TÉCNICO LISBOA

Cloud4Things

Using Cloud infrastructure to support Smart Place applications

Marcus Vinícius Paulino Gomes

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Advisor: Prof. Miguel Filipe Leitão Pardal
Co-Advisor: Prof. José Manuel da Costa Alves Marques

Examination Committee

Chairperson: Prof. José Carlos Alves Pereira Monteiro
Advisor: Prof. Miguel Filipe Leitão Pardal
Member of the Committee: Prof. Luís Manuel Antunes Veiga

November 2015

Acknowledgments

I would like to thank my advisors Prof. Miguel Pardal and Prof. José Alves Marques that helped me in the development of this Master Thesis. In special, I want to thank Prof. Miguel Pardal for the given support throughout this year and specially in the writing of this document. I also would like to thank the members of committee that evaluated this work and provided feedback.

Finally, I would like to thank my mother for supporting me during these last 5 years and my friends and colleagues that supported me throughout my academic path.

Abstract

Smart places are systems composed of sensors, actuators and computing infrastructure that acquires data about the surrounding environment and use that data to improve the experience of the people interacting with the place. The smart place runs an Internet of Things (IoT) application that transforms raw sensor data into informed action. For instance, RFID readers can detect a tagged object approaching and an automatic door is opened after the event is processed in a dedicated server.

Usually, IoT applications are latency-sensitive because actions need to be done in a timely manner. To meet this requirement these applications are usually provisioned close to the physical place, which represents an infrastructure burden because it is not always practical to deploy a physical server at a location. Utility Computing in the Cloud can solve this issue. However, the latency requirements must be carefully assessed. Fog Computing is a recent concept that brings the cloud close to the “ground” - i.e close to devices at the edge of the network -, aiming to provide low latency communication for applications and services.

The present work implemented an automatic provisioning mechanism to deploy IoT applications according in an Utility Computing platform. Our demonstration scenario is an automated warehouse that uses a RFID event processing software to track objects in the facilities. We compared the event latency performance of both approaches and data storage performance.

The results confirm that a fog-based approach is more adequate for latency-sensitive applications, presenting a better performance when compared with a cloud-based approach.

Keywords: Internet of Things, Cloud computing, Fog computing, Application Deployment, RFID, Fosstrak Platform

Resumo

Smart places são sistemas compostos de sensores, actuadores e infra-estrutura computacional que adiquirem dados do ambiente circundante e usam estes dados para melhorar a experiência das pessoas que interagem com este ambiente. O *smart place* possui uma aplicação para a Internet das Coisas (IoT) que é capaz de transformar estes dados em informação. Por exemplo, leitores RFID podem detectar um objecto que está a aproximar-se e uma porta automática é aberta após o evento ser processado.

Geralmente, aplicações IoT são *latency-sensitive* visto que as acções tem de ser rapidamente executadas, o que faz com que a infra-estrutura necessária para aprovisionar a aplicação tem de ser local, o que muitas vezes é ineficiente e dispendioso. A *Utility Computing* na nuvem pode resolver este problema. Entretanto, os requisitos de latência devem ser cuidadosamente avaliados. A Computação em Nevoeiro é um conceito recente que aproxima a nuvem e os dispositivos que encontram-se na periferia da rede, fornecendo comunicação de baixa latência para aplicações e serviços.

Neste trabalho foi implementado um mecanismo para automatizar o *deployment* de aplicações IoT basedas na nuvem e no nevoeiro. Nosso cenário é um armazém automatizado que utiliza um *software* para o processamento de eventos RFID para rastrear objectos nas instalações. Nós comparamos a performance da latência dos eventos para ambas as abordagens e a performance do armazenamento de dados.

Os resultados obtidos confirmam que a abordagem baseada em nevoeiro é mais adequada para aplicações *latency-sensitive*, apresentando uma melhor performance quando comparada com a abordagem em nuvem.

Palavras-Chave: Internet das Coisas, Computação em Nuvem, Computação em Nevoeiro, *Deployment* de Aplicações, Plataforma Fosstrak

Contents

List of Tables

List of Figures

List of Source Codes

Acronyms

ADSL Asymmetric Digital Subscriber Line. 24, 26, 30

ALE Application Level Events. 11, 12, 27, 28, 30–35, 39–42

AMI Amazon Machine Image. 29

AWS Amazon Web Service. 8, 21, 29, 42

CM Configuration Management. 8, 15, 16

CPU Computer Processing Unit. 6, 36, 40

EC2 Elastic Cloud Computing. 29, 42

EPC Electronic Product Code. 3, 10, 12

EPCIS Electronic Product Code Information System. 11, 13, 17, 21, 24, 25, 29, 30, 40, 42

ERP Enterprise Resource Planning. 10

FCServer Filtering & Collection Server. 21, 24, 25, 30

GB Gigabyte. 22, 29

GHz Gigahertz. 29

HAL Hardware Abstraction Layer. 12

HTTP Hypertext Transfer Protocol. 26, 29

IaaS Infrastructure as a Service. 5, 6

IoT Internet of Things. 1, 2, 6, 10, 27, 41, 43

JSON JavaScript Object Notation. 17

LLRP Low Level Reader Protocol. 11, 12, 24

LTE Long-term Evolution. 24, 26

LXC Linux Container. 6

M2M machine-to-machine. 9

MB Megabyte. 36, 37, 39

NoSQL	Not Only SQL. 7
OS	operating system. 6
PaaS	Platform as a Service. 5, 6
PIQL	Performance Insightful Query Language. 7
RAM	Random-access memory. 29
RDBMS	Relational Database Management System. 7
RFID	Radio-Frequency IDentification. 1–4, 6, 10–12, 24–28, 30, 35, 39, 41–43
SaaS	Software as a Service. 5, 6
SDK	Software Development Kit. 21
SOA	Service Oriented Architecture. 8
ubicomp	Ubiquitous Computing. 1
VM	Virtual Machine. 5, 6, 8, 17, 21, 22, 25, 26, 29, 42
WSN	Wireless Sensor Network. 10

Chapter 1

Introduction

In recent years, computing is becoming more ubiquitous in the physical world. The term Ubiquitous Computing (ubicomp) was introduced many years ago by Mark Weiser [1]. In this vision computational elements are embedded seamlessly in ordinary objects that are connected through a continuously available network. Technology advances such as the mobile Internet contributes to achieve this vision [2], as well as the Internet of Things (IoT), a system composed of physical items that are continuously connected to the virtual world and can act as physical access points to Internet Services [3]. However, there are some challenges that must be addressed in order to make these ubicomp systems truly ubiquitous [4] such as data, context awareness and infrastructure. An important concern regards ubiquitous data: *Where it is located?*, *Who can access it?* and *How much time should this data persist?* Also, ubiquitous systems are constantly interacting with the surrounding environment. Thus, these systems need to understand the context in which they are inserted and also to adapt to the changes that occur in this environment. Another import concern regards about the infrastructure burden of the ubiquitous systems. Many times these systems requires low-latency interaction with users and environments, which implies that at least part of an ubicomp system needs to be tightly bound to the local infrastructure of the interacting environment. This requirement for local infrastructure can be a barrier in the adoption of ubiquitous systems in a large-scale perspective. The IoT and the Utility Computing in the cloud paradigms can help to solve those issues.

On one hand, the Utility Computing in the cloud provides the illusion of infinite computing resources available on demand to the public users [5]. This paradigm can help to reduce the infrastructure burden of ubiquitous systems, while providing important features such as high availability and high scalability. The Utility computing provides data centralization, allowing users to store data during long periods of time and to define the access policies for this data. On the other hand, the IoT aims to solve a key problem in wider adoption of ubiquitous systems, the tight coupling with a particular embedded infrastructure. With the IoT a variety of *objects* or *things* - such as Radio-Frequency IDentification (RFID) tags, sensors, actuators, etc. - will be able to interact with each other and cooperate with the surrounding *things* to reach common goals [6]. Furthermore, IoT middleware solutions are able to processing the raw data collected from these *things* in order to understand the context where they are inserted.

1.1 Application Domains

The Internet of Things offers a great potential that makes possible the development of a huge number of applications. There are several environments and domains where IoT applications are expected to improve the quality of life of the people that live and work in these environments and also will provide competitive advantage against current solutions. Currently, some of the application domains that promise to play a big role in the adoption of IoT are:

Ambient Intelligence. Environments that use the intelligence of the objects within themselves to become a more comfortable and efficient environment. For instance, a room that has its temperature adapted according to the weather, an office that has its lights adapted according the time of the day and an automated industrial plant where its is possible to monitor the production progress.

Logistics. Real-time information processing based on technologies such as RFID allow the monitoring of almost every phase of the supply chain, ranging from raw material purchasing, transportation, storage, distribution and after-sales services.

Transportation. From personal vehicles to public transportation, mobile ticketing and transportation of goods. Cars, trains and buses are equipped with sensors, actuators and computational power that are able to provide information about the status of the vehicle, improve the navigation and even perform collision avoidance. Regarding the transportation of goods, it is possible to monitor the conservation status of perishable goods - temperature, humidity, etc. - during its transportation.

Healthcare RFID tags can be used to monitor the position of patients, hospital staff and also to control the inventory of materials. Sensors can be used to monitor patient conditions, hospital environment conditions - temperature, air quality, etc.

Beyond the presented examples, the IoT field covers many other domains. To give a more unified view over those domains, we propose the term *smart places*, that can be defined as a system composed of sensors - e.g. RFID - actuators - e.g. automatic doors - and computational infrastructure - e.g. servers - that are able to acquire data about the surrounding environment and use that data to improve the experience of the people interacting with the place.

1.2 Smart Places Challenges

Challenges [4] for the construction of smart places that resulted from leveraging part of the smart place infrastructure to the cloud:

- **Data** is continuously generated by the *things* that compose the system. These data must be stored, processed and presented in a seamless and efficient way. Several middleware solutions for smart places [7][8][9] relies on database management systems that are very efficient, but will those systems able to handle with the volume of data generated by smart place applications?
- **Deployment** of smart places usually is performed in an isolated and vertical manner where hardware, middleware and application logics are tightly coupled. This provisioning model presents some limitations that makes the deployment of a smart place an inefficient process, since that

each time that a smart place is deployed its software components and embedded devices need to be manually installed and configured. Therefore, new provisioning approaches need to be adopted in order to make the deployment of smart places more efficient and scalable.

- **Low-latency Interaction** is a key requirement that requires that both network access latency and data transmission latency be reduced. However, in a cloud-based deployment, part of the system's infrastructure is moved to the cloud - for instance the middleware layer, which is responsible for processing the information and take decisions based on the results. However, will a cloud-based deployment able to meet the low-latency requirements of many smart place systems?

- **Management** of smart places is an issue that must be carefully addressed. It has been shown that end-users are unable to manage their own personal computers systems well [10] and there is no reason to believe that they will had a better performance at managing a much more complex system. It is reasonable to assume that service providers will perform the management of the services . These managed services introduces new questions that must be answered. For instance, who will pay for this services and who will control this services?

- **Cost vs. Performance** of smart placeas infrastructure - i.e. hardware, software, maintenance and energy costs - is an important issue that can determine if the cloud platform is the most adequate to support smart place instead of a local infrastructure. By leveraging the smart place infrastructure to the cloud the cost of its infrastructure can be reduced in a significant amount. However, will be the performance of the smart place satisfactory when its infrastructure is leveraged to the cloud?

1.3 Objectives

The objective of this work is to determine if the cloud platform is able fulfill the fundamental requirements of smart places. In this work we focused to determining if a cloud-based deployment can meet the low-latency interaction and data storage performance requirements. Since smart places presents different requirements according its domain, our efforts will be engaged in determine if the cloud platform is suitable to support a smart warehouse that relies on the RFID technology [11].

1.3.1 Example Domain

Our smart place example domain is an automated warehouse. In the warehouse, products are transported through automated guided vehicles. These products are tagged with RFID tags that can be identified by RFID readers. Through the data collected by these readers is possible to gather information about the smart place. For instance, is possible to determine which products enter or leave the warehouse. A complete example of a platform that allows the transformation of that data into information is Fosstrak[1], an open source RFID software that implements the Electronic Product Code (EPC) Network standards.

To accomplish our objectives, in this work we will follow two approaches and determine which of them is more adequate to deploy a smart warehouse based on the RFID technology. . The first, is a more traditional deployment approach, where all the application middleware is provisioned in the cloud. The second is a deployment approach based on the Fog Computing[12] paradigm, a virtualized platform that is located close to the smart place and provides network, computing and storage resources between the

[1] http://fosstrak.github.io/

embedded devices in the physical place and the traditional cloud.

Since the cloud platform offers flexibility to deploy applications, we propose a solution that automates the deployment of RFID application middleware in the cloud regarding the chosen approach: cloud or fog. Our solution consists in a provisioning mechanism that automates the application deployment according pre-defined provisioning policies and software images.

Our initial hypothesis is that a fog-based deployment will present a best overall performance and as consequence will be more adequate to deploy the smart warehouse RFID application middleware. However the data storage capabilities of fog are inferior and they will have to be validated whether they are enough.

1.4 Dissertation Outline

The remainder of this document is organized as follows:

- **Chapter 2. Background** summarizes the relevant work in the field and introduces some key concepts that support our work such as a description of the Fog Computing paradigm, the EPCGlobal Network and Fosstrak platform.
- **Chapter 3. Solution** presents the approaches adopted to deploy and provision the smart place software stack, namely the Fosstrak platform. We also details the implementation for the proposed solution: the provisioning strategies, the virtualization technologies and the deployment approaches.
- **Chapter 4. Evaluation** describes the experiments made to meet the defined objectives and presents an analysis of the obtained results.
- **Chapter 5. Conclusion** summarizes the presented work, presents the main conclusions, some important research points for future work and the main contributions that resulted from this dissertation.

Chapter 2

Background

This chapter presents the background necessary to understand the area of the problem to be solved. First section reviews the relevant related work to solve the challenges of converging the Internet of Things and Utility Computing. The following sections present a description of the concepts that composes the basis of our work.

2.1 Cloud computing concepts and tools

As mentioned in Section 1.2, the leverage of part of the smart place infrastructure to the cloud brought some challenges that must addressed. In the next sections, we present the most relevant work regarding to solve the smart place deployment in the cloud and data storage performance. Furthermore, we present a brief description of Docker, a lightweight virtualization platform in alternative to traditional Virtual Machines. Finally we describe Chef, a configuration management tool that automates the provision of smart applications in the cloud.

2.1.1 Smart Place Computing

Deployment of smart places usually was performed in a physical and isolated manner. By leveraging the infrastructure of smart places to the cloud, new provisioning approaches have to be adopted. Currently, cloud service delivery models are being developed based on the existing layers of the cloud architecture [13]:

- *Infrastructure as a Service (IaaS)* refers to the provisioning of infrastructure resources on-demand - e.g. Virtual Machines (VMs), storage and network.
- *Platform as a Service (PaaS)* refers to providing platform layer resources such as operating system support and software development frameworks.
- *Software as a Service (SaaS)* refers to providing on demand application over the Internet.

Soldatos et al. [14] presented the idea of converging the IoT and the utility computing in the cloud. The proposed architecture is the core concept of the OpenIoT Project[1]. The cloud is used at IaaS level, which allows to measure the utility of the services provided by inter-connected objects. Distefano et al. [15] proposed a conceptual architecture by mapping various elements in both cloud and IoT to the

[1] `http://openiot.eu`

three layers of the cloud architecture (IaaS, PaaS and SaaS). In this proposal IoT resources are provided voluntarily by their owners, while management functions - such as node management and policy enforcement - are viewed as peer functions of cloud infrastructure management. A PaaS module is responsible for mashup IoT and cloud infrastructure (IaaS) resources to applications, which are delivered to the clients through SaaS. CloudThings [16] is an architecture that uses the cloud platform layers to integrate Internet of Things and Cloud Computing. The proposed architecture is an online platform which accommodates IaaS, PaaS, SaaS and allows system integrators and solution providers to leverage the complete application infrastructure for developing, operating and composing applications and services. Li et. al [17] proposed IoT PaaS, a cloud platform that supports scalable IoT service delivery. Solution providers are able to deliver new solutions by leveraging computing resources and platform services - domain mediation, application context management, etc. - to the cloud. The proposed architecture aims to enable virtual vertical service delivery, for that it has a multi-tenant nature which is designed to help at the isolation of the environments of different solutions.

Although a significant progress was achieved regarding the improvement for the deployment of IoT solutions, most of the work still are in a conceptual stage. What is certain is that cloud service delivery models will be the basis for the service delivery models of IoT solutions.

