This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3052139, IEEE Transactions on Services Computing

Resource Provisioning and Allocation in Function-as-a-Service Edge-Clouds

Onur Ascigil*, Argyrios G. Tasiopoulos*, Truong Khoa Phan*, Vasilis Sourlas†, Ioannis Psaras*, and George Pavlou*
*Department of Electronic and Electrical Engineering, University College London, UK.
†Institute of Communication and Computer Systems (ICCS-NTUA), Athens, Greece.
Email: {o.ascigil, argyrios.tasiopoulos, t.phan, i.psaras, g.pavlou}@ucl.ac.uk, v.sourlas@iccs.gr

***Abstract*—Edge computing has emerged as a new paradigm to bring cloud applications closer to users for increased performance. Unlike back-end cloud systems which consolidate their resources in a centralized data center location with virtually unlimited capacity, edge-clouds comprise distributed resources at various "computation spots", each with very limited capacity. In this paper, we consider Function-as-a-Service (FaaS) edge-clouds where *application providers* deploy their latency-critical functions that process user requests with strict response time deadlines. In this setting, we investigate the problem of *resource provisioning* and *allocation*. After formulating the optimal solution, we propose resource allocation and provisioning algorithms across the spectrum of fully-centralized to fully-decentralized. We evaluate the performance of these algorithms in terms of their ability to utilize CPU resources and meet request deadlines under various system parameters. Our results indicate that practical decentralized strategies, which require no coordination among computation spots, achieve performance that is close to the optimal fully-centralized strategy with coordination overheads.**

I. Introduction

Cloud computing, which is characterized by abundant resources deployed at centralized infrastructures, is increasingly being challenged by new and emerging "edge-applications" that have stringent Quality-of-Service (QoS) requirements. Examples of edge-applications include data analytics [1], virtual/augmented reality [2], interactive gaming, connected and automated driving and wearable cognitive assistance [3], to name a few. For such applications, the "cloud-to-user" application service delivery from distant, centralized infrastructures fails to meet the QoS requirements [4], [5].

Consequently, *edge computing* proposes building distributed *edge-cloud* infrastructures at the "edges" of the network, *i.e.*, close to end-users, for edge-application providers to deploy their services [6] and meet their QoS requirements. An edge-cloud comprises distributed *computation spots* (*e.g.*, cloudlets [7]), each with *very limited resource capacity* compared to core cloud infrastructures [8], [9].

In this paper, we aim to tackle two challenging aspects of edge-cloud management, namely *i)* allocation of resources for end-user requests (along with their routing and scheduling), and *ii)* provisioning of edge-application services. We consider a multi-tenant, Function-as-a-Service (FaaS) edge-cloud infrastructure where computation spots are hierarchically distributed along the paths to the back-end clouds as shown in Fig. 1 [10], [11]. In this setting, edge-application providers make their containerized services (*i.e.*, functions) available for provisioning in computation spots, while end-users submit *time-sensitive* computation requests for functions with *strict*

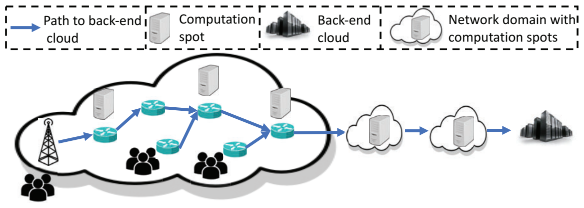


Fig. 1: On-path computation spots in an edge-cloud system.

deadlines on response latencies; that is, the delay between the end-user submitting a computation request to the network and getting the outcome of the computation delivered back to the end-user.

In traditional cloud resource management approaches, resource provisioning (*i.e.*, reserving FaaS containers) is performed by the tenants. However, the limited capacity and distributed nature of computation spots coupled with the volatile nature of end-user demands at the network edges make such an approach unpractical in an edge-cloud setting—that is, one where the edge-application providers (as tenants) manage the provisioning (*i.e.*, reserving) of resources to deal with dynamic end-user demands at individual computation spots and billed for the resources *reserved* (not actually used). Instead, we consider *Serverless* computing approach [12], where the edge-cloud infrastructure takes care of resource provisioning and allocation on behalf of edge-application providers and in turn bill the edge-application providers for the *actual* resource usage, and not for the resources reserved.

The resource management tasks in FaaS, Serverless edge-clouds include carefully deciding the number of function instances to provision for each edge-application and selection of requests to schedule at each computation spot. The main objectives in carrying out these tasks are: keeping the resources of computation spots utilized and meeting the response deadlines (*i.e.*, providing sufficient QoS) of end-user requests.

However, achieving the above-mentioned management objectives is more challenging in a Serverless edge-cloud than in a Serverless centralised cloud computing scenario for the following reasons: *i)* provisioning of functions and scheduling of end-user requests must take into account both response latency requirements of requested functions and proximity of computation spots to end-users, because individual computation spots can provide different lower-bounds on response latencies, *ii)* the overhead of "cold-starting" containers [13], [12]—*i.e.*, performing the necessary processing to bring a container to a running state and loading a function—is an overhead that can quickly become an obstacle to achieving low latency performance in edge-clouds with under-provisioned resources, and *iii)* a fully-centralized management of distributed computation spots is not practical due to the associated communication and coordination overheads.

Our main technical contributions are as follows:

1) We study the general problem of resource provisioning and allocation over a hierarchical FaaS edge computing infrastructure for edge-application functions with strict latency requirements and propose practical approaches with acceptable coordination and communication overheads.
2) We formulate the optimal function provisioning and resource allocation problem given the existence of request queuing and container provisioning (*i.e.*, cold-start) overheads. Since fully-centralized, optimal models require accurate prediction of upcoming requests, which is difficult to achieve in practice, we propose heuristic algorithms ranging from centralized to completely decentralized.
3) We evaluate the performance of the proposed approaches against a hypothetical fully-centralized one which has the global view of the entire edge-cloud system. Our results demonstrate that practical strategies, while having little or no coordination or communication overhead, can achieve a comparable performance with the fully-centralized one.

The rest of this paper is organized as follows. In Section II, we discuss related work. In Section III we introduce the functional components of FaaS edge-clouds. Then, we introduce centralized resource allocation strategies in Section IV followed by a discussion of fully-decentralized resource allocation strategies in Section V for hierarchical edge-clouds. We present our performance evaluation in Section VI, and we conclude the paper in Section VII.

II. Related work

Existing research has considered fully-centralized control of resource provisioning and allocation [14], [10], [15], [16], [17]. However, such solutions require significant communication and coordination overhead in a distributed edge-computing environment. Instead, we consider more practical approaches that require minimum coordination.

Our starting point is the efforts related to cache networks that temporarily store content [18], [19], [20]. Cache networks are characterized by storage resources at multiple points of presence, each with very limited capacity. From the distributed caching studies, we borrow the concept of *opportunistic request routing*: requests for computation are *opportunistically* processed in a hierarchy of computation spots along the path from users to a default back-end cloud where requests are guaranteed to be processed [10]. Opportunistic routing mechanism allows us to treat the request routing problem (*i.e.*, where to process a request) as an admission control problem, where individual spots execute admission policies independently on incoming requests to decide whether to process locally or send (*i.e.*, offload) upstream to the next spot in the hierarchy.

Several studies have similarly proposed the deployment of hierarchical edge-cloud deployment to form a continuum of resources from edges to the core of the Internet. Tong et al. [10] has proposed a hierarchy of edge-clouds as a way to handle peak user demand effectively. A similar hierarchical design with admission control on incoming requests has been proposed for radio access networks with only two levels of hierarchy consisting of cloudlets attached to the base stations and a back-end cloud [21], [22]. However, we consider a general hierarchy of edge-clouds deployed at edges and middle-tier networks, similar to a hierarchy of caches.

Existing work [23], [24] has considered caching of function code (image) at individual edge-cloud nodes, each with limited storage resources. These studies then considered caching strategies which determine when and which functions' codes to download from a centralized image repository (in the back-end clouds) in case of cache misses, *i.e.*, arrival of results for functions whose codes are not currently stored locally. The goal of these strategies is to efficiently use the limited secondary memory resources to store the most popular function codes and minimize the *bandwidth resources* consumed to download functions and assume immediate, on-demand provisioning of functions upon caching without any provisioning overheads. Instead, we consider *computing and main memory resources* as the limited resources as opposed to (cheaper) secondary memory resources. To that end, we focus on efficient (periodic) provisioning of containers in the main memory with acceptable cold-start costs and allocation of CPU resources to function containers to achieve highest possible QoS for end-users.

In a preliminary version of this work [25], we considered (fully-decentralised) caching approaches to determine which of locally stored functions to provision at each cloudlet. The cache replacement policies such as LFU and LRU are executed periodically rather than per-request, in addition to deadline-based policies such as strictest-deadline-first (SDF) which prioritizes functions with smaller remaining deadlines to execution. Here we extend this work with more sophisticated strategies with varying degrees of centralization. In addition, we introduce both in the model and simulations the queuing time of requests and the cold-start processing overhead of provisioned containers, during which a container is unavailable for requests and CPU resources are used for to bring the container to a running state.

Several studies [26], [17] have considered centralized approaches to solve the problem of optimal function provisioning (placement), request routing and scheduling in edge-clouds. These approaches require frequent coordination between a central controlling entity and the edge-clouds to obtain up-to-date information such as demand for each function, resource usage at each computation spot, and so on. Such approaches are mainly useful to derive theoretical bounds on performance, although they can be practical in small-scale, regional scenarios [17]. Auctions are another example of a centralized resource allocation approach which involves deriving prices for resource usage at each computation spot to control the flow of requests to individual edge-cloud nodes [27], [11]. That said, we also formulate a fully-centralized solution for function placement and request scheduling (Section IV-A); however, we also consider a more practical centralized approach where only periodic function provisioning is performed in a centralized manner and request routing is carried out in an opportunistic (*i.e.*, decentralized) way (Section IV-B).

