Osmotic Monitoring of Microservices between the Edge and Cloud

Arthur Souza*, Nélio Cacho*†, Prem Prakash Jayaraman‡, Rajiv Ranjan†, Alexander Romanovsky†, Ayman Noor†
*Federal University of Rio Grande do Norte, Natal, Brazil
†Newcastle University, Newcastle upon Tyne, UK
‡Swinburne University of Technology, Melbourne, Australia
arthurecassio@ppgsc.ufrn.br, neliocacho@dimap.ufrn.br,
raj.ranjan, alexander.romanovsky, a.i.h.noor2(@newcastle.ac.uk), pjayaraman@swin.edu.au

***Abstract*—Osmotic computing is a new IoT application programming paradigm thats driven by the significant increase in resource capacity/capability at the network edge, along with support for data transfer protocols that enable such resources to interact more seamlessly with Cloud-based services. Much of the difficulty in QoS and performance monitoring of IoT applications in an Osmotic computing environment is due to the massive scale and heterogeneity (IoT + Edge + Cloud) of computing environments. To, this end, this work presents an integrated monitoring system for monitoring IoT applications decomposed as microservices and executed in an Osmotic computing environment. A real-world smart parking IoT application is used for an experimental evaluation and for demonstrating the effectiveness of the proposed approach. Through rigorous experimental evaluation, we validate the Osmotic monitoring system ability to holistically identify variation in CPU, memory, and network latency of microservices deployed across Cloud and Edge layers.**

***Index Terms*—cross-layer monitoring; QoS; Edge.**

I. Introduction

The advent of Internet of Things (IoT) [1]–[3] and Smart City applications created a scenario where billions of users or devices get connected to applications on the Internet, which results in trillions of gigabytes of data being generated and processed in cloud datacenters [4], [5]. The increasing need for supporting interaction between IoT and cloud computing systems has led to the creation of the *Edge* [6], *Fog* [7] and *Osmotic* Computing [4]. Osmotic computing is a new paradigm to support the efficient execution of Internet of Things (IoT) services (microservices) and applications at the network edge [4] by providing increased resource and management capabilities at the edge of the network. One challenge that underpins such emerging approaches is the dynamic management of microservices across cloud and edge datacenters. For instance, defining when and how microservices can be migrated from edge resources to cloud-based resources (and vice versa), and characteristics which influence such migration, remains a challenge [4].

Monitoring [8] plays a central role in identifying "when" a certain microservice should be migrated. For migration to be effective, it is necessary to properly monitor the performance of the microservices. The monitoring of microservices in IoT environment is a recent topic and therefore few works have been carried out in this regard. The work presented in the paper seeks to explore this topic in the construction of a solution that meets the requirements of monitoring microservices as well as IoT and cloud applications.

A. Motivation: Smart Parking IoT Application

Within the scenario of vehicular traffic management in urban centers, the provision and efficient occupation of parking spaces is a common problem to be solved. The intelligent parking application is a multi-layer application widely deployed for such problem [9], [10]. The main purpose of this application is to alert the driver regarding available parking spaces near his/her location. This work leverages the intelligent parking application as a motivation example. Figure I-A depicts a conceptual implementation of this application using a microservice architecture. The smart parking application comprises there microservices: (i) *parking management*, (ii) *user data management* and the (iii) *selection of vacancies* according to user's preferences. The parking management deals with the monitoring of the available spaces in the urban space dedicated to parking lot. User management stores the data of locations already used by drivers, as well as their preferences. Finally, the selection of vacancies microservice schedules and recommends the possible vacancies available to users.

Parking management is responsible for sensory interfacing and monitoring (with sensors instrumented to indicate the presence of vehicle). Such a microservice is self-contained and deployed at the edge (i.e. at each parking lot). User management is deployed to the cloud, so user preference data is accessible to all city parking lots. The vacancy selection microservice is the most important one. It continuously runs an algorithm for selecting vacancies from the available parking lots according to user preferences. However, this microservice may need to run on the edge or cloud depending on several factors (typical of osmotic computing). For example, during periods of heavy vehicle traffic, and large numbers of vehicles searching for parking spaces (e.g. during weekends), the vacancy selection microservice would run in the cloud where

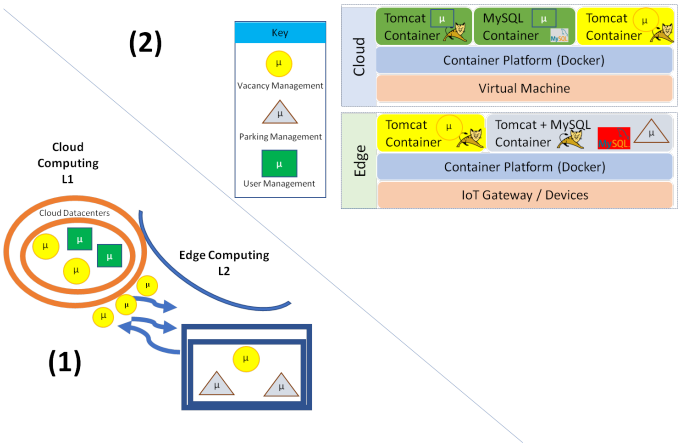

Fig. 1. Osmotic Movement of Microservices across Cloud and Edge (1); Parking Management Application (2)

greater processing power and high computing capacity would allow an easy horizontal scalability. However, in periods of low traffic and low demand this microservice could be running on the edge.

B. Research Contributions

Existing Quality of Service (QoS) monitoring tools and techniques suffers from serious technical limitations when subjected to Osmotic Computing. For example, there is an urgent need to find answers to the following research questions:

- how to ubiquitously monitoring QoS of microservices mapped to an Osmotic Computing (Edge+Cloud) environment?
- how to aggregate QoS measures of microservices running in Osmotic Computing environment to give a holistic view of IoT application's (e.g., smart parking) run-time performance?

To address the aforementioned challenges, in this paper we make following concrete research contributions:

- We develop a unified monitoring model for Osmotic Computing that provides an IoT application administrator with detailed QoS information related to microservices deployed across Cloud and Edge.
- We propose Osmotic Monitoring, a monitoring system for Osmotic computing that implements the proposed unified monitoring model.
- We conduct extensive experimental evaluation of Osmotic Monitoring system in order to study the scalability of the proposed solution.

