Big Data and Cloud Innovation (2019)

WHIOCE PUBLISHING PTE. LTD.

ORIGINAL RESEARCH ARTICLE

Toward using Cloud Elasticity on the Internet of Things Landscape

Rodrigo da Rosa Righi[1], Márcio Miguel Gomes[1], Cristiano Andrá da Costa[1], Helge Parzyjegla[2], Hans-Ulrich Heiss[3]

[1]Applied Computing Graduate Program, Universidade do Vale do Rio dos Sinos, Brazil, [2]Department of Informatics, Informatics Institute - Rostock University, Germany, [3]Communication and Operating Systems Group - TU Berlin, Germany

ABSTRACT

The digital universe is growing at significant rates in recent years. One of the main responsible for this sentence is the Internet of Things (IoT), or IoT, which requires a middleware that should be capable to handle this increase of data volume at real time. Particularly, data can arrive in the middleware in parallel as in terms of input data from radio-frequency identification readers as request-reply query operations from the user's side. Solutions modeled at software, hardware, and/or architecture levels present limitations to handle such load, facing the problem of scalability in the IoT scope. In this context, this article presents a model denoted Eliot - elasticity-driven IoT - which combines both cloud and high-performance computing to address the IoT scalability problem in a novel electronic product code (EPC) global-compliant architecture. Particularly, we keep the same application programming interface but offer an elastic EPC information services (EPCISs) component in the cloud, which is designed as a collection of virtual machines (VMs) that are allocated and deallocated on the fly in accordance with the system load. Based on the Eliot model, we developed a prototype that could run over any black box EPCglobal-compliant middleware. We selected the Fosstrak for this role, which is currently one of the most used IoT middlewares. Thus, the prototype acts as an upper layer over the Fosstrak to offer a better throughput and latency performances in an effortless way. The results are encouraging, where Eliot outperforms the non-elastic approach both in terms of response time and request throughput.

KEYWORDS: Internet of things, Cloud elasticity, Electronic product code global, Performance, Adaptivity, Electronic product code information services

***Received*:** September 27, 2018; ***Accepted*:** January 22, 2019; ***Published online*:** February 20, 2019

Citation: Rodrigo da Rosa Righi, *et al.* Towards using Cloud Elasticity on the Internet of Things Landscape. 2019; 1(1):1-20. DOI: 10.18063/bdci.v3i1.835

***Correspondence to:** Rodrigo da Rosa Righi, Applied Computing Graduate Program, Av. Unisinos 950, Postal Code 93000-750, São Leopoldo, RS, Brazil. Email: rrrighi@unisinos.br

1. Introduction

The number of digital devices is doubling in size every 2 years and may prove to be multiplied by 10 between the years 2013 and 2020. This means an increase from the current 4.4 trillion gigabytes of data to 44 trillion gigabytes in only 7 years[1]. This jump is expected thanks to the advent of the Internet of Things (IoT) which enables things to be connected anytime, anyplace, with anything, and anyone using the internet communication substrate [1,2].

Among different technologies such as near-field communication, ZigBee, and QR codes, the radio-frequency identification (RFID) has been seen as the most used communication standard to enable unique identification on objects

1 http://cloudtimes.org/2014/04/17/internet-of-things-will-multiply-the-digital-universedata-to-44-trillion-gbs-by-2020/.

or things. RFID works with a large interval of frequencies, ranging from 120 KHz to 3.1 GHz, consequently supporting different speeds and coverage radius so enabling a vast number of applications fields. Thus, the IoT ecosystem commonly consists of accessing applications, the RFID middleware, to process and store data captured from RFID tags, and the hardware itself with RFID-enabled sensors and tags [3].

As traditional curve of technological products, we can observe the lowering costs and the increasing sophistication in the production of RFID tags. This scenario has led to significant and renewed interest in this technology so causing the emergence of standards. In this scope, we can highlight the driving force from logistics and supply chain companies on adopting the electronic product code (EPC)global Class 1 Gen 2 standard [4-6], which provides the notion of EPC to unique identify a physical object stored in an RFID tag. Briefly, the main objective of the EPCglobal is to provide an architecture to collect vast amounts of raw data from a heterogeneous RFID environment, filter them, compile them into usable data structures, and send them to computational systems. To accomplish this, EPCglobal defines the following components [7]: (i) RFID readers (also denoted RFID sensors); (ii) application level events (ALEs), for filtering and collecting EPC data; (iii) EPC information services (EPCISs) to store EPC data, as well as to exchange this data along the EPCglobal network; (iv) EPC capturing applications, as a box in the middle between ALE and EPCIS, regulating how the former sends data to the last. Each company has its own set of components, so the idea is to generate value by providing a standard way of capturing data from objects along the partners involved in a particular application field (including, for example, suppliers, enterprises, resellers, clients, buildings, and users).

Figure 1 illustrates the EPCglobal architecture, emphasizing the hardware, the middleware, and the user application parts. According to the EPCglobal ideas, each company (member of a supply chain, for example) has an instantiation of the middleware, so it is possible to track data from any EPC object by looking up its location using the hierarchical-structured object naming service system. In addition, the components inside the middleware can be arranged arbitrarily among different servers in the local network, although the traditional deployment considers a single compute node for this purpose [2]. The deployment is particularly pertinent for performance purposes impacting mainly in two characteristics of the IoT systems [2,3,8,9]: (i) Real-time analysis of the huge amount of data that may be generated by sensors at any time; (ii) service level for time-bounded user applications that access and use processed RFID data.

At the EPCglobal terminology level, we need to properly deal with both the incoming data at the ALE software and the possible time penalties when a high traffic of requests accesses EPCIS or raw data from the ALE concomitantly. Thus, aligned with the performance issue, the modeling of a unique server to accommodate the components of an EPCglobal-compliant middleware clearly implies in scalability problems. Scalability can be seen as the ability of a system, network, or process, to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth [10]. Thus, the use of a centralized server incurs in two traditional performance problems: Limitation of processing capacity and network bottleneck (mainly when using transmission control protocol/internet protocol (TCP/IP) over Ethernet network substrates). Instead of using an underprovisioned infrastructure and then solving the scalability problem, one can take profit of a computational cluster which resources are commonly static allocated to handle sporadic peak loads properly. In this case, an information technology (IT) infrastructure becomes expensive resulting in low resource utilization and wastage of energy due to, in average, we would work with an overprovisioned

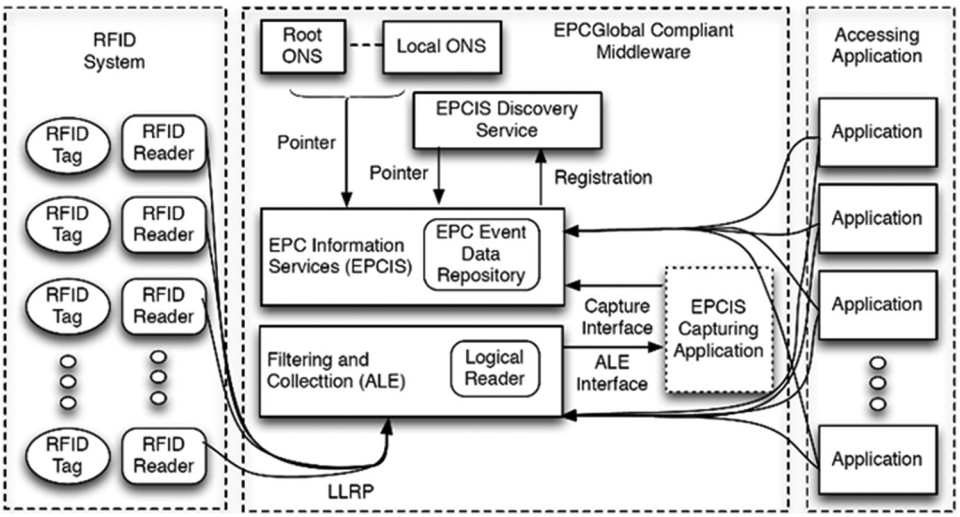

Figure 1: Electronic product code global architecture: Hardware, middleware, and application parts

resource infrastructure. An alternative to address the IoT deployment problem consists of exploring cloud computing, and more precisely, its concept of elasticity to manage the EPCglobal components. Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner such that at each point in time, the available resources match the current demand as closely as possible [11,12].

In this context, as an alternative to address the scale and dynamic IoT traffic at both RFID system and accessing application levels, we are proposing an EPCglobal-compliant architecture named Eliot[2] (elasticity-driven IoT). To the best of our knowledge, Eliot appears a pioneer model toward the joint analysis of cloud elasticity and high-performance computing techniques applied to the IoT panorama. The idea is to bring benefits both to the IoT platform administrators and IoT users. Besides an easier IoT deployment replication using VMs, the former group can save either energy, on private clouds, or budget, on pay-as-you-go driven public clouds so avoiding an overprovisioned resource infrastructure. Performance is the expected keyword to be delivered to end users since the system is capable of adaptations to accommodate resources in accordance with the incoming load. Eliot was modeled to answer the following problem statement:

- *How can we model an EPCglobal-compliant computational architecture and IoT algorithms to manage cloud elasticity in face of addressing the dynamic and scalable demands from user applications and RFID readers?*

To accomplish the aforesaid question, Eliot presents a manager in charge of controlling the horizontal elasticity using VM replication and lower and upper load thresholds. Since the EPCglobal interfaces were maintained, the on-demand resource reorganization is offered changing neither the hardware of readers nor any line of code in the users' applications. This article describes the Eliot's rationales, besides, a prototype that uses the Fosstrak[3] middleware as a black box counterpart and runs over the Amazon public cloud. Particularly, Fosstrak is one of the most used middlewares for IoT, being engaged on real cases in both industry and commerce areas [5,6]. The evaluation results are encouraging, where elasticity provided a better usage of network bandwidth, response time, and requests per second ratio when compared to the results obtained with the fixed deployment. In addition, we highlight the benefits on request timeout management when using an elastic-assisted IoT execution.

The remainder of this article will first introduce our work motivation in Section 2. Here, in brief, we will discuss about our previous work emphasizing conditions and deployments of ALE and EPCIS components to reach a scalability problem. Section 3 describes Eliot architecture in details, while Section 4 encompasses technical issues regarding prototype implementation. The evaluation setup and the discussion on the results are explained in Sections 5 and 6. Related work is addressed in Section 7 which presents a comparison table that considers the scalability thematic in IoT environments. Finally, Section 8 presents the final remarks, highlighting the scientific contribution to the state of the art and showing some directions of future work.

