POLITECNICO DI TORINO
Repository ISTITUZIONALE

Distributed Asynchronous Protocol for Service Provisioning in the Edge-Cloud Continuum

Original
Distributed Asynchronous Protocol for Service Provisioning in the Edge-Cloud Continuum / Cohen, I., Giaccone, P., Chiasserini, C.F.. - STAMPA. - (2023). (International Conference on Software, Telecommunications and Computer Networks (SoftCOM 2023) Split (Croatia) 21-23 September 2023) [10.23919/SoftCOM58365.2023.10271640].

Availability:
This version is available at: 11583/2979419 since: 2023-06-18T16:06:47Z

Publisher:
IEEE

Published
DOI:10.23919/SoftCOM58365.2023.10271640

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

25 November 2026

Asynchronous Distributed Protocol for Service Provisioning in the Edge-Cloud Continuum

Itamar Cohen
Ariel University, Israel

Paolo Giaccone
Politecnico di Torino, Italy

Carla Fabiana Chiasserini
Politecnico di Torino, Italy

***Abstract*—In the edge-cloud continuum, datacenters provide microservices (MSs) to mobile users, with each MS having specific latency constraints and computational requirements. Deploying such a variety of MSs matching their requirements with the available computing resources is challenging. In addition, time-critical MSs may have to be migrated as the users move, to keep meeting their latency constraints. Unlike previous work relying on a central orchestrator with an always-updated global view of the available resources and of the users' locations, this work envisions a distributed solution to the above issues. In particular, we propose a distributed asynchronous protocol for MS deployment in the cloud-edge continuum that (i) dramatically reduces the system overhead compared to a centralized approach, and (ii) increases the system stability by avoiding having a single point of failure as in the case of a central orchestrator. Our solution ensures cost-efficient feasible placement of MSs, while using negligible bandwidth.**

***Index Terms*—Edge computing, 5G mobile communication.**

I. Introduction

Today's networks offer bulk virtualized resources, embodied as a collection of datacenters on the continuum from the edge to the cloud [1]–[4]. These datacenters host a plethora of applications with versatile computational requirements and latency constraints. For example, time-critical services such as road safety applications require low latency, necessitating processing them in an edge datacenter, close to the user. In contrast, infotainment tasks require larger computational resources, but looser latency constraints and therefore may be placed on a cloud datacenter with abundant and affordable computation resources [2], [5]. Placing services over the cloud-edge continuum is thus challenging. It becomes even more complex when changes in the users' mobility or traffic demand require migrating services to reduce latency.

Most of the existing solutions [1]–[3], [6]–[9] rely on a central orchestrator to make all placement and migration decisions. The orchestrator periodically (i) gathers information about the state of resources and migration requirements, (ii) calculates new placement and resource allocation, and (iii) instructs datacenter local controllers accordingly. This centralized synchronous approach has several shortcomings. First, it does not scale well, thus failing to manage systems with multiple datacenters efficiently. In practice, gathering fresh state information causes significant communication bottlenecks, even within a single cloud datacenter [10]. Secondly, the orchestrator is a natural single point of failure, compromising the system's stability. Finally, the datacenters may be operated by distinct operators [11], [12], which are typically unwilling to share proprietary information and implementation details with competitors.

Our Contribution. We present a solution called Distributed Asynchronous Placement Protocol for the Edge-Cloud Continuum (DAPP-ECC), which overcomes the aforementioned issues effectively and efficiently. DAPP-ECC decreases communication overhead by using simple single-hop control messages transmitted by a node to only relevant neighbors. Moreover, DAPP-ECC requires no out-of-band communication or synchronization tools. DAPP-ECC can find a feasible solution even with restricted resources, where a feasible placement necessitates migrating also already-placed MSs. Finally, very importantly, our solution allows multiple datacenters – possibly of distinct providers – to cooperate without exposing proprietary information.

Paper organization. We introduce the system model in Sec. II and formulate the placement and migration problem in Sec. III. Sec. IV describes our algorithmic solution. Sec. V evaluates the performance of DAPP-ECC in various settings, using real-world mobility traces and antenna locations. Finally, Sec. VI reviews relevant related work, and Sec. VII draws some conclusions.

II. System Model

We consider a fat-tree cloud-edge continuum architecture, which comprises [5]: (i) a set of *datacenters*, $\mathcal{S}$, denoting generic computing resources, (ii) *switches*, and (iii) *radio Points of Access (PoA)*. Datacenters are connected through switches, and PoAs have a co-located datacenter [13]. Each user is connected to the network through a PoA, and they can change PoA as they move.

We model such logical multi-tier network as a directed graph $\mathcal{G} = (\mathcal{S}, \mathcal{L})$, where the vertices are the datacenters, while the edges are the directed virtual links connecting them. We assume the existence of a single predetermined loop-free path between each pair of datacenters.

