Using Osmotic Services Composition for Dynamic Load Balancing of Smart City Applications

Arthur Souza*, Zhenyu Wen†, Nélio Cacho*, Alexander Romanovsky†, Philip James†, Rajiv Ranjan†
*Federal University of Rio Grande do Norte, Natal, Brazil
†Newcastle University, Newcastle upon Tyne, UK
arthurecassio@ppgsc.ufrn.br, neliocacho@dimap.ufrn.br
alexander.romanovsky, zhenyu.wen, philip.james, raj.ranjan (@newcastle.ac.uk)

***Abstract*—Edge computing takes computation away from the Cloud closer to the physical world. Therefore, it reduces the cost of communication bandwidth between IoT devices and the Cloud. However, Edge computing imposes certain limitations in computation power because due to poor hardware capacity of the devices. This restriction may significantly affect the performance of the deployed applications, especially Smart City applications. This limitations also could be aggravated by unpredictable human behaviors wich will easily make the Edge computation node overloaded. Osmotic computing is a new IoT application programming paradigm that provides an opportunity to balance the workload between Edge and Cloud therefore to overcome the load imbalance problem of Smart City applications. To this end, we propose an *Osmotic Execution Framework* that leverages state-of-the-art microservices techniques to deploy and execute a Smart City application in a distributed environment including Edge and Cloud. Finally, we evaluate load balancing through latency time analysis of our framework with a real-world smart parking application.**

***Index Terms*—services framework, microservices, osmotic computing, scalability**

I. Introduction

Cloud computing plays a key role in Smart Cities, where massive data is collected and processed to manage assets and resources efficiently. Most of the efforts in Smart City research focus on building Internet of Things (IoT) middleware. Santana and his colleagues [1] report that 47 different platforms are deployed around the world. Some platforms focus on device management, acquisition and processing of data coming from IoT devices [2], [3]. Others offer various software components that facilitate the execution of applications in Smart Cities environments [4]–[6]. All platforms consist of three main concepts: Internet of Things (IoT), Big Data and Cloud Computing [1]. IoT links physical devices, sensors and vehicles through a universal network to enable the exchange of data between these things. Big Data is generated from massive sensors or IoT devices in the cities. Finally, Cloud computing provides infrastructure to process the collected IoT Big Data. However, this Cloud + IoT framework is not very economical for the Smart City environment, because shipping all Big Data to the Cloud will waste huge bandwidth and energy (battery).

Edge computing is the computational capacity present in the devices that make up the network between data-producing devices and Cloud computing. This new computing paradigm allows processing of data close to IoT devices; it therefore reduces latency and saves bandwidth by performing aggregation techniques [7], [8] over Edge computation nodes.

On the other hand, in modern urban environments, Edge nodes are easily overloaded because of unpredictable human behaviors [9]. Smart Parking Application, for example, consists of IoT devices, Edge nodes and Cloud. The IoT devices (sensors) send vacancies in car parks in real-time. The Edge node and Cloud provide the computing resources to find the best solution for a user's request. The details will be described in the next section. In this case, when an event happens such as a superstar concert, an Edge node may need to handle thousands or tens of thousands of requests simultaneously. Thus, it is easy to overload the Edge node, due to the limited computing power of the Edge.

Osmotic computing [10] is a new IoT application programming paradigm that provides an opportunity to execute multi-service applications between the Edge and Cloud. In fact Osmotic computing takes advantage of Edge computing to make possible dynamic workload balance between the Edge Network and Cloud. A new paradigm, Edge computing is supported by the increasing computational capacity of mobile devices deployed in the IoT networks [11]–[13].

In this paper, we aim to leverage the idea of *Osmotic computing* to build a dynamic load balancing framework for Smart City applications. This framework utilizes the advantage of containerization that allows developers to easily migrate or schedule the running of microservices over different computing resources on demand. To be precise, we develop a live migration method that dynamically moves microservices across Edge and Cloud to efficiently utilize computing resources while ensuring load balancing. The main contributions of this paper are listed as follows.

1) **Osmotic Execution Framework – OEF**: unlike SOA based applications (commonly used in middleware platforms for Smart Cities) that are notably composed of a set of structural elements (components or services of data storage, security, communication, analysis, etc), applications composed by microservices must ensure that each microservice is autonomous and decoupled from other microservices. In this context, we propose and develop an architecture, named Osmotic Execution

Framework (OEF), to leverage osmotic computing in Smart City environments.

2) **Osmotic Case Study**: we evaluate the feasibility of osmotic computing through a smart parking application. The proposed application consists of a set of osmotic microservices which can be automatically migrated between Edge and Cloud to optimize the computing resources utilization while ensuring load balancing.

The rest of this paper is organized as follows: Section II highlights the research problem of this paper. Section III analyses the state of the art, while Section IV presents the details of the Osmotic Execution Framework. Section V shows the internal architecture of Osmotic Containers to execute microservices, follow by Section VI that showed the implementation of smart parking osmotic application. At last, Section VII discusses the experimental evaluation design and results. Finally, the paper ends with a brief conclusion in Section VIII.

