Modeling and Emulation of an Osmotic Computing Ecosystem using OsmoticToolkit

Alina Buzachis
University of Messina
Messina, Italy
abuzachis@unime.it

Daiana Boruta
Universidad Nacional del Sur
Bahía Blanca, Argentina
daianambtw@gmail.com

Massimo Villari
University of Messina
Messina, Italy
mvillari@unime.it

Josef Spillner
Zurich University of Applied Sciences
Winterthur, Switzerland
josef.spillner@zhaw.ch

ABSTRACT

Digital services are increasingly becoming cyber-physical and osmotic, combining Cloud resources with Fog, Edge, and IoT devices. This trend can be observed in the e-health domain or in smart city applications where the location of software deployments and data processing matters. Before such applications go live, careful planning with real system emulation is necessary. We claim that the OsmoticToolkit, although in the early stages, is the first emulation environment designed to address this challenge. In this paper, we introduce the emulator's functionalities and validate experimentally with an e-health scenario, using a reference deployment of a microservice-based hospital application. The experimental results carried out show its effectiveness providing valuable support for understanding the impact on resources, workloads, and Quality of Service requirements within Cloud-Edge/Fog-IoT scenarios while preserving the users' Service Level Agreements (SLAs).

CCS CONCEPTS

• **Networks** → **Network services**; Network reliability; • **Computing methodologies** → *Simulation tools*; • **Computer systems organization** → *Dependable and fault-tolerant systems and networks.*

KEYWORDS

Cloud Computing, Osmotic Computing, Emulation, SDN, Microservices, Deployment, Orchestration

ACSW '21, February 1–5, 2021, Dunedin, New Zealand

ACM ISBN 978-1-4503-8956-3/21/02…$15.00
https://doi.org/10.1145/3437378.3444366

ACM Reference Format:
Alina Buzachis, Daiana Boruta, Massimo Villari, and Josef Spillner. 2021. Modeling and Emulation of an Osmotic Computing Ecosystem using OsmoticToolkit. In *Australasian Computer Science Week Multiconference (ACSW'21), February 1–5, 2021, Dunedin, New Zealand.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3437378.3444366

1 INTRODUCTION

Internet of Things (IoT) is a profound technology evolution incorporating billions of devices (sensors, RFIDs, smartphones, and wearables) owned by different organizations and people who are deploying and using them for pervasive digital services. IoT-Business-News forecasts 24 billion connected things generating $1.5 trillion in 2030 [1]. Their number, capabilities, scope of use and data volume, keep growing & changing rapidly. This leads to higher complexity in IoT applications. Thus, new distributed computing paradigms, such as Edge Computing or IoT-Cloud Computing, have been investigated to extend IoT resources into centralized data centers (e.g., clouds) or at the edge of IoT systems (e.g., edge micro datacenters). Among the most promising ones is Osmotic Computing (OC), motivated by the lack of a scalable, interoperable, configurable solution for delivering IoT applications in complex, heterogeneous and dynamic computing environments. The OC paradigm [3] looks at the opportunistic management of IoT MicroELements (MELs), i.e., MELs running on IoT, to improve the Quality of Service (QoS) and networking management, interoperability, and efficiency of next-generation IoT applications.

The main issue arising in using such combined computing models to support IoT applications is the management of different physical/virtual infrastructures (e.g., data centers, edge devices, IoT devices & gateways) according to specific application/service requirements (e.g., latency, data volume, responsivity, processing delay, privacy). In particular, it is hard to determine a priori how to deploy the MELs composing IoT applications into different infrastructures – since resource availability, system load, and connectivity features can unpredictably vary over time. OC provides convergence

and holistic planning for IoT, Edge, Fog, and Cloud Computing technologies in this scenario. It allows to manage resources available across such systems driven by specific application requirements.

In this paper, we present an OC emulation tool called *OsmoticToolkit* that executes workflows based on MELs in particular conditions where the edge has limited computation and networking capabilities. We evaluate the work with the case of a rural medical lab with limited processing power and infrequent ability to use a cloud service. The toolkit provides valuable support for understanding the impact of processing power, workloads, and QoS requirements while preserving the users' Service Level Agreements (SLAs).

The paper is organized as follows. After the background and related work of Sec. 2, we provide motivations (Sec. 3) and explain the tool design in Sec. 4, followed by the implementation in Sec. 5 and performance evaluation in Sec. 6. Finally, Sec. 7 highlights the advantages of OsmoticToolkit and reasons about future directions.

2 BACKGROUND AND RELATED WORK

Planning and testing applications in distributed computing environments that involve a high level of heterogeneity and complexity is costly since the provisioning and management of needed hardware are very expensive. In recent years, simulation techniques have been proven to be a partial solution for investigating different aspects of complex osmotic systems, e.g., service configuration and deployment, resource placement, or management strategies. Cost-effective emulation tools based on controlled service execution further align results with reality. To evaluate the current state of the art, we define four key criteria: (i) Hybrid Cloud-Edge/Fog-IoT Architecture, (ii) Dynamic Infrastructure Topology (modeling of physical networks and virtual topologies), (iii) Resource Provisioning Approach, and (iv) Real Application execution.

Tool	Hybrid	Dynamic	Resources	Execution
CloudSim	|	|	allocation	|
iFogSim	iot/fog	|	alloc+mon	|
EdgeCloudSim	edge	mobility	alloc+mon	|
myiFogSim	fog	migration	alloc+mon	|
IoTSim	iot/cloud	|	allocation	|
YAFS	fog	app	alloc+fail	|
Sphere	edge	workloads	orchestration	|
EmuFog	fog	|	placement	containers
Fogbed	fog	volatility	|	containers
MockFog	fog	|	allocation	VM-based

Table 1: Summary of simulation and emulation tools

2.1 Simulation Tools

While simulators cannot execute real applications, their designs are of interest to ensure that our emulator meets functional expectations on expressible scenarios. They simulate hybrid Cloud Computing and Edge based on simplified models. Many Edge Computing simulators extend CloudSim [4], such as iFogSim [8]. It provides an evaluation platform for resource allocation policies. A limitation of iFogSim is that the location of end devices is static and cannot be updated; further, it is limited to the Discrete Event Simulators (DES) and has poor scalability because of the CloudSim characteristics.

