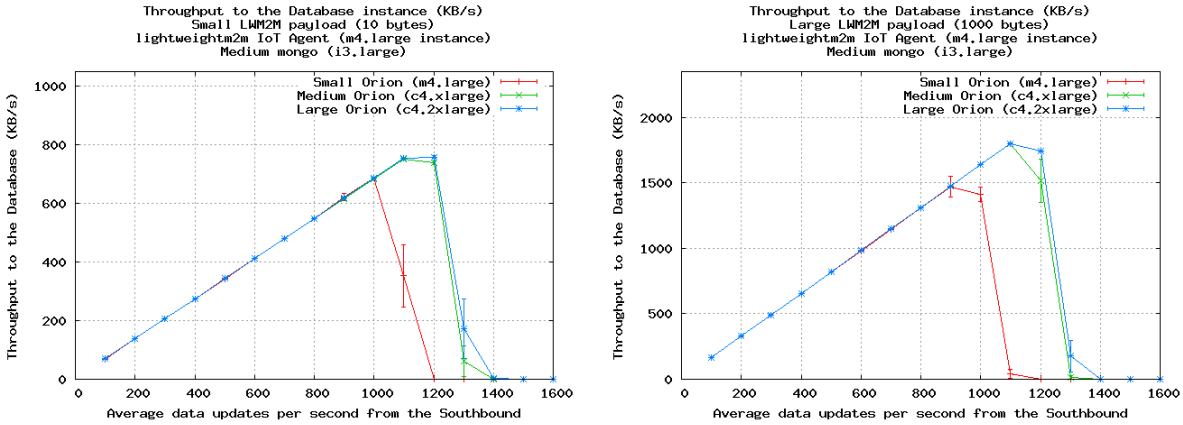


Figure 34. Effect of using increasingly larger instances for Orion Context Broker, keeping the database instance (i3.large) fixed.

4.2.1.2 Resource Utilization

When using a more powerful instance for the Context Broker, while keeping the same type of database instance, the CPU utilization of the database surpasses the utilization of the IoT Agent and the utilization can be driven further in comparison to the baseline configuration. It can be seen that the IoT Agent is operating to its maximum capacity and that the demand for CPU in the database grows faster than linear growth (figure 37).

The other resource metrics show similar behavior to the baseline scenario, namely the abnormal spikes in memory and load average for the IoT Agent. They are omitted here for brevity.

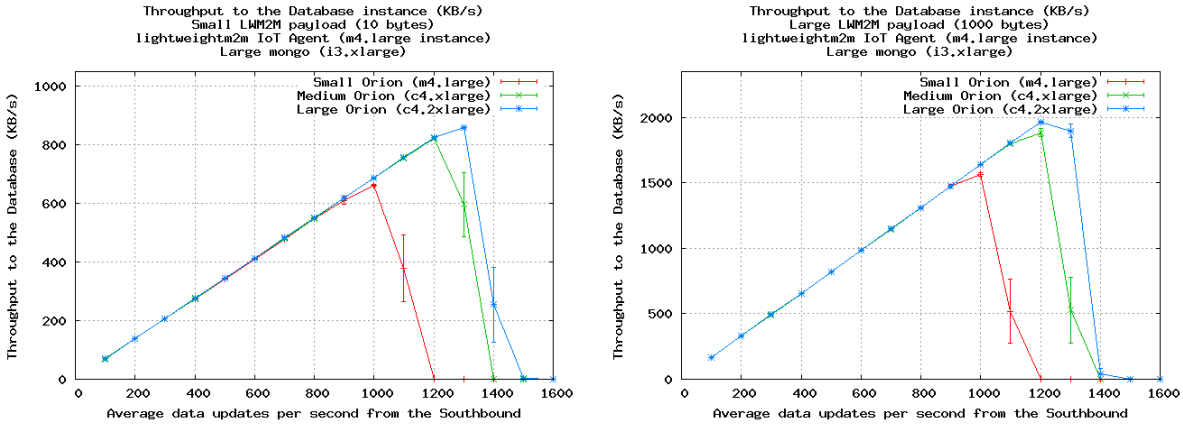
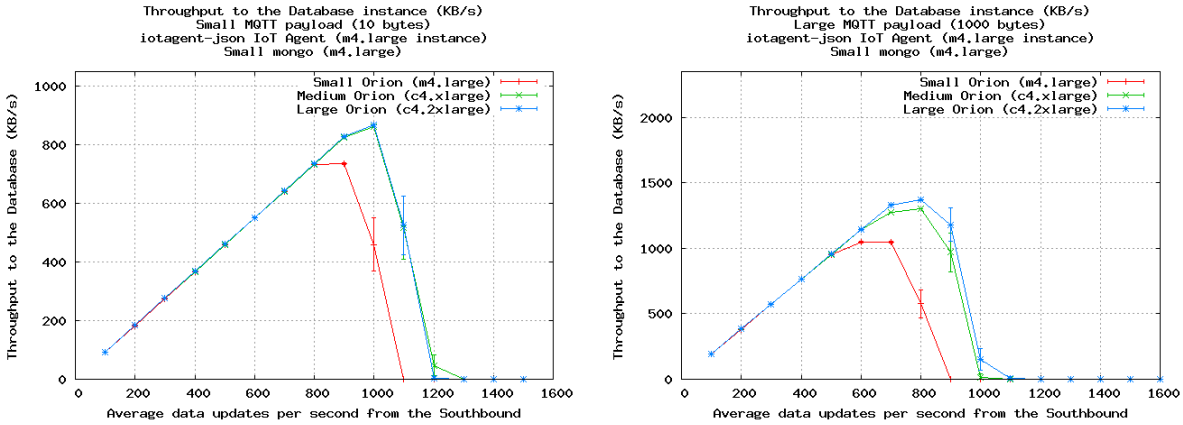


Figure 35. Effect of using increasingly larger instances for Orion Context Broker, keeping the database instance (i3.xlarge) fixed.



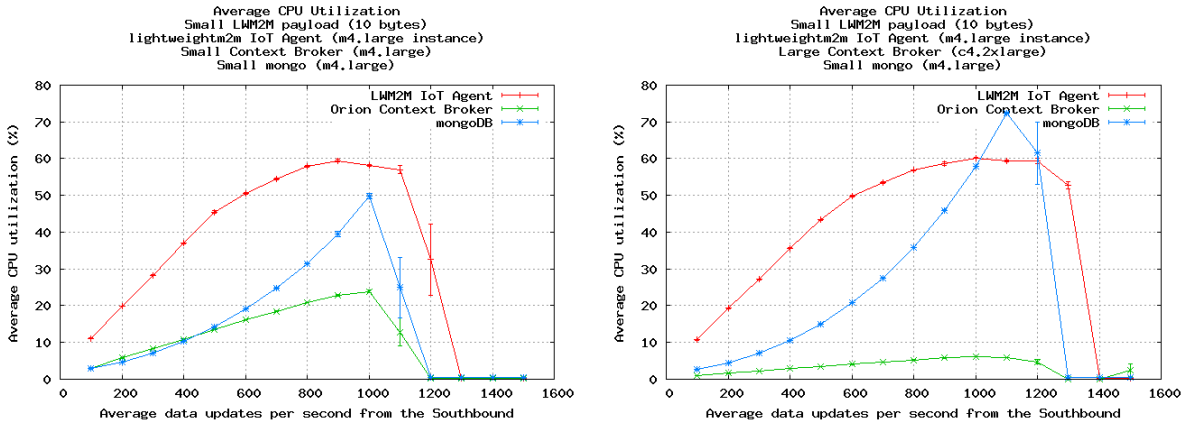
The behavior with the agent for MQTT is similar to the one with the LWM2M IoT Agent. Performance is improved when switching to a medium Context Broker, but there isn't a proportional difference when switching to a large Context Broker.

Figure 36. Throughput with the IoT agent for MQTT

4.2.1.3 Latency

Using more powerful instances for the Context Broker displays the same behavior as the baseline configuration, with low latencies that increase very gradually until a certain point where they rapidly increase from several milliseconds to several seconds (figures 4.2.1.3 and 40). For low levels of load, the median latencies are very similar between different instances for the Context Broker. In figure 38 we show the median latencies measured with the three different configurations for the Context Broker, and mongoDB in a *i3.xlarge* instance. The results are similar with the other instance types for the database, and are omitted here for brevity.

Latencies only begin to differ close to the point where the system is overloaded for a particular configuration of the FIWARE components. Therefore, it is not possible to improve the latencies



CPU utilization in the baseline configuration for the LWM2M IoT Agent can be seen on the left. On the righ, CPU utilization for the scaled up configuration with the largest evaluated instance for the Context Broker.

