Stateless or Stateful FaaS? I'll Take Both!

Carlo Puliafito[§]
Dept. Information Eng.
University of Pisa
Pisa, Italy

Claudio Cicconetti[§], Marco Conti
Inst. Informatics and Telematics
National Research Council
Pisa, Italy

Enzo Mingozzi
Dept. Information Eng.
University of Pisa
Pisa, Italy

Andrea Passarella
Inst. Informatics and Telematics
National Research Council
Pisa, Italy

***Abstract*—Serverless computing has emerged as a very popular cloud technology, together with its companion Function-as-a-Service (FaaS) programming model enabling invocations of stateless functions from clients. An evolution of serverless is now taking place, shifting it towards the edge of the network and broadening its scope to stateful functions, as well. In this paper we argue that stateless vs. stateful is not a dichotomy of the application *per se*, but rather a time-varying property of most (if not all) applications, as confirmed by the analysis of real traces collected in a production environment. Based on this observation, we propose a mathematical formulation of a resource allocation problem that jointly encompasses both operation modes, dubbed *lambda* vs. *mu*, which can be solved efficiently at run-time by an edge orchestrator. We evaluate the proposed solution via simulation experiments in realistic network and workload conditions, which leads the way to the practical realization of a system where applications can freely adapt their current operation mode and optimize their performance at a minimum cost of operation from the network's perspective.**

***Index Terms*—FaaS, Function-as-a-Service, distributed computing, edge computing, stateful functions**

I. Introduction

Edge computing is a powerful extension of cloud computing toward the network edge. It consists of geographically distributed compute nodes located in proximity to access networks (far-edge nodes) or within the core network of the telco operator (near-edge nodes) [1]. Edge nodes run *microservices*: small pieces of code that are often packaged inside containers rather than virtual machines because they are faster to boot up and more lightweight [2]. We can identify two main ways to operate microservices and realize an end user application: Function as a Service (FaaS) vs. Platform as a Service (PaaS).

FaaS was initially designed for cloud data centers [3] but is rapidly gaining momentum in edge computing, too [4]. With FaaS, a microservice (called *function*) can be instantiated in several containers that are equivalent to one another and, hence, can be autoscaled by the platform provider with maximum flexibility. Such an equivalence allows consecutive invocations from the same client to be forwarded to different containers, and a given container to serve multiple clients. Besides, FaaS enables a pure pay-per-use model, where billing is based on the number of function invocations or cumulative execution time, regardless of the rate of invocations. One disadvantage of FaaS is that the containers cannot keep any state associated to the application's session [5]: every time a function is invoked, if needed, it must read (write) the session state from (to) a remote storage service (e.g., located in the cloud), which increases latency and incurs extra costs. From now on, we refer to FaaS containers as *stateless*.

[§]C. Puliafito and C. Cicconetti share the first author role in this paper.

On the other hand, with PaaS a container is dedicated to a user application instance so that: i) all the invocations from the client are forwarded to that container; and ii) that container handles invocations from that client only. Since the container is dedicated to the client, it keeps the session state locally, hence we call it *stateful*. In contrast to the previous approach, this one reduces latency, as session state does not need to be accessed from a remote storage. As a result, this approach is widely used by edge platforms, and also FaaS platforms for edge computing are starting to consider stateful containers as a possible alternative to stateless [6]. Yet, this approach falls short of flexibility and cost-efficiency: in general, a dedicated container is expensive for the user (especially at the edge) as resources are paid for the whole time during which the application is active, which is inefficient with a sporadic use.

In the literature and market technology, the stateless and stateful operation modes are considered as alternatives, with the choice being made by the developer at design time. However, it can happen that the very same application has a heterogeneous usage pattern over time, which may result in degraded performance (during peaks under a stateless approach) or wasted resources (during sporadic use under a stateful approach). Therefore, in this work we get a new perspective, and propose instead to let an application *adapt dynamically* to the best operation mode, i.e., to switch from being stateless to stateful, and *vice versa*, depending on the current conditions. The contribution of this work is threefold:

- we show with a quantitative analysis of public traces obtained in the wild that alternating between operation modes minimizes the cost of operation, in terms of the container renting fees (stateful), function invocations and storage services (stateless), and migration overhead (Sec. III);
- we formulate a problem that jointly optimizes the placement of stateful containers and the distribution of function invocations to stateless containers at the edge, and propose an efficient solution and practical implementation (Sec. IV);
- we evaluate the performance of the proposed system through simulations under realistic network and workload conditions, to identify the key trade-offs incurred by the configuration of the system parameters (Sec. V).

The paper also includes Sec. II to position our work in the state-of-the-art and Sec. VI, which concludes the paper and outlines the future work.

II. Related work

Many big players offer FaaS solutions to their customers, such as Amazon with AWS Lambda, Microsoft with Azure Functions, and IBM with Cloud Functions, just to name a few. Although these systems were initially designed for cloud environments, there are now extensions toward the network edge: Amazon Lambda@Edge, Microsoft Azure Edge Zones, and IBM Edge Functions. All the above platforms adhere to the typical FaaS approach where functions are served as stateless containers. However, we highlight that stateful containers are gradually coming into the picture as a complementary approach. Specifically, Microsoft introduces the concept of *entity functions* [7], which are uniquely identified, dedicated resources that keep the session state locally as an in-memory object. *Long-lived functions* [8] from Amazon and *Durable objects* [9] from Cloudflare are other examples of dedicated resources from commercial FaaS platforms.

