Microservices Management on Cloud/Edge Environments

André Carrusca, Maria Cecília Gomes(✉), and João Leitão

NOVA LINCS & DI/FCT/UNL, Costa da Caparica, Portugal
a.carrusca@campus.fct.unl.pt, {mcg,jcleitao}@fct.unl.pt

Abstract. The microservices architecture is a promising approach for application development, deployment, and evolution, both on cloud and emerging fog/edge platforms. Microservices' single functionality, small size, and independent development/deployment support faster and cheaper scaling of pressing functionalities on cloud systems. They support applications' evolution via service reuse and smooth service modification/inclusion. Individual or sets of inter-related services may also be dynamically deployed onto resource-restricted nodes closer to end devices and data sources, which are typical of fog/edge computational platforms. The resulting system is very complex and impossible to be adequately managed manually. This work presents an automatic solution for microservices' deployment/replication in the fog/edge, adapting the system according to the runtime evaluation of client accesses and resource usage. The evaluation validates the adaptability and performance gains.

Keywords: Microservices architecture · Cloud and fog/edge computing · Self-adaptable applications

1 Introduction

The microservices architecture [7,10] presents several advantages for application development, deployment and evolution, in the accelerating omnipresent and omniscient digital world. Traditional monolithic architectures represent a single large application composed of tightly interdependent and non reusable components. In contrast, microservices applications combine small, single functionality, loosely coupled services, to implement more complex functionalities. Each microservice accesses its own private database, displays a well defined API, and may communicate with others directly (e.g. via RPC/REST protocols) or indirectly (e.g. via messaging/event systems). This allows their independent development with diverse technologies and their individual scaling, simplifying applications' reliability and continuous delivery responding to new requirements [1].

The constant need for evolving systems is intensified by the surge of both mobile and Internet of Things (IoT)/terminal devices, e.g. in the domain of

© Springer Nature Switzerland AG 2020
S. Yangui et al. (Eds.): ICSOC 2019 Workshops, LNCS 12019, pp. 95–108, 2020.
https://doi.org/10.1007/978-3-030-45989-5_8

Smart Cities/Health [27]. Traditionally, these types of applications are supported by services running on cloud platforms [20]. Yet such high number of client devices produce a large number of requests towards the *backend* services and generate huge amounts of data, requiring novel solutions adaptable to systems' evolution. Many of these services correspond to bandwidth intensive and increasingly popular applications like video-on-demand, streaming, or real time TV [3]. Also, the predicted huge number of IoT devices [9] will collect data needing to be mined and analysed (e.g. sensors dispersed over wide areas) and often with time restrictions (e.g. drone applications). It is so necessary to avoid latency degradation and guarantee the applications' QoS.

Hybrid Cloud/Edge Computing. Emerging solutions capitalise on the microservices architecture both as *cloud-native* applications [17] and applications distributed on novel hybrid cloud/edge platforms [32]. Applications rely on services in *cloud computing* [24] providing ubiquitous and on-demand access to shared resources perceived as unlimited (e.g. computational, storage, and network resources). Novel solutions capitalise on lighter, faster, and cheaper scaling of microservices in the cloud to support applications' variable geographical accesses and incremental evolution. This is the case of interactive applications with constant updates and performance/availability constraints [2,21]. Novel solutions also may capitalise on cloud technologies' expansion to the periphery of the network. Namely, edge/fog computing [18,34,37] represent the usage of diverse heterogeneous computational resources on the continuum from the cloud datacenters to end devices. The resources range from routers, base stations, to micro-datacenters/cloudlets. The result is a computational platform of highly heterogeneous nodes, geographically dispersed at large numbers, closer to end users and data sources, and that typically present reduce computational capabilities in comparison to cloud datacenters [3,6,8]. The microservices architecture is also adequate to exploit such capability restricted nodes since small services may be migrated/replicated in a faster way, according to user/application needs. This allows reducing the latency on accessing services, lowering the amount of data on transit in the communication infra-structure (e.g. by filtering/pre-processing data closer to data sources) and exploring an adequate usage of the computational infrastructure.

Problem and Goals. Computing in heterogeneous platforms composed by cloud nodes and a large number of highly heterogeneous edge resources presents several challenges for application development and management [19,35]. Also, microservices applications are composed of a large number of services (and their replicas), each one with diverse functionalities, possibly a database, and diverse hardware/software needs. Services may have different levels of interaction (e.g. frequent/sporadic invocation of other services depending on the workload), which aggravates the overall management and debugging [11]. The services may have to be upscale and their databases replicated to improve the applications' performance and energy efficiency. In this setting, examples of challenges are adequate and flexible resource management solutions with service location/deployment depending on the origin/volume of user accesses and on the computing

nodes' total/current resources and their cost; eventual dynamic migration/replication of microservices' databases following services' replicas; service coordination in a distributed context; the guarantee of security and privacy issues; etc.