2. Motivation: Benchmarking IoT performance and scalability

We started our IoT scalability investigation in Gomes *et al.* [13], where we proposed a micro IoT benchmark named μIB. Using Fosstrak, μIB aimed at measuring the IoT scalability in a single server so presenting the first thoughts about what conditions could be interesting for IoT. Thus, μIB combined different configurations including the number of tags, readers, sequential requests at each thread, as well as the number of threads from an accessing application performing simultaneous access to ALE and EPCIS components. ALE works with raw data from readers, performing data filtering and collection, while EPCIS stores preprocessed data in a datastore and retrieves it when an incoming request arrives on its query interface. Both tags and readers were emulated using the Rifidi system. The idea was to observe eventual performance bottlenecks and requests timeouts when enlarging the number of threads and/or tags.

ALE module has indicated a limitation that could process up to only 200 requests simultaneously. If a client runs 201 or more threads requesting SOAP queries, a timeout occurs and the ALE component crashes. Even requests to EPCIS did not respond afterward, being necessary to restart Tomcat (the Fosstrak container) to provide a stable system again. Analyzing the response time and central processing unit (CPU) usage in Figure 2a, it is not possible to visualize a saturation tendency that would justify the limitation of 200 threads. Hence, it seems to be a Fosstrak implementation issue.

The EPCIS component, in turn, has a different behavior when compared to ALE. As depicted in Figure 2b, CPU usage, network traffic, and response time increase as the threads and requests are growing as well. Close to 90% of CPU load, we observe that the inclination of both the CPU and incoming network curves is not so aggressive. Particularly, the draw of the CPU curve presents the same behavior of the traditional network throughput, where the measures grow quickly and remain near to

2 https://github.com/eliot-project/eliot.
3 https://code.google.com/p/fosstrak/.

the limit of the technology. The most important fact is that the outbound network traffic starts to decrease suddenly when CPU usage crosses the value of 95%. This indicates a possible timeout and requests drop problem as increasing the number of threads.

Figure 3 illustrates an EPCIS test that helps us to explain the average behavior described earlier in Figure 2b. First, Figure 3 presents a peak on the inbound network traffic due to the accumulation of requests made by 512 threads at the same time and a strong decreasing afterward. The CPU usage reaches around 100% in few seconds, remaining close to this rate up to the end of the test. From 4 to 26 s, the outbound network traffic is higher than the inbound, indicating that the RFID tags sent by the Rifidi emulator are being captured by Fosstrak Capture Application properly, saved in the MySQL, and returned as response of the EPCIS queries. However, around 26 s, the outbound network traffic begins to decrease abruptly, keeping values below the inbound traffic. Meanwhile, the inbound traffic keeps a normal average

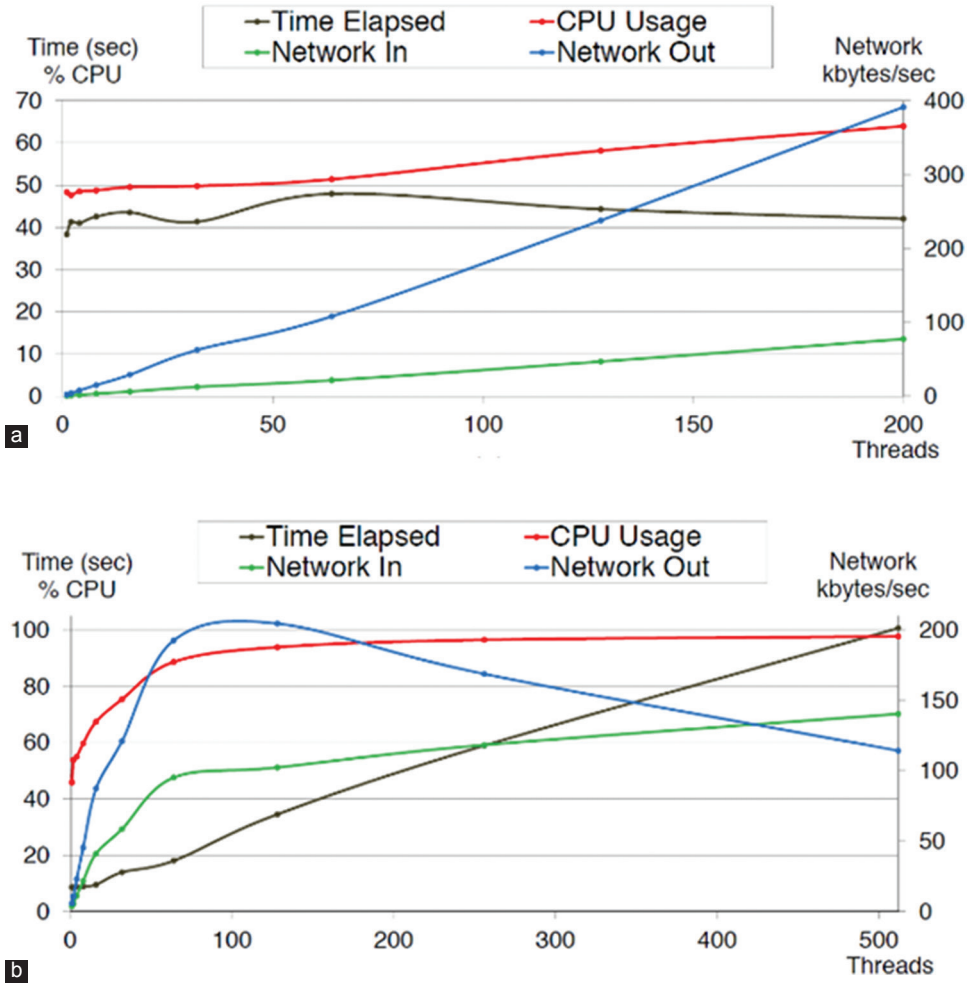

Figure 2: Average behavior of the evaluated components: (a) Application level events; (b) electronic product code information services. Both graphs were obtained using 16 tags and 16 requests per thread

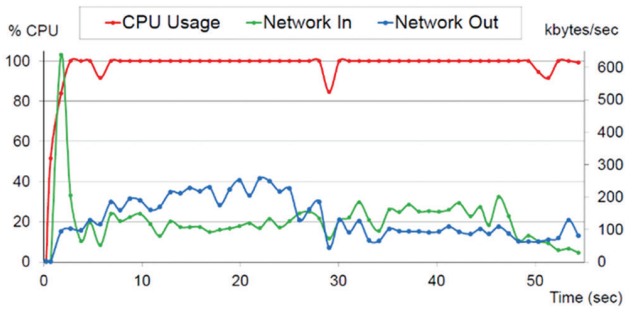

Figure 3: Electronic product code information service test with 512 threads, 8 requests per thread, and 16 tags

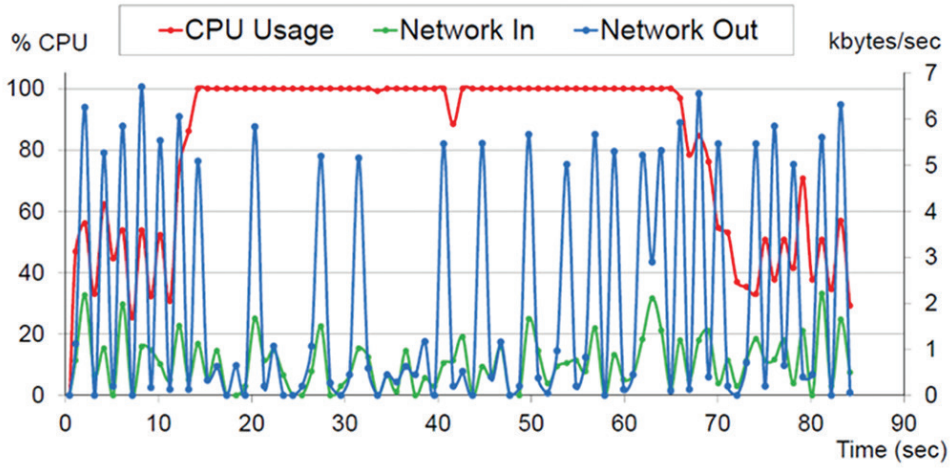

Figure 4: Execution of the application level events, where the requests come from a single thread. In parallel, the electronic product code information service is capturing requests from 512 threads

floating value. Considering the response of the queries, RFID data tags cannot be seen after this event. This can be explained analyzing the CPU, which became exhausted so not processing data sent by the Rifidi timely, resulting in a gap >10 s between the request and the response.

ALE generates data that were saved in the EPCIS component according to EPCglobal architecture. In this way, Figure 4 depicts the ALE response time while requesting EPCIS exhaustively. From 0 to 12 s, ALE is running normally, responding every 2 s (default Fosstrak parameter) in accordance with its cycle parameter. From 12 to 66 s, there are 512 threads making requests to the EPCIS. During this interval of time, CPU usage keeps almost constant around 100%, and the responses of ALE do not obey the 2 s of interval as seen earlier. It causes a cumulative delay, and RFID tags are saved in database with a high gap of time according to the system clock. It explains why outbound network traffic decreases significantly after crossing 25 s. Elapsed 66 s, the requests to EPCIS are concluded and the ALE response time returns to obey the 2 s interval.

In brief, our previous investigation [13] concluded the following gaps: (i) There is a technical limitation of parallel incoming requests, particularly observed for the ALE component; (ii) in EPCIS, the CPU load does quickly close to 100% when growing the number of parallel requests, emphasizing both its CPU dependency and a possible opportunity to develop distributed load balancing approaches; (iii) the most impressive observation was the performance degradation when stressing ALE and EPCIS concomitantly, delaying the response time of both components and so not meeting the requirements of time-constraint applications; and (iv) a high number of timeouts on query operations, particularly when employing 512 threads doing parallel requests to the middleware. Next section presents our approach to address the aforementioned gaps.

3. Eliot: Proposal of an elastic EPCglobal-compliant architecture for the IoT

This section describes Eliot - A novel proposal of an EPCglobal-compliant architecture that explores the concepts of resource elasticity from cloud computing and load balancing from high-performance computing. We modeled Eliot to use an existing EPCglobal-compliant middleware (addressed as a black box) so adding the incremental features but maintaining the standard EPCglobal application programming interface (API). In addition, we developed Eliot with the following design decision in mind: (i) Division of the EPCIS component in two subcomponents, each one created to answer distinct demands, i.e., internal demands derived from the ALE and external querying requests coming from user applications; (ii) horizontal elasticity at both EPCIS subcomponents, enabling the on the fly addition (scaling out operation) and reduction (consolidation or scaling in operation) of VMs when providing the IoT service; (iii) the model must be performance peak aware for not wasting time on unnecessary resource allocation and deallocation actions; (iv) to allow a quick deployment of the IoT infrastructure since its components will be encapsulates in VM templates so enabling the system replication on other companies/places easier.