2.1.1.1 Containers

Containers are a virtualization technique that is performed at the operating system (OS) level, different of hypervisor-based solutions - e.g. VMs - where the virtualization is performed at the hardware-level. In a OS level virtualization, all guests share the same operating system as the base machine [18]. Although the effect of both types of virtualization are similar, unlike the hypervisor-based virtualization an OS level virtualization does not provide the ability to run multiple VMs with different operating systems on the same physical machine. However, OS level virtualization provides significant benefits when compared to hypervisor-based solutions. Containers are small, they have low memory and Computer Processing Unit (CPU) overhead, they also are easy to port between different virtualization environments [19].

Linux Containers (LXCs)[2] is an OS virtualization environment that relies on Linux Kernel. LXC are hardware-agnostic and platform-agnostic, and also provides features such as application isolation and multi-tenancy. The present work will use containers to provisioning the RFID application software stack. Thus, we chose to use Docker as our container-based virtualization platform.

Docker Platform. Docker[3] is an open source project to pack, ship and run any application as a lightweight container. Since Docker containers are based on LXC, it can run anywhere[4], from a laptop to a cloud instance. Another benefit that Docker platform provides is the Docker Hub[5] service, a public repository that stores Docker images that are used to create the containers.

2.1.2 Data Storage

Smart places generate a large amount of data. The cloud must be able to store and process that data in an efficient manner. Currently, there are several alternatives to perform data storage in the cloud, from

[2]http://www.linuxcontainers.org

[3]https://www.docker.com/

[4]Initially, Docker required that the physical machine where the containers will be created is running a Linux kernel. Currently, Microsoft launched Windows Server Containers, an OS level virtualization mechanism where it is possible to perform the management of containers through Docker.

[5]https://hub.docker.com/

key-value stores to Relational Database Management System (RDBMS) clusters. In the next sections, we will summarize the most relevant work regarding these alternatives.

2.1.2.1 Key-value Stores

Since that data storage and retrieval in the cloud had specific requirements, cloud providers started to implement their own solutions.

Google Big Table, *Facebook Cassandra* and *Amazon Dynamo* [20] [21] [22] are key-value stores - Not Only SQL (NoSQL) databases - that have the ability to horizontally scale - i.e, distribute both data and load of simple operations through many servers - but it has a weaker consistency model than the ACID transactions of most RDBMSs systems [23].

Performance Insightful Query Language (PIQL) [24] is a SQL-like API built to run on top of existing performance predictable key-value stores, that provides many of the benefits of using a traditional RDBMS, such as the ability to express the queries in a declarative way, automatic data parallelism, physical data independence and automatic index selection and maintenance, all while maintaining the low-latency guarantees on application performance that come from the underlying key-value store.

2.1.2.2 Relational Database Clusters

Relational Database Management System are database management systems that store data in form of related tables. Unlike the key-value stores, RDBMS offers a complete pre-defined schema, a SQL interface and ACID transactions. Recently, some progress has been reached as regards the performance and scalability of these systems. Although most of the works still are in development, it is possible to highlight some solutions that are in a more mature state.

MySQL Cluster [25] is an in-memory clustered distributed RDBMS. Compared with the basic MySQL implementation it works by replacing the InnoDB engine with the NDB - a proprietary distributed layer from MySQL. MySQL Cluster is built on top of a shared-nothing[6] architecture and includes features such as failover, node recovery, synchronous data replication and no single point of failure. MySQL Cluster seems to be the solution that scales to more nodes than other RDBMS - 48 is the limit. However, it was reported that after scaling up to a few dozen nodes it starts to run with stability problems [26].

VoltDB [27] is a RDBMS designed for performance and scalability. VoltDB assumes a multi-node cluster architecture where the database tables are partitioned over multiple servers. To allow fast-acess of data, tables can be replicated over servers. Data recovery is supported through the replication of table rows (shards) across the cluster. The current implementation also supports database snapshots. Currently some features still are missing, but in its current implementation VoltDB already presents some features that improves the performance of SQL execution. As result, the number of nodes that are needed to support a given application load can be reduced in a significant way.

As shown there several solution to perform data storage in the cloud. Choose what is the best solution will depend of the application domain and its requirements.

[6]Shared-nothing architecture share neither data on disk or data in memory between the nodes in the cluster.

2.1.2.3 RFID Middleware Platforms

Gomes et. al [28] proposed a new infrastructure for EPCGlobal compliant middleware platforms in order to improve the performance of these middleware platforms. The Fosstrak platform was evaluated and the results show that Fosstrak is not the best option when executing the modules responsible for collecting the data (ALE) from RFID readers and storing the event data (EPCIS) in the same machine. In order to improve the performance of the EPCGlobal compliant middleware platforms, the new infrastructure adopts a decentralized architecture where the EPCIS is deployed in the cloud and it is connected to a NoSQL database instead of MySQL. The ALE module adopts a multi-threading architecture in order to support a high parallel demand of RFID readers. Currently, a prototype of the proposed infrastructure is being developed using OpenNebula, pthreads and Service Oriented Architecture (SOA).

2.1.3 Configuration Management Tools

Configuration Management (CM) tools are software management tools that allows to automate and specify the deployment of an application. Usually, users describe the system resources and their desired state and the CM tool is responsible for enforcing the desired state. For instance, CM tools allows the automation of the provisioning of physical and virtual machines, perform dependency management of software components and to perform the automation of management tasks.

Currently, there are several solutions to perform configuration management of software, where the most relevant are Chef[7], Puppet[8], Ansible[9] and Salt[10]. The main difference between these tools is that some of the them are more oriented to developers, which is the case of Chef and Puppet that requires some programming experience to be used, while others are more oriented to system administrators, which is the case of Ansible and Salt.

2.1.3.1 Chef

Chef is a configuration management tool that allows to describe the infrastructure as code. In that way it is possible to automate how the infrastructure is built, deployed and managed. Chef architecture is composed of the Chef Server - that stores the recipes and other configuration data - and the Chef Client - that is installed in each server, VM or container, i.e, the nodes that are managed with Chef. The Chef client periodically pulls Chef server latest policy and state of the network, and if anything on the node is out of date, the client update its state in order to be consistent with the latest policy.

The tool was built from the ground up with the cloud infrastructure in mind. With Chef, it is possible to dynamically provision and de-provision the application infrastructure on demand to keep up with peaks in usage and track. For instance, the *knife* command has a plugin for provisioning cloud resources across several cloud providers - Amazon Web Service (AWS)[11], Google Compute Engine[12] and Openstack[13].

[7]https://www.chef.io/
[8]https://puppetlabs.com/
[9]http://www.ansible.com/
[10]http://saltstack.com/
[11]https://aws.amazon.com/
[12]https://cloud.google.com/compute/
[13]https://www.openstack.org/

2.2 Fog computing for low latency responses

The Fog Computing [12] is a paradigm that aims to bring the cloud close to the “edge of the Network”. By bringing the cloud close to the ground - hence the fog analogy - the Fog will be able to meet the requirements of several applications that the traditional clouds are not able to accomplish. The most notable case is the Internet of Things, that requires mobility support, geo-distribution in addition to location awareness and low latency. The Fog aims to achieve that by virtualizing the computing, storage and network services between end devices and the traditional data centers in the cloud.

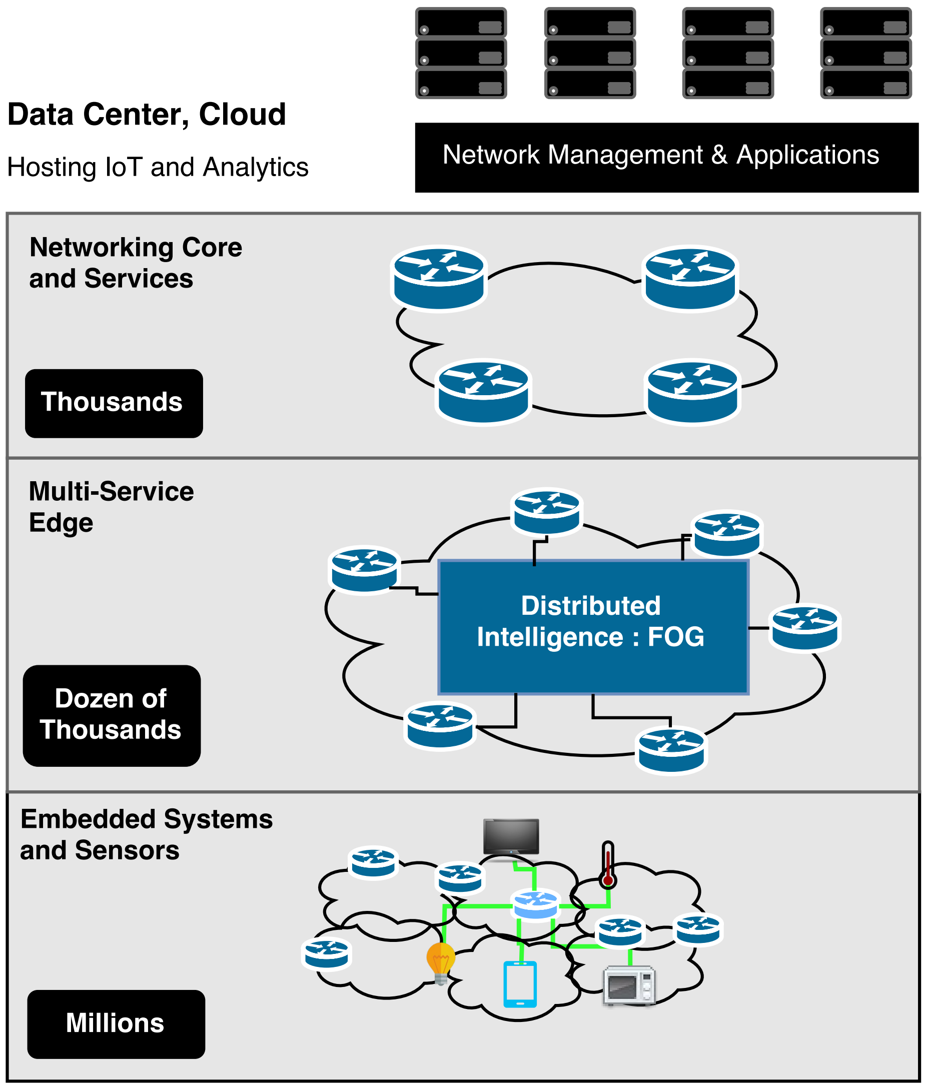

Data Center, Cloud
Hosting IoT and Analytics
Network Management & Applications
Networking Core and Services
Thousands
Multi-Service Edge
Distributed Intelligence : FOG
Dozen of Thousands
Embedded Systems and Sensors
Millions

Figure 2.1: The Internet of Things and Fog Computing (Bonomi et. al (2012)).

Bonomi et. al [12] presents the architecture of a Fog Computing platform. As illustrated in Figure 2.1 the distributed infrastructure of Fog is composed of heterogeneous resources that must be managed in a distributed way: the infrastructure comprising of several physical tiers, covering from data centers, core of the network, edge of the network and end devices.

Multi-Service Edge is the lowest tier of the Fog and it is responsible for performing machine-to-machine (M2M) interactions. It collects and process the data from the *Embedded Systems and Sensors* tier, issues commands to the actuators and also filters the data that is locally consumed and sent to the higher tiers. *Core Networking and Services* tier is responsible for providing network services that are used to exchange data between sub-networks. This tier also provides security services as well *QoS* and multicast services for the applications.

Since the interaction time between the different tiers can range from seconds - e.g. low-latency real-time analytics - to days - transactional analytics - the Fog must support several types of storage, from ephemeral storage at the lowest tiers to semi-permanent at the highest tier. It is important to point that the higher is the tier, the geographical coverage is wider and the time scale is larger [29]. The global coverage is given by the *Cloud* tier, which acts as a central repository for the persistent data and that is used to perform business analytics.

2.3 Internet of Things frameworks

Smart places cover several application domains, as mentioned in Section 1.1. Gubbi et. al [2] presented the technological characteristics for several domains of IoT applications, as illustrated in Table 2.1. The application domains are characterized according several aspects, including application network size, categorization of users, application bandwidth requirements and IoT devices.

The smart place applications usually uses RFID tags and Wireless Sensor Network (WSN) as IoT devices. The IoT devices are powered through rechargeable batteries - e.g. for applications deployed in physical spaces with easy access to the devices such as Offices and Warehouses - and/or through energy derived from external sources such as solar energy and wind energy - e.g. for applications that are deployed in large and/or remote physical spaces such as cities and forests. Usually, the IoT devices are connected to the Internet through a wireless connection such as 3G, 4G and Wi-fi. The data management can be performed through a local server or a shared server, in case of smart place application domains that are composed of multiple sub-domains - e.g. smart cities and smart transportation - and need to share data between those domains.

In the present work, our application domain characteristics are similar to the Smart Office/Home and is based on the RFID technology and the EPC Framework.

2.3.1 Internet of Things stack example: EPC Framework

The RFID middleware is the component of a RFID system that sits between the low level components - e.g. readers and tags - and the business client application - e.g. Enterprise Resource Planning (ERP) systems. The next paragraphs describe the EPCglobal, a framework that provides standardized interfaces that isolates hardware vendors from business applications, and Fosstrak, a open-source RFID middleware platform that implements the GS1 EPC Network standards.

2.3.1.1 GS1 EPCglobal Architecture

GS1[14] is an organization that is responsible for the development and maintenance of standards for supply chain. One of the standards developed by GS1 is the EPC, which is an unique serial identifier for

[14]http://www.gs1.org

RFID tags. GS1's subsidiary EPCglobal Inc.[15] created the EPCglobal Architecture Framework, that currently is the standard for RFID platforms. Figure 2.2 presents a high-level architecture of the EPCglobal framework that shows its main interfaces and roles.

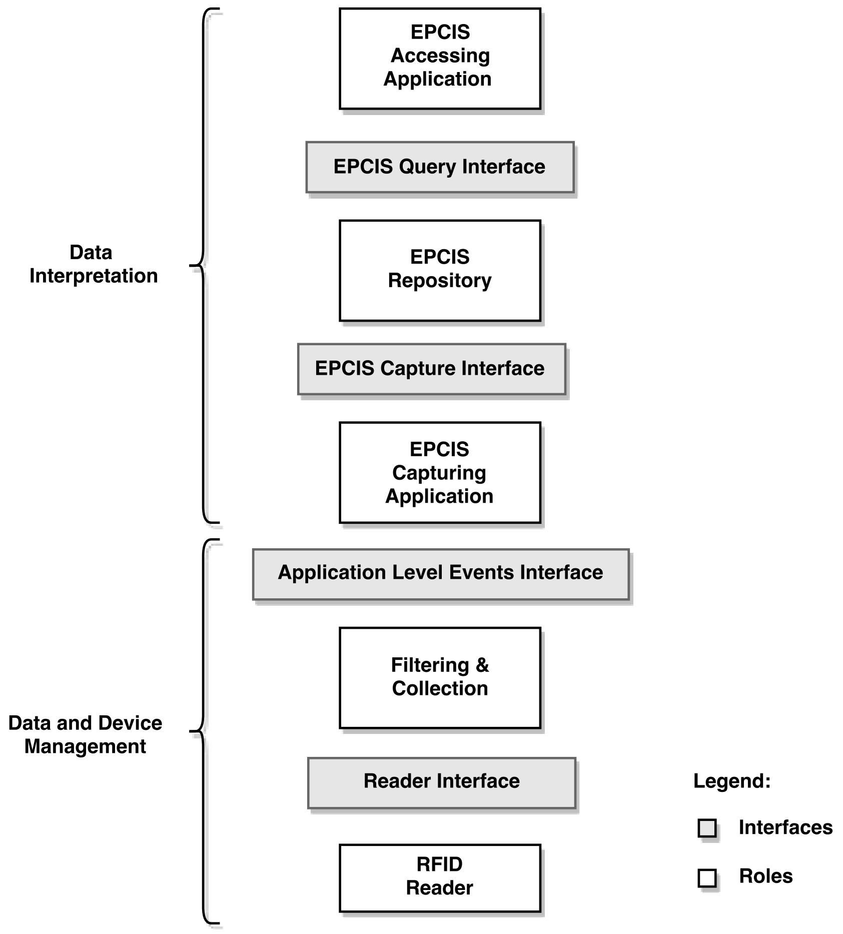


Figure 2.2: GS1 EPCGlobal Architecture Framework.

The framework[16] has a set of standardized interfaces that enables the interchange of information between entities. In the context of our work, the most relevant components of the framework architecture are:

- *Reader Interface* provides the interfaces that must be implemented by the RFID readers. The Low Level Reader Protocol (LLRP) standard provides interfaces that allows the control of all the aspects of RFID reader operation.