Besides platforms from companies, some open-source FaaS solutions are also available, e.g., Apache OpenWhisk, OpenFaaS, Kubeless, and Knative. All of them leverage Kubernetes as orchestration system underneath. In Kubernetes, function instances are called Pods, which can encapsulate one or more containers. Kubernetes defines both stateless and stateful Pods, the latter being implemented by matching persistent volumes to uniquely identified Pods [10].

In the scientific domain, there are some works in the direction of realizing a coexistence of stateless and stateful containers in FaaS systems, especially from the point of view of the programming model to be used and related Application Programming Interfaces (APIs). For instance, Baresi *et al.* [11] describe the proof-of-concept implementation of a FaaS platform for edge computing, based on Apache OpenWhisk, also mentioning stateful containers for uniquely identified resources. *However, none of the works so far consider the possibility for a function to dynamically adapt its operation mode over time, which we have hinted in our previous work [12] and investigate in detail here.*

On the other hand, a well-studied topic in edge computing is the optimal placement of dedicated microservices in the infrastructure. It is known that algorithms that are widely used in cloud data centers cannot be exploited *as-is* at the edge, due to the distinctive characteristics of this environment, e.g., wide-area deployment and resource limitations of edge nodes. As comprehensively described in related surveys [13], [14], most of the scientific works formalize the problem as a linear programming one where the objective function typically aims at optimizing latency, energy, or resource utilization. As optimization constraints, authors usually consider network limitations (e.g., bandwidth capacity or network latency) and compute ones (e.g., available processing power and memory).

Given the relatively newer topic, fewer works instead aim at optimizing the distribution of function invocations to stateless containers. The work in [15] proposes a decentralized framework where entry points to the system take autonomous decisions on where to forward function invocations, based on weights that are dynamically and locally updated to minimize the communication latency. In [16], function invocations are dispatched based on the queue length and service capacity of each container, with the aim to minimize latency. *To the best of our knowledge, there are no works formulating an optimization problem that jointly aims at optimizing placement of stateful containers and dispatching of invocations to stateless containers, which we address in Sec. IV.*

III. Motivation

In this section, we report the findings of our analysis of real FaaS traces collected in a period of two weeks in 2020 on Microsoft Azure Functions and made available in a public dataset[1], thoroughly analyzed in [17]. The dataset contains more than 44 millions of anonymized function invocations from 856 applications. For each invocation a set of data are included, from which we use the following: the timestamp, unique identifiers of the user ID and application name, and a flag specifying whether the application's state has been accessed in read or write mode. The applications sampled in the dataset are very heterogeneous, e.g., the number of daily invocations ranges from very few to millions. Read accesses are 77% of the total.

Our objective is to show that the majority of those applications can benefit from a policy that adapts their stateful vs. stateless nature over time, in terms of some performance metric, which in the following we assume to be the cost of operation under some reasonable simplifying assumptions. In particular, we assume that the cost of a stateful application is given only by the duration of the time window when it is assigned a dedicated container:

$$c_\mu = \Omega_\mu T_\mu, \tag{1}$$

where Ω_μ is the cost per time unit and T_μ is the time units the application spent as stateful. On the other hand, for a stateless application we assume that its cost is given by the number of invocations and the type of state access, as follows:

$$c_\lambda = \xi_\lambda \left(N_\lambda^R + N_\lambda^W\right) + \sigma_\lambda^R N_\lambda^R + \sigma_\lambda^W N_\lambda^W, \tag{2}$$

where ξ_λ is the cost per function invocation, σ_λ^R (σ_λ^W) is the cost per read (write) access, and N_λ^R (N_λ^W) is the number of function invocations with read (write) accesses.

Computing the cost of an application in the dataset with λ-only and μ-only policies is straightforward. For the hybrid case, called $\lambda+\mu$, where an application migrates from stateful to stateless, we have defined two migration costs (τ_{lambda}: from stateful to stateless; τ_{mu}: from stateless to stateful) and implemented the following policy:

[1]https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsBlobDataset2020.md

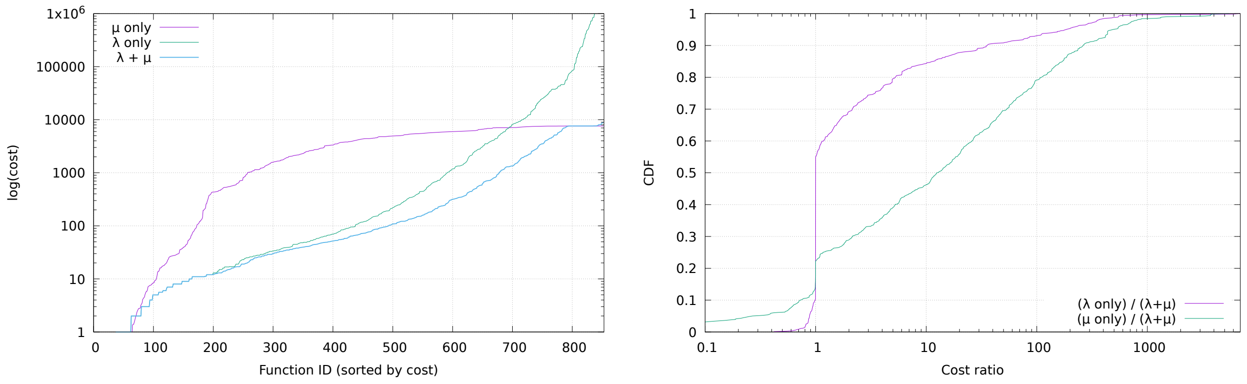

Fig. 1. Comparison of the cost of execution of the applications in the Microsoft Azure Database [17] with λ-only vs. μ-only vs. $\lambda+\mu$ policies, with $\xi_\lambda = 0.6$, $\sigma_\lambda^R = 0.4$, $\sigma_\lambda^W = 5$, $\tau_\lambda = \tau_\mu = 12$, and $\Omega_\mu = 6.3 \cdot 10^{-6}$: absolute (left) and relative (right). All costs in 10^{-6} \$.

