Fog Orchestration meets Proactive Caching

Francescomaria Faticanti*†, Lorenzo Maggi‡, Francesco De Pellegrini§, Daniele Santoro*, Domenico Siracusa*
*Fondazione Bruno Kessler, Trento, Italy
†University of Trento, Trento, Italy
‡Nokia Bell Labs, Paris, France
§University of Avignon, Avignon, France

***Abstract*—Running fog computing applications on edge servers requires to match activation of applications containers to time varying demands. In this paper we study the dynamic orchestration of a batch of applications over a network infrastructure including fog servers and a cloud. Cloud application deployment faces higher cost and high latency, but unlimited computational capacity. Fog servers, conversely, have limited computational resources, but ensure low latency at low cost. In this context we propose a new scheme for joint caching and placement in fog: the aim is to minimize the deployment cost while satisfying the applications' constraints. In fact, image caching appears mandatory to reduce the containers' activation time. On the other hand, proactively caching images on target servers is effective to match the expected activation pattern while optimizing load balancing via container replication. Using two-stage stochastic programming we derive a one-step-ahead policy to minimize the total running cost and satisfy applications' requirements. Extensive numerical results demonstrate the potential for this novel approach over traditional caching and placement algorithms.**

***Index Terms*—fog computing, proactive image caching, orchestration, two-stage stochastic programming**

I. Introduction

The IoT technology is meant to connect a massive number of smart objects and devices to the Internet to serve their data to services and applications. In order to process the resulting IoT data streams at the edge of the network, the fog computing paradigm has extended cloud computing to support local computation on edge servers. Proximity to data sources [1] improves user experience for IoT services reducing round-trip delays whereas information extraction at the network's edge prevents massive, diffused and continuous raw data injection into the communication infrastructure [2].

Virtualization is key for the flexible installation of services onto fog servers in the proximity of IoT objects [3]. In fact, heterogeneity of IoT technologies [4] is mitigated by packaging fog service modules in advance, e.g., in the form of Docker *images* adapted to the host OS system. Container-based orchestrators such as, e.g., Kubernetes [5], support availability and load balancing by means of container replication. Replicas of the image of a tagged fog application can be displaced on different target fog servers: once replicated among different servers, requests towards a tagged application are dispatched by applying a load balancing procedure (e.g, round-robin) among the servers. On the other hand, for a fog server to offload some container, it has to query the controller for the network topology in order to check which servers executing same application can take over. The Kubernetes monitoring ensures service availability, by removing stalled containers and activating replicas on different servers.

Typically, containerized services can not be migrated in a stateful manner with a proper infrastructure. Rather, the practice is to replicate and switch on a container at the new location and to turn off the old one. Clearly, this process introduces some delay which can be broken down into a *start up time*, a platform-dependent *orchestration overhead delay*, and the *image transfer delay*. The last component is critical because it involves unpredictable network delays and depends on the image size. Thus, caching containers' images on fog servers is key to reduce the total delay experienced by fog applications. For instance, under Kubernetes, every node, as a Docker host, can perform image caching operations. A cache hit means that the container image is available on a target fog server and, if not active, the start-up time can be performed within some milliseconds [6]. A cache miss, conversely, requires either to wait the image download onto the fog server – a delay usually unacceptable for most applications – or to redirect the request on other available instances, e.g., in cloud. Further, every new image download involves several operations, such as the image decompression, with an overhead on the server's CPU usage much higher than image activation.

This paper studies the joint optimization of application image caching and orchestration in fog. We consider single container fog applications, and we leave the extension to more complex microservice architectures for later works. The system infrastructure includes a central cloud and a fog region with a cluster of fog servers. Fog servers have limited CPU, memory and storage capacity but are cost free. In cloud, unlimited computational resources are available at a cost. The aim is to optimally orchestrate applications running on the described infrastructure. I.e., minimize deployment costs while complying with applications' delay figures.

Inspired by proactive caching systems, we study an optimal joint orchestration and caching policy. Proactive caching is aimed at preventing the download of containers from a central repository to avoid large image transfer delay. The control is the proactive placement of the applications' images stored in the container registries across the network infrastructure,

This work has received funding from the EU H2020 R&I Programme under Grant Agreement no. 815141 (DECENTER: Decentralised technologies for orchestrated Cloud-to-Edge intelligence).

978-3-903176-32-4

mapping applications' containers to the fog servers or to the cloud. Each application's container can be either cached or not on a fog server. Also, it can be available in two forms on each server: it can be either *active*, i.e., the container is running on the server, or it can be *disabled*, i.e., the image of the container is cached but not yet running.