Due to such management complexity, our long-term goal is to build an autonomic solution [14,18,28] for these systems. We envision a self-management solution composed of three dimensions for decoupled functionality/management:

(*a*) a service management component, discussed here, to deploy and scale individual and inter-related services (e.g. necessary for a particular functionality);
(*b*) a database management component responsible for the dynamic replication of microservices' databases whose replica instances may be widely dispersed;
(*c*) a monitoring component responsible for observing services (e.g. load/location of accesses) and infrastructure nodes (e.g. current consumed resources) and timely providing the necessary information to the other two components.

Each dimension is self-adaptable on fulfilling its objective and cooperates with the other two towards establishing a global self-managing solution. The database component guarantees the consistency model of (the existing/newer replicas of) a particular microservice's database. The monitoring component flexibly collects and delivers a variable set of metricas with diverse time intervals and without incurring unaffordable overheads over the infrastructure's nodes and network.

Whereas we have already presented solutions advancing the database and monitoring dimensions, this work presents an automatic solution for microservice migration/replication contributing to a self-adaptable service management and, in the future, to an autonomic solution able to learn from applications' evolution and previous decisions and to predict adaptation requirements.

Document Organisation. The following section describes the proposed solution and it is followed by Sect. 3 that discusses the implementation and evaluation results. Section 4 presents the related work and Sect. 5 concludes the paper.

2 Proposed Solution

We present an automatic management approach for node allocation and service migration/replication within the emergent cloud/edge platforms. The objective is to improve the application's performance and the clients' perceived latency, and to adequately operate the infra-structure's resources. This in spite of the system's inherent complexity and dynamics both in terms of the infra-structure volatility (with failing/new nodes) and the dynamic application requirements. Namely, many cloud applications experience a high variability of accesses, and other applications rely on the large volumes of data generated from end devices at diverse locations, at variable times.

To respond to such variability, we propose a self-adaptable mechanism with a decision process based on an modifiable set of user defined constraints and

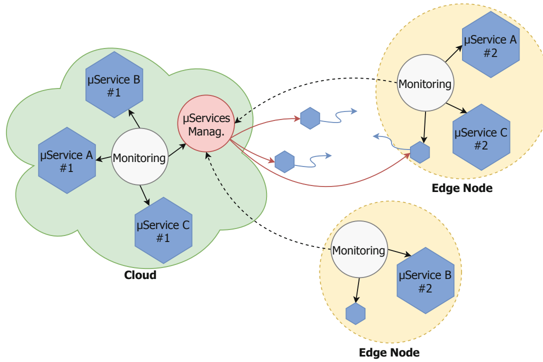


Fig. 1. A simplified view of the solution.

rules. The system's state is continuously monitored within an evaluation/decision feed-back loop, typical of self-adaptable systems [30] and autonomic systems [28]. At each iteration it evaluates (*a*) which services should be migrated or replicated, when, and to where, or be otherwise eliminated; and (*b*) which nodes need to be dynamically created/activated or eliminated, from a computational platform providing virtual nodes. The decisions in the loop are tuned to improve the applications' performance while avoiding possible system's destabilisations caused by too frequent updates. This means that some evaluations have to be confirmed in a few consecutive iterations before the corresponding decisions are applied. Figure 1 presents a simplified version of the solution's architecture with microservices' replicas deployed at the cloud and two edge nodes. A single (centralised) microservices management component, *uServices Manag.*, is located at the cloud and communicates with the monitoring components, one at each edge node. These are responsible to collect the relevant services' and nodes' metrics. In the continuous feedback loop, the *uServices Manag.* decides upon service scalability level and location, and the number and location of computational nodes.

2.1 The Architecture Components and Their Operation

The main architecture components in Fig. 1 include *microservices* (*uServices*) and their replicas, *computational nodes* in the cloud/edge, and the *service management component* and associated components e.g. a *monitoring component*. The uServices (typed as *frontend/backend*) and their interactions comprise the user application to be optimised. Each uService instance is placed in a container for its faster/lighter deployment on the nodes [26,29]. Each node is a *virtual machine* (VM), a basic resource management unit in cloud providers, where one or more containers may be deployed to [20]. The monitoring component collects the relevant service/node metrics as required by the service manager. The latter needs also the pre-configuration of services' and nodes' execution requirements, and the specification of the constraints/decision rules that guide the decision

process. The manager allows pre-scheduled events and relies on the *service registry* and *load balancing* components/patterns [29]. All this is described next.

Specification of Execution Requirements for Microservices and Nodes. Types of information specification for services:

- First execution: service type (frontend/backend, database); service image repository; service access ports; start command (e.g. parameter's initialization values); services' dependencies (e.g. service communication).
- Operational: running service's lowest/highest number of replicas; parameters/metrics limits for a replica's correct operation (e.g. minimum RAM).
- Monitoring: service latency; service access (number and source of accesses); bandwidth; service's used resources (CPU, RAM, ...).

