PAPS: A Framework for Decentralized Self-management at the Edge

Luciano Baresi, Danilo Filgueira Mendonça, and Giovanni Quattrocchi(✉)

Dipartimento di Elettronica, Informazione e Bioingegneria,
Politecnico di Milano, Milan, Italy
{luciano.baresi,danilo.filgueira,giovanni.quattrocchi}@polimi.it

Abstract. The emergence of latency-sensitive and data-intensive applications requires that computational resources be moved closer to users on computing nodes at the edge of the network (edge computing). Since these nodes have limited resources, the collaboration among them is critical for the robustness, performance, and scalability of the system. One must allocate and provision computational resources to the different components, and these components must be placed on the nodes by considering both network latency and resource availability. Since centralized solutions could be impracticable for large-scale systems, this paper presents PAPS (Partitioning, Allocation, Placement, and Scaling), a framework that tackles the complexity of edge infrastructures by means of decentralized self-management and serverless computing. First, the large-scale edge topology is dynamically partitioned into delay-aware communities. Community leaders then provide a reference allocation of resources and tackle the intricate placement of the containers that host serverless functions. Finally, control theory is used at the node level to scale resources timely and effectively. The assessment shows both the feasibility of the approach and its ability to tackle the placement and allocation problem for large-scale edge topologies with up to 100 serverless functions and intense and unpredictable workload variations.

Keywords: Edge computing · Serverless computing · Resource management · Service placement · Geo-distributed infrastructures

1 Introduction

The advent of mobile computing and the Internet of Things (IoT) is paving the ground to new types of applications. For most real-time, interactive applications, the latency from devices to cloud data centers can be prohibitive, and the transport and analysis of exponentially larger volumes of data may result in bottlenecks and consequently low throughput. Edge computing aims to fill this gap by means of densely-distributed computing nodes. Locality and decentralization mitigate network latency and helps reduce the amount of data that is transported to and processed by centralized servers.

The management of these geo-distributed infrastructures poses significant challenges. One must provision and allocate computational resources to the various components, but these components must be placed on edge nodes by taking

S. Yangui et al. (Eds.): ICSOC 2019, LNCS 11895, pp. 508–522, 2019.
https://doi.org/10.1007/978-3-030-33702-5_39

into account both latency and resource availability. The analysis of current workload, availability of resources, and performance of application components is key for the efficient placement of components and the allocation of resources, but it must be carried out in a timely manner for the entire topology. Network latency and time-consuming decisions, typical of centralized approaches, may jeopardize the overall effectiveness, especially with highly volatile workloads—a likely-to-happen scenario with densely distributed edge nodes that serve the needs of mobile/IoT devices.

On a parallel thread, serverless computing [2,11] is emerging as a novel cloud computing execution model that allows developers to focus more on their applications and less on the infrastructure. The user must only submit the application logic (stateless functions) to be executed. In turn, the provider offers dedicated containers for its execution and is in charge of resource allocation, capacity planning, and function deployment.

This paper proposes PAPS (Partitioning, Allocation, Placement, and Scaling), a framework for tackling the automated, effective, and scalable management of large-scale edge topologies through decentralized self-management and serverless computing. The approach partitions the large-scale edge topology into delay-aware network communities. Community leaders then tackle the joint allocation of resources and the placement of serverless functions—w.r.t both SLAs and the aggregate demand for each function. Finally, edge nodes exploit control theory to scale required containers timely while also giving valuable feedback to community leaders.

A prototype implementation of PAPS allowed us to assess the proposal on a set of experiments. Obtained results witness the feasibility of the approach and its ability to tackle the placement and allocation problem for large-scale edge topologies with up to 100 distinct functions and intense and unpredictable fluctuations of the workload. To the best of our knowledge, this is the first work that tackles the orchestration of such a high number of geo-distributed nodes and application components.

The rest of the paper is organized as follows. Section 2 presentes the context and introduces PAPS. Sections 3, 4, and 5 describe the self-management capabilities provided by PAPS at system, community, and node levels. Section 6 discusses the evaluation, Sect. 7 surveys related approaches, and Sect. 8 concludes the paper.

2 Context and PAPS

This paper focuses on a MEC topology [7,12] composed of a finite set of geo-distributed nodes $\mathcal{N}$. Figure 1 presents such a topology, where mobile and IoT devices access the system through cellular base stations. Each station is connected to a MEC node $i \in \mathcal{N}$ through the *fronthaul network*. MEC nodes in $\mathcal{N}$ are interconnected through the *backhaul network*. The total propagation delay $D_{i,j}$ between an end-user device that accesses the system through the base

station co-located with the MEC node $i \in \mathcal{N}$ and that is served by the MEC node $j \in \mathcal{N}$ is defined as:

$$D_{i,j} = \begin{cases} \gamma_i + \delta_{i,j}, & \text{if } i \,!= j \\ \gamma_i, & \text{if } i = j \end{cases} \tag{1}$$

where γ_i and $\delta_{i,j}$ are respectively the fronthaul and backhaul propagation delays.