Similarly, EdgeCloudSim [18] extends CloudSim. In contrast to iFogSim, it is focused on a more dynamic and realistic investigation of service usage and implements mobility models for mobile devices. MyiFogSim [13] extends iFogSim to support mobility through the migration of VMs between cloudlets. IoTSim [21] also extends CloudSim. It emphasizes the processing performance of large IoT applications that process huge amounts of data. As a result, it adds storage and big data processing layers with map-reduce cloudlets to CloudSim. EdgeCloudSim and IoTSim both inherit the same scalability and DES limitations as iFogSim.

Yet Another Fog Simulator (YAFS) [11] is a DES for Cloud & Fog networks. Its primary focus is the performance evaluation of placement, scheduling, and routing strategies. Applications are modeled as a set of modules that run services, following the concept defined by iFogSim. Sphere [7] extends SCORE [6] and allows creating a cloudlet network based on graphs, generating dynamic and parallel workloads, and specifying the geographic location, resource density, and deployment requirements. However, it does not support the nodes' mobility and lacks the migration model of a workload.

Simulators such as iFogSim, CloudSim, and YAFS support the dynamism and on-demand requirements of Fog services/applications via VM elasticity and migration, federation policies, and computational clustering nodes. EdgeCloudSim only supports federation and scalability between nodes of the same tier (only Cloud or only Fog), which means that it is impossible to achieve a proper orchestration along with the Cloud to Fog continuum.

2.2 Emulation Tools

EmuFog [14] is an emulation framework for Fog Computing built on top of MaxiNet [19]. It emulates Fog nodes and makes use of Docker to run applications. EmuFog is more realistic than simulation tools, implementing a Fog node placement algorithm based on arbitrary latency costs to the connections between hosts and switches. It does not support the mobility of clients and fog nodes. Fogbed [5] extends the Mininet network emulator. In contrast to EmuFog, it uses Docker containers to run virtual nodes and allows developers to dynamically add, connect, and remove nodes from the topology. This feature allows investigating real-world Fog infrastructures, where Cloud services are provided closer to the network edge. It does not support mobility, security,

fault tolerance, scalability, and reliability and does not implement any resource providing model for the application services. The application used to evaluate the emulator is not faithful to real-case scenarios and lacks configurability and extensibility. It is strongly-coupled with the virtual node code. EmuFog and Fogbed have scalability support regarding the communication and topology infrastructure but lack strategies to deal with applications' on-demand requirements inside computational nodes.

Finally, MockFog [9] allows the emulation of a Fog Computing infrastructure in arbitrary Cloud environments. It creates a VM for every node lacking scalability and being expensive when implementing an infrastructure model with a large number of nodes. It also has problems when smaller devices are involved, as they cannot be accurately emulated.

Concerning the service execution plan and resource provisioning in hybrid Cloud-Fog-IoT environments, several optimization solutions for service deployment have been investigated, each of them focusing on different target variables. In [17], the authors investigate a solution for maximizing the number of services deployed on Fog devices by applying heuristics to solve their service placement problem based on collected response times. The approach proposed by the authors is not realistic as they assume that each service of an application can be executed independently from workflow structures with chained output-input links. In [12], the authors address service provisioning as a Delay and Payment optimization problem, which is the trade-off among energy consumption, delay performance, and payment cost when deploying services. In [20], the authors propose a distributed alternating direction method of multipliers to approach the allocation as a trade-off between the users' Quality of Experience (QoE) and the fog nodes' power efficiency.

Table 1 summarises the capabilities of related simulation and emulation approaches regarding the four key criteria mentioned before.

3 MOTIVATIONS AND REQUIREMENTS

Osmotic capabilities are of increasing importance when considering the growing digitalization of life. To counter the pandemic in 2020, several national governments have released software applications with workflows encompassing mobile phones, telecom carriers and cloud services. The degree to which processing logic has been placed on one side depends on political strategies. Due to their volatility, engineering effort can be saved from a technical perspective when mechanism and policy are properly separated and the placement gains flexibility. The mechanism then entails a decomposition of the application into either resource-bound or portable parts, the MELs, that can be implemented as cloud functions, containers, other MicroService technologies (MS) and associated MicroData (MD) representations. The assignment of MELs to computing resources can become dynamic at deployment time. It requires osmotic management, where MELs can move across different infrastructures, based on several potential triggers (e.g., performance, networking, security/privacy, or cost-oriented). The Software-Defined Membrane (SDMem) in OC enforces these concepts filtering the MELs flows in the system.

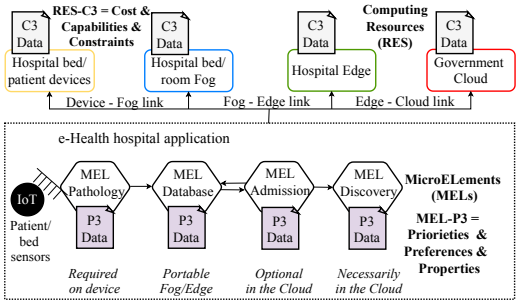


Figure 1: Scenario of osmotic e-health application

Let us consider the case of a rural medical lab with limited processing power and network access. The application deployment needs to prioritize which MELs can be deployed locally if there are resource contentions and otherwise needs to optimize within the given degrees of freedom among all the portable MELs. Hence, the application consists of a mandatory on-site part on health testing, a portable part on image detection that can run either on-site or in the cloud, and an optional part on recommending further tests that run in a particular cloud environment. Figure 1 explains this scenario. It involves descriptors of application requirements, including deployment priorities (P3), and corresponding hosting capabilities (C3), facilitating deployment dynamics.

In contrast to the streamlined onboarding of software applications in clouds, the decomposition and description of applications for such dynamic scenarios is currently a challenging engineering task. To give application engineers the ability to prepare, using an emulator will save precious time and effort and facilitate resource planning. To overcome the limitations of existing emulation tools, we require OsmoticToolkit to provide the following technological advances:

(1) Combined inherent support for all of the four key criteria outlined in Table 1 by design.
(2) Well-defined usage procedure with explicit infrastructure and application modeling, infrastructure instantiation and application pipeline deployment.
(3) Service-oriented integration with APIs/CLIs to fit into automated osmotic and cloud-native systems.