Information requirements for cloud/edge nodes:

- Operating data requirements: parameters/metrics constraints (e.g. RAM); location information for edge nodes (from continent to city);
- Monitoring data:used resources (CPU, RAM, ...), and bandwidth.

Decision Process. To perform decisions, the microservices manager uses a rule mechanism with *Event Condition Action (ECA) rules* [13,22]. Each rule encodes the conditions and the consequent actions to be performed, accepts multiple values (parameters) representing the current state of the system, and may have a priority level. The rules express the set of constraints on services and nodes, and the modification operations. A rules engine (see Sect. 3) performs their evaluation in the *analysis phase* of the adaptation feedback loop based on the current system state captured by the *monitoring phase* (Fig. 2b and Sect. 2.2).

Rules Related with Services and Their Replicas: The parameters may include %CPU, %RAM, transferred bytes, etc., and the actions are *replicate*, *migrate*, *stop*, *nothing*. Rules capture situations such as (*i*) if the argument values exceed the ones expressed in the rules, a service needs to be replicated or migrated; (*ii*) if the arguments are less than the defined minimum, a replica is marked to be removed; (*iii*) nothing is done, otherwise. The priority level of the fired rules define the final decision. For instance, the service replication/migration rules may privilege a (closer) edge node than the cloud for placing a replica. Nonetheless, the new replica is always located in the cloud in case no edge node is available.

Rules Associated with Nodes: The parameters are %CPU, %RAM, and the actions are *add*, *stop*, *nothing*. The rules encode (*i*) a node's creation, if the containers' execution resources are scarce; (*ii*) a node's removal, if its resources are underutilised; (*iii*) nothing is done, otherwise. A node's placement onto the edge vs cloud may also have a priority. In case an edge node's creation/activation is not possible, the node is allocated from the cloud's resources seen as unlimited.

To allow a more precise tuning of the adaptation actions in response to the current system state, both service and edge rules allow diverse parameter configurations: *precise/effective parameter value*, uses exactly the read value of a particular metric; *average value*, the evaluation process considers the average value of a set of particular metrics; *mean deviation percentage*, considers the deviation percentage of the current value in comparison to a given metric's average; *last value deviation percentage*, considers the current value's deviation percentage in relation to a specific metric's last read value. *Event Scheduling:* The definition of *pre-scheduled events* aims to improve the overall system performance by allocating a set of resources at some particular places and times. E.g., increasing the minimum number of replicas needed for a popular social network application expected to have high access volumes, at the time and place of a particular football game or pop music concert. Similarly, a pre-scheduled reduction of no longer needed resources is also possible.

Service Manager's Necessary Components and Functionalities. To dynamically create/destroy nodes and migrate/replicate microservices, the service manager relies on a few external components to support its operation:

- *Container manager*, to detect nodes' and services' failures and support the creation of services and nodes whereto services may be deployed (see Sect. 3).
- *Monitoring component*, to collect fresh metrics from services/nodes defining the system state, allowing its evaluation and necessary adjustments (see Sect. 3).
- *Service registry*, to record new services and replicas, including their location. When a service is created, replicated/moved, it has to be reachable/communicate with other services. This demands a more general communication process than a point-to-point one, which includes a *Register and discover services* component to bridge individual microservices' interaction.
- *Load balancer*, to adequately distribute service accesses to existing replicas deployed at diverse locations, improving the system's performance/efficiency.

Service Communication: The components *Service Registry* and *Register and discover services*, shown in Fig. 2a, support communication decoupling and some level of inter-service load balancing. The communication from a *uService A* to *uService B* is based on the target's type/name (i.e. B) and not on a fixed communication endpoint. This is fundamental to carry out service migration or replica selection, e.g. to access a *uService B*'s replica located on the same node as A.

The *Service Registry* extends the *service registry pattern* [29] to support migration and replication. It allows service registration and discovery by service name/type via the organisation of running services' endpoints according to service type. It also stores the location of services and their replicas, and if they are active. Whenever a microservice is migrated or replicated, the registry has to be notified to update the service's information. The registry is deployed in its own container and can be replicated to enable faster queries.