The rest of this paper is organized as follows: Section II analyses the state of the art, while Section III presents the details on Osmotic monitoring model and its implementation respectively. On the other hand, Seciton IV discusses the experimental evaluation results. The paper ends with a brief conclusion and future work in Section V.

II. Related Work

Several works already published have explored topics related to service monitoring, as well as models and metrics for QoS assurance. Whether in the cloud [11], using microservices [12] or even in monitoring services at the edge [13], varied solutions and results have been presented.

In [11], the authors present CLAMBS, a framework for monitoring and benchmarking applications in a multi-cloud environment. In addition, a model for multi-layer monitoring in the cloud is presented. In this way, QoS parameters relevant for each cloud service layer are listed. Finally, an experimental evaluation is performed in the IaaS level. The work presented here follows a similar approach for defining and experimenting with QoS parameters, although it is different from the use of the cloud and the edge, besides focusing more on the application level. The work presented in [12] presents a service quality model for multi-tier cloud applications deployed by microservices. The use of QoS parameters in the definition of Qualitative Meta data Markers can aid the software engineering tools in the composition of the microservices according to the monitored values for metrics linked to the non-functional requirements. Among the parameters listed, we can highlight the CPU usage and free amount of memory. These same parameters are evaluated by this work which, in addition, also monitors the use of the network, as well as, it defines a model that allows the monitoring of parameters connected to the IoT devices. The microservice monitoring in the edge environment is reported in the paper presented at [5]. In [5] a state-of-the-art review of self-adaptive applications using edge microservices and services in the cloud are performed. The results observed shows that the main parameters of QoS for virtual machines in the cloud are the usage of: CPU, memory and network.

Finally, the monitoring of services deployed in containers is present in the works [13] and [14]. In the work published in [13] the authors present a framework called PyMon that uses the Docker management API to obtain statistics of resources used by containers. Unlike [13], the present study uses libraries to monitor processes inside the containers, thus allowing the effective monitoring of a container that performs a multi-service or multi-process environment. The work presented in [14] brings an assessment of the use of Docker containers versus the use of Virtual Machines. To verify the QoS parameters to be compared for evaluation, the authors monitored the CPU usage by the installed Docker process, not verifying the parameters of the containers that are being executed or even of the processes internal to the containers.

Moreover, applications built using the microservices architecture obey a set of rules with the purpose of making the microservice self-sufficient and easily scalable.The implementation of microservices can be done in several ways, however, the use of containers in the construction of microservices has attracted significant attention recently [15]–[17]. The use of containers is becoming so popular that it is currently possible to run containers on IoT devices, such as IoT Gateways (e.g. RaspberryPi). Thus, the challenge of monitoring containers as

well as microservices running within these containers is highly relevant in the context of osmotic environment.

In summary, works such as CLAMBS model [11] enables efficient monitoring of services in a multicloud environment but lacks capability to monitor microservices at the edge. Current works on microservice monitoring [5] usually focus on single layer monitoring, i.e., microservices in the cloud [14] or microservices at the edge [13]. Our proposed work differs from the current approaches by presenting an advanced monitoring solution that can be used to monitor microservices deployed in Osmotic Computing environment i.e. the cloud and/or deployed at the edge.

III. Monitoring Microservices in Osmotic Computing (Edge to Cloud)

The proposed monitoring model is an extension of the CLAMBS [11]. In order to support Osmotic computing environment, several extensions to CLAMBS has been proposed and incorporated. First, the PUSH communication model between the agent and the manager was adopted. In Osmotic monitoring, the manager agent that is responsible to manage the monitoring of microservices runs in the cloud. The choice for this type of communication seeks to meet the restrictions of the IoT devices, as well as the security barriers imposed by IoT device networks for external access. Second, it was necessary to define a generic model of monitoring agents that can be extended to include support for new devices. In other words, the myriad of devices present in an IoT environment, as well as the various protocols and APIs involved, increase the complexity in providing aggregated and multi platform solutions.

The implementation of a specific agent can easily be extended from the generic agent. Third, we incorporate the concept of *Smart Agent* for devices that have a permissive computational power. Generally IoT devices have limited computing power and focus on the resolution of the sensing or actuation for which they are intended. However, some devices (sensors/actuators) have a more robust and computationally capable hardware such as they have connectivity through WIFI. In order to allow improved monitoring of these devices, the smart agents have two main abstractions: *Gateway Agents* and *Sensor Agents*.

A. Monitoring Model

Monitoring systems are commonly composed of *monitoring agents* and *management services*. Normally, monitoring agents are components that only read data from monitored services or machines. The management services store the data collected by the agents and expose this data via API or through graphical interfaces for system administrators.

1) Monitoring Agents: Usually the deployment and configuration of the monitoring agents are performed manually, each agent being specific to the target monitoring architecture. Monitoring agents (OMA) on the other hand are multi-platform monitors agents based on a Multi-cloud monitoring model. OMA supports monitoring of microservices implanted in osmotic environment comprising of heterogeneous cloud and / or edge resources. All monitor agents extend a common agent, called SmartAgent (see figure 2) as described earlier. SmartAgent represents a service consisting of three operations: 1 - *register*, 2 - *sendData*, 3 - *setConfiguration*. The *register* operation must make an HTTP PUT request that sends the agent registration information to the management system. The *sendData* operation must periodically perform an HTTP POST request to the management system to send the metrics obtained. *SetConfiguration* must send an HTTP GET request to the manager system to obtain the agent configuration parameters. Figure 3 shows the communication model used by the Osmotic monitoring agents. The first action performed by these is the agent registration with the Manager. After this, the manager can receive the data sent by the agent (action 2), as well as (action 3) can modify some agent configuration parameter.