II. Motivation: Smart Parking 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. Intelligent parking applications are developed for such problems [14], [15]. The main purpose of this application is to alert drivers to the available parking spaces near his/her location. Figure 1 depicts a conceptual implementation of this application using a microservice architecture. The smart parking application comprises three microservices: (i) *parking management*, (ii) *user data management* and (iii) *selection of vacancies*.

Parking management is responsible for the sensor interfaces therefore monitoring the usage of car parks. This microservice is self-contained and deployed on the Edge, continuously collecting data from sensors. *User data management* is deployed to the Cloud, so user preference data is accessible to all city parking lots. The *selection of vacancies* microservice is the most important one. It continuously runs an algorithm for responding to users' requests to select vacancies from the available parking lots according to their preferences.

This microservice runs on the Edge or Cloud depending on trade-off between efficiently utilizing the computing resources and low latency. For example, the Edge node must keep running 24/7 to collect sensor data. Hence, we can run the *selection of vacancies* microservice on the Edge to efficiently utilize the remaining computing resources. However, during periods of heavy vehicle traffic, and large numbers of vehicles searching for parking spaces (e.g. a big event), the *selection of vacancies* microservice should run in the Cloud to utilize the infinity computing resources to consume the surging requests. In order to dynamically balance the workload in the *selection of vacancies* microservice, we need an osmotic policy for this microservice that automatically decides when or in what condition to migrate the microservice to the suitable infrastructure.

III. Related Work

Several works [1]–[5] have already defined middleware platforms for smart cities. In contrast, our paper seeks to define an architecture that integrates the concepts of osmotic composition into a Smart City context in order to enable us to solve the load imbalance issue. Others studies have explored the use of multipurpose microservices, which focus on scalability or the use of microservices in the Cloud. However, few have addressed the use of microservices in smart cities [16]–[18]. This paper advocates the use of OEF to efficiently utilize computing resources while ensuring load balancing. In this sense, the study presented here brings a new perspective on the use of microservices, exploring the context of smart cities as well as osmotic computing.

Visti presented a generic framework [16] named MiCADO (Microservices-based Cloud Application-level Dynamic Orchestrator) to perform automatic management of Cloud infrastructure used by commercial web applications. Among the challenges addressed is the ability to run multiple microservices in a Cloud, as well as considering efficient resource utilization for running these microservices. In contrast, Khanda [19] proposed a solution that allows running microservices only on the Edge. The design of OEF builds upon those works. To the best of our knowledge, this paper is the first work that leverages osmotic computing to deploy microservices over the Cloud and Edge for efficient resource utilization.

Going forward in the construction of infrastructure for Smart Cities, DIMMER [17] is a platform for IoT built in microservices to provide functionalities for smart cities. In this way, DIMMER platform architecture is presented based on the services or functions of smart cities including Resources Catalog, Message Broker and City services like Energy Data, GIS Service.

In contrast, our proposed architecture utilizes both Edge and Cloud computing resources to provide services (or functions) of smart cities. Moreover, our work is the first attempt to implement the idea of osmotic computing in Smart City applications. Another difference of our work is to implement an osmotic application as a case study; and the developed system overcomes the trade-off between efficient resource utilization and low latency.

Mobile Edge Computing (MEC) [20] has primarily been driven by the advance of 5G networks to support user-provisioned services within the network. MEC is also driven by similar requirements of latency-sensitive service provisioning, and the ability to offer application management and service orchestration at the network Edge. Recent research conducted in the realm of MEC occurred in the context of mobile computing where smart phones act as both IoT devices and gateways. However, most of the existing MEC approaches focus on infrastructure-level QoS constraints such as minimization of energy or resource utilization, while giving very little attention to meeting the more complex and interdependent QoS requirements across microservices that need to be choreographed and orchestrated in a coordinated manner to