3.1. Architecture

Figure 5 illustrates the Eliot architecture. Before introducing the cloud elasticity support, we will first discuss about two Eliot's recommendations: The first is the adoption of a distributed datastore to address the EPCIS repository and the second concerns a multicore machine to run the ALE component. The EPCglobal specification affirms that an

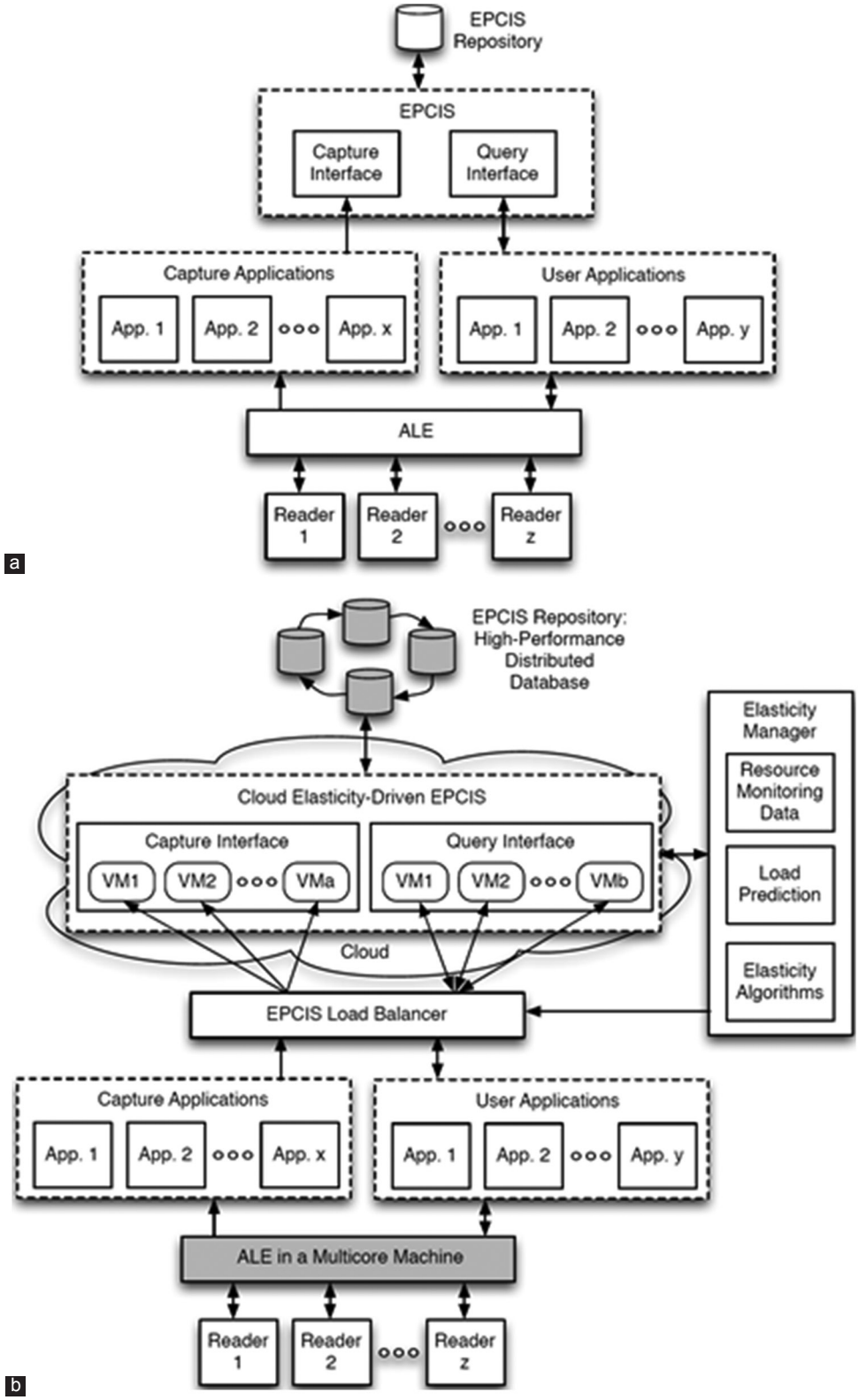

Figure 5: (a) Standard view of the electronic product code (EPC)global architecture; (b) Eliot EPCglobal-compliant architecture which offers cloud elasticity at EPC information service (EPCIS) level, not changing either capture or user applications. The transparency feature is enabled by the EPCIS load balancer, while the elasticity manager on the fly controls the number of capture and query virtual machines informing the former about monitoring data resource reorganization updates. Finally, the shading parts of the architecture refer to Eliot's recommendations to explore performance in an effortless way

implementation of the EPCIS is free both to use any internal representation for data and to implement the queries using any database or query technology, as long as the results seen by a client are consistent with EPCIS specification. In this way, the use of distributed and/or low latency-driven NoSQL databases that present automatic load balancing among the nodes represents an effortless way to improve I/O performance. Yet, it is not uncommon to observe the support of automatic data replication and high availability on such systems. Technically, even with the prefix "No" in the name, this kind of database normally offers an SQL-compatible API so expanding its use to legacy systems.

Although focusing on EPCIS, our study started by analyzing cloud elasticity in the scope of the ALE component. Each EPCglobal deployment has traditionally a single ALE, which is in charge of managing and exchanging data with physical readers. The standard communication protocol for this is the low-level reader protocol (LLRP), which is normally implemented using Sockets. LLRP establishes that the communication between the ALE and a reader must be preconfigured and must remain static while both sides are turned on.

This preconfiguration includes some properties of the reader, including its IP address and a communication port so establishing a fixed one-to-one (particular ALE to a particular reader) interaction style. In this way, this static configuration does not make viable the proposition of a dynamic load balancing at ALE level in a black box fashion, i.e., without reimplementing the ALE source code. Considering this, we are only proposing to execute ALE in a multicore machine to profit from a possible multithreaded implementation.

The standard definition of the EPCIS enables two communication interfaces for this component: (i) Capture interface, which stores information from the capturing applications, performing only write operations and (ii) query interface, in charge of enabling database queries which were requested by users or client applications so performing uniquely read operations. Considering this clear division on the provided services, Eliot architecture divides the EPCIS in subcomponents: One for the capture interface and another for the query interface. Each subcomponent was included in a VM template, which can be seen as a building block to instantiate or consolidate VMs in accordance with the elasticity algorithms. This decoupling decision turns the performance monitoring simpler, besides, aiding the selection of thresholds and the use of load balancing algorithms, particularly applied to each case.

The main Eliot's modules are the elasticity manager and the EPCIS load balancer. For the sake of simplicity, the names manager and load balancer will be used in the remaining of the manuscript. The manager is in charge of identifying overprovisioning and underprovisioning situations periodically, so deallocating or allocating VMs to offer resources as close as possible to the application needs. Eliot uses reactive and horizontal cloud elasticity, hiding from the user the management of both rules-conditions-actions statements and lower and upper thresholds to provide cloud elasticity. Besides a periodical resource monitoring, the manager uses time series to compute a load prediction (*lp*) in which will serve as input for the elasticity algorithms. The idea is to take decisions for the future not only observing a single value in the past but also a collection of them.

As detailed in Section 3.2, this strategy is useful to avoid elasticity actions when detecting either performance peaks or sudden drops.

Besides dispatching requests to VMs in accordance with their respective status, the main role of the load balancer is to offer transparency to capture and user applications on accessing the EPCIS component. A simple manner of providing this capability but not depending on network devices comprises the use of HTTP protocol both in terms of client and server. We modeled an HTTP server to receive requests and an HTTP client to dispatch them to the VMs in accordance with a load balancing policy. The manager updates the load balancer once the status of the VMs or the number of them suffers a change. Finally, Figure 5b depicts the manager and the load balancer outside the cloud, but they could be mapped inside the cloud without changing the Eliot's concepts. Particularly, the manager uses the API provided by the cloud middleware to perform both resource monitoring and VM provisioning (allocation and deallocation).

3.2. Elasticity metric and scaling in and out decisions

This subsection discusses about Eliot elasticity capacity, highlighting how monitoring data are organized and which type of data is captured. Eliot employs time series analysis over a collection of CPU load data to extract meaningful statistics and characteristics of the system. Particularly, we adopted the Weighted Moving Average (WMA) technique, also known as the Aging process, to avoid false-positive and false-negative elasticity actions. Basically, we are assigning a weight equal to 1/2 for the most recent load observation, dividing this by 2 at each subsequent element in the time series. For example, considering an upper threshold of 80% and observations such as 72, 70, 78, 71, and 81 (the most recent), the computed load will inform the value of 74.62 as the result of the following development: $\frac{81}{2}+\frac{74}{4}+\frac{78}{8}+\frac{70}{16}+\frac{72}{32}$. In this way, the last value 81 will not trigger elasticity action since it is evaluated as a false positive. Thus, our strategy can

amortize the importance of peaks since an erroneous allocation will not really be needed if we analyze the historical data of the IoT system.

Considering we are addressing EPCIS as two groups, at each monitoring period, we evaluate the need of elasticity actions on them separately. For the sake of simplicity, the remaining of this subsection will only discuss about query interface elasticity, but the same ideas are applied to the complementary interface. Periodically, the elasticity algorithm takes place and captures the value of load as presented in Equation 1. As depicted in Figure 5b, here, *b* represents the current number of VMs for addressing the query EPCIS interface. *l*(*j*) is an arithmetic average of all running VMs at monitoring index *j*, where *load*(*i*,*j*) means the CPU load of VM *i* at monitoring index *j*. Taking into account this value, Eliot computes the *lp* starting by the last observation *j*. *lp*(*j*) is a recurrent function that implements the Aging concept, working in accordance with a parameter denoted *window*. *window*, in turn, refers to the number of elements to be analyzed in the time series. Today, this parameter is fixed and informed at Eliot's compilation time, but we intend to explore it with adaptivity depending on the system stability as future work.

$$l(j)=\frac{1}{b}\cdot\sum_{i=1}^{b} load(i,j) \tag{1}$$

$$lp(j)=\begin{cases} \frac{1}{2}l(j) & if\ j=t-window+1 \\ \frac{1}{2}lp(j-1)+\frac{1}{2}l(j) & if\ j\neq t-window+1 \end{cases} \tag{2}$$

As a reactive elasticity controller, the manager operates with lower and upper thresholds to offer a simple strategy for scaling in and out operations. Both thresholds are set at prototype level, being informed, for instance, at development, or application launching time. First, if *lp*(*j*) is larger than the upper threshold, the manager instantiates a new VM. On the other hand, if *lp*(*j*) is shorter than the lower threshold, an existing VM is deallocated (always maintaining at least a single instance). Our elasticity modeling follows the standard way to provide this feature on Web applications [14]: There is a manager that handles incoming requests and spawns additional VMs if the volume of requests exceeds a given threshold, or yet, terminates VMs when demand goes down.