In this work, we approach the resource allocation and provisioning problem for edge-clouds with a systems-oriented mindset: we formulate the resource allocation as an admission

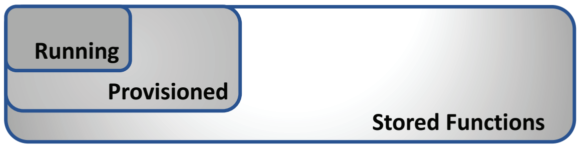


Fig. 2: Functions: stored, provisioned (warm), and running.

control design problem combined with both selection and scheduling of a limited number of provisioned functions on severely limited CPU resources. We explicitly incorporate system-level mechanisms in our model and simulations such as queuing of admitted requests which is important to cope with small bursts in demand in the absence of auto-scaling capabilities. Also, we explicitly take into account the varying levels of QoS requirements by different application services in the form of deadlines on response delays rather than simply minimizing treating different application requests equally as simply done by the existing work.

III. Edge Cloud System Components

We consider hierarchically organized on-path computation spots (*e.g.*, cloudlets), which process requests for containerized functions originating from user applications. The latter indicate in their requests a function to be executed, input data to the function possibly containing computation state (*e.g.*, obtained as a result of a previous request), and a deadline for completion. The FaaS edge-cloud system then performs *periodic provisioning* of container resources and allocates an *active pool* of containers; each one is a pre-provisioned function instance which is ready to process incoming requests.

In a FaaS edge-cloud system, the periodic provisioning decision aims to adapt the amount of resources allocated for each function at each computation spot (in terms of number of warm containers in the active pool) according to changes in the state of the edge-cloud system, *e.g.*, changes in the demand for functions, success of offloaded requests in meeting their deadlines, and so on. This periodic process, essentially re-distributes the available containers in the active pool, whose size is limited by the main memory capacity of each computation spot, to individual application functions. From the active pool of functions, only a subset of them can process requests simultaneously, subject to the number of CPU cores in the spot, as demonstrated in Fig. 2.

In addition to the periodic provisioning which happens at timescales of seconds to minutes, the computation spots perform per-request, opportunistic resource allocation through an admission control system which works together with a scheduling component. Below, we elaborate on these system components that are key to the resource provisioning and allocation functionality at each spot.

A. Admission

An *admission control* system at each computation spot selectively admits a subset of the incoming requests and places those requests in the request queue where they wait for their turn to be processed by a function instance. In this work, we consider processing of user requests with strict deadlines. This means that the utility of users from receiving the response to a request drops sharply to zero once the response time exceeds the deadline. Therefore, the computation spots do not waste resources processing requests whose deadlines cannot be met, and such requests are simply rejected. Rejected requests are offloaded to the next computation spot along the path to the back-end cloud, whose queues and CPU cores might be less utilized and the request deadline can be met. On the other hand, admitted requests are guaranteed to meet their deadlines.

User applications record in their requests a fixed timestamp value as a *remaining deadline*. The remaining deadline is initially set by user applications in their requests as an upper-bound on the expected response time from the edge-cloud, including network and processing delays. Each time a request is outsourced to an upstream computation spot, its remaining deadline is updated by deducting the *estimated RTT* to reach the upstream computation spot from the existing remaining deadline value. In order to have an accurate estimate of the RTT, the client applications and the computation spots periodically send probe packets to their (immediate) upstream computation spots, which then respond with a probe response packet. As a result, the remaining deadlines of requests observed by computation spots indicate the residual amount of time that the requests can afford spending for queuing and processing. We assume that admission decision processing has a negligible delay within a computation spot and is not reflected in the remaining deadline updates. On the other hand, once a request is admitted and queued, its remaining deadline is updated with the passage of time.

A computation spot immediately rejects an incoming request for a function, if there are currently no (warm) instances of that function in the active pool. This is because of our design choice that enforces periodic provisioning decisions for computation spots as opposed to on-demand provisioning of warm function instances as explained in Section III-C. On the other hand, a request R is considered for admission if there are instances of functions that R is requesting. This process involves checking if R's remaining deadline can be met. To that end, the admission control system computes a new *time-to-completion* value for all the requests currently in the queue, assuming R is enqueued and scheduled, taking into account the request scheduling policy (Section III-B); time-to-completion of a request is the sum of waiting time in the queue and processing time by the function, and we assume processing times of functions are fixed and known locally at each computation spot. If the new time-to-completion of all the requests do not exceed their remaining deadlines, then R is accepted; otherwise, R is rejected. A special case is where R's remaining deadline is smaller than its projected processing time, in which case R is immediately rejected. We discuss the details of request scheduling which impacts the computation of time-to-completion below.

B. Scheduling

A *scheduling policy* determines the order of processing requests in the request queue. In this work, we consider the *Earliest Deadline First (EDF)* scheduling which we have shown to out-perform First-In-First-Out (FIFO) in our earlier work [25]. EDF prioritizes requests with smallest remaining (*i.e.*, earliest) deadlines.

We assume non-preemptive scheduling where each CPU core is dedicated to the execution of a function scheduled on

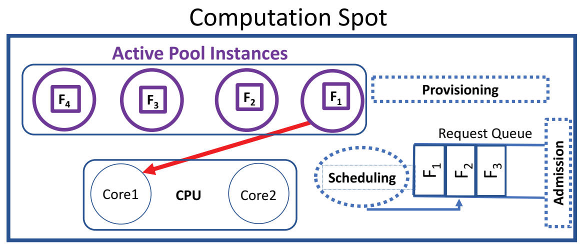


Fig. 3: A computation spot with one warm instance of functions F_1, F_2, F_3, and F_4. F_1's instance is running on Core1 (shown with a red arrow), therefore the next queued request to schedule on core2 is for F_2, skipping the request for F_1.

that core without interruptions (*i.e.*, no context switches) until completion of a request, and each function instance processes exactly one request at a time. Given that (i) function processing times are accurately estimated (*e.g.*, though profiling [28], [29]) with non-preemptive scheduling, (ii) only active pool instances are scheduled for requests, and (iii) the remaining deadline of requests are known, the only complicating factor in the estimation of time-to-completion of arriving requests under EDF policy is the availability of function instances.

A function instance is unavailable (*i.e.*, busy) while processing a request, and if there are no other available instances of that function, the scheduler is unable to schedule and run another request for the same function until an instance becomes available. An example is shown in Fig. 3 where the request for F_1 at the head of the queue is skipped and another request (for F_2) is scheduled, because the only instance of F_1 is currently busy. The request for F_1 stays at the head of the queue and will eventually get scheduled when the instance of F_1 running in core-1 completes its current request. This way, EDF conserves work and keeps cores busy as long as there are requests waiting in the queue that can be scheduled right away, despite slightly violating the earliest-deadline-first ordering.

Given the work-conserving EDF scheduling policy, the admission controller's decision on an incoming request R involves simply simulating the scheduling of all existing requests with the inclusion of R to compute a time-to-completion. For a request to meet its deadline, it must be scheduled by the time its remaining deadline is equal to its estimated processing time, which we refer to as the *critical deadline* below. Given a scheduling policy and the current load on a computation spot (waiting and currently running function requests), an arriving request R for computing a function F with a critical deadline D (where $D \geq$ current time) is rejected (*i.e.*, offloaded upstream) under one of the two conditions below:

1) *Insufficient or non-existing F instances, while CPU resources are available:* The computation spot either does not currently have an instance of F, or it has insufficient instance(s) of F in its active pool to handle R and all other requests for F that have arrived very close in time.
2) *Insufficient CPU resources:* The computation spot has at least one instance of F which will be available before D, but the CPU will be allocated to other requests (with earlier deadlines) until D.

The first scenario is a signal that function provisioning, which we discuss next, should reevaluate the provisioned function instances to better utilize the CPU resources. Although, the second scenario does not signal a problem, but requires further analysis as to whether the instances of the "right" functions are provisioned—*e.g.*, ones whose requests can not meet their deadlines, if offloaded upstream—by a provisioning policy, as we discuss next.

C. Function Provisioning

The goal of provisioning is to keep CPU cores utilized which directly translates to revenue for edge-cloud providers under the Serverless model. With under-provisioned resources, a computation spot aims to avoid rejecting requests with feasible deadlines while cores are idle. In this case, the main reason for rejecting requests is the unavailability (or lack) of warm instances of a requested function in the active pool (AP) ready to process requests. The launching of a new warm instance (*i.e.*, cold-start) is an overhead, which can be significant when considering requests with possibly smaller deadlines.

Therefore, we make the design choice of performing only periodic provisioning decisions at fixed intervals called *replacement intervals*, rather than on-demand (*e.g.*, per-request) provisioning. *A provisioning policy* decides how many instances to launch for each application function as part of the active pool of functions. Provisioning decisions are based on the ranking of all the existing stored function types (*i.e.*, set Stored Functions in Fig. 2). Next, we introduce several resource allocation and provisioning strategies for different degrees of centrality, *i.e.*, varying levels of required coordination among the computation spots of an edge-cloud.

IV. Centralized Edge-Clouds

In the centralized edge-clouds, a centralized *controller* is involved to some degree in the resource allocation and provisioning tasks. We consider two centralized approaches that are defined according to the extent in which the controller is involved in those tasks: *i)* a *fully-centralized* approach where the controller performs per-request admission and provisioning decisions, and *ii)* a *coordinated* approach where the controller is involved only in the periodic function provisioning at the edge-cloud, while request admission decisions are opportunistically performed individually by each computation spot. We formulate below the fully-centralized provisioning problem and then present the coordinated provisioning algorithm that we evaluate in Section VI.