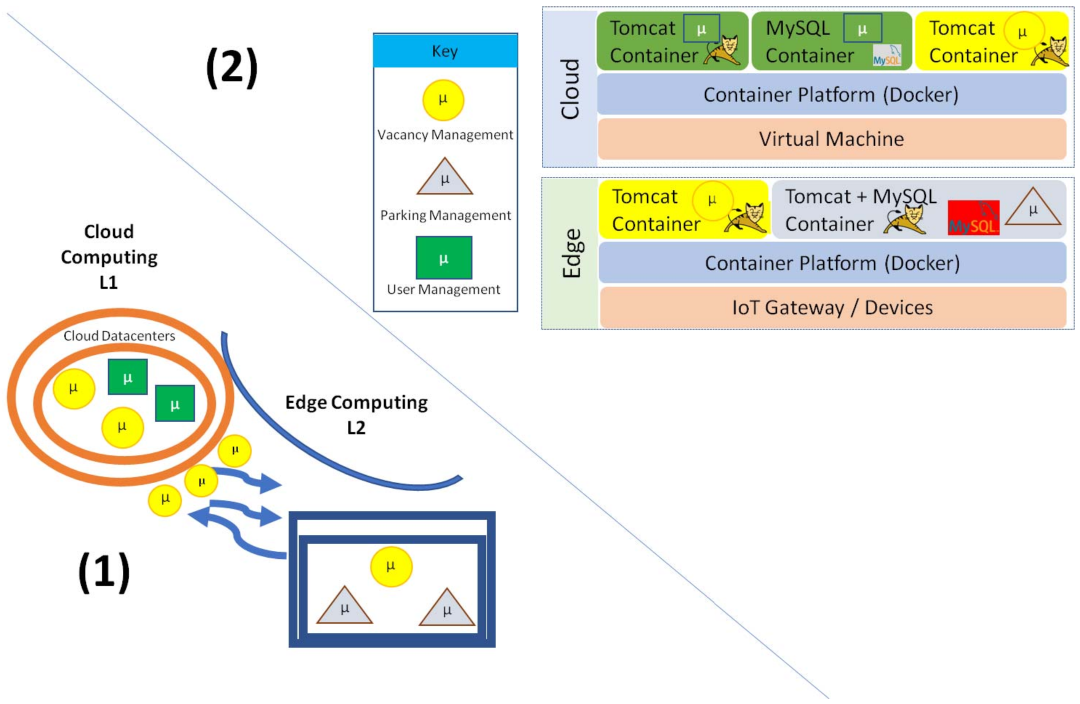

Figure. 1. (1) Osmotic Movement of Microservices across Cloud and Edge; (2) Parking Management Application

realize an IoT application.

Jiang et al [21] proposed E2MR, an algorithm focused on load balancing in a network for application execution in intelligent cities. The authors explore the energy-efficient networking problem with multiple constraints for smart city applications. The E2MR algorithm acts directly at the packet level to schedule the workloads to different devices. However, we explore load balancing between the services of an osmotic application to improve the scalability of smart city applications.

Our previous work [22] proposed a design idea of Osmotic Message-oriented middleware (MOM) which seamlessly integrated Cloud-based MOM into Edge computing. In other words, it aims to dynamically moving or provisioning message brokers from the Cloud or the Edge based on current demands. Although two papers share the some design concept i.e., Osmotic computing, this paper is the first system implementation of Osmotic computing that dynamically deploys and executes a chained microservices across Edge and Cloud.

IV. Osmotic Execution Framework

The highly dynamic nature and diversity of the Smart City environment brings a challenge of constructing a Smart City application, because the human behaviors are unpredictable and the infrastructures are provided by several stakeholders [23]. In other words, we need a platform that: (i) provides a flexible architecture to adopt new technologies, and (ii) supports new functional and non-functional requirements to suit the diversity of the multiple and constantly evolving city environments where they are deployed [18].

Osmotic computing leverages microservices to communicate and manage the network of distributed services in the osmotic network, and to ensure the security of the application. Inspired by this, we propose an *Osmotic Execution Framework* (see Figure 2) that combines elements of middleware platforms and osmotic computation to provide the main elements necessary to execute an osmotic application in a Smart City environment.

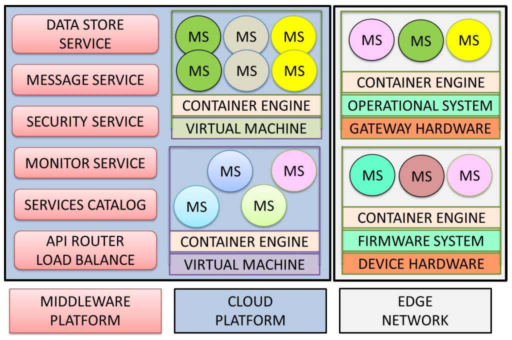

Figure. 2. Architectural View of the Osmotic Execution Framework

The architecture shown in Figure 2 represents a conceptual infrastructure necessary to execute osmotic microservices (MS) in a Smart City environment. The Cloud infrastructure provides the computing resources for deploying the Smart City platform and the osmotic microservices. A Smart City application consists of a set of services. For example, [17], [18], [24] explore the composition of multiple services or microservices in smart cities. In this paper, our osmotic application includes the following services:

1) **Services Catalog**: stores a catalog with the endpoints and their respective microservices. Moreover, it organizes the implantation and/or removal of the microservices. The cataloged microservices must have a state that indicates the availability and the parameters related to the performance of the microservice in order to allow Load Balancer to perform its actions (scale up/down).
2) **Monitor**: performs monitoring of composite services of the osmotic applications, allowing the creation of alert messages to indicate the need to scale up or down the application. It works in conjunction with Services Catalog and Load Balancer, promoting the orchestration of the microservices.
3) **API Router/Load Balancer**: is responsible for receiving all requests directed to the application, routing between the various endpoints registered in the Services Catalog. It also allows the monitoring of the response time for each request from the microservice API.
4) **Message**: deals with the communication between the various services of the middleware platform and the microservices of the osmotic application. For example, the Monitor sends an alert to the Service Catalog that indicates the need for scaling up the application. The Service Catalog initiates the creation and/or migration of a microservice.
5) **Security**: organizes the security of the microservices, offering authentication, authorization, proxies of APIs. It also stores the credentials for the access control in the Cloud and on the Edge.
6) **Data Store**: offers a service for storing data in the Cloud. It plays an important role of service migration and scheduling. It will be detailed in Section V.