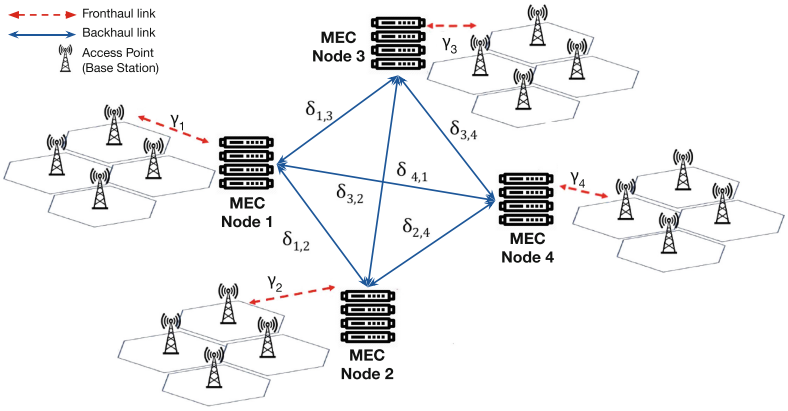


Fig. 1. Example topology of geo-distributed MEC nodes.

Our framework targets the dynamic allocation and placement of the containers required for the execution of serverless functions [2]. Even if the allocation model of different serverless vendors can vary [6], typically functions are given access to a fixed CPU share proportional to their memory requirement. In addition, we assume a deployment descriptor for each function that provides the memory required by the container in charge of executing the function and the SLA between the MEC operator and the application provider, that is, the owner of the function to execute.

The SLA associated with each function is specified through: a *Response Time* (RT_{SLA}), which states the upper limit for the round trip time between the arrival of a request to execute a function, its execution, and the returned value (if the invocation is synchronous), and a *Maximum Execution Time* (E_{MAX}), which limits its execution time. The latter is a common attribute in cloud-based serverless computing platforms, and it is key for us to guide the decision on the joint allocation and placement of the function.

In this context, for a given topology $\mathcal{N}$ and a set of admitted functions $\mathcal{F}$, the adaptation problem is twofold: one must decide *how many* containers are needed for each function and *where* (onto which nodes) should each container be placed. Each allocated container works as a server for a specific function $f \in \mathcal{F}$. Most vendors of serverless solutions try to use existing containers, if possible, and allocate new ones as soon as they are needed. In contrast, if one queued

requests for a short period Q, resources (containers) may become available, and thus the number of used resources may decrease. The perceived response time is then defined as:

$$RT = D + Q + E \tag{2}$$

where D represents the total propagation delay (see Eq. 1), Q the queuing time, and E the execution time. MEC operators must scale the number of containers allocated to each function $f \in \mathcal{F}$ and place them onto MEC nodes in $\mathcal{N}$ to minimize the difference between RT and RT_{SLA}. The goal is twofold: (i) to maximize the efficient use of resources, and thus the number of functions and users that can be admitted into the system; (ii) to prevent SLA violations.

This paper introduces PAPS, a framework to manage the allocation and placement problems in large-scale edge systems. Figure 2 shows that the self-management capabilities provided by PAPS work at three different levels: system, community, and node level. The next three sections describe how each level works in detail.

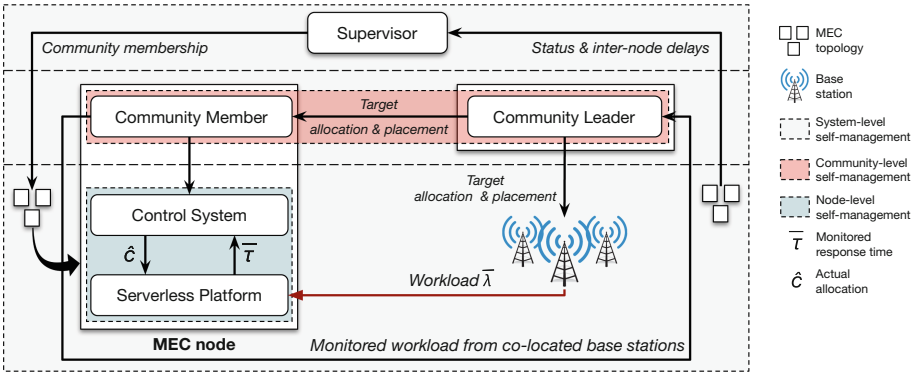


Fig. 2. PAPS in a nutshell.

3 System-Level Self-management

Self-management at system level aims to tackle the complexity of managing the large scale decentralized infrastructure by partitioning it into delay-aware network *communities*. In complex networks, a network is said to have a *community structure* if its nodes can be (easily) grouped into (potentially overlapping) sets of nodes such that each set is densely connected internally [14]. PAPS extends this definition and considers a set of logically interconnected MEC nodes, whose propagation delay from one another is below a threshold, as a *delay-aware* network community.