A. Fully-Centralized Model & Problem Formulation

We consider a system where time is slotted and indexed by $t \in \mathcal{T}$ for $\mathcal{T} \triangleq \{1, 2, ..., T\}$. We assume the existence of $\mathcal{F} \triangleq \{1, 2, ..., F\}$ stored functions which are requested by a set of $\mathcal{G} \triangleq \{1, 2, ..., G\}$ groups of users, where a group is defined based on its users' access point, *i.e.*, all the users with the same network attachment point (*e.g.*, access router or gateway) belong to the same group. We denote the number of requests of group $g \in \mathcal{G}$ for function $f \in \mathcal{F}$ at time-slot t as $D_g^f(t)$ and we associate each request for function f with a deadline, T_f, expressed in time-slots. A user request for function f is considered to be satisfied only if the request is

$\mathcal{T}$	Slotted time-slots set.
$\mathcal{F}$	Set of Functions.
$\mathcal{G}$	Set of user groups defined by their point of attachment to the network.
$D_g^f(t)$	Demand of group g for function f at time-slot t.
T_f	Propagation deadline for function's f request.
$\mathcal{H}$	Set of cloudlets.
L_{gh}	RTT from user group $g \in \mathcal{G}$ to cloudlet $h \in \mathcal{H}$.
$\mathcal{V}_h$	Set of containers at cloudlet $h \in \mathcal{H}$.
C_h	Number of cores at cloudlet $h \in \mathcal{H}$.
$\tilde{t}$	Time-slot overhead of launching a function instance.
z_{fv}^t	Binary decision variable indicating the beginning of instantiation of function f as container v at time-slot t.
$\tilde{C}_h^t$	Cloudlet h's cores occupied loading an image at t.
$\mathcal{V}_h^{f,t}$	Set of function f instances at cloudlet $h \in \mathcal{H}$ during time-slot t.
u_v^t	Binary variable indicating the association of container v to a core at time-slot t.
$C_h^{f,t}$	Number of warm instances of function f at cloudlet h upon time-slot t.
$\mathcal{K}$	Set of queuing classes at each cloudlet.
$Q_h^{k,f}(t)$	Number of requests for function f at cloudlet h queued at class k during time-slot t.
$x_h^{k,f}(t)$	Total requests of function $f \in \mathcal{F}$ arrived at queue class $k \in \mathcal{K}$ of cloudlet h upon time-slot t.
$y_h^{k,f}(t)$	Total requests of function $f \in \mathcal{F}$ served from the queue class $k \in \mathcal{K}$ of cloudlet h upon time-slot t.
$\tilde{x}_{gh}^f(t)$	Group g's forwarding requests of f to cloudlet h at t.
$\tilde{x}_g(t)$	Group g's function forwarding request matrix at time-slot t.
ΔT_{gh}	Latency forwarding overhead, in time-slots, from group g to cloudlet h.
k_{gh}^f	Arrival queue class of function f's requests from group g at cloudlet h.
$\tilde{x}_h^{k,f}(t)$	Requests of function $f \in \mathcal{F}$ traversing cloudlet h which would arrive at queue class $k \in \mathcal{K}$ upon time-slot t.
$\rho_v^f(m)$	Requests served on average by an instance of function f running on container v when it is associated to a core every m time-slots.

TABLE I: Model Notation

served and the results arrive back at the user within T_f time-slots. We assume for simplicity that the processing time of requests for all the functions is fixed to a single time-slot.

We denote by $\mathcal{H} \triangleq \{1, 2, ..., H\}$ the set of computation spots (*i.e.*, cloudlets) equipped with hardware capable of hosting instances of any function $f \in \mathcal{F}$. Each cloudlet $h \in \mathcal{H}$ can maintain in its active pool a set of $\mathcal{V}_h \triangleq \{1, 2, ..., V_h\}$ containers each of which is a particular function's instance. However, in order for a function instance to serve a request, it must be first scheduled at a CPU core whose size is limited to $C_h \leq |\mathcal{V}_h|$ at each cloudlet h. In other words, C_h is the maximum number of requests that can be served simultaneously at h within a single time-slot. Lastly, given the position of a cloudlet $h \in \mathcal{H}$, the users of a group $g \in \mathcal{G}$ can access h at a Round Trip Time (RTT) of L_{gh} time-slots. The notations used in this section are shown in Table I.

Cloudlet Management Model: Launching a function instance can be considered as a task occupying a core for a period of $\tilde{t}$ time-slots, which we consider, for simplicity, identical for every function image $f \in \mathcal{F}$.

We denote by z_{fv}^t the binary decision variable of triggering the launch of an image of function f as a container $v \in \mathcal{V}_h$ at time-slot t, *i.e.*, $z_{fv}^t = 1$ when container v is to run f or $z_{fv}^t = 0$ otherwise. Then the occupied cores upon time-slot t in loading a function image at cloudlet h, $\tilde{C}_h^t$, is:

$$\tilde{C}_h^t = \sum_{f \in \mathcal{F}} \sum_{v \in \mathcal{V}_h} \sum_{i=t-\tilde{t}}^{t} z_{fv}^i, \ \forall h \in \mathcal{H}, \forall t \in \mathcal{T}. \tag{1}$$

At the same time, each container in the active pool can launch at most one function at a time which is expressed as:

$$\sum_{f \in \mathcal{F}} \sum_{i=t-\tilde{t}}^{t} z_{fv}^i \leq 1, \ v \in \mathcal{V}_h, h \in \mathcal{H}, \forall t \in \mathcal{T}. \tag{2}$$

Therefore, the active pool instances of function f at cloudlet h during time-slot t define the set $\mathcal{V}_h^{f,t} \subseteq \mathcal{V}_h$ where $\mathcal{V}_h^{f,t}$ evolves over time according to:

$$\begin{aligned} \mathcal{V}_h^{f,t} = \mathcal{V}_h^{f,t-1} \bigcup \{v \in \mathcal{V}_h : z_{fv}^{t-\tilde{t}} = 1\} \Big\backslash \\ \{v \in \mathcal{V}_h^{f,t-1} : z_{f'v}^t = 1 : f' \neq f \in \mathcal{F}\}, \\ \forall f \in \mathcal{F}, \forall h \in \mathcal{H}, \forall t \in \mathcal{T}. \end{aligned}$$

In other words, the set of instances of function f at cloudlet h during time-slot t depends on how many new instances have been launched for f as well as the number of containers that have been reassigned to functions other than f.

A cloudlet processes a request for f by scheduling an available active pool instance for f on an idle core, *i.e.*, performing a container-to-core association, at which point the container is in a *running* state. We denote by u_v^t the binary variable indicating the association of container v to a core at time-slot t. Then, the number of running instances of function f at cloudlet h at time-slot t is defined by:

$$\sum_{v \in \mathcal{V}_h^{f,t}} u_v^t = C_h^{f,t}, \ \forall h \in \mathcal{H}, \forall f \in \mathcal{F}, \forall t \in \mathcal{T}. \tag{3}$$

Clearly, the number of running instances is limited by the total number of cores at h as well as the cores occupied in loading an image, captured by the constrain:

$$\tilde{C}_h^t + \sum_{f \in \mathcal{F}} C_h^{f,t} \leq C_h, \ \forall h \in \mathcal{H}, \forall t \in \mathcal{T}. \tag{4}$$

Queuing Model: We define a set of $\mathcal{K} \triangleq \{1, 2, ..., K\}$ *queuing classes* at each cloudlet where the arriving requests are placed according to their *adequate time-slots*; that is, the number of remaining time-slots before critical deadline. For instance, if a request's adequate time-slot is one, then the request enters the queue class $k = 1$. The cloudlets perform EDF scheduling and serve requests from the lowest queue class and proceed with the higher ones, *i.e.*, $k = 1, 2, 3, ...$, until reaching their processing capacity, *i.e.*, all the cores are busy.

Given a cloudlet h at time-slot t, we denote by $Q_h^{k,f}(t)$ the number of requests for function $f \in \mathcal{F}$ at queue class $k \in \mathcal{K}$. Then, the queue length evolves as follows:

$$\begin{aligned} Q_h^{k,f}(t+1) = [Q_h^{k+1,f}(t) + x_h^{k+1,f}(t) - y_h^{k+1,f}(t)]^+, \\ \forall f \in \mathcal{F}, \forall h \in \mathcal{H}, \forall k \in \mathcal{K}, \forall t \in \mathcal{T} \end{aligned} \tag{5}$$

where $x_h^{k,f}(t)$ and $y_h^{k,f}(t)$ is the total number of requests for function f arrived to and served from class k, respectively, of cloudlet h during time-slot t.[1] In other words:

$$y_h^{k,f}(t) \leq Q_h^{k,f}(t) + x_h^{k,f}(t), \ \forall f \in \mathcal{F}, \forall h \in \mathcal{H}, \forall t \in \mathcal{T}. \tag{6}$$

Note that the requests, which belong to class $k+1$ and not served at time-slot t, are forwarded to queue class k at time-slot $t+1$ since their adequate time-slots is reduced by one.

[1] $[\cdot]^+ = \max[\cdot, 0]$.

Scheduling Model: Consider the case where a group $g \in \mathcal{G}$ is interested in forwarding $x_{gh}^{f}(t)$ requests of function $f \in \mathcal{F}$ to cloudlet h at time-slot t in order to satisfy its corresponding request demand:

$$\sum_{h \in \mathcal{H}} x_{gh}^{f}(t) \le D_{g}^{f}(t),\ \forall g \in \mathcal{G}, \forall f \in \mathcal{F}, \forall t \in \mathcal{T}. \quad (7)$$

We define the *function forwarding request matrix* at time-slot t for group g and all services as $x_g(t) \triangleq (x_{gh}^{f}(t) : \forall f \in \mathcal{F}, \forall h \in \mathcal{H})$. Specifically, we denote the time-slot based overhead for group g of forwarding a request to cloudlet h as $\Delta T_{gh} = \lceil \frac{L_{gh}}{2} \rceil$. That is, cloudlet h will receive a request of function f sent at time-slot t from group g upon time-slot $t + \Delta T_{gh}$ at queue class $k_{gh}^{f} = T_f - 2\Delta T_{gh} - 1$ corresponding to the request's adequate time-slots. For example, if $T_f = 5$ time-slots and $\Delta T_{gh} = 1$, then each request when forwarded at h can spend up to $k = 2$ time-slots in the queue.