In order to describe a preliminary implementation of the proposed architecture, the next sections describe some key components implemented like a prototype for this paper.

V. Container Structure for OEF

The challenge of executing an osmotic microservice is that the same microservice must be able to be migrated in real-time between Cloud and Edge while ensuring consistent status. For instance, if we move a microservice from Edge to Cloud, this relocated microservice must keep a status consistent with that it had when it checked out from Edge. Allied to this challenge we have to ensure the compatibility of the containers with the components of the middleware platforms.

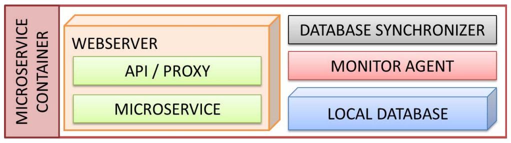

Figure. 3. Container Structure for OEF

In order to overcome these challenges, we propose a container infrastructure to support the execution of osmotic microservices for Smart City environments as shown in Figure 3. This infrastructure comprises the following elements:

1) **API/Proxy**: responsible for aggregating the calls to the microservice APIs, as well as providing a proxy that acts as a firewall to filter only correctly authenticated requests for access to the microservice.
2) **Local Database**: stores the data of the microservice within the strict scope of the container. The data to be stored reflect the direct performance of the microservice processing. Formally the architecture in microservices defines that each microservice is autonomous and decoupled, which implies a dedicated database for each microservice. In the Smart City environment due to the large amount of data being manipulated, the premise of using only a local database can create an obstacle to scalability. Due to this requirement, it is necessary to define a synchronization service between the local database and the Cloud storage service.
3) **Database Synchronizer**: synchronizes the local microservice database with the Cloud data storage service provided by the middleware platform.it acts like a cache of data from the microservice has easy and direct access without any dependency of network.
4) **Monitor Agent**: responds by monitoring the container and collects usage metrics for the infrastructure such as CPU usage, memory, request latency time, etc. It acts together with the Monitor service of the middleware platform.
5) **Microservice**: represents the microservice itself that will be executed on the container, being the main element of a container.

It is important to emphasize that some works [16], [25] already evidenced the use of containers for construction of microservices. Those proposals commonly use the elements API, Microservice and Local Database but the Database Synchronizer and Monitor Agent services are unpublished and specific elements for the execution of osmotic services in the Smart City environment. For example, each time the microservice is instantiated, the Database Synchronizer (DS) component should make a download of the data from the Data Storage Service hosted on the Cloud.

During the life cycle of the container while the microservice is running, DS acts on data synchronization between the local database and the Data Store Service. To enable better performance and reduce the network traffic load, the communication between the DS and the Data Store Service must use publish/subscribe communication through the Cloud Message Service. In addition, the Monitor Agent (MA) service must perform real-time monitoring of important metrics in the scheduling and migration of microservices.

Ideally the migration of the microservices, that is, of the containers should occur in an automated way transporting the microservice from the Cloud to the Edge or from the Edge to

Algorithm 1: : Osmotic control of load balancing

```
Input: $Latency \rightarrow L$; $Threshold\rightarrow S$;
OSControl($L$, $S$)
    // Deploy the proposed application
    $App \leftarrow$ Deploy()
    while $App$ is running do
        // Get the latency of SV
        // If the latency is greater than threshold
        $L \leftarrow$ getLatency()
        if $L > S$ then
            // Move SV to the Cloud
            Deploy($SV$, $Cloud$)
        end
        else
            // If SV is already deployed in the Cloud
            if Check($SV$, $Cloud$) then
                // Move SV to the Edge
                Deploy($SV$, $Edge$)
            end
        end
    end
```

the Cloud. The decision on the migration of microservices is controlled by Algorithm 1 that summarizes the control workflow of load balancing (OSControl). OSControl performs load balancing that aims to automatically balance the workload of *job selection service*. To this end, OSControl continuously evaluates the parameters: latency time (L) value and maximum latency time limit (i.e. Threshold (S)). Those parameters are defined by the application's QoS attributes and the control occurs for as long as the application is running (line 4). Once the application is running, OSControl compares whether the latency obtained for the Selection Vacancies microservice is greater than the defined threshold (line 8). If the latency exceeds the threshold, the service will be migrated from Edge to Cloud (line 10). If the latency is not greater than the threshold, whether the service is deployed to the Cloud (line 14) is checked in order to reverse the migration. If yes, SV will be migrated back to the Edge. In the current OEF architecture the OSControl is plugged into the monitoring component. Our previous paper [26] discusses the details of the monitoring component.