4. Prototype implementation

Aiming at evaluating the Eliot's algorithms, we developed a prototype in Java that runs over the Amazon public cloud. We are also using the code of the Fosstrak v.2.1.2 to enable the EPCglobal components, so Eliot implements the load balancer and the manager besides the creation of the cloud templates for the EPCIS VMs. Despite recommending a non-SQL datastore, this version of the prototype uses MySQL which is the standard option for EPCIS repository in the Fosstrak software package. Following we will discuss development details about the Eliot's components and how we modeled and implemented user applications for the tests.

4.1. Elasticity manager and load balancer

The manager acts in accordance with the push method to get and update monitoring data about the VMs. Hence, each VM contains a monitoring module that verifies periodically the CPU usage as well as the input/output network traffic. We established the period as 1 s, so after expiring it, each monitoring module sends data to the manager using the UDP transport protocol. At the manager side, the data are used to update a table with VM information, and then, the *lp* computation takes place to decide about any elasticity action (Subsection 3.2 for details). Technically, both allocation and deallocation procedures are triggered using API directives from the EC2 Amazon development package when either *lp* is shorter than the lower threshold or when *lp* is greater than the upper threshold.

This version of the prototype only implements a load balancer to handle the EPCIS query interface. The load balancer consists of a HTTP server (in Java: com.sun.net.httpserver.HttpServer), a HTTP client (in Java: org.apache.http. client. HttpClient), and a load balancing policy. Hence, the HTTP server offers a unified SOAP interface to receive requests from clients, redirecting an incoming call to a target VM using the HTTP client, and a load balancing policy. Today, the load balancer implements the Round-Robin policy to dispatch the entering demands to VMs. This strategy was selected because the VM template (including data and setup) is the same among the instances, besides, the fast scheduling calculus associated to the Round-Robin strategy.

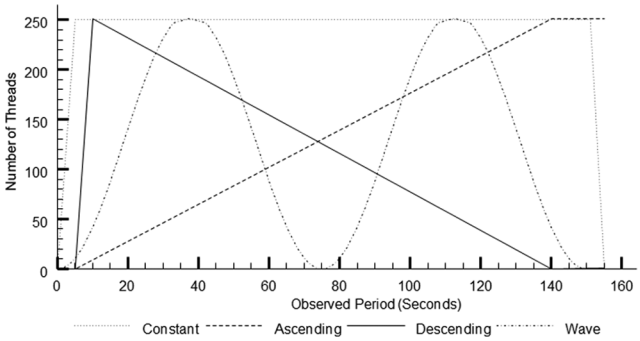

Figure 6: Four different load patterns that drive the input workload, i.e., the simultaneous requests in accordance with the number of active threads

4.2. User application

We developed a synthetic application that has the following characteristics:

(i) It launches a set of threads, which perform simultaneous queries to the EPCIS query interface and (ii) we are evaluating different load patterns to observe the elasticity behavior under distinct input workloads. Concerning the topic (i), the application starts launching up to 512 threads that will sleep automatically, waking up in accordance with the load pattern. Particularly, we implemented the constant, ascending, descending, and wave load patterns as illustrated in Figure 6. Hence, the difference among the profiles concerns the number of concurrent threads triggering EPCIS queries at the same time. In the constant profile, for example, the application quickly uses the maximum number of threads, maintaining this number up to near the end of the execution. The descending load pattern is characterized by a sudden increase on the number of threads, decreasing this value up to 1 when arriving the end of the application. Finally, the wave explores a sinusoid function with two periods so reaching twice the peak of threads.

5. Evaluation methodology

This section presents the evaluation methodology, mainly discussing about the assessed deployments and observed metrics. First, considering Equation 2, we are applying the aging concept with a window equal to 7, so $\frac{1}{2^7}$ is equal to 0.0078 implying that we are attributing a weight lower than 1% for the last element in the time series.

Aiming at observing the possible gains involving the EPCglobal components decoupling and the cloud elasticity, we are considering three IoT infrastructure deployments for the tests:

i. Centralized: This deployments concern the use of a single server and, consequently, not offering elasticity. This deployment does not use any Eliot software, but only the Fosstrak ALE and EPCIS components executing in a unique machine.
ii. Distributed, but not elastic: Here, we are using Eliot but not allowing any elasticity action. Eliot was configured to execute with a lower threshold equal to 0 and an upper threshold equal to 100. The *lp* in Equation 2 will never exceed these values and, consequently, elasticity actions will never take place.
iii. Distributed, with elasticity support: This deployment represents our main focus of attention since we expect to observe if elasticity is or not really beneficial for performance and under which conditions this happens. To accomplish this, the values of 30% and 70% are loaded for the lower and upper thresholds, respectively. According to Dawoud *et al.* [15], these values are representative to test reactive, and, in particular, horizontal, elastic systems.

All deployments were executed in the EC2 Amazon cloud. Particularly, Figure 7 illustrates the deployments ii and iii. Both start with a single VM for the EPCIS query interface, but the last deployment can expand this number up to 5. The comparison between deployments i and ii is pertinent to observe the impact of decoupling the EPCglobal components in different machines. Our previous tests (Section 2) revealed that the adoption of a single compute resource is highly prejudicial for the performance of an EPCglobal system. In addition, comparisons considering the deployments i and iii, and ii and iii, are useful to evaluate the Eliot's elasticity algorithm, its *lp* function, besides, performance issues considering latency and throughput measures.

For each combination of load pattern (in the application side) and deployment (in the IoT infrastructure side), we are generating graphs to observe the response time along the execution, the number of simultaneous requests (threads), the

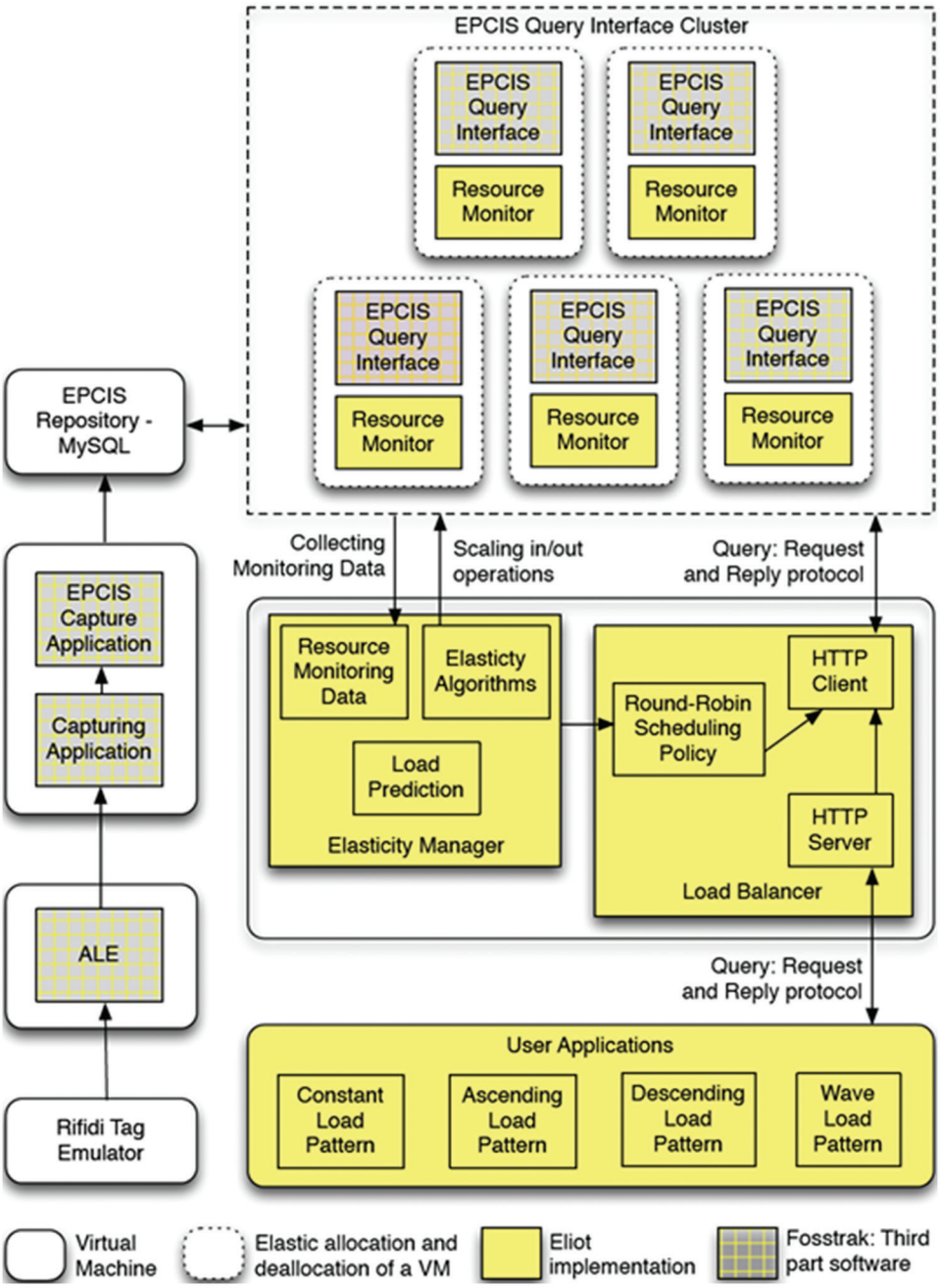

Figure 7: Deployments ii (distributed, but not elastic) and iii (distributed, with elasticity support) in the EC2 Amazon cloud. Both start with a single virtual machine to handle the electronic product code information services query interface. In addition deployment iii can extend the number of 1 virtual machine up to 5

value of the *lp* (Equation 2), and the output network throughput. The horizontal axis is expressed in seconds denoting the elapsed execution time, while the vertical axis was converted to present normalized percentage values, so all metrics can be plotted in the same graph. Thus, our idea in this point focuses mainly on verifying possible relation among the collected metrics. In addition to the graphs, each aforesaid combination also reports data about the average behavior as follows:

- Mean network output throughput in k bytes per second: Informs the average volume of data that were dispatched by the load balancer;
- Mean response time per request in milliseconds: This index expresses the average time between the sending of requests to the load balancer and the return of the responses;
- System throughput as an average of the requests per second: This index represents the total number of requests divided by the total execution time of the test.