The *Register and discover services* component is essential to microservices' migration/replication and supports basic load balancing towards backend

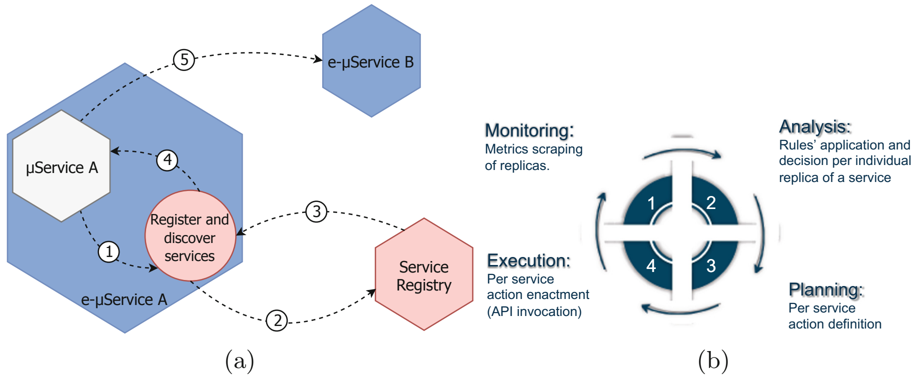


Fig. 2. (a) Service communication via the *Register and discover services* subcomponent and the *Service Registry* component. (b) Service reconfiguration; adapted from [14].

services. It has some contact points to the function of *sidecars* in the recent *service mesh pattern* [29], since it is coupled to a particular microservice to bridge its accesses. Namely, to support the use of the Service Registry features described above, each microservice in the adaptability system has to exist within an *extended microservice* wrapper container that also includes a *Register and discover services* component. For instance, in Fig. 2a, the *e-uService A* contains the *uService A* and a *Register and discover services* component. This latter component has the following functionalities: (a) registers its microservice creation/deletion in the service registry and updates the registry periodically to inform that the service is still active; (b) queries the endpoints of other services taking into account the location of its service, and in case of several equal possibilities (e.g. same edge node) chooses one endpoint at random, e.g. service A may access a local replica of B. The numbers in Fig. 2a illustrates the process when *uService A* wants to communicate with a service named B: to obtain an endpoint for B, A contacts the *Register and discover services* (1); the latter requests the (all possible) endpoint(s) from the *Service Registry* (2, 3), selects the best endpoint and sends it microservice A (4) that uses it to communicate with B (5).

Load Balancing Service Requests: The *Load Balancer* component distributes client requests towards a microservice's replicas to adjust their load. Clients access a load balancer preferably in their own region and only the calls to a *frontend* microservice are balanced. Yet the load balancer can be replicated to the same *regions* as the frontend replicas to level the load at each location. All load balancers' replicas have access to all service replicas regardless their location, allowing them to redirect accesses when a region has no replicas or the local ones are overloaded. Figure 3a represents an extended microservice *e-uS A* with a single replica and a single load balancer in the cloud. Figure 3b shows a scenario with the *e-uS A* and the load balancer replicated in two regions. The *Load Balancer #1* serves the clients in the USA and gives priority to the *e-uS A* replicas #1 and #2 in the same region. However, it redirects the requests

to the replica #3 located at an edge node, in case the first two microservices become overloaded. The replica selection algorithm uses (*i*) the *Least Connections* method and (*ii*) a weight assigned to each replica to privilege replicas in the same region/location as the load balancer. The choice was tuned based on the number of each replica's active connections and its weight, to access less loaded replicas but also to reduce clients' communication with remote replicas. E.g. the closest replica may still be chosen if its connections' number is just slightly higher than a farther one.

2.2 Adaptation Process and Migration/Replication Scenarios

To respond to services' and nodes' overload and comply to the applications' QoS requirements, the adaptation process uses migration and horizontal scalability of services/nodes for the system's dynamic reconfiguration, instead of vertical scalability (increase a VM's capacity). The creation of multiple service/node replicas allows a simpler and faster management process, e.g. replicating a pre-existing service with the same resources, and, above all, supports large-scale scalability via replica deployment onto geographically dispersed edges devices.

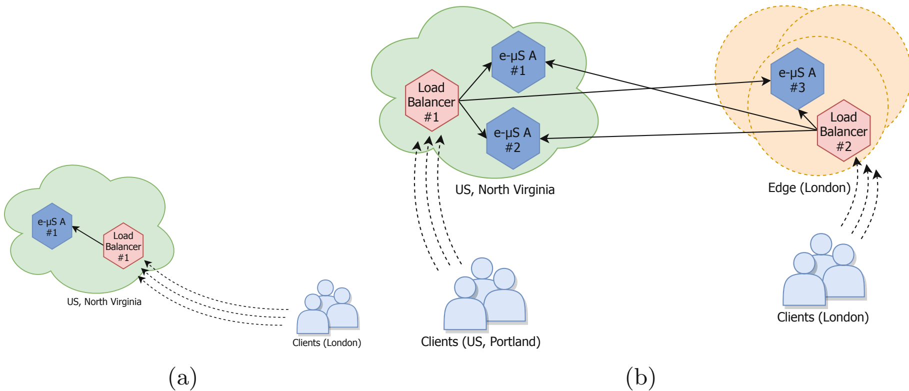


Fig. 3. Function of the load balancer: (a) cloud only, (b) replicas in cloud/edge.