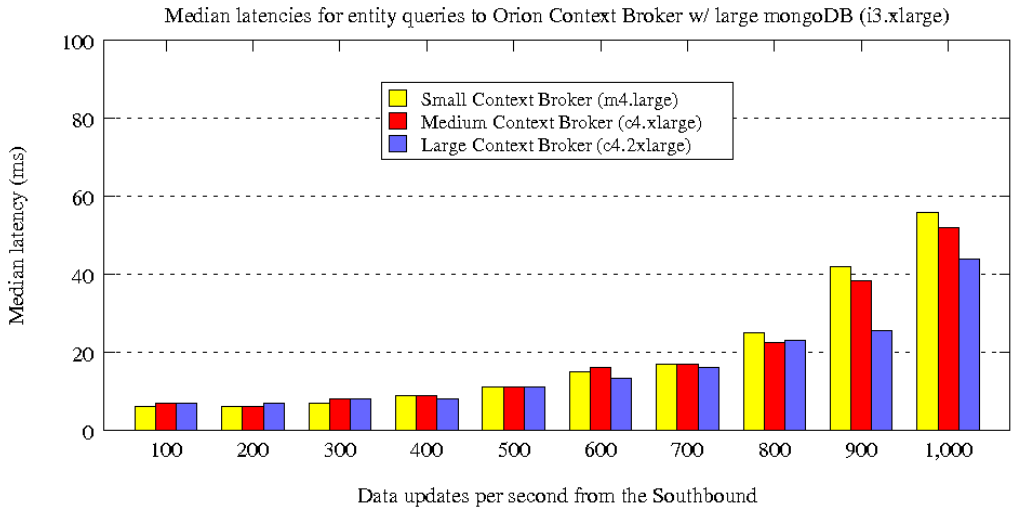
Figure 37. Comparison of CPU utilization between a baseline configuration and a scaled up Orion Context Broker

for lower levels of load by using a more powerful Context Broker, according to our results. We also scale up the database in section 4.2.2 and evaluate its impact on latencies and throughputs.

In our tests, scaling up the Context Broker involved going from a *m4.large* instance with 2 vCPUs to *c4.xlarge* and *c4.2xlarge* instances with 4 and 8 vCPUs, respectively. Their monthly costs in the eu-central Frankfurt datacenter of Amazon Web Services are 94.43 USD, 166.17 USD and 332.33 USD, respectively. Increases of up to 30% in the peak rate of requests from the southbound that can be handled were achieved, while no noticeable improvements for entity query latencies could be observed. The number of vCPUs and the cost almost doubles for each succesive instance type, but the performance gains are not proportional. In table 18 we make a comparison of the monthly costs of the different configurations that we evaluated and their cost efficiency regarding the peak rate of requests that could be handled from the southbound interface.

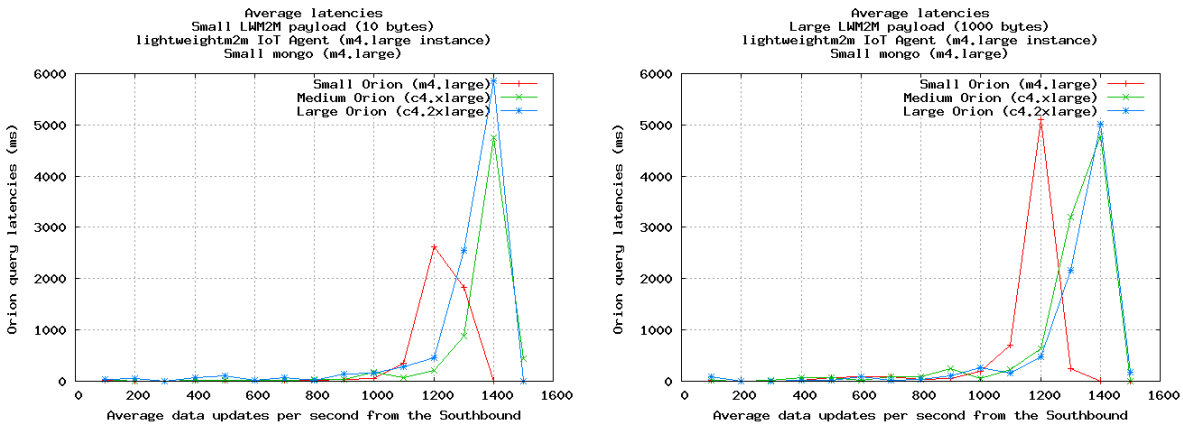
4.2.2 Scaling mongodb

The other dimension for scaling our setup to receive data updates from the southbound interfaces involves scaling up the database instance. In this section we describe the results we obtained by going from a basic general purpose instance with two vCPUs to IO optimized instances with 2 and 4 vCPUs, respectively.



Latencies were only improved close to the point where the system is under heavy load and the IoT Agent crashes.

Figure 38. Median latencies for different levels of load in a large database instance (i3.xlarge)



The configuration with the small Context Broker is overloaded before the others, which can be seen in the respective spike in latency.

Figure 39. Evolution of latency vs load when using a more powerful instance for the Context Broker, while using the smallest evaluated instance for the database (m4.large)

4.2.2.1 Throughput to mongoDB

The updates per second to the database that can be handled by the platform show only a small improvement for the larger payload when using a more powerful instance for the database and using the smallest instance in our tests for the Context Broker. If a more powerful instance is used for the Context Broker, then slight improvements can be achieved by upgrading the database to an instance with 4 vCPUs. Changing from a general purpose to an IO optimized instance (*m4.large* to *i3.large*) with the same number of vCPUs did not improve the maximum amount of requests handled by the system. It is possible that a difference could be noticeable

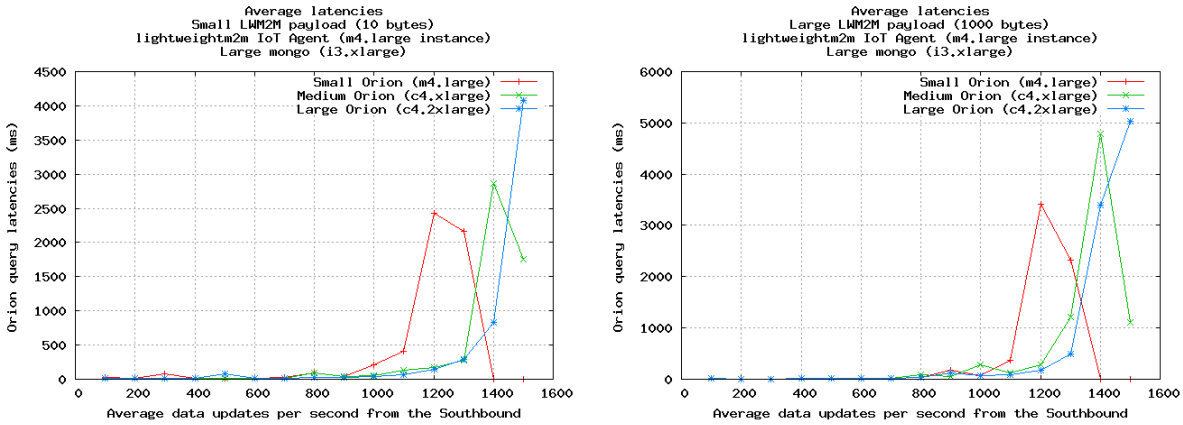


Figure 40. Evolution of latency vs load when using a more powerful instance for the Context Broker, using the largest evaluated instance for the database (i3.xlarge).

with a database so large that not all the data can fit into caches.

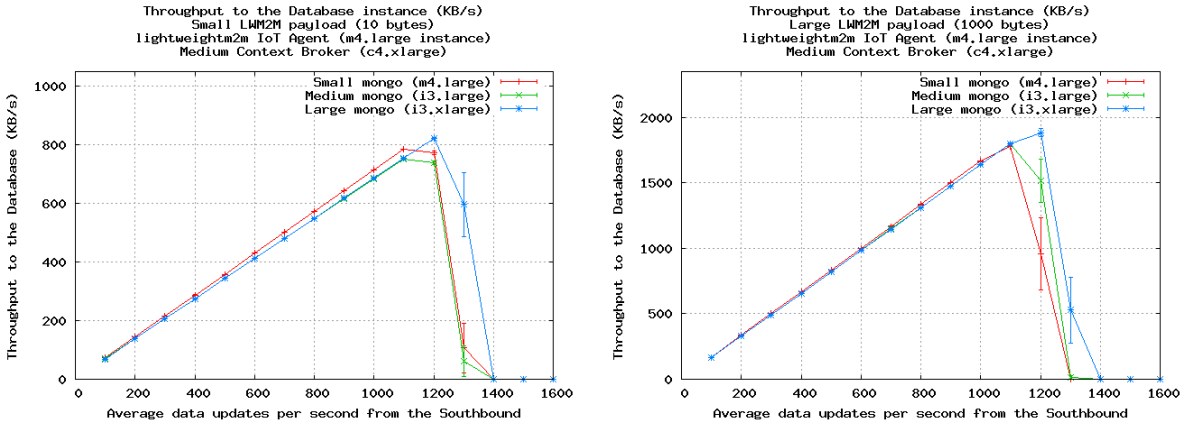


Figure 41. Effect of vertically scaling the database instance with a medium Orion instance (c4.xlarge).

4.2.2.2 Resource Utilization

The same trends that were observed previously in section 4.2.1.2 also apply when scaling up the database instance.