- if an application is currently run as stateful, it migrates to stateless if keeping the container occupied until the next function invocation, which costs $\Omega_\mu \left(t_{\text{next}} - t_{\text{now}}\right)$, is more expensive than migrating to stateless right now;
- otherwise, if an application is currently run stateless, we perform a simulation in a look-ahead window of future call invocations in the cases migration-to-μ vs. keep-as-λ; we then migrate it to stateful if a break-even point is reached.

Note that both policies are heuristic but require prophetic powers to predict the precise future pattern of function invocations, which is almost always not available to the platform or the application logic programmer as it depends on external circumstances. However, this assumption is consistent with our goal of showing that a suitable policy *exists*, not how it could be realized effectively in real settings.

The values used for the cost model are reported in the caption and they are inspired from publicly available prices of Amazon Lambda@Edge[2], where (e.g.) the invocation of 1 million functions costs \$0.6, and the cost of a `GET` (`PUT`) operation to read (write) the state is about \$0.4 (\$5) for 1 million operations. In the absence of a more realistic model, the migration cost in either direction has been arbitrarily estimated as twice the cost of function invocation + read + write. The figures reported are purely indicative, e.g., they do not include storage costs and they do not take into account volume discounts, and subject to change depending on the region, provider, as well as to adapt to the evolution of technology and business models. However, we believe these simplified assumptions are sufficient for our purposes. We show in Fig. 1 the costs obtained with the three policies. As can be seen from the left part of the figure, showing the absolute costs, the μ-only and λ-only curves intersect: some applications are better served *always* as stateful while others as stateless, the latter being the majority in the dataset used with the cost model values adopted.

Key observation. *However, by using a $\lambda + \mu$ hybrid policy, the cost can be minimized for all functions, which confirms our intuition that all applications should be able to alternate between stateful and stateless in their lifetime.*

The relative advantage, in terms of cost, of $\lambda+\mu$ compared to λ-only and μ-only, respectively, is shown in the right hand side of Fig. 1: most of the applications have a cost ratio > 1, which becomes substantial for a significant fraction of them, especially in the μ-only case. We note that for very few applications the cost ratio is < 1: this happens because of edge effects of the analysis and only for applications that *absolutely always* are required to remain as either stateful or stateless to minimize their cost. We have decided not to prune the dataset from such applications, for better transparency of the analysis, but such applications have negligible statistical significance, and they are anyway of little interest for our work.

The tool source code and scripts for this cost analysis on the Azure dataset are publicly available on GitHub[3].

IV. System Model

Our system is modeled as follows and illustrated in Fig. 2. We have a set of *clients* that use services provided by edge or cloud nodes, which are reached through *brokers* located at the network edge that represent entry points to the system. We assume for simplicity of notation that each client hosts a single application and we only consider those that are alive and active. The *cloud* resources are assumed to be unlimited, while *edge nodes* have a finite number of containers reserved for the service, but as all the clients are located at the edge of the network it is always "cheaper" to run applications on them compared to the cloud. Such a cost could refer to the use of network resources (point of view of the edge infrastructure operator) or to the latency (point of view of the end users). Note that the "cost" in this section is different from that in Sec. III: the latter is assumed to be minimized by applications by switching back and forth between the μ vs. λ modes of operation to adapt to a changing environment; instead, in the following we adopt the perspective of the infrastructure operator and strive to minimize the *operational costs*.

[2]https://aws.amazon.com/lambda/pricing/?nc1=h_ls

[3]https://github.com/ccicconetti/support, tag `dataset-001`, check out the instructions in `Dataset/001_Mu_Lambda/README.md`.

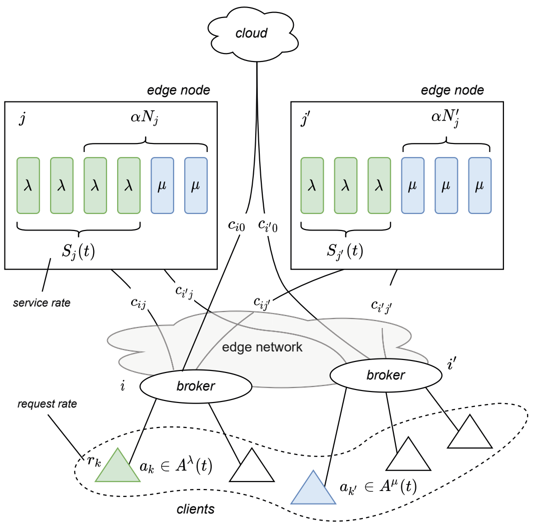

Fig. 2. System model and notation.