- *Filtering & Collection* is the module that coordinates the RFID readers that are in the same physical space and also abstracts the readers from the upper layers. It allows the execution of read and write operations on tags. Furthermore, it is responsible for filtering, aggregating and grouping the raw tag data when requested.

- *Application Level Events (ALE) Interface* defines the control and delivery of filtered and collected data from the Filtering & Collection module to the Electronic Product Code Information System

[15]`http://www.gs1.org/epcglobal`

[16]For more information about the standards of the EPCglobal Framework, the full documentation is available at `http://www.gs1.org/gs1-architecture`

(EPCIS) Capturing Application. The ALEs are a selection of the events that are meaningful for the client applications.

- *EPCIS Capture Application* supervises the operation of the lower EPC layers, and provides business context based on information involved in the execution of a particular step of a business process.
- *EPCIS Repository* is the module where all the business events generated by the EPCIS Capturing Applications are stored to later be accessed by the EPCIS Accessing Application. The EPCIS Query Interface defines how client applications can retrieve information from the repository.

2.3.1.2 Fosstrak Platform

The Free and Open Source Software for Track and Trace (Fosstrak) is an EPCglobal Network compliant RFID software platform that was developed by Floerkemeier et. al [7]. Figure 2.3 presents the architecture of the Fosstrak platform.

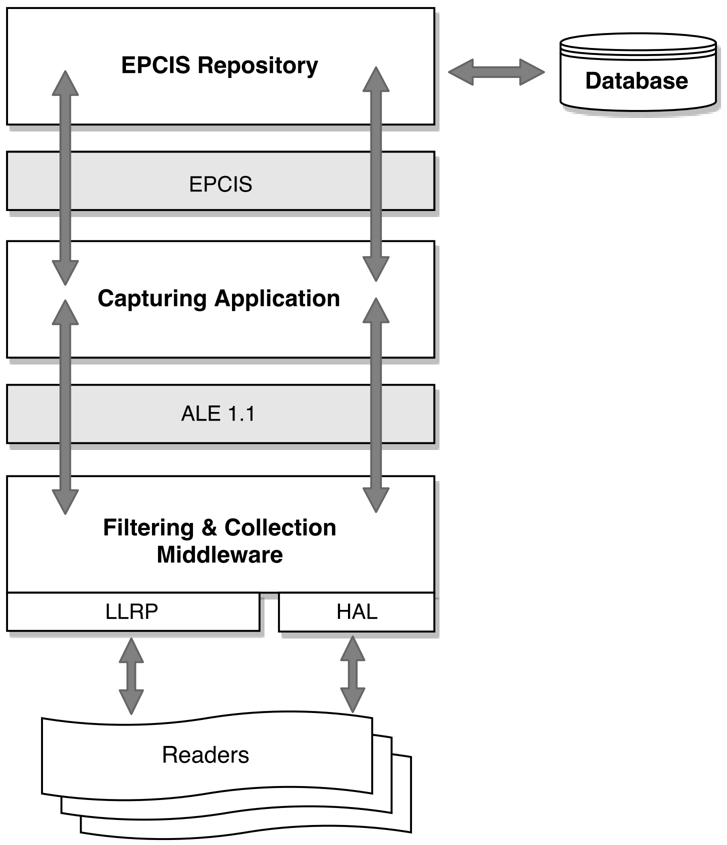


Figure 2.3: Fosstrak architecture.

The Fosstrak platform is composed of three modules that implements the corresponding roles in the EPC Network: *Reader Module*, *Filtering and Collection Middleware Module* and *EPCIS Module*. For our work the relevant modules of the platform are:

- *Filtering & Collection Server* is the module responsible for filtering and collecting data from RFID readers. To communicate with the readers, the module uses the LLRP standard for LLRP compliant readers and uses the Fosstrak Hardware Abstraction Layer (HAL) for unsupported readers.

The module internally abstracts the readers as LogicalReaders instances that are defined and configured through a *LRSpec* document, as defined by EPCglobal. Fosstrak also implements the *Event Cycle*, that is an interval of time during which tags are collected. The output of an *Event Cycle* is the *ECReport* document that is sent to the Capturing Application.

- *Capturing Application* is part of the EPCIS module. This module is responsible for transforming uninterpreted events received on the *ECReports* into meaningful business events. Regarding its implementation, the Capturing Application is built on top of the Drools[17] engine where rules can be specified in the form of: "when" something happens, "then" do "this". Unfortunately, the rules are static and once defined they can not be updated in runtime.

- *EPCIS Repository* provides an EPCglobal-certified EPCIS Repository, which means that all Fosstrak EPCIS modules and interfaces are compliant with the EPCglobal standard. This module provides persistence for EPCIS events. For storing new events the module provides the *capture interface* and the *query interface* for retrieving historical events is provided. Furthermore, the module provides two EPC Network-conformant interfaces to a relational database (currently MySQL).

[17] `http://www.drools.org/`

	Smart Home/Office	Smart Retail	Smart City	Smart Agriculture/Forest	Smart Water	Smart Transportation
Network Size	Small	Small	Medium	Medium/Large	Large	Large
Users	Very Few, family members	Few, community level	Many, policy makers and general public	Few, policy makers and landowners	Few, government	Large, general public
Energy	Rechargeable battery	Rechargeable battery	Rechargeable battery, Energy Harvesting	Energy Harvesting	Energy Harvesting	Rechargeable battery, Energy Harvesting
Internet Connectivity	Wifi, 3G, 4G LTE, backbone	Wifi, 3G, 4G LTE, backbone	Wifi, 3G, 4G LTE, backbone	Wifi, Satellite Communica-tion	Wifi, Satellite Communica-tion, Microwave Links	Wifi, Satellite Communica-tion
Data Management	Local Server	Local Server	Shared Server	Local Server, Shared Server	Shared Server	Shared Server
IoT Devices	RFID, WSN	Smart Retail	RFID, WSN	WSN	Single Sensors	RFID, WSN, Single Sensors
Bandwidth Requirements	Small	Small	Large	Medium	Medium	Medium/Large

Table 2.1: Smart Place application domains characteristics.

Chapter 3

Solution

Usually, provisioning the components of a smart place application is an operation that is manually executed and requires expertise since that the software components must be correctly installed and configured. Furthermore, every time that a new smart place is deployed these operations must be repeated. With that in mind, in order to make the deployment of smart places more efficient in Section 3.1 we propose a solution that automates the provision of smart places application middleware in the cloud.

In the present work our main goal is to determine if a cloud-based deployment can meet the requirements of RFID-based smart place applications, as mentioned in Section 1.3. To achieve our goals we will follow two approaches to deploy the smart warehouse application middleware: cloud and fog. In Section 3.2, we describe the alternative architectures of the smart warehouse regarding the chosen deployment approach.

3.1 Smart Place Provisioning

In this section we propose a mechanism that automates the provisioning of software for smart warehouses in the cloud. Our solution relies on Configuration Management (CM) tools that leverage existing software stacks. Figure 3.1 presents the approach for the proposed mechanism.

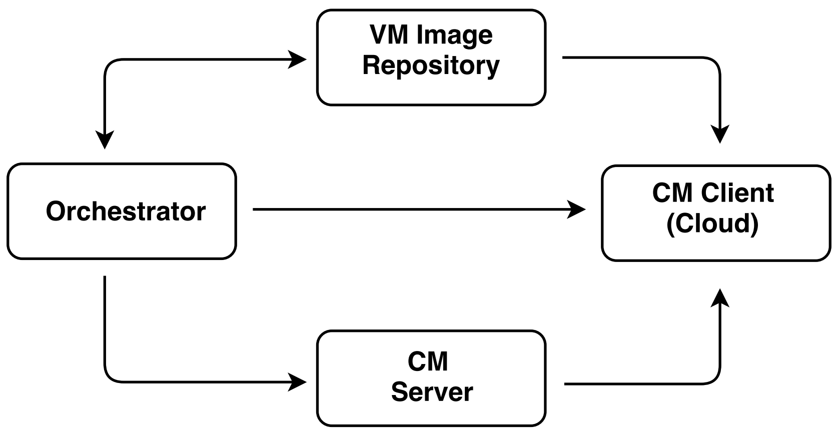


Figure 3.1: Provisioning mechanism conceptual architecture.

In the proposed approach, the provisioning of a smart warehouse is based on provisioning policies and software images that are defined and configured in a development environment. The provisioning policies allow to define which components of software must be provisioned in a given instance, configure management tasks such as to trigger a notification when a resource state changes. The software images contains all the software components required to deploy the smart warehouse application.

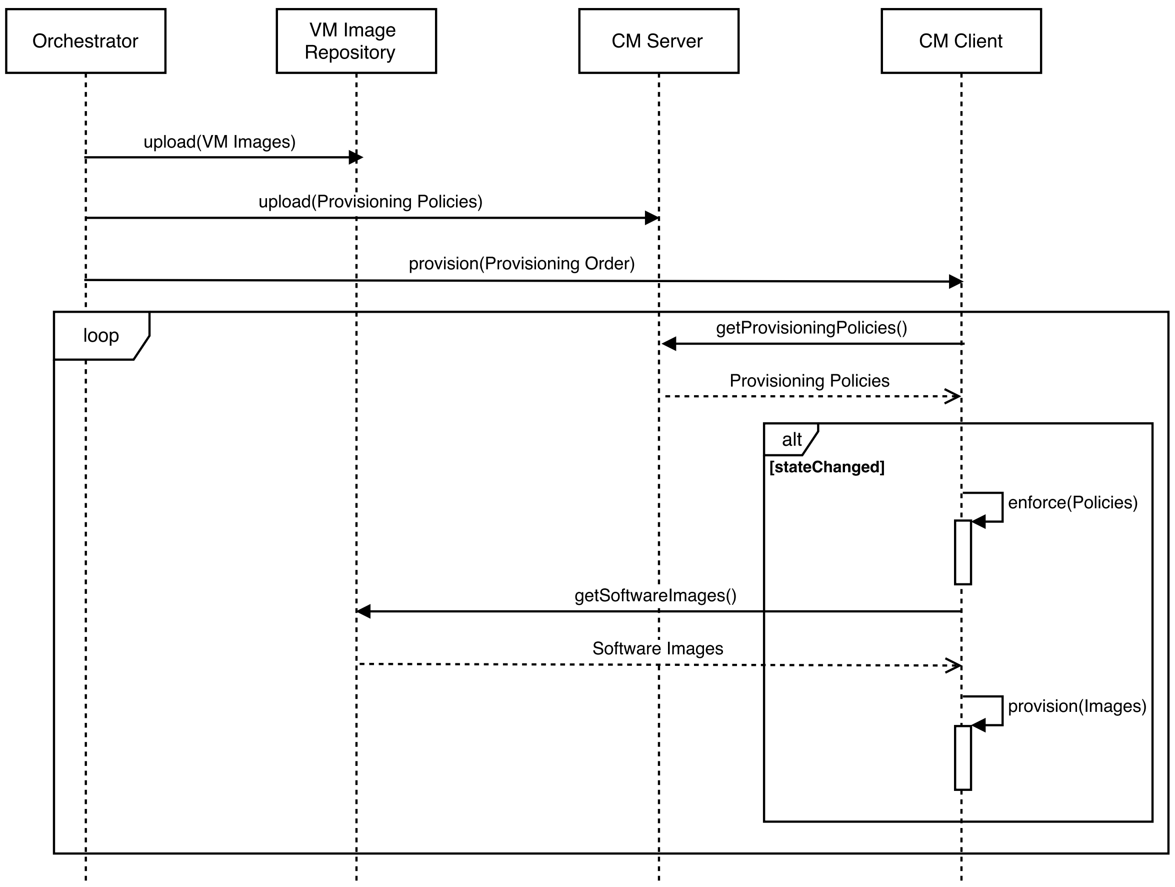


Figure 3.2: Provisioning Mechanism Sequence Diagram.

After the provisioning policies were defined and configured, the Orchestrator uploads them to its respective remote repositories (CM Server and VM Image Repository). When the provisioning request is performed - through a configuration management interface provided by the Orchestrator - the configuration management client (CM Client) in the cloud server pulls the polices from the configuration management server (CM Server), a centralized server that is responsible to maintain a consistent state of the provisioned nodes in the cloud. In order to enforce the polices, the CM Client pulls the software images from a central repository and then performs the provisioning and configuration of the software. After provisioning the infrastructure, the CM client periodically polls the CM server in order to determine if its current state is consistent with the most recent policy, as illustrated in Figure 3.2

3.1.1 Implementation Details

The implementation of the provisioning mechanism relies on the Chef tool. Chef provides several features that allow to describe infrastructure as code. In the implementation of the provisioning mechanism, the main features used were the Chef *recipes*, *roles* and *knife* command-line tool[1]. In the current implementation, we used Docker containers to provisioning the smart warehouse software at the cloud providers in alternative to traditional VMs.

3.1.1.1 Provisioning Recipes

The *recipes* that describe the smart warehouse infrastructure are based on *cookbooks* available at the Chef Supermarket[2] and also in custom recipes that were defined specifically for this work to describe how the Fosstrak software stack is provisioned in the cloud. Since we are using Docker containers the recipes were defined based on the official Docker cookbook for describing how the containers must be provisioned. For instance, Listing 1 present the provisioning recipe for the Docker container that runs the EPCIS Repository:

```
# Pull latest image
docker_image 'cloud4things/fosstrak-epcis'

# Run container exposing port 8080
docker_container 'cloud4things/fosstrak-epcis' do
  detach true
  container_name 'fosstrak-epcis'
  link 'fosstrak_db:db'
  expose '8080'
end
```

Listing 1: EPCIS Docker container provisioning recipe.

The recipe specification describes that first the Docker image identified as *cloud4things/fosstrak-epcis* must be pulled from the central repository. In particular, this image contains the software required to deploy the EPCIS repository web application, namely an Apache Tomcat web-server and the EPCIS repository source code. Finally, the recipe describes that the image *cloud4things/fosstrak-epcis* must be used to create a container named *fosstrak-epcis* and the port *8080* need to be exposed. Furthermore this container must be linked to the container *fosstrak-db* that internally is represented by the alias *db*. The *detach* parameter describes if the container must run in the background or not. Optionally, the recipe attributes can be parametrized through a JavaScript Object Notation (JSON) file that specifies the attributes value.

We also defined the recipes that provision Docker containers for the remaining Fosstrak stack, namely the *Filtering & Collection Server* (Listing 2), *Capturing Application* (Listing 3) and *MySQL* database (Listing 4).

[1]The tool uses culinary analogy in most of its concepts
[2]https://supermarket.chef.io/

```
# Pull latest image
docker_image 'cloud4things/fosstrak_ale'

# Run container exposing port 8081 and remaping to port 8080
docker_container 'cloud4things/fosstrak_ale' do
  detach true
  container_name 'fosstrak_ale'
  link 'fosstrak_capture:capture'
  port "8081:8080"
end
```

Listing 2: ALE Docker container provisioning recipe.

```
# Pull latest image
docker_image 'cloud4things/fosstrak_capture'

# Run container exposing port 9999
docker_container 'cloud4things/fosstrak_capture' do
  detach true
  container_name 'fosstrak_capture'
  link 'fosstrak_epcis:epcis'
  expose '9999'
end
```

Listing 3: Capturing application Docker container provisioning recipe.

```
# Pull latest image
docker_image 'cloud4things/fosstrak_db'

# Run container exposing port 3306
docker_container 'cloud4things/fosstrak_db' do
  detach true
  container_name 'fosstrak_db'
  expose '3306'
  volume '/mnt/docker:/docker-storage'
end
```

Listing 4: MySQL Docker container provisioning recipe.

3.1.1.2 Provisioning Roles

A *role* is a categorization that describes what are the responsibilities of a specific node, what settings and software components should be given to it. For instance, it is possible to define what are the nodes that includes the database, web server, etc. The roles allows to describe the smart place warehouse topology in the cloud.

The roles that describe the smart warehouse application topology were defined based in the architecture of the smart warehouse for the cloud and fog deployment approaches, presented in Section 3.2. In the next sections, we describe with more detail the roles that were defined to provisioning the smart warehouse infrastructure.

Cloud Provisioning Roles. In a cloud-based deployment the Fosstrak software stack is provisioned in a single instance, as illustrated in Figure 3.8. Thus, we defined a single role to describe the responsibilities of the node provisioned in the cloud.

```
{
  "name" : "fosstrak-server",
  "run_list" : [
    "recipe[docker]",
    "recipe[docker::fosstrak-db]",
    "recipe[docker::fosstrak-epcis]"
    "recipe[docker::fosstrak-capture]",
    "recipe[docker::fosstrak-ale]",
  ]
}
```

Listing 5: Cloud Deployment: provisioning role.