These communities provide a reduced space in which a solution to the placement and allocation problem can be computed. Furthermore, they allow for the decentralization of resource management (w.r.t. a single orchestrator) and its localization within distinct geographical areas.

The definition of these communities may follow different approaches. PAPS assumes the availability of a *supervisor* that has a global view of the MEC topology and uses a dedicated search algorithm to create communities. This algorithm takes the maximum inter-node delay (D_{MAX}) and the maximum community size (MCS) as parameters. The former is used to produce a sub-graph (G_{DA}). Each of its vertices maps to a node in the MEC topology, and an edge exists between two vertices if the network delay between their respective MEC nodes is lower than D_{MAX}. The second parameter limits the number of MEC nodes that can belong to a community, and it is useful to limit the complexity of community-level self-management.

The produced sub-graph G_{DA} in then used to feed the algorithm in charge of creating the communities. In particular, we adopt the SLPA method [14], whose complexity is $O(t * n)$, where t is a predefined maximum number of iterations (e.g. $t \leq 20$) and n is the number of nodes. Since the complexity is linear, the solution can also be used for very large topologies. Xie et al. [14] suggest a modest value ($t = 20$) for the maximum number of iterations needed to find good quality communities.

MEC nodes are co-located with fixed infrastructures. We assume that nodes and inter-node delays are expected to remain stable. However, topological changes caused by catastrophic failures, system upgrades, and other eventualities may require the adaptation of the community structure. The primary goal of the supervisor is, therefore, to ensure that communities remain consistent in their size and membership. While defining the best approach to tackle the adaptation of community structures, we took into account the amount of information that needs to be monitored, as well as the complexity of the community search procedure. The presence and health of the MEC nodes across the topology can be obtained through light-weight *heartbeat* messages sent by each node to the supervisor. This approach is commonly adopted in distributed systems of different scales and does not prejudice the scalability of the proposed solution.

The supervisor harnesses its global system view to tackle the adaptation of the community structure. We model the system-level adaptation as a master-slave MAPE loop [13] in which: *Monitoring* is performed by all nodes through heartbeat messages that contain the inter-node delay to all other nodes; the supervisor performs *Analysis* and *Planning* by deciding when and how to adapt the community structure in the advent of topological changes; *Execution* means that each affected node adapts by updating its community membership.

4 Community-Level Self-management

Self-management at community-level aims to ensure that the MEC nodes in the community operate under feasible conditions, that is, it aims to minimize the

likelihood of SLA violations to occur and, if they occur, to react to bring the community back to its equilibrium.

Inter-community Allocation. A first challenge that emerges when the MEC system is partitioned into communities refers to resource allocation to shared community members. This is to say that one must decide the share of resources that each overlapping community gets from its common members. One trivial, but possibly inefficient, solution is to privilege one community and give it all the "shared" resources: disadvantaged communities might need more resources while the common members might be underutilized by the privileged community. Since changes to the workload are expected to happen frequently, and without any warning, resources from common nodes must flow from one overlapping community to the other to prevent SLA violations.

PAPS tackles this problem by weighting the aggregate demand and capacity of each overlapping community. The aggregate demand refers to the number of containers needed to cope with the aggregate workload. The latter refers to the rate of requests that come from the base stations co-located with MEC nodes whose network latency w.r.t. the common node is below the inter-node delay threshold (D_{MAX}), plus a proportional demand share from the base station(s) co-located with the common node itself. The aggregate capacity, in turn, refers to the sum of the resources from the previous nodes, excluding the common node. A share of the capacity of the common node is then allocated to each overlapping community proportionally to their aggregate demand-capacity ratio.

Algorithm 1 details our inter-community allocation approach. This procedure is greedily performed for all MEC nodes in the topology that belong to two or more overlapping communities.

Algorithm 1. CapacityDemandRatio($community, node, D_{MAX}$)

```
neighborsInRange ← getNeighbors(community, node, D_MAX)
aggDemand ← 0, aggCapacity ← 0
for all n ∈ neighborsInRange do
    aggDemand ← getAggregateDemand(n)
    aggregateCapacity ← getAggregateCapacity(n)
end for
ovCount ← getOverlappingCount(node)
demandShare ← getDemand(node) / ovCount
aggDemand ← aggDemand + demandShare
return aggDemand / aggCapacity
```

Intra-community Allocation and Placement. The intra-community allocation aims to distribute resources among member nodes given the aggregate demand and capacity within the community. Each community has a leader responsible for solving the joint allocation and placement problem introduced in Sect. 2. Such a *centralization within decentralization* (i) allows the placement problem

to be solved in a single step for the whole community, and (ii) eliminates the need for a more complex coordination protocol. More importantly, the leader-based approach allows the placement problem to be solved by well-known centralized optimization techniques.