Let us consider a generic user generating a *service request* r, originating at the PoA p^r, to which the user is currently connected. Each request is served by placing an instance of

C. F. Chiasserini and P. Giaccone are also with CNIT, Parma, Italy. This work was supported by the European Union's NextGenerationEU instrument, under the Italian National Recovery and Resilience Plan (NRRP), M4C2 Investment 1.3, "Telecommunications of the Future" (PE00000001), program "RESTART", and by the European Commission through Grant No. 101095890 (Horizon Europe SNS JU PREDICT-6G project).

a microservice (MS) on a datacenter. Denote the instance of the MS for service request r by $\mathbf{m}^r$. Let $\mathcal{R}$ denote the set of service requests, and $\mathcal{M}$ the set of corresponding MSs that are currently placed, or need to be placed, on datacenters.

Each service is associated with an SLA, which specifies its requirements in terms of KPI target values [14]. Let us consider *latency* as the most relevant KPI, although our model could be extended to others, like throughput and energy consumption. Due to these latency constraints, each request r is associated with a list of *delay-feasible* datacenters $\mathcal{S}_r$. The delay-feasible servers in $\mathcal{S}_r$ are not too far from r's PoA (p^r), or, more formally, their list is a prefix of the path from p^r to the root [1], [4], [6]. The top delay-feasible datacenter of request r is denoted by $\bar{s}_r$.

To successfully serve request r on datacenter $s \in \mathcal{S}_r$, s should allocate (at least) $\beta^{r,s}$ CPU units, where $\beta^{r,s}$ is an integer multiple of a basic CPU speed. As there exists a known method to calculate $\mathcal{S}_r$ and $\beta^{r,s}$ given the characteristics of $\mathbf{m}^r$ [6], we refer to $\mathcal{S}_r$ and $\beta^{r,s}$ as known input parameters. Each datacenter $s \in \mathcal{S}$ has a total processing capacity C_s, expressed in the number of CPU cycles/s.

III. The Placement and Migration Problem

The delay experienced by an MS may vary over time due to either (i) a change in the user's PoA, which changes the network delay, or (ii) a fluctuation in the traffic, and hence in the processing latency [15]. Each user continuously monitors its Quality of Experience (QoE) and warns its PoA as its latency approaches the maximum acceptable value[1]. The PoA then checks the request, and if the user is indeed *critical* – namely, its latency constraint is about to be violated – the PoA triggers a migration algorithm. The PoA also handles new requests awaiting service.

Decision variables. Let $\mathbf{y}$ be the Boolean placement decision variables, i.e., $y(r,s) = 1$ if MS $\mathbf{m}^r$ is scheduled to run on datacenter s. Any choice for the values of such variables provides a *solution* to the *Placement and Migration Problem (PMP)*, determining (i) where to deploy new MSs, (ii) which existing MSs to migrate, and (iii) where to migrate them.

Constraints. The following constraints hold:

$$\sum_{s\in\mathcal{S}} y(r,s) = 1 \qquad \forall r \in \mathcal{R} \qquad (1)$$

$$\sum_{r\in\mathcal{R}} y(r,s)\beta^{r,s} \le C_s \qquad \forall s \in \mathcal{S}\,. \qquad (2)$$

Constraint (1) ensures that at any point in time, each MS $\mathbf{m}^r$ is associated with a single *scheduled* placement. (2) assures that the capacity of each datacenter is not exceeded.

Costs. The system costs are due to migration and computational resource usage, as detailed below.

Migrating MS $\mathbf{m}^r$ from datacenter s to datacenter s' incurs a *migration cost* $\psi^m(r,s,s')$. Let $x(r,s)$ denote the *current placement indicator parameters*[2], i.e., $x(r,s) = 1$ iff MS $\mathbf{m}^r$ is currently placed on datacenter s. We assume that a user does not become critical again before it finishes being placed based on the decision made by any previous run of the algorithm solving the PMP. The migration cost incurred by a critical MS $\mathbf{m}^r$ is then:

$$\sum_{s\neq s'\in\mathcal{S}} x(r,s)\cdot y(r,s')\cdot \psi^m(r,s,s').$$

Placing MS r on datacenter s incurs a *computational cost* $\psi^c(r,s)$. As computation resources in the cloud are cheaper [2], [5], we assume that if s is an ancestor of s', placing MS $\mathbf{m}^r$ on s is cheaper than placing $\mathbf{m}^r$ on s'.

Objective. Our goal is to minimize the cost function:

$$\phi(\mathbf{y}) = \sum_{s\neq s'\in\mathcal{S}}\sum_{r\in\mathcal{C}} x(r,s)\cdot y(r,s')\cdot\psi^m(r,s,s') + \sum_{s\in\mathcal{S}} y(r,s)\sum_{r\in\mathcal{C}}\psi^c(r,s) \qquad (3)$$

subject to constraints (1), (2).

Following Proposition 2 in [6], it is easy to see that PMP is NP-hard. We are interested in a *distributed* solution, where no single datacenter (or any other entity) has a complete fresh view of the status (e.g., the current place of each MS, or the amount of available resources in each datacenter). Instead, the placement and migration protocol should run on an as small as possible subset of the datacenters. Furthermore, the solution should be *asynchronous*, as distinct PoAs may independently invoke different, simultaneous runs of the protocol.