6. Experimental results

Here, we are presenting the performance results and the behavior of each observed metric in subsection 6.

6.1. Performance and metrics behavior when varying the load patterns and IoT deployments

Figure 8 depicts the behavior of 256 requests in a constant fashion. For the non-distributed and distributed but not elastic deployments, parts (a) and (b) of the figure, the measures remain constant without major fluctuations. On the

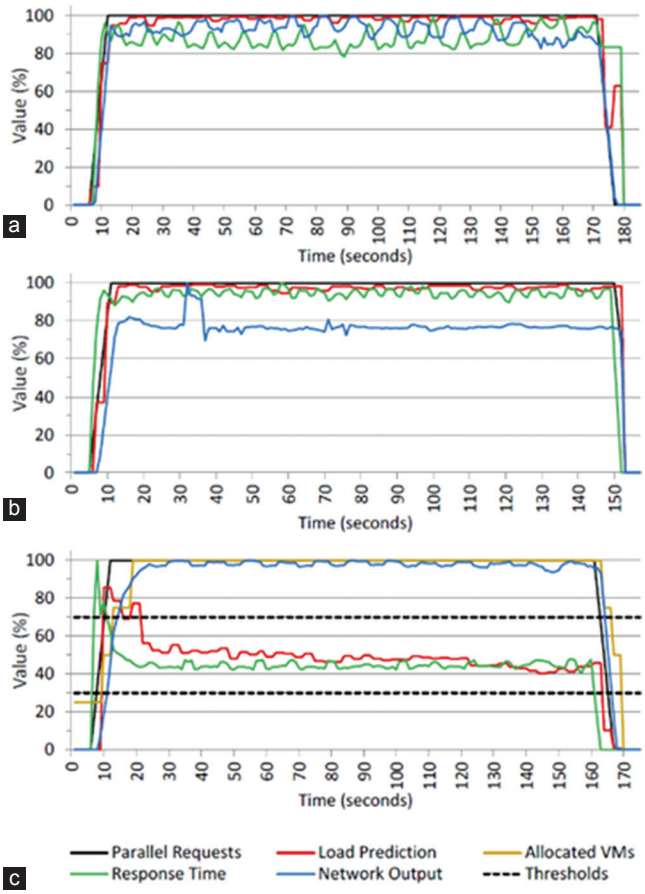

Figure 8: Results when executing the constant function on three IoT deployments: (a) Centralized; (b) distributed, but not elastic; and (c) distributed, with elasticity support. The observed metrics were normalized to be plotted in the same graph

Table 1: Mean *lp* and mean CPU usage on each verified load pattern

Load patterns and measures	IoT deployments		
	(i) Centralized	(ii) Distributed not elastic	(iii) Distributed and elastic
Constant			
Mean amount of used CPUs for EPCIS	1.0	1.0	3.9
Mean *lp* (%)	96.3	96.3	50.3
Ascending			
Mean amount of used CPUs for EPCIS	1.0	1.0	4.2
Mean *lp* (%)	89.2	87.5	53.3
Descending			
Mean amount of used CPUs for EPCIS	1.0	1.0	2.9
Mean *lp* (%)	87.0	75.3	53.2
Wave			
Mean amount of used CPUs for EPCIS	1.0	1.0	2.7
Mean *lp*p (%)	85.5	79.8	50.8

CPUs: Central processing units

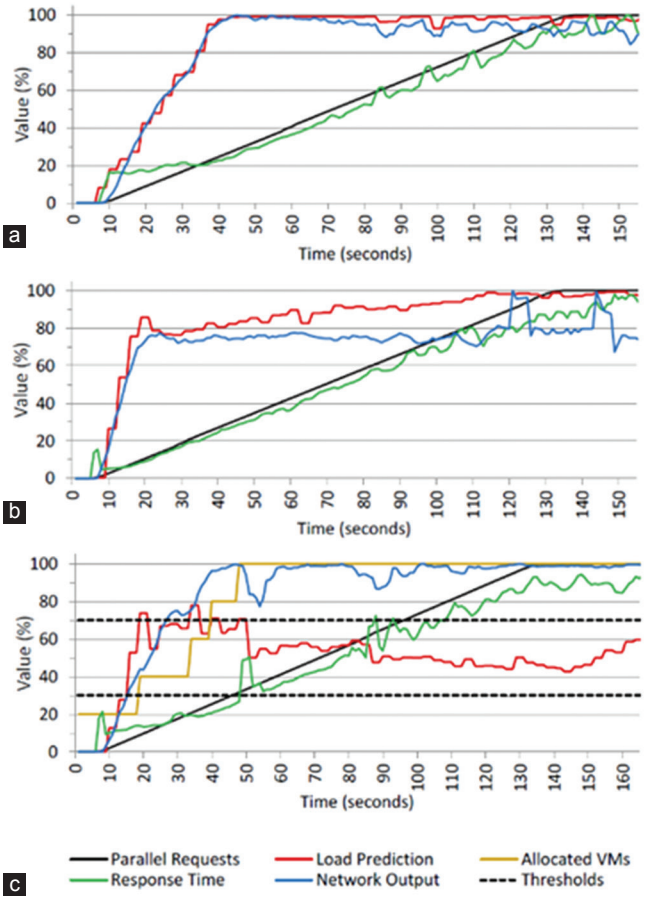

Figure 9: Results when executing the ascending function on three IoT deployments: (a) Centralized; (b) distributed, but not elastic; and (c) distributed, with elasticity support

other hand, we can observe two behavior patterns at the elastic configuration: First, the response time of the requests is directly related to the *lp*; second, the output network traffic is directly related to the number of allocated VMs. Besides, in Figure 8c, we identify that the elasticity is really obeying the employed thresholds since scaling in and out operations take place when the *lp* exceeds the lower and upper bounds. Figure 9 illustrates the graphs for the ascending load pattern, which ends with 256 threads performing simultaneous requests. In (a) and (b), we can visualize that the response time is proportional to the number of simultaneous requests. Contrary to (a) and (b), in part (c), we emphasize the fact that the output network traffic comes off in relation to the *lp* curve, accompanying the tendency of increasing the number of allocates VMs. In this way, *lp* measure becomes independent, not influencing the other measures.

Figure 10 presents the results for the descending load pattern, which jumps quickly for 256 requests and decreases this number as the application advances as well. Again, the response time is proportional to the number of requests. However, the configuration that comprises a distributed and elastic architecture presented an output network traffic that comes away from the *lp* curve, now following the number of allocated VMs. We can highlight that the number of allocated VMs remains high while the number of parallel requests decreases. This occurs thanks to the value of 30% for the lower threshold, which is only achieved at the end of the execution. Therefore, a possible improvement concerns the use of adaptable thresholds to avoid resource wasting, as presented in part (c) of Figure 10. Figure 11 presents the behavior of the wave load pattern. Both parts (a) and (b) of this figure show that the number of parallel requests dictates the behavior of the other curves. In part (c), we observe that the resource allocation is in accordance with the number of requests, being allocated in the ascending part and deallocated in the descending one. Particularly, the second period presents an underprovisioned situation since the *lp* does not exceed the upper threshold.

Table 1 and Figure 12 present information related to averages. We emphasize the reduction on the response time in about 47% when comparing the distributed scenario with and without elasticity support. In addition, corroborating with our previous work [13] (discussed in Section 2), 63.5% of gain was observed when comparing deployments ii against i. Thus, the fact of separating database, EPCIS and ALE in different machines, is beneficial to avoid bottlenecks and

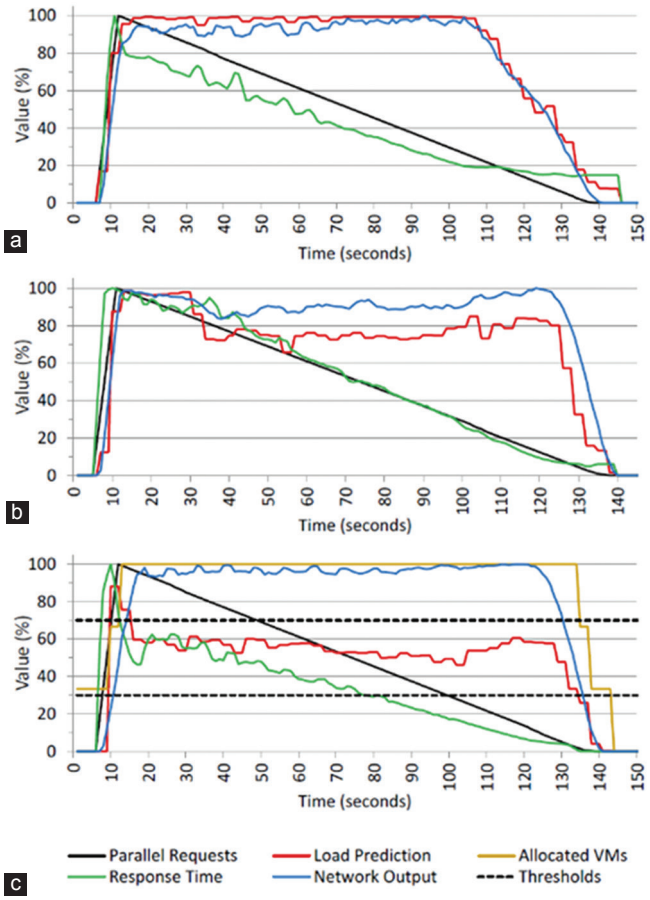

Figure 10: Results when executing the descending function on three IoT deployments: (a) Centralized; (b) distributed, but not elastic; and (c) distributed, with elasticity support

performance penalties related to interferences among the EPCglobal components when at least one of them is overloaded. The ascending function presented a larger consumption of CPUs, maintaining five EPCIS query interface VMs for more than a half of the execution. This measure is an average of execution time in seconds and consumed CPU on each single second (our monitoring period equal to 1 s was previous explained in Section 4).