The complexity of the container placement problem implies high-resolution time and prevents communities to promptly adapt to workload fluctuations. Before a solution is computed, the workload may have significantly changed, and limit the efficiency and efficacy of the solution. Pro-active adaptation could be used to mitigate this problem. For example, if the workload is characterized by a well-known probabilistic distribution (e.g., a Poisson distribution), the allocation problem might then benefit from techniques such as queueing theory to predict the number of containers that are needed to keep the response time below a threshold. Unfortunately, the decentralized infrastructure model makes the previous assumption less realistic. Not only users can freely enter and exit different areas, but the aggregate workload to be served by each MEC node is limited compared to typical cloud data centers and thus may vary more abruptly. Because of this, PAPS favors a reactive adaptation approach for solving the joint allocation and placement problem.

Our solution draws inspiration from the Ultra-Stable system architecture [10]. The community-level self-management acts as the second control loop in the Ultra-Stable system. When workload fluctuations are significant enough to impact or to throw the node-level self-management out of its limits, the community-level self-management provides the community with a new allocation and placement solution. In turn, the node-level self-management works as the primary feedback loop in the Ultra-Stable system. Through its sensors, the MEC node monitors subtle changes in the environment (i.e., in the actual workload for each function). It accordingly responds, through its actuators by changing the actual number of containers hosted for each function. Hence, the community-level placement does not target a single solution, but a solution space in which the scaling of containers at node-level ultimately takes place.

The community-level self-management consists of an instance of the *regional planner* MAPE loop [13]. Each community member takes advantage of its privileged position within the MEC topology to *monitor* and *analyze* the workload coming from adjacent base stations (see Fig. 1). The number of containers needed to cope with a given workload while satisfying the SLA is determined at the node level by using a feedback loop with a short control period—compatible with the container start-up time (i.e., up to a few seconds). In turn, the community leader extrapolates this information to *plan* for the number of containers needed to satisfy the SLA given the aggregate workload over a longer control period—compatible with the time needed to compute the optimal placement (i.e., up to a few minutes).

Informed load balancers composing the community infrastructure use the computed optimal allocation and placement to route the workload coming from different base stations to their respective destinations (i.e., MEC nodes).

Each affected node in the community *executes* the plan with the update of the target allocation. Depending on how the new placement solution diverges, community members may have to remove/add function(s).

Our decentralized solution provides each MEC node with the freedom to decide the actual number of containers it hosts for each placed function based on monitored workload, SLA, and available computing resources. As the workload fluctuates, the response time deviates from its target value, and the node-level controller takes care of the timely creation and termination of containers to optimize resource usage while preventing SLA violations. The community-level solution is enforced by members in case of resource contention until a new optimal allocation and placement solution is enacted by the community leader.

Optimal Container Placement. PAPS is agnostic about the formulation of the optimal allocation and placement problem. In this paper, we formulate it as a mixed integer programming (MIP) problem as follows:

$$\min_{x} \quad \sum_{i\in\mathcal{N}}\sum_{j\in\mathcal{N}}\sum_{f\in\mathcal{F}} d_{i,j} * x_{f,i,j} \tag{3a}$$

$$\text{subject to} \quad d_{i,j} * x_{f,i,j} \leq x_{f,i,j} * D_f \qquad \forall i \in \mathcal{N}, \forall j \in \mathcal{N}, \forall f \in \mathcal{F} \tag{3b}$$

$$\sum_{i\in\mathcal{N}}\sum_{f\in\mathcal{F}} c_{f,i} * m_f * x_{f,i,j} \leq M_j \qquad \forall j \in \mathcal{N} \tag{3c}$$

$$\sum_{j\in\mathcal{N}} c_{f,i} * x_{f,i,j} = c_{f,i} \qquad \forall i \in \mathcal{N}, \forall f \in \mathcal{F} \tag{3d}$$

where the decision variable $0 \leq x_{f,i,j} \leq 1$ denotes the fraction of the demand for containers $c_{f,i}$, from any base station co-located with node $i \in \mathcal{N}$, for function $f \in F$, hosted on node $j \in \mathcal{N}$. The objective function (Eq. 3a) minimizes the overall network delay that results from placing containers. The first constraint (Eq. 3b) limits the propagation delay. Specifically, D_f is calculated by using the following equation:

$$D_f = \beta * (RT_{SLA,f} - E_{MAX,f}) \tag{4}$$

where $0 < \beta \leq 1$ defines the fraction of the marginal response time $RT_{SLA,f} - E_{MAX,f}$ for function $f \in F$ that can be used for networking. Conversely, the complement $1 - \beta$ defines the fraction of the marginal response time used for queuing requests for function f hosted on node j:

$$Q_{f,j} = (1 - \beta) * (RT_{SLA,f} - E_{f,j}) \tag{5}$$

where $E_{f,j}$ is the monitored execution time for function f hosted on node j. The queue component $Q_{f,j}$ is particularly important for the control-theoretic solution for scaling containers (see Sect. 5) since it provides an additional margin for the control actuation and thus mitigates the likelihood of overshooting.

