Voilà: Tail-Latency-Aware Fog Application Replicas Autoscaler

Ali Fahs, Guillaume Pierre, Erik Elmroth

Voilà: Tail-Latency-Aware Fog Application Replicas Autoscaler

Ali J. Fahs[1], Guillaume Pierre[1], and Erik Elmroth[2,3]

[1]Univ Rennes, Inria, CNRS, IRISA
[2]Umeå University
[3]Elastisys AB

***Abstract*—Latency-sensitive fog computing applications may use replication both to scale their capacity and to place application instances as close as possible to their end users. In such geo-distributed environments, a good replica placement should maintain the tail network latency between end-user devices and their closest replica within acceptable bounds while avoiding overloaded replicas. When facing non-stationary workloads it is essential to dynamically adjust the number and locations of a fog application's replicas. We propose Voilà, a tail-latency-aware auto-scaler integrated in the Kubernetes orchestration system. Voilà maintains a fine-grained view of the volumes of traffic generated from different user locations, and uses simple yet highly-effective procedures to maintain suitable application resources in terms of size and location.**

I. Introduction

Humans and IoT devices produce ever-increasing volumes of data. It is expected that, by 2025, 75% of all enterprise data will be generated out of the data centers [20]. Transmitting these data over long-distance networks to the cloud before processing them is becoming increasingly undesirable and sometimes even infeasible. Instead, "fog computing" aims at processing data using resources within very low latency to the end users [25]. In contrast to cloud computing where large numbers of resources are co-located in a handful of datacenters far away from the users, fog computing scatters nodes on the network edge, in the immediate vicinity of the end users.

Although some fog computing applications are designed to serve the needs of a single end user, many others aim to serve requests from a *population* of end users located within a broad geographical area such as a city or a region [1]. To deliver low-latency processing of their requests, fog applications may be designed as a set of functionally-equivalent *service replicas* which can be placed in strategic network locations such that every user request can be served by a nearby replica.

The number of service replicas an application should deploy is mainly determined by two factors. First, the geographical distribution of the end users requires one to create enough replicas such that a nearby replica exists for every source of traffic. Second, any replica necessarily has a limited processing capacity, which may require one to create multiple replicas to serve workloads originating from major sources of traffic.

Fog computing resources are precious in a multi-tenant environment, so fog applications must carefully adjust their deployments so that they satisfy their QoS objectives while reducing their resource usage as much as possible. On the other hand, any user-produced workload may largely vary over time [19], which motivates the need for using an auto-scaler to dynamically adjust the number and locations of a fog application's replicas.

A fog application replica auto-scaler aims to reach three objectives: (1) **network proximity** such that every request may be routed to a nearby replica with a network round-trip latency lower than some threshold l_o; (2) **processing capacity management** such that no replica receives more requests than its processing capacity c_o; and (3) **high resource utilization** such that the majority of the provisioned resources are actually being utilized according to their capacity. Following best practice from commercial content delivery networks [13], we aim to optimize the *tail network latency* rather than its mean, which practically requires minimizing the number of user requests which incur a network round-trip latency $l > l_o$.

We propose Voilà, a tail-latency-aware fog application replica auto-scaler. Voilà integrates seamlessly with Kubernetes, the de-facto standard container orchestration framework in clusters and data centers [12]. Kubernetes is also a promising basis for designing future-generation fog computing platforms [4], [7], [14], [24], [27]. Voilà continuously monitors the request workload produced by all potential traffic sources in the system, and uses efficient algorithms to determine the number and location of replicas that are necessary to maintain the application's QoS within its expected bounds despite potentially large variations in the request workload characteristics.

Our evaluations based on a 22-nodes cluster and a real traffic trace shows that Voilà guarantees 98% of the requests are routed toward a nearby and non-overloaded replica. The system also scales well to much larger system sizes.

Section II surveys the technical background, and Section III discusses the related work. Then Section IV presents the system and Section V evaluates it. Finally, Section VI concludes.