At a given time, as already discussed, an application can be in one of two possible states depending on its internal operation and environmental conditions: i) *stateless* (we call it a λ-app), where a pool of containers is shared among a set of applications invoking stateless functions, vs. ii) *stateful* (we call it μ-app), where the application invokes stateful function calls, which require persistence on a dedicated containerized microservice. We assume that the transition from one state to another is mediated in the edge domain by an orchestrator (not shown in Fig. 2), which is in charge of: i) handling transition requests from the applications, ii) deciding whether to assign a container of a μ-app to the cloud or to the edge, and in the latter case on which edge node, and iii) configuring the brokers so that the stateless functions invocations can be dispatched to the containers shared by λ-apps. In this work, we focus on the decision process for resource allocation of μ- and λ-apps, which we model mathematically in Sec. IV-A, and for which we provide a solution and an implementation scheme, respectively in Sec. IV-B and Sec. IV-C. We do not elaborate on the protocols and interfaces that would be needed for the practical deployment, which is left for future work.

A. Problem formulation

We now formally define the resource allocation problem, taking into account jointly the μ- and λ-apps. Again, with reference to Fig. 2, let $A = \{a_k\}$ be the set of application clients and $B = \{b_i\}$ the set of brokers. We define the association y_{ki} between a client a_k and a broker b_i as follows:

$$y_{ki} = \begin{cases} 1, & \text{if } a_k \text{ is bound to } b_i \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

Besides, y_{ki} has the following property:

$$\sum_i y_{ki} = 1, \forall k \tag{4}$$

which states that each client is bound to one broker only. A broker b_i receives function invocations from its clients and dispatches such invocations to containers, which are function instances running on compute nodes. In our system, $E = \{e_j\}$ is the set of compute nodes. Each compute node e_j is assumed to be deployed at the network edge (i.e., edge nodes) and to have N_j containers instantiated on it. The only exception is represented by e_0, which is a cloud node having $N_0 = |A|$ containers instantiated on it.

Moreover, we indicate with $A^\lambda(t)$ and $A^\mu(t)$ the subset of clients requiring at time t to be served by λ- and μ-containers, respectively. At any time t, it occurs that $A^\lambda(t) \cup A^\mu(t) = A$ and $A^\lambda(t) \cap A^\mu(t) = \emptyset$. We also define $x_{kj}(t)$ as follows:

$$x_{kj}(t) = \begin{cases} 1, & \text{if } a_k \text{ has a } \mu\text{-container on } e_j \text{ at time } t \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

At any time t, $x_{kj}(t)$ is subject to the following two constraints:

$$\sum_j x_{kj}(t) = 1, \forall k \in A^\mu(t) \tag{6}$$

$$x_{kj}(t) = 0, \forall k \in A^\lambda(t) \tag{7}$$

Eq. (6) guarantees that each client requiring a dedicated container at time t receives exactly one. Eq. (7) instead states that clients requiring at time t to be served by λ-containers cannot be given at the same time a dedicated μ-container.

To guarantee that, at any time t, enough resources are available to λ-containers on any node e_j, we define a further constraint as follows:

$$\sum_k x_{kj}(t) \leq \alpha \cdot N_j, \tag{8}$$

where $0 \leq \alpha \leq 1$. The above constraint states that the number of μ-containers that are instantiated on node e_j at time t cannot exceed a pre-defined fraction of N_j. Limit cases: $\alpha = 0$ means that μ-containers *cannot* be assigned to edge node j; with $\alpha = 1$ all the resources can be used by μ-containers.

For what concerns clients $a_k \in A^\lambda(t)$, we define r_k as their request rate, i.e., the rate at which those clients invoke λ-containers. Therefore, we can define the request rate exiting any broker b_i at time t as:

$$R_i(t) = \sum_{k \in A^\lambda(t)} y_{ki} \cdot r_k \tag{9}$$

In a similar way, s_j indicates the service rate of a λ-container running on e_j, namely the rate at which that type of container can serve invocations. Given that the cloud node is considered to have unlimited resources, we set $s_0 > \max_k\{r_k\}$. We define the available service rate at time t of any node e_j as:

$$S_j(t) = s_j \cdot \sum_k (1 - x_{kj}(t)), \tag{10}$$

which is an aggregate of the service rates of all the containers in e_j that are not assigned to μ-apps.

Any broker b_i dispatches invocations to λ-containers by distributing such invocations toward compute nodes, based on weights $w_{ij}(t)$: over a sufficiently large time horizon, the ratio between the function invocations dispatched by the broker i toward the edge nodes 1 and 2 will be w_{i1}/w_{i2}. At any time t, these weights are subject to the following three constraints:

$$w_{ij}(t) \geq 0, \forall i, \forall j \tag{11}$$

$$\sum_j w_{ij}(t) = 1, \forall i \tag{12}$$

$$\sum_i w_{ij}(t) \cdot R_i(t) \leq \beta \cdot S_j(t), \forall j, \tag{13}$$

where $0 < \beta < 1$. Specifically, constraint (13) ensures stability by stating that at any time t the request rate entering any node e_j cannot exceed a fraction of the available service rate of that node. Parameter β is introduced to allow for some service capacity over-provisioning.

Finally, we define $c_{ij} > 0$ as a cost over the path interconnecting b_i and e_j. Following the considerations made at the beginning of Sec. IV, this cost could be related to the usage of network resources, to the communication latency, or to a combination of both. Note that for any broker b_i, we set $c_{i0} > \max_j\{c_{ij}\}$, which means that reaching the cloud node is always more expensive than reaching any edge node.