State-of-the-art solutions for images caching, such as Kubernetes' caching for instance, are *reactive* and thus not suitable for delay-constrained fog computing applications. In fact, a container not present on a target host is fetched from the central repository (e.g., a Docker hub). Proactive edge caching has become popular in 5G networks [7] and has appeared recently in the literature [8].

However, fog proactive caching appears fundamentally different compared to standard multimedia edge caching. First, it maybe tempting leveraging on standard caching policies as done in proactive multimedia edge caching. Most frequently used (MFU), for instance, is provably optimal under stationary content popularity [9]. In fog computing such analogy is misleading because the more requests a container receives, the more the processing resources consumed in terms of CPU, storage and communications on the servers it is installed on. Thus, the server "occupation" depends on the fog application's "popularity" and a fog application's performance depends by the presence of other active containers on the same host. As proved by our numerical experiments, traditional caching techniques are suboptimal for the considered problem, especially under variable demand patterns.

Main contribution: we propose a one-step-ahead joint caching and orchestration scheme accounting simultaneously for caching, load-balancing, and replication of fog containers' images. One application may be cached in advance on several servers and may be activated or not depending on the current application's load. In our model, the CPU load generated at a specific server depends on the number of replicas installed elsewhere and made run in parallel precisely to smooth the peak processing delay per instance. Our work is, to the best of the authors' knowledge, the first one to study jointly proactive containers' image caching and orchestration.

The paper is structured as follows. Section II introduces the system model, and in Section III the joint caching and orchestration problem is presented in the form of a two-stage stochastic optimization problem. Section IV describes the main methods adopted for the problem resolution, and Section V reports on numerical results. A concluding section ends the paper.

II. System Model

We consider a network infrastructure consisting of a fog cluster and a cloud. The fog cluster is composed by S *fog servers*. Each server s has a limited amount of CPU, memory and disk storage available, denoted by the triple $\mathbf{C_s} = (C_s^P, C_s^M, C_s^D)$. On the other hand, in cloud there are unlimited but expensive resources.

Applications. We consider a batch of N different applications to be run on the infrastructure; they can be deployed either in cloud or on the fog cluster. Each application consists of a container to be replicated across several fog servers and/or in cloud. Each container of application i has requirements in terms of CPU, memory and storage, described by the triple (c_i^P, c_i^M, c_i^D).

TABLE I
Main notation used throughout the paper

Symbol	*Meaning*
N	number of applications
S	number of fog servers
λ_i^t	arrival rate at time t for application i
δ_i	maximum processing delay tolerable by application i
r	resource type: processing (P), memory (M) and storage (D).
C_s^r	available resource in server s, with $r \in \{P, M, D\}$
c_i^r	resource occupation of a container of application i

Cache. We assume that each fog server has the ability to store containers in a *cache*, and possibly to disable cached containers when not needed. This allows to predict future application requests and reduce startup delays for new applications, that is to download the respective containers if not cached locally. We consider that each container may consist of successive *layers*, corresponding to incremental software updates. Yet, out of all layers downloadable from a central repository, a container of application i only needs a portion $\eta_i \in (0, 1]$ of them to run properly.

Control. The orchestrator decides at each period t: i) which new containers should be downloaded to the fog servers' caches and ii) which containers, among those who are are present in the respective caches, should be activated. We remark that the activation policy is constrained by the delay requirements of each application, and we shall provide some constraints on the churn rate of cached images in order to disincentive disruptive reconfigurations.

Let $x_{i,s}^t$ be the binary variable indicating whether a container of the application i is active on server s at time t ($s = 0$ denotes the cloud for the notation's sake). The caching control is represented by continuous variable $y_{i,s}^t \in [0, 1]$: it describes the fraction of container's layers cached on server s at time t for application i. In fact, containerized applications may be operational even when some features are missing. But, if $y_{i,s}^t < \eta_i$, i.e., the portion of cached layers is insufficient, then $x_{i,s}^t = 0$, hence the application can not be activated. We denote $\mathcal{A}(y^t) = \{(i, s) | y_{i,s}^t < \eta_i\}$ the set of containers that can not be activated: $x_{i,s}^t = 0$, for all pairs $(i, s) \in \mathcal{A}(y^t)$.