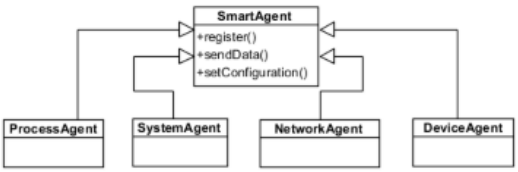

Fig. 2. Monitoring Agents Model

The *SystemAgent* and *NetworkAgent* agents are the most commonly found in the monitoring tools. `SystemAgent` monitors the system as a whole, for example, a virtual machine or a container. *NetworkAgent* is responsible for network monitoring. Although network metrics can be related to a single system, which would lead to the inclusion of these metrics in the *SystemAgent*, the possibility of multiple network interfaces in one system justifies the need for the *NetworkAgent*.

ProcessAgent is responsible for collecting metrics related to a specific process running on a system. This type of agent is already present in most virtual machine monitoring tools in a cloud environment. As for the monitoring of processes executed in containers, the current tools focus on the monitoring of the container itself. The execution of only one process per container is the most common scenario in the construction of applications in microservices, however, in an osmotic environment the use of several containers can make it difficult to migrate from the cloud to the edge or vice versa in a way that is monitoring of multiple processes in the same container. Therefore, this work has built a *ProcessAgent* that can run internally to the container. Finally, *DeviceAgent* handles the collection of metrics or data from IoT devices. IoT devices increasingly see improving processing power, so some of these devices need to be monitored. The monitoring of the IOT devices can serve for simple gauging of acquired data, availability, as well as, to prevent failures of misuse of the device.

2) Manager Agent: The Osmotic monitoring data management agent is called *SmartManager*. SmartManager basically performs various services that receive the data from the

monitoring agents. The data obtained is persisted in a database or data storage services. SmartManager must also provide an API for accessing data saved by other services or other applications.

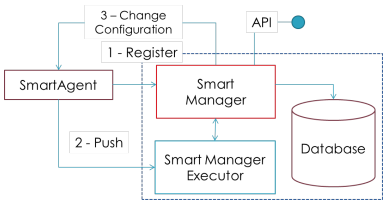

Fig. 3. Agents to Manager Communication Model

The sending of data by the monitoring agents to the management system occurs according to a well defined sequence of steps (figure 3). Initially, *SmarAgent* on startup sends a registration request to *SmartManager*. The *SmartManager* receives the request (*1-Register*) and registers the *SmartAgent*, returning to the *SmartAgent* an access key and an endpoint to send the data. From there, the *SmarManagerExecutor* (*2-Push*) is enabled to receive the data sent by the *SmartAgent*. *SmartAgent* periodically queries *SmartManager* for its configuration parameters (*3 - Change Configuration*). Dynamic configuration enables real-time agent management. It is expected that applications deployed in an osmotic environment will have a degree of self-management. Mainly, in cases of self-managed microservice migration between the cloud and the edge. In this way, the real-time management of the monitor agent is highly relevant, since it allows the application that makes use of the monitored metrics to change the agent at runtime.

B. Osmotic Monitoring: System Implementation

The monitoring model presented previously underpined the development of Osmotic Monitoring system, a proof-of-concept solution for monitoring microservices in osmotic computing environment. The implementation was performed in the Java language, making use of the RESTLet framework and the Hyperic SIGAR library(https://github.com/hyperic/sigar). The use of the Java language allows the construction of a multiplatform solution, easily transferable in Multicloud environments, being still compatible with some equipment of edge computing. The RESTLet(https://restlet.com/) is a framework that facilitates the construction of WEB API in Java. RESTLet provides a set of abstractions for the development of REST architectural style APIs. Rest API is a standard between several container monitoring tools [5] and encourages the integration of applications, as well as composite microservices.

The metrics adopted for the monitoring of osmotic microservices followed the same metrics for the SAAS level of microservices in Clouds defined in [5], [11]. Although there may be a discussion as to what level, whether SaaS or PaaS, the osmotic microservices are better related, the metrics defined in PaaS level [11], namely: SystemUpTime, SystemServices, SystemDesc, Utilization are already easily obtained by the current monitoring tools containers [5]. The choice of parameters of the level of SaaS is corroborated by

TABLE I
MONITORING METRICS PROVIDED BY SMART MANAGER

Scope	Metric	API Path
Process	% CPU	[app]/process/
Process	Memory Usage	[app]/process/
System	Memory Usage	[app]/system/
System	Memory Free	[app]/system/
System	Avaliability	[app]/system/
Network	Rx_Bytes	[app]/network/
Network	Tx_Bytes	[app]/network/
Device	Avaliability	[app]/device/
Agents	-	[app]/agents/

the premise that each microservice is directly related to an application that is deployed on a Container platform such as Docker or Linux Container (LXC). In other words, the PaaS level is more correlated to the container platform than to the osmotic microservices. Thus, the metrics used for the monitoring model were: *CPU usage*, *memory usage*, *amount of free memory*, *the amount of bytes being downloaded*, *amount of bytes being uploaded*, and *availability*. The availability metrics were applied to two different scopes, the system (microservice or container) and device scopes. Table I presents the API specification of the smart manager to support monitoring data provided by the monitoring agents.

The metrics have been grouped by Application, that is, any data sent to the Manager API must indicate to which application (parameter *[app]* the URL) that element (process, system, network or device) is bound. In this way, it is possible to provide an overview of the monitoring of the elements used in the execution of a specific application, even though this application has several microservices deployed in various cloud and/or edge environments. All data-sending calls to the Manager API are made up of *HTTP POST* requests to the specific *Path*, for example to send process data the request would be in the *Path [app]/process*.

The metrics monitored for the processes were the percentage use of the CPU and the amount of memory used. The first metric captures the CPU usage percentage of a process for a specific application and the second the amount of memory used by the process in *MegaBytes*. Within the monitoring model presented in section III, the *SystemAgent* element represents a complete system, that is, it can represent a Virtual Machine(VM), a Container or Microservice within an Osmotic Computing environment. This work treated a System as a Microservice or a Container, when the microservice is fully contained in a container (see Selection Microservice on figure I-A).