Therefore, the requests for function f arrived at cloudlet h's queue class k upon time-slot t are estimated by:

$$x_{h}^{k,f}(t) = \sum_{g \in \mathcal{G}: k = k_{gh}^{f}} x_{gh}^{f}(t - \Delta T_{gh}),$$
$$\forall f \in \mathcal{F}, \forall h \in \mathcal{H}, \forall k \in \mathcal{K}, \forall t \in \mathcal{T}. \quad (8)$$

Request Admission Model: In our setting, the admission of a request by a cloudlet is associated with a guarantee of getting served within its remaining deadline. In other words, the number of requests served from queue class $k = 1$ has to always satisfy the recently arrived requests as well as the existing requests in the queue:

$$Q_{h}^{1,f}(t) + x_{h}^{1,f}(t) \le y_{h}^{1,f}(t)\ \forall f \in \mathcal{F}, \forall h \in \mathcal{H}, \forall t \in \mathcal{T}. \quad (9)$$

Moreover, the served requests of queue class k have to respect the queue class priority as well as the number of active pool instances of function f in h and time t, *i.e.*, $C_{h}^{f,t}$:

$$y_{h}^{k,f}(t) \le C_{h}^{f,t} - \sum_{k'=1}^{k-1} y_{h}^{k',f}(t), \quad (10)$$

for $\forall f \in \mathcal{F}, \forall h \in \mathcal{H}, \forall t \in \mathcal{T}, \forall k \in \mathcal{K}/\{1\}$, while for the class with the highest priority, $k = 1$, the served requests are restricted by $y_{h}^{1,f}(t) \le C_{h}^{f,t}$, $\forall f \in \mathcal{F}, \forall h \in \mathcal{H}, \forall t \in \mathcal{T}$.

Problem Formulation: Our objective is the minimization of unserved requests by deriving the service forwarding request matrix $x_g(t)$ for each group $g \in \mathcal{G}$ over time as captured by the following problem formulation:

$$\text{max.} \sum_{g \in \mathcal{G}} \sum_{f \in \mathcal{F}} \sum_{t \in \mathcal{T}} \sum_{h \in \mathcal{H}} x_{gh}^{f}(t), \quad (11)$$
$$\text{s.t.: } (1) - (10), \quad (12)$$
$$y_{h}^{k,f}(t) \ge 0, x_{h}^{k,f}(t) \ge 0,\ \forall f \in \mathcal{F}, \forall h \in \mathcal{H}, \quad (13)$$
$$\forall k \in \mathcal{K}, \forall t \in \mathcal{T}, \quad (14)$$
$$x_{gh}^{f}(t) \ge 0,\ \forall g \in \mathcal{G}, \forall h \in \mathcal{H}, \forall s \in \mathcal{S}, \forall t \in \mathcal{T}, \quad (15)$$
$$z_{fv}^{t} \in \{0,1\},\ \forall f \in \mathcal{F}, \forall h \in \mathcal{H}, \forall v \in \mathcal{V}_h, \forall t \in \mathcal{T}, \quad (16)$$
$$u_{v}^{t} \in \{0,1\},\ \forall h \in \mathcal{H}, \forall v \in \mathcal{V}_h, \forall t \in \mathcal{T}. \quad (17)$$

However, the optimal solution is infeasible because the formulated problem falls into the Mixed Integer Programming category where the matrix associated with its constraints is not totally unimodular, and therefore the problem is NP-complete [30]. Furthermore, the constraints include non-linear convex restrictions (5). Lastly, the problem suffers from the practical challenges of *a*) the limited information about the future demand, and *b*) the enormous number of constraints and variables involved, known as curse of dimensionality in dynamic programming [31].

In the experiments in Section VI, we consider a hypothetical fully-centralized system where a controller has up-to-date view of all the cloudlets, *e.g.*, their input queue size, status of active pool instances, *etc.*, and schedules each user request to be processed at the farthest on-path cloudlet (*i.e.*, closest to the back-end cloud) which can satisfy its deadline. This approach either leaves cloudlets closer to users for the most deadline-sensitive functions or only uses them when the rest of the cloudlets are over-loaded. The fully-centralized approach uses the same provisioning approach used in the coordinated provisioning which is discussed next.

B. Coordinated Provisioning

Because the combined problem of deriving optimal request scheduling and the selection of optimal active pool function instances (*i.e.*, provisioning) for the cloudlets is overly complex, we focus on the provisioning problem assuming opportunistic, on-path processing of requests similar to decentralized strategies. Furthermore, we assume that the average demand $D_{g}^{f} = D_{g}^{f}(t)$ for each $t \in \mathcal{T}$ changes slowly over the time-scales considered. As a result, the dimension of the problem is greatly reduced.

Single Cloudlet: Consider the problem from the perspective of a single cloudlet h which has full information of functions' demand D_{g}^{f} $\forall g \in \mathcal{G}$ and $\forall f \in \mathcal{F}$. The cloudlet's goal is to maximize the utilization of its CPU resources since idle cores, *i.e.*, cores that are not serving any requests, are indicative of insufficient number of instances when there are *missed/unserved* requests. That is, a cloudlet h has to decide the type of functions to instantiate in each active pool container $v \in \mathcal{V}_h$. Crucial to this decision is the impact of container's expected usage frequency, *i.e.*, the frequency m at which the container is scheduled at a core.

Let group g direct its total demand of function f, D_{g}^{f}, to cloudlet h's queue class k_{gh}^{f} which is equipped with a single instance of the corresponding function. Then if the instance is always scheduled at a core, the maximum average number of requests served by the container at each time-slot is $\min\{D_{g}^{f}, 1\}$, since a single container cannot serve more than a single request per time-slot while the number of requests arriving is D_{g}^{f}.

On the other hand, if the instance is scheduled every m time-slots then the maximum number of requests served on average per time-slot is bounded by $1/m$. Furthermore, the demand served is aggregated for over m time-slots and it can be up to $m \times D_{g}^{f}$ as long as $m \times D_{g}^{f} < 1/m$. However, requests of f coming from group g cannot be queued for more than k_{gh}^{f} time-slots. Therefore, the rate of requests served on average by the unique instance of f at h is:

$$\rho_v^f(m) = \min\Big\{\frac{1}{m}, \sum_{g\in\mathcal{G}} \min\{m, k_{gh}^f\} D_g^f\Big\},\ \forall v \in \mathcal{V}_h, \forall f \in \mathcal{F}. \tag{18}$$

Clearly, the summation of requests have to respect the total number of cores at the cloudlet, imposing the restriction of:

$$\sum_{v\in\mathcal{V}_h} \rho_v^{f_v}(m_v) \le C_h,\ \forall h \in \mathcal{H}, \tag{19}$$

where f_v is the function instantiated at container v and m_v is the frequency at which v is expected to be scheduled at a core. Moreover, m takes discrete values in $[1, 2, ...]$; however, it is clear that as soon as $\rho_v^f(m) = 1/m$ for a specific m there is no point in assigning higher values to m since $\rho_v^f(m)$ after that point is a decreasing function, *i.e.*, $\rho_v^f(m+1) = 1/m+1$ even if $\sum_{g\in\mathcal{G}} \min\{m+1, k_{gh}^f\} D_g^f > \sum_{g\in\mathcal{G}} \min\{m, k_{gh}^f\} D_g^f$ as we see in equation (18).

Based on service rates, cloudlet h takes function instantiation decision by following the 6 steps described next that compose SINGLECLOUDLETPROV mechanism. For simplicity, we assume that functions are sorted according to their increasing order of execution deadlines, *i.e.*, $T_1 \le T_2 \le ... \le T_f$.

Step 0: Set the function under investigation $f = 1$ and the container under investigation $v = 1$. Set the instantiation threshold $\underline{\rho} = 0$ and previous container function $\underline{f} = \emptyset$.

Step 1: Find the m^* that maximizes $\rho_v^f(m)$ without exceeding $C_h - \sum_{v'\in\mathcal{V}_h\setminus\{v\}} \rho_{v'}^{f_{v'}}(m_{v'})$ according to (18).

Step 2: If $\rho_v^f(m^*) > \underline{\rho}$, instantiate f at v with parameter m^*, *i.e.*, $V_h^f = V_h^f \cup \{v\}$, and update the demand of instantiated function f according to:

$$\sum_{g\in\mathcal{G}} D_g^f = \sum_{g\in\mathcal{G}} D_g^f - \rho_v^f(m^*) \tag{20}$$

and the demand of the previous function $\underline{f}$ according to:

$$\sum_{g\in\mathcal{G}} D_g^{\underline{f}} = \sum_{g\in\mathcal{G}} D_g^{\underline{f}} + \underline{\rho}. \tag{21}$$

Step 3: If $C_h - \sum_{v'\in\mathcal{V}_h} \rho_{v'}^{f_{v'}}(m_{v'}) < \epsilon$ then STOP. Also if $\rho_v^f(m^*) \le \underline{\rho}$ and $f = F$, is the last function we consider, then STOP again. Otherwise, if simply $\rho_v^f(m^*) \le \underline{\rho}$ set $f = f+1$ and go to Step 1.

Step 4: If $\rho_v^f(m^*) < 1/m^*$ and $f = F$, *i.e.*, the last function in $\mathcal{F}$, then STOP. Else if $\rho_v^f(m^*) < 1/m^*$ in the next iteration consider the deployment of the next function in the sequence, $f = f+1$. Otherwise if $\rho_v^f(m^*) = 1/m^*$, consider the same function $f = f$.

Step 5: Find the container, v_{min}, that contributes the least according to:

$$\underline{v} = \arg\min_{v'\in\mathcal{V}_h} \rho_{v'}^{f_{v'}}(m_{v'}).$$

Set $\underline{\rho} = \rho_{\underline{v}}^{f_{\underline{v}}}(m_{\underline{v}})$, $v = \underline{v}$, and $\underline{f} = f_{\underline{v}}$ before going to Step 1.