Requests. For the sake of model tractability we divide the time into slots. During slot t, application i receives data to be processed at a rate of λ_i^t – that can be thought of as a measure of application's *popularity* – and we denote by $\bar{\lambda}^t$ the vector of all arrival (or demand) rates. Although theoretically a time-slot can be defined at one's will, in practice we suggest to define a new slot whenever the arrival rates change considerably. This assumption is aligned with the majority of monitoring systems. Indeed, in most of the existing systems as Kubernetes [5], a new orchestrating decision is taken whenever a change in the applications arrival rates is detected and a potential inconsistency between the required and guaranteed requirements is detected. We remark that demand rates λ at

future slots $t' > t$ are not know, but can only be predicted; a discussion on models for IoT data traffic is beyond the scope of this work [10].

Latency. The presence of several active applications on the same fog servers impacts the processing time of each of them. The processing delay $d^t_{i,s}$ experienced by a tagged application i on a server s can be modelled as a convex function of the application's demand rate, and of the the number of applications active on the server and processing capacity of the server:

$$d^t_{i,s}(\bar{x}^t_s, \lambda^{t+1}_i) = \left(\frac{c^P_i}{\sum_{i=1}^N x^{t+1}_{i,s}} - \frac{\lambda^t_i}{\sum_{s=0}^S x^t_{i,s}} \right)^{-1} \quad (1)$$

By (1) over-exploiting fog servers by increasing container replicas reduces the CPU share each receives thus increasing the processing delay. Conversely, a uniform load balancing policy can split the demand rate per application evenly across all servers on which the application is active. On the other hand, in cloud there is no computational bottleneck, so that a container of application i activated in cloud has constant processing time d^i_0. However, we assume a fixed latency Δ_0 between the fog cluster and cloud.

Cache churn rate. The cache storage occupation can not exceed the cache capacity C^D_s, i.e., $\sum_{i=1}^N y^t_{i,s}\, c^D_i \leq C^D_s$, for each fog server s. Also, due to limited download speed, we fix an upper bound ϵ on how much cache content can vary between consecutive slots: a fair resource share imposes $|y^{t+1}_{i,s} - y^t_{i,s}| \leq \epsilon$ for each application i and server s.

Orchestration constraints. We assume application i to tolerate a maximum processing delay of δ_i seconds. When active in fog, this is described by constraint $d^t_{i,s}(\bar{x}^t_s, \lambda^t_i)\, x^t_{i,s} \leq \delta_i$ for any fog server s. In cloud, the fog-to-cloud latency is factored in as $(d^i_0 + \Delta_0)\, x^t_{i,0} \leq \delta_i$.

Furthermore, CPU and memory occupation of all containers activated on a given server s can not exceed the total CPU and memory available at time slot t, i.e., $\sum_{i=1}^N x^t_{i,s}\, c^r_i \leq C^r_s$ for every request $r \in \{P, M\}$.

Finally, every application i for which $\lambda^t_i > 0$ must be served at time slot t, so that at least one active container is needed, i.e., $\sum_{s=0}^S x^t_{i,s} \geq 1$.

Objective. In order to minimizing the financial cost of running the overall infrastructure, we aim at minimizing the number of containers $\sum_{i=1}^N x^{t+1}_{i,0}$ deployed in cloud over an horizon of T slots. We do so by jointly controlling caching and activation of containers, while fulfilling caching and orchestration constraints, per server and per application as described above.

III. Solution: One-step ahead programming

Our goal is to devise an orchestration policy that jointly decides the caching and the activation of containers for the requested applications with the aim of minimizing the number of containers deployed in the cloud. This allows to eventually minimize the overall deployment cost.

The frequency at which new requests arrive and (already cached) containers are activated is much higher than the frequency at which new containers can be downloaded and cached at fog servers [6]. For this reason, we decide to articulate the decisions on caching and activation in hierarchical and sequential fashion. Caching decision are planned in advance in stage 1, to anticipate the scenario that may materialize in stage 2 once the container download is terminated. Finally, at stage 2, new containers are cached in the servers and they are activated in accordance with the actual requests.

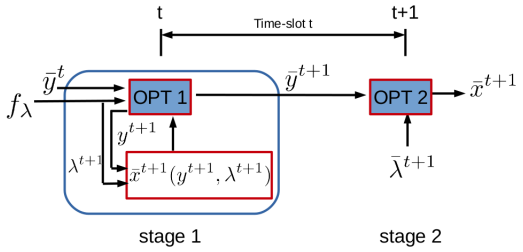

Fig. 1. Two-stage optimization and its one-step ahead (OSA) solution

We remark that the two stages are interleaved in time: at time t, the caching decisions are made (stage 1), while containers are activated given the caching decisions taken at previous time $t-1$ and materialized at time t (stage 2).