Migration indeed involves a flow of well-established steps. So, data synchronization and the redirection of requisitions are extremely important. Data synchronization is possible through the interaction between the Data Store Service and the Database Synchronyzer. The redirection of requests is implemented in the API Route/Load Balance. To clarify the concepts, Figure 4 shows one scenario of migration from Edge to Cloud.

We assume that the microservice (i.e. *job selection service*) that is running on the Edge exceeds the threshold of latency time as defined by Quality of Service level attribute (Step 1). This event is perceived by the Monitor Service, which sends a message to the Service Catalog (Step 2) in order to initiate

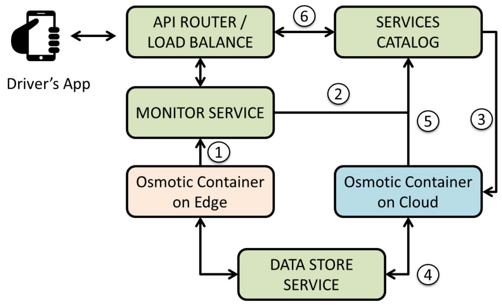
Driver's App
API ROUTER / LOAD BALANCE
SERVICES CATALOG
MONITOR SERVICE
Osmotic Container on Edge
Osmotic Container on Cloud
DATA STORE SERVICE
1
2
3
4
5
6

Figure. 4. Steps to Migrate from Edge to Cloud

the deployment of the microservice in the Cloud (Step 3). Once the Cloud container is running and the microservice has synchronized the database with the Data Store service (Step 4 and 5) the Service Catalog notifies the Route API to route the requests to the Cloud microservice rather than to the Edge (6). Finally, the Service Catalog can start removing the container from the Edge. It is important to note that removing the container from the Edge can only occur when the container's Database Synchronizer signals that all content in the Local Database is properly synchronized with the Data Store Service.

VI. Implementation

An evaluation was carried out to validate and evaluate the Osmotic Execution Framework (see Sections IV and V), as well as to investigate how osmotic computation can influence the load balance. Thus, an initial version of a microservice-based Smart Parking application (as discussed in Section II) was developed and deployed an Osmotic Execution Framework instance across Edge and Cloud nodes. Subsequently, a set of load tests were performed to verify the behavior of microservices running on both Edge and Cloud in Section VII.

A. Smart Parking as Osmotic Application

The Smart Parking application (as depicted in Section 1) searches for real-time mapping of parking spaces available in a city. A driver accesses the Smart Parking application to know the best places available according to his personal preferences. 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 selection vacancies service searches for possible available positions, Smart Parking alerts the driver the available vacancies. With this scenario, three microservices specified in Section II have been implemented, namely: *User Management, Selection Vacancies* and *Parking Management*.

User Management (UM) is the service that is deployed in the Cloud. It is responsible for storing user data as well as for providing system communication with the user. User interaction can occur via an application deployed on his/her smart phone. *Selection Vacancies* (*SV*) continuously receives UM job requisitions. The incoming requisitions are processed through a selection algorithm that continuously queries the

Parking Management microservice to check the status of the vacancies. Once the vacancies are defined the *SV* notifies the *UM*. *Parking Management* (*PM*) continuously monitors vacancy status. *UM*, *SV* and *UM* are microservices deployed on the Osmotic Execution Framework running on the Edge and Cloud. In particular, *PM* is deployed on the Edge and communicates with the IoT sensors that identify the occupation or the release of a vacancy. Theoretically, PM must be replicated between the various parking lots as many times as necessary.

Inter-service communication occurs through a REST API to access its functionality. For experimental evaluation, specific API calls were implemented for each service, namely: for the *UM* a call to query the user data; for the *PM*, a call to check 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 microservices were developed in Java, running on an Apache Tomcat server (http://tomcat.apache.org/) as web server. For the *UM* and *PM* microservice that require persistence of contextual data of their entities, the Local Database was implemented using MongoDB (https://www.mongodb.com/). The smart parking application with the three microservices was deployed in containers. The containers were built for execution on the Docker platform[1] and follow the structure defined in Section V.