The second constraint (Eq. 3c) ensures that the number of containers placed at a node j does not violate its memory capacity M_j. An additional constraint (Eq. 3d) ensures that the required containers for all $f \in \mathcal{F}$ are properly placed.

5 Node-Level Self-management

Self-management at node-level aims to efficiently and effectively scale the containers needed to satisfy the SLA (response time) of each admitted function given the fluctuations in the workload and the target allocation defined by the community leader. With a static allocation of resources, the response time of a function can change due to various reasons: for example, variations in the workload, changes in the execution time (e.g., due to input variation), and disturbances in the execution environment (e.g., at the operating system or hardware level). While some factors are harder to quantify and account for, others can be monitored and taken into account while determining the number of containers needed to prevent SLA violations. Our framework leverages a control-theoretic approach [3] to scale containers at node-level.

The control system is responsible for the deployment of containers onto the pool of virtual machines running on the MEC node. We consider a dedicated controller for each admitted function $f \in F$. Considering a discrete time, for each function, we define $\lambda(k)$ as the function of the measured arrival rate of requests at each control time k, while $\bar{\lambda}(k)$ is the corresponding vector for all admitted functions.

At time k, the function is executed in a $c(k)$ number of containers, while $\bar{c}(k)$ is the vector for all $f \in F$. The disturbances are defined as $\bar{d}$ and cannot be directly controlled and measured. Finally, $\bar{\tau}$ is the system output and corresponds to the response time vector that comprises all functions, whereas $\bar{\tau}^{\circ}$ corresponds to the vector of the desired response time for each function (or control *set-point*).

In our current set-up, function $\bar{\tau}^{\circ}(k)$ does not vary over time, that is, we target a constant response time for each function. These values should be less than the agreed SLA to avoid violations. For example, a reasonable target response time for non-critical functions is $0.8 * SLA$, while a lower value like $0.4 * SLA$ implies a more conservative allocation and can be used for safety-critical applications. Moreover, since a response time cannot be measured instantaneously, but by aggregating it over a predefined time window, many aggregation techniques could be used without any change to the model and controller. In our framework, we compute the average of the response time values in $\bar{\tau}$ within each control period, but stricter aggregation functions, such as the $99th$ percentile, could be used given the needs of the service provider.

We also use a characteristic function to model the system with enough details to govern its dynamics. We assume that this function needs not be linear but regular enough to be linearizable in the domain space of interest. Moreover, we consider this function be dependent on the ratio between the number of allocated containers c and the request rate λ. The characteristic function monotonically decreases towards a possible lower horizontal asymptote, as we can assume that once available containers are enough to allow a function to reach the foreseen degree of parallelism, the addition of further containers would provide no benefits in terms of response time. We found that a practically acceptable function is:

$$f\left(\frac{c(k)}{\lambda(k)}\right) = \widetilde{u}(k) = c_1 + \frac{c_2}{1 + c_3 \frac{c(k)}{\lambda(k)}} \tag{6}$$

where parameters c_1, c_2, and c_3 were obtained through profiling of each function.

As control technique, we rely on PI controllers because they are able to effectively control systems dominated by a first-order dynamic [1] (i.e., representable with first-order differential equations) such as the studied ones. Algorithmically, for each admitted function:

$$\begin{aligned} e &:= \tau_r^{\circ} - \tau_r; & c &:= max(min(Kmax, c), Kmin); \\ x_R &:= x_{R_p} + (1-p) * e_p; & x_{R_p} &:= (p-1)/(\alpha - 1) * f(c/\lambda) - e; \\ c &:= \lambda * f_{inv}((\alpha - 1)/(p-1) * (x_R + e)); & e_p &:= e; \end{aligned}$$

where e is the error, the p subscript denotes "previous" values, that is, those that correspond to the previous step, f and f_{inv} correspond to the characteristic function and its inverse, respectively, $\alpha \in [0, 1)$ and $p \in [0, 1)$ are the single pole of the controller and the system respectively, and x_R is the state of the controller. The higher the value of α is, the faster the error converges—ideally to zero—at the expense of a more fluctuating allocation.

At each control step, the function controllers run independently (i.e., without synchronization) to compute the next number of containers for the corresponding function, which is added to vector $\hat{c}$. The number of containers in $\hat{c}$ is not immediately actuated since the sum of required containers could be greater than the entire capacity of the resource pool. Instead, $\hat{c}$ is passed to a contention manager. This component outputs a vector $\bar{c}$, which contains the actual number of containers per function, defined as:

$$\bar{c}(k) = \begin{cases} \hat{c}(k), & \text{if } \textit{no resource contention} \\ \textit{solveContention}(\hat{c}(k)), & \text{otherwise} \end{cases} \tag{7}$$

where function *solveContention* scales the values in $\hat{c}$ according to the thresholds defined by the placement solution provided by the community leader (see Sect. 3). The contention manager also updates the state of each controller (variable x_{R_p}) to make it become consistent with the actual allocation.