Memory utilization is stable throughout all the scenarios for mongoDB, staying around 400 MB. CPU utilization always grows quickly. In figures 41 and 42 it can be noticed that the throughputs achieved with the small (*m4.large*) and medium (*i3.large*) instances for the database are close, while with the large (*i3.xlarge*) instance more requests can be processed. The resource usage view shows that the load average when the throughputs are similar are close, while the load average is lower for the best performing configurations.

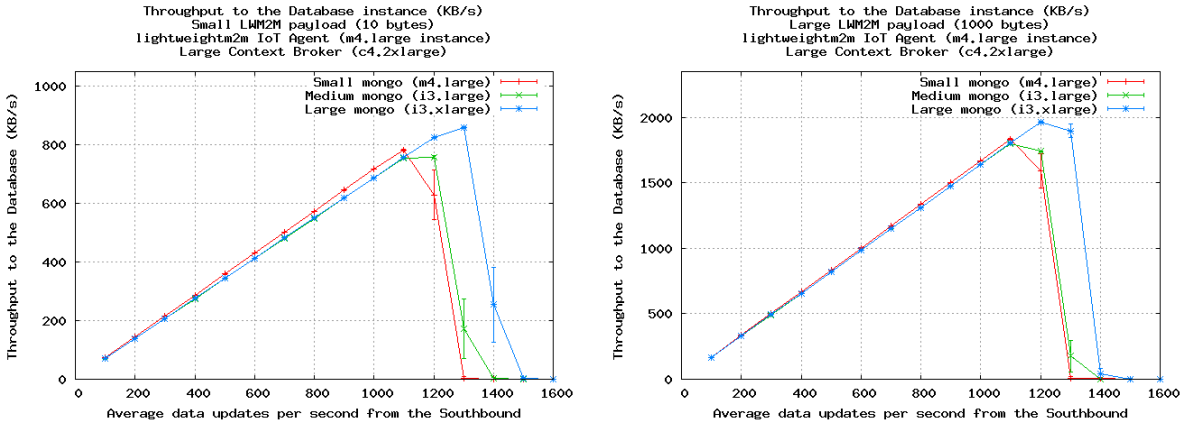
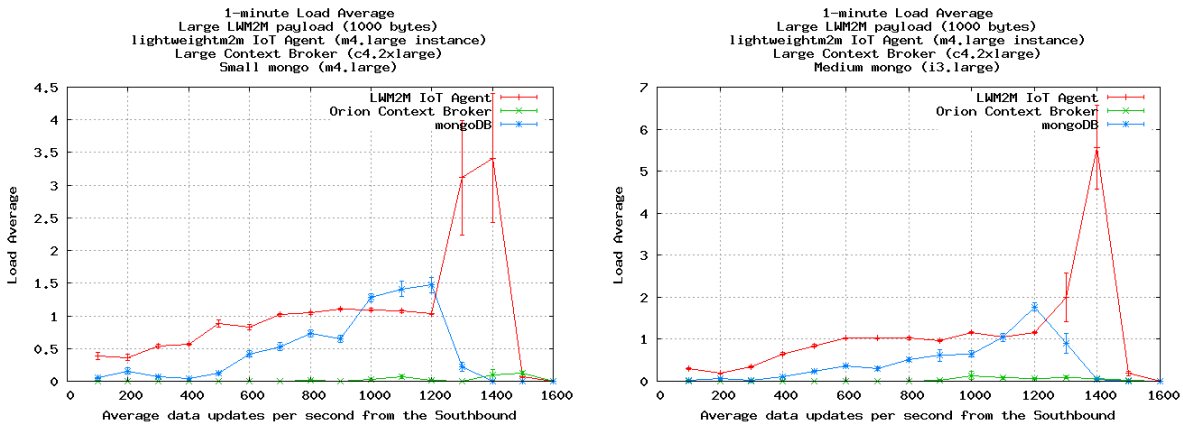


Figure 42. Effect of vertically scaling the database instance with a large Orion instance (c4.2xlarge).



Similar load averages for the similar-performing m4.large and i3.large mongoDB instances in figure 42. The m4.large instance has a slightly higher load average and slightly lower performance with respect to requests per second handled42.

Figure 43. Comparison of load average between scaled up mongoDB instances

The number of requests per second that can be processed is lower whenever the database is highly loaded, which is reflected in the load average.

4.2.2.3 Latency

The latencies for entity queries to the Context Broker did not show any considerable improvement when using a more powerful instance for mongoDB. When the system is not overloaded the median latencies are similar, and at the level of load where the system is overloaded the latencies grow from under 60 milliseconds to several seconds, as described in section 4.2.1.3

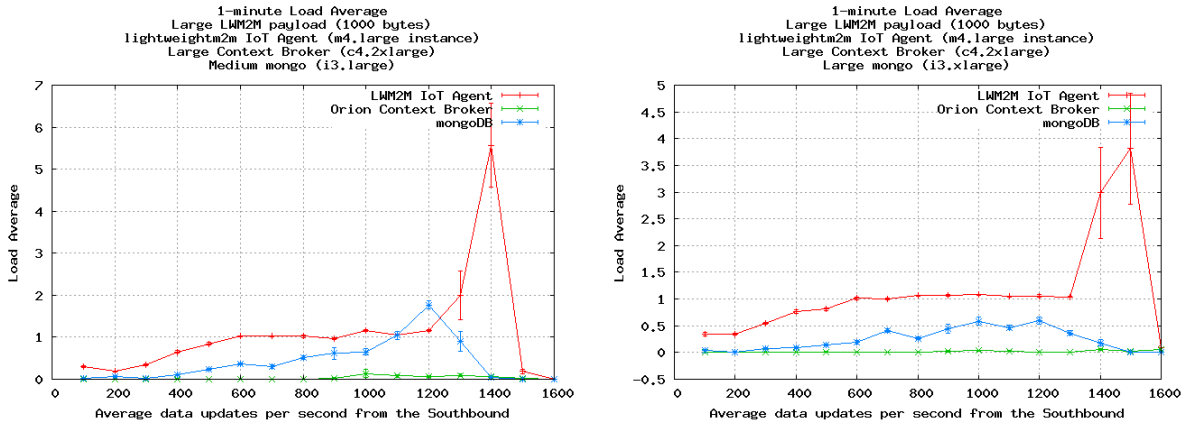


Figure 44. The load average is lower for the large database instance (i3.xlarge), which also supports the highest number of requests per second.

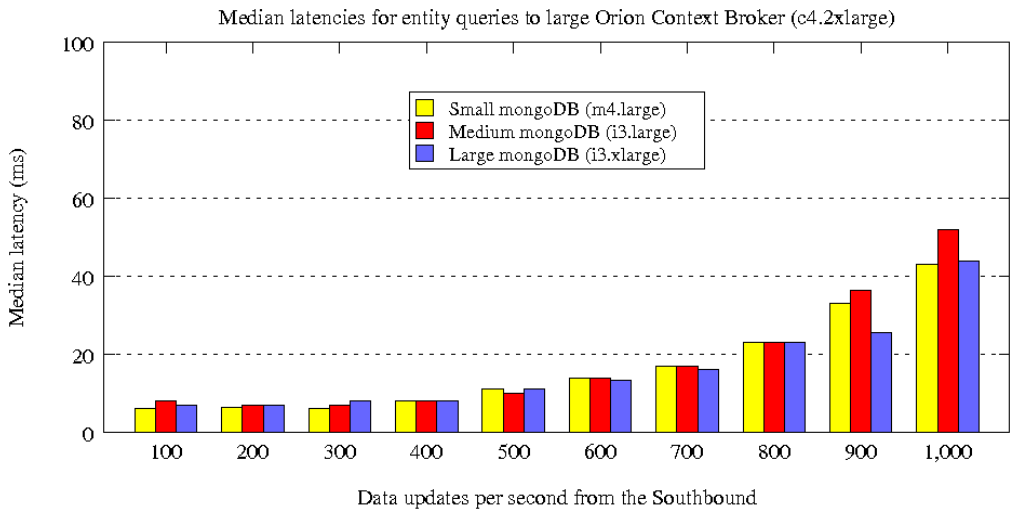


Figure 45. Median latencies for each of the evaluated database instances, with a large instance (c4.2xlarge) for the Context Broker.

4.2.3 Summary

After scaling up Orion Context Broker and mongoDB, the same trends that were observed in the baseline configuration continue. The system can handle more requests per second (with improvements up to 30% compared to the baseline configuration), but the IoT Agents keep crashing. The same patterns of utilization of computational resources (CPU, memory, load average) can be observed, but the additional requests per second handled increase the CPU utilization and the load average in mongoDB's instance. The IoT Agents are challenged to handle the load, but these additional tests reveal that more processing power is required to handle the updates in mongoDB.