IV. The DAPP-ECC Algorithmic Framework

In this section, we present our algorithmic solution to PMP, named Distributed Placement Protocol for the Edge-Cloud Continuum (DAPP-ECC). We start with a high-level description and then provide the details of the single algorithms. In our description, we let s.proc() denote a run of procedure proc() on datacenter s. As our protocol is distributed, each datacenter s maintains its local variables, denoted by a subscript s. We will use the procedure *Sort()* that sorts MSs in non-increasing timing criticality, realized by a non-decreasing $|\mathcal{S}_r \setminus \mathcal{T}(s)|$, i.e., the number of ancestor datacenters on which the MS may be placed. *Sort()* breaks ties by non-decreasing $\beta^{r,s}$ and breaks further ties by users' FIFO order.

A. Protocol overview

Following the intuition, one would reduce the system costs by placing MSs in the network continuum as close as possible to the cloud, since cloud resources are cheaper and this may prevent future migrations. However, such an approach may make the algorithm fail to find feasible solutions, even when they exist [6].

Our solution to this conflict between feasibility and cost-efficiency is inspired from [6]. The proposed DAPP-ECC algorithm initially *assigns* – or, better, reserves – CPU for each request *as close as possible to the edge*. We dub this stage Seek a Feasible Solution (SFS). Once such a solution is found, the protocol Pushes Up (PU) the MSs *towards the cloud* as much as possible, to reduce costs. If SFS cannot find a feasible solution, non-critical MSs will be migrated via the Push-Down (PD) procedure, to make room for a critical MS.

[1] If the user can predict its near-future location, it can inform the PoA before the target delay is violated.

[2] $x(r,s)$ are not decision variables, as they indicate the current deployment.

B. The DAPP-ECC algorithms

We now detail the algorithmic framework we developed. We will denote by $\tilde{\mathcal{M}}$ a list of currently unassigned requests, and by $\mathcal{P}^U$ a list of assigned requests that may be pushed-up to a closer-to-the-cloud datacenter, to reduce costs. Let $\mathcal{P}^D$ denote a set of push-down requests. a_s denotes the available capacity on datacenter s. Upon system initialization, each datacenter s assigns $\tilde{\mathcal{M}}_s = \mathcal{P}^U_s = \mathcal{P}^D_s = \emptyset$ and $a_s = C_s$.

Seek for a feasible solution: s.SFS() is presented in Alg. 1. It handles the unassigned MSs as follows. If the locally available capacity suffices to locally place an unassigned MS $\mathbf{m}^r$ (Ln. 4), s reserves capacity for $\mathbf{m}^r$ (Ln. 6). If $\mathbf{m}^r$ cannot be placed higher in the tree (Ln. 7), s.SFS() not only assigns $\mathbf{m}^r$, but also locally places it. Otherwise, the procedure inserts $\mathbf{m}^r$ to the set of potentially-placed MSs, which s will later propagate to its parent. If s.SFS() fails to place a request r that cannot be placed higher, it calls s.PD() (Lines 12-15). The arguments for s.PD() are (i) the identity of the initiator datacenter, s^*; (ii) a list of MSs that s^* asks its descendants to push-down, and (iii) deficitCPU, namely, the amount of CPU resources that must be freed from s^* to find a feasible solution.

In Lines 16-17, s.SFS() checks whether there exist MSs that are not yet assigned, or may be pushed-up to an ancestor. If so, s.SFS() initiates a run of SFS() on s's parent (Ln. 18). If there are no pending push-up requests from its ancestors, s initiates a push-up (Lines 20-21).

Algorithm 1 s. SFS($\tilde{\mathcal{M}}, \mathcal{P}^U$)

1: $\mathcal{P}^U_s \leftarrow \mathcal{P}^U_s \cup \mathcal{P}^U$
2: $\tilde{\mathcal{M}}_s \leftarrow$ Sort ($\tilde{\mathcal{M}}_s \cup \tilde{\mathcal{M}}$)
3: **for each** $\mathbf{m}^r \in \tilde{\mathcal{M}}_s$ **do**
4: **if** $a_s \geq \beta^{r,s}$ **then** ▷ enough available CPU to place $\mathbf{m}^r$ on s
5: remove $\mathbf{m}^r$ from $\tilde{\mathcal{M}}_s$
6: $a_s \leftarrow a_s - \beta^{r,s}$ ▷ assign $\mathbf{m}^r$ on s
7: **if** $\bar{s}_r = s$ **then** ▷ must place $\mathbf{m}^r$ on s for a feasible sol
8: place $\mathbf{m}^r$ on s
9: **else** ▷ Will place $\mathbf{m}^r$ on s only if it won't be pushed-up
10: insert $\mathbf{m}^r$ to s.potentiallyPlacedRequests
11: insert $(\mathbf{m}^r, s)$ to $\mathcal{P}^U_s$
12: **else if** $\bar{s}_r = s$ **then** ▷ $\mathbf{m}^r$ can't be placed here, nor higher in the tree
13: $\mathcal{P}^D_s \leftarrow \left\{ r \mid \mathbf{m}^r \in \tilde{\mathcal{M}}_s \textbf{ and } \bar{s}(r) = s \right\}$ ▷ "over-provisioned" MSs
14: $\text{deficitCpu} = \sum_{r \in \mathcal{P}^D_s} \beta^{r,s} - a_s$ ▷ capacity to free for finding a sol
15: run PD $(\mathcal{P}^D_s, \text{deficitCpu}, s)$
16: $\mathcal{P}^U_{prnt} \leftarrow \left\{ (\mathbf{m}^r, s') | (\mathbf{m}^r, s') \in \mathcal{P}^U_s \textbf{ and } s.\text{prnt} \in \mathcal{S}_r \right\}$
17: **if** $\tilde{\mathcal{M}}_s \neq \emptyset$ **or** $\mathcal{P}^U_{prnt} \neq \emptyset$ **then** ▷ is there any req to send to parent?
18: send $\left(s.\text{parent, SFS}(\tilde{\mathcal{M}}_s, \mathcal{P}^U_{prnt}) \right)$
19: $\mathcal{P}^U_s = \mathcal{P}^U_s \setminus \mathcal{P}^U_{prnt}$
20: **if** $\mathcal{P}^U_{prnt} = \emptyset$ **then** ▷ No pending PU replies from parent
21: run s.PU$\left(\mathcal{P}^U_s\right)$