The *fosstrak* role describes that the nodes must have installed all the modules of Fosstrak that are specified in the docker *cookbook*: *docker*, *fosstrak-db*, *fosstrak-epcis*, *fosstrak-capture* and *fosstrak-ale* recipes. The nodes that have assigned this role are identified as *fosstrak-server*.

Fog Provisioning Roles. In a fog-based deployment the Fosstrak software stack is distributed across the fog and cloud, as illustrated on Figure 3.9. Therefore, we defined two different roles that describe the responsibilities of the provisioned nodes.

```
{
  "name" : "fog-server",
  "run_list" : [
    "recipe[docker]",
    "recipe[docker::fosstrak-capture]",
    "recipe[docker::fosstrak-ale]"
  ]
}
```

Listing 6: Fog Deployment: Fog provisioning role.

The *fog* role describes that fog nodes must have installed the following resources specified in the recipes of the docker *cookbook*: *docker*, *fosstrak-capture* and *fosstrak-ale*. The nodes that have assigned this role are identified as *fog-server*, as illustrated in Listing 6.

The *cloud* role describes that cloud nodes must have installed the remaining modules of Fosstrak that are specified in the docker *cookbook*: *docker*, *fosstrak-db* and *fosstrak-epcis* recipes. The nodes that have assigned this role are identified as *cloud-server*, as illustrated in Listing 7.

```
{
  "name" : "cloud-server",
  "run_list" : [
    "recipe[docker]",
    "recipe[docker::fosstrak-db]",
    "recipe[docker::fosstrak-epcis]"
    ]
 }
```

Listing 7: Fog deployment: Cloud provisioning role.

3.1.1.3 Provisioning Mechanism

To provisioning the resources in the cloud instances we used *knife*, a command-line tool developed by Chef that provides an interface between the local Chef repository and the Chef server. The provisioning mechanism architecture is illustrated in Figure 3.3.

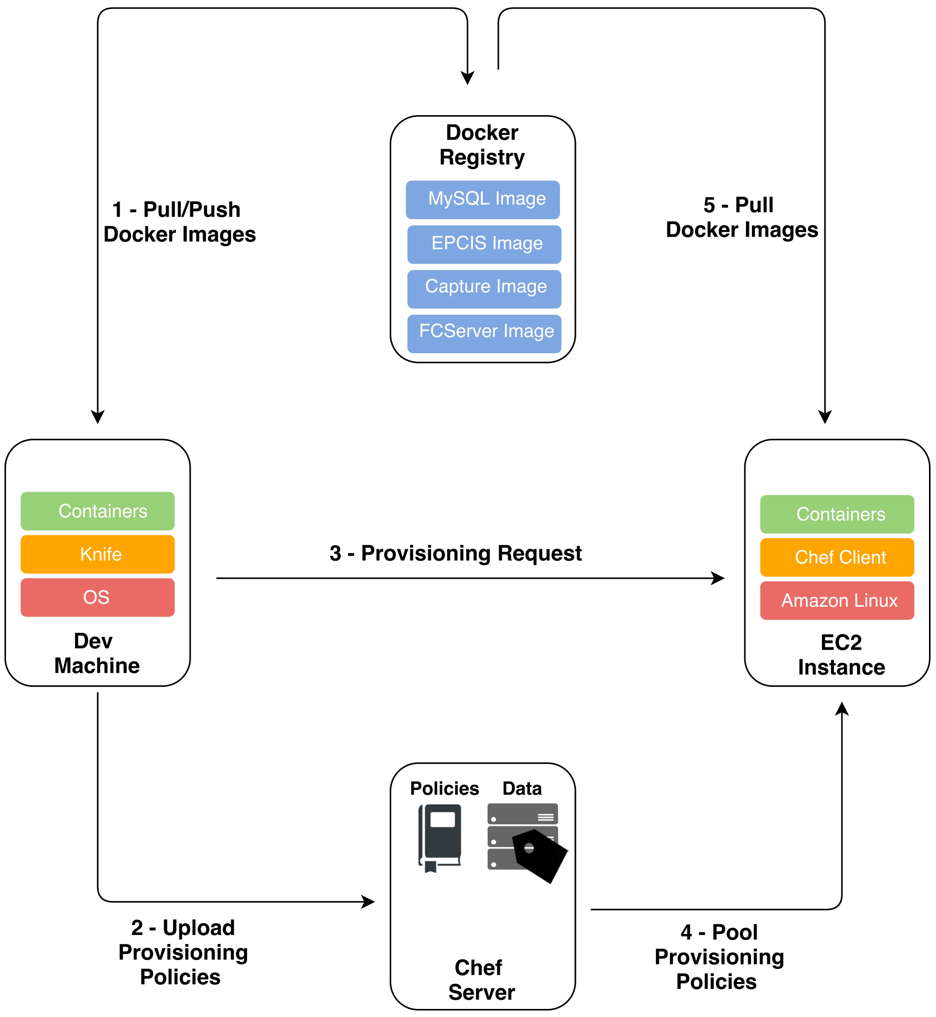


Figure 3.3: Automatic provisioning mechanism architecture.

In a development environment the Docker images are built and then uploaded to the Docker Registry repository (1). The provisioning of the cloud resources is described in the cookbooks that are uploaded to the Chef server (2). The provisioning request (3) is performed using *knife*, that allows to describe the image type, the instance type and the policies - e.g. the *role(s)* and/or *recipe(s)* - that need to be applied on each provisioned node. Then the Chef client runs the configuration policies that are pulled from the Chef server (4). In our solution the Chef client apply the configuration recipes that are described in the role assigned to the node. The Chef client pulls the Docker images from the remote repository, build the containers based on those images and finally applies the configuration that is associated to each container.

We decide to chose Chef instead of its competitors - i.e. Puppet and Ansible - for several reasons. First, the *knife* tool is very powerful and allow us to interact with our entire infrastructure. For instance, it is possible to bootstrap a new server, apply a role to a set of nodes in our environment. Furthermore, with *knife ssh* it is possible to execute a command on a certain number of nodes in our environment. For instance, if we change the role configuration that is assigned to a set of nodes in our infrastructure, knife allows to update all these nodes with the most recent policy with a single command. Also, knife provides plugins for several cloud providers, such as AWS[3], and Google Compute Engine[4]. These plugins allow the provisioning of the application in the cloud providers infrastructure using the same resources - e.g. roles, recipes, etc - for all available providers.

In order to make the provisioning of the instances in the cloud more efficient, our provisioning mechanism should be able to provision multiple instances with a single provisioning order. However, the current implementation of Chef tool does not support the provisioning of a cluster of nodes, but there are third-part plugins that already support this operation such as the spiceweasel[5] command-line tool.

3.1.1.4 Docker Containers

Docker containers are used to provisioning the software stack of Fosstrak platform. A complete installation of Fosstrak requires a compatible Java Software Development Kit (SDK), a full MySQL database and a Apache Tomcat server.

In the current implementation, we are provisioning a single container for each component of Fosstrak, the EPCIS repository, the Capture application, the Filtering & Collection Server (FCServer) and also for the MySQL database, as illustrated in Figure 3.4. Currently, the container images of Fosstrak are published in the Docker Hub repository to later be used to create the containers.

By default each container runs a process that is isolated from the other processes that shares the same environment (kernel). Compared with the isolation provided by traditional VMs - which are fully isolated - the isolation provided by Docker containers is less secure, since that if a container has its security broken, it is possible that other containers and host may be compromised.

In order to connect the different modules of the Fosstrak, our containers are linked through the *linking system* provided by Docker. This mechanism creates a secure tunnel between the containers, allowing the recipient container to access selected data about the source container. For instance, our EPCIS container - which is linked to the MySQL database container - is able to access information about the MySQL container.

[3]`https://aws.amazon.com/`
[4]`https://cloud.google.com/compute/`
[5]https://github.com/mattray/spiceweasel

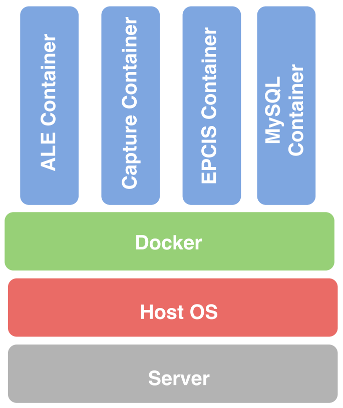


Figure 3.4: Fosstrak containers stack.

The reasons that we chose Docker containers instead of traditional VMs is that containers require less disk space and read/write (I/O) operations in the disk when compared with traditional VM images [30]. Furthermore, Docker containers are easier to port to another infrastructure when compared with traditional VMs because when a container requires that the application and all of its dependencies are ported together while the VMs require that the entire application, the guest operating system, the binaries and libraries are ported, which can be several Gigabytes (GBs) in size.

3.2 Smart Warehouse Deployment

Our smart place is an automated warehouse where automated vehicles transport tagged objects that can be identified by readers and sensors that are deployed in the place, as described in Section 1.3.1. In traditional solutions, the application is provisioned in a local infrastructure. Although such approach guarantees that the low-latency requirements are meet, this solution comes with several downsides - such as the low scalability, infrastructure and maintenance costs - that can be a barrier for these applications.

Leveraging the infrastructure required to provisioning the smart warehouse application to the cloud guarantees that the downsides of traditional solutions are solved. However, we also need to guarantee that the latency requirements of these applications are fulfilled. The cloud and fog concepts give us more flexibility to perform the deployment of smart warehouse applications, which allow us to provisioning the application modules in a more distributed way. The following sections describe the deployment approaches of smart warehouse applications based in the cloud and fog concepts.

3.2.1 Cloud Deployment

Figure 3.5 presents the architecture of a cloud-based smart warehouse deployment. The warehouse is composed of smart objects, sensors and readers that capture the events that occurs in the warehouse. The application middleware is provisioned in the cloud, which virtualizes the computing, storage and network resources needed to support the application.

The smart warehouse can be connected to the cloud through a physical or wireless connection.

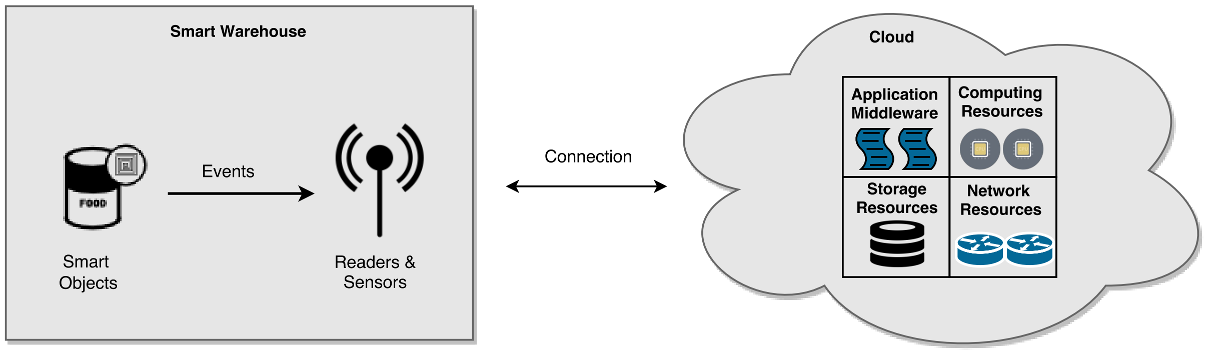


Figure 3.5: Cloud deployment: smart warehouse conceptual architecture.

3.2.2 Fog Deployment

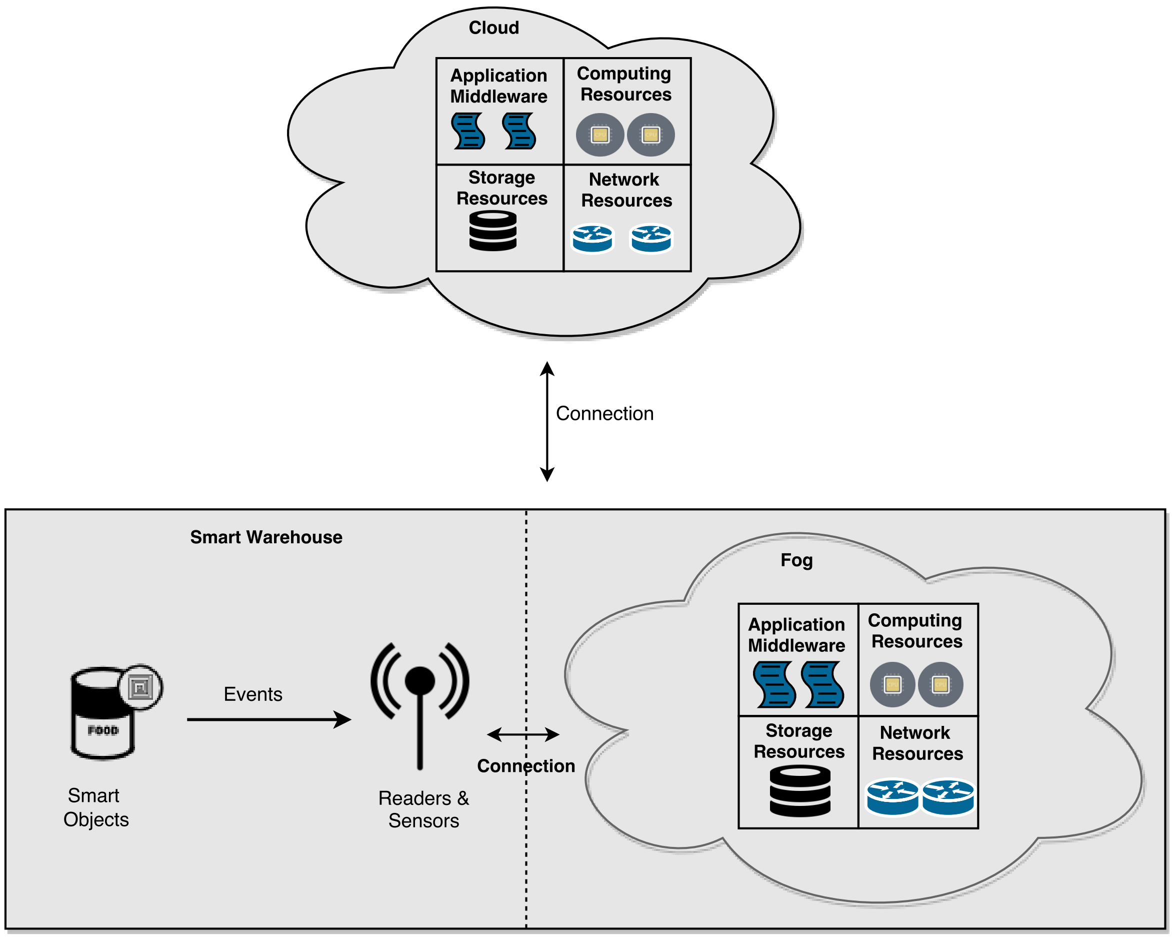


Figure 3.6: Fog deployment: smart warehouse conceptual architecture.

Figure 3.6 presents the architecture of a fog-based smart warehouse deployment. As in the cloud-based deployment the warehouse is composed of smart objects, sensors and readers. The proposed approach aims to extend the cloud paradigm to the edge of the network. The fog achieves that by virtualizing computing, storage and network resources. Unlike the cloud infrastructure, that usually is provisioned thousands of kilometers from the smart warehouse, the fog infrastructure usually is provisioned closest to the smart warehouse.

Regarding the application middleware, the application components are distributed across the cloud and fog. The components responsible for storing the data during a long period of time are provisioned in the cloud. The components responsible for performing real-time processing of the data generated in the warehouse, and the components that filter the data that is consumed locally and must be delivered to the cloud are provisioned in the fog.

Both the smart warehouse as the fog can be connected respectively to the fog and cloud through several types of connection, from a physical connection to a wireless connection.

3.2.3 Implementation Details

The smart warehouse setup was based in a demonstration scenario described by Correia et al. [31], as illustrated in Figure 3.7.