4 OSMOTICTOOLKIT WORKFLOW DESIGN

This section highlights OsmoticToolkit's features and overviews a high-level design workflow outlining the emulation abstractions, the toolkit core components and their interactions.

4.1 Design Principles

OsmoticToolkit should support four main features:

Hybrid Topology. OC ecosystems consist of hybrid complex IoT-oriented computing systems where both resource-constrained Edge/Fog nodes and Cloud-hosted services in public/private, hybrid or multi-cloud are involved. The generation of such topologies should be realistic with a high degree of confidence, allowing to assign capacities (e.g., CPU, memory) and capabilities (e.g., hosted services, applications) for each infrastructure component trivially.

Dynamicity. In OC, computation is dynamically distributed across nodes based on QoS requirements and available infrastructure resources. Particularly, services with short lifecycles are frequently instantiated and offloaded. It also occurs at the lowest level of the infrastructure, where Edge and IoT nodes may join and permanently leave the network according to service usage, failures, policies, and maintenance operations. OsmoticToolkit should provide a holistic approach for managing the network infrastructure and application.

Resource Provisioning and Orchestration. Applications range from simple IoT-based sensing to complex data processing inherent to e-health or smart city systems with different QoS and SLA (e.g., location/latency awareness, security levels, heterogeneity, interoperability), processing (e.g., batch, real-time), mobility. OsmoticToolkit should consider these aspects during the orchestration allowing dynamic and flexible resource provisioning and monitoring mechanisms.

Execution. OsmoticToolkit should allow the execution of realistic applications on top of the infrastructure topology. This feature should minimize the effort in preparing applications, avoiding costly changes in stack and tools.

4.2 OsmoticToolkit Infrastructure Model

Figure 2 illustrates the high-level design workflow of OsmoticToolkit. In this scenario, the DevOps Engineer (DOE) is involved in several phases, as explained in the following.

Phase #1: Infrastructure Modeling. An OC ecosystem comprises Infrastructure Elements (IEs) such as compute nodes, Network Elements (NEs) such as switches and routers, and Application Elements (AEs) deployed on top of the infrastructure, at different levels. The infrastructure topology is modeled as directed graph T = (V, E) where V is a set whose elements are called vertices (e.g., IE), and E is a set of paired

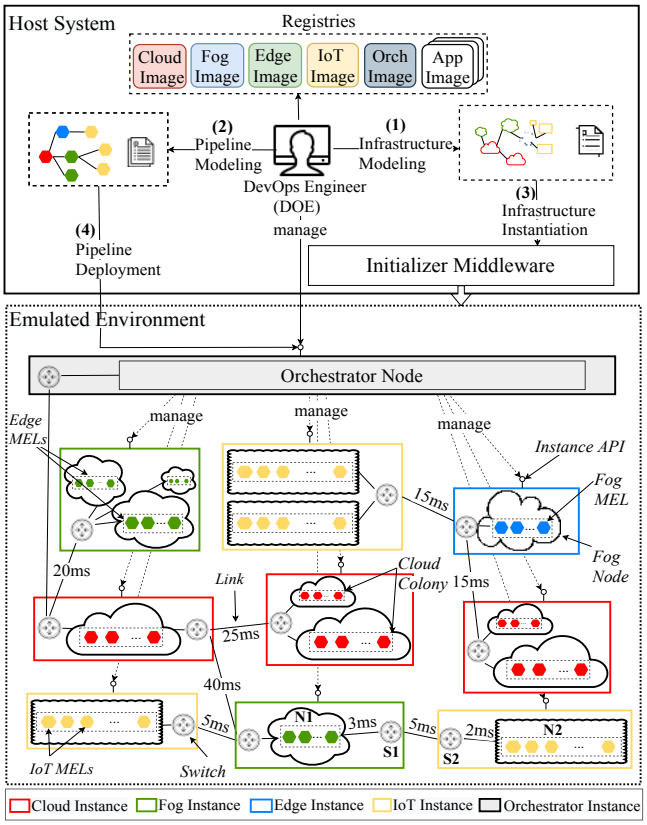


Figure 2: OsmoticToolkit general workflow.

vertices, whose elements are called links. Each IE is characterized by different computing properties (e.g., CPU, memory), while different network parameters characterize each link and thus NEs (e.g., latency, bandwidth, packet loss). As exemplified in Figure 2, if the link between N1 and S1 (N1-S1) has a delay of 3 ms, S1-S2 has 5 ms, and S2-N2 has 2 ms, the overall delay is 10 ms. During this phase, the DOE starts with such a graph specification before assigning properties to vertices and links using a template primitive. OsmoticToolkit relies on pre-configured container images, e.g., Cloud, Fog, Edge, and IoT, retrieved as IEs the toolkit's registry. Thus, each IE in the emulated network executes in an independent and isolated way, increasing the emulation's realism and affording behavior similar to that in production infrastructures.

Phase #2: Application Modeling. Similar to Infrastructure Modeling, applications deployed in an osmotic ecosystem are structured as graph P = (V, E), where V = MELs and E = interconnections. OsmoticToolkit associates the concept of the pipeline to an application. Namely, the pipeline's anatomy describes MELs properties and how they are interconnected. The DOE defines independent pipelines inside the toolkit and interacts with each separately. Each pipeline is described

within a template primitive. For each MEL inside the pipeline, the DOE can specify different resource requirements, constraints, and scheduling policies.

Phase #3: Infrastructure Instantiation. During this phase, the Initializer Middleware loads the template primitive containing the infrastructure description and instantiates it by deploying instances, e.g., for Cloud (CI), Fog (FI), Edge (EI) and IoT (IoTI), as well as Compute or Managed Nodes (MNs), switches, and links into an emulated environment. An example of a running environment with 9 instances is shown in Figure 2.

The instance abstraction allows the management of MNs and switches as a single entity. Generally, each instance involves one or more switches and MNs. An instance containing more of those is called a colony, e.g., Cloud Colony (see Figure 2). This colony composes a new and isolated network slice. It is relevant in mobile networks, where the limited radio resources are shared among multiple users that experience variable radio quality conditions over time. To properly control and manage the QoE in the network slice, the NEs are adapted to the different service requirements, and the applications adjust the configuration to the network capabilities over time dynamically.