Push-Up: s.PU(), detailed in Alg. 2, first displaces and regains the CPU resources for all the MSs pushed-up from s to a higher-level datacenter. Next, s.PU() handles all the push-up requests as follows. Consider a request to push up MS $\mathbf{m}^r$, currently placed on datacenter s' that is a descendent of s. If s has enough available capacity for that request, then s.PU() locally places $\mathbf{m}^r$ (Lines 4-5) and updates the relevant record in $\mathcal{P}^U_s$ (Ln. 6). This record will later be propagated to s' which will identify that $\mathbf{m}^r$ was pushed up, and regain the resources allocated for it. In Lines 7-11, s propagates each push-up request to the child c that is delay-feasible for the MS in question.

Algorithm 2 s.PU($\mathcal{P}^U_s$)

1: dis-place all MSs pushed-up from me, and update a_s and $\mathcal{P}^U_s$ accordingly
2: $\mathcal{P}^U_s \leftarrow$ Sort ($\mathcal{P}^U_s$)
3: **for each** $(\mathbf{m}^r, s') \in \mathcal{P}^U_s$ **do**
4: **if** $a_s \geq \beta^{r,s}$ **then**
5: place $\mathbf{m}^r$ on s and decrease a_s accordingly
6: remove $(\mathbf{m}^r, s')$ from $\mathcal{P}^U_s$ and insert $(\mathbf{m}^r, s)$ into $\mathcal{P}^U_s$
7: **for each** child c **do**
8: $\mathcal{P}^U_c = \left\{ (\mathbf{m}^r, s') \mid (\mathbf{m}^r, s') \in \mathcal{P}^U_s \textbf{ and } c \in \mathcal{S}_r \right\}$
9: **if** $\mathcal{P}^U_c \neq \emptyset$ **then**
10: send $\left(c, \text{PU}(\mathcal{P}^U_c) \right)$
11: $\mathcal{P}^U_s = \mathcal{P}^U_s \setminus \mathcal{P}^U_c$

Algorithm 3 Datacenter s called by PD($\mathcal{P}^D_s, \text{deficitCpu}, s^*$)

1: **if** $s^*_s \neq$ null **and** $s^*_s \neq s^*$ **then** ▷ Running another push-down procedure
2: send (caller, PD($\mathcal{P}^D_s$, deficitCpu, s^*))
3: return
4: $s^*_s \leftarrow s^*$
5: $\tilde{\mathcal{M}}_s \leftarrow$ Sort ($\tilde{\mathcal{M}}_s$)
6: add s.potPlacedRequests, and then s.placedRequests to the end of $\mathcal{P}^D_s$
7: **for each** child c **do**
8: **if** I can push-down to myself enough MSs from s^* to nullify deficitCpu **then**
9: break
10: $\mathcal{P}^D_c = \left\{ \mathbf{m}^r \mid r \in \mathcal{P}^D_s, c \in \mathcal{S}_r \right\}$ ▷ MSs to push-down relevant to child c
11: **if** $\mathcal{P}^D_c \neq \emptyset$ **then** ▷ child c may help in freeing space
12: $\mathcal{P}^D_c, \text{deficitCpu} = c.\text{PD}\ (\mathcal{P}^D_c, \text{deficitCpu}, s^*)$ ▷ req. c & get ack
13: dis-place MSs pushed-down from me; update a_s, $\mathcal{P}^D_s$, $\tilde{\mathcal{M}}_s$ accordingly
14: **for each** $r \in \mathcal{P}^D_s$ s.t. r.curPlace is s or an ancestor of s **and** $\beta^{r,s} \leq a_s$ **do**
15: place $\mathbf{m}^r$ and update a_s, $\mathcal{P}^D_c$ and deficitCpu accordingly
16: **if** $s \neq s^*_s$ **then** ▷ I'm not the initiator of this reshuffle
17: send $\left(s.\text{parent}\ (\mathcal{P}^D_s, \text{deficitCpu}, s^*_s) \right)$
18: run s.SFS($\tilde{\mathcal{M}}_s$) in F-mode
19: $s^*_s \leftarrow$ null

Push-Down: s.PD(), in Alg. 3, runs the same when either a parent calls its child, or vice versa. If no PD procedure is currently handled by s, then s^*=null. Else, at any time instant, each run of PD is unequivocally identified by its initiator s^*. Each such run is associated with a single current value of deficitCpu. PD runs sequentially, in a DFS manner, in the sub-tree rooted by s^*, and terminates once deficitCPU is nullified.