However, when the microservice is distributed in more than one container (see User Microservice on figure I-A), it consists of two systems, one for each container. The data monitored for the System elements were: amount of memory used, amount of free memory and availability. The amount of memory used registers the use of memory in *MegaBytes* for all the processes that are running on that System. The amount of free memory registers the available memory for use by the System. And, finally, availability assesses whether the System is accessible and available. Network usage metrics

captured include the amount of bytes in *KiloBytes* downloaded (*RX_Bytes*) or uploaded (*TX_Bytes*) by a system or a network interface in an instant of time. The last monitored data was the availability of a device used by the application. This metric only shows the true or false value and its implementation is very specific for each device. For example, in our evaluation it was necessary to use a specific XloBorg[1] sensor library to check device availability.


```
{
  "_id" : ObjectId("5a86b9c0d85ba520acf622db"),
  "application": "park-manager-osmotic",
  "agent-key": "5a7d8f9aff60ec393865d111",
  "agent-type": "process",
  "timestamp": "2018-01-06 21:47:44",
  "process-id": 1
  "process-name": "mysqld",
  "process-cpu": 9.64,
  "memory-used": 120
}
```


Fig. 4. Example of Metric stored on MongoDB

The agent running in the container, VM and or any system that hosts the microservice captures the metrics explained above and sends it to the Smart Manager for further processing. The received data is stored in a MongoDB (https://www.mongodb.com/) database in the JSON format. The data stored on MongoDB are grouped by application and identified by the type of agent and the unique key of the agent. An example of recorded data for process monitoring is shown in figure 4. The green attribute (*_id*) is the identifier of the document in the database, being generated automatically by MongoDB. The attributes in red, namely: *application*, *agent-key*, *agent-type*, and *timestamp* are the attributes required for each saved metric. The *application* identifies the application. The *key-agent* represents the agent's unique key. The *type-agent* represents the type of agent. At last, the *timestamp* stores the instant the metric was saved. The attributes in blue are variable according to the type of agent that was sent. In the case of the example the *process-id* and *process-name* attributes identify the monitored process, while the *process-CPU* and *memory-used* attributes represent the metrics.

Each monitoring agent must make an initial registration for sending monitoring data. The agent registration is done through an HTTP PUT request to the *Path [app]/agents*. The configuration attributes required for the registration of an agent are: *agent-type*, *access-key*, *access-password*, and *application*. The *application* is informed directly in the URL. The *agent-type*, *access-key* and *access-password* are informed in the request body. The *agent-type* tells the agent type so that the manager selects the correlated endpoint. The *access-key* and *access-password* attributes are used for agent authentication. Every time that the Manager processes an agent registration request, *an endpoint*, *an agent-key*, and *a read-frequency* are returned to the agent. The *endpoint* identifies the Manager Service that will handle POST sending requests for the agent. The *agent-key* uniquely identifies the agent and ensures that the agent has been correctly registered. The *read-frequency* indicates the time interval the agent should wait for each request to send data. If the Manager deems its necessary to modify any of the returned parameters, it only sends new values when the agent sends an HTTP GET request to the *Path [app]/agents* to verify that the attributes have not been changed.

The specific implementation of agents: *ProcessAgent*, *SystemAgent* and *NetworkAgent* basically made use of the Hyperic SIGAR library. SIGAR is a multiplatform library (Unix, Win, Solaris, FREEBSD, MAC OS, etc) written in Java that provides an functionalities for accessing operating system information. Although the use of the SIGAR library has been presented in the work [5], the present work explores the same library in the monitoring containers as well as virtual machines. The use of SIGAR in the construction of the aforementioned agents is relatively simple, being composed of the instantiation of an object of class *org.Hyperic.sigar.sigar* and the invocation of methods present in this abstraction. For example, for access to CPU usage of a process it is enough to invoke the *getProcCPU* method, informing the process id (*pid*). To access information about the system memory it is necessary to invoke the *getMem* method.

It is important to note that for the measurement of the network traffic rate, it was necessary to deploy timed counters since SIGAR only returns the amount of *RX_Bytes* and *TX_Bytes* of a network interface at a given instant of time. Another change was the addition of all network data of all interfaces to constitute the traffic of a system. Agent-specific settings such as which processes to monitor, which network interfaces to monitor, which the initial endpoint of the manager, and access attributes were informed in an initial agent configuration file. Thus, an agent developed to capture and process metrics can be easily reusable in another system. Agents developed for *ProcessAgent*, *SystemAgent* and *NetworkAgent* can be used on any system that supports JAVA language version 7 or higher. However, the agents developed for the *DevicesAgent* were totally specific to the devices used in the experimental evaluation (see subsection **??**).

IV. Experimental Evaluation

An experimental evaluation of the monitoring system described in section III was carried out in order to prove the efficiency and efficacy of the monitoring of micro-services in cloud and the edge. Thus, an initial version of a microservice-based Smart Parking application (as discussed in section I-A) was developed and deployed in an osmotic computing environment (edge and cloud). Subsequently, variable load tests were performed in order to verify if the data monitored reflected the variations introduced by the corresponding load tests. The tests were not intended to measure performance, although they may have presented some data relevant to that scope.

A. Monitored Application

The Smart Parking application (as depicted in Figure 1) searches for real-time mapping of parking spaces available in a city. The citizen as a driver accesses the Smart Parking to know the best places available according to his personal preferences.

[1]https://www.piborg.org/sensors-1136/xloborg

The main use case follows the flow: 1 - the driver travels by a road, 2 - The Smart Parking application is notified of the position of the driver, 3 - the job selection service searches possible available positions, Smart Parking alerts the driver to the vacancies available. With this scenario in mind, basic versions of the three micro-services specified in the section have been implemented, namely: *User Management, Selection Vacancies and Parking Management*.

User Management (UM) is the service that is deployed to the cloud. It is responsible for storing user data as well as for providing system communication as the user. User interaction can occur via an application deployed on your phone. *Selection Vacancies* (SV) continuously receives UM job requisition notifications. The incoming requisitions are processed through a selection algorithm that continuously consults with Parking Management to check the status of the vacancies. Once the vacancies are defined the SV notifies the UM. *Parking Management* (PM) continuously monitors vacancy status. To do this, it is implanted on the edge and communicates with the IoT sensors that identify the occupation or the release of a vacancy. Theoretically, the PM must be replicated between the various parking lots as many times as necessary.