Note that ϵ is considered a very small quantity that is crucial for identifying in practice the desirable resource utilization. Step 0 of SINGLECLOUDLETPROV is a straightforward initialization of the mechanism. On the other hand, Step 1 is simply trying to get the most out, in terms of served requests, from container v when supporting function f with respect to cloudlet's cores. Step 2 checks if there is actual improvement from instantiating function f at v and reduces the corresponding amount of aggregated requests while recovering the demand of the previously supported function $\underline{f}$ from container v. Step 3, applies a termination utilization condition. On the other hand, if there is no improvement and function f is the last to be investigated, then the mechanism terminates again.

Step 4 defines the function to be considered for deployment at the next iteration. Specifically, if the number of requests served are not restricted by m but by the number of aggregated expected requests, then there is no point for an additional instance of f; in that case, we should consider the next function in the sequence. Finally, Step 5 identifies the container to be replaced in the next iteration. Note that if a set of containers is not supporting any function, then one of them becomes the $\underline{v}$ since $\rho_v^{\emptyset}(m) = 0$. The algorithm terminates by either managing to utilize on average the available number of cores (Step 3), or by supporting the set of instances that achieves the highest possible core utilization.

Hierarchy of Cloudlets: Given the demands from user groups and remaining deadlines of functions observed at a single cloudlet, the provisioning is done using a deadline-based prioritization of functions in SINGLECLOUDLETPROV. Next, we consider provisioning resources of a hierarchy of cloudlets. In this case, updating the aggregate demand upon instantiating/replacing containers (step 2 of SINGLECLOUDLETPROV), implies updating the global demand from user groups. Also, different user groups traffic is reachable by different cloudlets as the traffic flows upward in the hierarchy visiting only on-path cloudlets after originating near a leaf cloudlet.

The coordinated strategy attempts to process requests as farther away from the users as possible (*i.e.*, as close to the back-end cloud as possible) while ensuring high resource utilization at each cloudlet. Below, we introduce COORDINATEDPROV where the algorithm considers each function separately (in increasing order of execution deadlines) and executes SINGLECLOUDLETPROV on the cloudlets one-by-one, starting with the root cloudlet of the hierarchy and proceeding with the cloudlets at increasingly lower levels as described below.

Step 0: Set $f = 1$ and initialize the set of functions by $\mathcal{F} = \{f\}$.

Step 1: Set $\underline{\rho} = 1 - \epsilon$.

Step 2: Execute the SINGLECLOUDLETPROV for $\underline{\rho}$ for each cloudlet starting from the ones located at the core and moving towards the edge.

Step 3: If $\underline{\rho} = 0$ and $f = F$, *i.e.*, last function, then STOP. Else if $\underline{\rho} = 0$ and $f \ne F$ then set $f = f+1$ and update the set of functions $\mathcal{F} = \mathcal{F} \cup \{f\}$ before going to Step 1. Else go to Step 4.

Step 4: If:

$$\sum_{g\in\mathcal{G}} \Big(\sum_{f\in\mathcal{F}} D_g^f - \sum_{h\in\mathcal{H}} \sum_{v\in\mathcal{V}_h^f} \rho_v^f(m_v) \Big) > \epsilon$$

then set $\underline{\rho} = \underline{\rho} - \epsilon$ and go to Step 1. Else if $f = F$ STOP. Otherwise, set $f = f+1$ and update the set of functions $\mathcal{F} = \mathcal{F} \cup \{f\}$ before going to Step 1.

The COORDINATEDPROV is simply trying to satisfy the demand for a function by serving the highest possible number of request at a single location. In order to achieve that, in the beginning it considers only the function with the strictest deadline, *i.e.*, $f = 1$, while introducing a high minimum number of requests served per container threshold $\underline{\rho}$ (at Steps 0 and 1). That is the underlying SINGLECLOUDLETPROV is modified to skip its 5th Step and go back to Step 1 until all of its containers are utilized first, since by default Step 5 will continue instantiating functions until reaching its capacity of active pool.

Then, the algorithm continues decreasing the threshold until either all the demand of the subset of functions considered is served, as we see in Step 4, or until setting a threshold of 0, at Step 3. After that the set of functions is incremented by including the next stricter function in terms of deadline. In the case that all functions have been introduced, the algorithm terminates. Finally, Step 2 of COORDINATEDPROV assumes a loose cloudlet hierarchy. Specifically, by starting applying the SINGLECLOUDLETPROV from the core (farthest to users) of the network to the edge (closest to users) it attempts to identify the optimal distance from the edge that is sufficient for aggregating the requests of the functions with the strictest deadline first. This leads to execution of requests with loose deadlines at cloudlets further away from its origin, leaving headroom on the resources of cloudlets closer to users for requests with stricter deadlines.

By executing requests at the computation spots with optimal distance from their users, the above method maximizes the number of function executions that meet their deadlines, provided that the execution times of all the functions are roughly equal (by our assumption) and the average demand is constant.

V. DECENTRALISED EDGE-CLOUDS

In this section, we take one step ahead and consider fully-decentralized strategies where individual computation spots make independent decisions for admission, scheduling, and provisioning. Our starting point is the following observation: determining which function's instance(s) to launch at which spot highly resembles the content caching problem where simple decentralized replacement policies, *e.g.*, Least Frequently Used (LFU), can be very effective. Such policies simply maintain a ranking of cached content and are also applicable for managing hierarchical caches. In case of function provisioning, the policy determines the quantity of the instances for each function to include in the active pool (AP) of each computation spot. This means that at each spot more than one instance of a function can be provisioned based on its popularity and deadline demands, whereas in the content caching schemes all requests can be satisfied by the same cached content.

Another difference from content caching policies that are typically executed upon the arrival of each request or content, we consider function provisioning policies which are executed only periodically because very frequent launching events lead to high overhead. More specifically, each computation spot runs a provisioning process periodically after each "replacement interval" of fixed duration (*e.g.*, 30 seconds) to determine the instances in its AP in the next interval.

Similar to the caching policies, we adopt a ranking approach that is applied to both the current function instances in AP and to functions that have currently no instances in AP. In order to perform the ranking, *a measurement module* at each computation spot collects statistics based on the demand (*i.e.*, request workload) observed in the previous interval. The measurement module collects and maintains the following for each function f:

1) *The utilization of accepted requests:* is the total execution time of the admitted requests for f at the computation spot.
2) *The missed utilization of the offloaded requests:* is the sum of the estimated execution times of the requests for f that were rejected due solely to insufficient or non-existing available instances as explained in Section III-B.
3) *The success rate of offloaded requests:* is the ratio of the offloaded requests for f (due to insufficient or non-existing available instances) that eventually meet their deadlines.
4) *The average adequate time:* is the total adequate time averaged over the requests of F.

In addition to bookkeeping of utilization due to admitted requests and the missed utilization due to rejected requests (*i.e.*, metrics one and two above), each computation spot can also track the success of offloaded requests in meeting their deadlines. This can be done by simply observing the remaining deadlines in the requests and then comparing them with the elapsed time between the arrival of the requests and the arrival of corresponding responses. This requires symmetry of request/response traffic which is the case in the setting of hierarchical edge-cloud network, where each individual edge-cloud node forwards traffic along the overlay of edge-cloud nodes towards the back-end cloud.

Each individual edge-cloud node also computes an average adequate time which determines the strictness of the requests' remaining deadline at the time of arrival. Only if the adequate time is positive, a request's deadline is satisfiable. If, on the other hand, the adequate time is negative, then the deadline is missed. The execution of a request can be performed upstream in case the adequate time is larger than the RTT to reach the next upstream node on the path to the back-end cloud.

Each provisioning policy maintains a ranking of two individual lists: *i)* Instantiated Functions (IF) and *ii)* Candidate Functions (CF). The instantiated functions are the ones that have at least one instance in the AP. The list of candidate functions are the ones that are considered for replacing an existing instance. We examine the following set of *decentralized* provisioning policies, each of which uses one or more of the above list of metrics:

- **Strictest Deadline First (SDF)**: This policy quantifies the value of functions in IF using the reciprocal of average adequate time as the metric. SDF ranks higher those functions with smallest positive adequate times. The set of functions in CF comprises functions with both non-zero missed utilization and positive adequate times. The ranking of CF is also done using the reciprocal of average adequate time as the valuation metric. Essentially, this policy uses the strictness of deadlines as the main consideration when provisioning functions. During the replacement phase, the policy iterates through the sorted IF list in increasing order of values, and replaces an existing instance of a function

f_{IF} in IF with a f_{CF} in CF ($f_{IF} \neq f_{CF}$) with the largest value, only if the value of f_{IF} is strictly less than f_{CF}. Upon replacing, the policy removes f_{CF} from the CF list and continues with the next f_{IF} in IF. The replacement phase terminates when an f_{IF}'s value is strictly greater than the f_{CF} with the largest value.

- **Least Frequently Used (LFU)**: This policy uses the utilization of accepted requests, *i.e.*, utilization, to quantify the value of the functions in IF. The set CF comprises functions with both positive adequate times and positive amount of missed utilization of offloaded requests that eventually violate their deadlines. The latter metric, which can be obtained by combining the metrics two and three above, is used for the valuation of the functions in CF. This policy considers only the functions that are *admittable*, *i.e.*, have positive adequate time, and have non-zero offloaded requests that missed their deadlines. The replacement phase proceeds just like in SDF, where instances of functions in IF is replaced with the instances of the functions in CF with larger values.
- **Hybrid**: This policy uses the single cloudlet provisioning algorithm SINGLECLOUDLETPROV (Section IV-B). The CF comprises functions with non-zero missed utilization and are sorted in increasing order of adequate times (*i.e.*, most deadline sensitive to least). The algorithm iterates through each function f_{CF} in the sorted CF list and replaces the function in IF that has the least utilization with an instance of f_{CF}, if f_{CF} has sufficiently high demand (*i.e.*, missed utilization) as described in step 2 of SINGLECLOUDLETPROV. The Hybrid policy is named as such, because it combines deadline strictness (SDF) and utilization (LFU) as the replacement criteria. Finally, this policy imposes a hard limit on the number of instance replacements in a single replacement phase (*i.e.*, for reasons of stability) to at most $1/15$ of the instances in the active pool, while SINGLECLOUDLETPROV does not impose a limit on the number of replacements.