If s.PD() is invoked while s takes part in another run of PD() (realized by a different initiator s^*), the procedure replies with the minimal data necessary to retain liveness (Lines 1-3). Otherwise, s.PD() adds to the given set of requests $\mathcal{P}^D_s$ its locally assigned MSs. To reduce the number of migrations, the locally assigned MSs are added to the end of $\mathcal{P}^D_s$, so that the procedure will migrate already-placed MSs only if necessary for finding a feasible solution. In Lines 7-13, s serially requests its children to push-down MSs, to free space in s^*. The amount of space to be freed from s^* is deficitCpu. Before each such call, s checks whether nullifying deficitCpu without calling an additional child is possible. If the answer is positive, s skips calling its children (Lines 8-9). Upon receiving a reply from child c, the procedure updates deficitCpu and s's state variables according to the MSs that were pushed-down to c's sub-tree (Lines 11-13). In Lines 14-15, s.PD() tries to push-down to s MSs from the push-down list, $\mathcal{P}^D_s$. Later, if s is not the initiator of this push-down procedure, it calls its parent

(Lines 16-17). Finally, s.PD() calls SFS in F-mode (described below) to place all its yet-unassigned MSs, if such exist.

The following theorem assures the convergence of DAPP-ECC. The full proof is omitted due to space constraints.

Theorem 1. *If there are no new requests, the protocol either fails or finds a feasible solution after exchanging a finite number of messages.*

Proof sketch. In the worst case, each request $r \in \mathcal{R}^*$ initiates a distinct run of DAPP-ECC, and each such run involves runs of SFS(), PU(), and PD(). Each run of SFS(), PU() and PD() exchanges a finite number of messages. After all these runs, the algorithm either fails or successfully places at least one additional request. □

C. Reducing the communication overhead

F-mode. Intuitively, a run of s.PD() indicates that a recent run of SFS() – either in s, or in an ancestor of s – failed, and hence called PD(). In such circumstances, there is a high risk of failing again, and therefore finding a feasible solution takes precedence over cost reduction: it does not make sense to push-up MSs just to push them back down slightly later. Hence, we define an *F* (feasibility)-mode of the protocol. Each time s.PD() is called, s enters F-mode (if it was not already in F-mode), and remains so for some pre-configured *F-mode period*. While in F-mode, DAPP-ECC does not initiate new push-up requests, and only replies to existing push-up requests with the minimum necessary details to prevent deadlocks. If s.SFS() does not find a feasible solution while s is in F-mode, DAPP-ECC terminates with a failure.

Accumulation delay. Theoretically, each attempt to place a single MS may result in a unique run of PD that involves all the datacenters, thus incurring excessive overhead. To avoid such case, observe that, typically, several users move together in the same direction (e.g., cars moving simultaneously on the road, on the same trajectory). Naively, such a scenario may translate to multiple invocations of DAPP-ECC, each of them for placing a single request. To tackle this problem, we introduce short *accumulation delays* to our protocol. We let each datacenter receiving a SFS message wait for a short *SFS accumulation delay* before it begins handling the new request. To deter long service delays or even deadlocks, each datacenter maintains a single SFS accumulation delay timer that operates as follows: if a run of SFS reaches Ln. 3 in Alg. 1 while no SFS accumulation delay timer is ticking, the procedure initiates a new SFS accumulation delay timer. This current run of SFS, as well as all the subsequent runs, halt. After the SFS accumulation delay terminates, only a single SFS process resumes (see Alg. 1, Ln. 3).

Likewise, to initiate fewer runs of PD(), we let each datacenter retain a single PD accumulation delay mechanism that works similarly to the SFS accumulation delay timer. Significantly, the accumulation delay only impacts the time until the protocol finds a new feasible placement, not the delay experienced by applications in the data plane. We assess the impact of the accumulation delay in Sec. V-B.

V. Numerical Evaluation

A. Simulation settings

Service area, network, and datacenters. We consider two mobility traces, representing real-world scenarios with distinct characteristics: the vehicular traffic within the centers of the cities of (i) Luxembourg [16], and (ii) the Principality of Monaco [17]. For the PoAs, we rely on real-world antenna locations, publicly available in [18]. For each simulated area, we consider the antennas of the cellular telecom provider having the largest number of antennas in the simulated area. For both traces, we consider the 8:20-8:30 am rush hour period. Further details about the mobility traces can be found in [6], [16], [17].