Each instance has associated a resource model defining the amount of computing resources distributed among its MNs. Each MN in the infrastructure model is mapped to a running container. The resource model allows the DOE to apply limitations that impose each instance's available resources according to a specific scenario.

To efficiently control and manage this complex osmotic ecosystem, an effective control system becomes essential. The Orchestrator Node (ON) handles this system. During this phase, the ON is instantiated within an Orchestrator Instance (OI) and it is the point of entry for DOE via API endpoints and automation handlers. Thus, it manages the entire infrastructure, deploys pipelines or offloads MELs, spawns new nodes, and forwards configuration details.

Phase #4: Pipeline Deployment. The application is deployed on the emulated infrastructure through ON's Instance API endpoint for this phase. The ON firstly evaluates the MEL's predefined constraints and scheduling policies specified by the DOE within the template. This filtering step allows selecting a set of nodes obeying the specified restrictions. Next, it evaluates the computation requirements for each MEL. It generates an optimal execution plan for the pipeline for describing the MELs contextualization across the Cloud, Fog, Edge, Fog, and IoT MNs through an optimization algorithm. Namely, it assigns each MN satisfying the computation requirements of one or more MELs by minimizing a specific cost function. It is assumed that the ON has full control over which MELs are executed on each instance. Finally, the MELs are instantiated and run in Docker containers on top of the MNs (i.e., Docker-in-Docker) according to the previously generated optimal scheduling plan. The DOE interacts with the pipeline using the orchestrator APIs by controlling the MELs status, performing updates, tearing down the pipeline, or deploying a new one.

5 EMULATOR IMPLEMENTATION

This section discusses the architectural components, along with their interactions (illustrated in Figure 3), and motivates the set of technologies used to implement OsmoticToolkit. The technological choices are constrained by several non-functional requirements such as flexibility, ease of use, cost-effectiveness, scalability, extensibility.

5.1 Core Components and APIs

OsmoticToolkit core. The toolkit is based on Containernet [15] that extends the Mininet emulation framework by adding Docker containers at runtime as compute instances within the emulated topology. We chose Containernet/Mininet because they are highly prevalent in the distributed computing community, open-source, scalable, easily extensible, and flexible. The core offers three convenient APIs.

Topology API. This API is based on the Mininet Python API. It interacts with the core to allow the DOE to generate different topologies straightforwardly. Switches are implemented leveraging Open vSwitch. Standard SDN controllers configure the switches as part of the Mininet emulation environment, e.g., OpenFlow. Other advanced network protocols and forwarding setups can be implemented by the DOE.

Instance API. It provides an Infrastructure-as-a-Service (IaaS) endpoint allowing to manage MNs within instances in an adaptable way. The core interacts with this API to control the instance semantics. The default approach adds one specific Instance API to each type of instance. With this kind of abstraction, an instance can be managed in different ways, e.g., as a colony with different resource allocation or placement policies for each MN. The Instance API can be easily extended, and DOEs can implement their management interfaces on top.

Resource API. This API lets the DOE apply resource limits for each instance, such as constrained CPU and memory, and specify additional parameters such as a pricing model. OsmoticToolkit supports two kinds of resource models:

(1) Predefined Resource Model: It assigns predefined resources to each instance; in particular, there are 7 fixed models, e.g., m1.small (CPU: 1 and memory: 512 MB), m1.medium (CPU: 1 and memory: 1024 MB) and so on. If no resource model is specified, the m1.medium is

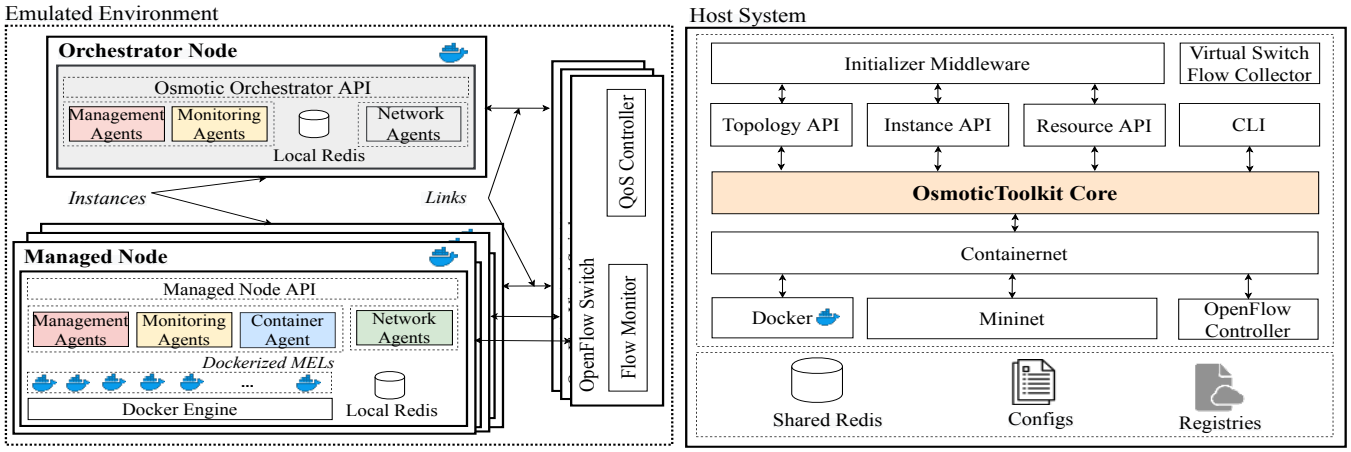


Figure 3: OsmoticToolkit architecture

applied for all instances by default, while the OI uses m2.medium.

(2) Customized Resource Model: Conversely, this model allows defining custom resource limitations for each kind of instance. For Cloud-based instances, some real resource limitations and pricing models can be configured through a Python-based script that scrapes current container pricing data from different service providers, e.g., AWS, Azure.

Initializer Middleware. It is implemented as a Python-based script and runs either on the DOE host system or within the build pipeline. Its primary purpose is to load the infrastructure topology definition template and start the emulation.