The adaptation process consists of a four stage feedback loop inspired on [14,30], as shown in Fig. 2b for service management (similar for nodes): (*i*) *service replicas' monitoring*, to collect relevant metrics (e.g. data transfer, CPU usage); (*ii*) *analysis per service replica*, to evaluate the action to apply to each particular service (e.g. migrate or replicate/eliminate if overloaded/underused); (*iii*) *planning for all services*, considers the overall system state to decide the action for each service; (*iv*) *plan's execution*. The system follows a set of rules like previously discussed and that may be configured by the application administrator. To avoid a constant system reconfiguration causing its instability, a problem well known in self-adaptability, there is a time gap between the first indication

for reconfiguration and its effective execution. For instance, the indication to remove a edge node has to be confirmed in three consecutive loop cycles before the node is effectively deactivated. This allows that in case of sudden changes in nearby client accesses meanwhile, the decision may be to keep the node active.

The migration of microservices to respond to local latency variations diverges from the usual meaning within the cloud domain (e.g. migration of VMs). The *migration process* is based on replication to keep the service available but also in a way to promote further flexibility depending on the perceived ongoing changes in the system state. Its steps may be: 1. create a local replica of an overloaded service; 2. eventually move the replica to e.g. a edge node with higher service accesses, in the next loop cycles; 3. eliminate the original service if underused in the next cycles, e.g. all client requests are now better served by the replica at the edge. Although taking a few (parametrisable) iteration cycles of the adaptation loop, the service eventually migrates to a new location.

Discussion on the Adaptation Solution and Scenarios. Microservices applications executing on Cloud/edge systems have to deal with [11] inter-service communications and dependencies that may cause network overheads and a cascade of QoS violations; microservices' diversity, with different bottlenecks that may change as the load increases; or cloud applications' latency variability. When considering live migration/replication of microservices, an adequate decision on which ones to migrate/replicate, when, and where to, becomes even more pressing to guarantee a good application QoS and efficient resource usage. Our solution, via its self-adaptable management with a gradual replication/reduction of nodes/services according to the ongoing system modification expressed via diverse metrics/conditions in rules, aims to address the concerns above. Our system detects when services/nodes exceed some resources' threshold and need to be replicated

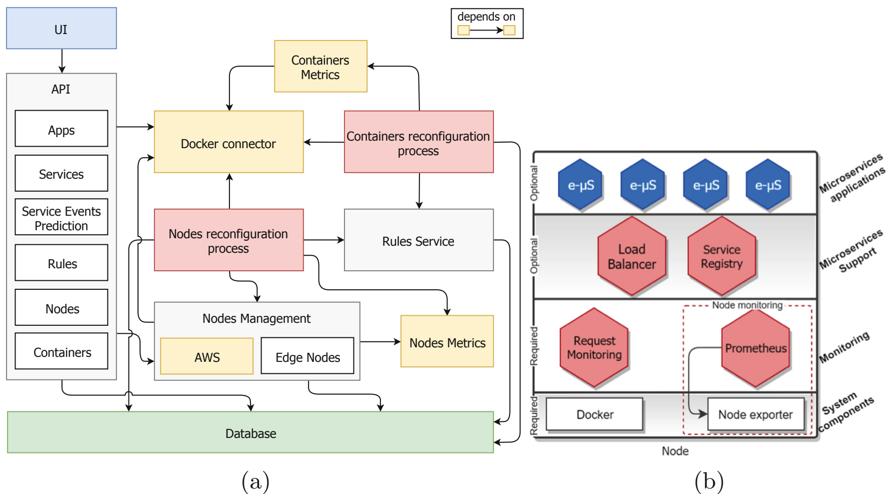


Fig. 4. (a) Detailed view of the uS management component. (b) Node architecture.

and decides where to based on the collected metrics. A service may so move to an edge node closer to a high number of clients to reduce their perceived access time and the communication traffic. In case the service performs some additional local data filtering, the data on transit volume may be reduced even further, and if it has predictable peak loads its resources can be provisioned a priori. The decision process also has to assess the type of metrics underlying the decisions, if the service has a database which impacts transfer costs, or if there are dependencies between services. For instance, the evaluation in section Sect. 3 focus on a frontend service communicating frequently with a backend catalog service upon clients' accesses. When the clients view the full catalog the whole database is transferred. The migration/replication of both services to a nearby edge node reduces hence the communication traffic.

