# On Uncoordinated Service Placement in Edge-Clouds

Onur Ascigil*, Truong Khoa Phan*, Argyrios G. Tasiopoulos*, Vasilis Sourlas*, Ioannis Psaras*, and George Pavlou*
*Department of Electronic and Electrical Engineering, University College London, UK.
Email: {o.ascigil, t.phan, argyrios.tasiopoulos, v.sourlas, i.psaras, g.pavlou}@ucl.ac.uk

***Abstract*—Edge computing has emerged as a new paradigm to bring cloud applications closer to users for increased performance. ISPs have the opportunity to deploy *private edge-clouds* in their infrastructure to generate additional revenue by providing ultra-low latency applications to local users. We envision a rapid increase in the number of such applications for "edge" networks in the near future with virtual/augmented reality (VR/AR), networked gaming, wearable cognitive assistance, autonomous driving and IoT analytics having already been proposed for edge-clouds instead of the central clouds to improve performance. This raises new challenges as the complexity of the resource allocation problem for multiple services with latency deadlines (*i.e.*, which service to place at which node of the edge-cloud in order to satisfy the latency constraints) becomes significant. In this paper, we propose a set of practical, uncoordinated strategies for service placement in edge-clouds. Through extensive simulations using both synthetic and real-world trace data, we demonstrate that uncoordinated strategies can perform comparatively well with the optimal placement solution, which satisfies the maximum amount of user requests.**

## I. Introduction

Cloud computing is facing new challenges in meeting the quality-of-service (QoS) requirements of emerging applications such as virtual/augmented reality (VR/AR) [1], interactive networked gaming, wearable cognitive assistance [2], and edge (video, IoT) analytics [3] to name a few. We argue that the most pressing requirement of those emerging applications is the response latency; that is, the delay between submitting a request to the network and getting the outcome of the computation delivered back to the user. Centralized data-centers are not the appropriate place for achieving small response latencies because of the potentially high network transmission delays in wide area networks.

At the same time, there has been an increasing variety of network edge and access devices with general-purpose computational resources such as base stations, access points, and edge routers. In addition, deployment of small-scale data centers at the "network edges" (*i.e.*, close to users), named cloudlets [4] or micro-clouds have been proposed to accommodate heavier demands. Moreover, with the recent advances in virtualization technology, only the application code and shared libraries can be packaged exclusively (*e.g.*, Docker containers, Unikernels). In particular, Unikernels are very lightweight in size and can also be instantiated (*i.e.*, boot up) in up to just few hundreds of milliseconds.

As a result, there is a current trend for deploying small-scale cloud resources closer to users at the ISPs in a decentralized manner [5], [6], [7]. In a decentralized edge-cloud, a set of paying Application Providers (APs) run virtualized instances

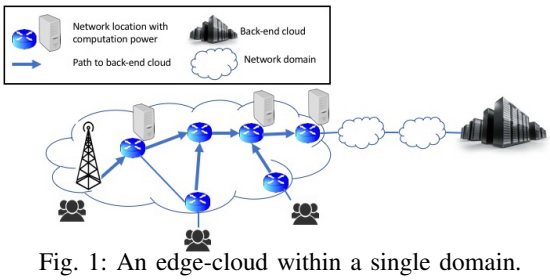


Fig. 1: An edge-cloud within a single domain.

of their services. This has the potential to greatly reduce the round-trip delay and to provide higher end-to-end bandwidth to users, while bringing revenue to the edge-cloud providers. Specifically, we assume a *private edge-cloud* business model (possibly operated by an ISP) who owns and manages a distributed edge infrastructure consisting of *computation spots*, *i.e.*, network locations/nodes where computational capabilities are offered, similar to the one shown in Fig. 1. As opposed to large-scale, purpose-built cloud architectures that can provide on-demand invocation of services with essentially unlimited elasticity, the computational resources at the edge-clouds are expected to be highly contended, and fully-utilized especially at peak times.

In an environment where computing power is available at several different points (*e.g.*, attached to network routers and PoPs) along the path from the client to the back-end cloud, network operators need to be prepared to allocate computation facilities to user applications according to the applications' requirements. We envision that those requirements will come mainly in terms of response latency. We also envision that the code to run the packaged application is stored in repositories within the domain or even locally stored at the computation spots. In this work, we specifically assume that code for any application is available at every computation spot/node. Given that computation capacity is less than storage capacity (following price trends), only a subset of those services are then instantiated (*i.e.*, one or more VMs are dedicated and running) and can serve incoming requests.

The problem of which services to instantiate at which node highly resembles the well-investigated problem of resource storage allocation. For instance, the problem of choosing which content to cache locally, given that local storage capacity at the cache node is less than the storage available in the back-end cloud, has been studied extensively, *e.g.*, in [8]. In this work, we argue that the problem of which services to instantiate at a computation spot can be seen from a similar angle. We, therefore, apply principles of cache management

2330-2186/17 $31.00 © 2017 IEEE
DOI 10.1109/CloudCom.2017.46

research to allocate computing resources and attempt to find out whether well-known resource placement and allocation techniques, such as Least Recently Used (LRU) and Least Frequently Used (LFU), can also be applied to edge-computing environments.