II. Background

A. Kubernetes

Kubernetes is an open-source orchestration engine which automates the deployment, scaling and management of containerized applications [16]. As shown in Figure 1, a Kubernetes cluster consists of a master node responsible for the

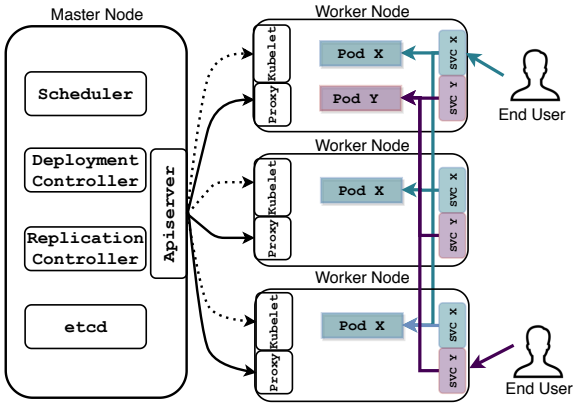


Fig. 1. Kubernetes architecture.

monitoring and the management of the deployed applications, and any number of worker nodes that constitute the system's computing, network, and storage resources.

In Kubernetes, an application deployment is composed of a set of *pods*, defined as a set of logically-related containers and data volumes to be deployed on a single machine. Application replication is ensured by deploying multiple identical pods. These pods can be then exposed to external end users as a single entity by creating a *service* which exposes a single IP address to the end users and acts as a front end which routes requests to one of the corresponding pods.

The Kubernetes scheduler is in charge of determining which pod will be placed in which worker node. By default Kubernetes uses latency-*un*aware filtering/scoring algorithms. The objective of this work is to design a new scheduler which controls both the number of replicas and their placement within the fog computing infrastructure to maintain QoS guarantees across large variations in the end-user request workload.

loud like availability, utilization, deployment

B. Network Proximity

In a geo-distributed system like a fog computing platform, every user request should be routed to a nearby node to be processed as quickly as possible. In Kubernetes, request routing works in two phases. First, some external routing mechanism must ensure that end user's requests are routed to any of the Kubernetes' worker nodes. Fog computing platforms typically make use of SDN/NFV technologies to route end user requests to a nearby node which then acts as a gateway to enter the Kubernetes system. In a second phase, Kubernetes further routes the request toward one of the worker nodes which execute a pod of the concerned application, where the request gets processed.

a) Estimating network latencies: To determine which worker node should receive each incoming request, and to reason about the necessary placement of an application's pods within the available worker nodes, it is important to know the network latency between every pair of worker nodes in the system. Rather than relying on simple metrics like geographical distance (which delivers poor accuracy [15]) or frequent pairwise measurements (which would require $O(N^2)$ measurements), we rely on Serf [10], a mature implementation of Vivaldi coordinates [5]. Serf relies on a lightweight gossip-based algorithm to maintain reasonably accurate latency estimates between every pair of nodes in the system.

Since the first part of a request route (from the end-user device to the first Kubernetes node) is independent from the Kubernetes system itself, in this work we consider only the network latencies between this gateway node and the worker nodes holding application replicas.

b) Routing requests to a nearby pod: Kubernetes' internal routing is based on simple location-unaware load balancing techniques such as round robin between all available pods of the concerned application. However, this approach may dispatch requests to far-away replicas even if there exists nearby ones. To ensure that requests are received by nearby pods, Voilà relies on Proxy-mity, which overrides the routing rules to favorize replicas reachable through low-latency routes [7].

c) Pod placement and auto-scaling: Routing requests to the closest available pod is not sufficient to guarantee low latency for every end-user requests. Depending on the end-users' locations within the system, it is necessary to carefully choose the number of pods and their node placement such that Proxy-mity can find a nearby pod to route requests to.