UM, SV and PM are services deployed on the microservices architecture and were responsible for a very specific functionality as described earlier. Inter-service communication occurs through a REST API to access its functionality. For experimental evaluation, specific API call were implemented for each service, namely: for the UM a call to query the user data; for the PM, a call to consult the vacancies and their states; and, for the SV a call that returns a vacancy available to a user when it accesses a parking lot. All services were developed in Java, running on an Apache Tomcat server(http://tomcat.apache.org/). For the services UM and PM that require persistence of contextual data of the entities MySQL(https://www.mysql.com/) database was employed. The smart parking application with the three microservices were deployed in containers. The containers were built for execution on the Docker platform[2]. As well, it tried to make the environment of execution of the SV a little more equal, since the same microservice was executed on a virtual machine in the cloud and in a RaspberryPi on Edge. Although the deployment of microservices through Docker Containers is not an unpublished topic [16], this work explores for the first time the use of Docker Containers for deployment of the same service that runs in the Cloud or the Edge.

The use of Containers Docker allows the use of two possible deployment cases, namely: a container for each microservice or several containers for each microservice. In the first case (F1) in a same image of the container are installed all the components used by the microservice. In the second, each component is installed in its own container, that is, the Tomcat server will compose one container and the MySQL database will be in another. The second case (F2) most commonly used by users of the Docker platform since it does not require the construction of specific images, instead using standard images already available in the Docker HUB catalog of images.

Considering the above two cases, as well as, trying to cover most possible scenarios of execution of microservices in an osmotic environment, an explicit plan for experimentation is in Table II.

In the Cloud environment, the UM micro service can be instantiated by only one container (F1), scenario C1, or for two containers (F2), scenario C2. Similarly, the PM service can also be instantiated in scenarios E1 for F1 and E2 for F2. Finally, the SV service that only requires the use of Tomcat used only the F1 case, although it presents two scenarios: S1 for the Cloud environment and S2 for the Edge environment.

B. Experimental Design

We used Apache JMeter (https://jmeter.apache.org/) to generate HTTP requests to test and validate the Osmotic Monitoring system's capability. The JMeter test cases are presented in table II. The tests consisted of performing 10, 100, and 500 simultaneous requests to the Osmotic Monitoring system at a fixed interval of 5 minutes. Initially, the idea was to increase the load of the tests by 10 times with each new test battery, however 1000 test requests for the services deployed in RaspberryPi caused a stack overflow and made the service unavailable. Due to this, the last test was performed with 500 requests and even with this number of concurrent requests, some faulty responses were observed in the Edge environment, a fact not observed in the tests of 10 and 100 requests.

The containers with the microservices of the Smart Parking Application, were deployed in an Openstack (https://www.openstack.org/) cloud of the Metrópole Digital Institute (https://www.imd.ufrn.br/portal/), and in a RaspberryPI (https://www.raspberrypi.org/) 1 Model B, in the Edge. The cloud used a virtual machine that runs a Linux system with Ubuntu (https://www.ubuntu.com/), version 14.03, on a virtual hardware with configuration of 2 vCPU, 4 GB of memory and 20 GB of disk. On the virtual machine was installed the platform Docker in its version 1.10. The UM service in scenario C1 (as described in table III) deployment was based on the MySQL 5.7 image https://hub.docker.com/_/mysql/ obtained via Docker HUB. This image included a version of the Java virtual machine, version 8 and the Tomcat server version 7. The scenario C2 made use of the same image for the container of MySQL whereas for the container of Tomcat used the image (https://hub.docker.com/_/tomcat/) for Java 8 and with Tomcat 7. The same Tomcat image was used in scenario S1 of the SV service. For the Edge environment scenarios (S2, E1, and E2), here simulated by the RaspberryPI 1 Model B that runs a Raspbian system in a configuration from a CPU core to a 700 Mhz clock with 512 MB of RAM and a 4GB SD memory. We used a specific image (https://hub.docker.com/r/hypriot/rpi-mysql/) for MySQL 5.7 and another one (https://hub.docker.com/r/dordoka/rpi-tomcat/) to Tomcat 7. The P1 scenario that used an integrated image made use of RPI-MYSQL image on which a Java 8 version and a Tomcat version 7 were installed.

[2]https://www.docker.com/what-docker

TABLE II
MICROSERVICE SCENARIOS DEPLOYED AT DOCKER

Environment	Scenario	Microservice	Containers
Cloud	C1	User Management	1 - Tomcat + MySQL
Cloud	C2	User Management	1 - Tomcat, 2 - MySQL
Cloud	S1	Selection Vacancies	1 - Tomcat
RaspberryPi	S2	Selection Vacancies	1 - Tomcat
RaspberryPi	E1	Parking Management	1 - Tomcat + Mysql
RaspberryPi	E2	Parking Management	1 - Tomcat, 2 - MySQL

The main objective of the experimental design was to produce a computational load and a network traffic load for the microservices that could be measured by the Osmotic monitoring system in order to prove the effectiveness of the monitoring model. A raffle was not carried out in the order of testing or in the choice of scenarios to be prioritized. The tests were repeated 10 times each to obtain the mean results of the measured metrics.

C. Latency Time Results

The average latency time results, in milliseconds, obtained for the requests made in each scenarios are shown in table III, as well as the number of Bytes, in KB, sent by the requests. The values obtained for the latency time clearly reveal the computational power difference of the Cloud and the Edge, as well as a slightly better performance of the C2 scenario in relation to the C1. The best result of scenario C2 indicates that the use of multi container architecture per service exploits the hardware of the virtual machine more efficiently. Another important note, and that the behavior presented in the Cloud environment did not recur in the Edge environment. In fact, in the Edge the behavior was inverse i.e., scenario E1 presented better performance than the E2. Probably, the workload required to execute more than one container on hardware with little computational capacity influenced the performance of the microservices. Although the actual performance observations highlighted here are not the main objective of this work, the proposed Osmotic monitoring system presents a novel way to monitor the performance of such microservices in osmotic computing environments. This provides IoT system administrators who are generally challenged with managing multiple of such microservices deployed across cloud and edge the ability to clearly understand the performance of such microservices.