5.2 Emulated Environment

The ON is the central component serving RESTful API, as an MN with additional responsibilities. The ON extends MaestroNG [2] by implementing dynamic scheduling, monitoring, service offloading, and policy management. Specifically, MaestroNG is a simple and easily extensible orchestrator for Docker-based, multi-host environments that offer service-level and container-level controls that rely on declared service dependencies static placement.

As shown in Figure 3, the ON in the topmost level exposes an API through which the DOE interacts with the emulated ecosystem. In the lower levels, there are several agents, each one dealing with specific tasks. We use Celery, a simple, flexible, and reliable distributed task queue system, to roll up these agents. Celery is configured to use the local Redis database as a message broker. The main agents instantiated on every MN within the emulated environment are described below.

Monitoring Agents. One Monitoring Agent (MonA) is the Resource Monitoring Agent (RMA). It is responsible for periodically collecting utilization metrics from the MNs, allowing the VON at every level to be aware of the capabilities of the MNs present within the topology. RMA has been implemented through a non-intrusive Python library, e.g., psutil, and resides on each VN. Another MonA is Healthcheck Monitoring Agent (HMA). It is responsible for maintaining the infrastructure's topology by regularly monitoring the health of the MNs.

Management Agents. Management Agents (MAs) interface with deployment and control interaction within the infrastructure components. The Node Management Agent (NMA) is the most crucial component residing on the ON. It is responsible for handling every interaction with DOE through ON's APIs. It also deals with two primary operations, MEL scheduling and offloading. According to the pipeline's deployment requirements set by DOE, the NMA generates a resource provisioning plan via an optimization algorithm for describing the MELs contextualization across the MNs. Its implementation is extensible to allow the DOE to plug other resource provisioning strategies. OsmoticToolkit supports two provisioning approaches natively: a) static - inherited from MaestroNG and b) dynamic.

Using a static provisioning strategy, the DOE has to know a priori the network topology, e.g., the IP addresses of the MNs on which to schedule the MELs. The static approach does not use any optimization algorithm for resource provisioning; consequently, the nodes can result in over-/under-provisioned. In OsmoticToolkit, dynamic scheduling is treated as an assignment problem. Because of its simplicity and ability to find the optimal solution without requiring validation, we choose Kuhn's Hungarian algorithm [10] to solve the

assignment problems for generating the optimal scheduling plan for MELs.

Each assignment problem is associated with a cost matrix. The rows contain the workers or MNs we wish to assign, and the columns comprise jobs or MELs we want to assign to them. The cost function c_{ij} used to compute the cost matrix used is given in Equation 1 [16] and is defined as the weighted sum of the following five parameters: (i) number of containers running on each MN (n_c), (ii) percentage of memory used (n_{mem}), (iii) average CPU utilization (n_{cpu}), (iv) amount of CPU the MEL requires and finally (a_{cpu}) (v) amount of memory the MEL requires (a_{mem}). Where i varies from 1 to the number of VN N, j varies from 1 to the number of parameters 5. w_{ij} is a weight between 0 and 1, $component_i$ is an array containing the values of the parameters as mentioned above, and $f_j(t)$ represents that these parameters vary in time. The weights used are w_1 = 0.4, w_2 = 0.04, and w_3 = 0.01 (see [16] for further details).

$$C = \sum_{i=1}^{N} \sum_{j=1}^{5} w_{ij} \times component_{ij} \times f_j(t) \quad (1)$$

Kuhn's Hungarian algorithm treats the Optimal Assignment Problem (OAP) as a combinatorial problem to efficiently solve an $n \times n$ task assignment problem in $O(n^3)$ time. It a complete bipartite graph, G = {V, U, E}, where V and U are the sets of nodes in each partition of the graph, and E is the set of edges. The cost estimations become edge weights and each node and MEL becomes a vertex. Starting with an empty matching, Kuhn's Hungarian algorithm's basic strategy is to search for augmenting paths in the equality subgraph repeatedly.

If an augmenting path is found, the current set of matches is augmented by flipping the matched and unmatched edges along this path. Because there is one more unmatched than a matched edge, this flipping increases the cardinality of the matching by one, completing a single stage of the algorithm. If an augmenting path is not found, additional edges are added into the equality subgraph by making them admissible, and the search continues. n such stages of the algorithm are performed to determine n matches, at which point the algorithm terminates.

It should be noted that the orchestrator has to be also able to recognize situations when a MEL on the Edge must be offloaded on a Cloud MN or vice-versa. The Osmotic Orchestrator Agent (OOA) is a MA that periodically performs an orchestration by running the Hungarian algorithm to check whether the actual scheduling plan is optimal. If it is not, a new optimal scheduling plan is generated. The OOA contacts an NMA's API to update the new scheduling plan and perform the necessary MEL's offloading.

The offloading is implemented as live migration to limit the service downtime. This technique used is based on lazy or post-copy memory migration using Checkpoint-Restore in Userspace (CRIU). Another NMA is represented by the Node Agent (NA) as the main component of the VNs, exposing the necessary APIs to communicate with it, e.g., healthcheck endpoint, resource monitoring, migration, as previously explained. It is implemented as Python Flask service.

Container Agent. On every Cloud, Fog, Edge, and IoT MN, the Docker engine is installed. The Container Agent (CA) is represented by the persistent process that exposes the Docker API. It is used for communication with the Docker daemon, allowing it to control the status of containers. The ON contacts remotely the MNs' Docker daemon via this API every time it needs to deploy a new MEL.

Network Agents. The Network Agents (NA) collect flow statistics on each MN by running flow monitors for each network interface on virtual nodes and virtual switches. These data are stored in the Redis local instance for future analysis.

Database Instances. Our system involves two Redis DB instances. The local instances are used to store the configuration parameters and VN statistics, such as resource utilization. The shared Redis instance is for the versioning of the scheduling plan. It is also used to store the infrastructure topology description. There are also store the resource utilization metrics gathered from the RMA.