6 Experimental Evaluation

We created a prototype implementation of the PAPS framework[1] based on PeerSim[2]. The implementation was used to evaluate the allocation, placement, and scaling mechanisms of PAPS, given different partitioning of the MEC topology. A node in the topology was implemented as a dynamic pool of threads, where

[1] Source code available at: https://github.com/deib-polimi/PAPS.
[2] http://peersim.sourceforge.net/.

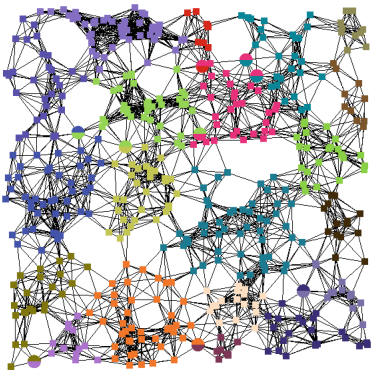


Fig. 3. Communities found in a large scale topology with 250 nodes.

Table 1. Results.

			RT		
Test	Conf	V	μ	σ	$95th$
OPT	10/50	6.4%	84.9	13.9	111.9
CT	10/50	0.6%	74.4	4.9	81.2
OPT	10/75	7.1%	89.6	15.1	113.4
CT	10/75	0.7%	75.6	7.8	81.8
OPT	10/100	8.9%	92.7	18.7	146.8
CT	10/100	0.9%	76.3	8.1	86.0
OPT	25/50	6.8%	92.6	16.7	176.0
CT	25/50	0.9%	75.6	10.7	85.8
OPT	25/75	10.5%	95.1	19.9	210.7
CT	25/75	1.8%	88.1	12.0	101.6
OPT	25/100	11.7%	101.6	23.0	221.3
CT	25/100	2.0%	85.8	20.0	107.3
OPT	50/50	7.4%	114.9	23.6	243.6
CT	50/50	1.4%	77.7	9.9	89.8
OPT	50/75	12.4%	118.9	27.6	260.6
CT	50/75	1.6%	78.7	15.6	91.6
OPT	50/100	14.0%	125.9	29.6	270.6
CT	50/100	2.2%	90.6	17.3	114.7

one container is a thread that executes the incoming requests. All the experiments were run using two servers running Ubuntu 16.04 and equipped with an Intel Xeon CPU E5-2430 processor for a total of 24 cores and 328 GB of memory.

The maximum number of containers that can be allocated onto a node depends on its memory capacity and the memory requirements of the functions that are to be deployed: 96 GB and 128 MB, respectively, in our experiments.

First, we assumed a large-scale edge topology of 250 nodes and normally distributed node-to-node latencies. We used the SLPA algorithm to partition the topology in communities of 10, 25, and 50 nodes (parameter MCS) with membership probability $r = 0.35$. Figure 3 shows the partitioning when MCS was set 25. Colored squares represent edge nodes within a single community; those that belong to overlapping communities are rendered with multi-color circles.

Then, we run two types of experiments to evaluate (i) the feasibility, performance, and scalability of the approach and (ii) the benefit of having a multi-layered self-management solution. The first experiment, called *testOPT*, tested the behavior of communities under an extremely fluctuating workload by only using community-level allocation and placement. Each node kept the target resources allocated to each running function constant between two community-level deci-

sions. The second experiment, called *testCT*, used both community-level and node-level adaptations to provide more refined and dynamic resource allocation for the incoming random workload.

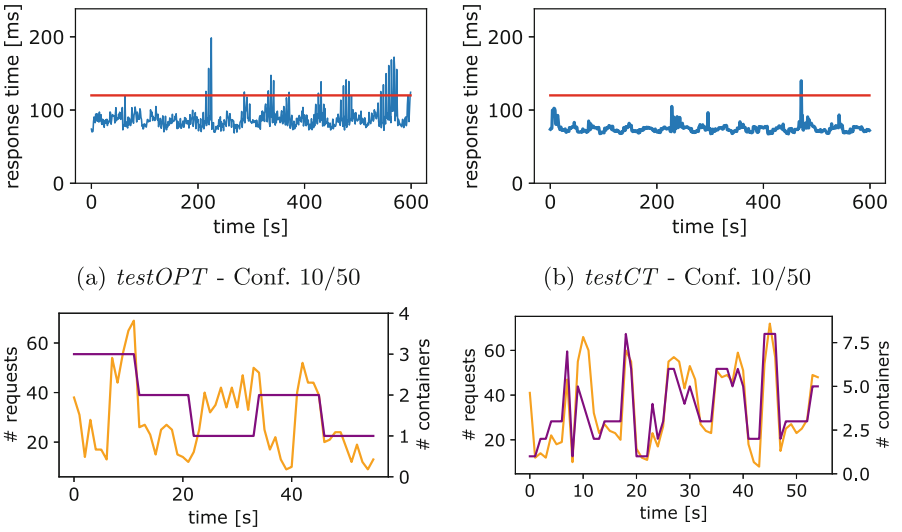
For each of the three community sizes, we tested the system with an increasing number of types of functions: 50, 75, 100. Each execution lasted 10 min and tested one of the nine combinations of community sizes and number of functions. For each configuration we executed 5 runs of *testOPT* and 5 runs of *testCT* for a total of 90 experiments.