D. CPU Results

The CPU usage values for all evaluated scenarios are presented in figures 5, 6 ,7, 8, 9, 10. For an analysis of the results obtained for different scenarios in the same environment, for example, for the Cloud environment, figures 5 and 6 show the percentage of CPU usage for the UM service deployed in scenarios C1 and C2 respectively. Evaluating only the result for scenario C1, for the tests of 10 and 100 requisitions the presented variation was relatively little from 10% to 30%, whereas for the one with 500 requisitions it reached 70% of use. For scenario C2, the results of the variance were similar

TABLE III
REQUEST RESULTS FOR ANALYZED SCENARIOS

Number of Requests	Scenario	Latency Avarage (ms)	Bytes (KB)
10	C1	29,93	0,7
100	C1	36,2	7
500	C1	29,3	35
10	C2	30,83	0,7
100	C2	29,89	7
500	C2	29,26	35
10	S1	35,5	0,7
100	S1	36,89	7
500	S1	73,17	35
10	S2	290,3	0,7
100	S2	6966,97	7
500	S2	10535,53	35
10	E1	97,32	0,7
100	E1	575,29	7
500	E1	4623,52	35
10	E2	118,6	0,7
100	E2	843,67	7
500	E2	6075,03	35

in behavior, that is, for 10 and 100 the variation was little 2% to 4% compared to the 500 that presented 15% to 25% of use. However, it is important to point out that comparing the results obtained for C1 and C2, the multi container microservice architecture had a lower CPU consumption, that is, it presented a better performance. This performance observation had already been indicated by the analysis of the latency times of the requests (see table III).

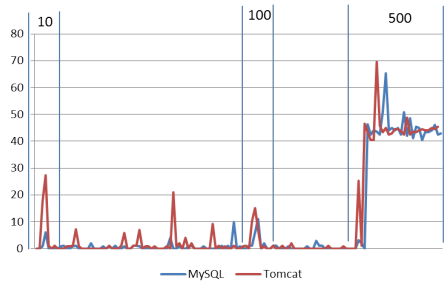

Fig. 5. % CPU Usage for User Microservice on One Container (C1)

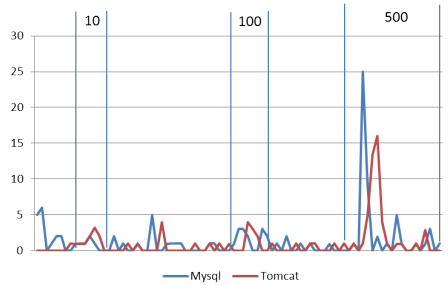

Fig. 6. % CPU Usage for User Microservice on Two Containers (C2)

For the scenarios similar to C1 and C2 in the Edge environment, that is, scenarios E1 and E2 (see figures 7, 8), the observed behavior was quite different. Again, the monitoring of the metrics was effective and reflected the increase in CPU

usage with the increase in the number of requests. Specifically for the scenario with a container running two processes, E1, there was an expressive increase in CPU usage in relation to the tests with 10 requests, 10% of use, while by 100 the use was 60%. The test use with 500 requisitions ranged from 60% to 80%. For the E2 scenario that explores a single process running per container, a greater variation in CPU consumption was observed in relation to the metrics obtained for scenario E1, as well as a considerable increase of the test of 10, use of 18% for the test of 100, use of 70% to 80%. Still referring to the E2 scenario, the results of the 100 and 500 requisitions tests showed little increase in the variation, 70% to 80% of use for 100 and 80% to 90% of use for 500.

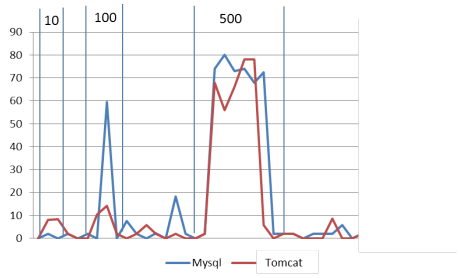

Fig. 7. % CPU Usage for Parking Microservice on One Container (E1)

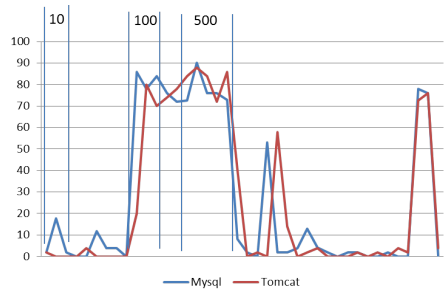

Fig. 8. % CPU Usage for Parking Microservice on Two Containers (E2)

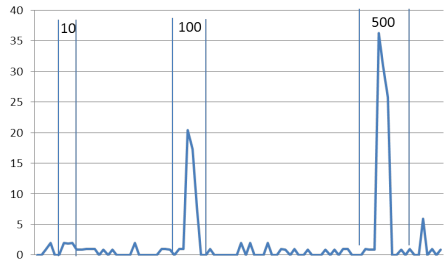

Fig. 9. % CPU Usage for Selection Microservice on Cloud (S1)

The variation in observed CPU usage for the S1 and S2 scenarios reveals a significant increase in the CPU utilization rate that corresponds directly to the execution period of our tests. For example, as depicted in figure 9 the percentage of CPU usage by the SV service deployed under scenario S1 in the Cloud, shows maximum peaks precisely in the periods that the tests were performed. This same behavior is seen in figure 10 that presents the SV results implanted in scenario S2 in the

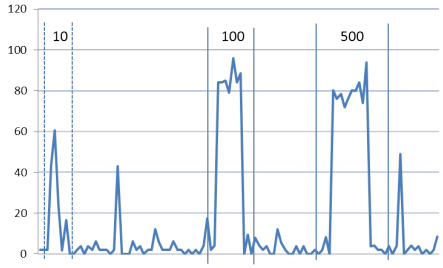

Fig. 10. % CPU Usage for Selection Microservice on Edge (S2)

Edge environment. Although the increase in usage behavior is not as prominent as in figure 9, the variation of CPU usage was again measured by the Osmotic monitoring system.