The above function provisioning policies do not require any coordination. Each edge-cloud node passively monitors the incoming request/response traffic and use the monitoring information from the previous replacement interval to determine the set of function instances to replace for the next replacement interval. The provisioning of a function instance involves adding a special task in the input queue which executes the start-up procedure and possibly tearing down of the instance that is replaced by the new one.

VI. PERFORMANCE EVALUATION

In this section, we evaluate the performance of both centralized and decentralized strategies, presented in Sections IV and V, using a wide range of parameters. The objective is to evaluate the performance of the strategies in terms of success in meeting request deadlines and achieving high resource utilization. Next, we describe the setup of our evaluations before presenting the experiments in the remaining sections.

A. Evaluation Setup and Metrics

For the evaluation of the proposed replacement strategies, we use a packet-level, discrete time event simulator based on Icarus [32]. Icarus is originally a simulator for evaluating the performance of cache networks. The simulator code with our modifications and the scripts to generate the results are made publicly available[2]. We simulate the strategies using a binary tree topology with a height of five (*i.e.*, root and four sub-levels) as shown in Fig. 4. The root of the tree is the back-end cloud for all the functions containing unlimited resources, and the leaf nodes act as the gateway nodes without computing power where users are attached to. The internal nodes of the tree form the network of computation spots (*e.g.*, cloudlets), each deployed at various Points of Presences (PoP) such as exchange points. We assume computational resource capacity of cloudlets to be comparable to what is available in a Content Distribution Network PoP, which is typically 4-16 servers. Therefore, in terms of number of CPU cores, we assume a total of $C = 50$ cores per each cloudlet.

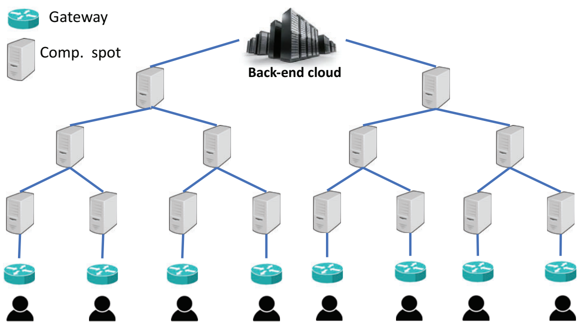


Fig. 4: The tree topology used in the experiments.

We consider a scenario, where the function population P of interest is a set of 10^3 functions. This set consists of the functions whose providers have a business relationship with the edge-cloud and are not meant to represent all possible functions in the Internet. Each function is associated with a processing time and a deadline.

In order to generate a deadline for a function (say f), we first generate a random time value (in ms) between the following lower- and upper-bounds: RTT from gateway node to its adjacent computation spot at the lowest level of the tree (lower-bound), RTT to reach the back-end cloud from the gateway node (upper-bound). This randomly generated value is added to the processing time of f to generate its deadline. Thus, we consider only latency-critical functions, whose requests can only be satisfied (*i.e.*, response delay within the deadline) through edge-cloud nodes (*i.e.*, computation spots) and not the back-end cloud.

That said, we choose a small, fixed processing time for each function selected from a range of 1–5 ms and assume homogeneous nodes, where requests of the same application require the same processing time everywhere. The fixed processing times for functions executed at a given location is, to a large extent, a property of Serverless tasks that are inherently stateless. For example, the processing times of image processing tasks (e.g., detection of objects) are very stable per video frame or image [28]. Different image processing tasks (e.g., ones that detect different objects) which require different processing times are considered as different functions in our model.

[2]https://github.com/oascigil/IcarusEdgeSim

TABLE II: Default evaluation parameters.

Parameter	Value
Number of edge-cloud nodes (computation spots) in the tree	14
Size of function population	$P = 10^3$
Number of CPU cores	$C = 50$
Active pool capacity	$F = 3 \times C$
Average request rate per second	10^5
Function starting time (cold-start)	0.1–1 seconds
Zipf exponent (for synthetic workloads)	$\alpha = 0.75$
Processing times of functions	1–5 ms
Scheduling policy	EDF

We assign inter-connection latencies as follows: each path connecting consecutive computation spots has a propagation delay of 10 ms, the paths connecting nodes (at the top-level of the tree) to the back-end cloud have a latency of 60 ms to represent the wide area network latencies [33]. The gateway (leaf) nodes connect to the edge-cloud nodes (at the lowest-level of the tree) with a latency of 2-3 ms and to users with negligible latency. In order to simplify the function size provisioning in the experiments, we assume that each function has the same memory footprint for its instance. The total memory capacity for the entire edge-cloud network is a parameter in our experiments, and we uniformly distribute the total capacity, which determines the capacity F of the active pool of each cloudlet. As mentioned before, we assume that each edge-cloud node prefetches and stores the function codes in advance.

Our evaluation is based on the following performance metrics:

- **Satisfaction Rate (in percentage of issued requests)**: The ratio of requests that are processed and returned back to their originating user within the service deadlines.
- **Percent Idle Time**: The ratio of the time that the CPU cores are staying idle, *i.e.*, not executing any function. This metric indicates the utilization of resources at the edge-clouds.
- **Overhead**: The average number of new function instances launched per computation spot over the replacement periods.

During the experiments, we compute the above metrics periodically at the end of each *replacement period* using the requests that arrived during that period. We set the default length of the replacement period to 30 seconds in the experiments. Initially, each cloudlet contains a single instance from a randomly selected subset of the population P. In our experiments, we set the default size of the active pool to $F = 100$ instances, which is twice the size of the number of cores (C) at each cloudlet. At the end of each replacement period, cloudlets may replace their instances with new ones, maintaining the number of containers in their active pool fixed with F instances. The launching of a new instance results with a "start-instance" request to be added in the input queue, and such requests are scheduled together with the other (*e.g.*, user) requests in the queue for processing. A start-instance request involves all the processing necessary for launching a new function instance, during which a CPU core is allocated. The required processing time for launching an instance is function specific, and we set the starting-up processing time of each function to a randomly chosen duration within the range of $0.1 - 1$ second.

We use synthetic workloads in all the experiments where users send traffic with an aggregate mean rate of 10^5 requests per second. Each request is assigned to a randomly chosen gateway node where it originates from. The association of request to a function type in synthetic workload is generated using a Zipf distribution, which determines the function popularity. We use a default Zipf exponent of 0.75 for the synthetic workload, and in Section VI-E, we consider different Zipf exponents. The rest of the default parameters are listed in Table II. In the rest of this section, we investigate the impact of various system parameters listed in Table II on the performance of both the centralized and decentralized provisioning policies.

B. Comparison of Strategies

In Fig. 5, we depict the performance of the provisioning policies over a time period of 600s using the synthetic workload. Each data point in all the plots corresponds to the average performance over a replacement period of 30s. Initially (*i.e.*, t=0) all the strategies start with an identical set of randomly chosen function instances in the active pool at each cloudlet. Consequently, SDF, LFU, hybrid and coordinated strategies, which all use opportunistic, on-path function execution, perform exactly the same (leftmost and middle plots in Fig. 5) during the first replacement interval (*i.e.*, t=0-30s). On the other hand, because the fully-centralized strategy (labeled as "F. Central." in the figures) differs from the others in how it selects an on-path cloudlet to process each request (*i.e.*, centralized selection as opposed to opportunistic), and therefore its initial performance for 0-30s differs from the rest of the strategies despite having the same initial set of active pool functions at each cloudlet.

In the leftmost plot of Fig. 5, we present the satisfaction rates for all the policies. We observe that all the strategies except SDF gradually improve their satisfaction rates during the subsequent replacement periods (*i.e.*, every 30s). Eventually, all the strategies stabilize and maintain a roughly constant satisfaction rate using a synthetic workload with a stationary distribution of function popularities. Both the centralized and the coordinated strategies stabilize much faster than the decentralized strategies, as a result of using the knowledge of the global demand as an input to the provisioning process. Unlike the coordinated strategies, in SDF, hybrid and LFU strategies, each edge-cloud node makes an individual decision based on the locally observed demand without the knowledge of the global demand. While the leaf cloudlet nodes can observe all the demand from a user group, the upper-level nodes can only observe the offloaded requests that are not processed and therefore filtered at the lower-levels.

Unlike the rest of the strategies, SDF fails to retain the peak satisfaction rate of around 18% at 60s and gradually performs worse. This is mainly because SDF does not take utilization of functions into consideration in provisioning decisions. Instead, SDF focuses entirely on the adequate time, and as a result, under-utilized functions with smaller adequate times are increasingly provisioned. According to our earlier results, SDF performs comparably well with the other strategies only when the functions with strict deadlines are also the most demanded [25]. The synthetic workload we generate

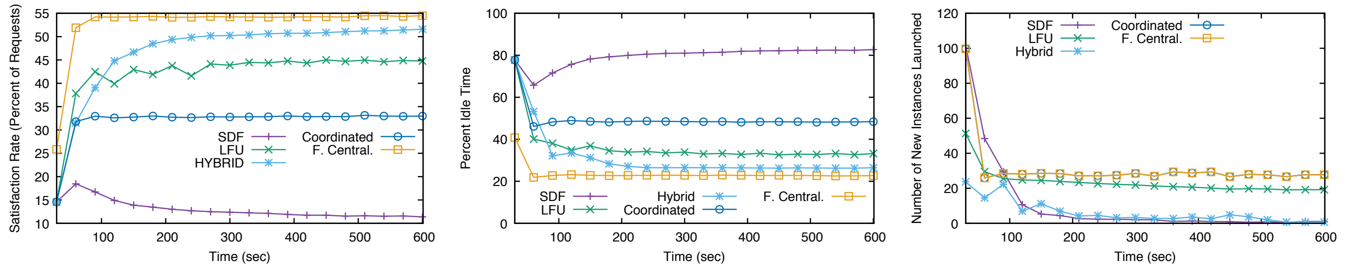


Fig. 5: Performance of strategies over a time period of 600 seconds.

for the experiments have randomly assigned deadlines and such a correlation between popularity (*i.e.*, high demand) and deadline strictness do not exist.