Technically speaking, our solution approach follows the classic paradigm of *two-stage stochastic programming* [11] and its natural *one-step ahead* (OSA) associated solution, illustrated in Figure 1 and described formally below. We will describe the two stages in backward fashion, since stage 1 relies on the solution of the problem solved by stage 2.

OSA - Stage 2: Activation of cached containers. By time $t+1$, the new arrival rates $\bar{\lambda}^{t+1}$ are observed. Moreover, we assume that stage 1 has already taken place at time step t, producing caching decision $\bar{y}^{t+1}$; hence, by time $t+1$ new containers are downloaded accordingly. Then, as the second stage of our optimization, cached containers are activated so as to minimize the number of them in cloud, while fulfilling all the orchestration constraints. The corresponding mathematical program writes as follows.

`Input`: Actual arrival rates $\bar{\lambda}^{t+1}$, container caching $\bar{y}^{t+1}$
`Output`: Container activation $\bar{x}^{t+1}(\bar{y}^{t+1}, \bar{\lambda}^{t+1})$
`Compute` at time $t+1$:

$$\bar{x}^{t+1}(\bar{y}^{t+1}, \bar{\lambda}^{t+1}) = \underset{x^{t+1}}{\operatorname{argmin}} \quad \sum_{i=1}^N x^{t+1}_{i,0} := F^{(2)}(x^{t+1}) \qquad \text{(OPT2)}$$

subject to:

$$\begin{aligned}
& d^{t+1}_{i,s}(x^{t+1}_s, \bar{\lambda}^{t+1}_{i,s})\, x^{t+1}_{i,s} \leq \delta_i, \\
& \forall i \in \{1, \ldots, N\}, \forall s \in \{1, \ldots, S\} \\
& (d^i_0 + \Delta_0)\, x^{t+1}_{i,0} \leq \delta_i, \quad \forall i \in \{1, \ldots, N\} \\
& x^{t+1}_{i,s} = 0, \quad \forall (i,s) \in \mathcal{A}(\bar{y}^{t+1}) \qquad (2) \\
& \sum_{s=0}^S x^{t+1}_{i,s} \geq 1, \quad \forall i : \bar{\lambda}^{t+1}_i > 0 \\
& \sum_{i=1}^N x^t_{i,s}\, c^r_i \leq C^r_s, \quad \forall s \in \{1, \ldots, S\}, \forall r \in \{P, M\} \\
& x^{t+1}_{i,s} \in \{0, 1\}, \quad \forall i \in \{1, \ldots, N\}, \forall s \in \{1, \ldots, S\}.
\end{aligned}$$

OSA - Stage 1: Container caching. At time t, one has to devise a *caching policy* $\bar{y}^{t+1}$ deciding the percentage[1] of

[1]relative to the most recent container version present in the main repository

container layers to be cached in each server. Future arrival rates at time $t+1$ are unknown and can only be predicted, hence $\bar{y}^{t+1}$ is computed by minimizing the *expected* number of containers deployed on the cloud at time $t+1$. Yet, for each *possible* arrival rate scenario λ^{t+1} and for each caching $\bar{y}^{t+1}$ one can compute the optimal container activation $\bar{x}^{t+1}(\bar{y}^{t+1}, \lambda^{t+1})$ through the analogous version of (OPT2).

Hence, we define the objective function $F^{(1)}$ of the first stage optimization as the expected value of the objective realized of stage 2, *i.e.*, $\mathbb{E}_{f_\lambda}[F^{(2)}]$. Here, f_λ is the predicted arrival rate distribution at time $t+1$, *i.e.*, $f_\lambda(a) = \Pr(\lambda^{t+1} = a)$. We can also account for the fact that more up-to-date containers will generally provide better performance by introducing an increasing function $R(.)$ of the number of cached container layers, which results in the following objective function

$$F^{(1)}(y^{t+1}, \bar{x}^{t+1}) := \mathbb{E}_{f_\lambda}\left[F^{(2)}(\bar{x}^{t+1})\right] - \sum_{i,s} R(y_{i,s}^{t+1}) \quad (3)$$

where $\bar{x}^{t+1} := \bar{x}^{t+1}(\bar{y}^{t+1}, \lambda^{t+1})$. The optimization problem solved at stage 1 is described below.