The control periods of the community-level and node-level self-management were set to 1 min and 5 s, respectively. If no feasible optimal solution is found at the community level, PAPS solves a constraint-relaxed version of the optimization problem of Sect. 4, and the next placement starts after 1 min. Moreover, we set the fraction of the marginal response time β to 0.5 and the value of the pole of the node-level controller (see Sect. 5) to 0.9.

The workloads were generated by using normal distributions for both function execution times (E_k), while inter-arrival rates were generated by using three different scenarios (low, regular, high) that were chosen randomly every 15 s to simulate an extremely fluctuating traffic. Within each scenario, the time between two requests was computed by using an exponential distribution. Finally, the RT_{SLA} of all the functions was set to 120 ms, and ET_{MAX} was set to 90 ms.

Table 1 shows obtained results, where $Test$ can be either *testOPT* or *testCT*, $Conf$ shows used configuration (e.g., 10/50 means each community had 10 nodes, and there were 50 different function types), V shows the percentage of control periods in which the average response time violated the SLA, while columns μ, σ and $95th$ show, respectively, the overall average, the standard deviation, and the 95th percentile of the response time of the system aggregated over the five repetitions. If we focus independently on *testOPT* and *testCT*, we can observe that even by increasing the number of nodes and functions the percentage of failures is kept under 14.0% and 2.2%, respectively. These are reasonable values if we consider we used extremely variable workloads (changes every 15 s). Note that the control period used for the community-level decision is four times longer than the time between two scenarios. Instead, if we compare the results of both tests, we can easily notice the benefit of the node-level self-management. The control-theoretical planners reduce the number of violations by one order of magnitude: for example, from 6.4% to 0.6% in configuration 10/50, from 10.5% to 1.8% in configuration 25/75, and from 14% to 2.2% in configuration 50/100. Moreover, on average, the standard deviation and the 95th percentile of the response time are significantly lower in all *testCT* experiments.

The charts of Fig. 4 help better visualize obtained results. Figure 4(a) and (b) show the average response time for *testOPT* and *testCT* with configuration 10/50, where the horizontal line at 120 ms is the SLA. The first chart shows some violations, while the second chart only shows one violation close to 500 s and the response time is more constant (lower standard deviation) given the faster actuation of the node-level manager. Figure 4(c) and (d) show the number of requests (lighter line) and the allocation (darker line) during the execution of



(a) *testOPT* - Conf. 10/50

(b) *testCT* - Conf. 10/50

(c) *testOPT* - Sample of workload and allocation of a function on a node.

(d) *testCT* - Sample of workload and allocation of a function on a node.

Fig. 4. Experiment results.

a function on a single node for the two types of experiments (same configuration as before). *testOPT* exploits a longer control period given the complexity of the optimization problem. Therefore, the allocation is often sub-optimal and quite approximated w.r.t. the actual user needs (workload). On the other hand, the faster adaption used in *testCT* allowed the system to fulfill user needs better and follow the actual workload more closely.

7 Related Work

A few works combine the benefits of serverless and edge computing. Baresi et al. [4] propose a serverless architecture for Multi-Access Edge Computing (MEC). The authors also propose a framework [5] for the opportunistic deployment of serverless functions onto heterogeneous platforms, but they do not tackle the allocation and placement problem across nodes.

The platform proposed by Nastic et al. [9] extends the notion of serverless computing to the edge via a reference architecture to enable the uniform development and operation of data analysis functions. An orchestrator receives the information on how to con the application as high-level objectives and decides how to orchestrate the underlying resources. The implementation of the orchestration is left open.

Nardelli et al. [8] propose a model for the deployment of containerized applications. The number of required containers is defined by the user, the solution acquires and releases virtual machines and places containers onto these machines.

A possibly-new deployment configuration is defined in each adaptation cycle. In contrast, PAPS is in charge of both the target number of containers—to cope with agreed SLAs—and their placement onto MEC nodes. While PAPS works at the level of both nodes and communities, the multi-level formulation proposed in [8] could only be adopted in the latter case.

Zanzi et al. [16] propose a multi-tenant resource orchestration for MEC systems. The authors introduce a MEC broker that is responsible for procuring slices of the resources available in the MEC system to the various tenants based on their privilege level. At each optimization cycle, the broker decides on placing single-component applications onto the MEC node of choice (gold users), or onto any feasible node according to resource availability and network delay. We have instantiated our framework with a similar MEC topology, but our solution tackles the placement of a dynamic number of instances of various serverless functions onto stateless containers. We take into account the response time as SLA and a varying workload from different sources in the topology.