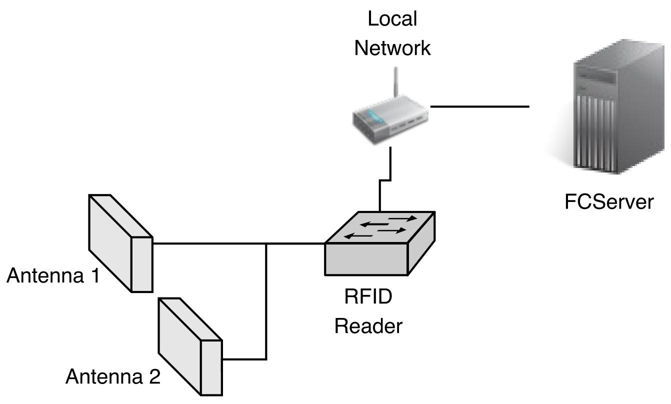


Figure 3.7: RFID application setup.

The warehouse is composed of a robot transporting tagged products that are identified by RFID readers deployed in the physical space. To monitor the robot inside the warehouse, the Fosstrak RFID middleware is used. In our implementation, the RFID readers are emulated through the Rifidi Emulator, which uses the LLRP protocol to communicate with the Fosstrak platform.

3.2.3.1 Cloud Deployment

The RFID middleware is provisioned in the cloud in a single virtual machine. In the Fosstrak implementation the FCServer, EPCIS repository and the Capture application requires an Apache servlet container to deploy and run the web applications. The EPCIS repository is connected to a MySQL database that stores the event data. The technological architecture for a cloud-based deployment is presented in Figure 3.8.

The smart warehouse can be connected to the cloud through a physical (e.g. Asymmetric Digital Subscriber Line (ADSL) or Fiber-optic) to a wireless connection (e.g. Wi-Fi, 3G or Long-term Evolution (LTE)).

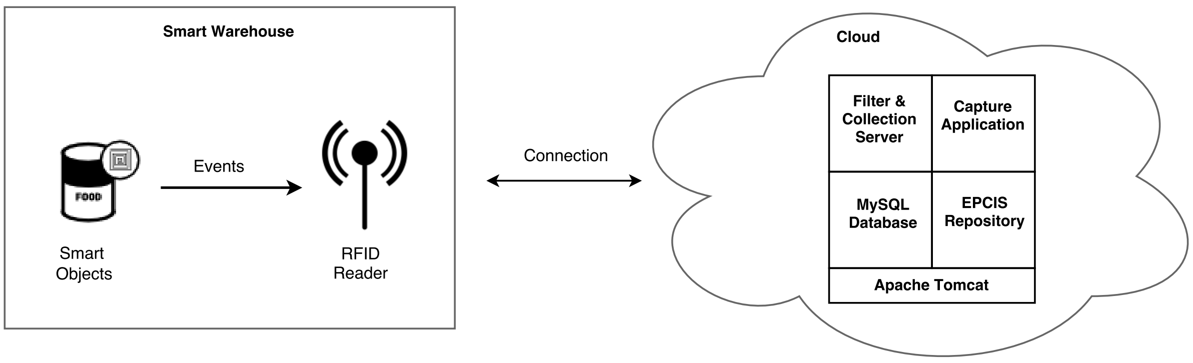


Figure 3.8: Cloud deployment: smart warehouse technological architecture.

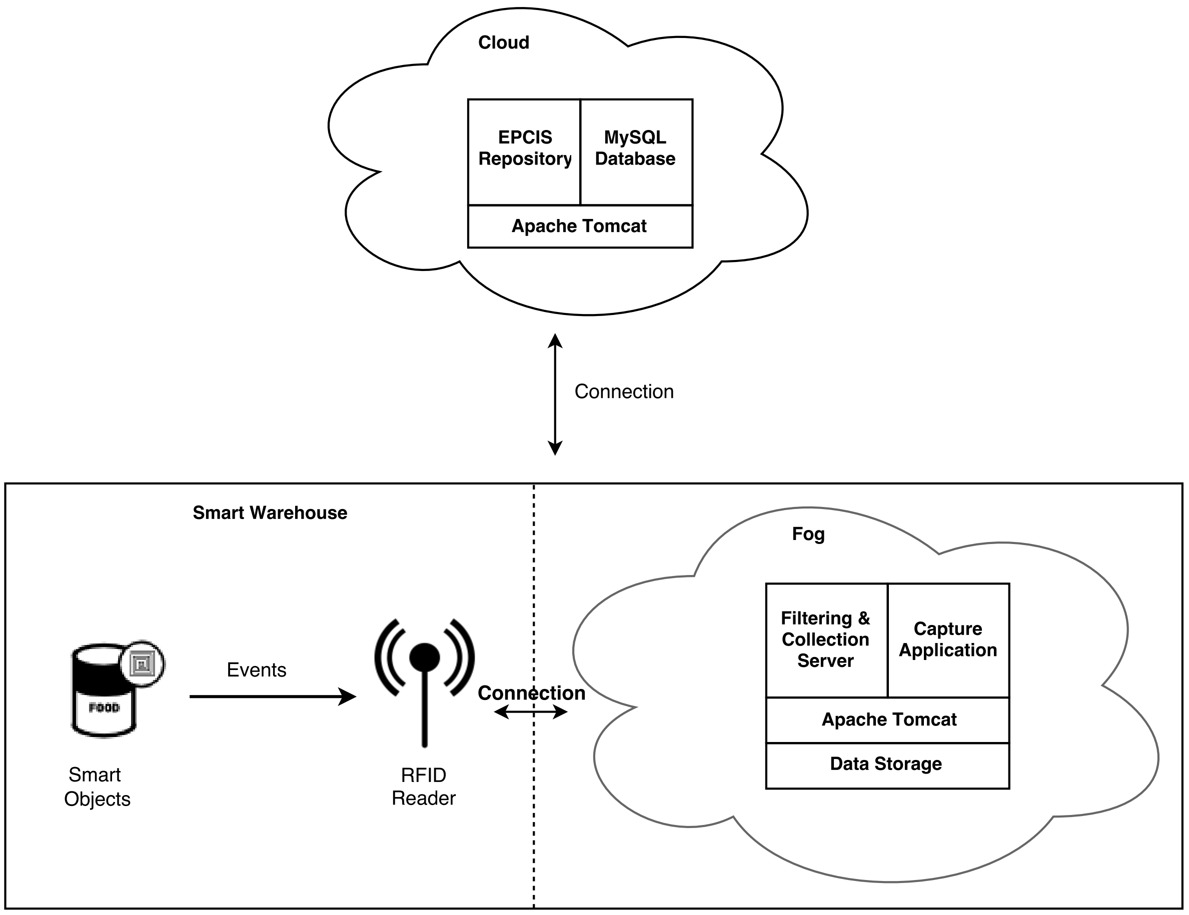


Figure 3.9: Fog deployment: smart warehouse technological architecture.

3.2.3.2 Fog Deployment

Figure 3.9 presents the technological architecture for a fog-based deployment. The RFID middleware is provisioned across the fog and the cloud. At the cloud, all the software components are provisioned in a single VM. The EPCIS repository is deployed and running on top of an Apache Tomcat servlet instance. The repository is connected to a MySQL database, which stores the event data. In the current implementation the fog was built with a traditional VM. The FCServer and the Capture application are deployed and running on top of a single Tomcat servlet instance. The Capture application sent the events collected by the FCServer to the EPCIS repository through the *EPCIS Capture Interface* - via

Hypertext Transfer Protocol (HTTP) requests.

Both the smart warehouse as the fog can be connected respectively to the fog and cloud through several types of connection, from a physical connection (e.g. ADSL or Fiber-optic) to a wireless connection (e.g. Wi-Fi, 3G or LTE).

3.3 Summary

In this chapter we proposed a solution to automate the provisioning of smart warehouse applications based on the RFID technology in the cloud. The implementation details for the proposed solution and the technological components used in our implementation also were discussed.

Our provisioning mechanism is based on the Chef tool. To provisioning Fosstrak's software stack we are using Docker containers in alternative to the traditional VMs. The provisioning mechanism relies on the *recipes*, *roles* and the *knife* command, which are provided by Chef. We defined a set of recipes that allow to describing how the Fosstrak software stack should be installed in a node while the roles allow to attributing responsibilities to a specific node. The knife command is used to execute the provisioning request and to interact with the provisioned infrastructure.

The provisioning mechanism implemented in this work allows to perform the deployment of a smart warehouse application based in Fosstrak across the cloud and fog, although it can be extended to support other RFID middleware platforms. The flexibility in the deployment provided by the cloud and fog concepts will allow us to evaluate the approaches and chose which one is more adequate to support smart warehouse applications based on the RFID technology.

Chapter 4

Evaluation

The present chapter describes the evaluation methodology as well as the experiments performed to determine if a cloud-based or a fog-based deployment is more adequate to support smart warehouse applications based on the RFID tehcnology. The evaluation process will compare the approaches in regard to low-latency and data storage performance.

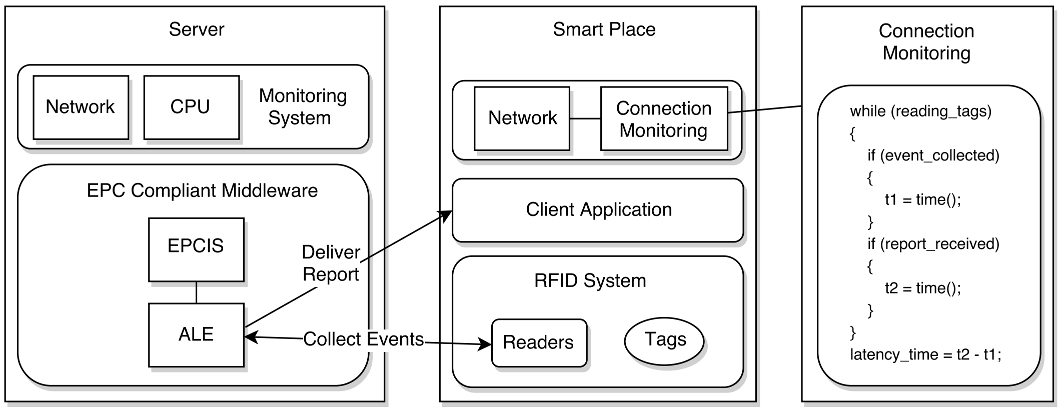
Our main goal is to obtain basic statistical values that allow us to decide which approach is more suitable to fulfill the expected requirements. Furthermore, we present a discussion regarding which approach is more adequate to deploy an IoT application according its domain.

4.1 Evaluation Methodology

In order to determine which deployment approach is more adequate to meet the latency and data storage scalability requirements, we propose the following methodologies to perform the evaluation:

4.1.1 Latency Interaction

The response time between an event that occurs in the smart warehouse and the corresponding action that is triggered in the physical space, the proposed methodology consists in perform the monitoring of the smart warehouse network and determine how much time is spent between the ALE collect the event triggered and client application receive the notification report, as illustrated in Figure 4.1.



```
while (reading_tags)
{
    if (event_collected)
    {
        t1 = time();
    }
    if (report_received)
    {
        t2 = time();
    }
}
latency_time = t2 - t1;
```

Figure 4.1: Latency interaction evaluation methodology.

The ALE module is responsible for collecting and processing the reader events and take the correct decisions based on those events. In the Fosstrak implementation the collection and processing of reader events is performed according to an *Event Cycle* specification. The *Event Cycle* is a set of periodical cycles where the ALE module collect the events from the readers. The data about the *Event Cycle* is delivered to the client application through a report. The information in the report can be used to notify the client regarding an event in the smart warehouse or even to trigger a new event in the warehouse such as open or close a door.

The smart warehouse is running a monitoring system that stores information about the incoming and outgoing packets in the warehouse network. In particular, we will perform the network monitoring with *tcpdump*[1], a command-line tool that allows the monitoring of the packets that are being transmitted or received over a network. Through the logs produced by this tool, we are able to determine how the connection time is spent.

The Fosstrak ALE module can be configured to generate information to register when a new event is processed and also when a new report is delivered to the client. Thus, with the information provided by the monitoring system and the ALE module it is possible to calculate the latency request for an event that occurs in the warehouse.

With this methodology we intend to obtain information regarding how the communication time is spent when an event is triggered in the warehouse for the deployment approaches described in Chapter 3. In order to determine which approach is more adequate to deploy the application, we propose a set of metrics that allow us to measure how much time is spent in the network communication, event processing and the latency of the events that occur in the physical space:

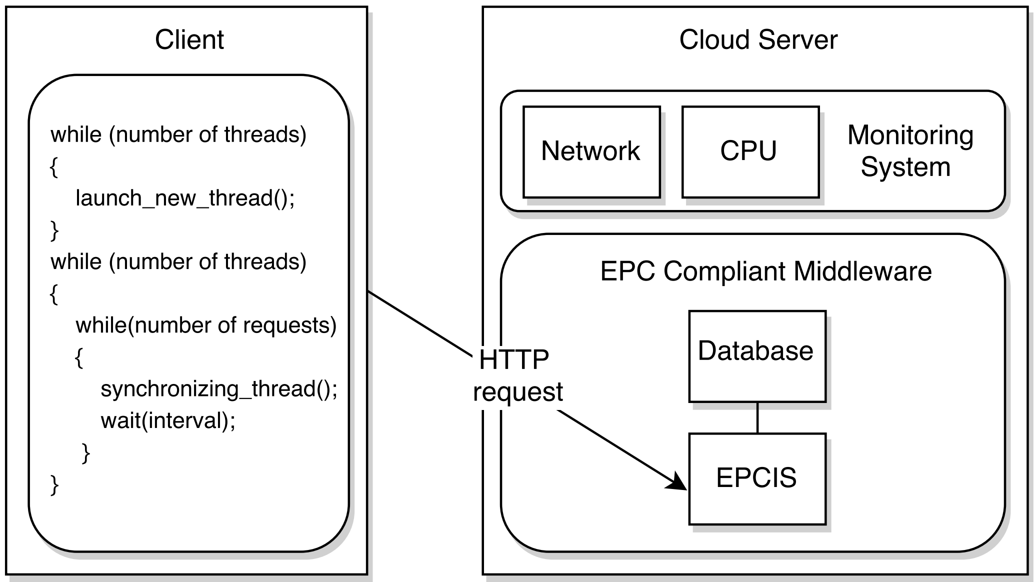
1. Latency
    (a) *Event Latency*: the time spent since an event is triggered in the warehouse until the client application receives the notification of the event.
2. Network Communication
    (a) *Upload Time*: the time spent since that an event is triggered in the warehouse until the ALE module receives the event.
    (b) *Response Time*: the time spent since that the ALE module delivers the *Event Cycle* report until the client receives it.
3. Event Processing
    (a) *Tag Processing Time*: the time spent since that the ALE module receives an event notification until the RFID tag is processed.
    (b) *Idle Processing Time*: the time spent for the ALE module where the collected tags already exists and no further action is required.
    (c) *Filtering & Aggregation Time*: the time spent for the ALE where the collected tags are filtered and aggregated.
    (d) *Report Creation Time*: the time spent for the ALE module to create the *Event Cycle* report for the current *EventCycle*.

[1]http://www.tcpdump.org/

The analysis of these metrics will allow us to compare the performance of the fog-based and cloud-based deployment approaches and help to determine which one is more adequate to deploy the smart warehouse application.

4.1.2 Data Storage Performance

Smart warehouse applications usually generates a large amount of data that must be stored and processed in a efficient way, as described in Section 1.2. To evaluate the data storage performance for a smart warehouse application based on Fosstrak, the proposed methodology consists in stress the EPCIS Repository by simulating several readers that are concurrently triggering events - through HTTP requests - to the repository that is running in the cloud, as illustrated in Figure 4.2.



```
while (number of threads)
{
   launch_new_thread();
}
while (number of threads)
{
   while(number of requests)
   {
      synchronizing_thread();
      wait(interval);
   }
}
```

Figure 4.2: Data storage performance evaluation methodology.

The cloud server is running a monitoring system that periodically stores data related to a set of system metrics, such as *CPU Utilization* and the volume of incoming network traffic by the instances (*Network In*). In the present work, the monitoring of the VMs in the cloud and for collecting the system metrics from the instances, we will use AWS CloudWatch[2], a monitoring service provided by Amazon.

The analysis of these metrics allows to observe how the performance and behavior of the EPCIS module is affected regarding the amount of events that are generated in the smart warehouse.

4.2 Evaluation Setup

To perform the evaluation experiments we chose AWS as cloud provider. All the experiments were conducted in AWS Elastic Cloud Computing (EC2) instances running the Amazon Linux Amazon Machine Image (AMI) operating system. The VMs presents a configuration with a 2.5 *Gigahertz (GHz)* single-core processor with 1 *GB* of Random-access memory (RAM). In the fog-approach configuration, the experiments were conducted in a VM with a 2.6 *GHz* dual-core processor with 2 *GB* of RAM and running the *Linux Ubuntu 14.04.2 LTS* operating system. The smart warehouse was connected to the

[2]http://aws.amazon.com/cloudwatch/

cloud and fog through a ADSL connection with a bandwidth of 10 Mbps.