Given the above definitions and constraints, we formulate the following optimization problem:

$$\min_{x_{kj}(t), w_{ij}(t)} \left\{ \Omega \sum_{k,i,j} c_{ij} \cdot x_{kj}(t) + \sum_{i,j} c_{ij} \cdot w_{ij}(t) \cdot R_i(t) \right\}, \tag{14}$$

where Ω is big enough that the first term always dominates over the second one. The above problem aims at instantiating μ-containers on compute nodes and finding the weights $w_{ij}(t)$ that allow to dispatch invocations to λ-containers so as to minimize a combined overall cost in the system.

Key observation. *The objective function in Eq.* (14) *stipulates that the use of edge resources is preferred for μ-applications, which is counterbalanced by the selection of a minimum amount of containers $(1-\alpha)N_j$ reserved for λ-applications in each edge node e_j.*

B. Solution

The constraints Eq. (3)-Eq. (13) and the objective function Eq. (14) form a mixed integer linear programming problem, as the variables $x_{ky}(t)$ (integer) and $w_{ij}(t)$ (real) only exhibit linear relationships. Furthermore, thanks to our assumption that $\Omega \gg 1$, it is possible to separate the problem into two sub-problems, which can be solved sequentially and still achieve the global optimum, as given by the following objective functions with the following procedure at time t:

1. μ-apps allocation sub-problem: find $x_{kj}(t)$ with objective function Eq. (15):

$$\min_{x_{kj}(t)} \sum_{k,i,j} c_{ij} \cdot x_{kj}(t), \tag{15}$$

which means that all the μ-apps will be assigned to a container on the edge nodes (or in the cloud). As a result of the allocation in the previous step, all the containers for which it is $x_{kj}(t) = 1$ will not contribute to the execution of λ-app function invocations, as $(1 - x_{kj}(t))$ will be 0 in Eq. (10).

2. λ-apps allocation sub-problem: find $w_{ij}(t)$ with objective function Eq. (16):

$$\min_{w_{ij}(t)} \sum_{i,j} c_{ij} \cdot w_{ij}(t) \cdot R_i(t), \tag{16}$$

which means that load balancing of λ-apps at each broker b_i will happen in accordance with the weights found; we recall that stability is ensured by Eq. (13).

Both sub-problems in steps 1 and 2 above are instances of well-known optimization problems. More specifically, the first one is a case of *assignment problem* and the second one of *transportation problem*, and both can be solved (exactly) with efficient algorithms from the operations research literature.

For example, to carry out the performance evaluation in the next section, we use the following algorithms: for the μ-apps allocation problem we adopt the Hungarian method, which has $\mathcal{O}(|A^\mu(t)|^3)$ worst-case time complexity; on the other hand, we transform the λ-apps allocation problem into an equivalent minimum cost flow problem, which we then solve using the "successive shortest path", having worst-case time complexity $\mathcal{O}\left(\bar{R} \cdot (E + V \log V)\right)$, where:

$$\begin{aligned}
\bar{R} &= \sum_{k \in A^\lambda(t)} r_k, \\
E &= \bar{A} \cdot \bar{N} + 2 \cdot (\bar{A} + \bar{N}), \\
V &= 2 \cdot (\bar{A} + \bar{C} + 1), \\
\bar{A} &= |A^\lambda(t)|, \\
\bar{N} &= \sum_j \left[N_j - \sum_k x_{kj}(t) \right].
\end{aligned}$$

C. Overall operation

The solution illustrated in the previous section provides the optimal allocation, under the given constraints and costs, for a given set of applications $A^\lambda(t) \cup A^\mu(t)$. In principle, this implies that whenever any of the following happens, the algorithm has to be re-run: i) a new application becomes active; ii) an active application become inactive; iii) an application migrates from μ-app to λ-app or *vice versa*. If the population of users is large or the frequency of changes is high, the allocation will have to be adjusted very often, which in turn has two consequences. First, the orchestrator may become a performance bottleneck: even though we have formulated the problem so that efficient solutions can be used, finding an exact solution in a short amount of time can be a challenge with large problem instances, which needs to be addressed either by using a big amount of computational resources (costly and with environmental sustainability concerns) or by finding approximate solutions (possibly degrading the performance). Second, whenever a new allocation of resources is found by the orchestrator, some of the current μ-apps may need to be migrated from one edge node to another, which is undesirable both for the edge infrastructure operator (network resources are consumed) and for the end users (possible service interruptions and latency spikes).

To address this issue we propose to run the resource allocation algorithm in Sec. IV-B only periodically (we call the period *epoch*). Of course, asynchronous events may happen in between epochs, which we can handle in a best-effort manner as follows:

1. a new λ-app becomes active: we configure brokers in such a way that invocations of unknown functions are directed automatically to the cloud, hence no reconfiguration is needed at the edge (even though the allocation of w_{ij} is sub-optimal in general);
2. a new μ-app becomes active: the orchestrator assign it to the edge container with minimum cost c_{ij} among those edge nodes j that have available resources for this according to Eq. (8), which is a $\mathcal{O}(|E|)$ operation; if no such edge node is available, then the container is created in the cloud; in any case, no re-allocation of the other currently active μ-apps is done;
3. a λ-app becomes inactive: no operation needed on the brokers, who will simply not receive anymore function invocations from the corresponding client;
4. a μ-app becomes inactive: the container assigned is deallocated, which frees resources on the corresponding node;
5. migration $\mu \rightarrow \lambda$: execute the actions in point 4 followed by those in point 1;
6. migration $\lambda \rightarrow \mu$: execute the actions in point 2.

The epoch duration is a system parameter that has to be tuned appropriately since it incurs a performance trade-off, which we study with simulations in Sec. V.