The main advantage of such uncoordinated resource allocation techniques is their simplicity—*i.e.*, they require no central coordination among the distributed set of computation spots. In a single or multi-ISP setting, we consider computation spots available and provisioned *on-path* towards a central (back-end) cloud, where the default path for each service request terminates. We assume that each service is associated with a deadline on the response delay and user requests contain their remaining deadlines associated with the service they request. The edge-clouds use only the remaining deadline on the incoming requests to perform *admission*, *scheduling* and *placement* decisions.

Related work in the area of resource allocation and request routing for edge-clouds have considered optimal allocation of services and dispatching of requests to edge-cloud nodes, *e.g.*, [9], [10], [11]. In general, these proposals attempt to minimize the average response time of the edge-clouds without considering individual QoS requirements of applications, which we model in the form of deadlines on response delays. More importantly, the existing proposals for optimal allocation require some form of coordination among the nodes, for instance, in determining the least-congested node for routing the requests. In this work, we consider resource allocation strategies, which require no coordination and communication overhead. Our findings with synthetic and trace-based workloads demonstrate that the examined uncoordinated resource allocation strategies can perform comparably well in comparison to a centralized scheme, which has the knowledge of all future requests and instantaneous load on the edge-cloud nodes.

The rest of this paper is organized as follows. In Section II, we discuss related work. We discuss the uncoordinated resource allocation strategies involving admission, scheduling and placement decisions in Section III. Then in section IV, we briefly explain the centralized, optimal resource allocation scheme that we use as a benchmark in our evaluations in Section V. Finally, we conclude the paper in Section VI.

## II. Related work

Existing research has outlined the incentives for ISPs to deploy cloud resources for third-party applications closer to the users [4], [6]. Although extensive research has been conducted on the placement of service instances in decentralized edge-cloud architectures, the majority of those have considered computation offloading tasks for a single service (*e.g.*, image processing) *e.g.*, [10], [12], [13]. Instead, we focus on the use case of bringing multiple third-party cloud applications closer to users, where each application has its unique service quality (QoS) requirement. We model the QoS requirement with an upper-bound (*i.e.*, deadline) on the response latency; that is the delay between sending a request to the network and getting the result back by the user. We argue that response delay is the most pressing requirement of the emerging applications (*e.g.*, virtual/augmented reality (VR/AR) [1], interactive networked gaming, wearable cognitive assistance [2], and edge (video, IoT) analytics [3]) to motivate the adoption of edge-clouds.

Similar to our work, Hou et al. considered placement of services at an edge-cloud that is connected to a back-end cloud through a backhaul connection [14]. This work considers various strategies to determine when to replicate a service in an edge-cloud based on the downloading cost (in terms of bandwidth) of the service. An LRU-like replacement policy is used to choose which service to remove when a new one is downloaded. However, this work does not consider queuing and execution times of service requests, which we take into account. More importantly, we focus on the utilization of the edge-cloud resources and QoS of applications when replicating services instead of just the downloading cost.

In [11], Tan et al. consider the problem of optimally scheduling and dispatching of tasks to distributed edge-clouds assuming central coordination to get up-to-date (computational) congestion level at each node so that tasks can be submitted to least-congested nodes. This work considers scheduling of services, and propose Highest Residual Density First (HRDF) scheduling, which aims to minimize the completion times of tasks. In our work, we consider deadline-based scheduling to satisfy hard deadlines on user requests. An admission policy determines whether a request can be successfully executed, and the edge-cloud nodes only accept a request if they can meet the deadline.

Tong et al. propose hierarchical cloud deployment as a way to handle peak loads effectively [10]. Here we also adopt a similar provisioning of edge-cloud nodes as on-path resources. In particular, the requests are simply routed towards the back-end cloud and are opportunistically executed by the edge-cloud nodes along the path (see Fig. 1). In addition to the distributed (*i.e.*, uncoordinated) placement methods, we also provide a centralized, offline method for the placement of application instances, in order to meet the service deadlines. Similar offline solutions for edge-clouds have been proposed, albeit without considering individual service quality (*i.e.*, in terms of deadlines).

## III. Uncoordinated Resource Allocation Strategies

We perform opportunistic, on-path execution with uncoordinated resource allocation in edge-clouds. Resource allocation involves admission, scheduling and placement mechanisms.

### A. Admission and Scheduling of requests

An *admission decision* precedes *scheduling* and is made on each incoming service request to decide whether to accept and perform the task associated with the service. Because resources are constrained, computation spots only admit requests whose deadlines can be met. A request $r$ for service $s$ might be rejected for execution at a given computation spot for the following reasons: *i*) $s$ has not been placed in the spot (i.e., not instantiated in a VM) and *ii*) requests admitted and have to be served before $r$ according to the scheduling policy need more time than $r$'s deadline (*i.e.*, congestion). Requests rejected at a computation spot are forwarded towards the central cloud (and not towards peer computation spots), and

may be opportunistically executed at another computation spot along the path, as shown in Fig. 1. If rejected by all spots/nodes along the path, a request arrives at the central cloud, where it is guaranteed to be executed; albeit missing its deadline.