`Input`: Predicted arrival rates distribution f_λ for time $t+1$
`Output`: Caching policy $\bar{y}^{t+1}$
`Compute` at time t:

$$\bar{y}^{t+1} = \underset{y^{t+1}}{\operatorname{argmin}} \quad F^{(1)}(y^{t+1}, \bar{x}^{t+1}(y^{t+1}, \lambda^{t+1})) \quad \text{(OPT1)}$$

subject to:

$$\sum_{i=1}^{N} y_{i,s}^t \, c_i^D \leq C_s^D, \quad \forall s \in \{1, \dots, S\} \quad (4)$$

$$|y_{i,s}^{t+1} - y_{i,s}^t| \leq \epsilon, \quad \forall i \in \{1, \dots, N\}, \quad \forall s \in \{1, \dots, S\} \quad (5)$$

$$y_{i,s}^{t+1} \in [0,1], \quad \forall i \in \{1, \dots, N\}, \forall s \in \{1, \dots, S\}$$

We highlight that at this stage one optimizes over both container caching y^{t+1} and activation $\bar{x}^{t+1}(y^{t+1}, \lambda^{t+1})$ for *each* possible scenario λ^{t+1}. However, *only* caching decisions $\bar{y}^{t+1}$ are deployed in the system. In fact, container activation for time $t+1$ is performed only once the true requests $\bar{\lambda}^{t+1}$ materialize at time $t+1$, since activation is almost instantaneous in practice. Hence, at stage 1 the activation variables x are only auxiliary, as they have the sole purpose of evaluating the quality of caching.

IV. Solving Stage 1: Derivative-free methods

As described above, the caching problem (OPT1) requires to solve (OPT2) for each possible demand rate scenario λ^{t+1}. This entails two major technical difficulties: *i)* (OPT2) is computationally hard (being formulated as an Integer Program) and ii) the objective function $F^{(2)}$ is not in closed-form. Regarding *i)*, we describe below a simple heuristic for (OPT2). To tackle *ii)* we resort to *derivative-free* optimization techniques, that only need to sample the value of the function, without needing to resort to their derivative. The general paradigm we employ is coordinate-descent [12], which at each iteration selects one coordinate $y_{i,s}$, keeps the others fixed, and optimizes function $F^{(1)}$ over $y_{i,s}$ via a univariate, derivative-free line search method such as Bayesian Optimization (BO) [13] or Golden Section Search (GSS) [14].

Algorithm 1: `Coordinate-Descent-Caching(CDC)`

Input: arrival distribution f_λ, $\epsilon > 0$
Output: Caching decisions $\bar{y}^{t+1}$

1 $\bar{y} \leftarrow \bar{y}^t$;
2 **for** $k = 1, \dots, K$ **do**
3 $\quad (i,s) \leftarrow coord\_select(N, S)$;
4 $\quad l \leftarrow \max\{y_{i,s}^t - \epsilon, 0\}$;
5 $\quad u \leftarrow \min\{y_{i,s}^t + \epsilon, \frac{C_s^S - \sum_{j \neq i} c_j^S \, y_{j,s}}{c_i^S}\}$;
6 $\quad y_{i,s} \leftarrow search(F^{(1)}(\bar{y}, \bar{x}^{t+1}(\bar{y}, \bar{\lambda^{t+1}})), f_\lambda, [l,u], \bar{\eta})$;
7 $\bar{y}^{t+1} \leftarrow \bar{y}$;
8 **return** $\bar{y}^{t+1}$

We dub this procedure Coordinate-Descent-Caching (CDC) and we describe it in Algorithm 1. There, K is number of iterations; *coord_select* and *search* are the procedures for the coordinate variable selection and the derivative-free line search method, respectively. Each time a new coordinate (i,s) is selected, a lower and an upper bound for $y_{i,s}$ are computed in lines 4 and 5 to confine the search, respectively. Specifically, the upper bound u for variable $y_{i,s}$ is the maximum between the value for the *cache churn rate* and the residual storage capacity of server s with all $y_{j,s}$s, $j \neq i$, fixed. This allows CDC to output a feasible solution.

Proposition 1: At each iteration of CDC it holds that $\bar{y}^{t+1} \in \mathcal{Y}^{t+1} := \{y^{t+1} \in [0,1]^{N \times S} | (4), (5) \text{ hold}\}$. Hence, the caching solution computed by CDC is feasible for (OPT1).