Choosing the number and placement of an application's pods necessarily results from a tradeoff between using the smallest possible number of pods (to reduce costs), while maintaining enough of them to cover the regions where users are located, and without overloading any of the pod replicas. The number and placement of pods is therefore a complex function of the Quality-of-Service requirements defined by the application, and the request workload imposed by its users.

d) Optimizing the tail vs. mean network latency: In latency-sensitive environments such as fog computing applications, it is important that each individual request gets processed with very low latency. For instance, virtual reality applications usually require consistently low response times. In such applications *"lag spikes and dropouts need to be kept to a minimum, or users will feel detached"* [6]. Aiming to minimize the mean latency between the user devices and their closest replica does not allow one to provide such extremely demanding type of guarantees. Instead, for each application we define a round-trip latency threshold l_o (e.g., $l_o = 20\,ms$), and aim to minimize the number of "slow" requests which incur a round-trip network latency $l > l_o$.

Likewise, requests addressed to an overloaded replica may incur delays due to the saturation of server resources. We define a replica load threshold c_o (e.g., $c_o = 50\,req/s$) and also flag requests addressed to overloaded replicas as "slow."

C. Non-stationary traffic properties

Any online application which processes incoming requests from an unbounded user population notoriously experiences significant workload variations across time [19]. This also applies to fog applications. However, geo-distributed systems such as fog computing platforms must not only handle variations of the aggregate workload produced by their entire user

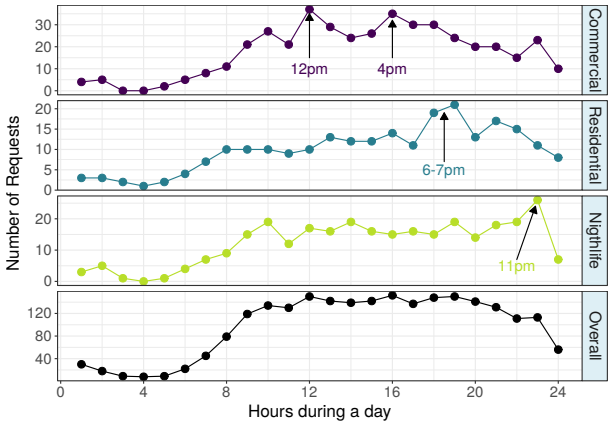


Fig. 2. Load variation according to location.

population, but also variations in the location from which the users generate their traffic.

To highlight the non-stationarity in time and in space experienced in the fog, we analyze a geo-distributed request workload derived from telecommunication traffic traces from the Trentino region in Italy, and emulated proportionally to the number of Internet requests per user found in the trace [2].

Figure 2 shows the aggregated requests of users in 10 different areas of the city, and highlights the traffic produced by three of them. Overall we see a typical day/night pattern where most of the workload is produced between 9am and 11pm. However, different zones observe workload peaks at different times of the day. Commercial districts with shopping malls and offices peak at 12pm and 4pm, whereas residential areas peak at 7pm and nightlife neighborhoods peak at 11pm.

For a fog replica auto-scaler such as Voilà, this means that application replicas should not only be created in the morning and removed in the evening to follow the aggregated traffic intensity. Also during the day, to maintain proximity between the users and the application, replicas must be created/deleted/relocated from one neighborhood to another.

III. State Of The Art

Fog computing servers are considered as precious resources. Numerous resource management mechanisms have therefore been proposed to address various aspects of fog computing resource management [9], [17].

A number of systems do not vary the number of replicas but rather propose to offload selected tasks to a backend cloud to reserve precious fog resources only for the most demanding tasks. *Mukherjee et al.* reduce the overall task completion latency by solving Quadratically Constraint Quadratic Programming optimization problem [11]. *Vu et al.* offload services in fog radio access networks to optimize the energy consumption and offload latency [22]. *Yousefpour et al.* propose a delay-minimizing offloading policy for fog nodes to reduce service delay for the IoT nodes [26]. In all these works, the fog layer network latencies are either not considered [11] or defined as a constant between all the nodes [22], [26] In contrast, we consider the fog as a set of dispersed nodes where placing a pod on one node or another strongly influences the resulting quality of service. Also, choosing some tasks to be processed in the backend cloud does not reduce the tail execution latency. We thus aim at processing all the requests in the fog layer, without offloading to a backend cloud.