3 Prototype and Evaluation

The service management's modules are detailed in Fig. 4a[1]: *User Interface (UI)*, APIs for e.g. container/node/service creation, rules definition, etc.; *Docker container manager*, e.g. to start/stop containers and resource usage data; *Rules module*, an engine for ECA rules' management e.g. creation/deletion and analyses i.e. which rules to trigger based on current nodes' and services' metrics; the application administrator uses the UI to manage ECA rules, e.g. defining a rule's triggering conditions and affected entities (services or nodes); *Node management*, integrates the management for cloud (in AWS) and edge nodes, e.g creates/suspends VMs, and uses the *node metrics module* (*Prometheus*) to get each node's resource usage; *reconfiguration process*, it is subdivided into containers and nodes, and periodically decides the actions to perform on services/nodes.

Figure 4b shows the necessary management components to support service deployment in a node. A node is a VM that is created in the context of a cloud platform or fog/edge device, and a set of nodes forms a cluster for service deployment. *Docker* manages the node's containers, e.g. identifies a container's resource usage (CPU, RAM) and failures. The *Node exporter* (from Prometheus) collects the node's resource usage (CPU/RAM), e.g. to know if is possible to deploy another service. The components that support microservices, i.e. the load balancers and the service registry, are only present in some nodes, which is decided by the system. Finally, the uS are encapsulated into extended uS (Sect. 2.1).

Evaluation. To allocate VMs from data centers in different regions we used the EC2 service from AWS. The evaluation setting for service deployment on the cloud uses nodes in North Virginia, US, with clients in London (Fig. 3a). The setting for cloud/edge execution uses the same cloud region with users in Portland, and edge nodes and clients in London, UK (Fig. 3b). The load tests use

[1] Sw used: uS management, Java/Spring Boot; UI, JavaScript library React; container manager, https://www.docker.com; rules engine, https://www.drools.org/; monitoring, https://prometheus.io; AWS cloud, https://aws.amazon.com; Load tests, https://docs.k6.io/docs & https://loadimpact.com/insights/; Sock shop https://microservices-demo.github.io/.

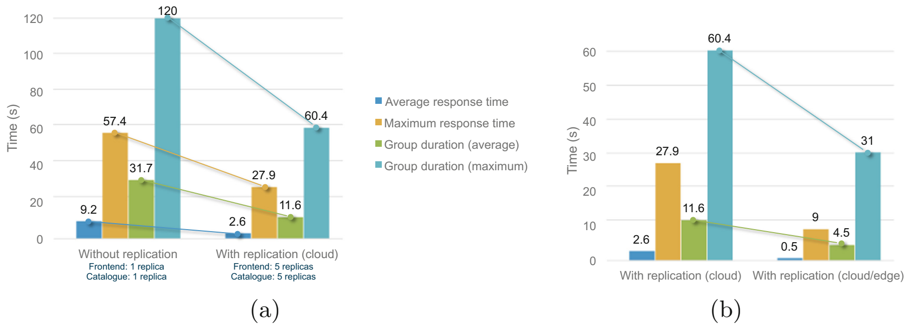
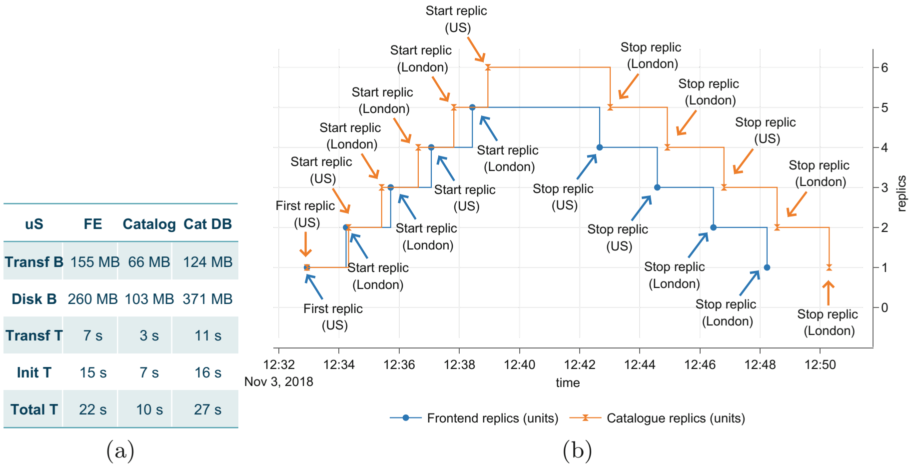


Fig. 5. (a) Cloud execution results without (left) and with replication (right). (b) Comparison of cloud only (left) and cloud/edge (right) replication; clients in London.



uS	FE	Catalog	Cat DB
Transf B	155 MB	66 MB	124 MB
Disk B	260 MB	103 MB	371 MB
Transf T	7 s	3 s	11 s
Init T	15 s	7 s	16 s
Total T	22 s	10 s	27 s

Fig. 6. (a) Microservices' replication costs. (b) Catalog test cloud/edge replica variation.