Here we assume two non-preemptive scheduling policies: *i*) Earliest Deadline First (EDF) and *ii*) First-In-First-Out (FIFO). These policies determine the order of execution for the admitted requests waiting in the input queue. Once a request is admitted by a node, it is guaranteed by the node to meet its deadline. To that end, at the time of admission, a "*completion time*" is computed for each request. Only if the calculated completion time is not greater than the current time plus the remaining deadline, then the request is admitted. In FIFO policy, the computation of the completion time is straightforward and is computed only once for each request. In particular, the completion time in FIFO is equal to the execution time of the given request $r$ plus the waiting time for the execution (*i.e.*, summation of the corresponding execution times) of the requests that have already been scheduled for execution before the corresponding request.

In EDF requests for services with smaller (*i.e.*, stricter) remaining deadlines are always executed before requests (already admitted) with larger remaining deadlines. In EDF the completion time of each admitted request has to be recomputed for each new request arriving at the given computation spot. Particularly, assuming only one CPU core for the spot, the completion time for a request $r$, is equal to the execution time of $r$ plus the waiting time for the execution of the requests with stricter deadlines that have already been scheduled for execution. In EDF requests with higher remaining deadline are always "pushed" further down in the input queue after new requests with shorter deadlines. If after the computation of the completion time of every request in the queue all the deadlines can still be satisfied, then the new request is finally admitted and scheduled for execution otherwise is rejected.

### *B. Service Placement*

In order to admit and schedule a service request, a computation spot/node has to either *i*) have loaded the service as one of its VMs, or *ii*) have the service code in its local memory to initialize it upon request. Constrained by the main memory capacity, a service placement policy decides which services will be loaded/instantiated in the node as the *running* services. Placement/replacement decisions are based on the ranking, for a given metric, of all services in the service population (*i.e.*, set $P$). We define as $R$ the set of services running in a given spot ($|R| << |P|$). To perform the ranking, each computation node collects the following information for each service $s$, namely the number of requests admitted (if $s$ was in $R$ already) and the number of requests rejected and propagated upstream due to congestion (or because $s$ was not instantiated). Also, for each request $r$ for a service $s$, each node computes an *Adequate time* which is equal to the current time plus remaining deadline time minus the completion time (see above). The Adequate time for a service is the average of the Adequate times of each request for that service.

Based on the total number of requests (admitted and rejected) and the average Adequate time of each service, a node uses two different ranking metrics to periodically decide which of them will be included in the running set $R$ (*i.e.*, some will be remained and some will be replaced). Generally, the goal of any placement policy is to keep cores/VMs "busy" (*i.e.*, maximize the use of the available edge-cloud resources) and meet the corresponding request deadlines. Here we will examine the following ranking policies:

- **Strictest Deadline First (SDF)**: The ranking is done based on the Adequate time. SDF ranks higher services whose requests have the least remaining deadline and can still be executed at the node. The node includes in the $R$ set as many services as possible based on the corresponding ranking. In case a service with strict deadline is running at a VM, but there are still rejected requests, the node will initialize more VMs to satisfy the corresponding service before serving other (possibly more popular) services with less strict deadlines (assuming that a VM can process one request at a time).
- **Least Frequently Used (LFU)**: The ranking is done based only on the total number of requests (used as the utilization metric). This metric does not prioritize services with strict deadlines, but it considers placing a service $s$ in $R$, only if requests for $s$ is feasible, *i.e.*, it had arrived with sufficiently large remaining deadlines to allow execution at the node.
- **Hybrid**: The ranking utilizes both the number of requests and the Adequate time. Hybrid first ranks the services by their Adequate time. The node then forms a list of services by eliminating services whose average Adequate time permits delegation to upstream spots/centralized cloud. This list of services is then sorted by their utilization metric (admitted and rejected requests) to obtain a sorted list of services. Hybrid includes in the $R$ list as many of these services as the memory size of the node permits.
- **Least Recently Used (LRU)**: Service replacements are performed according to the least recently used ordering of services.

Using the above ranking policies, each node can perform service placement/replacement procedures independently. This procedure can take place either periodically at specific or random periods or upon the arrival of a request for a given service. Here we will examine both of these approahes. For the per-request placement, we use the LRU ranking policy. Particularly, when a request arrives and the corresponding service is not currently running (*i.e.*, not in set $R$) at the node, the service code is loaded from the secondary storage as a VM (with some boot up delay) and one of the least popular service's VM is replaced. However, the request itself is rejected since the boot up latency typically exceeds the deadline of the request. We assume that the edge-cloud nodes prefetch the service VM kernels (*i.e.*, code) beforehand (possibly from a repository in the central cloud) to prevent extra delays in instantiation.

In the periodic placement schemes, each node reevaluates the services that it is currently running (*i.e.*, set $R$), and at given (*i.e.*, replacement period) or random periods uses one of the ranking policies to perform the placement of the services

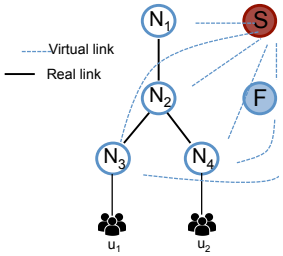

Fig. 2: Auxiliary graph.

in the local memory of each node. These periodic schemes are similar in rationale to the off-line cache replacement schemes found in literature (*e.g.*, [8]).