Regarding the software components, the application stack is composed of the *Apache Tomcat 7.0.52.0*[3] with *Java* version *1.7.0* update *79*. The RFID middleware used was the Fosstrak (described in section 2.3.1.2) and the versions were: *a) FCServer* version *1.2.0*; *b) Capture Application* version *0.1.1*; and *c) EPCIS Repository* version *0.5.0*. Furthermore, the EPCIS Repository was connected to a *MySQL server* version *5.5*. The Rifidi Emulator[4] used to emulate the RFID readers is in version *1.6.0*.

4.3 Experiments Performed

The experiments performed in our evaluation were based on the scenario and data amount from the RFIDToys [32] Master Thesis. In short, the RFIDToys is a warehouse demonstrator consisting of a robot transporting tagged products that are identified by RFID readers deployed in the physical space, as detailed in Section 3.2.3. In the performed experiments, we used a scenario where a tagged robot was programmed to execute a given number of laps in the warehouse plant where an automatic door opens when one of the RFID readers that are placed in the plant detects that the robot is approaching.

Based on this scenario and application domain we defined a set of non-functional requirements that our solution must accomplish. The evaluation requirements are presented in Table 4.1.

Name	Description
R1	The event latency must be $<$ than the robot wait time.
R2	The network latency *Upload Time* $+$ *Response Time* must be at the $< 100ms$.
R3	The idle time of an *Event Cycle* must be $<$ than half of the event latency.
R4	The EPCIS must support at least 5 users sending simultaneously a large amount of events ($\approx 18 \times 10^3$).

Table 4.1: Evaluation requirements.

4.3.1 Latency Interaction

To evaluate the latency interaction according the methodology proposed in Section 4.1.1, During the lap the robot stops during 5 seconds in front of the door and then continues its lap. The door must be opened before the robot starts to moving again.

To perform the simulation we defined two different specifications (*ECspec*) for the *Event Cycles* of the ALE module, *Baseline Event Cycle* and *Half-period Event Cycle*. The configuration parameters for the *ECspecs* are presented in Table 4.2.

Event Cycle Specification	Period	Duration	Iterations
Baseline Event Cycle	10s	9.5s	10
Half-period Event Cycle	5s	4.75s	20

Table 4.2: Event Cycle parameters.

[3]`http://tomcat.apache.org/`
[4]`http://rifidi.org`

The evaluation of the event latency for the proposed approaches was performed in two steps. First, we want to determine during a *Event Cycle* how much time the ALE is processing an event and how much time the module is in an idle state. Furthermore, we want to determine how much time each stage of the event processing pipeline takes. The event processing pipeline is divided in the following stages: (i) the event data is uploaded; (ii) event data is processed; (iii) event data is filtered and aggregated; (iv) the event cycle report is created; and finally (v) the event cycle report is delivered to the client application, as illustrated in Figure 4.3.

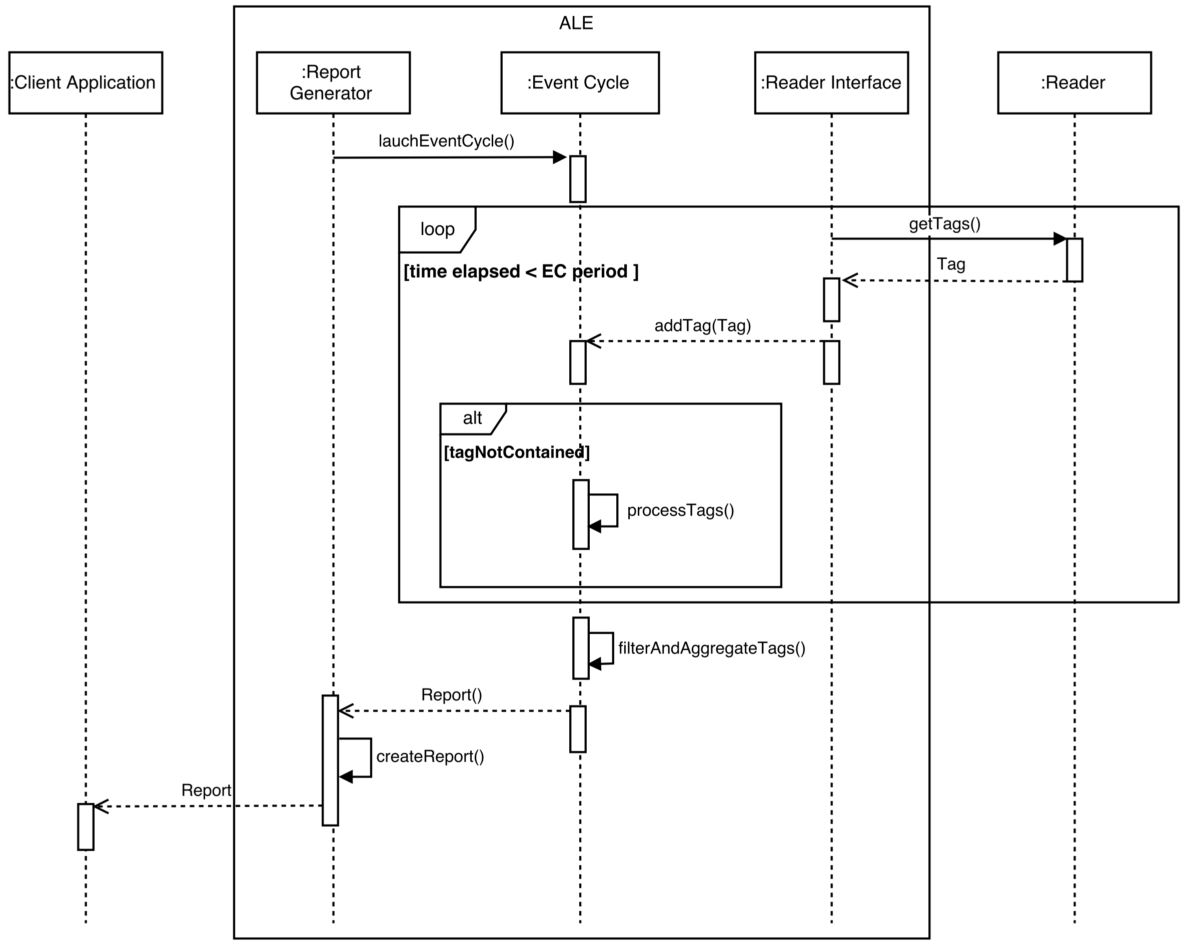


Figure 4.3: Latency Interaction sequence diagram.

4.3.1.1 Cloud-based warehouse latency

The behavior expected when the ALE is configured with a faster *Event Cycle* specification is that the event latency presents a better overall performance. According the metric values presented in Table 4.3 it is possible to observe that the event latency decrease from 8.244s to 4.266s. The values for the network latency improved when the ALE is configured with the faster *ECspec*, close to $\approx 65\%$ of improvement for the *Upload Time* metric - from 0.294s to 0.103s - and $\approx 40\%$ for the *Response Time* metric - from 0.228s to 0.149s. The values for the time where the ALE remains in an idle state also presented a significant improvement, from 7.346s to 2.569s.

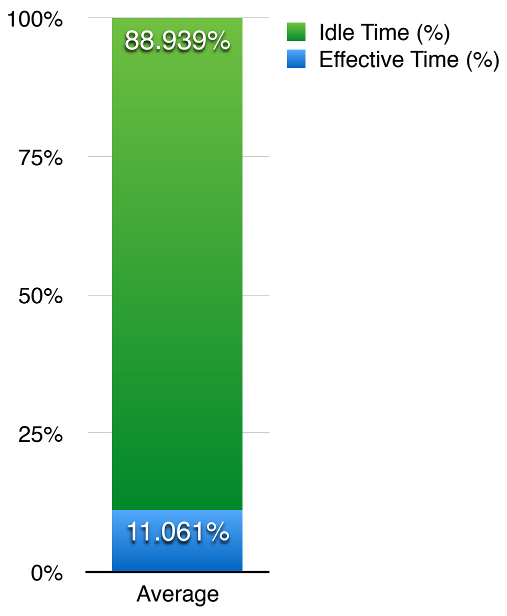
The value for the *Tag Processing Time* increased $\approx 1000\%$ when the ALE is configured with *Half-period ECspec* - from 0.002s to 0.024s. The value for the *Filtering & Aggregation Time* metric increased

Metric	Base Event Cycle		Half-period Event Cycle	
	s	%	s	%
Upload Time	0.294	3.244	0.103	2.373
Tag Processing Time	0.002	0.029	0.024	0.608
Idle Processing Time	7.346	88.939	2.569	62.472
Filter & Aggregation Time	0.370	4.675	1.490	31.019
Report Creation Time	0.003	0.035	0.003	0.063
Response Time	0.228	3.078	0.149	3.467
Event Latency	8.244	100.00	4.266	100.00

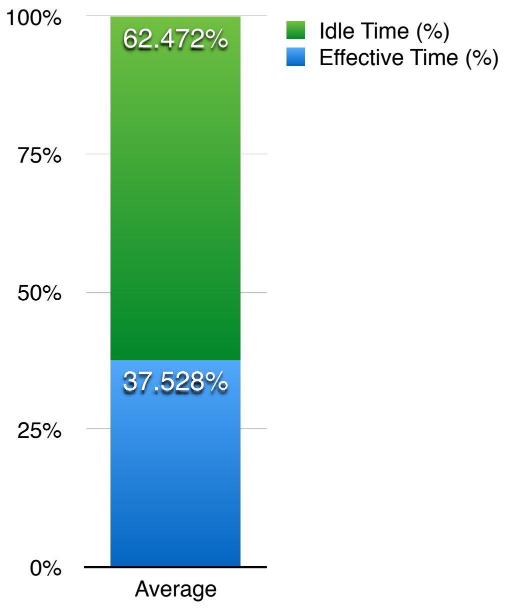
Table 4.3: Cloud deployment: performance metrics results.

$\approx 300\%$ - from 0.370s to 1.490s.

Event Time Breakdown. Figure 4.4 summarizes the latency breakdown for an event that occurs in a smart warehouse deployed in the cloud. In the current experiment the ALE module was configured with the *Baseline ECspec* and the *Half-period ECspec*.



(a) Baseline Event Cycle: latency breakdown.



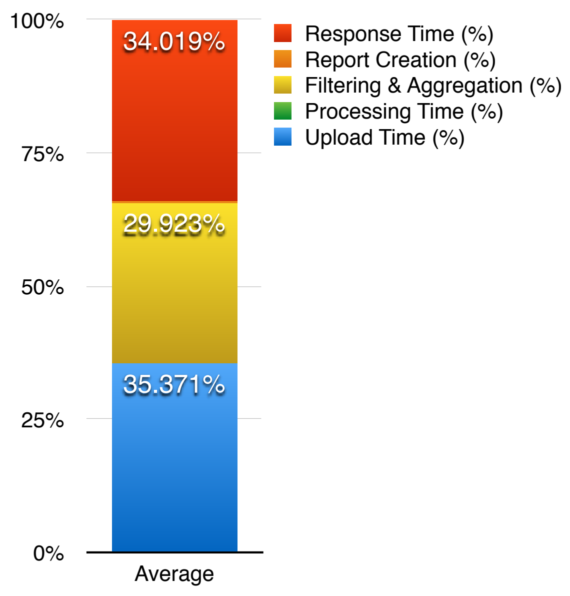
(b) Half-period Event Cycle: latency breakdown.

Figure 4.4: Cloud deployment: Event Cycle latency breakdown.

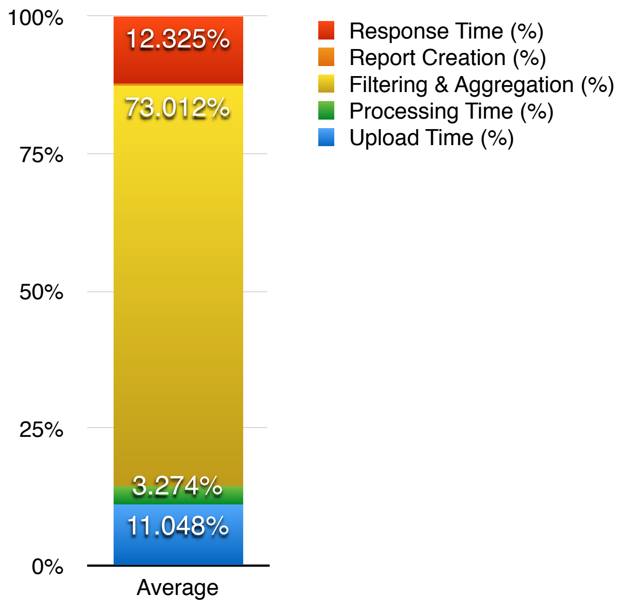
The graphs presented in Figure 4.4a and Figure 4.4b show that, during most of time of an *Event Cycle* the ALE module is in an idle state. Also, it is possible to observe that the *ECspecs* affected the percentage of time where ALE is processing the events (*Effective Time*) and where is in an idle state (*Idle Time*). With the *Baseline ECspec* the ALE remains $\approx 89\%$ of the *Event Cycle* period in an idle state while when configured with the *Half-period ECspec* this value decreases to $\approx 62\%$. This means that during the *Event Cycle* period the ALE module can be in an idle state during 9 seconds when configured with the *Baseline ECspec* while with the *Half-period ECspec* this value can last for 3 seconds.

Event Effective Time Breakdown. Figure 4.5 presents the time breakdown for the stages of the event processing pipeline. Figure 4.5a presents the how much time is spent in each phase of the pipeline when the ALE is configured with *Baseline ECspec* and in Figure 4.5b when it is configured with the *Half-*

period ECspec.



(a) Baseline Event Cycle: event pipeline breakdown.



(b) Half-period Event Cycle: event pipeline breakdown.

Figure 4.5: Cloud deployment: event processing pipeline breakdown.

Comparing the obtained results, it is possible to observe that time breakdown is evenly distributed between the *Upload* ($\approx 35\%$), *Filtering & Aggregation* ($\approx 30\%$) and *Response* ($\approx 34\%$) stages when the ALE module is configured with the *Baseline ECspec*, while the *Tag Processing* and *Report Creation* stages represents a small percentage of the total time, less than $\approx 1\%$. When the ALE is configured with the *Half-period ECspec*, the *Filtering & Aggregation* stage is the most time consuming, representing close to $\approx 73\%$ of the total time. As when configured with the *Baseline ECspec*, the *Upload* and *Response* stages presents similar results, respectively close to $\approx 11\%$ and $\approx 12\%$. Regarding the *Processing* stage, the time required to process the event data increased in a significant way, from less than $\approx 0.3\%$ to $\approx 3\%$. The *Report Creation* stage presented the same values from the *Baseline ECspec* configuration ($\approx 0.3\%$).

Experiment Results. In our scenario, that means the warehouse door only will open if the ALE module was configured with the *Half-period ECspec*, otherwise the robot will crash with the door.

4.3.1.2 Fog-based warehouse latency

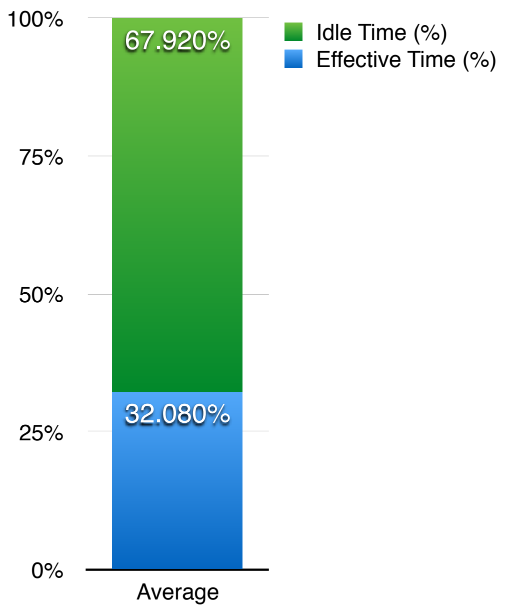
As in the previous experiment the event latency presented a better overall performance for the faster *ECspec*. According the metric values presented in Table 4.4 it is possible to observe that the event latency improves in a significant way - from 7.450s to 4.250s. This result is achieved thanks to the improvement in the latency of at the *Filtering & Aggregation Time* by $\approx 52\%$ - from 2.530s to 1.230s - and the amount of time that ALE is in an idle state - from 4.944s to 2.747s. Regarding the network latency, the values for the *Upload Time* and *Response Time* improved 1ms for both metrics.