Several papers are based on Kubernetes. *Zheng et al.* propose to vary the number of pods according to the load, but does not address the question of pod placement nor efficient routing between the end users and their closest pod [27]. Several other works propose location-aware pod placement mechanisms for Kubernetes [4], [8], [14], [24]. However, they consider the number of replicas as a constant and do not aim to vary it to accommodate non-stationary workloads.

Finally, few systems propose auto-scaling for the fog. ENORM aims to reduce latency between users and computing devices, and the network traffic to the cloud [23]. However, it chooses the resources regardless of their location, and essentially considers every fog nodes as equivalent to one another. ORCH proposes to dynamically add compute nodes in the system to resolve workload surges [18]. In contrast, we consider the set of worker nodes as a constant and aim to place the right number of replicas in the right set of nodes.

To our best knowledge, Voilà is the only dynamic fog resource manager which considers at the same time auto-scaling to adjust the necessary number of application replicas across any significant variation of the request workload, placement/replacement to choose where these replicas should execute, and routing of end-user request to nearby replicas. It aims at optimizing the tail request latency rather than its mean while avoiding replica overloads. This paper presents the autoscaling and placement algorithms, and evaluates their implementation integrated in Kubernetes on a testbed of 22 fog nodes. Request routing is addressed in a separate paper [7].

IV. System Design

The objective of this work is to dynamically scale and place an application's replicas in a cluster of geo-distributed fog nodes to predominantly minimize the number of slow requests while maintaining efficient resource utilization. A request is said to be *slow* in two cases: (1) it encounters a network round-trip time between the Kubernetes gateway and the serving pod greater than the threshold latency l_o defined by the application provider; or (ii) it is addressed to a pod whose current workload is greater than the specified pod capacity c_o. System administrators are also requested to define $\mathcal{E}_o$, the maximum acceptable percentage of slow requests.

A. System model and monitoring

Table I summarizes the main variables used to describe our system model. A fog computing cluster is defined as a set of n server nodes $\Delta = \{\delta_1, \delta_2, \ldots, \delta_n\}$, where every δ_i is an object of class `Node` which holds the status of the server node and the list of pods it currently hosts. An application is defined as a set of r replicas $\Phi = \{\varphi_1, \varphi_2, \ldots, \varphi_r\}$, where every φ_i is an object of class `Pod` which holds the status of the pod and the identifier of the server node where it is running. All these variables are maintained by Kubernetes as part of its

normal operations. They can be obtained through simple call to Kubernetes' `etcd` service.

The latency matrix L contains all round-trip latencies between pairs of nodes, where every l_{ij} is the RTT latency between nodes δ_i and δ_j as estimated by Serf. We can obtain an up-to-date estimate of any such RTT latency with a simple call to Serf's `rtt` API at the master node.

Every worker node in a Kubernetes cluster has two different roles. First, it may host a pod of the application which processes user requests. Second, it may act as a gateway. End user requests may be sent to any gateway node, which is then in charge of routing the request to one of the application's pods. For clarity, we distinguish these two roles as gateway g_i and server node δ_i.

Kubernetes routes network requests from the gateway nodes to the server nodes using IP-level routing. This means that we have access to precise kernel-level counters measuring exactly how many network packets have been routed from which gateway to which server node, and back. To ensure that gateways route incoming requests to nearby nodes, Proxy-mity defines a matrix P where every p_{ij} represents the probability a request received by gateway g_i should be routed to server node δ_i, defined using a monotonically decreasing function of the estimated latency between g_i and each server node where pods may be located (so that requests have high probability of being routed to nearby server nodes). Being filled with probabilities, the matrix P maintains the following properties:

$$0 \leq p_{ij} \leq 1 \quad \forall (i,j) \in [\![1,n]\!]^2$$
$$\sum_{j=1}^{n} p_{ij} = 1 \quad \forall i \in [\![1,n]\!]$$

The specific values p_{ij} are defined by Proxy-mity for all gateways and server nodes, *as if* every node actually hosted a pod of the application.

B. Replica placement quality evaluation

We evaluate the quality of any potential replica placement decision as the percentage of slow requests among all received requests ($\mathcal{E}\%$). Any placement decision consists of a set of host nodes ω_i. Voilà calculates $\mathcal{E}(\omega_i)$ of any potential placement according to the current load and latency distribution.