## IV. Centralized Resource Allocation Strategy

We develop a centralized algorithm working for service placement and routing decisions for predefined time slotted intervals. The algorithm requires centralized execution (i.e., coordination) and the knowledge of all future requests. We formulate the placement and routing problem using mixed integer linear program (MILP). To that end, we first create an auxiliary graph based on the hierarchical cloud model, containing computation spots as the nodes, as shown in Fig. 2. Also, we introduce two additional virtual nodes $S$ and $F$ namely "virtual source" and "virtual fail" nodes, respectively. We add virtual links to connect $S$ to all the other nodes (including $F$) whereas node $F$ only connects to $S$ and the leaf nodes in the tree topology. All the virtual links have zero latency. Based on the auxiliary graph, the MILP model will find paths from the virtual source $S$ to every "user". A user in our formulation refers to a group of users whose network attachment point is near the same leaf node (*i.e.*, computation spot). Each path has a format of the form: *[S, serving node, [set of intermediate nodes], user]*, which is a routing solution to show how a user reaches the node running the required service. This node is the one just after $S$ in the path. For example, a path [$S$, $N_2$, $N_3$, $u_1$] would mean that user $u_1$ will be served at $N_2$ and the route to reach $N_2$ will be [$u_1$ - $N_3$ - $N_2$]. A special path [$S$, $F$, user] (there is no intermediate node) would mean this user request is to fail/rejected as it is served by the "virtual fail" node $F$. The objective of the MILP model is to minimize those failed requests. Only when a user cannot find a serving node within its deadline, the MILP model forces this user to be served at node $F$.

The corresponding MILP objective function is:

$$\min \sum_{ij \in D} D^{ij} x^{ij}_{SF} \tag{1}$$

We use variable $x^{ij}_{Sv}$ to indicate fraction of requests from user $i$ for service $j$ which will be served at node $v$. $D^{ij}$ is the demand volume of user $i$ for service $j$. Note that we allow to have multi-path routing where it is possible to have more than one path from $S$ to a user $i$. For instance: $x^{ij}_{SF} = 0.2$, $x^{ij}_{SN_1} = 0.3$ and $x^{ij}_{SN_2} = 0.5$ would mean the following: for the user $i$ and service $j$: 20% of the requests are to fail, while 30% and 50% of them will be served at node $N_1$ and $N_2$, respectively. By using the objective function (1) we try to minimize the number of failures. Given the objective, we add the following constraints for the formulation.

$$\sum_{u \in V, v \in N(u)} (x^{ij}_{uv} - x^{ij}_{vu}) = \begin{cases} 1 & \text{if } u = S \\ -1 & \text{if } u = i \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Eq. (2) is the flow conservation constraint to guarantee finding paths from the virtual source $S$ to every user $i$. There will be no flow outgoing from the user $i$ and no flow incoming to the virtual source $S$. For intermediate nodes, the outgoing and incoming flows should be equal.

$$\sum_{(ij) \in D} D^{ij} x^{ij}_{Sv} \leq C_v \tag{3}$$

Constraint (3) is a capacity constraint ($C_v$ is the capacity of each node $v$). Each node on the network has limited resources in terms of cores (*i.e.*, VMs to serve requests).

$$y^{ij}_{uv} - x^{ij}_{uv} \geq 0 \tag{4}$$

$$R^j_v y^{ij}_{Sv} + \sum_{uv \in P^i_k} L_{uv} y^{ij}_{uv} \leq T^{ij} \tag{5}$$

We use constraint (4) to force binary variable $y^{ij}_{uv}$ to be 1, if user $i$ and service $j$ use the link $uv$. Using this binary variable, constraint (5) is used to compute service response time for each request $(i, j)$. The response time includes round-trip network latency ($\sum_{uv \in P^i_k} L_{uv} y^{ij}_{uv}$) and service execution time ($R^j_v y^{ij}_{Sv}$) at the node serving the request. In constraint (5), we consider all possible paths ($P^i_k \in P^i$) connecting the virtual source $S$ and the user $i$ and make sure that if a path $P^i_k$ is used, the response time using this path should be less than the deadline $T^{ij}$. It is noted that the special path [$S$, $F$, $i$] always satisfies the deadline since the network latency is zero (all virtual links have zero latency). However, as the objective (1) minimizes the number of failing requests, user requests are assigned to the virtual node $F$ if and only if there are no other available nodes that can serve the requests within their deadlines.

Resource allocation in edge-cloud infrastructure is known to be an NP-hard problem [10], [13], and in our MILP model the complexity depends on the number of possible paths between the virtual source $S$ and the potential users (can be of exponential size).

## V. Performance Evaluation

In this section, we evaluate the performance of LRU, LFU, SDF, and Hybrid strategies, presented in Section III, using a wide range of parameters. We compare the performance of these uncoordinated strategies with the optimal (*i.e.*, centrally coordinated) placement strategy introduced in Section IV. The objective is to evaluate their performance in terms of success in meeting service request deadlines. 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 [15]. Icarus is originally a simulator for evaluating the performance of networks of caches. The simulator code with our modifications and the scripts to generate the results are made publicly available[1]. We simulate the strategies using a binary tree topology with a height of five (*i.e.*, root and four sub-levels). The root of the tree is the back-end cloud for all the services containing infinite 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 autonomous edge-cloud nodes (*i.e.*, computation spots), each performing placement, admission and scheduling in an uncoordinated manner. We uniformly distribute computational resources for the execution of services in terms of number of CPU cores and assign two cores per each node. The small number of cores per computation spot is due to scalability of the simulator; thus, each node in the network is meant to represent a miniature model of a computation spot.