6 EXPERIMENTS AND EVALUATION

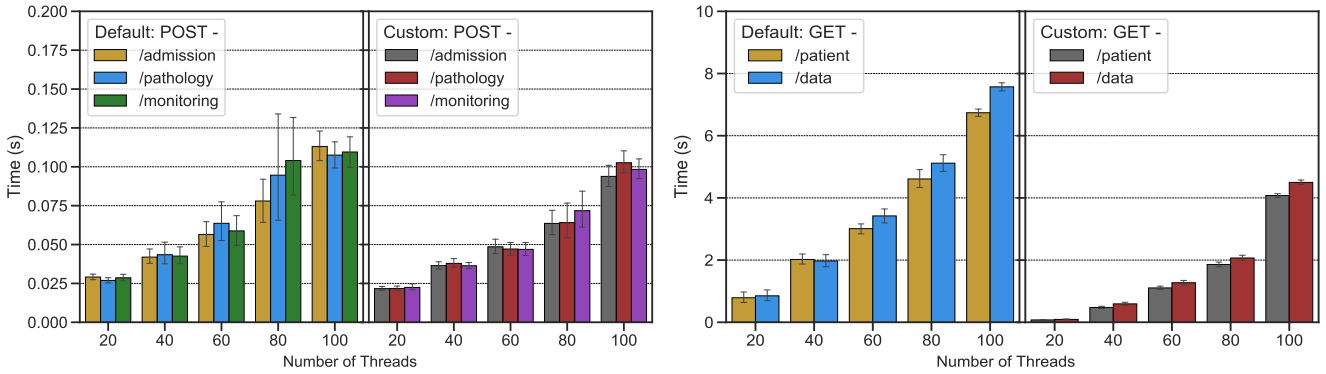
In this section, we present experiments and evaluations that we undertook to quantify the efficiency of OsmoticToolkit in modeling and simulating Osmotic Computing environments.

6.1 Methodology

The evaluation criteria leverage a set of metrics that can be used to evaluate the proposed emulator's effectiveness in terms of a) Responsiveness, b) Reactiveness and c) Agility.

Responsiveness. Assures that the system continues to have adequate response times even when the load rises. Generally, a system that strives to handle many requests with acceptable latency requires more computation resources. Hence, the system can be over-provisioned to keep system responsiveness. Such resources are expensive and a system should always optimize the use to be cost-effective. One of the responsiveness properties is SLA preservation. The guarantees provided by the SLA concern the fact that response times to user requests should never exceed a certain threshold.

Reactiveness. Indicates the reaction time of an environment composed of multiple individual applications blending into one unit while staying aware of each other to produce a workflow execution.



(a) POST **(b) GET**

Figure 4: Hospital Application MEL's Average Response Times (s)

Agility. Indicates the ease in applying changes in the environment. One helpful metric in this context is the system downtime when there is a dynamic reconfiguration. It represents the amount of time that the system or a portion of the system is not working correctly.

6.2 Application Use Case Scenario and Infrastructure Setup

To validate the effectiveness of the toolkit, we implemented the microservice-based rural hospital application illustrated in Figure 1. Its workflow involves each patient in several steps. The patient is initially examined, with preliminary clinical trials conducted to identify possible pathologies. The "MEL Pathology" manages these trials. It updates the patient's Electronic Medical Record (EMR) with the collected health data. Then, he passes through admission for recovery. "MEL admission" handles this by updating the EMR with the recovery information. Through continuous monitoring, health data are collected and stored within the EMR for local processing during the recovery phase. In the case of abnormal values, the EMR is submitted to offloaded processing.

All MELs composing the e-health application are implemented with Java 12 and SpringBoot 5. To discover MELs, we use the REST-based Eureka server. The patient's actions are simulated using JMeter and with periodic requests. We configure a test plan with 100 threads (equivalent to 100 patients), a duration of 8 minutes (480 s), and ramping-up over 2 minutes with 3 minutes of hold time. Hold time confirms that the system handles the load and its performance stays stable and does not deteriorate. There is a tunable interarrival time between one API call and the next one. It is managed by setting up a Gaussian random timer with a deviation of 500 ms, and a constant delay offset of 1000 ms.

The infrastructure topology used to run the MELs consists of 1 VCI, 1 VFI (connected to VCI), 1 VEI (connected to VFI), and a VIoTI (connected to VEI). We consider two resource models for the experimental investigation: (i) default: all the VIs have minimal hardware resources (e.g., 1 CPU and 1024 GB of memory), (ii) custom: different user-defined hardware resources characterize each VI, such as cloud (CPU: 1, memory: 4096 MB), fog (CPU: 1, memory: 2048 MB), edge (CPU: 1.4, memory: 1024 MB) and IoT (CPU: 1, memory: 512 MB). The experiments have been conducted on an OpenStack cluster instance by CloudLab[1]. The instance has 16 GB RAM, 8 VCPU and 240 GB of storage and runs Ubuntu 18.04 LTS.

6.3 Results and Findings

Network Parameters Selection. For the emulated infrastructure, we modeled the described scenario with suitable latency and packet loss values for the links to reflect as much as possible a real site. To do so, we conducted a series of experiments to measure the latency and packet loss in three different setups: (i) Edge to Cloud and (ii) Fog to Edge, and (iii) IoT to Fog by performing 3000 ping requests.

As Cloud node, we used the OpenStack instance previously listed. As Edge node, we used a Raspberry Pi 3 with 4 cores @ 1.4 GHz and 1 GB RAM with Raspberry Pi OS Lite connected to a router via Ethernet and, as an IoT device, an iPhone 7+ with iOS 14.1 connected via WiFi with the same router. As Fog node, we used a MacBook Pro 3.3 GHz Dual-Core Intel Core i7 with 16 GB RAM and MacOS Catalina 10.15.7 connected via WiFi with the same router. The router, Edge, Fog, and IoT devices are physically located in the same room.

First, we ran a ping on the Cloud to measure the Round Trip Time (RTT) from the Edge node. We found the average RTT is 67.56 ms and 4.0% of packet loss. Then, we ran a ping on the Edge to measure the RTT from the Fog. We found the average RTT is 42.189 ms and 0.2% of packet loss. Finally, we

[1]CloudLab: info.cloudlab.zhaw.ch

ran a ping on the IoT to measure the RTT from the Edge. We found an average RTT of 7.57 ms and 0.0% of packet loss. We used the measured average RTTs from the real experiment and provided that as an input parameter for the emulated links in our model (see Figure 1).