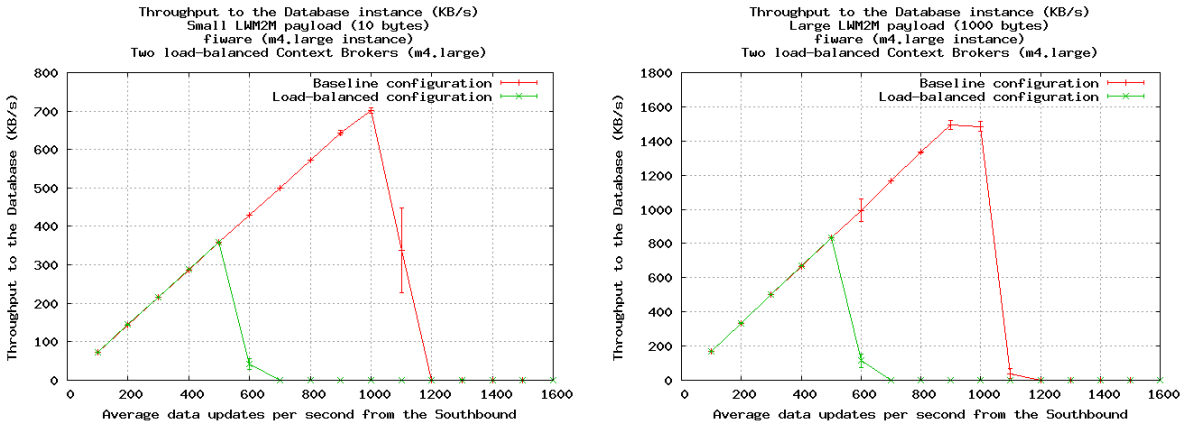
4.3 Scaling Horizontally

This section presents the results obtained with scale out configurations of the Orion Context Broker and mongoDB.

4.3.1 Scaling Orion horizontally

4.3.1.1 Throughput to mongoDB

Making the IoT Agent communicate with several Orion Context Brokers has the counterintuitive effect of making the IoT Agent crash earlier, handling considerably less load (figure 46). In this setup, the IoT Agent crashes after five hundred data updates per second from the southbound instead of beyond a thousand data updates per second like in the baseline scenario. This happens because Orion Context Broker is not a bottleneck at those particular levels of load, and scaling it horizontally does not relieve the pressure on mongoDB. This is only apparent after extensive review of resource usage metrics, and observing the gains in performance when scaling out mongoDB (at which point the database is not a bottleneck anymore). It should be noted that our custom Leshan LWM2M server doesn't crash. The achieved throughput levels off at the point where the load average surpasses 1 in the database instance (figure 49).



Throughputs to the database using a configuration with two load balanced Orion servers. No more than 500 requests per second can be handled, since FIWARE's IoT Agent crashes.

Figure 46. Throughput using load balancing with Orion Context Broker

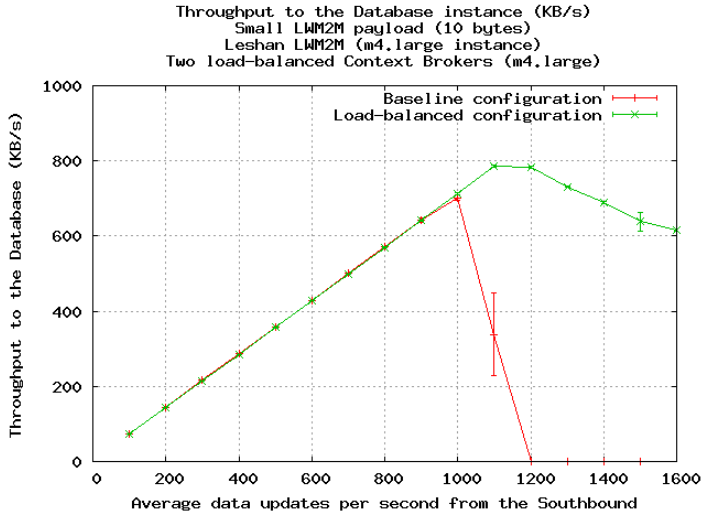
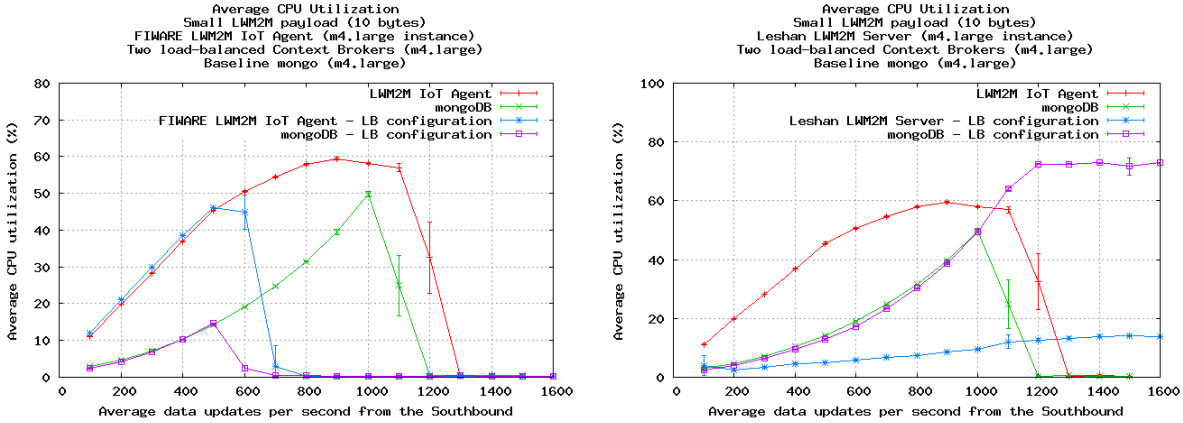


Figure 47. Throughput to the database using Leshan and two Orion Context Brokers behind a load balancer (LB).

4.3.1.2 Resource Utilization

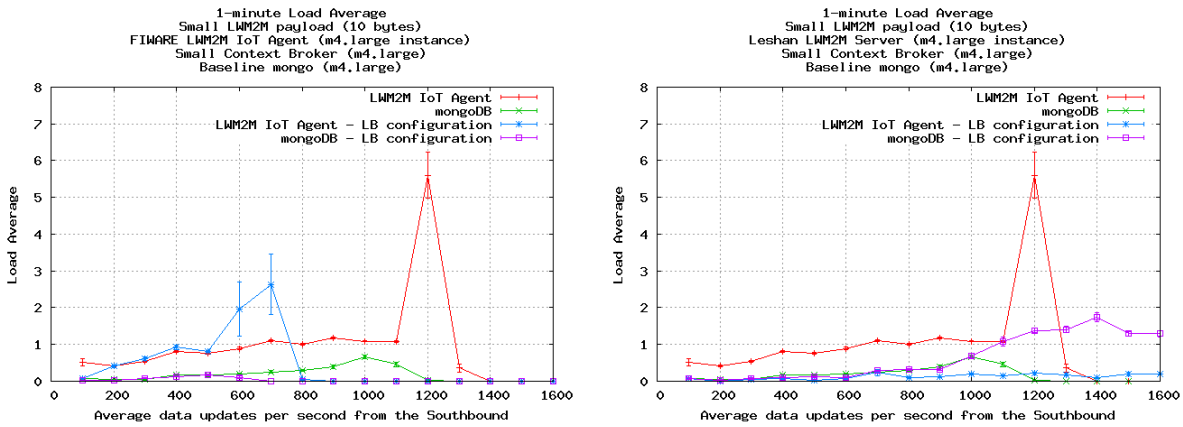
In figure 48, the CPU utilization of the IoT Agent and the database is shown. On the left side, a comparison between the baseline configuration and the load balanced configuration shows that when load balancing is used the IoT Agent crashes after 500 requests per second. Our customized Leshan, on the other hand, does not crash. Its CPU utilization stays under 20%. The CPU utilization of the database levels off around 70%, and is able to be maintained at this level because the Leshan LWM2M server can continue sending the update requests unlike FIWARE's IoT Agent.

In figure 49, it can be seen that when the load average of the database surpasses 1 with the Leshan server, the throughput stops increasing. The database is at its full capacity for handling updates at this point. These results point to the database (its processing power in particular) as the critical resource to scale in order to improve the rate of requests that can be handled. However, our evaluation of vertical scalability found that the database is also overloaded when using a larger instance with more cores. The database is not able to take advantage of the additional cores for handling the updates. The documentation manual of mongoDB states that sharding improves concurrency by distributing collections over several instances. Locks apply to each shard and not to the whole database. Implementing a sharded cluster can increase the number of cores and available processing power by distributing the load over several small instances. If mongoDB can only use one core per instance for the updates, by having a sharded cluster with n instances the limitations in a single instance can be overcome. In section 4.3.2 we evaluate performance when scaling the database layer horizontally, by implementing a sharded cluster.



The early-crashing FIWARE IoT Agent is shown on the left and a Leshan LWM2M server on the right.

Figure 48. CPU utilization for the baseline and load-balanced scenarios



Using the early-crashing FIWARE agent on the left and a Leshan LWM2M server on the right. FIWARE's agent load average spikes and the agent crashes. With the Leshan LWM2M server, the load on the LWM2M server is low, while the load on the database increases since the system is working correctly and can deliver the updates from the southbound.

Figure 49. Load average for the baseline and load-balanced scenarios

4.3.2 Scaling horizontally - Sharding mongodb

The previous tests showed that scaling vertically the instance of the Context Broker and the database produces slight gains in the rate of requests handled. However, only 100 to 300 more requests per second were supported by doubling or quadrupling resources like the number of cores in an instance. In this section we complete the configurations to be tested by evaluating performance of the platform by scaling the database layer horizontally with a sharded mongoDB cluster.