We consider a scenario, where the service population of interest is $10^3$ services. Note that $10^3$ items is not meant to represent all possible edge-cloud services in the Internet. Instead, this set consists of the services that a private edge-cloud can execute within an edge domain (*e.g.*, one or more ISPs). Each service is associated with an execution time and a deadline. In order to generate a deadline for a service (say $s$), we first generate a random time value (in ms) between the following lower- and upper-bounds: RTT from gateway node to its adjacent edge-cloud at the lowest level of the tree (lower-bound), RTT to reach the central cloud from the gateway node (upper-bound). This randomly generated value is added to the execution time of $s$ to generate its deadline. Thus, we consider only latency critical services, whose requests can only be satisfied (*i.e.*, response delay within the deadline) through edge-cloud nodes and not the back-end cloud. That said, we choose a small execution time for each service selected from a range of 1–5 ms, and we assume homogeneous nodes, where requests of the same application require the same execution time.

In our simulations, we ignore the data sizes and the transmission delays; instead we consider propagation delays of links and queuing delays of user requests waiting to be executed. We assign inter-connection latencies as follows: each link connecting two edge-clouds has a propagation delay of 20 ms, the links connecting edge-clouds (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 [16]. The gateway (leaf) nodes connect to the edge-cloud nodes (at the lowest-level of the tree) with a latency of 2 ms and to users with negligible latency. In order to simplify the service size provisioning in the experiments, we assume that each service has the same storage requirement for its instance in the memory. The total storage capacity for the entire edge-cloud network is a parameter in our experiments, and we uniformly distribute the total capacity (*e.g.*, $B$ services) among the edge-cloud nodes. As mentioned before, we assume that each edge-cloud node prefetches and stores service source codes (*e.g.*, Unikernel image) in advance, before instantiation.

[1] github.com/oascigil/icarus_edge_comp

TABLE I: Default evaluation parameters.

| Parameter | Value |
|---|---|
| Number of edge-cloud nodes in the tree | 14 |
| Size of service population $P$ | $10^3$ |
| Total service storage capacity $B$ | $1.0 \cdot P$ |
| Average request rate per second | $10^5$ |
| Zipf exponent (for synthetic workloads) | 0.75 |
| Service Execution Time | 1–5 ms |
| Replacement period | 30s |
| Scheduling Policy | EDF |

Our evaluation is based on the following performance metrics:

- **Mean satisfaction rate** (in percentage of issued requests): The ratio of requests that are executed and returned back to their originating user within their service deadlines.
- **Percent Idle Time**: The ratio of the time that the computational cores are staying idle, *i.e.*, not executing any services. This metric indicates the utilization of resources at the edge-clouds.

During the experiments, we compute these metrics periodically at the end of each *replacement period* using the requests that arrived during the period. As mentioned before, at the end of a replacement period, each edge-cloud reevaluates the set of currently running services and may replace a subset of those based on the replacement strategy. We use a default replacement period of 30 seconds. We first present experiments using synthetic workloads followed by trace-based experiments. In both workloads, user requests are generated in the network with an aggregate mean rate of $10^5$ requests per second, and each request is assigned to a randomly chosen gateway node where it originates from. The association of request to a service type in synthetic workload is generated using a Zipf distribution, which determines the service popularity. We use a default Zipf exponent of $0.75$ for the synthetic workload, and in Section V-E, we consider different Zipf exponents. The rest of the default parameters are listed in Table I.

For our evaluations, we use three synthetic workloads, which have different correlation between service popularity and deadline strictness: *i*) positive, *ii*) negative and *iii*) uncorrelated. In the positive (negative) correlated workload, the most popular (unpopular) services have the strictest deadlines.

### B. Comparison of Strategies

In the leftmost plot of Fig. 3, we depict the satisfaction rate of the strategies using the synthetic workload with uncorrelated service popularities and deadlines. The first data point in this plot and all the other plots correspond to the first replacement period (0-30s), during which *a randomly chosen set of services* are hosted (instantiated at $t = 0$) at each edge-cloud node. We observe that LFU and Hybrid strategies improve their satisfaction rates gradually during the subsequent replacement periods

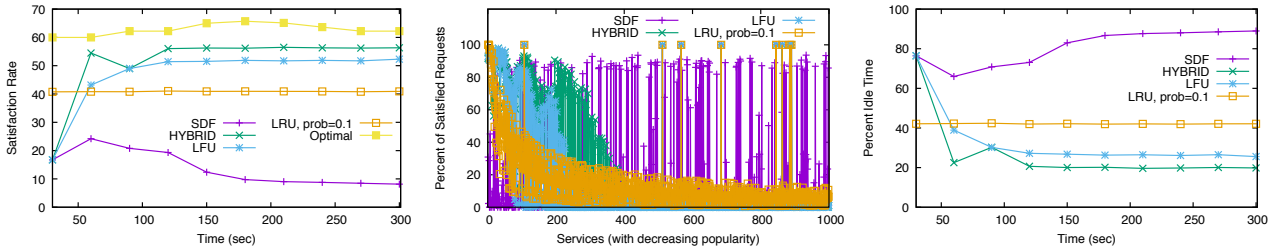


Fig. 3: Performance of strategies with service popularity uncorrelated with deadlines.