In a general perspective, considering the four evaluated load patterns, the response time for the queries is deeply related to the number of parallel requests. In addition, the larger the number of requests, the higher the use of CPU. Besides data processing, the communication among the EPCglobal components occurs over TCP for reliability purposes.

Particularly, this transport protocol performs intermediary memory data copies, checksum computation and maintains timers for data retransmission and flow control [16,17]. All these procedures implemented in software and executed in the kernel of the operating system, so spending cycles of the CPU that hosts the operating system. In other words, the standard network communication protocol relies on CPU penalties to offer a reliable end-to-end communication semantics for end users. In this way, the larger the number of parallel requests, the higher the network flow and, consequently, the CPU usage as well. Moreover, we concluded that the output network traffic is directly influenced by the *lp* when analyzing both the centralized and the distributed, but not elastic architectures. Nevertheless, the network flow in the distributed and elastic configuration advances close to the number of allocated VMs for handling EPCIS query interface requests.

6.2. Observing the aging concept behavior

Instead of only allocating a new VM when the instantaneous use of the CPU exceeds the upper thresholds, Eliot uses the Aging technique for load forecast and peak smoothing. Figure 13 explores the benefits of using this technique over an execution that implements the descending load function. After crossing 100 s from the start of the test, we have allocates three EPCIS query interface VMs. In this moment, 70 simultaneous requests are being performed with a mean response time of 100 ms. After this moment, the use of the CPU grows up, and the *lp* increases as well. Close to 110 s,

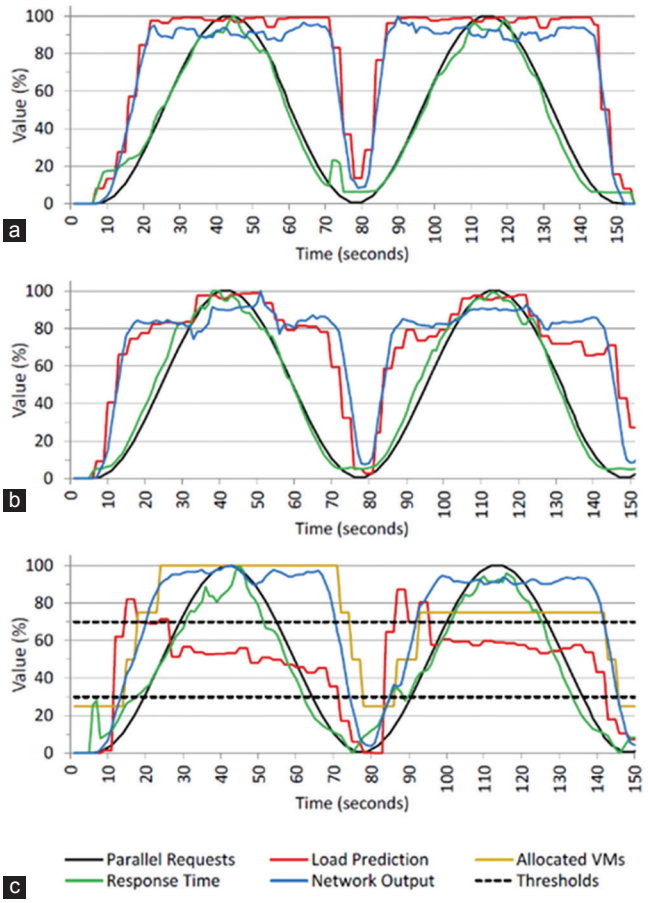

Figure 11: Results when executing the wave function on three IoT deployments: (a) Centralized; (b) distributed, but not elastic; (c) distributed, with elasticity support.

the CPU use exceeds the upper threshold; however, the *lp* (as defined in Equation 2) does not surpass this upper limit. We are using the aging concept for the *lp* computation, which employs a weighted moving average over a time series composed by a set of CPU load observations. Hence, the analysis of not only one but also several points incurs on not performing scaling out operations in the mentioned moment. In addition, we can observe that the load decreases in the next steps, showing the correctness of the Eliot's procedure. After 130 s, deallocations take place since the *lp* is below than the lower threshold.

6.3. Possible improvements on elasticity management

We are using only Aging technique to decide about elasticity actions, where the most recent observation receives the weight ½, the second observation is multiplied by ¼, and so on. At present, this technique is employed uniquely over the CPU metric; more precisely, each load observation takes into account an arithmetic average of the CPU load of each VM that is able to receive EPCIS queries. This subsection starts by analyzing the possible impacts of adding the response time as a decision factor together with the aging-based *lp*. Thus, the response time could enter in Eliot's algorithm to indicate a quality of service or QoS. The addition of a single VM normally implies on response time reduction, but the obtained time could not be enough to maintain a quality of service. In this way, at each scaling out action, it is possible to check the new response time and allocate more than one VM whether this metric remains above a predefined QoS. The complementary situation is also truth: Although not reaching yet the lower threshold in accordance with the aging-based *lp*, if the response time could be satisfied with a fewer number of VMs, then, we can anticipate the scaling in action.

Besides the use of the response time, other possible improvement concerns the use of adaptable thresholds. The current version of Eliot's prototype uses fixed values for the lower and upper load thresholds; however, the use of adjustable limits can imply on both better system reactivity and energy saving. First, in an application that presents a long but not steep load increase, we can improve reactiveness by reducing the upper threshold to execute scaling out

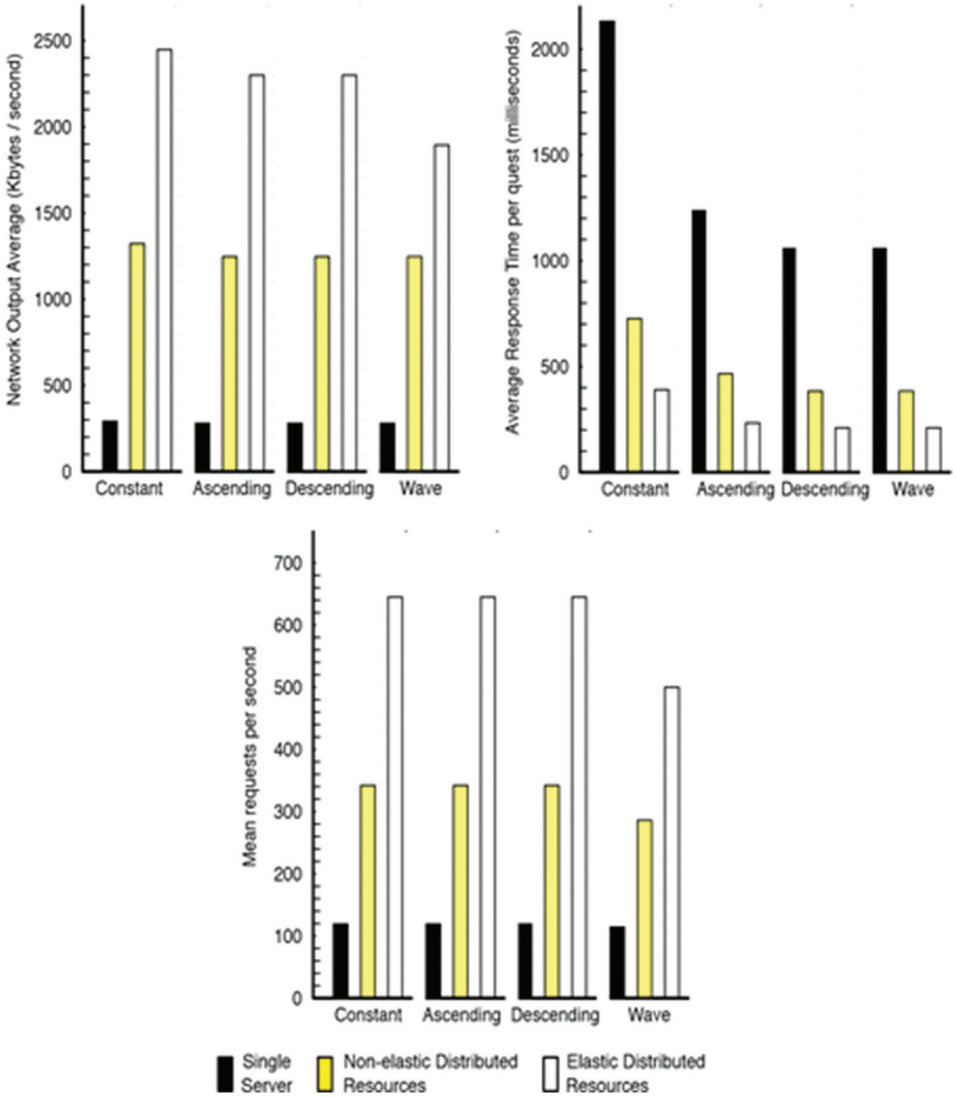

Figure 12: Average measures collected in the tests: Mean network output throughput in k bytes per second; mean response time per request in milliseconds; and system throughput as an average of the requests per second

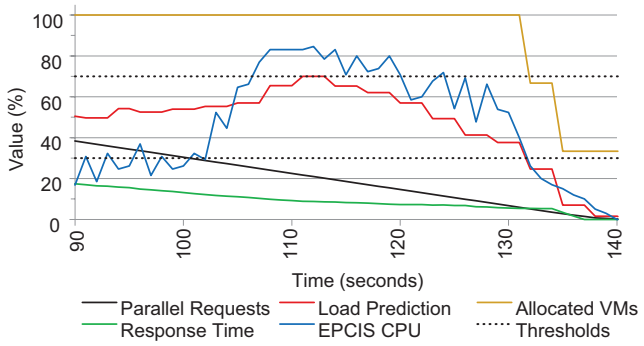

Figure 13: Comparison involving the aging-based load prediction (*lp*) and the central processing unit electronic product code information service curve: Even this last exceeding the upper threshold, *lp* avoids elasticity actions due to understands a peak on application load

operations sooner. On the other hand, we can elevate the value of the lower threshold to anticipate VM consolidation, helping on energy and budget savings. Particularly, this last idea could be applied in the scope of Figure 10c, which shows a situation in which the load is decreasing, but the resources remain allocated to the end of the execution.