Responsiveness. To ensure the SLA preservation, it is helpful to see the MEL's average response times when the load is low and use them as a reference for checking the degraded performance when the load increases. As shown in Figure 4a, we note the average response times increase with the number of patients. The three endpoints have different requests over time; JMeter periodically performs a set of requests to different API endpoints over time. Figure 4a shows that the response time per each API endpoint. In particular, the results obtained by applying a custom resource model revels lower average response times and show a more stable trend when the number of patients increases. The average response times obtained with both configurations are acceptable (around 0.125 s using a default resource model and 0.1 s using a custom resource model when the maximum number of patients is reached), preserving the SLA. The trend is even more evident in Figure 4b.

In conclusion, the slowdowns are not particularly severe, and the responsiveness results when both resource models are applied show that the application remained responsive throughout all executions.

Reactiveness. To assess the proposed system's reactiveness, we evaluated the workflow from the infrastructure bootstrap/tear down and pipeline deploy/undeploy sides. We collected these results using both resource models default and custom and dynamic and static resource provisioning approaches. According to Figure 5, we notice the infrastructure bootstrap with a custom resource model requires, on average, 22.5 s, while the default one takes on average 24 s. A similar trend is obtained when the infrastructure is torn down. This is because the nodes initialized with the custom resource model have a higher amount of resources assigned and require more time to instantiate them. The same is also applied when the infrastructure is torn down and all allocated resources are released.

For the pipeline deployment, we used both static and dynamic provisioning approaches. Figure 5 shows the obtained response times when a static resource provisioning approach has been used. In the case of a dynamic approach, the response times must be summed up the time the Kuhn's Hungarian algorithm takes to provide the scheduling plan. That is, on average, 4.29 ms. As we can see, Kuhn's Hungarian algorithm's execution time is almost negligible and does not impact response times. We notice the opposite trend with respect to the bootstrap/tear down response times in terms of deploy and undeploy response times. We notice that deploy/undeploy operations performed on top of the nodes initialized with the custom resource model require less time respect when the default resource model is used.

This is because the nodes initialized with the custom resource model are more powerful in terms of hardware resources than those instantiated with the default resource model. In this way, all the processes run more fluently are quicker to initialize components and respond. Similar behavior is obtained when the status of the pipeline is checked by calling the corresponding API. The response times collected for deploying the pipeline do not consider the time necessary to download the MEL's images. To download all the images, there are required almost 50 s on average more. Therefore, the overall response times are acceptable and in line with what we were expecting.

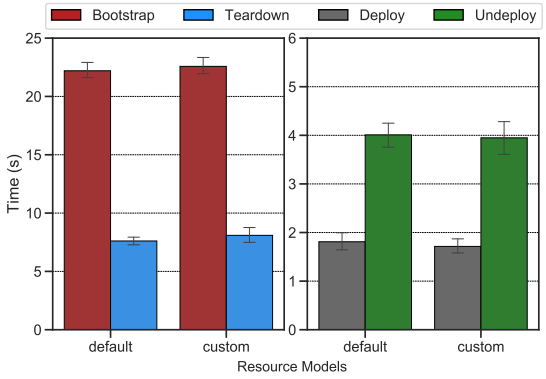


Figure 5: Infrastructure and pipeline instantiation

Agility. Evaluation of the system's agility is done by dynamically performing a MEL offloading from a Cloud node to an Edge node. MEL's offloading allows minimizing cost and energy consumption of services to end-users by improving QoE leveraging the awareness of their location, network, mobility, and context information. The main objective is to show that the offloading of a MEL causes no system downtime minimizing as much as possible the application's downtime in case of reconfiguration. We performed the MEL's offloading when both resource models were used to initialized the nodes. We therefore used for this evaluation two MELs of different sizes to understand how the image size impacts the overall times. In particular, one of 75.3 MB and another of 182.41 MB. As explained, offloading is implemented as live migration consisting mainly of two phases: (i) checkpoint and (ii) restore. We performed 30 subsequent executions and gathered the average response, checkpoint/restore and downtime times. The response time measures the elapsed time between the first POST call on the /offload endpoint and the MEL restore phase's start on the destination node. Figure 6 show

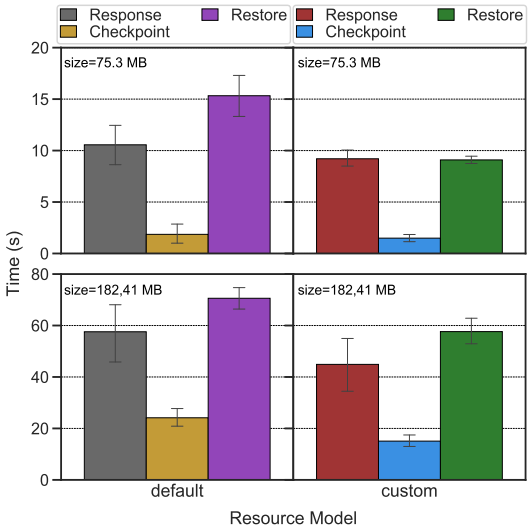


Figure 6: MEL offloading

that the average response time is greater when a default resource model is applied. The same behavior is obtained for the checkpoint/restore times. The mean MEL downtime during the offloading is 0.5 s when the default model is applied and 0.35 ms with the custom one. This fact highlights that the service has been offloaded with minimum downtime. This is because, with the custom model, the nodes are more powerful and perform faster operations. However, for an image of 75.3 MB, the timings are higher than with an image size of 182.41 MB. The image size impacts heavily on the checkpoint/restore performances. Thus, the offloading operations can be performed dynamically and asynchronously while the system is running, introducing zero downtime for the system and improving the overall agility.

7 CONCLUSIONS AND FUTURE WORK

With this work, we contributed and evaluated OsmoticToolkit, emulating resources and network connections for distributed applications deployment to IoT devices and Fog, Edge and Cloud resources. As the emulation approximates real deployments and eases their planning, it exceeds simulation approaches. Moreover, it is the first emulator to combine four key characteristics: hybrid topologies, dynamicity with service offloading, resource provisioning/orchestration, and realistic container execution.