V. Evaluation

In this section we assess the performance of the framework proposed in Sec. IV using numerical simulations. For reproducibility purposes, the tool used is released as open source on GitHub[4], together with the artifacts and the scripts to run the experiments and analyze the output.

The network topology used is depicted in Fig. 3: it was generated with "ether: Edge Topology Synthesizer"[5], which produces realistic edge network models with mixed compute nodes: end-user devices (our brokers, in blue), small PCs (far-edge nodes, in yellow), and servers (near-edge, in green). We have used as network cost c_{ij} the logical distance between broker i and edge node j, in number of hops. The cloud node is not included in the topology but it is considered in the resource allocation with cost $c_{i0} = 2 \cdot \max\{c_{ij}\}$. The request rate of λ-apps is assumed to be $r_i = 1$ for all applications, while the service rate is $s_j = 10$ for far-edge nodes and $s_j = 20$ for near-edge nodes; similarly, the number of containers available in each node is $N_j = 4$ in far-edge devices and $N_j = 8$ in near-edge devices. In the following we report the results obtained with two types of simulation: snapshot and dynamic.

[4]https://github.com/ccicconetti/serverlessonedge/tree/v1.3.0, see instructions in `README.md` within simulations `010`, `011`, and `012`.

[5]https://github.com/edgerun/ether.

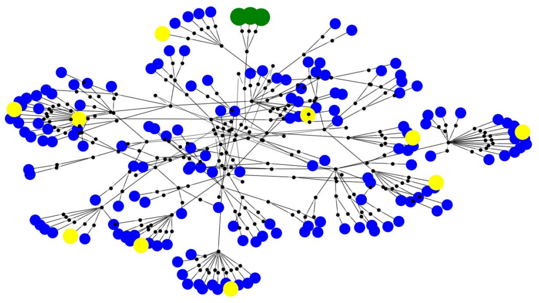

Fig. 3. Network topology: blue nodes are the brokers, yellow nodes are far edges, green nodes are near edges, black dots are network devices.

A. Snapshot simulations

Snapshot simulations follow a Monte Carlo approach: a number of λ-apps (μ-apps) is drawn from a Poisson distribution with mean $E[|A^\lambda|]$ ($E[|A^\mu|]$); each app is assigned to a random broker selected independently from those available with uniform distribution probability; we execute the algorithms in Sec. IV-B and find the costs of λ- and μ-apps. We then repeat the same process for several replications (6400 in our experiments), all contributing to the same experiment instance. We have run instances with $E[|A^\lambda|] = 50$ while changing $\alpha \in \{0/8, 1/8, \dots, 7/8\}$, $\beta \in \{0.1, \dots, 0.9\}$, and $E[|A^\mu|] \in \{25, 50, 75\}$ in a full combinatorial manner.

In Fig. 4 we study the effect of α and β on the (unitary) cost of λ-apps, defined as the value of the summation in Eq. (16) divided by the number of λ-apps in the snapshot. As the fraction of containers per edge node reserved to λ-apps is $(1-\alpha)$, the cost increases with α. The impact of α is much more prominent with small values of β, when the cost is higher. In fact, β controls how much margin the orchestrator should reserve to cope with deviations of the actual rate of requests from applications compared to the nominal values s_i provided to the algorithm: e.g., $\beta = 0.1$ means that we consider only 10% of the nominal service rate capacity in each container, which increases the use of the cloud (= higher cost) compared to bigger values of β. As can be expected, all curves tend asymptotically to a minimum cost, which is that incurred when serving all the λ-apps on far-edge nodes.

On the other hand, the impact of α on μ-apps is shown in Fig. 5, in terms of the fraction of μ-apps that are assigned a dedicated container in the cloud, hence at a higher cost in our model: the curves with all values of $E[|A^\mu|]$ decrease almost linearly. The results confirm in a quantitative manner the intuition that the choice of α creates a trade-off between the performance of μ-apps (better with bigger α) and λ-apps (better with smaller α), which leads the way to an auto-tuning

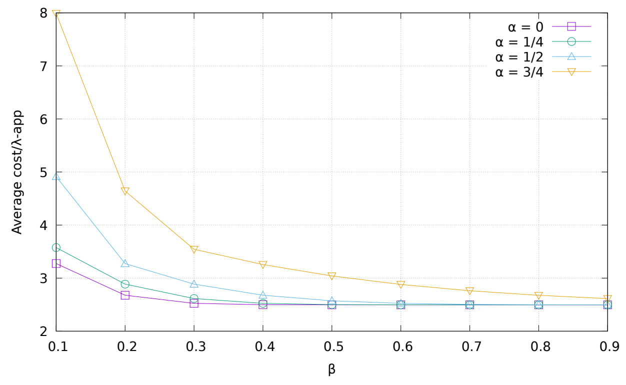

Fig. 4. Average unitary cost of λ-apps vs. β for various values of α, with $E[|A^{\lambda}|] = E[|A^{\mu}|] = 50$.

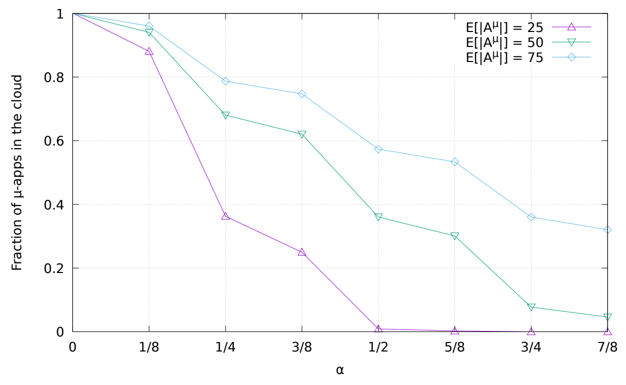

Fig. 5. Number of μ-apps assigned to the cloud vs. α.