Network and datacenters. The cloud-edge continuum is structure as a 6-height tree; a topology level is denoted by $\ell \in \{0, 1, \ldots, 5\}$. The leaves (level 0) are the datacenters co-located with the PoAs (antennas). Similarly to [8], [9], [12], the higher levels $5, 4, 3, 2, 1$ recursively partition the simulated area. In both Luxembourg and Monaco, if no PoAs exist in a particular rectangle, the respective datacenters are pruned from the tree. The CPU capacity increases with the level ℓ to reflect the larger computational capacity in datacenters closer to the cloud. Denoting the CPU capacity at each leaf datacenter by C_{cpu}, the CPU capacity in level ℓ is $(\ell + 1) \cdot C_{\text{cpu}}$.

Services and costs. Each vehicle that enters the considered geographical area is randomly marked as requesting either real time (RT) or non-RT services, with some probability defined later. We calculate $\mathcal{S}_r$, $\beta^{r,s}$, and $\psi^c(r, s)$ for each $r \in \mathcal{R}, s \in \mathcal{S}$ using the GFA algorithm and the same data-plane latency parameters as in [6]. We thus obtain the following values. Each RT request can be placed on levels 0, 1, or 2 in the tree, requiring CPU of 17, 17, and 19 GHz, associated with costs of 544, 278, and 164, respectively. Each non-RT request can be placed on any level, with a fixed allocated CPU of 17 GHz and associated costs of 544, 278, 148, 86, 58, and 47 for placing the MS on levels 0, 1, 2, 3, 4 and 5, respectively. The migration cost is $\psi^m(r, s, s') = 600$ for every request r and datacenters s, s'.

Delays. The delay experienced by each packet consists of (i) transmission delay and (ii) propagation delay.

The transmission delay is calculated as the packet's size over the capacity allocated for the control plane at each link, through a dedicated network slice. We assume that this capacity is 10 Mbps. We now detail the size of each field in the messages exchanged by DAPP-ECC. As DAPP-ECC uses only single-hop packets, we assume a fixed 80-bits header. The IDs of datacenters, and requests, are represented using 12-bits, and 14-bits. Each MS belongs to a concrete *class* of timing constraint, expressed through a 4-bit classId. The CPU allocation of an MS on a datacenter $\beta^{r,s}$ is represented through a 5-bits field. deficitCpu is at most the highest capacity of any single datacenter; we assume that this requires 16 bits. For the propagation delay, we use a pessimistic approach, where the length of every single link in the network corresponds to the diameter of the simulated area, and the propagation speed

is $2 \cdot 10^8$ ms^{-1}. Consequently, the propagation delay of each link in Luxembourg and Monaco is 22 μs and 8 μs (resp.). For a datacenter at level ℓ, SFS accumulation delay, and PD accumulation delay are $(\ell+1) \cdot T_{ad}^{SFS}$, and $(\ell+1) \cdot T_{ad}^{PD}$ (resp.). We assign $T_{ad}^{SFS} = 0.1$ ms and $T_{ad}^{PD} = 0.4$ ms. F-mode period (recall Sec. IV-C) is 10 s.

Benchmark algorithms. We are unaware of any fully distributed, asynchronous algorithm for the PMP. Hence, we consider *centralized* placement schemes that identify the currently critical and new users once in a second and solve the respective PMP. We will consider the following algorithms.

Lower Bound (LBound): An optimal solution to the PMP that can place fractions of an MS on distinct datacenters. Also, the LP formulation considers all MSs in the system every 1 s period, not just critical MSs. Hence, it serves as a lower bound on the cost of any feasible solution to the problem.

F-Fit: It places each request r on the lowest datacenter in $\mathcal{S}_r$ that has sufficient available resources to place MS $\mathbf{m}^r$. This is an adaptation to our problem of the placement algorithm proposed in Sec. IV.B in [19].

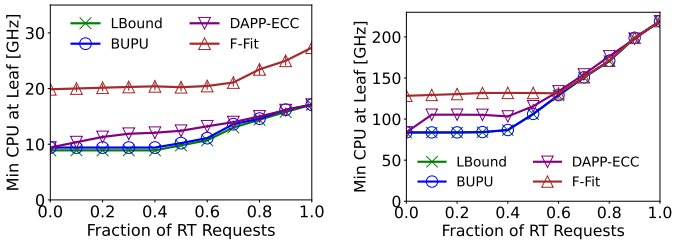
BUPU [6]: It consists of two stages. At the *bottom-up*, it places all the critical and new MSs as low as possible. If this stage fails to place an MS while considering datacenter s, the algorithm re-places *all* the MSs associated with s's sub-tree from scratch. Later, BUPU performs a push-up stage similar to our DAPP-ECC's PU() procedure.

Simulation methodology. We simulate users' mobility using SUMO [20]. The benchmark algorithms use the Python code publicly available in [21]. DAPP-ECC is implemented using OMNeT++ network simulator [22]. Each new user, or an existing user which becomes critical, invokes a run of DAPP-ECC datacenter co-located with the user's PoA. DAPP-ECC's code is available in [23]. LBound is computed using Gurobi optimizer [24].