The use of Containers Docker allows the use of two possible deployment cases, namely: a container for each element of architecture microservice execution (Web Server, Database, Database Synchronizer) or several containers for each element. In the first case ($F1$) the same container is installed with all the components used by the microservice. In the second case ($F2$), each component is installed in its own container, that is, the Web Server will run on one container and the Local Database will be in another, etc. $F2$ is the 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. This work makes use of $F1$ since it represents a more simplified scenario for osmotic microservice observation since the management of multiple containers for each microservice can add more complexity in the microservice composition. It is important to note that the use of the same container architecture (i.e both Cloud and Edge microservices are implanted on the Docker) allows the *SV* environment of execution to be made more uniform. Keeping this in mind, Table I shows all microservices implemented for experimentation. In the Cloud environment, the UM micro service was instantiated by using just one container with Tomcat and MongoDB. Similarly, the PM microservice is instantiated with a similar Docker Image. Finally, the SV microservice requires two Tomcat Docker Images. The reason we used two different images for SV is the operation system required for Raspberry Pi is different to Cloud VM. Unlike the Docker images, the version of Tomcat used was the same on both the Edge and the Cloud. The SV does not require a local database because it does not store state entities, it only performs processing.

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

TABLE I
SMART PARKING SERVICES DEPLOYED AT DOCKER

Environment	Microservice	Containers
Cloud	User Management	Tomcat + MongoDB
Cloud	Selection Vacancies	Tomcat
RaspberryPi	Selection Vacancies	Tomcat
RaspberryPi	Parking Management	Tomcat + MongoBD

VII. EVALUATION

We used Apache JMeter (https://jmeter.apache.org/) to generate HTTP requests to test and validate the Osmotic Execution Framework capability. The JMeter test cases are presented in Table I. The tests consisted of performing 10, 100, and 500 simultaneous requests to the microservices running on the Osmotic Execution Framework at a fixed interval of 5 minutes. Notably, when we test the Raspberry Pi (https://www.raspberrypi.org/) 1 Model B with 1000 requests, the device was overloaded and stopped responding. Therefore, we set the maximal number of the requests to be 500.

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 Raspberry Pi on the Edge. The Cloud used a virtual machine that runs a Linux system with Ubuntu (https://www.ubuntu.com/), version 14.03, on virtual hardware with a configuration of 2 vCPU, 4 GB of memory and 20 GB of disk. Docker platform version 1.10 was installed on the virtual machine. The UM service deployment was based on the MongoDB 3.2 image https://hub.docker.com/_/mongo/ obtained via Docker HUB. This image included a version of the Java virtual machine, version 8 and the Tomcat server version 7. We used a specific image for Tomcat 7 (https://hub.docker.com/r/dordoka/rpi-tomcat/) and included a MongoDB 3.2 for the Edge environment. The Edge environment was simulated by the RaspberryPi 1 Model B that runs a Raspbian system in a configuration of: 1 CPU core (700 Mhz clock), 512 MB of RAM and a 4GB SD memory. For Selection Vacancies Service was used two images of Tomcat 7 on Cloud: (https://hub.docker.com/_/tomcat/), on Edge: (https://hub.docker.com/r/dordoka/rpi-tomcat/). The other elements (Database Synchronizer, API Proxy, and the Monitor Agent) of the Osmotic Execution Framework have been implemented in Java. All the communication between the components was developed through the use of API in REST style.

A. Latency Time Results

Latency is one of the most important metrics that affects user satisfaction [18]. In this section, we focus on the experimental analysis of the latency of the microservices. Other metrics such as CPU usage, memory usage, the amount of network traffic will be briefly reported in the discussion subsection.

In this experiment, we made the following three observations: 1 – observe the behavior of microservices, especially osmotics microservice (Selection Vacancies), 2 – validate the proposed architecture in the construction of osmotic applications for Smart City, and 3 – obtain the latency caused by migrating a microservice from Edge to Cloud. All the experimental results in the following are the average of ten executions.

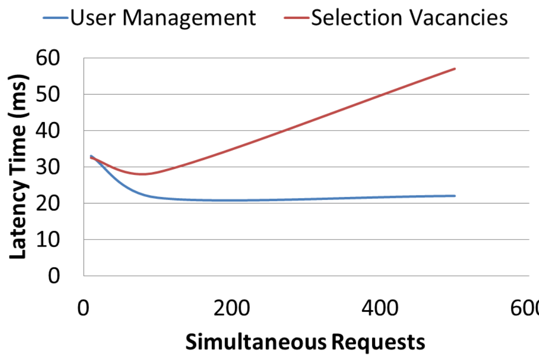

Figure. 5. Dispersion of medium Latency Time for Microservices on Cloud

For the Cloud environment (see Figure 5) the two microservices evaluated presented very similar behavior regarding 10 and 100 requests. For the test of 10 requests for the UM and for the SV the average latency is 33 and 32.5 milliseconds, respectively. The results for the 100 requests are 21.5 (UM) and 28.5 (SV) milliseconds. The reason that the latency of 10 requests is higher than that of 100 requests is because both UM and SV are not saturated when the request number is 10. Thereafter, the latency of SV increases significantly when the number of requests increases. This is because, when SV process a request, it needs to interact with both SV and PM. Therefore, the latency of processing a request by SV is much longer than processing a request by UM.