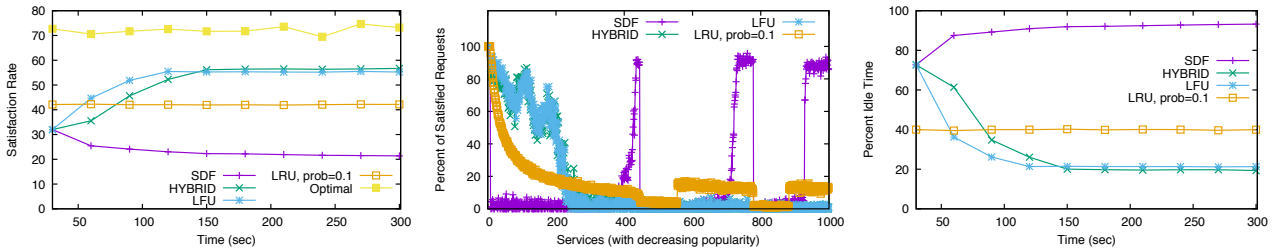


Fig. 4: Performance of strategies with service popularity negatively correlated with deadlines.

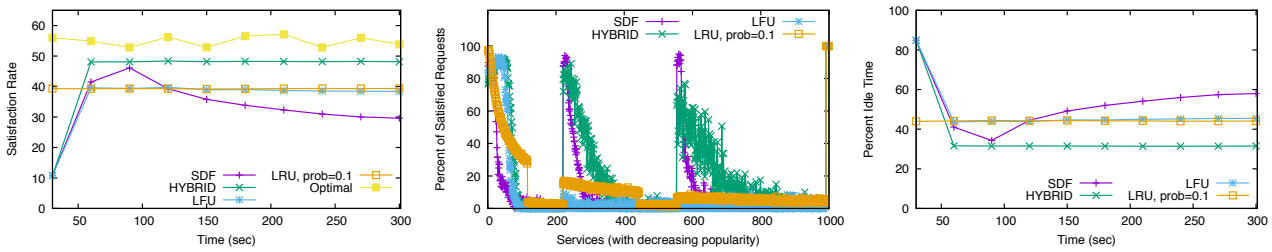


Fig. 5: Performance of strategies with service popularity positively correlated with deadlines.

(*i.e.*, every 30s) with some minor fluctuations in the beginning. In general, such fluctuations occur due to overestimation of demand for some popular services, for which individual edge-cloud nodes at different levels of the tree simultaneously instantiate VMs. Eventually, the nodes closer to the leaves serve increasingly more requests for those services, which filters the requests reaching upstream. This in turn leads to a different observable popularity of services at the nodes at higher levels.

The Hybrid strategy has slightly higher satisfaction rate than the LFU strategy and performs less than $10\%$ worse than the optimal strategy as can be seen in Fig. 3. As opposed to the Hybrid, LFU and SDF strategies that perform service placement periodically, the LRU strategy potentially performs one (re)placement per request, which leads to frequent intervals with VMs booting up and shutting down. As a result, LRU achieves steady satisfaction rate around $40\%$ worse than LFU and Hybrid strategies. In all the experiments, we use a probability of instantiation of $0.1$ for the LRU strategies. A higher probability of instantiation, leads to worse satisfaction rates for LRU (we omit those results due to space limitations).

On the other hand, the SDF strategy is unable to achieve and retain a satisfactory performance. This is mainly because SDF does not consider service popularity. In particular, the SDF strategy instantiates an additional VM for a service (say $s$) with the smallest remaining deadline as long as there is at least one request for $s$ that has been "missed/rejected". This leads to increasing amount of underutilized VMs as can be observed in the idle times metric shown in the rightmost plot in Fig. 3. Conversely, the LFU strategy replaces the VMs of the least utilized services (during the last replacement period) with services that have the highest number of missed requests, and as a result, it attains significantly lower idle times than SDF. The Hybrid strategy also performs similarly and eliminates services with low popularity to increase the utilization.

In the middle plot of Fig. 3, we demonstrate the satisfaction rates of individual services. The services in the $x$-axis are sorted in decreasing order of popularity. As expected, the LRU strategy performs increasingly well for services with increasing popularity. The Hybrid and LFU strategies are more selective in services to instantiate in that they both achieve significantly higher satisfaction rates for the most popular services, while less popular services have negligible satisfaction rates. SDF strategy, on the other hand, achieves high satisfaction rate for services with the least remaining deadlines, but since those services are not necessarily popular, the overall satisfaction rate is low.