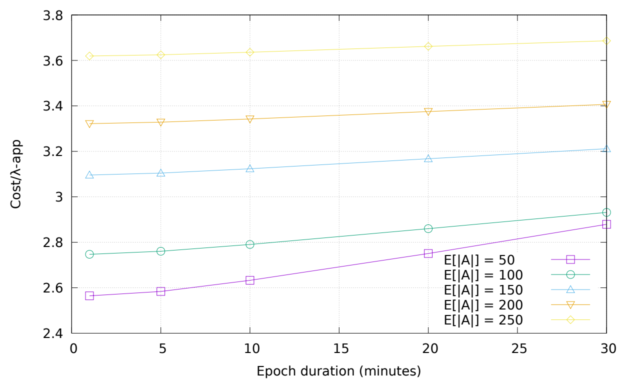

Fig. 6. Average cost of λ-apps vs. epoch duration for different workloads.

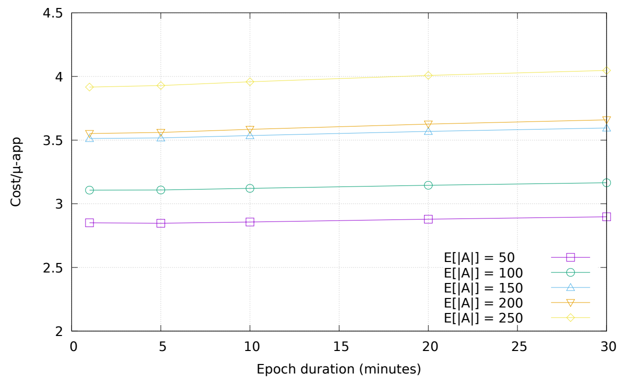

Fig. 7. Average cost of μ-apps vs. epoch duration for different workloads.

of this parameter based on long-term optimization objectives, which is left for future work.

B. Dynamic simulations

In the dynamic simulations we use the Microsoft Azure traces described in Sec. III to drive an event-driven simulation of i) the clients, whose applications alternate over time between the μ and λ operation modes according to the pattern that minimizes the respective user cost, and ii) the orchestrator, which performs both periodic optimization in Sec. IV-B and the best-effort measures reported in Sec. IV-C. For simplicity of analysis, we set $\alpha = \beta = 0.5$. Each simulation replication lasts 24 hours of simulated time and we run 6400 replications for each epoch duration, which increases from 1 minute to 30 minutes (we discard the samples in the first epoch as warm-up period to reduce initial bias effects). The workload is composed of a number of applications drawn from a Poisson distribution with average $E[|A|] \in \{50, \dots, 250\}$, each assigned to a random broker in the network and exhibiting a random operation mode pattern from the traces, with randomized initial offset and wrap-around at the trace's end.

As can be seen in Fig. 6, for all workloads, the unitary cost of λ-apps increases with the epoch duration. This is because with larger optimization periods, there are (on average) more asynchronous transitions to λ operation mode, which triggers the best-effort procedure in Sec. IV-C that merely directs them to the cloud. The increase is more prominent with lighter workloads (i.e., $E[|A|] = 50, 100$) where the cloud is used more sparingly when solving the λ-apps allocation.

The cost of μ-apps also increases with the epoch duration (see Fig. 7), but only slightly because the best-effort procedure in Sec. IV-C recycles containers that are currently available for μ-apps. In Fig. 8 we report the cumulative distribution over all the replications of the unitary cost of μ-apps, for the representative case of 1 minute epoch duration. As expected, the cost increases with the workload, but it is interesting to note that from $E[|A|] = 100$ to $E[|A|] = 150$ there is a wider gap, which happens because that is precisely when the orchestrator begins to use the cloud due to a shortage of edge resources for μ-apps. For a similar reason the curves $E[|A|] = 150$ and $E[|A|] = 200$ are almost overlapping: in that area, the marginal cost of adding more μ-apps is small because most of them are still served by edge nodes.

We conclude with Fig. 9, which shows the migration rate of μ-apps caused by executing the periodic optimization. As can be seen, all the curves decrease significantly with the epoch duration. This suggests that this system parameter should be set to a large value, which also reduces the rate of execution

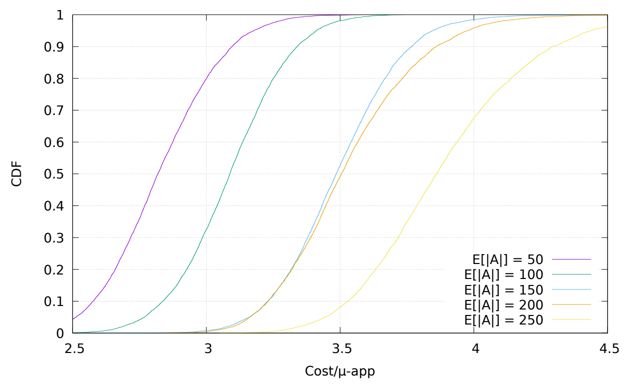

Fig. 8. Distribution of the cost of μ-apps with 1 minute epoch duration.