B. Resources required for finding a feasible solution

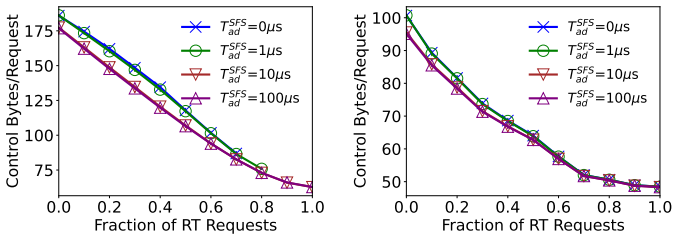
We now study the amount of resources each algorithm requires to find a feasible solution. We vary the fraction of RT MSs. For each setting, a binary search is used to find the minimum amount of resources needed by the algorithm in question to successfully place all the critical MSs along the trace. Fig. 1 presents the results of this experiment. The amount of resources required for obtaining a feasible solution consistently increases with the ratio of RT chains, as tighter delay constraints dictate allocating more resources closer to the edge. This phenomenon is especially noticeable in Monaco, where edge resources are scarcer (only 231 leaf datacenters in Monaco, compared to 1,524 in Luxembourg), thus forcing *any* solution (even LBound) to use excessive resources.

For each concrete setting, the amount of CPU required by BUPU is almost identical to LBound. Despite being fully distributed and asynchronous, the amount of CPU needed by DAPP-ECC is only slightly higher than BUPU. Finally, F-Fit requires a processing capacity that is 50% to 100% higher than LBound to provide a feasible solution.



(a) Luxembourg (b) Monaco

Fig. 1: Minimum required processing capacity for finding a feasible solution when varying the ratio of RT service requests.



(a) Luxembourg (b) Monaco

Fig. 2: Per-request signaling overhead due to DAPP-ECC.

C. Communication overhead

For each simulated scenario, we now set the CPU resources to 10% above the amount of resources required by LBound to find a feasible solution when 100% of the requests are RT. While maintaining this amount of CPU resources, we vary the ratio of RT requests, and measure the overall amount of data for signaling used by DAPP-ECC. Fig. 2 presents the per-request signaling overhead, defined as the overall signaling data exchanged by DAPP-ECC ("control bytes") over the overall number of critical/new requests along the trace. We consider several values of the SFS accumulation delay parameter T_{ad}^{SFS}. A run of PD() may migrate also non-critical MSs, thus incurring a higher overhead than PU(). Hence, we set $T_{ad}^{PD} = 4 \cdot T_{ad}^{SFS}$. The results show that increasing the fraction of RT requests decreases the signaling overhead. The reason is that RT requests can be placed only in the three lowest levels in the tree, thus avoiding any signaling message between higher-level datacenters. When increasing the accumulation delays, the protocol aggregates more requests before sending a message, thus decreasing the signaling overhead. However, an accumulation delay of about 10 μs suffices. We stress that the accumulation delay only impacts the time until the protocol finds a new feasible placement, not the delay experienced by the user's application. Indeed, delaying the migration decision may deteriorate the user's QoE. However, this performance deterioration can be mitigated using an efficient prediction mechanism for the user's mobility. Furthermore, in practical scenarios, an accumulation delay of about 10 μs may be negligible compared to the more considerable delay incurred by the migration process. Finally, in all the settings considered, the signaling overhead associated with each request is ~ 100 bytes, implying a very low bandwidth overhead.

D. Cost comparison

In our next experiment, we compare the cost of various solutions for the PMP. We set the ratio of RT requests to 30%, and vary the resource augmentation.

The results (not detailed here due to lack of space) show that the costs obtained by BUPU and DAPP-ECC are almost identical, and both are up to 10% higher than LBound (depending on the concrete setting). That is, despite being distributed and asynchronous, DAPP-ECC obtain costs that are almost identical to those obtained by BUPU, which relies on a centralized controller with an always-accurate view of the system state. F-Fit typically only finds *any* feasible solution when resources are abundant, in which case all the placement algorithms easily obtain close-to-optimal costs.

VI. Related Work

State-of-the-art solutions to the PMP [2], [8], [9]. assume a centralized orchestrator that possesses fresh, accurate information about the locations, trajectories, and computational demands of all users, and the available resources at all the datacenters. Such an assumption may be impractical in a large system, possibly operated by several distinct operators.

Other solutions [12] independently select an optimal destination for each migration request, based on multiple considerations, such as topological distance, availability of resources at the destination, and the data protection level in the destination. However, such a selfish user-centric approach may fail to provide a feasible system-level solution when multiple RT users compete for resources in the edge.

The work [19] uses dynamic clustering of datacenters to handle multiple simultaneous independent placement requests. However, the complex dynamic clustering mechanism may result in significant communication and computational overhead. Also, [19] does not consider migrating non-critical requests to make room for a new user, as we do.

The PMP combines several properties of the Multiple Knapsack problem [25] with added restrictions typical for bin-packing problems (e.g., each item can be packed only on a subset of the knapsacks, and a feasible solution must pack all the items). However, in contrast to the usual settings of such problems, we aim at a distributed and asynchronous scheme that runs independently on multiple datacenters ("knapsacks"), using only little communication between them.

The work [26] optimizes lower-level implementational details of the migration process, to decrease its overhead. LSTM [27] considers learning algorithms that predict future service requests. These solutions are orthogonal to the PMP and hence could be incorporated into our solution to boost performance.