In the leftmost plot of Fig. 4, we depict the satisfaction rate of the strategies using the synthetic workload with inverse correlated service popularities and deadlines (*i.e.*, the most popular services have the least strict deadlines). The optimal strategy achieves better performance than in the uncorrelated

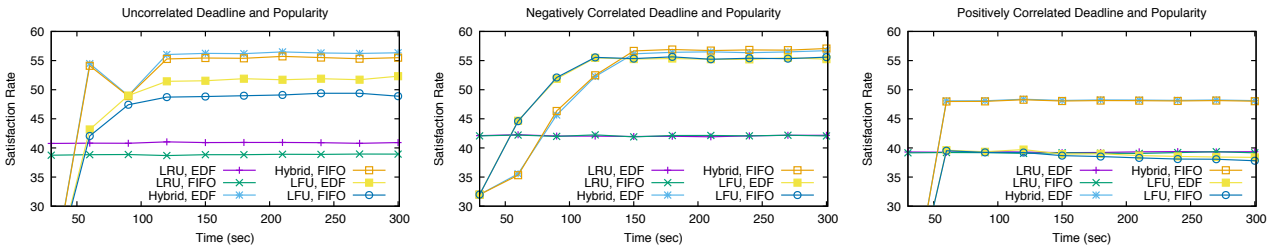


Fig. 6: Performance of LFU, SDF and Hybrid strategies for FIFO and EDF scheduling.

workload, since it is possible to satisfy the deadlines of a higher percentage of requests when there is a larger number of requests for less strict services, *i.e.*, such requests can be successfully executed at one or more edge-cloud nodes along the path leading to the central cloud. The existence of popular services with more flexible deadlines is a challenge for uncoordinated strategies, because the majority of requests can be executed at multiple levels of the tree and instantiating the same set of services at multiple levels can lead to underutilized nodes. Nevertheless, both the Hybrid and LFU strategies achieve a similar satisfaction rate as with the uncorrelated workload. SDF strategy performs slightly better with the negatively correlated workload. As can be seen from the middle plot of Fig. 4, the LFU and Hybrid strategies satisfy the most popular services. SDF, on the other hand, selects services with minimum remaining deadline at each one of the three levels of the tree topology (independent of their popularity), and we observe three peaks corresponding to those three levels.

We depict the performance of the strategies using the synthetic workload with positively correlated popularity and deadlines (*i.e.*, most popular services have the strictest deadlines) in Fig. 5. In this scenario, the majority of the requests can only be successfully executed at the lowest levels of the tree. Therefore, there is an increasing contention for such popular service requests, which leads to a reduction in the overall performance of all the strategies with the exception of SDF, as shown in the leftmost plot of Fig. 5. The Hybrid strategy performs significantly better than the LFU, because Hybrid can prioritize services that are both popular and have the smallest remaining deadlines at each node at various levels of the tree (as can be observed with the three peaks in the middle plot). On the other hand, LFU instantiates only the most popular services (with strict deadlines), which can tolerate only minimal additional queuing delays when the computational cores are busy. As a result, the LFU strategy is not able to admit some of the incoming requests, when the node is already busy processing requests for the same (popular) service.

Overall, the Hybrid strategy performs the best and achieves near optimal results for the uncorrelated and positively correlated workloads, and suffers from coordination problems (*i.e.*, over-replicated instances at different levels of the tree) in the case of the negatively correlated workload. LRU strategy is not affected by correlation in the workload and achieves the same (considerably worse) satisfaction rate.

### C. Impact of Storage Capacity

Here, we examine the impact of the aggregate storage budget (in terms of number of services) on the performance of the uncoordinated replacement strategies. Due to space limitations we omit the corresponding plots, but we report that as expected, all the strategies have increasingly higher satisfaction rates as a result of being able to host a larger number of services (*i.e.*, $50\%$-$60\%$ increase in the satisfaction ratio of the system when increasing total storage capacity from 500 to 1500 services).

### D. Impact of Scheduling Policy

In all the previous experiments, we have used EDF scheduling by default. In this section, we investigate the performance gain from EDF scheduling by comparing it with the simple FIFO for LRU, LFU, and Hybrid strategies using the three types of synthetic workloads in Fig. 6. We observe that all the three strategies achieve better satisfaction rates with EDF scheduling compared to FIFO using the uncorrelated workload (leftmost plot). In both the positively and negatively correlated workloads (middle and rightmost plots, respectively), there is negligible difference in the performance of the strategies with EDF and FIFO. This is expected, because with a correlated workload, the majority of the requests have similar deadlines, and therefore EDF's ordering of requests by their remaining deadlines have little impact on the satisfaction rate.

### E. Impact of Service Popularity Distribution

In the above scenarios, we used a default Zipf exponent value of 0.75 when determining the items popularity. As expected, the higher values of Zipf exponent leads to higher satisfaction rate for all the strategies. The LRU strategy, in particular, performs comparably well with LFU and Hybrid for Zipf exponents close to one. For space considerations, we omit the plots for the experiments with various zipf values. In the experiments so far, we used synthetic workloads with stationary distributions of services in the requests. In the next section, we use a trace-based workload with non-stationary distribution of services.

### F. Trace-based Simulations

We use a data set containing the request arrivals to a Google cluster [17], which contains over three million requests issued in a period of seven hours. Each request is associated with a "ParentID" field that identifies its service, and there are a total of $9218$ unique services in the data set. The most popular service has more than $200K$ requests, while

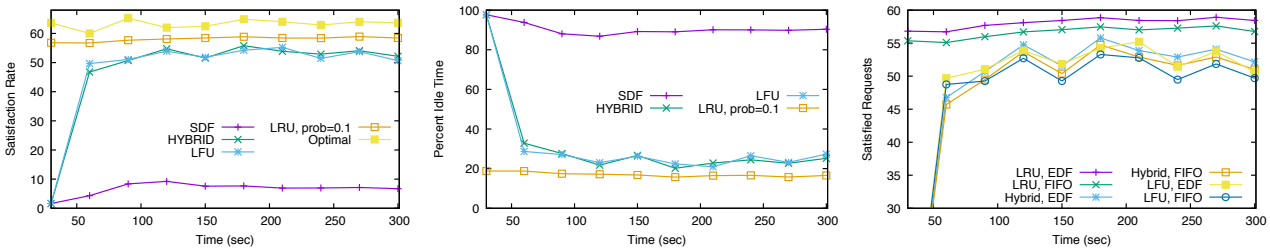


Fig. 7: Performance of strategies using trace-based workload.