Heuristic for (OPT2). As mentioned, optimizing stage 1 requires to solve the container activation problem (OPT2) as a sub-routine, for each possible demand rate λ^{t+1}. This clearly calls for a heuristic approach to solve (OPT2) which is originally formulated as an Integer Program. We propose a greedy placement algorithm which selects, for each application i, an admissible set of fog servers where a container can be activated. Hence, the server with minimum memory and CPU occupation is chosen. Once this applications-servers mapping is obtained, if all the applications delay constraints are met then the mapping is considered as a valid activation. Otherwise, the applications violating constraint are moved to the cloud if their fog-to-cloud latency permits. We note that (OPT2) performs also load balancing by containers replication among the fog servers whereas our heuristic does not; this will be studied in future work.

Coordinate Selection and Search methods. The *coord_select* procedure can have several variants; we choose a method where coordinates are randomly permuted and selected sequentially. Other methods can be envisioned, *e.g.*, coordinate selection on the basis of the real arrival rates of the previous time-slot t. In this way, applications with highest arrival rates in the previous time-slot would be prioritized for the caching in the fog servers. We leave such variants for future works.

For the derivative-free line *search* procedure we evaluated Bayesian Optimization (BO) [13] and Golden Section Search (GSS) [14]. GSS is a classic dichotomy procedure that samples the function at two middle points of the current search interval and then restricts the interval. It returns the global optimum of

TABLE II
APPLICATIONS' CONTAINERS REQUIREMENTS [15].

Requirement	*Mean Value*	*Range*
CPU	1250 MIPS	$[500, 2000]$ MIPS
Memory	1.2 Gbytes	$[0.5, 2]$ Gbytes
Storage	3.5 Gbytes	$[1, 8]$ Gbytes

a univariate unimodal function, otherwise – which is our case – it returns a local optimum. The BO method is usually applied when the utility function is expensive to evaluate because it has high sample efficiency. Indeed, the term related to the caching computation ($x_{i,0}^{t+1}(\bar{y}^{t+1}, \bar{\lambda})$) is inherently computationally expensive to evaluate, even by using a heuristic activation as we do. By inferring the function at unknown points via a Gaussian Process, BO selects points having high probability of achieving low cost.

Computational Complexity. The computational complexity of CDC is mainly dominated by the computation of the placement function $\bar{x}^{t+1}(y^{t+1}, \lambda^{t+1})$, the size $|f_\lambda|$ of the support of f_λ (determining the number of possible demand rate scenarios λ^{t+1}), and the search method. The computational complexity of the greedy heuristic for container activation is $O(NS)$. Under sequential coordinate selection and with a fixed number of iterations K, the total complexity is $O(|f_\lambda| KNS \log(\tau^{-1}))$. The logarithmic factor appears due to the convergence rate of iterative methods such as the Golden Section Search method [14] where τ is a tolerance parameter. Hence, remarkably, the total complexity remains polynomial in the size of the input.

V. NUMERICAL RESULTS

In this section we evaluate our solution in a specific fog computing scenario where we test our joint caching and orchestration scheme. Three main goals are in order: (i) select the best method to perform the proactive caching optimization, i.e., select the most suitable coordinate descent algorithm using two candidate search methods, namely Golden Section Search (GSS) and Bayesian Optimization (BO); (ii) compare our approach with standard placement algorithms used to drive the activation step; (iii) demonstrate that our approach outperforms baseline schemes in terms of cloud deployment cost.

Simulation Settings. In our simulation experiments we consider one input batch of applications to be deployed on a system composed of one fog region with 3 servers and a cloud. Resources and applications requests per server are represented by triples of *CPU*, *memory* and *storage* units. Servers' capacity triples are $\mathbf{C_1} = (15000, 8, 50)$ and $\mathbf{C_2} = \mathbf{C_3} = (44000, 16, 60)$, where CPU is measured in MIPS and memory in Gbytes. The applications' requirements per container are listed in Table II. In our tests, we have assumed different sizes of the applications' batch, namely $N = 10, 15, 20, 25$. The demand rates and the maximum tolerable processing delay of each application are generated uniformly at random in $[0, 300]$ jobs/s and in $[5, 6]$ sec, respectively. Fog-to-cloud latency has been set to 500 ms, and the cloud processing delay is generated uniformly at random in $[1, 5]$ sec, for each application.

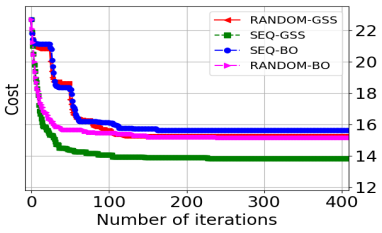

Fig. 2. Expected cost of GSS and BO derivative-free line-search methods for $N = 25$, averaged across 10 instances.