Yu et al. [15] propose a fully polynomial-time approximated solution for tackling the joint QoS-aware application placement and data routing problem in an edge system. Their formulation also admits multiple workload sources across the topology. Differently from PAPS, they focus on the placement of single-instance, single-component applications. While their solution tackles the allocation of bandwidth and the routing of data, it does not consider the allocation of computational resources, which is a crucial requirement in edge-centric systems.

A number of other works tackle the placement of applications onto geo-distributed infrastructures. Due to their combinatorial nature, tackled problems are usually NP-Hard [15], and many of the existing solutions are based on heuristics and approximations. These solutions are demonstrated for a limited number of nodes and applications or do not consider abrupt workload variations. PAPS targets different objectives, where scalability and unpredictable workload are first-class requirements. It tackles the optimal resource allocation and component placement by scaling containers at the node level through control theory.

8 Conclusions and Future Work

This paper presents PAPS, a comprehensive framework for the effective and scalable self-management of large edge topologies that works at different levels. It partitions the edge topology into smaller communities. Each community elects a leader that is in charge of placing and allocating containers for the incoming workload. Each node exploits control theory to scale containers properly and timely. The evaluation demonstrates the feasibility of the approach, its performance under extremely fluctuating workloads, and highlights the benefit of the multi-level solution.

As for future work, we plan to integrate PAPS into a real-world serverless framework and to extend our community-level allocation and placement algorithm to consider also the cost of migrating containers.

References

1. Åström, K.J., Hägglund, T.: PID Controllers: Theory, Design, and Tuning, vol. 2. ISA Research Triangle Park, Durham (1995)
2. Baldini, I., Castro, P., et al.: Serverless computing: current trends and open problems. In: Research Advances in Cloud Computing, pp. 1–20 (2017)
3. Baresi, L., Guinea, S., Leva, A., Quattrocchi, G.: A discrete-time feedback controller for containerized cloud applications. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 217–228. ACM (2016)
4. Baresi, L., Filgueira Mendonça, D., Garriga, M.: Empowering low-latency applications through a serverless edge computing architecture. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (eds.) ESOCC 2017. LNCS, vol. 10465, pp. 196–210. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67262-5_15
5. Baresi, L., Mendonça, D.F., Garriga, M., Guinea, S., Quattrocchi, G.: A unified model for the mobile-edge-cloud continuum. ACM Trans. Internet Technol. **19**, 29:1–29:21 (2019)
6. Lloyd, W., et. al.: Serverless computing: an investigation of factors influencing microservice performance. In: Proceedigns of the 6th IEEE International Conference on Cloud Engineering, pp. 159–169 (2018)
7. Mach, P., Becvar, Z.: Mobile edge computing: a survey on architecture and computation offloading. IEEE Comm. Surv. Tutorials **19**(3), 1628–1656 (2017)
8. Nardelli, M., Cardellini, V., Casalicchio, E.: Multi-level elastic deployment of containerized applications in geo-distributed environments. In: Proceedings of the 6th IEEE International Conference on Future Internet of Things and Cloud, pp. 1–8 (2018)
9. Nastic, S., Rausch, T., et al.: A serverless real-time data analytics platform for edge computing. IEEE Internet Comput. **21**, 64–71 (2017)
10. Parashar, M., Hariri, S.: Autonomic computing: an overview. In: Unconventional Programming Paradigms, pp. 257–269 (2005)
11. Roberts, M.: Serverless architectures. https://martinfowler.com/articles/serverless.html. Accessed May 2018
12. Several authors: Mobile edge computing (mec); framework and reference architecture. Technical report, ETSI GS MEC, January 2019. http://www.etsi.org/deliver/etsi_gs/MEC/001_099/003/01.01.01_60/gs_MEC003v010101p.pdf
13. Weyns, D., et al.: On patterns for decentralized control in self-adaptive systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 76–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_4
14. Xie, J., Szymanski, B.K., Liu, X.: SLPA: uncovering overlapping communities in social networks via a speaker-listener interaction dynamic process. In: Proceedings of the 11th IEEE International Conference on Data Mining Workshops, pp. 344–349 (2011)
15. Yu, R., Xue, G., Zhang, X.: Application provisioning in FOG computing-enabled Internet-of-Things: a network perspective. In: Proceedings of the 37th IEEE International Conference on Computer Communications, INFOCOM, pp. 783–791 (2018)
16. Zanzi, L., Giust, F., Sciancalepore, V.: M^2ec: a multi-tenant resource orchestration in multi-access edge computing systems. In: Proceedings of the 19th IEEE Wireless Communications and Networking Conference, WCNC, pp. 1–6 (2018)