VII. Conclusions

We proposed a distributed asynchronous protocol for service provisioning in the cloud-edge continuum. Our solution is carefully designed to reduce signaling overhead by using only small control messages between immediate neighbor datacenters. Numerical results, derived using realistic settings, show that our approach may provide a feasible solution while using only slightly higher computing resources than a centralized scheduler, which may be impractical for large communication networks. Also, our protocol obtains reduced costs and incurs only a small communication overhead.

References

[1] B. Kar *et al.*, "QoS violation probability minimization in federating vehicular-fogs with cloud and edge systems," *IEEE Transactions on Vehicular Technology*, vol. 70, no. 12, pp. 13 270–13 280, 2021.

[2] D. Zhao *et al.*, "Mobile-aware service function chain migration in cloud–fog computing," *Future Generation Computer Systems*, vol. 96, pp. 591–604, 2019.

[3] S. Svorobej *et al.*, "Orchestration from the cloud to the edge," *The Cloud-to-Thing Continuum*, pp. 61–77, 2020.

[4] Y.-D. Lin, C.-C. Wang, C.-Y. Huang, and Y.-C. Lai, "Hierarchical cord for NFV datacenters: resource allocation with cost-latency tradeoff," *IEEE Network*, vol. 32, no. 5, pp. 124–130, 2018.

[5] L. Tong, Y. Li, and W. Gao, "A hierarchical edge cloud architecture for mobile computing," in *IEEE INFOCOM*, 2016, pp. 1–9.

[6] I. Cohen *et al.*, "Dynamic service provisioning in the edge-cloud continuum with bounded resources," *IEEE Transaction on Networking*, in press, 2023.

[7] H. Yu, J. Yang, and C. Fung, "Elastic network service chain with fine-grained vertical scaling," in *IEEE GLOBECOM*, 2018, pp. 1–7.

[8] I. Leyva-Pupo *et al.*, "Dynamic scheduling and optimal reconfiguration of UPF placement in 5G networks," in *ACM MSWiM*, 2020, pp. 103–111.

[9] X. Sun and N. Ansari, "PRIMAL: Profit maximization avatar placement for mobile edge computing," in *IEEE ICC*, 2016, pp. 1–6.

[10] I. Cohen *et al.*, "Parallel VM deployment with provable guarantees," in *IFIP Networking*, 2021, pp. 1–9.

[11] A. De La Oliva *et al.*, "Final 5g-crosshaul system design and economic analysis," *5G-Crosshaul public deliverable*, 2017.

[12] T. Ouyang *et al.*, "Adaptive user-managed service placement for mobile edge computing: An online learning approach," in *IEEE INFOCOM*, 2019, pp. 1468–1476.

[13] S. Wang *et al.*, "Dynamic service migration in mobile edge-clouds," in *IEEE IFIP Networking*, 2015, pp. 1–9.

[14] Martín-Pérez *et al.*, "OKpi: All-KPI network slicing through efficient resource allocation," in *IEEE INFOCOM*, 2020, pp. 804–813.

[15] M. Nguyen, M. Dolati, and M. Ghaderi, "Deadline-aware SFC orchestration under demand uncertainty," *IEEE Transactions on Network and Service Management*, pp. 2275–2290, 2020.

[16] L. Codecá *et al.*, "Luxembourg SUMO traffic (LuST) scenario: Traffic demand evaluation," *IEEE Intelligent Transportation Systems Magazine*, pp. 52–63, 2017.

[17] L. Codeca and J. Härri, "Monaco SUMO traffic (MoST) scenario: A 3D mobility scenario for cooperative ITS," *EPiC Series in Engineering*, vol. 2, pp. 43–55, 2018.

[18] "Opencellid," https://opencellid.org/, accessed on 3.10.2021.

[19] M. Goudarzi, M. Palaniswami, and R. Buyya, "A distributed application placement and migration management techniques for edge and fog computing environments," in *IEEE FedCSIS*, 2021, pp. 37–56.

[20] P. Alvarez *et al.*, "Microscopic traffic simulation using sumo," in *IEEE International Conference on Intelligent Transportation Systems*, 2018.

[21] "Service function chains migration." [Online]. Available: https://github.com/ofanan/SFC_migration

[22] "OMNeT++ discrete event simulator," 2023. [Online]. Available: https://omnetpp.org

[23] "Distributed SFC migration." [Online]. Available: https://github.com/ofanan/Distributed_SFC_migration

[24] "Gurobi optimizer reference manual," 2023. [Online]. Available: https://www.gurobi.com

[25] M. S. Hung and J. C. Fisk, "An algorithm for 0-1 multiple-knapsack problems," *Naval Research Logistics Quarterly*, vol. 25, no. 3, pp. 571–579, 1978.

[26] K. Ha *et al.*, "You can teach elephants to dance: Agile VM handoff for edge computing," in *ACM/IEEE SEC*, 2017, pp. 1–14.

[27] T. Subramanya and R. Riggio, "Centralized and federated learning for predictive VNF autoscaling in multi-domain 5G networks and beyond," *IEEE TNSM*, vol. 18, no. 1, pp. 63–78, 2021.