the *Weaveworks' Sock Shop* demo composed of one *front end* (FE) microservice that communicates with seven back end microservices. One is the *Catalog* service managing the socks' catalog data and images stored in its database. The test for the *Catalog* access to retrieve the products' data is an example of inter-service dependency (the test targeting *Login and Registration* had similar results [4]). When a user communicates with the FE service, it contacts the Catalog service which responds with the required data (metadata or the full socks' database). The FE then resends this data to the user. The *evaluation points* were (a) the application's response times; (b) the replication mechanism, i.e. the replicas' number per service, the replicas' execution place, the replication cost; (c) the replication removal mechanism. Figure 5 shows the evaluation results considering the settings in Fig. 3 for obtaining the catalog's metadata (*response time*) and the full catalog (*group duration*), for an increasing number (until fifty) of virtual users. Figure 5a shows the results for the Catalog in the Cloud only with/without

replication. Figure 5b shows the reduction times when the FE and the Catalog are replicated at both cloud and edge nodes. In this case, the applied rules use the *transmitted bytes per second* rate: *uS replication*: replicate when the rate is >= 2.5 MB/s for two consecutive loop iterations; *uS removal*: stop the uS when its rate for three consecutive loop iterations is <0.5 MB/s. Figure 6a presents the transfer bytes (*Transf B*) for the FE and the catalog metadata (*Catalog*) and with its database (*Cat DB*), and their replication costs as transfer time (*Transf T*) and initialisation time (*Init T*) at the target edge node. These seem adequate for a fast replication towards the edge. Figure 5b shows the system's evolution on self-adapting the replicas' number according to the execution conditions. First the replicas are located in the cloud/US but due to client accesses in the London edge node are replicated here and later removed.

4 Related Work

This work follows the concepts of computation offloading and *Osmotic computing* [33] on automatic deploy of microservices in containers into the cloud/edge, for efficient resource usage and service access. This concept admits edge nodes' highly diverse and restricted capacity, whereas existing container managers (e.g. Kubernetes) include too heavy modules for those nodes and target cloud environments. Our work implements microservice replication with the vision that diverse microservices have different requirements and dependencies [4,11,18] and, along with edge resource management needs, require an adaptable tripartite solution on data, monitoring and service management. E.g. selecting a service to migrate/replicate needs adaptable monitoring for evaluating the dynamic evolution of its dependencies/communication and its dynamic database replication [23]. Coexisting solutions like *Caus* and *Enorm* [16,36] offered single-parameter configuration for microservices' auto-scaling on the cloud. *Caus* has no automatic node management nor dynamic uS placement on the edge. *Enorm* supports dynamic uS placement but on a single edge node. Other works offer interesting multi-variable auto-scaling solutions but only in cloud environments or FaaS [5,12,15]. Recent work [31] also uses a MAPE loop [14] for uS adaptive scaling and nodes' saving based on affinity. Another [25] uses an unsupervised learning approach to automatically decompose an application into uS and select the adequate resource type. Both do not consider fog/edge platforms.

5 Conclusions and Future Work

This work defends the autonomic management of microservices applications deployed on hybrid cloud/edge infra-structures relying on three dimensions, service, data, and monitoring self-management, to cope with these systems' complexity. It focuses on the service component based on an automatic microservice migration and replication solution. The approach is evaluated in the context of a demo application deployed in the Amazon AWS. The results show the adaptability of the system in the presence of varied client access scenarios and present

promising values in terms of lower latencies and the system's efficiency. In future the solution will be extended with a hierarchical service managing system and integrated with the adaptable database and monitoring components in progress and a novel security component. The autonomic service will also include machine learning mechanisms to better analyse and predict access patterns.