Table 2: Comparison of RFID middlewares

MARM [18]	Multiagent systems	Not addressed
Fosstrak [5]	Separated server, simulation mode, or embedded on RFID reader	Subscription of readers
WinRFD [19]	Distributed middleware modules	Not addressed
Hybrid [20]	Multiring P2P network	P2P systems
RF²ID [21]	Virtual paths between virtual readers and physical readers	Path management
LIT [22]	Common reader management interface	State-based execution model

RFID: Radio-frequency identification, P2P: Peer to peer, RF²ID: Reliable framework for radio-frequency identification, LIT: Logistics information technology, MARM: Multiagent-based radio-frequency identification middleware

7. Related work

This section first presents commercial and open-source systems available today to implement an IoT infrastructure [5,18-23]. After that, considering the scalability keyword, we can classify the IoT academic research initiatives in the following groups: (i) Non-EPCglobal compliant [24-31] and (ii) extensions or performance-driven implementations for the EPCglobal standard [32-37]. Thus, Subsections 7.2 and 7.3 discuss academic initiatives in details.

7.1. Commercial and open-source systems

The main purpose of an RFID middleware is to collect large amounts of raw data from a heterogeneous RFID environment, filter them, compile them into usable format, and offer them to enterprise systems. According to Li [38], the development of an RFID-enabled application with real-time data processing is a challenge effort in many ways: Architecture efficiency to process high volumes of data intelligently and in real time; scalability according to business demands; and compliance with ongoing standards. Particularly, scalability and load balance are indispensable characteristics when enabling high-performance RFID middlewares. In this way, Table 2 summarizes the efforts of commercial and open-source systems and middlewares that address these topics. These systems refer to projects available in the Internet for download.

Multiagent-based RFID middleware [18] was designed using the concept of agent-oriented software engineering.

Its architecture is divided mainly in three layers: Device management, data management, and interface. Fosstrak [5] is an EPCglobal middleware that includes reader interface, filtering and collection component (ALE), and EPCIS. Particularly, an EPCIS provides a standard interface for access and persists EPC data, allowing business events and making them available to the application.

WinRFID [19] is a multilayered middleware developed using .NET framework. It focuses on infrastructure scalability and administration, event and data intelligent process and dispatching, enterprise application, and business partner integration.

Hybrid middleware [20] is a middleware based on group communication in peer-to-peer (P2P) networks. It is purely designed for an electronic parking management system. Other work is the reliable framework for radio-frequency identification [21], which was designed to offer reliability, load balancing, high throughput, and scalability. Logistics IT (LIT) [22] is a middleware designed using the concepts of ALE and EPCIS. ALE architecture in LIT consists of four layers: Application, state-based execution, continuous query, and reader.

7.2. Non-EPCglobal-compliant initiatives

Li *et al.* [24] argued the idea of using cloud to centralize the entire IoT infrastructure of an enterprise that presents several branches. Thus, services such as domain mediation, application context management, billing, and metering can be done easier, generating knowledge over enterprise's data. Im *et al.* [25] explored scalability by proposing a new mashup (MaaS) service model, called IoT MaaS as a service, defined as composition of thing model, software model, and computation resource model. In this way, cloud computing is used to host the IoTMaaS instance, including a service driver, interface adapter, processing engine, web service substrates, and protocols to different sources of data.

Biswas and Giaffreda [26] investigate the idea of using IoT-centric cloud as a paradigm that extends cloud computing and services to the edge of the network, close to objects. The idea is to distribute data to move it closer to the end users to eliminate latency, reduces high traffic, numerous hops, and supports mobile computing and data streaming. To accomplish

this, they proposed the idea of local and global clouds. Local cloud maintains the infrastructure layer, while the global cloud is responsible for data movements and knowledge extraction. Nastic *et al.* [27] presented an experimental system named Patricia, which comprises both a framework and a novel programming model for IoT applications on cloud platforms. Its core idea is to enable the development of value-added IoT applications, which are executed and provisioned on cloud platforms but leverage data from different sensor devices and enable timely propagation of decisions to the edge of the infrastructure. Although presenting an Execution Manager responsible for elasticity, any elasticity algorithm or idea was proposed in the article.

Sarkar *et al.* [29] presented a research initiative called distributed internet-like architecture for things (DIAT), which is a simple, scalable, accommodates heterogeneous objects, and also allows interoperability among these heterogeneous objects. They defined a cognitive entity, called observer, that plays a key role in automated machine-to-machine communication to provide a service intelligently. Several cognitive functions such as dynamic service creation, modeling, and execution are incorporated in the proposed architecture. DIAT and Patricia are non-EPCglobal-compliant proposals that do not explore cloud elasticity. A scalable IoT service search is proposed in Ben Fredj *et al.* [30]. To overcome scalability issues, the authors use Semantic Web Technologies in conjunction with hierarchical distributed processing and mechanisms to reduce the searching cost such as clustering and aggregation. Athreya *et al.* [31] proposed a framework for self-managing devices, comprising measurement-based learning and adaptation to changing system context and application demands. Particularly, they developed a generalized optimization framework that allows software agents to manage and control protocol parameters and behaviors.

7.3. Extensions or performance-driven implementations for the EPCglobal standard

Guinard *et al.* [32] defended that deploying and maintaining IoT systems are time consuming. In this way, they affirm that cloud computing simplifies the deployment and maintenance of the EPC software stack and contributes to a wider adoption of the EPC Network standards and tools. Particularly, focusing on performance, EPCglobal initiatives can be divided in accordance with the component addressed in the standard: (i) Tag [33]; (ii) discovery service [34,35]; (iii) EPCIS [36,39]; and (iv) ALE [37].

- Tag: Wireless identification and sensor data management middleware is a system to support ubiquitous computing with passive sensor-enabled RFID [33]. By extending the EPCglobal standard, they work at the tag level, working with sensor tags data streams that store an EPC and sensor data. As Eliot, the prototype also uses Fosstrak but extends it to work with sensor-enabled RFID.
- Discovery service and ONS: Tieyan and Deang proposed the use of P2P to develop an efficient, fault tolerant, and scalable IoT discovery service [34]. The protocol relies on a novel P2P discovery service for a reader to retrieve RFID tag information from EPCglobal network. Li *et al.* [35] proposed a new central indexing discovery service system using distributed NOSQL database HBase to better support big data and parallel processing. The new storage schema uses object ID as row key, event timestamp as column identifier, and event index content as cell value.
- EPCIS: Li *et al.* [36] proposed an extension of the EPCIS data model to reduce event data volume on a large scale by extracting common information of event data into one configuration file. Itsuki and Fujita [39] are using P2P to organize the EPCIS repository. Particularly, the EPC of a product and the product information as server content are hashed and the hash values are registered to other servers as a new node in the network. Again, we have the problem of maintaining either an underprovisioned or overprovisioned infrastructure.
- ALE: Schmidt *et al.* [37] also exploited the use of P2P technology, now in the scope of the ALE EPCglobal component. ALE may be connected to several hundreds of readers. Thus, they propose an efficient way to solve this problem based on a distributed hash table; more precisely, a mechanism to distribute the ALE using chord, a well-known P2P lookup system, and being transparent for business applications. Any ALE is translated into a node of the P2P system. When receiving specifications, it has to split and distribute it to other involved ALEs and merge all reports locally before sending the final report to the business application. Besides not presenting elasticity (the solution works with a fixed number of ALE nodes), we can also envision the following problem: Although redistributing the requests, we still have a one-to-one connection between the ALE node and reader, so the network can be a bottleneck in this case.

8. Conclusion

As argued by Yin *et al.* [40], data obtained at scale can bring intelligence to user applications since existing adequate communication and data capturing and filtering substrates. Together with this affirmation, we also observe that is common

tech trend presenting the exponential growth of IoT from today up to 2020[3]. In this context, this article described Eliot: An elastic-driven IoT architecture responsible to distribute EPCglobal components in the cloud in accordance with the dynamic requests from readers and user applications. The elasticity facility is offered in an effortless way to IoT users, who do not need to change any line of code in their applications. In addition, at the IoT infrastructure perspective, the administrator can use any EPCglobal-compliant middleware as a black box, needing only to install the elasticity manager and load balancer components from Eliot, besides, creating VM templates to support both the EPCIS query and capture interfaces. At the scientific contribution viewpoint, we emphasize the use of the aging concept in the Eliot's architecture to address scaling in and out operations so avoiding possible VM allocation oscillations and VM thrashing.

The results were conducted over three deployments: (i) Single server with Fosstrak; (ii) EPCIS, EPCIS repository and ALE components distributed in different machines, but not using elasticity; and (iii) distributed configuration as mentioned in ii, but here with elasticity support. Network operations use the TCP protocol, which is processed at software level in the OS kernel, so the components of this protocol overload a single CPU when stressed with multiple network connections. Hence, the simple fact of spreading the EPCglobal components in a distributed fashion presents a significant improvement. Deployment iii is pertinent to support input load variations, outperforming deployment ii in the response time, network throughput, and requests throughput metrics. Besides performance benefits, cloud elasticity can yield significant cost savings when compared to the traditional approach of maintaining an overprovisioned infrastructure modeled to support demand peaks. Moreover, the use of VM templates to accommodate an EPCglobal component is pertinent for replicating the IoT among different companies: Prepare VMs once, run anywhere.

Future work includes the tuning of the Eliot's elasticity algorithm by adding the metric response time, besides, the CPU load currently used. Furthermore, we think to explore the idea of adaptable lower and upper thresholds to improve system reactiveness and energy savings. Finally, we plan to work with real trace data on the next Eliot's evaluation. Particularly, we intend to perform agreements with global courier delivery services companies, such as Fedex and DHL, to obtain their log file of a single day of work.

9. Acknowledgment

This work was partially supported by the following Brazilian agencies: CNPq, FAPERGS, and CAPES.

3 http://www.gartner.com/technology/research/top-10-technology-trends/.