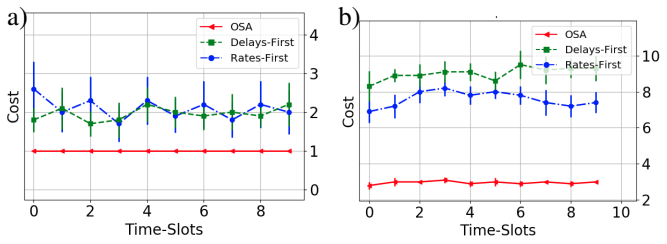

Fig. 3. a) Evaluation of placement methods for $N = 10$; b) Evaluation of placement methods for $N = 15$.

Search Methods. In stage 1, the orchestrator computes the caching policy by the coordinate descent Algorithm 1 (CDC) described in Section IV. As a sub-routine, CDC computes the minimum of the objective function $F^{(1)}$ on a line via a derivative-free method. We hence compared Golden Section Search (GSS) and Bayesian Optimization (BO) on Gaussian Processes. Figure 2 shows the expected cost of deployment achieved by the two methods for $N = 25$. Each point is the average of ten instances where the infrastructure is fixed and the application arrival rates' distribution changes. We implemented two variants for the coordinate selection method. The *sequential* (SEQ) approach applies each method to each coordinate of the caching matrix $\bar{y}^t$ in sequential manner. The *random* (RANDOM) approach selects a random coordinate at each iteration. The figure highlights the capability of GSS to obtain a better value of the expected cost with respect to the BO method. BO is sensitive to input hyper-parameters, and it is generally no easy task to select the most suitable ones.

Given its better performance, in the next experiments we adopt GSS as our preferred search algorithm as a sub-routine in CDC, to solve stage 1.

Container activation algorithms for Stage 2. We now discuss how to solve the container activation problem (OPT2) in stage 2. Due to the lack of suitable benchmarks in the literature, we devised two reasonable baseline heuristics for the joint caching and orchestration problem. We call *Delays-first* and *Rates-first* our baseline activation algorithms: they give priority to applications with the lowest delay constraints and highest demand rates, respectively. These two heuristics apply both to the activation function $x_{i,0}^{t+1}(\bar{y}^{t+1}, \bar{\lambda})$ used in the first stage of our approach, i.e, to cache containers, and to the activation phase of the second stage; yet, the baseline heuristics do not perform proactive container replication. The caching policies are standard proactive edge caching ones as

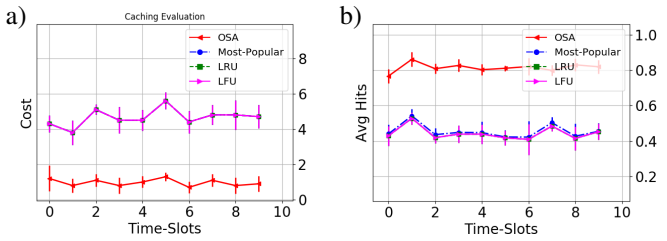

Fig. 4. a) Cost incurred by each caching policy; b) Average number of hits per fog server.

described later on.

Figure 3 shows the overall performance comparison among the proposed methods, averaged over 10 instances along with their 95% confidence interval. We set $\epsilon = 0.6$ for the *cache churn rate* constraint, meaning that at most 60% of cached containers can be changed at each step. We can observe from the figure that heuristics are far from the optimal one, which in turn is attained by our approach (OSA). In fact, prioritizing containers' activation on the basis of their delay constraints would lead to deploying applications with strict constraints in fog; this will cause an increase of the delay experienced per application due to CPU sharing on fog servers. Lack of replication of same containers on different servers increases the delay as well. Thus, all the remaining applications will be deployed in cloud, hence incurring a larger cost. A similar argument applies to the *Rates-first* heuristics since in that case applications with higher demand rates will be deployed together on the fog servers; but this is possible as long as their delay constraints are satisfied, while other applications with lower arrival rates must be deployed in cloud.

Caching. The performance gain of our one-step-ahead (OSA) approach over the two heuristics can be ascribed to its efficient caching strategy. Hence, we now highlight the key differences between standard proactive edge caching and our optimized proactive fog caching. As mentioned, classical caching strategies are used by the two heuristics: (i) the *Most-Popular (MP)* which, at each time-slot t, for each server, prioritizes applications with highest popularity, *i.e.*, applications presenting the highest arrival rate at the previous time-slot t; (ii) *Least Recently Used* (LRU) strategy, evicting the least recently requested containers on each server; (iii) *Least Frequently Used* (LFU) strategy which discards the least frequently requested container since the first time-slot. MP, LRU and LFU track demand rates and epochs by updating a table of file requests.