4.3.2.1 Throughput to mongoDB

The throughput to the database using the sharded cluster has a different rate of traffic growth because of the communication with the shards. Using FIWARE's IoT Agent, the rate of updates per second processed from the southbound interface is improved, going from 1000 to 1200 or 1400 depending on the payload size (figure 50). At first sight, this is not different from the gains achieved through vertical scaling. However, when replacing FIWARE's IoT Agent, which we know to be unstable from previous tests, we achieve the throughputs and the rates of updates from the southbound shown in figure 51

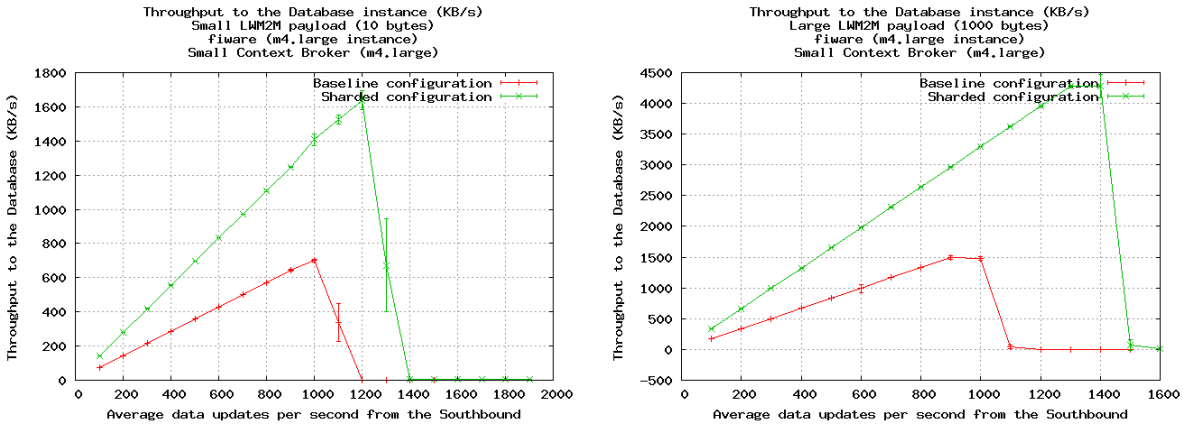
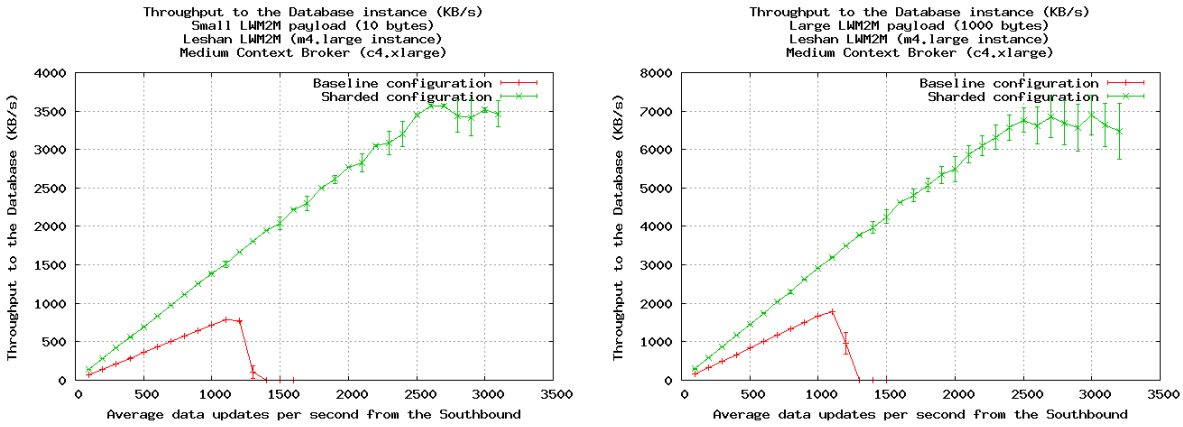


Figure 50. Throughputs to the database using a configuration with two mongoDB shards and FIWARE LWM2M IoT Agent, for both evaluated payload sizes (10 and 1000 bytes).



This configuration scales better than all the others.

Figure 51. Throughputs to the database using the scale out configuration for mongoDB with two shards and a Leshan LWM2M server, for both evaluated payload sizes (10 and 1000 bytes)

The rate of requests from the southbound that can be handled increases considerably in compari-

son to the baseline configuration, as can be seen in the charts. The results with this configuration show that both FIWARE's IoT Agent and the database become a bottleneck when handling thousands of data updates from the southbound. The tests with scale up configurations and the test with the Context Broker behind a load balancer showed that the demand for CPU in the database instance has superlinear growth with the rate of updates to process. The throughput could not increase after the load average in the database instance surpassed 1. The sharded cluster provided the performance for updates needed to support more load.

At the same time, CPU utilization and the load average for the IoT Agent were already at their maximum even before the database is overloaded. FIWARE's IoT Agents continue to crash with the sharded database cluster. Even if the database has the capacity to support the rate of updates, high throughputs cannot be achieved due to the lack of robustness of the IoT Agents.

With the sharded configuration, using a more powerful Context Broker can drive the throughput higher (figure 52). However, the load average, memory utilization and CPU utilization of the Leshan server are approaching their limit at this point. This would make it necessary to scale this layer. FIWARE's IoT Agents cannot scale up and in the documentation it is not stated whether they can scale out. In the Leshan project, a clustered mode to scale horizontally is currently under development.

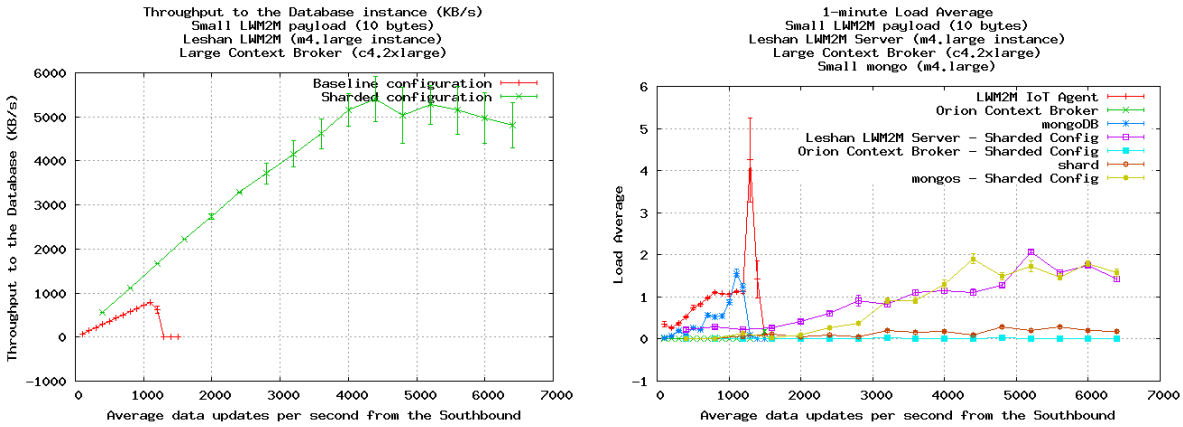
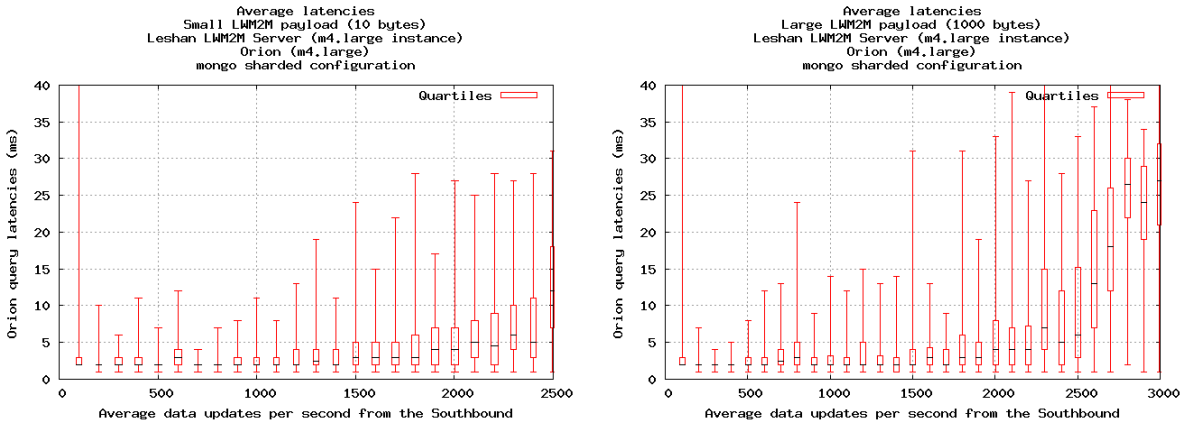


Figure 52. Driving the throughput further using a *c4.2xlarge* Context Broker, increasing load on the Leshan server.