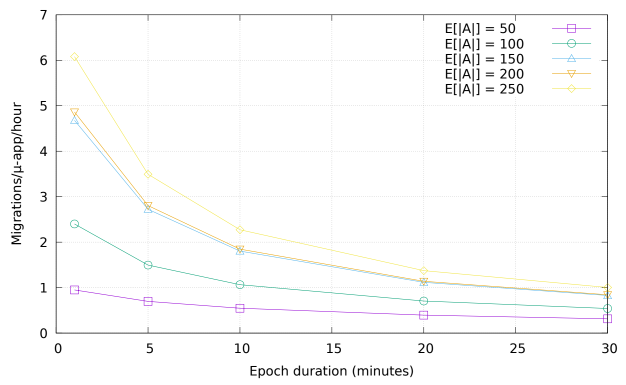

Fig. 9. Hourly rate of migrations vs. epoch duration for different workloads.

of the optimizations, hence the computational burden on the orchestrator. However, this indication is in contrast with the cost analysis of both λ- and μ-apps, thus a fundamental trade-off exists, which we plan to investigate more deeply in our future work, under realistic migration overhead costs.

VI. Conclusions

In this paper we have taken a novel perspective on the schism between the stateless (λ) and stateful (μ) operation modes for edge-cloud applications. In particular, based on the analysis of publicly available traces collected in a Microsoft Azure production environment, we have found that applications can benefit, in terms of operation costs, from alternating between the two operation modes over time. This observation has led us to the definition of a mixed integer linear problem that jointly optimizes the resource allocation, in terms of containers assigned to μ-apps and load distribution for λ-apps, depending on the instantaneous mode preferred by each application. We have formulated the problem so that it can be solved efficiently in two sequential steps, meant to be executed periodically, and we have proposed best-effort measures to handle asynchronous changes in between consecutive optimization runs. We have evaluated the performance of the proposed solution through comprehensive simulation experiments on a synthetic, but realistic, edge network topology and using a trace-driven workload composition. The results have shown that our framework is flexible enough to adapt to a wide set of scenarios through the configuration of system parameters, including: the fraction of containers reserved for λ-apps $(1-\alpha)$, the over-provisioning factor to absorb the peaks of λ-apps (β), and the epoch duration. In our future work we will investigate how to set dynamically these parameters to achieve long-term optimisation objectives, we will study the impact of realistic migration overheads, and we will analyze the management plane protocols and programming interfaces for a practical implementation.

Acknowledgment

This work was partially supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No 957337, project MARVEL, and by the Italian Ministry of Education and Research (MIUR) in the framework of the CrossLab project (Departments of Excellence).

References

[1] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed, "Edge computing: A survey," *Elsevier Fut. Gen. Comp. Sys.*, vol. 97, pp. 219–235, 2019.

[2] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, "Consolidate IoT Edge Computing with Lightweight Virtualization," *IEEE Network*, vol. 32, pp. 102–111, 2018.

[3] E. van Eyk *et al.*, "The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms," *IEEE Int. Comp.*, vol. 23, pp. 7–18, 2019.

[4] R. Xie, Q. Tang, S. Qiao, H. Zhu, F. Richard Yu, and T. Huang, "When Serverless Computing Meets Edge Computing: Architecture, Challenges, and Open Issues," *IEEE Wire. Comm.*, pp. 1–8, 2021.

[5] S. Eismann *et al.*, "The State of Serverless Applications: Collection, Characterization, and Community Consensus," *IEEE Trans. on Soft. Engin.*, 2021.

[6] P. G. Lopez, A. Slominski, M. Behrendt, and B. Metzler, "Serverless Predictions: 2021-2030," Tech. Rep., 2021. [Online]. Available: http://arxiv.org/abs/2104.03075

[7] Microsoft, "Entity functions," Dec. 2019. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities?tabs=csharp

[8] Amazon, "Run Lambda functions on the AWS IoT Greengrass core," 2019. [Online]. Available: https://docs.aws.amazon.com/greengrass/v1/developerguide/lambda-functions.html#lambda-lifecycle

[9] G. McKeon, "Durable objects - now generally available," 2021. [Online]. Available: https://blog.cloudflare.com/durable-objects-ga/

[10] Kubernetes, "StatefulSets," 2020. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

[11] L. Baresi and D. Filgueira Mendonca, "Towards a serverless platform for edge computing," in *IEEE ICFC*, 2019, pp. 1–10.

[12] C. Puliafito, C. Cicconetti, M. Conti, E. Mingozzi, and A. Passarella, "Stateful function-as-a-service at the edge," *IEEE Computer*, 2022, in press.

[13] B. Sonkoly, J. Czentye, M. Szalay, B. Nemeth, and L. Toka, "Survey on placement methods in the edge and beyond," *IEEE Commun. Surv. & Tutor.*, vol. 23, pp. 2590–2629, 2021.

[14] F. A. Salaht, F. Desprez, and A. Lebre, "An overview of service placement problem in fog and edge computing," *ACM Comput. Surv.*, vol. 53, 2020.

[15] C. Cicconetti, M. Conti, and A. Passarella, "A decentralized framework for serverless edge computing in the internet of things," *IEEE Trans. on Net. and Serv. Man.*, vol. 18, pp. 2166–2180, 2021.

[16] V. Mittal *et al.*, "Mu: An Efficient, Fair and Responsive Serverless Framework for Resource-Constrained Edge Clouds," in *ACM SoCC*, 2021, pp. 168–181.

[17] F. Romero *et al.*, "Faa$T: A transparent auto-scaling cache for serverless applications," in *ACM SoCC*, 2021, pp. 122–137.