As for the Edge environment, the results obtained for the PM are: 133 ms (10 req.), 883.5 ms (100 req.), 4786 ms (500 req.). The latency increases with the increasing number of requests. For SV, latency significantly goes up when the number of requests increases from 10 to 100. This is because the Edge has limited computing resources and will take more time to process the requests, compared to the case in the Cloud. After that, when the number of requests increases, the latency increases slowly. This behavior could be explained due to the device reached your own full capacity of processing, resulting in dropping some requests.

Lastly, the load pressure tests were applied to the SV microservices. In this experiment, we want to simulate the case where the number of requests exceeds the threshold, our system will automatically migrate SV into the Cloud. To this end, we first start SV on Edge. When the latency reaches 6 seconds, the API Route will redirect new requests to the Cloud. In previous experiments, we observe that when the latency reaches 7.3 seconds the Edge node will not able to

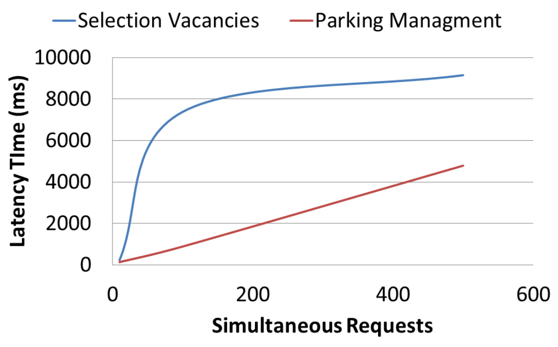

Figure. 6. Dispersion of Medium Latency Time for Microservices on Edge

process all requests. Thus, we set the threshold as 6 seconds. Figure 7 shows the expected behavior and proves that the use of osmotic services can ensure the low latency of the smart parking application through the new dynamic load balancing method.

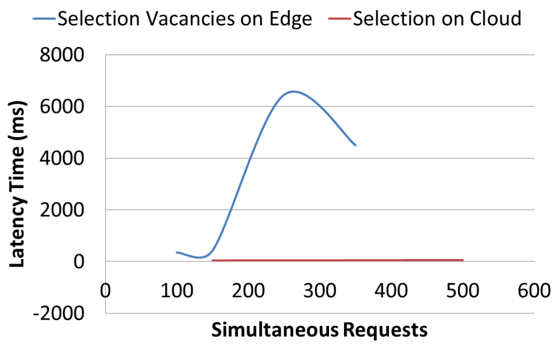

Figure. 7. Dispersion of Medium Latency Time for Microservices in Migration Scenario

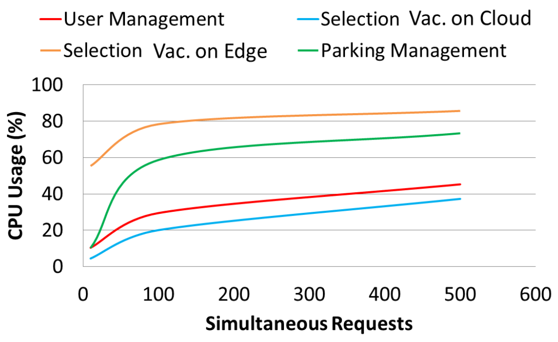

Figure. 8. Percentage of CPU use for all microservices

B. Discussion

The above results demonstrate the effectiveness of using the Osmotic Execution Framework for deploying an Osmotic Application in the Smart City environment. It also shows that

Osmotic Computation can dynamically balance the workload of the microservices that are deployed on Edge and Cloud. Due to the limited computing resources on Edge, the CPU usage increase significantly with the increasing number of requests as shown in Figure 8. However, this workload only slightly increase the CPU usage of the Cloud node.

In future work, one possible analysis to be carried out is to estimate the ratio of how many copies of microservices at the Edge should be deployed in order to reduce the impact of this on the performance of the application. The results obtained for variation of memory and network traffic are similar to those observed for CPU. That is, the variation is greater in the microservices implanted in the border. It is worth noting that the variation in the consumption of memory is smaller when compared to the one of the use of the CPU while the number of bytes traveled in the tests of 500 requests is much higher than in the tests of 10 and 100. This indicates that in this type of scenario the quality of the network has a high impact on the performance of the microservices.

VIII. Conclusion

This paper describes an Osmotic execution framework that partitions microservices to a distributed environment including Edge and Cloud. The core contribution of this paper is that we propose a novel microservice migration method to balance the workload of the microservices. Importantly, our framework can automatically mitigate the workload of microservices deployed on the Edge when their computing resources are limited. We evaluate our framework with a real Smart City application, namely, smart parking. This optimistic application shows that we can efficiently utilize the computing resources from the Edge while ensuring low latency.

IX. Acknowledgment

This work is supported by the SmartMetropolis Project[2], Flood-PREPARED (RES/0541/7295), LANDSLIP(RES/0541/7295) and EPSRC/UK STRATA projects. Nélio Cacho is supported in part by CAPES-Brazil (88881.119424/2016-01).

References

[1] 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.