4.3.2.2 Latency

Latencies with the sharded cluster for the database and Leshan LWM2M server stay under 40 milliseconds. The larger payload has more variability and in the response times and also a higher median for higher rates of requests, but also stays under 40 milliseconds in our tests. (figure 53)



The sharded cluster provides the best performance.

Figure 53. Latencies for entity queries to the Context Broker, using a configuration with two mongoDB shards, for both evaluated payload sizes (10 and 1000 bytes)

4.3.3 Summary

Scaling Orion Context Broker horizontally proved not to be an effective strategy for handling more load in the southbound interface. On the other hand, using a sharded cluster for mongoDB proved to be the most effective strategy, which confirms the previous observation that mongoDB needed more computational resources to handle the high rate of updates. Replacing FIWARE's IoT Agent was necessary since it crashes even when the database layer is not overloaded.

4.4 Discussion

Our results show that horizontally scaling the database is the most effective strategy for increasing the performance of the system, when handling thousands of data updates per second from the southbound interfaces. The implementation of FIWARE's IoT Agents should be fixed to gracefully handle an overloaded state, or replaced by a custom device management layer. We begin this discussion by reflecting on our methodology and comparing it to previous testing efforts. We then proceed to discuss our results in light of their practical application in real deployments, considering costs and the need to test application-specific characteristics.

Previous load tests performed by FIWARE, as part of their non-functional quality assurance process, have focused on different components of FIWARE's architecture, which includes the Internet of Things Services Enablement Architecture and other Generic Enablers related to other purposes.

As part of the testing for Orion Context Broker, FIWARE performed some load tests using Apache JMeter, which they describe as stress tests. The tests are centered on an application-level perspective of Orion's NGSI APIs, covering functionalitites like updates, queries, notifications and subscriptions in the different versions of the API provided by Orion.

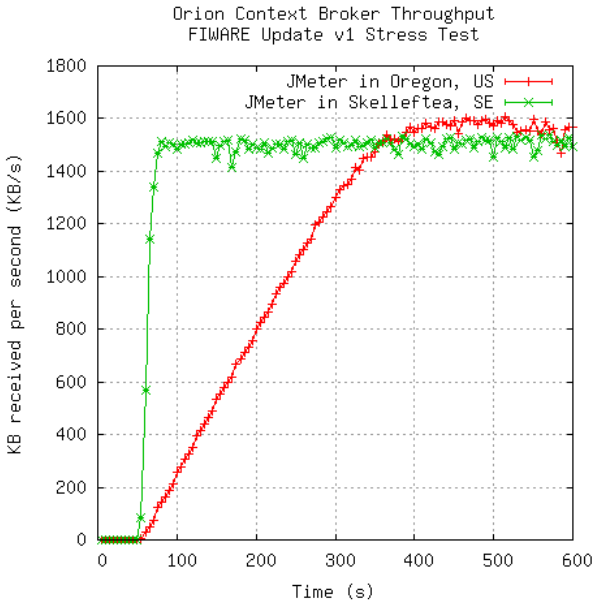
In FIWARE tests, blocking HTTP requests are generated as the load on the system. In our testbed and methodology, we have introduced asynchronous requests and important IoT protocols, namely the CoAP-based LWM2M and MQTT.

We would like to point out some issues that can arise concerning the methodology for testing IoT systems. When using a thread-based testing tool like JMeter, it is important to be aware that that no more outstanding requests than the number of active threads can be issued at any point in time to the system under test. The threads that generate requests are also throttled due to 1) the tranmission delay and 2) the time spent blocked while the system under test services the request.

To illustrate the first point, we repeated FIWARE's load test deploying the Context Broker in Amazon Web Services datacenter in Frankfurt and running the load generator in a virtual machine in the same datacenter and one in the Oregon datacenter (figure 54). FIWARE's test gradually spawns more threads to generate load. In the figure it can be seen how the throughput to the database almost instantly reaches its maximum when running from the same datacenter with just the first threads. In contrast, the throughput when running the load generator from Oregon needs many more threads to reach the maximum. The threads are limited to issue one request every round trip time, which explains the difference since the latencies from the same datacenter are a few millisecond while from Oregon it can be over a hundred milliseconds. The number of threads needed to reach the maximum rate supported by the platform can change depending on the latency to the system under test and other configurations, and experimenters should be careful to verify this.

From the perspective of maximizing the generated load, running the load generator close to the System Under Test makes sense, but more care should be taken when wanting to simulate the traffic of particular scenarios for capacity planning. The testing approach with JMeter usually models each user using a thread, and realistic rates of requests issued will be affected by the round trip time and the think time in case of blocking clients.

One observation that we have related to previous tests is that reported throughputs change widely depending on the database configuration used when using asynchronous requests. Previous tests did not experiment with different database setups, including either scaling vertically or implementing a sharded cluster for the database. Performance tests of Orion Context Broker



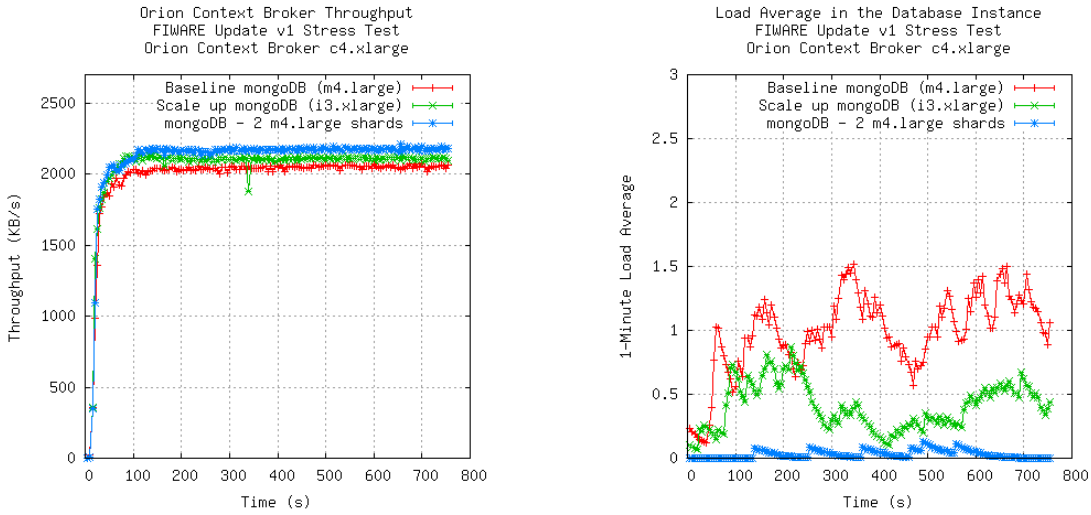
Latency to the system under test has an effect on the generated traffic and the response of the system due to the use of blocking clients.

Figure 54. FIWARE update stress test run from different locations

performed by FIWARE report throughputs and number of requests in update scenarios, but we have shown that these change with the choice of the database instance and the testing methodology.

In figure 55 we also ran some of FIWARE's tests using different configurations, and found that throughput can change although not as much as with the asynchronous tests. The reported utilization of resources in FIWARE tests is lower, which is worth investigating. If resource utilization is low and throughput can change with database configurations, it is not the limit of the Context Broker what is being tested, but rather the database or other unknown bottlenecks. It is not a proper stress test if the components are under low load. An evaluation of FIWARE for usage in the field of precision agriculture [51] concludes that scalability in the number of entities should focus on the database, while scaling the operation requests to manage entities should focus on adding additional Orion Context Broker instances behind a load balancer. However, according to our results scaling the database is important to be able to increase the number of operation requests that can be processed by Orion. In the context of an IoT scenario, this also means that the IoT Agent layer should work correctly or be replaced by a more robust implementation that can handle high load.

Our results can also be considered from a financial viewpoint, considering the cost of running the configurations to support certain level of load. In table 18, we compare the monthly cost of



For FIWARE's update stress tests, throughputs for Orion change slightly with different database configurations . However, this is not the full potential of the system. The load applied to the system is reaching a stable state but not pushing it to its limits.

Figure 55. FIWARE update stress test with different database instance capacities

every configuration and calculate its efficiency by dividing the peak rate of requests from the southbound that could be handled in the configuration by the monthly cost of running them in the *eu-central-1* region of Amazon Web Services.