Based on the achieved toolkit, we intend to conduct broader studies on integrating commercial cloud providers and minimizing the runtime behavior differences between real deployments and emulation by supporting further cloud-native MELs and SDMems.

ACKNOWLEDGMENTS

UniME grant for Ph.D. positions in Distributed Systems: Osmotic Computing and Big Data Analytics in eHealth, signed with the Consiglio Nazionale delle Ricerche (CNR), Istituto per la Ricerca and Innovazione Biomedica (IRIB), Scientific Responsible Ing. Pioggia Giovanni. We thank Piyush Harsh for the preparation of the hospital microservices.

REFERENCES

[1] [n.d.]. IoT Business News - The IoT in 2030: 24 billion connected things generating $1.5 trillion. https://iotbusinessnews.com/2020/05/20/03177-the-iot-in-2030-24-billion-connected-things-generating-1-5-trillion/ Last accessed 27 July 2020.

[2] [n.d.]. MaestroNG. http://maestro-ng.readthedocs.io Last accessed 24 July 2020.

[3] A. Buzachis, A. Galletta, A. Celesti, L. Carnevale, and M. Villari. 2019. Towards Osmotic Computing: a Blue-Green Strategy for the Fast Re-Deployment of Microservices. In *2019 IEEE Symposium on Computers and Communications (ISCC)*. 1–6.

[4] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. 2011. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Exp.* 41, 1 (2011), 23–50. https://doi.org/10.1002/spe.995 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.995

[5] A. Coutinho, F. Greve, C. Prazeres, and J. Cardoso. 2018. Fogbed: A Rapid-Prototyping Emulation Environment for Fog Computing. In *2018 IEEE International Conference on Communications (ICC)*. 1–7. https://doi.org/10.1109/ICC.2018.8423003

[6] Damián Fernández-Cerero, Alejandro Fernández-Montes, Agnieszka Jakóbik, Joanna Kołodziej, and Miguel Toro. 2018. SCORE: Simulator for cloud optimization of resources and energy consumption. *Simulation Modelling Practice and Theory* 82 (2018), 160 – 173. https://doi.org/10.1016/j.simpat.2018.01.004

[7] Damián Fernández-Cerero, Alejandro Fernández-Montes, F. Javier Ortega, Agnieszka Jakóbik, and Adrian Widlak. 2020. Sphere: Simulator of edge infrastructures for the optimization of performance and resources energy consumption. *Simulation Modelling Practice and Theory* 101 (2020), 101966. https://doi.org/10.1016/j.simpat.2019.101966 Modeling and Simulation of Fog Computing.

[8] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K. Ghosh, and Rajkumar Buyya. 2017. iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments. *Software: Practice and Experience* 47, 9 (2017), 1275–1296. https://doi.org/10.1002/spe.2509 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2509

[9] Jonathan Hasenburg, Martin Grambow, and Elias et al. Grünewald. 2019. MockFog: Emulating Fog Computing Infrastructure in the Cloud. https://doi.org/10.1109/ICFC.2019.00026

[10] H. W. Kuhn. 1955. The Hungarian method for the assignment problem. *Naval Research Logistics* 2 (1955), 83–97.

[11] I. Lera, C. Guerrero, and C. Juiz. 2019. YAFS: A Simulator for IoT Scenarios in Fog Computing. *IEEE Access* 7 (2019), 91745–91758.

[12] L. Liu, Z. Chang, X. Guo, S. Mao, and T. Ristaniemi. 2018. Multiobjective Optimization for Computation Offloading in Fog Computing. *IEEE Internet of Things Journal* 5, 1 (Feb 2018), 283–294. https://doi.org/10.1109/JIOT.2017.2780236

[13] Márcio Lopes and Wilson et al. Higashino. 2017. MyiFogSim: A Simulator for Virtual Machine Migration in Fog Computing. 47–52.

https://doi.org/10.1145/3147234.3148101

[14] R. Mayer, L. Graser, H. Gupta, E. Saurez, and U. Ramachandran. 2017. EmuFog: Extensible and scalable emulation of large-scale fog computing infrastructures. In *2017 IEEE Fog World Congr.* 1–6. https://doi.org/10.1109/FWC.2017.8368525

[15] M. Peuster, H. Karl, and S. van Rossem. 2016. MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 148–153. https://doi.org/10.1109/NFV-SDN.2016.7919490

[16] G. Rakshith, M. V. Rahul, G. S. Sanjay, B. V. Natesha, and G. Ram Mohana Reddy. 2018. Resource Provisioning Framework for IoT Applications in Fog Computing Environment. In *2018 IEEE Intl. Conf. Advanced Networks and Telecommunications Systems (ANTS)*. 1–6.

[17] Olena Skarlat, Matteo Nardelli, Stefan Schulte, Michael Borkowski, and Philipp Leitner. 2017. Optimized IoT service placement in the fog. *Service Oriented Computing and Applications* 11 (2017), 427–443.

[18] Cagatay Sonmez, Atay Ozgovde, and Cem Ersoy. 2018. EdgeCloudSim: An environment for performance evaluation of edge computing systems. *Trans. Emerging Telecommunications Tech.* 29, 11 (2018), e3493. https://doi.org/10.1002/ett.3493 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/ett.3493

[19] Philip Wette, M Dräxler, Arne Schwabe, F Wallaschek, M Zahraee, and H Karl. 2014. MaxiNet: Distributed emulation of software-defined networks. *2014 IFIP Networking Conference, IFIP Networking 2014*, 1–9. https://doi.org/10.1109/IFIPNetworking.2014.6857078

[20] Y. Xiao and M. Krunz. 2017. QoE and power efficiency tradeoff for fog computing networks with fog node cooperation. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 1–9.

[21] Xuezhi Zeng, Saurabh Kumar Garg, Peter Strazdins, Prem Prakash Jayaraman, Dimitrios Georgakopoulos, and Rajiv Ranjan. 2017. IOTSim: A simulator for analysing IoT applications. *Journal of Systems Architecture* 72 (2017), 93 – 107. https://doi.org/10.1016/j.sysarc.2016.06.008 Design Automation for Embedded Ubiquitous Computing Systems.