5150 services only have one request. Based on this trace, we generate a workload associating each request with a service using the trace data. We randomly assign deadlines to each service without a correlation between popularity and strictness of deadlines. We use the same default request rate of $10K$ requests per second. This results with a total simulation time of around $300s$. As before, we report the popularity of the strategies at each replacement period of $30s$.

LRU strategy achieves the highest overall satisfaction rate among all the strategies as shown in the leftmost plot of Fig. 7. This is because of the non-stationary nature of the trace-based workload. We observe sudden changes in the distribution of requests in the trace data, and LRU can respond to such changes much faster than the other strategies, because it performs service (re)placement with each request as opposed to every $30s$. We have observed better performance for Hybrid and LFU with smaller replacement periods, but we do not provide results for varying replacement periods due to space considerations. The smaller replacement periods also have an additional cost: instantiation (*i.e.*, boot time) of a Unikernel typically takes between 2-400 milliseconds on an ARM processor [18]. During the instantiation, a VM is unable to process requests, and this cost is already included in all our experiments by keeping the VM idle for a random period of time (randomly chosen in the interval $[2-400]$ ms). Similar to our previous results in Section V-D, we observe that the EDF scheduling improves the satisfaction rates of all the strategies as can be seen in the rightmost plot of Fig. 7.

## VI. Conclusions

Inspired by the extensively studied field of distributed cache resource management, we investigated several uncoordinated resource allocation/management strategies in an edge-cloud for the deployment of latency-critical services. Specifically, the edge-cloud nodes with computing power act autonomously and make individual resource allocation decisions, and user service requests are processed opportunistically at one of the nodes along the shortest path to the back-end cloud. We have proposed resource allocation strategies to carry out user request admission, scheduling, and (re)placement of service VMs (to cater for the changing user demand) in order to maximize the QoS experienced by users. We modeled QoS requirements of latency-critical services with upper-bounds (*i.e.*, deadlines) on the response delay experienced by users, which is the time from the submission of a request until the results are returned.

We have formulated a centralized algorithm for the optimal resource allocation. We have shown with both synthetic and trace-based workloads that uncoordinated resource allocation strategies can achieve near optimal performance, which comes with no communication or coordination overhead as opposed to optimal, centralized solutions. The uncoordinated solution only requires each node to monitor the RTT delay to upstream nodes, and deduct this delay from the deadline of a request placed in the packets when forwarding requests upstream.

## Acknowledgment

This work has been supported by the EC H2020 UMOBILE project (GA no. 645124), the EC H2020 ICN2020 project (GA no. 723014), the EPSRC INSP fellowship (EP/M003787/1), and EPSRC CHIST-ERA CONCERT project (number I1402).

## References

[1] J. Cho *et al.*, "Acacia: Context-aware edge computing for continuous interactive applications over mobile networks," in *ACM CoNEXT*, 2016.
[2] K. Ha *et al.*, "Towards wearable cognitive assistance," in *International conference on Mobile systems, applications, and services*. ACM, 2014.
[3] M. Satyanarayanan *et al.*, "Edge analytics in the internet of things," *IEEE Pervasive Computing*, 2015.
[4] ——, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, 2009.
[5] M. Patel *et al.*, "Mobile-edge computing introductory technical white paper," *Mobile-edge Computing (MEC) industry initiative*, 2014.
[6] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *ACM SIGCOMM CCR*, 2014.
[7] B. Confais *et al.*, "Performance analysis of object store systems in a fog/edge computing infrastructures," in *IEEE Cloudcom*, 2016.
[8] 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.
[9] A. Ceselli *et al.*, "Cloudlet network design optimization," in *IFIP NETWORKING*, 2015.
[10] L. Tong *et al.*, "A hierarchical edge cloud architecture for mobile computing," in *IEEE INFOCOM*, 2016.
[11] 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.
[12] F. Hao *et al.*, "Online allocation of virtual machines in a distributed cloud," *IEEE/ACM Trans. Netw.*, 2017.
[13] B. Yang *et al.*, "Seamless support of low latency mobile applications with nfv-enabled mobile edge-cloud," in *IEEE Cloudnet*, 2016.
[14] I.-H. Hou *et al.*, "Asymptotically optimal algorithm for online reconfiguration of edge-clouds." in *MobiHoc*, 2016, pp. 291–300.
[15] 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.
[16] Q. Duan, *Delay Characteristics of Packet Switched Networks*. Boston, MA: Springer US, 2010, pp. 203–223.
[17] J. L. Hellerstein, "Google cluster data. google research blog, jan. 2010." https://research.googleblog.com/2010/01/google-cluster-data.html.
[18] M. Skjegstad, "Just-in-time summoning of unikernels (v0.2)," http://www.skjegstad.com/blog/2015/08/17/jitsu-v02/.