Table 18. Cost comparison

Strategy	No	Context Broker	mongoDB	Monthly Cost (USD)	Cost efficiency
Baseline	1	m4.large	m4.large	188.86	5.29
Scale up	2	m4.large	i3.large	230.59	4.34
	3	m4.large	i3.large	366.74	2.73
	4	c4.xlarge	i3.xlarge	260.6	4.22
	5	c4.xlarge	i3.large	302.33	3.64
	6	c4.xlarge	i3.large	438.48	2.74
	7	c4.2xlarge	m4.large	426.76	2.58
	8	c4.2xlarge	i3.large	468.49	2.56
	9	c4.2xlarge	i3.xlarge	604.64	2.15
Scale out - Orion	10	m4.large x 2	m4.large	283.29	3.88
Scale out - mongoDB	11	c4.xlarge	m4.large x 3	449.46	5.78

Our results show that scaling up is not a cost effective alternative for supporting a larger rate of data updates from the southbound interfaces. The most efficient alternatives are the base-

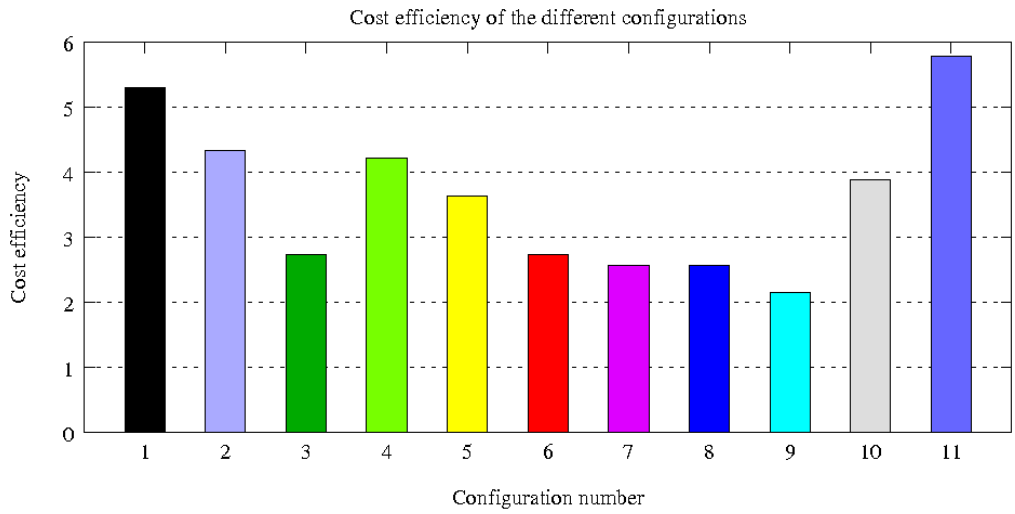


Figure 56. Cost efficiency of the evaluated configurations.

line configuration and the configuration with a sharded mongoDB cluster, with cost efficiencies of 5.29 and 5.78, respectively. The baseline configuration would be the best choice for deployments that don't need to process thousands of data updates per second. For deployments that need to scale, a sharded mongodb cluster should be the first step to increase the supported throughput. Whenever the Context Broker becomes a bottleneck, it is possible to use load balancing to scale out. For FIWARE's IoT Agents, no information is provided about its capacity to scale horizontally. Assuming that the existing stability issues are fixed, this layer will also need to scale. For the Leshan library, a clustered mode is currently under development to serve scenarios with high levels of load.

Future testbeds and testing tools should take into account the characteristics of the clients that interact with the sytem under test, and also the interactions of the components inside the system. For instance, concrete deployments can have clients that interact with the Context Broker through asynchronous requests (like the IoT Agents) and others that issue synchronous requests. Similarly for the IoT Agents, concrete scenarios can have blocking or non-blocking requests, an issue that has also been highlighted previously in the evaluation of FIWARE for usage in precision agriculture applications [51]. Moreover, particular applications can have dataset sizes and schema characteristics that further affect the performance of the dabase. We have contributed our approach and a software tool for generating asynchronous LWM2M requests, which complements and adds to the already existing approach used in FIWARE testing, and enables more scenarios to be tested.

5 Conclusions

This thesis provides an evaluation of the performance of the IoT agents and Orion Context Broker components that are defined in FIWARE's Internet of Things Services Enablement Architecture. This performance evaluation considered the achieved throughput, utilization of computational resources and latencies for entity queries to the northbound interface. Our performance evaluations also considered the different strategies that can be used for scaling the components of the platform and its overall functionalities in the cloud, to accomodate the data that is actively sent from a fastly increasing number of devices in the Internet of Things.

Our work increases the comprehensiveness of the performance evaluation of the studied components, in comparison to previous efforts [15][51]. First, our testbed design supports the MQTT and LWM2M IoT protocols, which to the best of our knowledge had not been tested previously. Second, our distributed generation of asynchronous requests created the necessary stress conditions to observe the behavior and limitations of the studied FIWARE components under load. Methodological limitations in previous work did not allow these limits to be observed. Finally, our work considered the capacity of the components to handle more load through vertical or horizontal scaling, giving a broader characterization of the capabilities and the potential of the platform to serve large-scale scenarios, in comparison to previous work.

Our results enable us to makek recommendations for implementers of FIWARE in smart cities or other environments. For lower levels of load (under 1000 data update requests per second, for all our studied configurations) scaling vertically is not a cost-effective strategy. Scaling Orion horizontally behind a load balancer also did not support considerably more requests per second. According to our results, implementers of smart applications for smart cities can rely on our baseline configuration for low levels of load. If there is a need to scale robustly, efforts should be focused on deploying an appropiate sharded cluster for the database. At the same time, FIWARE's implementation of IoT agents needs to be improved to support the load of current and future large-scale scenarios for smart cities. We pointed out its abnormal usage of memory and TCP connections under high load, and the architectural limitations of using a single core for processing.

Our work provides other contributions that can be used beyond FIWARE. The testbed that we have constructed allowed us to generate data updates, like sensor values or other information sent actively by devices in the Internet of Things, simulating the large rates of active updates in large-scale deployments of the Internet of Things. This setup can be used to test platforms other than FIWARE. The emulation of Internet of Things devices and data updates for functional and non-functional testing has been tackled partially by some of the projects described in section 2.9,

and the mock LWM2M library and the plugin for JMeter that we describe in our testbed are a contribution for this effort. We have provided a library and a tool written in Java to perform large scale testing of IoT systems that use the LWM2M standard [64]. This contribution addressed the lack of a dedicated testing tool for platforms that use this technology.

Finally, through the lessons learned about the way to deploy components of the FIWARE platform to support more load, this work helps to bring the IoT vision for smart cities and its sustainability benefits one step closer to reality. The results about the usage of computational resources under load can be used by implementers to serve given levels of load using the most economical configurations, reducing costs and the number of active computing instances. This can save not only money but also energy, depending on the characteristics of every particular deployment. The results of this work can also be used by developers to optimize the current implementations and fix the components that act unstable under load.

We finish by outlining some areas for future research. New applications and techniques will need to be developed to accomodate new requirements for testing and optimizing Internet of Things platforms. Existing load testing tools and workload generators can also evolve to accomodate some of these requirements. Some of these tools have been used successfully for web systems or databases, and have been a driver for the scale and robustness of contemporary web systems. The importance of the storage layer for the overall performance of IoT platforms suggests that it can be a fruitful area for improving performance. There is already growing interest for the generation of realistic datasets and synthetic workloads, to be used in the evaluation of algorithms, tools and the development of benchmarks for applications in the Internet of Things. An appropiate load testing process can uncover performance bottlenecks in IoT systems, and contribute to achieve the vision of the Internet of Things with millions of connected devices. At the same time, it is necessary to increase the observability of performance and traffic characteristics within existing IoT platforms, to give researchers the necessary data and tools to study the performance of these systems in depth.

Beyond the application level and the deployment in computing clouds, performance will also need to be analyzed at the network and transport levels. This is a concern for network operators and service providers, and research into the impact of the IoT has already begun (including technologies like 5G). At the same time, performance beyond computational measures can be taken into account for the ensemble of these systems. In the context of the PERCCOM program [70], the sustainability performance can be considered in more depth, which might involve life cycle analyses of the techologies deployed for smart cities, along with precise estimations of their impact.

REFERENCES

[1] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys & Tutorials*, 17(4):2347–2376, 2015.

[2] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[3] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of things for smart cities. *IEEE Internet of Things journal*, 1(1):22–32, 2014.

[4] Renata Paola Dameri. Searching for smart city definition: a comprehensive proposal. *International Journal of Computers & Technology*, 11(5):2544–2551, 2013.

[5] B Bowerman, J Braverman, J Taylor, H Todosow, and U Von Wimmersperg. The vision of a smart city. In *2nd International Life Extension Technology Workshop, Paris*, volume 28, 2000.

[6] Corinna Morandi, Andrea Rolando, and Stefano Di Vita. *From Smart City to Smart Region: Digital Services for an Internet of Places*. Springer, 2015.

[7] Sense Smart Region. Sense smart region. Available at: `http://en.sensesmartregion.se/`.

[8] Li Da Xu, Wu He, and Shancang Li. Internet of things in industries: A survey. *IEEE Transactions on industrial informatics*, 10(4):2233–2243, 2014.

[9] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[10] Manuel Dıaz, Cristian Martın, and Bartolom Rubio. State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing. *Journal of Network and Computer Applications*, 67:99 – 117, 2016.

[11] André B Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, pages 195–203. ACM, 2000.

[12] Ngo Manh Khoi, Saguna Saguna, Karan Mitra, and Christer Ç^{o}hlund. Irehmo: an efficient iot-based remote health monitoring system for smart regions. In *E-health Networking, Application & Services (HealthCom), 2015 17th International Conference on*, pages 563–568. IEEE, 2015.

[13] Upkar Varshney. Pervasive healthcare and wireless health monitoring. *Mob. Netw. Appl.*, 12(2-3):113–127, March 2007.

[14] FIWARE. Fiware community. Available at: `https://www.fiware.org/about-us/`, 2017.