E. Memory Results

The results obtained for the memory consumption (Figures 11 to 15) although less explicit or as elucidating as the results of CPU consumption reveal some interesting conclusions. For example, for the Cloud environment in the UM service both the amount of memory used by the MySQL and Tomcat processes and the system was practically the same in the two scenarios evaluated, scenarios C1 (fig. 11) and C2 (fig. 12). The only significant increase occurs in Tomcat in scenario C1 in the test of 500 requests where the amount of memory consumed increase 50%, from 100 MB to 200 MB. The MySQL process practically does not suffer memory variation always consuming 200 MB. Also, System memory variation is very low, with values ranging from 1500 MB to 1650 MB. Tomcat's largest change in memory consumption can be explained by the increase in the number of requests since since the requests were always of the same type, queries to MySQL were always the same, being easily managed by the database cache.

The graphs shown in Figures 13 and 14 presents results of experimental scenarios E1 and E2 conducted in the Edge environment. The memory consumption of MySQL is practically the same in both scenarios, having a value of approximately 50 MB. The consumption of Tomcat also varied little in the two scenarios, being from 40 MB to 49 MB in E1 and 30 MB to 40 MB in E2. Again, a variation in memory consumption was observed by Tomcat while it did not occur with MySQL. System process consumption, System, experienced a similar variation in the two scenarios from 190 MB to 200 MB in E1 and 260 MB to 270 MB in E2. It is important to note that the same average variation was observed for the two scenarios, that is, for E1 the variation in the three tests (10,100,500) was 10 MB, as was the case for E2, a variation of 10 MB. Thus, the effectiveness of the monitoring system is once again proven, since for the same simulated tests in the two scenarios the monitored values were the same.

Another relevant observation, and the largest memory consumption in scenario E2, that is separate containers. Probably the greatest consumption occurs because of the need to keep the data of specific states of each container in memory. In other words, whereas the E1 scenario uses only one container

(Tomcat + MySQL), the E2 scenario uses one container for Tomcat and another for MySQL. In E2, therefore, the memory consumption is specified by the libraries and data container state is duplicated. The memory consumption of the SV microservice (see Figure 15) presented a practically linear variation in the two scenarios explored. Differently from the absolute values presented in figures 11, 12, 13, and 14, Figure 15 presents the results in percent of memory usage so that we can evaluate the variations of the system's performance in both Cloud and Edge environments. In the Cloud environment, scenario S1, consumption varied from 3% to 5%. On the Edge environment, scenario S2, consumption was between 7% and 10%. The variation measured by the monitoring system again was very similar to the same tests of 10, 100 and 500 requisitions.

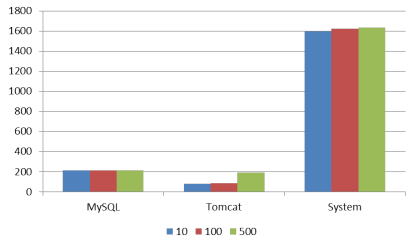

Fig. 11. Memory Usage (MB) for User Microservice on One Container (C1)

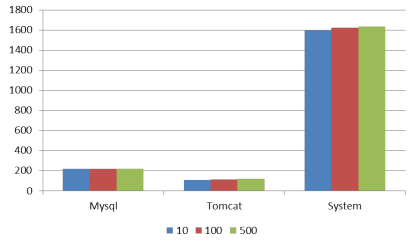

Fig. 12. Memory Usage (MB) for User Microservice on Two Container (C2)

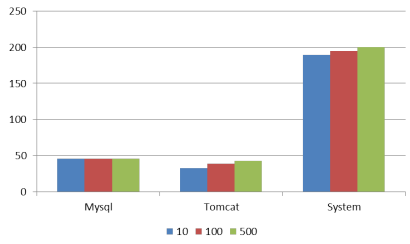

Fig. 13. Memory Usage (MB) for Parking Microservice on One Container (E1)

F. Network Results

The results obtained in the monitoring of network traffic by the microservices were based on the network traffic of the container or the containers where the microservices were deployed. In scenarios C1, E1, S1 and S2 only the download and upload rate of a container is presented, while in scenarios C2 and E2 each container is MySQL or Tomcat has its own traffic rate. The first important observation is precisely related to the use of the architecture that uses more than one container for the same microservice because it allows the isolated monitoring of each process of the microservice making the perception of loadings more effective in these scenarios. Regarding the variation in traffic caused by the load tests, it was verified that the variation in the requisition numbers increased the network flow that was correctly registered by the monitoring system. For the Cloud environment, scenarios C1 and C2, the metrics obtained are shown in figure 16. The traffic for the 10 and 100 requisitions tests obtained little variation (10 to 80 KB for download or upload) when compared to the 500 test that you get a range of 400 KB to 500 KB. For the Edge environment, scenarios E1 and E2, the results presented in figure 17 presented a similar behavior to that described for the Cloud, where only for the test of 500 requisitions was obtained a download and upload with significant variance rate. It is worth mentioning the similar behavior of the two groups of Figures 16 and 17 as to the change in the download and upload rates caused by the 10, 100 and 500 tests, especially in relation to the MySQL Container, where the graphs are practically the same in varying. In the osmotic environment, the scenarios S1 and S2, the results can be seen in figure 18. The observed behavior again reflects the variation induced by the increase in the number of requisitions of the tests of 10, 100 and 500.

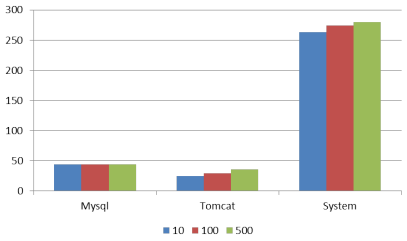

Fig. 14. Memory Usage (MB) for Parking Microservice on Two Container (E2)

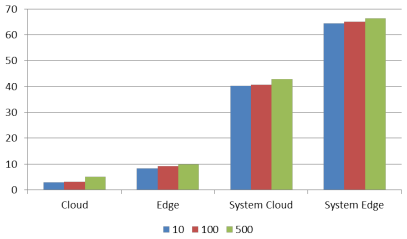

Fig. 15. % Memory Usage for Selection Microservice (S1 and S2)