References

1. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables devops: migration to a cloud-native architecture. IEEE Softw. **33**(3), 42–52 (2016)
2. Bucchiarone, A., Dragoni, N., Dustdar, S., Larsen, S.T., Mazzara, M.: From monolithic to microservices: experience from the banking domain. IEEE Softw. **35**(3), 50–55 (2018)
3. Carlini, S.: The drivers and benefits of edge computing. APC white paper 226
4. Carrusca, A.: Gestão de micro-serviços na Cloud e Edge. Master's thesis, UNL (2018). http://hdl.handle.net/10362/59505
5. Danayi, A., Sharifian, S.: PESS-MinA: a proactive stochastic task allocation algorithm for FaaS edge-cloud environments. In: ICSPIS, pp. 27–31 (2018)
6. Dastjerdi, A.V., Buyya, R.: Fog computing: helping the internet of things realize its potential. IEEE Comput. **49**(8), 112–116 (2016)
7. Dragoni, N., et al.: Microservices: yesterday, today, and tomorrow. Present and Ulterior Software Engineering, pp. 195–216. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67425-4_12
8. Edge, O.: Open edge computing. http://openedgecomputing.org/
9. Evans, D.: The internet of things. Technical report, cisco Systems (2011)
10. Fowler: Microservices. https://martinfowler.com/microservices/
11. Gan, Y. et al.: An open-source benchmark suite for microservices and their HW-SW implications for cloud & edge systems. In: ASPLOS 2019. ACM (2019)
12. Guerrero, C., Lera, I., Juiz, C.: Resource optimization of container orchestration: a case study in multi-cloud us-based applications. J. Supercomput. **74**(7) (2018)
13. Huebscher, M.C., McCann, J.A.: A survey of autonomic computing: degrees, models, and applications. ACM Comput. Surv. **40**(3), 7:1–7:28 (2008)
14. IBM: An architectural blueprint for autonomic computing. Technical report, IBM (2005)
15. Jindal, A., Podolskiy, V., Gerndt, M.: Performance modeling for cloud microservice applications. In: Proceedings of ICPE 2019. ACM, New York (2019)
16. Klinaku, F., Frank, M., Becker, S.: CAUS: an elasticity controller for a containerized microservice. In: Companion of ICPE 2018, pp. 93–98. ACM (2018)
17. Kratzke, N., Quint, P.: Understanding cloud-native applications after 10 years of cloud computing. J. Syst. Softw. **126**, 1–16 (2017)
18. Leitão, J., Costa, P.Á., Gomes, M.C., Preguiça, N.M.: Towards enabling novel edge-enabled applications. CoRR abs/1805.06989 abs/1805.06989 (2018)
19. Mahmud, R., Kotagiri, R., Buyya, R.: Fog computing: a taxonomy, survey and future directions. In: Di Martino, B., Li, K.-C., Yang, L.T., Esposito, A. (eds.) Internet of Everything. IT, pp. 103–130. Springer, Singapore (2018). https://doi.org/10.1007/978-981-10-5861-5_5
20. Marinescu, D.C.: Cloud Computing: Theory & Practice. Morgan Kaufmann, Boston (2013)
21. Mauro, T.: Adopting microservices at netflix. NGiNX (2015)

22. McCarthy, D., Dayal, U.: The architecture of an active database management system. SIGMOD Rec. **18**(2), 215–224 (1989)
23. Mealha, D., Preguiça, N., Gomes, M.C., Leitão, J.A.: Data replication on the cloud/edge. In: PaPoC 2019 Eurosys Workshop. ACM, New York (2019)
24. Mell, P.M., Grance, T.: The NIST definition of cloud computing. NIST (2011)
25. Abdullah, M., Iqbal, W., Erradi, A.: Unsupervised learning approach for web application auto-decomposition into microservices. J. Syst. Softw. **151** (2019)
26. Newman, S.: Building Microservices, 1st edn. O'Reilly Media Inc., Sebastopol (2015)
27. OpenFog: Size & impact of fog computing market. Technical report, OpenFog (2017)
28. Parashar, M., Hariri, S.: Autonomic computing: an overview. In: Banâtre, J.-P., Fradet, P., Giavitto, J.-L., Michel, O. (eds.) UPP 2004. LNCS, vol. 3566, pp. 257–269. Springer, Heidelberg (2005). https://doi.org/10.1007/11527800_20
29. Richardson, C.: Microservices patterns (2017). http://microservices.io/index.html
30. Salehie, M., Tahvildari, L.: Self-adaptive software: landscape and research challenges. ACM Trans. Auton. Adapt. Syst. **4**(2), 14:1–14:42 (2009)
31. Sampaio, A.R., Rubin, J., Beschastnikh, I., Rosa, N.S.: Improving microservice-based applications with runtime placement adaptation. J. Internet Serv. Appl. **10**(1), 1–30 (2019). https://doi.org/10.1186/s13174-019-0104-0
32. Satyanarayanan, M.: The emergence of edge computing. Computer **50**(1), 30–39 (2017)
33. Sharma, V., Srinivasan, K., Jayakody, D.N.K., Rana, O.F., Kumar, R.: Managing service-heterogeneity using osmotic computing. CoRR abs/1704.04213 (2017)
34. Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge computing: vision and challenges. IEEE Internet Things J. **3**(5), 637–646 (2016)
35. Varghese, B., Wang, N., Barbhuiya, S., Kilpatrick, P., Nikolopoulos, D.S.: Challenges and opportunities in edge computing. In: IEEE SmartCloud, NY (2016)
36. Wang, N., Varghese, B., Matthaiou, M., Nikolopoulos, D.S.: ENORM: a framework for edge node resource management. IEEE Trans. Serv. Comput. (2017)
37. Yi, S., Li, C., Li, Q.: A survey of fog computing: concepts, applications and issues. In: Mobidata 2015 Workshop Proceedings. ACM, New York (2015)