However, when configured with a faster *Event Cycle* specification the tag processing time presented an inferior performance, where time to process the event data increases $\approx 470\%$ - from 0.049s to 0.279s. Also the report creation time increased 300% - from 0.001s to 0.003s.

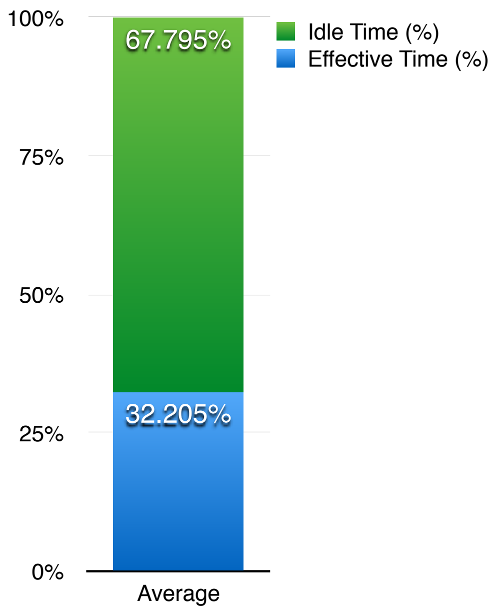
Metric	Base Event Cycle		Half-period Event Cycle	
	s	%	s	%
Upload Time	0.005	0.065	0.004	0.088
Tag Processing Time	0.049	0.519	0.279	5.743
Idle Processing Time	4.944	67.920	2.747	67.795
Filter & Aggregation Time	2.530	31.318	1.203	26.031
Report Creation Time	0.001	0.021	0.003	0.084
Response Time	0.010	0.157	0.009	0.258
Event Latency	7.540	100.00	4.245	100.00

Table 4.4: Fog deployment: performance metrics results.

Event Time Breakdown. Figure 4.6 summarizes the latency breakdown for an event that occurs in a smart warehouse deployed in the fog. Figure 4.6a shows the latency breakdown for an event when the ALE module is configured with the *Baseline ECspec* and in Figure 4.6b when it is configured with the *Half-period ECspec*.



(a) Baseline Event Cycle: latency breakdown.

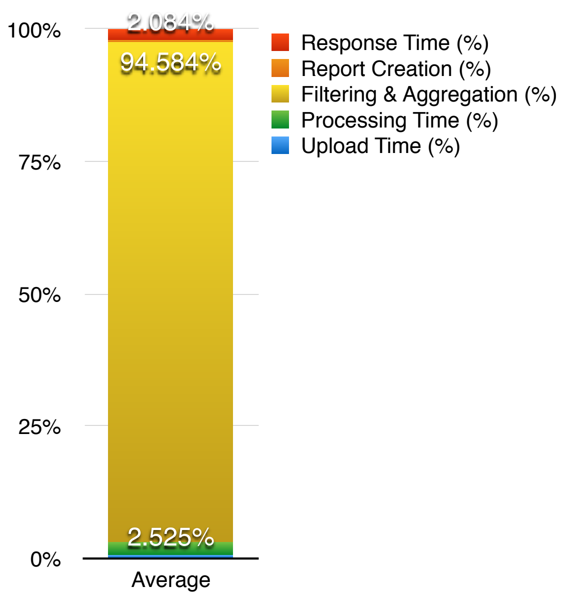


(b) Half-period Event Cycle: latency breakdown.

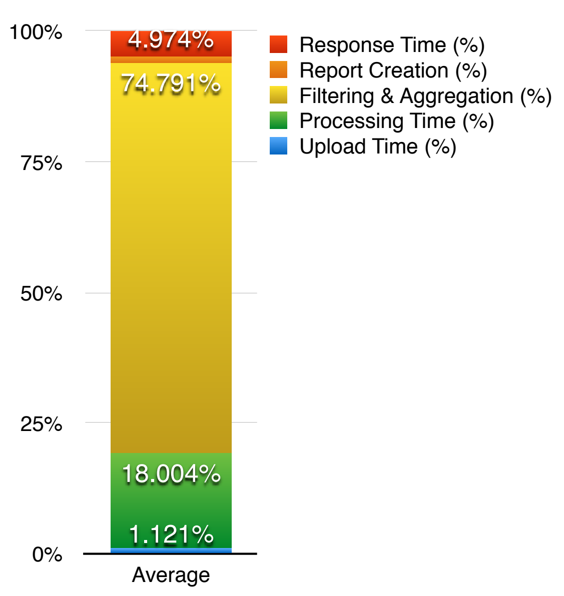
Figure 4.6: Fog deployment: Event Cycle latency breakdown.

Comparing the graphs for both *ECspecs* is possible to conclude that during most of the time of an *Event Cycle* the ALE module is in an idle state (*Idle Time*) - close to $\approx 68\%$ in both configurations - while in the remaining time the ALE is processing the event that was collected (*Effective Time*). Considering the duration of the *ECspecs* it means that in average when the ALE is configured with the *Baseline ECspec* the module can be in an idle state during 7s while with the *Half-period ECspec* this idle state can last for 3 seconds.

Event Effective Time Breakdown. Figure 4.7 summarizes how the time is spent during the stages of the event processing pipeline. Figure 4.7a presents the time breakdown for each stage of the pipeline when the ALE is configured with *Baseline ECspec* and in Figure 4.7b when it is configured with the *Half-period ECspec*.



(a) Baseline Event Cycle: event pipeline breakdown.



(b) Half-period Event Cycle: event pipeline breakdown.

Figure 4.7: Fog deployment: event processing pipeline breakdown.

It is possible to observe that the *Fltering & Aggregation* stage is the most time consuming for both *Event Cycle* specifications. With the *Baseline ECspec* this stage occupies close to $\approx 95\%$ of the total time, while with the *Half-period ECspec* this value is close to $\approx 75\%$. The reason for this difference is in the *Tag Processing* stage. With the *Baseline ECspec* the time for processing the event data represents close to $\approx 2.5\%$ of the total time while with the *Half-period ECspec* this value is close to $\approx 18\%$. The *Upload* and *Response* stages together represents a small percentage of the time spent to process the event - close to $\approx 5\%$ for both specifications - while the percentage of time to create the reports represents less than $\approx 1\%$ of the total time.

Experiment Results. As in the previous experiment, in our scenario the warehouse door only will open if the ALE module was configured with the *Half-period ECspec*, otherwise the robot will crash with the door.

4.3.2 Data Storage Performance

To evaluate the data storage performance for the Fosstrak middleware we use the data recorded with the Rec&Play module - which is able to record RFID sessions that stores the events occurred in the warehouse maintaining the order and time from the beginning of the session - were used as base to execute the tests. The sessions were recorded in a scenario developed by Correia et. al [32]. In this scenario tagged a robot is moving in a closed circuit where two RFID antennas were used facing each other, as illustrated in Figure 4.8.

As described in Section 4.1.2, the methodology consists in simulating a given number of readers that are sending events in the warehouse. This simulation was performed through JMeter[5], a Java application designed to perform load testing and measure application performance. In order to reproduce some situations that can occur in a real smart warehouse, we perform the following variations in the tests corresponding to the robot movements:

[5]`http://jmeter.apache.org/`

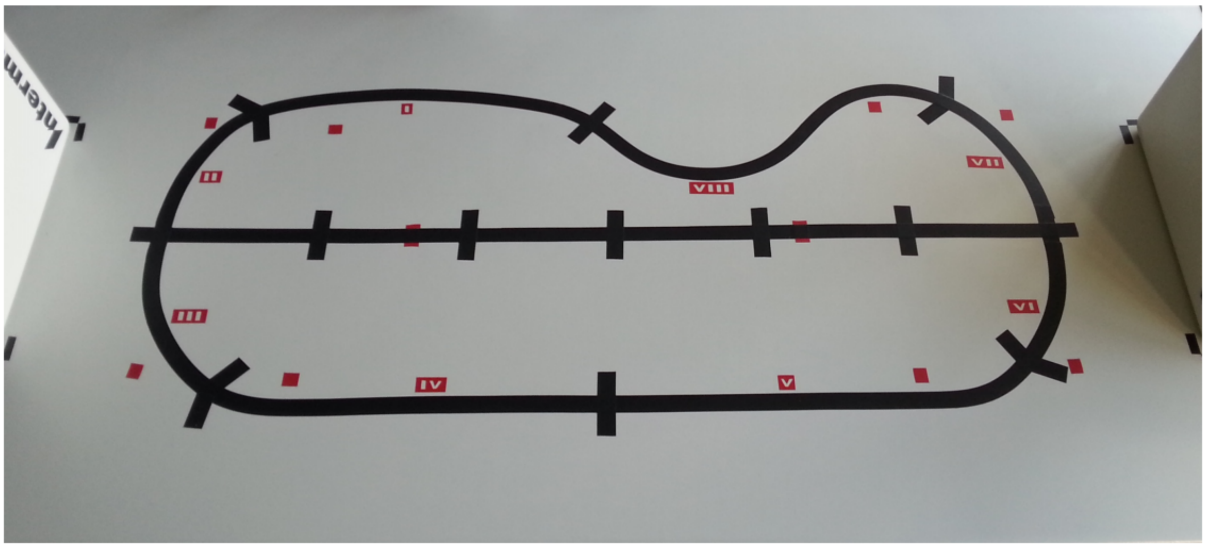


Figure 4.8: Robotic Warehouse demonstrator developed by Correia et. al [32].

- **Standard** The test is executed with the amount of events and period from the recorded session, corresponding to the robot moving at standard speed inside the warehouse.
- **Double of Requests** The test is executed with twice of events from the recorded session, corresponding to two robots moving at standard speed inside the warehouse.
- **Half of Interval** The test is executed with the amount of events and half of the period from the recorded session, corresponding to the robot moving at twice the speed inside the warehouse.

The evaluation was executed in two scenarios, **Product pick up run** and **3 product pick up run**. In the **Product pick up run** the robot picks the product, transport it to the destination and then returns to its original position. In the **Triple Product pick up run** the robot executes the same operation of the **Product pick up run** scenario, but in this the operation is repeated for 3 consecutive products. In both scenarios, we are simulating up to 5 readers sending events concurrently.

4.3.2.1 Product pick up run

In this scenario the session contains the data recorded based in the events generated during a product pick up run. In the current experiment each reader triggered 1593 events with a time period of 82ms between each event.

Figure 4.9 presents the system metric *CPU Utilization* for the current experiment. Comparing the values obtained in the experiment for the proposed variations, the *Standard* and *Double of Requests* variations the presents similar results - maximum CPU close to 14% - and its behavior tend to assume a linear pattern. For the *Half of Interval* variation, it is possible to observe that the metric value is always higher - maximum close to 16% - when compared with the other variations. The metric behavior changes according the number of threads that are sending events and tend to take a sinusoidal pattern.

The system metric *Network In*, presented in Figure 4.10, assumes a similar behavior of the previous metric. For the *Standard* and *Double of Requests* variations, they presented similar results - maximum close to ≈ 2.5 Megabyte (MB) - and its behavior tend to assume a linear pattern. It is possible to take

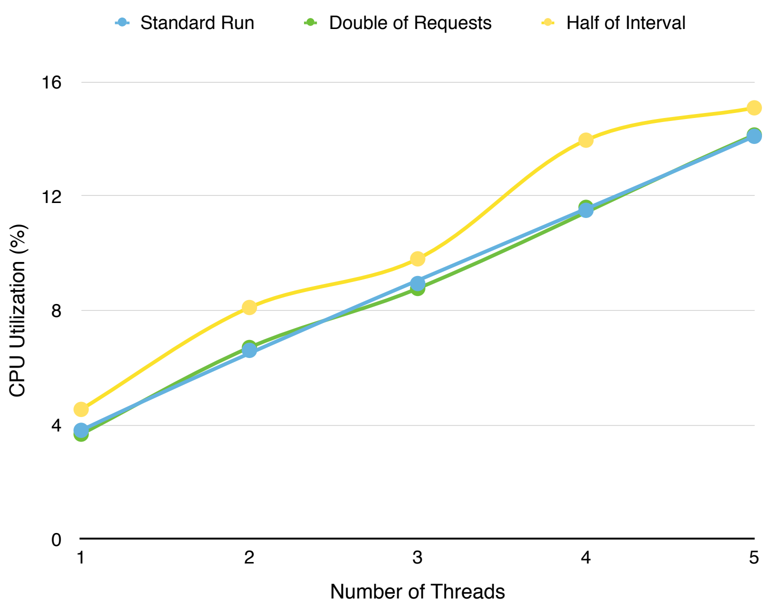


Figure 4.9: CPU Utilization performance results.

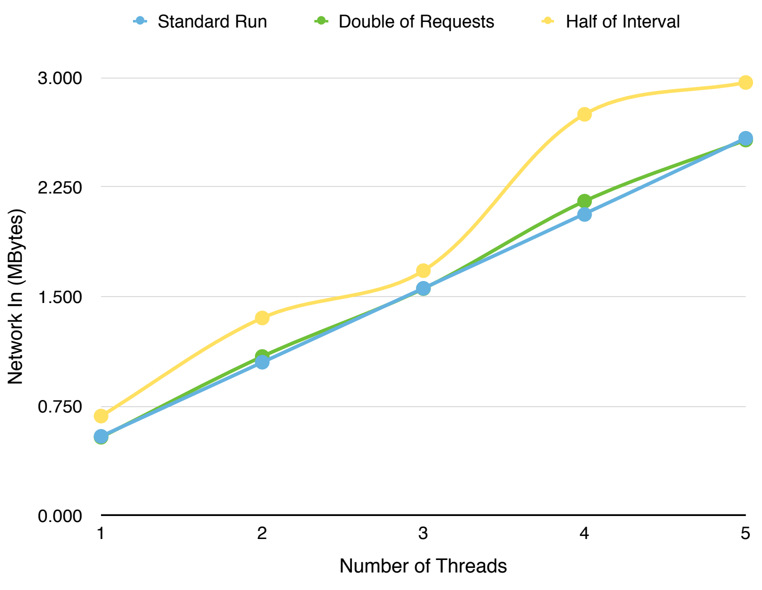


Figure 4.10: Network traffic performance results.

the same conclusions for the *Half of Interval* variation, where the results are always higher - maximum close to ≈ 3 MB - and the behavior tends to assume a sinusoidal pattern.

4.3.2.2 Triple product pick up runs

In this scenario the session contains the data recorded based in the events generated during 3 products pick up runs. In the current experiment each reader triggered 8895 events with a time period of 57ms between each event.

Figure 4.11 presents the system metric *CPU Utilization* for the current experiment. Comparing the

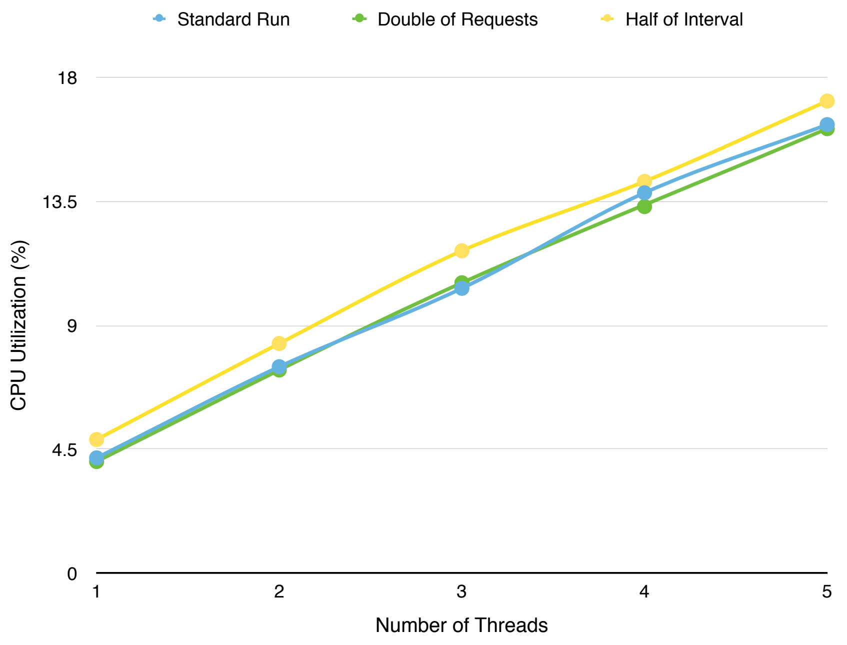


Figure 4.11: CPU Utilization performance results.