We observe from the middle graph of Fig. 5 that the average idle times of the edge-cloud nodes over time with hybrid strategy is lower than the other decentralized strategies which is reflected to the achieved satisfaction rates. The hybrid strategy converges to around 27%, while the LFU strategy retains around 34% idle time. This results with hybrid strategy having a higher satisfaction rate than the LFU. The SDF strategy fails to utilize the CPU resources of the edge-cloud nodes; the resources remain idle for more than 80% of the time for t$>$150s. As can be observed, the portion of the idle time that reaches the lowest point initially at around t=60s coincides with the point where SDF retains its highest satisfaction rate of around 18%.

The rightmost plot of Fig. 5 presents the average number of new instances launched (*i.e.*, replacements) per edge-cloud node during provisioning decisions at the end of each replacement period by each strategy. At the end of the first replacement period (t=30), all the strategies launch a large number of instances replacing the randomly chosen instances at t=0. We observe that coordinated provisioning used by the centralized strategies perform slightly more replacements than others. This is because the coordinated provisioning approach is highly-reactive to changes in the demand and does not impose a limit on the number of replacements in a single period (unlike the hybrid strategy), which results in high number of new instances to be launched, replacing other existing instances. Note that, even though the average *global* demand for each function is roughly constant, there is fluctuation in the demands of individual functions particularly at the cloudlets in the lower levels of the hierarchy, because the simulator picks the origin of each request uniformly at random in order to account for the volatile nature of user demand in the edge networks.

We observe that the decentralized LFU strategy launches the highest number of instances at the end of replacement intervals slightly lower than the centralized strategies. The hybrid strategy uses both utilization of resources and deadline sensitivity of functions, and as a result it achieves a lower overhead of new instance launches. The overhead of launching instances is the processing of a cold-start process, which takes CPU resources.

The two centralized strategies achieve substantially different satisfaction rates from each other. While the fully-centralized strategy has the highest satisfaction rate during all the replacement periods among all the strategies, the coordinated strategy performs significantly worse than the decentralized strategies LFU and hybrid. Both centralized strategies use the same coordinated provisioning approach to periodically (every 30s) determine the active pool functions at each edge-cloud node. This can be seen in the rightmost graph of Fig. 5 with identical number of function instances that are launched for the coordinated and fully-centralized strategies.

The coordinated provisioning approach deploys active instances at the highest (*i.e.*, closer to the back-end cloud) possible level of the edge-cloud hierarchy where the requests can be processed. This approach therefore aims to utilize CPU resources of higher (lower) level edge cloud nodes with the instances of functions that are less (more) strict in terms of their deadlines. If after this process, there is still unused active pool capacity at the lower-level nodes, then additional (*i.e.*, back-up) instances for popular functions are launched.

As expected, the coordinated provisioning approach works better with a fully-centralized request scheduling approach that processes requests at the farthest on-path cloudlets. In practice, this approach requires a per-request communication and coordination overhead as each on-path cloudlet must be informed on whether it should admit and process a request. The coordinated provisioning approach similarly attempts to place instances in a top-down approach with farthest locations preferred for provisioning functions with loose deadlines. However, when a function with a loose deadline is also popular, then the coordinated approach typically places extra instances at cloudlet locations close to users in order to deal with the demand. When the demand is not controlled carefully and placed opportunistically, the resulting performance suffers as computing resources of nearby cloudlets are taken over by requests that could be processed further away.

We also observe that the decentralized strategies can experience fluctuations in satisfaction rates as experienced by the LFU strategy for ten replacement periods (*i.e.*, t=30-300). Such fluctuations occur due to overestimation of demand for popular functions with loose deadlines (*i.e.*, admittable by multiple on-path cloudlets along the hierarchy), for which individual on-path edge-cloud nodes at different levels of the hierarchy simultaneously instantiate functions. We do not observe instability in hybrid strategy as the instantiation decisions consider not only popularity but also strictness of remaining deadlines.

C. Active Pool Capacity

The performance of the strategies with varying active pool size per cloudlet is depicted in Fig. 6. Each data point in the plots is an average performance that is observed over 600 seconds. As expected, we observe the satisfaction rates of all

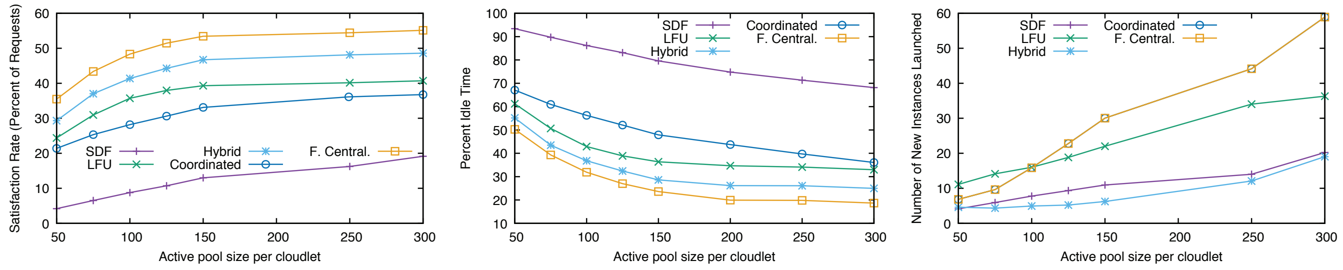


Fig. 6: Performance of strategies with different active pool capacity per Computation Spot.

strategies improve as the number of containers in the active pool per cloudlet increases (leftmost plot of Fig. 6. This is a result of the decreasing idle times of the strategies as the active pool size increases which increases the number of processed requests (middle plot of Fig. 6). Similar to the previous results, the hybrid strategy achieves the best performance among the decentralized strategies with a satisfaction rate only 10% less than the fully-centralized strategy. We also observe that the number of new instances launched increases roughly linearly with increasing containers in the active pool as shown in the rightmost plot in Fig. 6.

D. Compute Resources

The performance of the strategies with varying CPU cores is depicted in Fig. 7. We observe that the gap between the satisfaction rates of the fully-centralized strategy and the decentralized strategies (*i.e.*, LFU and hybrid) narrows down as the number of cores increases. This is because decreasing contention of requests for CPU resources among requests at each cloudlet which makes the processing location of requests less important. On the other hand, with higher contention among requests, it is important to manage which requests are processed where in order to open room for requests with strict deadlines at the edge-clouds.

A similar trend can be seen in the middle plot in Fig. 7 where the difference in idle time difference of fully-centralized and the other strategies (except SDF) reduces with increasing number of CPU cores. We observe that the gap between LFU and hybrid slightly increases for number of cores greater than 30. This is because of the increasing number of new instances launched by the LFU strategy (rightmost plot in Fig. 7) for $t>30$, which leads to increasing usage of CPU resources for cold-starts. Again the reason for LFU to launch a higher number of new instances is because of its reactive nature similar to fully-centralized strategy where small changes in the demand leads to immediate action by the edge-clouds in replacing function instances.

E. Service Popularity Distribution

In the above scenarios, we used a default Zipf exponent value of 0.75 when determining the functions' popularity. We present the performance of strategies for varying function popularity distribution, *i.e.*, Zipf exponent values, in Fig. 8. As expected the higher values of Zipf exponent leads to higher satisfaction rates for all strategies except SDF. SDF is unable to take advantage of locality of reference in the requests as a result of increasing Zipf exponent value because SDF considers adequate time (remaining deadline) as the only criteria when provisioning functions. The rest of the strategies consider utilization and can take advantage of requests for a less diverse set of functions.

The idle times (middle plot in Fig. 8) also show a decreasing trend for all strategies but SDF. All strategies except SDF also reduce their overhead of launching new instances as there are more requests for a shrinking set of popular functions (rightmost plot in Fig. 8).

Summary of results: In general, we observe that hybrid strategy achieves a performance that is comparable with the hypothetical, fully-centralized one, which requires a central controller to coordinate the edge-clouds and make per-request decisions. LFU strategy also performs close to hybrid, but suffers from a higher overhead due to being more reactive in responding to changes in request patterns. The coordinated strategy performs worse than both LFU and hybrid because its top-down provisioning approach does not agree with opportunistic admission. SDF achieves the worst performance due to its lack of concern for utilization of CPU resources.

We observed that the decentralized strategies are effective for a wide range of system parameters. The only downside of decentralized strategies is their slower convergence than centralized strategies and the possibility of experiencing fluctuations in performance due to lack of coordination in provisioning decisions. However, in our earlier work [25], we have demonstrated with real-world traces that decentralized strategies can in fact adapt to fluctuations in the demand, given appropriate setting of provisioning periods. On the other hand, coordinated provisioning strategies converge more rapidly to their maximum performance upon changes in demand.

VII. Conclusions

Following the success of *Serverless* computing paradigm for the management of FaaS clouds, we investigated a similar approach for FaaS edge-clouds. In this approach, edge-clouds take care of the resource provisioning and allocation tasks for the application providers who simply make their function image (*i.e.*, code) available to the edge-clouds. We considered various resource provisioning and allocation strategies for edge-clouds—consisting of distributed computation spots hierarchically arranged on the paths to back-end clouds of application-providers—for the deployment of latency-critical services with hard deadlines on computations.

Resource provisioning attempts to fully utilize the CPU resources of the computation spots by maintaining a set of warm (active pool) containers, each capable of instantiating a single instance of a function. To that end, provisioning

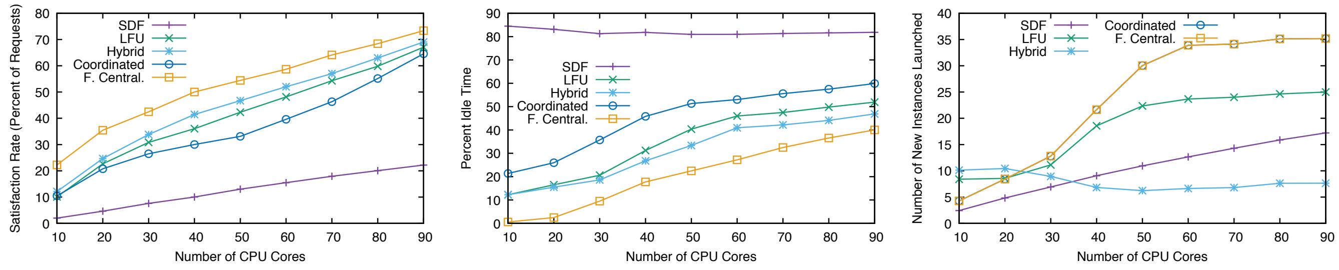


Fig. 7: Performance of strategies with different CPU resources.