[2] J. Soldatos, N. Kefalakis, M. Hauswirth, M. Serrano, J.-P. Calbimonte, M. Riahi, K. Aberer, P. P. Jayaraman, A. Zaslavsky, I. P. Žarko, *et al.*, "Openiot: Open source internet-of-things in the cloud," in *Interoperability and open-source solutions for the internet of things*, pp. 13–25, Springer, 2015.

[3] B. Cheng, S. Longo, F. Cirillo, M. Bauer, and E. Kovacs, "Building a big data platform for smart cities: Experience and lessons from santander," in *Big Data (BigData Congress), 2015 IEEE International Congress on*, pp. 592–599, IEEE, 2015.

[4] W. Apolinarski, U. Iqbal, and J. X. Parreira, "The gambas middleware and sdk for smart city applications," in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2014 IEEE International Conference on*, pp. 117–122, IEEE, 2014.

[5] T. Zahariadis, A. Papadakis, F. Alvarez, J. Gonzalez, F. Lopez, F. Facca, and Y. Al-Hazmi, "Fiware lab: managing resources and services in a cloud federation supporting future internet applications," in *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pp. 792–799, IEEE, 2014.

[6] Z. Wen, R. Yang, P. Garraghan, T. Lin, J. Xu, and M. Rovatsos, "Fog orchestration for internet of things services," *IEEE Internet Computing*, vol. 21, no. 2, pp. 16–24, 2017.

[7] G. Cormode, M. Garofalakis, P. J. Haas, C. Jermaine, *et al.*, "Synopses for massive data: Samples, histograms, wavelets, sketches," *Foundations and Trends® in Databases*, vol. 4, no. 1–3, pp. 1–294, 2011.

[8] R. Ranjan, O. Rana, S. Nepal, M. Yousif, P. James, Z. Wen, S. Barr, P. Watson, P. P. Jayaraman, D. Georgakopoulos, *et al.*, "The next grand challenges: Integrating the internet of things and data science," *IEEE Cloud Computing*, vol. 5, no. 3, pp. 12–26, 2018.

[9] H. Wang, J. Gong, Y. Zhuang, H. Shen, and J. Lach, "Healthedge: Task scheduling for edge computing with health emergency and human behavior consideration in smart homes," in *Big Data (Big Data), 2017 IEEE International Conference on*, pp. 1213–1222, IEEE, 2017.

[10] 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, no. 6, pp. 76–83, 2016.

[11] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[12] J. Traub, S. Breß, T. Rabl, A. Katsifodimos, and V. Markl, "Optimized on-demand data streaming from sensor nodes," in *Proceedings of the 2017 Symposium on Cloud Computing*, pp. 586–597, ACM, 2017.

[13] Z. Wen, P. Bhatotia, R. Chen, M. Lee, *et al.*, "Approxiot: Approximate analytics for edge computing," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2018.

[14] 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.

[15] 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.

[16] H. Visti, T. Kiss, G. Terstyanszky, G. Gesmier, and S. Winter, "Micado–towards a microservice-based cloud application-level dynamic orchestrator," *PeerJ PrePrints*, vol. 2016, no. 10, 2016.

[17] A. Krylovskiy, M. Jahn, and E. Patti, "Designing a smart city internet of things platform with microservice architecture," in *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*, pp. 25–30, IEEE, 2015.

[18] A. d. M. Del Esposte, F. Kon, F. M. Costa, and N. Lago, "Interscity: A scalable microservice-based open source platform for smart cities,"

[19] K. Khanda, D. Salikhov, K. Gusmanov, M. Mazzara, and N. Mavridis, "Microservice-based iot for smart buildings," in *Advanced Information Networking and Applications Workshops (WAINA), 2017 31st International Conference on*, pp. 302–308, IEEE, 2017.

[20] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computinga key technology towards 5g," *ETSI white paper*, vol. 11, no. 11, pp. 1–16, 2015.

[21] D. Jiang, P. Zhang, Z. Lv, and H. Song, "Energy-efficient multi-constraint routing algorithm with load balancing for smart city applications," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 1437–1447, 2016.

[22] T. Rausch, S. Dustdar, and R. Ranjan, "Osmotic message-oriented middleware for the internet of things," *IEEE Cloud Computing*, vol. 5, no. 2, pp. 17–25, 2018.

[23] M. Vögler, J. M. Schleicher, C. Inzinger, S. Dustdar, and R. Ranjan, "Migrating smart city applications to the cloud," *IEEE Cloud Computing*, vol. 3, no. 2, pp. 72–79, 2016.

[24] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, "Open issues in scheduling microservices in the cloud," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 81–88, 2016.

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

[26] A. Souza, N. Cacho, P. P. Jayaraman, R. Ranjan, A. Romanovsky, and A. Noor, "Osmotic monitoring of microservices between the edge and cloud," in *20th IEEE International Conference on High Performance Computing and Communications*, 2018.

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