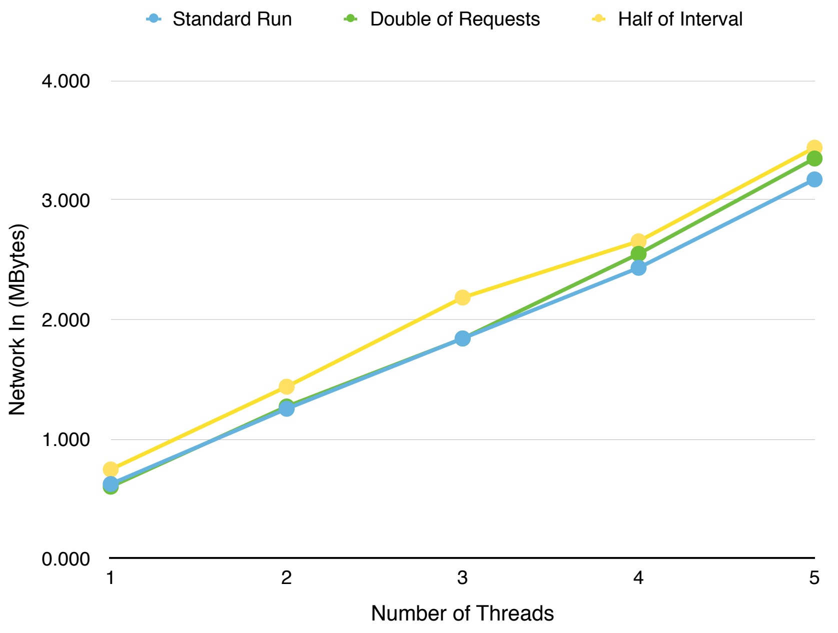


Figure 4.12: Network traffic performance results.

results obtained in the experiment, it is possible to conclude that they are very similar for all variations. The *Half of Interval* continues to present the highest values - maximum close to 18% - while the other variations presents almost identical results - maximum close to 16%. Regarding the metric behavior, all variations tends to assume a linear pattern.

As in the previous experiment, the system metric *Network In*, presented in Figure 4.12, is very similar to the previous regarding its values and behavior. The *Half of Interval* variation still presents the highest values - maximum close to ≈ 3.5 MB - but as the number of threads increase, it is possible to observe that the values of the *Double of Requests* variation tend to be close to the *Half of Interval*.

4.4 Results Analysis and Discussion

Here we present the result analysis and discussion based on the results obtained in the performed experiments.

4.4.1 Interaction Latency

Regarding the latency interaction, the results presented in Tables 4.3 and 4.4 show that the fog-approach presented a better overall performance than the cloud-based approach.

Event Latency. Regarding the results for the *Event Latency* metric, the fog-based deployment presented the best results. In order to meet the requirement *R1* (presented in 4.3), the event latency must be less than 5s. According the results it is possible to conclude that both approaches meet *R1*, but only when the ALE is configured with the *Half-period ECspec*.

Network Latency. The most relevant results are in network latency values for both approaches. The values for the *Upload* and *Reponse* metrics presented a substantial difference, where the fog-based approach is the best one. This allows to conclude that only a fog-based deployment is able to meet the requirement *R2*.

However, it is important to point that there some aspects that can improve the network latency of the cloud-based deployment. For instance, the network connection is a possible bottleneck for the performance of such approach. We believe that if the experiments were conducted through a faster network connection - e.g. a Fiber-optic connection - the overall performance of the cloud-based deployment will be better, but not as good as in the fog-based deployment.

Idle Time. Another important concern regards with the *ECspec* configuration. In the experiments performed for both approaches, we notice that the time where the ALE module remains in an idle state decreased significantly when the defined *ECspec* was configured with a smaller period. However, in all experiments performed the ALE spent more time in the idle state than processing the events. Based on these results we are able to conclude that none of the approaches meet the requirement *R3*.

Tag Data Processing. During the evaluation process we noticed a behavior in the performance of the Fosstrak platform. In the experiments performed for both fog-based and cloud-based approaches we observe that when the ALE module was configured with the *Half-period ECspec*, it takes more time to process the data from the RFID tags. The reason for this performance behavior is unknown and needs further investigation in order to determine what causes that performance issue.

Data Filtering and Aggregation. Another point observed at the performance of the Fosstrak platform regards about the performance of the *Filtering and Aggregation Time* metric. In the experiments performed for the fog-approach the value of the metric improved when the ALE was configured with the *Half-period ECspec*. However, in the experiments performed for the cloud-approach, the metric performance decrease. To determine what is the reason for this performance issue we need to perform more tests and verify if for faster *ECspecs* the performance of the *Filtering and Aggregation Time* continues to decrease.

4.4.2 Data Storage Performance

Regarding the data storage performance for the Fosstrak middleware, is possible to conclude that the metrics of *CPU Utilization* and *Network In* increase as the threads and requests are growing. It is important to point out that for this evaluation scenario, it is possible to conclude that the consumption of computing resources increases as faster the robot moves in the warehouse.

The obtained results in the performed experiments are consistent with the ones obtained by Gomes et al. [28], which evaluated the performance of the Fosstrak middlware. In the experiments performed Gomes detected that when the *CPU Utilization* crosses the value of 95%, the outbound traffic started to decrease. After analyzing the stored data, the conclusion was that the CPU exhaustion caused by the EPCIS affected the performance of ALE module - when executed in the same machine - resulting in a delay of data storage in the EPCIS repository, which explains the observed behavior.

According these results, is reasonable to assume that the performance of the data storage mechanism of the Fosstrak implementation can be a bottleneck. However, in the general case the overall performance is able to meet the requirements defined for our scenario (*R4*).

Chapter 5

Conclusion

The present work explored the deployment of IoT applications for smart warehouses based on the RFID technology with two different deployment approaches: one based in a traditional cloud deployment approach (cloud-based) and other according the Fog Computing platform (fog-based). More specifically, the present work focuses to determine if a cloud-based deployment is able to meet the low-latency requirements of many IoT applications, since that low-latency is an essential requirement of IoT applications. If a cloud deployment is not able to meet the network latency requirements for those applications, the cloud platform is not a viable option to perform the provisioning of IoT applications.

To improve the provisioning of RFID application middleware in the cloud, we developed a mechanism based on Docker containers and the Chef tool that automates the installation and configuration of the modules that composes the Fosstrak platform RFID middleware. This mechanism was of extreme importance, because it allowed us to perform the application provisioning of the cloud instances in a very efficient way. Although our experiments were conducted in a single cloud provider, the developed mechanism gave us the flexibility to choose between several cloud providers to provision the RFID middleware.

Regarding the system evaluation, we defined two methodologies for evaluate the latency of an event that occurs in the physical space and the data storage performance for the Fosstrak platform. With the methodologies proposed, we were able to compare the event latency performance for both cloud-based and fog-based deployments. We defined two experiments to evaluate the latency performance of the deployment approaches. The obtained results shows that the event latency performance presented better results when the application was deployed according a fog-based deployment. However, we identified some issues regarding the behavior of a Fosstrak module (ALE) that affected the performance of the event latency for both deployment approaches. Regarding the data storage performance of the RFID middleware, the results show that the Fosstrak platform is able to process with an acceptable performance the amount of data that is generated in a smart warehouse.

5.1 Contributions Summary

RFID Smart Place Deployment. A deployment approach based on the Cloud Computing platform for EPCGlobal compliant RFID middleware platforms. We propose an architecture that focuses to improve the network latency performance of RFID applications by distributing the middleware components across the fog and the cloud. A mechanism that automates the provisioning of RFID application middleware in the cloud. The provisioning mechanism allows to provisioning the Fosstrak modules in several cloud providers. Furthermore, it is based on Docker container images to provisioning the application stack, it is not specific for the Fosstrak middleware and can be extended for other EPCGlobal compliant RFID platform. To provisioning the Fosstrak software stack, we developed a set of Docker images that used to create the Docker containers with the Fosstrak modules, namely ALE, Capture Application, EP-CIS Repository and MySQL database. The images are open-source and available at Docker Hub. Our provisioning mechanism was implemented through the Chef configuration management tool. Since the resources for provisioning the stack using Chef does not exists, we defined a set of *recipes* and *roles* that allow to deploy and configure the Fosstrak software stack.

Interaction Latency Evaluation. Experiments were performed in order to find the best cloud-based deployment approach that meets the low-latency requirements of RFID applications. Moreover, we compared both cloud-based and fog-based approaches based on the *Event Cycle* metrics in order to determine how the deployment approach affects the performance of the *Event Cycle* stages.

5.2 Future Work

In the present work, we achieved our initial goals and determined that Utility Computing is adequate to deploy a smart place application based in RFID technology, both in cloud and fog deployments. However, our solution is not perfect and there some aspects that can be improved in the future.

Fog Implementation. Our solution proposes that the RFID application is deployed following a fog-based deployment. This means that we need to have a cloud close to the ground and this cloud must meet the same requirements of a remote cloud such as high scalability, security and multi-tenancy. Unfortunately, we were not able to implement a fog that meet these requirements and in our implementation the fog was built on top of a traditional Virtual Machine. In the future, the fog needs to be correctly implemented providing all the features of the remote cloud and in addition features such as location-awareness, mobility support and geo-distribution.

Containers Deployment. In the current implementation we used Docker containers to provisioning the Fosstrak software stack. In the evaluation of our solution we deploy the containers in a EC2 VM, which overlays two different mechanisms of virtualization. Although we still are able to take advantage of some benefits from the containers such as the portability, other benefits such as the low I/O and disk space are hidden by the VM hypervisor. A future improvement that can be made is to perform the deployment of the containers on top of the bare-metal or in a cloud-based container service - e.g. Google Kubernetes [1] or AWS EC2 Container Service[2] - in order to improve the overall performance of the solution.

Cloud Providers Evaluation. The evaluation was performed only in AWS EC2 instances. For the future is important to evaluate our solution in other cloud providers to compare which offers the best cost/performance relation.

[1] `http://kubernetes.io/`
[2] `https://aws.amazon.com/ecs/`

Latency Interaction Evaluation. To evaluate the latency interaction, we defined only two different *ECspecs*. For the future work, we want to evaluate the latency performance for our solution with *ECspecs* that presents smaller periods in order to determine which specification is more suitable for our solution.

Evaluation Scenario. In the evaluation scenario we used a virtual RFID reader instead of a physical one, which does not allow reproducing the environment conditions of a real smart warehouse such as interferences in the RFID tags antennas, network bandwidth variations, etc. However, in the evaluation experiments we used for some experiments traces from the work developed by Correia et. al [32], which have the real data traces mentioned above. A future improvement is to conduct the system evaluation in a real scenario in order to have more accurate results.

Multi Domain Evaluation In the present work, we confirmed that a cloud based deployment is adequate to support a smart warehouse application based in RFID technology that requires low latency interaction for both cloud and fog configurations. But it will be this approach the best choice for all application domains? Since that IoT covers several domains (as described in 2.1), in the future we want to perform a multi domain evaluation in order to determine if the Utility Computing is adequate to deploy the applications for these domains.

Smart Place Cost vs. Performance Analysys The Utility Computing allows to leverage the smart place infrastructure to the cloud, where the resources are available in a pay-as-you-use model. However, there is a trade-off between performance and costs. By leveraging the smart place infrastructure to the cloud, we can reduce the costs of the smart place operation, but the application performance can be compromised. For the future work, we want to perform an analysis to establish the relation between the performance and cost of a smart place regarding its deployment approach: cloud, fog and local. This analysis will allow to choose which is the most adequate approach to deploy an smart place based in the performance of the application and the costs of the smart place operation.

5.3 Cloud infrastructure for Smart Place applications

With this work we have contributed to the validation of the suitability of the Cloud for Things-based applications. We believe that using the cloud infrastructure to support smart place applications will be the most adopted approach, even for applications that have strict requirements such for low-latency and context-awareness. The flexibility provided by the cloud paradigm will allow that IoT applications from several domains may be deployed in a cloud infrastructure and take advantage of the benefits offered by this paradigm. However, reliable and fast network connections are a precondition. Another way to improve would be to have utility computing principles near the edge of the network, near the devices, as represented by the fog approach in this work.

Bibliography

[1] M. Weiser, "The computer for the 21st century," *Scientific american*, vol. 265, no. 3, pp. 94–104, 1991.

[2] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[3] F. Mattern and C. Floerkemeier, "From the internet of computers to the internet of things," in *From active data management to event-based systems and more*, pp. 242–259, Springer, 2010.

[4] R. Cáceres and A. Friday, "Ubicomp systems at 20: progress, opportunities, and challenges," *IEEE Pervasive Computing*, vol. 11, no. 1, pp. 14–21, 2012.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[6] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[7] C. Floerkemeier, C. Roduner, and M. Lampe, "Rfid application development with the accada middleware platform," *Systems Journal, IEEE*, vol. 1, no. 2, pp. 82–94, 2007.

[8] M. Eisenhauer, P. Rosengren, and P. Antolin, "Hydra: A development platform for integrating wireless devices and sensors into ambient intelligence systems," in *The Internet of Things*, pp. 367–373, Springer, 2010.

[9] L. M. S. De Souza, P. Spiess, D. Guinard, M. Köhler, S. Karnouskos, and D. Savio, "Socrades: A web service based shop floor integration infrastructure," in *The internet of things*, pp. 50–67, Springer, 2008.

[10] W. J. Doll and G. Torkzadeh, "The measurement of end-user computing satisfaction," *MIS quarterly*, pp. 259–274, 1988.

[11] R. Want, "An introduction to rfid technology," *Pervasive Computing, IEEE*, vol. 5, no. 1, pp. 25–33, 2006.

[12] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pp. 13–16, ACM, 2012.

[13] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of internet services and applications*, vol. 1, no. 1, pp. 7–18, 2010.

[14] J. Soldatos, M. Serrano, and M. Hauswirth, “Convergence of utility computing with the internet-of-things,” in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pp. 874–879, IEEE, 2012.

[15] S. Distefano, G. Merlino, and A. Puliafito, “Enabling the cloud of things,” in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pp. 858–863, IEEE, 2012.

[16] J. Zhou, T. Leppanen, E. Harjula, M. Ylianttila, T. Ojala, C. Yu, and H. Jin, “Cloudthings: A common architecture for integrating the internet of things with cloud computing,” in *Computer Supported Cooperative Work in Design (CSCWD), 2013 IEEE 17th International Conference on*, pp. 651–657, IEEE, 2013.

[17] F. Li, M. Vögler, M. Claeßens, and S. Dustdar, “Efficient and scalable IoT service delivery on cloud,” in *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pp. 740–747, IEEE, 2013.

[18] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens, “Quantifying the performance isolation properties of virtualization systems,” in *Proceedings of the 2007 workshop on Experimental computer science*, p. 6, ACM, 2007.

[19] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, “Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors,” in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 275–287, ACM, 2007.

[20] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A distributed storage system for structured data,” *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[21] A. Lakshman and P. Malik, “Cassandra: a decentralized structured storage system,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[22] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: amazon’s highly available key-value store,” in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 205–220, ACM, 2007.

[23] R. Cattell, “Scalable sql and nosql data stores,” *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2011.

[24] M. Armbrust, N. Lanham, S. Tu, A. Fox, M. Franklin, and D. A. Patterson, “Piql: A performance insightful query language for interactive applications,” in *First Annual ACM Symposium on Cloud Computing (SOCC)*, 2010.

[25] M. Ronstrom and L. Thalmann, “Mysql cluster architecture overview,” *MySQL Technical White Paper*, 2004.

[26] C. Bunch, N. Chohan, C. Krintz, J. Chohan, J. Kupferman, P. Lakhina, Y. Li, and Y. Nomura, “An evaluation of distributed datastores using the appscale cloud platform,” in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pp. 305–312, IEEE, 2010.

[27] M. Stonebraker and A. Weisberg, “The voltdb main memory dbms.,” *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 21–27, 2013.

[28] M. M. Gomes, R. d. R. Righi, and C. A. da Costa, “Future directions for providing better iot infrastructure,” in *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication*, pp. 51–54, ACM, 2014.

[29] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, “Fog computing: A platform for internet of things and analytics,” in *Big Data and Internet of Things: A Roadmap for Smart Environments*, pp. 169–186, Springer, 2014.

[30] D. Merkel, “Docker: lightweight linux containers for consistent development and deployment,” *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[31] N. Correia, M. L. Pardal, M. Romano, and J. A. Marques, “Rfid and arduino: Managing rfid events on a real world prototype,”

[32] N. Correia, “RFIDToys: A Flexible Testbed Framework for RFID Systems,” Master’s thesis, Instituto Superior Técnico, Portugal, 2014.