We imposed the *cache churn rate* constraint, with $\epsilon = 0.3$, for all caching strategies. Also, each file is cached at the minimum level required for its activation. LRU, LFU and MP refer to the plain memory occupation as the reference metric to evaluate cache space on fog servers. Figure 4 reports on the performance of each fog caching policy for $N = 20$. Data points represent an average over ten instances. For each instance a distribution over the applications' demand rates is generated and, at each time-slot, a vector of arrival rates is chosen with probability defined by the initial distribution. In these experiments all the vectors of demand rates are sampled with uniform probability. In Figure 4a) the comparison in terms of deployment cost is showed. All classical caching policies have same cost due to the low percentage of discarded files during the sampled period, whilst approach (OSA) performs significant savings. Furthermore, Figure 4b) shows the average number of hits per fog server. The highest rate is achieved by OSA with respect to baseline approaches, as expected. These results prove that in a fog environment joint caching and orchestration of applications is key in order to reduce expensive costs of deployment in cloud.

VI. Conclusions

In this paper we have developed a framework to perform the orchestration of fog applications and minimize their deployment cost. We have proved that proactive caching greatly improves the performance of fog orchestration by matching in advance the expected demand rates of applications and the available resources on fog servers. Actually, fundamental differences exist between fog caching and edge caching due to the multidimensionality of resources per deployed container, the impact of fog-to-cloud delay and the effect of resource sharing on fog servers. Our scheme performs a two-stage stochastic optimization by forecasting the impact of containers' activation onto servers capacity and applications computing delays. Several novel aspects deserve further study to this respect, e.g., exploring new lightweight heuristic solutions to approximate the behaviour of the optimal one.

References

[1] M. Chiang and T. Zhang, "Fog and IoT: an overview of research opportunities," *IEEE IoT Journal*, vol. 3, no. 6, pp. 854–864, Dec 2016.

[2] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of things: Vision, applications and research challenges," *Ad Hoc Networks*, vol. 10, no. 7, pp. 1497 – 1516, 2012.

[3] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the Internet of things," in *Proc. of ACM MCC*, Helsinki, Finland, August 13–17 2012.

[4] G. Li, J. Wu, J. Li, K. Wang, and T. Ye, "Service popularity-based smart resources partitioning for fog computing-enabled industrial Internet of Things," *IEEE Trans. on Industrial Informatics*, vol. 14, no. 10, pp. 1–1, Oct. 2018.

[5] Kubernetes. http://kubernetes.io/.

[6] R.-S. Schmoll, T. Fischer, H. Salah, and F. H. Fitzek, "Comparing and evaluating application-specific boot times of virtualized instances," in *2019 IEEE 2nd 5G World Forum (5GWF)*. IEEE, 2019, pp. 602–606.

[7] E. Bastug, M. Bennis, and M. Debbah, "Living on the edge: The role of proactive caching in 5G wireless networks," *IEEE Communications Magazine*, vol. 52, no. 8, pp. 82–89, Aug 2014.

[8] X. Gao, X. Huang, Y. Tang, Z. Shao, and Y. Yang, "Proactive cache placement with bandit learning in fog-assisted IoT systems," in *Proc. of IEEE ICC*, 2020, pp. 1–6.

[9] S. Li, J. Xu, M. van der Schaar, and W. Li, "Popularity-driven content caching," in *Proc. of IEEE INFOCOM*, 04 2016, pp. 1–9.

[10] F. Metzger, T. Hossfeld, A. Bauer, S. Kounev, and P. Heegaard, "Modeling of aggregated IoT traffic and its application to an IoT cloud," *Proceedings of the IEEE*, vol. 107, pp. 679 – 694, 03 2019.

[11] J. R. Birge and F. Louveaux, *Introduction to stochastic programming*. Springer Science & Business Media, 2011.

[12] J. M. Ortega and W. C. Rheinboldt, *Iterative solution of nonlinear equations in several variables*. SIAM, 2000.

[13] C. K. Williams and C. E. Rasmussen, *Gaussian processes for machine learning*. MIT press Cambridge, MA, 2006, vol. 2, no. 3.

[14] J. Kiefer, "Sequential minimax search for a maximum," *Proceedings of the American mathematical society*, vol. 4, no. 3, pp. 502–506, 1953.

[15] A. Brogi, S. Forti, and A. Ibrahim, "How to best deploy your fog applications, probably," in *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*. IEEE, 2017, pp. 105–114.