References

1. Gubbi J, Buyya R, Marusic S, *et al.*, 2013, Internet of things (IOT): A vision, architectural elements, and future directions. *Future Gen Comput Syst*, 29(7): 1645–1660. https://doi.org/10.1016/j.future.2013.01.010.
2. Guo J, Zhang H, Sun Y, *et al.*, 2014, Square-root unscented kalman filtering based localization and tracking in the internet of things. *Pers Ubiquitous Comput*, 18(4): 987–996. https://doi:10.1007/s00779-013-0713-8.
3. Leung J, Cheung W, Chu S C, 2014, Aligning rfid applications with supply chain strategies. *Inf Manag,* 51(2): 260–269. http://dx.doi.org/10.1016/j.im.2013.11.010.
4. Kang M, Kim D H, 2011, A real-time distributed architecture for rfid push service in large-scale epcglobal networks. In: Kim T H, Adeli H, Cho H S, *et al.*, editors, *Grid and Distributed Computing*. Vol. 261. *Communications in Computer and Information Science*. Berlin Heidelberg: Springer. pp489–495.
5. Jose J I, Pastor J M, Zangroniz R, *et al.*, 2013, *Rfid Tracking for Urban Transportation Using Epcglobal-Based Web Services.* Washington, DC, USA: Proceedings of the 2013 27th International Conference on Advanced Information Networking and Applications Workshops, WAINA '13, IEEE Computer Society. pp1295–1300. http://dx.doi.org/10.1109/WAINA.2013.65.
6. Schapranow M P., 2013, *Real-time Security Extensions for EPC Global Networks: Case Study for the Pharmaceutical Industry*. Springer Publishing Company, Incorporated.
7. Ranasinghe D C, Harrison M, Cole P H, 2008, EPC network architecture. In: Cole P H, Ranasinghe D C, editors. *Networked RFID Systems and Lightweight Cryptography*. Berlin Heidelberg: Springer. pp59–78. http://doi:10.1007/978-3-540-71641-9_4.
8. Hodges S, Taylor S, Villar N, *et al.*, 2013, Prototyping connected devices for the internet of things. *Computer*, 46(2)L 26–34. http://doi:10.1109/MC.2012.394.
9. Miorandi D, Sicari S, Pellegrini F D, *et al.*, 2012, Internet of things: Vision, applications and research challenges. *Ad Hoc Netw.,*

10(7): 1497–1516. http://doi:10.1016/j.adhoc.2012.02.016.

10. Coulouris J, Dollimore T. Kindberg G. *Blair, Distributed Systems: Concepts and Design*. 5th ed. Boston, MA, USA: Addison-Wesley.
11. Heinze T, Pappalardo V, Jerzak Z, *et al.*, 2014, *Auto-Scaling Techniques for Elastic Data Stream Processing.* New York, NY, USA: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, ACM. pp318–321. http://doi.acm.org/10.1145/2611286.2611314.
12. Bersani M M, Bianculli D, Dustdar S, *et al.*, 2014, *Towards the Formalization of Properties of Cloud-based Elastic Systems*. New York: Proceedings of the 6th International Workshop on Principles of Engineering Service-Oriented and Cloud Systems, PESOS, ACM. http://doi.acm.org/10.1145/2593793.2593798.
13. Gomes M M, Righi R D, da Costa C A, 2014, *Future Directions For Providing Better IOT Infrastructure.* New York, NY, USA: Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication, UbiComp '14 Adjunct, ACM, 2014. http://doi.acm.org/10.1145/2638728.2638752.
14. Omote Y, Shinagawa T, Kato K, 2015, *Improving Agility and Elasticity in Bare-Metal Clouds.* New York, NY, USA: Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, ACM, 2015. pp145–159. http://doi:10.1145/2694344.2694349.
15. Dawoud W, Takouna I, Meinel C., 2011, Elastic VM for cloud resources provisioning optimization. In: Abraham A, Lloret J Mauri J, *et al.*, editors. *Advances in Computing and Communications*, Vol. 190 of *Communications in Computer and Information Science*. Berlin Heidelberg: Springer. pp431–445. http://doi:10.1007/978-3-642-22709-7_43.
16. Clark D, Jacobson V, Romkey J, *et al.*, 2002, An analysis of TCP processing overhead. *Commun Magazine IEEE,* 40(5): 94–101. http://doi: 10.1109/MCOM.2002.1006979.
17. Nguyen B, Banerjee A, Gopalakrishnan V, *et al.*, 2014, *Towards Understanding TCP Performance On LTE/EPC Mobile Networks,* New York, NY, USA: Proceedings of the 4th Workshop on All Things Cellular: Operations, Applications, & Challenges, All Things Cellular '14, ACM. pp41–46. http://doi:10.1145/2627585.2627594.
18. Massawe L V, Aghdasi F, Kinyua J, 2009, *The Development of a Multi-agent Based Middleware For RFID Asset Management System Using The Passi Methodology.* Washington, DC, USA: Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations, ITNG '09, IEEE Computer Society. pp1042–1048. http://doi:10.1109/ITNG.2009. 230.
19. Prabhu B S, Su X, Ramamurthy H, *et al.*, 2005, *Win RFID a Middleware for the Enablement of Radio Frequency Identification (RFID) Based Applications*. In: In Mobile, Wireless and Sensor Networks: Technology, Applications and Future, John Wiley and Sons, Inc. pp331–336.
20. Cervantes L F, Lee Y S, Yang H, *et al.*, 2007, *A Hybrid Middleware for RFID Based Parking Management System Using Group Communication in Overlay Networks.* Washington, DC, USA: Proceedings of the The 2007 International Conference on Intelligent Pervasive Computing, IPC '07, IEEE Computer Society. pp521–526. http://doi:10.1109/IPC.2007.12.
21. Ahmed N, Kumar R, French R S, *et al.*, 2007, *RFID: A Reliable Middleware Framework for RFID Deployment*. In: IPDPS, IEEE. pp1–10.
22. Kabir A, Hong B, Ryu W, *et al.*, 2008, Lit middleware: Design and implementation of Rfid middleware based on the Epc network architecture. In: Kreowski H J, Scholz-Reiter B, Haasis H D, editors. Dynamics in Logistics. Berlin, Heidelberg: Springer. pp221–229.
23. Solanas A, Domingo-Ferrer J, Mart´ınez-Ballest´e A*, et al.*, 2007, A distributed architecture for scalable private RFID tag identification. *Comput Netw*, 51(9): 2268–2279. http://doi:10.1016/j.comnet.2007.01.012.
24. Li F, Voegler M, Claessens M, *et al.*, 2013, *Efficient and Scalable IOT Service Delivery on Cloud.* Washington, DC, USA: Proceedings of the 2013 IEEE 6th International Conference on Cloud Computing, CLOUD '13, IEEE Computer Society. pp740–747. http://dx.doi.org/10.1109/CLOUD.2013.64.
25. Im J, Kim S, Kim D., 2013, *IOT Mashup as a Service: Cloud-based Mashup Service for the Internet of Things*. In: Services Computing (SCC), 2013 IEEE International Conference, 2013. pp462–469. http://doi:10.1109/SCC. 2013.68.
26. Biswas A, Giaffreda R, 2014, IOT and Cloud Convergence: Opportunities and Challenges. In: Internet of Things (WF-IOT), 2014 IEEE World Forum on. pp375–376. http://doi:10.1109/WF-IoT.2014.6803194.
27. Nastic S, Sehic S, Vogler M, *et al.*, 2013, Patricia – A Novel Programming Model for IOT Applications on Cloud Platforms. In: Service Oriented Computing and Applications (SOCA), 2013 IEEE 6th International Conference on, 2013. pp53–60. http://

doi:10.1109/SOCA.2013.48.

28. Sarkar C, Nambi S, Prasad R, *et al.*, 2014, *A Scalable Distributed Architecture Towards Unifying IOT Applications*. In: Internet of Things (WF-IOT), 2014 IEEE World Forum on, 2014. pp508–513. http://doi:10.1109/WF-IoT. 2014.6803220.
29. Sarkar C, Uttama A, Nambi S N, *et al.*, 2015, Diat: A scalable distributed architecture for IOT. *Internet Things J*, 99: 1. http://doi:10.1109/JIOT.2014.2387155.
30. Ben Fredj S, Boussard M, Kofman D, *et al.*, 2013, *A Scalable IOT Service Search Based on Clustering and Aggregation*. in: Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing, 2013. pp403–410. http://doi:10.1109/GreenCom-iThings-CPSCom.2013.86.
31. Athreya A P, DeBruhl B, Tague P, 2013, *Designing for Self-configuration and Self-adaptation in the Internet of Things*, In: Collaborate Com'13. pp585–592.
32. Guinard D, Floerkemeier C, Sarma S, 2011, *Cloud Computing, Rest and Mashups to Simplify RFID Application Development and Deployment.* New York, NY, USA: Proceedings of the 2nd International Workshop on Web of Things, WoT '11, ACM, 2011. pp91–96. http://doi:10.1145/1993966. 1993979.
33. Wickramasinghe A, Ranasinghe D, 2014, *A Sample, Wind Ware: Supporting Ubiquitous Computing with Passive Sensor Enabled RFID*. In: RFID (IEEE RFID), 2014 IEEE International Conference on, 2014. pp31–38. http://doi: 10.1109/RFID.2014.6810709.
34. Li T, Deng R, 2008, *Scalable RFID Authentication and Discovery in EPC Global Network*. In: Communications and Networking in China. ChinaCom 2008. 3rd International Conference. pp1138–1142. http://doi:10. 1109/CHINACOM.2008.4685227.
35. Li M, Zhu Z, Chen G., 2013, *A Scalable and High-efficiency Discovery Service using a New storage*. In: Computer Software and Applications Conference (COMPSAC), IEEE 37th Annual. pp754–759. http://doi:10.1109/COMPSAC.2013.125.
36. Li J, Liu S, Wang D, 2013, *An Extensible EPCIS Data Model*. In: Conference Anthology, IEEE. pp1–6. http://doi:10.1109/ANTHOLOGY.2013.6784744.
37. Schmidt L, Mitton N, Simplot-Ryl D, *et al.*, 2011, DHT-Based Distributed ale Engine in RFID Middleware. In: RFID-Technologies and Applications (RFID-TA), IEEE International Conference. pp319–326. http://doi:10.1109/RFID-TA.2011.6068656.
38. Li C M, 2006, *An Integrated Software Platform for RFID-Enabled Application Development*. In: Sensor Networks, Ubiquitous, and Trustworthy Computing, 2006. IEEE International Conference. Vol. 1. p4. http://doi:10.1109/SUTC.2006.1636197.
39. Itsuki R, Fujita A, 2009, Consideration for Efficient RFID Information Retrieval in Traceability System. Piscataway, NJ, USA: Proceedings of the 14th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA'09, IEEE Press. pp429–432. http://dl.acm.org/citation.cfm?id=1740954.1741015.
40. Yin H, Jiang Y, Lin C, *et al.*, 2014, Big data: Transforming the design philosophy of future internet. *Network*, 28(4): 14–19. http://doi:10.1109/MNET.2014.6863126.