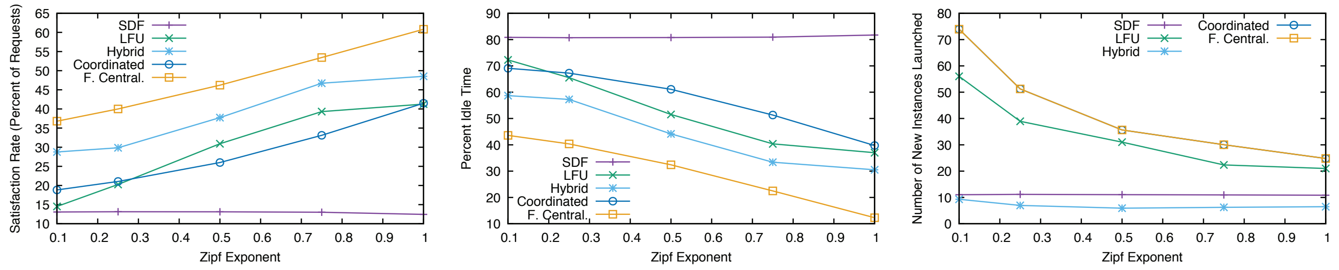


Fig. 8: Performance of strategies with varying popularity of functions.

periodically adapts the function associations for a fixed-size warm containers by launching new function instances to *replace* existing instances of other functions according to demand; while resource allocation selectively admits incoming requests, if and only if their deadlines can be met. In a fully-decentralized approach, computation spots independently make provisioning and allocation decisions, resembling the existing cache management strategies such as opportunistic caching with LFU replacement policy. In the other extreme approach, a fully-centralized approach makes use of a controller to orchestrate provisioning and allocation of resources at each computation spot.

We formulated an optimal resource provisioning and allocation for the fully-centralized strategy, which takes queuing and scheduling of requests into account. This strategy assigns requests to optimal computation spots for maximizing the deadline satisfaction rate and also periodically provisions the optimal set of function instances to maximize CPU utilization. Because of the coordination overheads of such an approach, we proposed a practical centralized approach which only requires periodic coordination among computation spots for provisioning and leaves admission decisions to computation spots. Finally, we investigated the performance of all the centralized and decentralized strategies under various system parameters in a packet-level discrete event simulator. Our main finding is that a fully-decentralized strategy—a hybrid of LFU and a deadline-centric SDF—can achieve comparable performance to a hypothetical fully-centralized one, whose performance merely demonstrates a theoretical upper-bound.

References

[1] M. Satyanarayanan *et al.*, "Edge analytics in the internet of things," *IEEE Pervasive Computing*, 2015.

[2] J. Cho *et al.*, "Acacia: Context-aware edge computing for continuous interactive applications over mobile networks," in *ACM CoNEXT*, 2016.

[3] K. Ha *et al.*, "Towards wearable cognitive assistance," in *International conference on Mobile systems, applications, and services*. ACM, 2014.

[4] I. Psaras, "Decentralised edge-computing and iot through distributed trust," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2018, pp. 505–507.

[5] B. Zhang *et al.*, "The cloud is not enough: Saving iot from the cloud." in *HotStorage*, 2015.

[6] F. Bonomi *et al.*, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13–16.

[7] M. Satyanarayanan *et al.*, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, 2009.

[8] M. Patel *et al.*, "Mobile-edge computing introductory technical white paper," *Mobile-edge Computing (MEC) industry initiative*, 2014.

[9] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *ACM SIGCOMM CCR*, 2014.

[10] L. Tong *et al.*, "A hierarchical edge cloud architecture for mobile computing," in *IEEE INFOCOM*, 2016.

[11] A. Tasiopoulos *et al.*, "Fogspot: Spot pricing for application provisioning in edge/fog computing," *IEEE Transactions on Services Computing*, 2019.

[12] E. Jonas *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[13] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, 2019, pp. 225–236.

[14] B. Yang *et al.*, "Cost-efficient nfv-enabled mobile edge-cloud for low latency mobile applications," *IEEE Transactions on Network and Service Management*, vol. 15, no. 1, pp. 475–488, 2018.

[15] T. K. Phan *et al.*, "Utility-maximizing server selection," in *IEEE IFIP Networking*, 2016.

[16] ——, "Utility-centric networking: Balancing transit costs with quality of experience," *IEEE/ACM Transactions on Networking (TON)*, 2018.

[17] L. Baresi *et al.*, "Paps: A framework for decentralized self-management at the edge," in *International Conference on Service-Oriented Computing*. Springer, 2019, pp. 508–522.

[18] I. Psaras *et al.*, "Probabilistic in-network caching for information-centric networks," in *Proceedings of the second edition of the ICN workshop on Information-centric networking*. ACM, 2012, pp. 55–60.

[19] V. Sourlas *et al.*, "Distributed cache management in information-centric networks," *IEEE Transactions on Network and Service Management*, vol. 10, no. 3, pp. 286–299, 2013.

[20] J. Li *et al.*, "DR-Cache: Distributed resilient caching with latency guarantees," in *IEEE INFOCOM*, 2018.

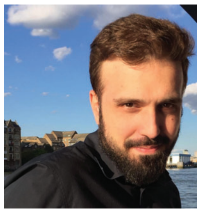
[21] S. Li *et al.*, "Joint admission control and resource allocation in edge computing for internet of things," vol. 32, no. 1, 2018, pp. 72–79.

[22] A. G. Tasiopoulos *et al.*, "Edge-map: Auction markets for edge resource provisioning," in *2018 IEEE 19th International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*. IEEE, 2018, pp. 14–22.

[23] T. Zhao *et al.*, "ReD/LeD: An asymptotically optimal and scalable online algorithm for service caching at the edge," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 8, pp. 1857–1870, 2018.

[24] D. Zeng *et al.*, "Joint optimization of task scheduling and image placement in fog computing supported software-defined embedded system," *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3702–3712, 2016.
[25] O. Ascigil *et al.*, "On uncoordinated service placement in edge-clouds," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017, pp. 41–48.
[26] H. Tan *et al.*, "Online job dispatching and scheduling in edge-clouds," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, 2017, pp. 1–9.
[27] J. Xu *et al.*, "Zenith: Utility-aware resource allocation for edge computing," in *2017 IEEE international conference on edge computing (EDGE)*, 2017, pp. 47–54.
[28] S. Yi *et al.*, "Lavea: Latency-aware video analytics on edge computing platform," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, 2017, pp. 1–13.
[29] C. Shi *et al.*, "Cosmos: computation offloading as a service for mobile devices," in *Proceedings of the 15th ACM international symposium on Mobile ad hoc networking and computing*, 2014, pp. 287–296.
[30] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*. Springer, 1972, pp. 85–103.
[31] D. P. Bertsekas, *Dynamic programming and optimal control*. Athena scientific Belmont, MA, 1995, vol. 1, no. 2.
[32] L. Saino *et al.*, "Icarus: a caching simulator for information centric networking (icn)," in *Proceedings of the 7th International Conference on Simulation Tools and Techniques (ICST)*, 2014, pp. 66–75.
[33] Q. Duan, *Delay Characteristics of Packet Switched Networks*. Boston, MA: Springer US, 2010, pp. 203–223.

Onur Ascigil received his PhD. degree from the Computer Science Department, University of Kentucky, Lexington, USA in 2014. From 2008 to 2014 he worked as a research assistant and from Jan. 2015 to Aug. 2015 as a Post-doctorate Research Associate at the Laboratory for Advanced Networking, University of Kentucky. In September 2015 he joined the Electronic and Electrical Engineering Department, UCL, London as a Research Associate.



Argyrios G. Tasiopoulos received the B.Sc. in Informatics and the M.Sc. in Computer Science both from the AUEB. He obtained his Ph.D. degree in Electronic and Electrical Engineering from the University College London, in 2018. He is currently a Research Associate at the EE department of UCL. His research efforts lies at the intersection of operations research, telecommunications, and economics.

Truong Khoa Phan received his PhD degree from INRIA/I3S, Sophia, France. He is currently a Senior Research Associate at the Department of Electronic and Electrical Engineering, University College London. His research interests include network optimisation, cloud computing, multicast and Peer-to-Peer Networks.

Vasilis Sourlas received his Diploma degree from the Computer Engineering and Informatics Department, University of Patras, Greece, in 2004 and the M.Sc. degree in Computer Science from the same department in 2006. In 2013 he received his PhD from the Department of Electrical and Computer Engineering, University of Thessaly (Volos), Greece. In Jan. 2015 he joined the Electronic and Electrical Engineering Department, UCL, London to pursue his two years Marie Curie IEF fellowship.

Ioannis Psaras is an EPSRC Fellow at the Electrical and Electronic Engineering Department of UCL. He is interested in resource management techniques for current and future networking architectures with particular focus on routing, caching and congestion control. Before joining UCL in 2010, he held positions at the University of Surrey, and Democritus University of Thrace, Greece, where he also obtained his PhD in 2008. He has held research intern positions at DoCoMo Eurolabs and Ericsson Eurolabs.

George Pavlou is Professor of Communication Networks in the Department of Electronic and Electrical Engineering, University College London, UK. He received a Diploma in Engineering from the National Technical University of Athens, Greece and M.S. and Ph.D. degrees in Computer Science from University College London, UK. His research interests focus on networking and network management. In 2011 he received the Daniel Stokesbury award for "distinguished technical contribution to the growth of the network management field".