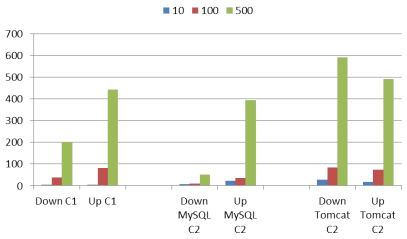

Fig. 16. Network Traffic (KB) for User Microservice (C1 and C2)

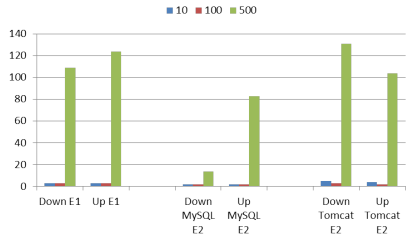

Fig. 17. Network Traffic (KB) for Parking Microservice (E1 and E2)

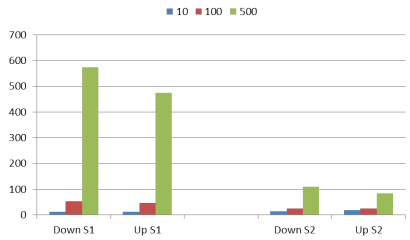

Fig. 18. Network Traffic (KB) for Selection Microservice (S1 and S2)

G. Discussion

The above results validates the capability of the proposed Osmotic monitoring system in its ability to capture fine-grained performance of microservice-based IoT Application deployed in osmotic computing environment (cloud and edge), including each individual microservice of the IoT application, each underlying infrastructure e.g. databases and the performance of container/VM hosting the microservice. Moreover, the Osmotic monitoring system accurately captured several variations introduced to impact the performance of the microservices highlighting the effectiveness of our monitoring system. For example, the system was holistically able to identify variation in CPU, memory and network latency across cloud and edge at the application level, microservice level and infrastructure level (e.g. databases, containers, VM) which as identified in Section II is currently a significantly limitation with other cloud monitoring solutions.

V. Conclusion and Future Work

This work presents an integrated system for monitoring applications decomposed in microservices and executed in an osmotic environment. In its core there is a model for monitoring the QoS parameters of an application by analyzing microservices executed in containers in a cloud environment and/or on the edge. The paper introduces an smart parking application that runs in an osmotic environment in the context of smart cities. This osmotic application is used for an experimental evaluation of the monitoring system in order to demonstrate the effectiveness of the proposed approach. This case study shows that it is possible to apply our approach for microservices deployed in osmotic computing environments. Through experimental evaluations we validates the effectiveness and capability of the proposed monitoring system's ability to monitor the performance of microservice deployment using containers and/or VM's through an exhaustive list of scenarios. The Osmotic monitoring system was holistically able to identify variation in CPU, memory and network latency across cloud and edge at the application level, microservice level and infrastructure level (e.g. databases, containers, VM).

Our future work will expand the model, especially for device monitoring, and ensure an extended evaluation through the execution of new load tests.

VI. Acknowledgment

This work is supported by the SmartMetropolis Project[3].

References

[1] H. Arasteh, V. Hosseinnezhad, V. Loia, A. Tommasetti, O. Troisi, M. Shafie-khah, and P. Siano, "Iot-based smart cities: a survey," in *Environment and Electrical Engineering (EEEIC), 2016 IEEE 16th International Conference on*, pp. 1–6, IEEE, 2016.

[2] E. F. Z. Santana, A. P. Chaves, M. A. Gerosa, F. Kon, and D. S. Milojicic, "Software platforms for smart cities: Concepts, requirements, challenges, and a unified reference architecture," *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, p. 78, 2017.

[3] C. Balakrishna, "Enabling technologies for smart city services and applications," in *Next Generation Mobile Applications, Services and Technologies (NGMAST), 2012 6th International Conference on*, pp. 223–227, IEEE, 2012.

[4] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan, "Osmotic computing: A new paradigm for edge/cloud integration," *IEEE Cloud Computing*, vol. 3, pp. 76–83, Nov 2016.

[5] S. Taherizadeh, A. C. Jones, I. Taylor, Z. Zhao, and V. Stankovski, "Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review," *Journal of Systems and Software*, vol. 136, pp. 19 – 38, 2018.

[6] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, pp. 78–81, May 2016.

[7] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, *Fog Computing: A Platform for Internet of Things and Analytics*, pp. 169–186. Cham: Springer International Publishing, 2014.

[8] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of grid computing*, vol. 12, no. 4, pp. 559–592, 2014.

[9] Z. Ji, I. Ganchev, M. O'Droma, and X. Zhang, "A cloud-based intelligent car parking services for smart cities," in *General Assembly and Scientific Symposium (URSI GASS), 2014 XXXIth URSI*, pp. 1–4, IEEE, 2014.

[10] W. He, G. Yan, and L. Da Xu, "Developing vehicular data cloud services in the iot environment," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 2, pp. 1587–1595, 2014.

[11] K. Alhamazani, R. Ranjan, P. P. Jayaraman, K. Mitra, F. Rabhi, D. Georgakopoulos, and L. Wang, "Cross-layer multi-cloud real-time application qos monitoring and benchmarking as-a-service framework," *IEEE Transactions on Cloud Computing*, 2015.

[12] L. Knight, P. Štefanic, M. Cigale, A. C. Jones, and I. Taylor, "Towards a methodology for creating time-critical, cloud-based cuda applications,"

[13] M. Großmann and C. Klug, "Monitoring container services at the network edge," in *Teletraffic Congress (ITC 29), 2017 29th International*, vol. 1, pp. 130–133, IEEE, 2017.

[14] E. Preeth, F. J. P. Mulerickal, B. Paul, and Y. Sastri, "Evaluation of docker containers based on hardware utilization," in *Control Communication & Computing India (ICCC), 2015 International Conference on*, pp. 697–700, IEEE, 2015.

[15] J. Stubbs, W. Moreira, and R. Dooley, "Distributed systems of microservices using docker and serfnode," in *Science Gateways (IWSG), 2015 7th International Workshop on*, pp. 34–39, IEEE, 2015.

[16] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using docker technology," in *SoutheastCon, 2016*, pp. 1–5, IEEE, 2016.

[17] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[3] http://smartmetropolis.imd.ufrn.br