[15] FIWARE. Fiware qa activities - fiware forge wiki. Available at: `https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE_QA_Chapter`, 2017.

[16] FIWARE. Fiware foundation. Available at: `https://www.fiware.org/foundation/`, 2017.

[17] Mohammad Abdur Razzaque, Marija Milojevic-Jevric, Andrei Palade, and Siobhán Clarke. Middleware for internet of things: a survey. *IEEE Internet of Things Journal*, 3(1):70–95, 2016.

[18] FIWARE. Internet of things (iot) services enablement architecture r5, 2016.

[19] David J Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge university press, 2005.

[20] Jeremy L Caradonna. *Sustainability: A history*. Oxford University Press, 2014.

[21] Graham Hubbard. Measuring organizational performance: beyond the triple bottom line. *Business strategy and the environment*, 18(3):177–191, 2009.

[22] John Elkington. Enter the triple bottom line. *The triple bottom line: Does it all add up*, 11(12):1–16, 2004.

[23] Frans Berkhout and Julia Hertin. Impacts of information and communication technologies on environmental sustainability: Speculations and evidence. 2001.

[24] Molly Webb et al. Smart 2020: Enabling the low carbon economy in the information age. *The Climate Group. London*, 1(1):1–1, 2008.

[25] Shancang Li, Li Da Xu, and Shanshan Zhao. The internet of things: a survey. *Information Systems Frontiers*, 17(2):243–259, 2015.

[26] Ghofrane Fersi. Middleware for internet of things: a study. In *Distributed Computing in Sensor Systems (DCOSS), 2015 International Conference on*, pages 230–235. IEEE, 2015.

[27] Hasan Derhamy, Jens Eliasson, Jerker Delsing, and Peter Priller. A survey of commercial frameworks for the internet of things. In *Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on*, pages 1–8. IEEE, 2015.

[28] Srdjan Krco, Boris Pokric, and Francois Carrez. Designing iot architecture (s): A european perspective. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pages 79–84. IEEE, 2014.

[29] FIWARE. Fiware open specification iot backend device management. Available at: `http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE.OpenSpecification.IoT.Backend.DeviceManagement_R5`, 2016.

[30] FIWARE. Backend device management - idas. Available at: `https://catalogue.fiware.org/enablers/backend-device-management-idas`, 2016.

[31] Zach Shelby, Klaus Hartke, and Carsten Bormann. The constrained application protocol coap. RFC 7252, IETF, 2014.

[32] C Borman, S Lemay, V Solorzano Barboza, and H Tschofenig. A tcp and tls transport for constrained application protocol (coap): drafttschofenig-core-coap-tcp-tls-05.

[33] Open Mobile Alliance. Lwm2m specification 1.0. *URL: http://technical. openmobilealliance. org/Technical/technicalinformation/release-program/current-releases*.

[34] OASIS. Mqtt 3.1. 1. edited by andrew banks and rahul gupta. 29 october 2014. oasis standard. Available at: `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`, 2014.

[35] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context aware computing for the internet of things: A survey. *IEEE Communications Surveys & Tutorials*, 16(1):414–454, 2014.

[36] Satish Narayana Srirama. Mobile web and cloud services enabling internet of things. *CSI Transactions on ICT*, 5(1):109–117, 2017.

[37] Darrell M West. E-government and the transformation of service delivery and citizen attitudes. *Public administration review*, 64(1):15–27, 2004.

[38] Hug March and Ramon Ribera-Fumaz. Smart contradictions: The politics of making barcelona a self-sufficient city. *European Urban and Regional Studies*, 23(4):816–830, 2016.

[39] Luis Sanchez, José Antonio Galache, Veronica Gutierrez, Jose Manuel Hernandez, Jesús Bernat, Alex Gluhak, and Tomás Garcia. Smartsantander: The meeting point between future internet research and experimentation and the smart cities. In *Future Network & Mobile Summit (FutureNetw), 2011*, pages 1–8. IEEE, 2011.

[40] Luis Sanchez, Luis Munoz, Jose Antonio Galache, Pablo Sotres, Juan R. Santana, Veronica Gutierrez, Rajiv Ramdhany, Alex Gluhak, Srdjan Krco, Evangelos Theodoridis, and Dennis Pfisterer. Smartsantander: Iot experimentation over a smart city testbed. *Computer Networks*, 61:217 – 238, 2014. Special issue on Future Internet Testbeds Part I.

[41] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. Pearson Education, 2013.

[42] Maged Michael, Jose E Moreira, Doron Shiloach, and Robert W Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[43] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7), 2008.

[44] Neil Gunther, Paul Puglia, and Kristofer Tomasette. Hadoop superlinear scalability. *Queue*, 13(5):20:20–20:42, May 2015.

[45] Alexander Gluhak, Srdjan Krco, Michele Nati, Dennis Pfisterer, Nathalie Mitton, and Tahiry Razafindralambo. A survey on facilities for experimental internet of things research. *IEEE Communications Magazine*, 49(11), 2011.

[46] B. Cheng, S. Longo, F. Cirillo, M. Bauer, and E. Kovacs. Building a big data platform for smart cities: Experience and lessons from santander. In *2015 IEEE International Congress on Big Data*, pages 592–599, June 2015.

[47] Nadir Javed and Bilhanan Silverajan. *Connectivity Emulation Testbed for IoT Devices and Networks*, pages 146–155. Springer International Publishing, Cham, 2014.

[48] Google. Distributed load testing using kubernetes. Available at: `https://cloud.google.com/solutions/distributed-load-testing-using-kubernetes`.

[49] Alli Mäkinen et al. Emulation of iot devices. Master's thesis, 2016.

[50] V. Looga, Z. Ou, Y. Deng, and A. Yla-Jaaski. Mammoth: A massive-scale emulation platform for internet of things. In *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*, volume 03, pages 1235–1239, Oct 2012.

[51] Ramón Martínez, Juan Ángel Pastor, Bárbara Álvarez, and Andrés Iborra. A testbed to evaluate the fiware-based iot platform in the domain of precision agriculture. *Sensors*, 16(11):1979, 2016.

[52] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: A cautionary tale. In *NSDI*, volume 6, pages 18–18, 2006.

[53] Raoufehsadat Hashemian, Diwakar Krishnamurthy, and Martin Arlitt. Web workload generation challenges–an empirical investigation. *Software: Practice and Experience*, 42(5):629–647, 2012.

[54] Allen Davis Malony. *Performance Observability*. PhD thesis, Champaign, IL, USA, 1990. AAI9114332.

[55] S. Li, L. D. Xu, and X. Wang. Compressed sensing signal and data acquisition in wireless sensor networks and internet of things. *IEEE Transactions on Industrial Informatics*, 9(4):2177–2186, Nov 2013.

[56] Subhasri Duttagupta, Mukund Kumar, Ritesh Ranjan, and Manoj Nambiar. Performance prediction of iot application: An experimental analysis. In *Proceedings of the 6th International Conference on the Internet of Things*, IoT'16, pages 43–51, New York, NY, USA, 2016. ACM.

[57] J. W. Anderson, K. E. Kennedy, L. B. Ngo, A. Luckow, and A. W. Apon. Synthetic data generation for the internet of things. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 171–176, Oct 2014.

[58] Jaime Jimenez, Michael Koster, and Hannes Tschofenig. Ipso smart objects. In *Position paper for the IOT Semantic Interoperability Workshop*, 2016.

[59] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. Riotbench: A real-time iot benchmark for distributed stream processing platforms. *arXiv preprint arXiv:1701.08530*, 2017.

[60] Hemika Kodikara. mqtt-jmeter. Available at: `https://github.com/hemikak/mqtt-jmeter`, 2016.

[61] Nicolas Niclausse. Tsung. Available at: `http://tsung.erlang-projects.org/`, 2013.

[62] Apache Software Foundation. Apache jmeter. Available at: `http://jmeter.apache.org/`, 2017.

[63] Matthias Kovatsch, Martin Lanter, and Zach Shelby. Californium: Scalable cloud services for the internet of things with coap. In *Internet of Things (IOT), 2014 International Conference on the*, pages 1–6. IEEE, 2014.

[64] Victor Araujo. Lighthweightm2m plugin for jmeter. Available at: https://github.com/vears91/lwm2m-jmeter.

[65] Prometheus Authors. Prometheus - monitoring system & time series database. Available at: https://prometheus.io/, 2017.

[66] Node.js Foundation. The node.js event loop, timers, and process.nexttick() | node.js. Available at: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/, 2017.

[67] mongoDB. Sharded cluster components mongodb manual 3.4. Available at: https://docs.mongodb.com/manual/core/sharded-cluster-components/, 2017.

[68] Amazon. Elastic load balancing. Available at: https://aws.amazon.com/elasticloadbalancing/, 2017.

[69] Amazon. How elastic load balancing works. Available at: http://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/how-elastic-load-balancing-works.html, 2017.

[70] Alexandra Klimova, Eric Rondeau, Karl Andersson, Jari Porras, Andrei Rybin, and Arkady Zaslavsky. An international master's program in green ict as a contribution to sustainable development. *Journal of Cleaner Production*, 135:223–239, 2016.