To allow Voilà to evaluate a large number of potential placements in reasonable time, $\mathcal{E}$ should be evaluated as efficiently as possible. Voilà defines the probability matrix P once per placement cycle, and then exploits it to evaluate the quality of any replica placement.

Procedure 1 illustrates the computation of $\mathcal{E}(\omega_i)$. First, it removes the matrix's rows which correspond to idle gateways. It also sets to 0 the columns which correspond to server nodes which do not host a replica in the evaluated placement. After normalizing the matrix such that the sum of probabilities per row equals 1, the resulting matrix $\hat{P}$ contains only the routing probabilities from active gateways to potential replicas.

Using $\hat{P}$ and the set of active gateways $\hat{G}$, Voilà calculates the number of slow requests due to high network latency:

TABLE I
SYSTEM MODEL'S VARIABLES.

Variable	Definition
Cluster Variables	
Δ	$= \{\delta_1, \delta_2, \ldots, \delta_n\}$, set of all server nodes.
δ_i	$\in \Delta$, a server node of index i.
n	$= \lvert\Delta\rvert$, number of nodes in the cluster.
Φ	$= \{\varphi_1, \varphi_2, \ldots, \varphi_r\}$, set of application's replicas.
φ_j	$\in \Phi$, an application replica of index j.
r	$= \lvert\Phi\rvert$ with $r \leqslant n$, number of application replicas.
L	$[l_{ij}]$, symmetric $n \times n$ matrix of inter-node RTT latencies.
l_{ij}	$= l_{ji}$ RTT latency between nodes δ_i and δ_j.
Testing Variables	
G	$= \{g_1, g_2, \ldots, g_n\}$, set of all end user's gateways.
g_i	$\in G$, a gateway located at node δ_i.
$g_i.load$	The number of requests redirected by gateway g_i.
$\varphi_j.load$	The number of requests received by server node φ_j.
P	$[p_{ij}]$, $n \times n$ matrix of the request route probabilities.
p_{ij}	Probability of following the route from g_i to δ_j.
T	$[t_{ij}]$, $n \times n$ Test matrix.
t_{ij}	$\in [0,1]$, labels the routes $g_i \rightarrow \varphi_j$ as suitable or not.
$\mathcal{E}$	The percentage of slow requests per cycle.
$\mathcal{E}_T$	The overall percentage of slow requests over a full test.
Ω	$= \{\omega_1, \omega_2, \ldots\}$, set of all possible placements.
ω_i	$\subset \Delta$ given $\lvert\omega_i\rvert = r$, one possible placement solution.
Provider-Defined Variables	
l_o	RTT latency threshold in ms.
c_o	Pod capacity threshold in $req/pod/s$.
$\mathcal{E}_o$	$\mathcal{E}$ per cycle threshold in $\%$.
τ	Cycle duration in s.

$$V_{l_o}(\hat{G}, \hat{P}, L) = \sum_{i=1}^{|\hat{G}|} \sum_{j=1}^{n} \hat{p}_{ij} \times \hat{g}_i.load \times f_1(l_{ij})$$
$$\text{where } f_1(l_{ij}) = \begin{cases} 1 & \text{if } l_{ij} > l_o \\ 0 & \text{else} \end{cases}$$

Similarly, function V_{c_o} counts the requests which would be routed to an overloaded replica φ_j hosted at δ_k:

$$V_{c_o}(\Phi, \hat{P}) = \sum_{j=1}^{r} f_2(\varphi_j) \qquad \varphi_j.load = \sum_{i=1}^{|\hat{G}|} \hat{g}_i.load \times \hat{p}_{ik}$$
$$\text{where } f_2(\varphi_j) = \begin{cases} \varphi_j.load - c_o \times \tau & \text{if } \varphi_j.load > c_o \times \tau \\ 0 & \text{else} \end{cases}$$

The variable $\mathcal{E}$ is then computed as the sum of V_{l_o} and V_{c_o} divided by the total load:

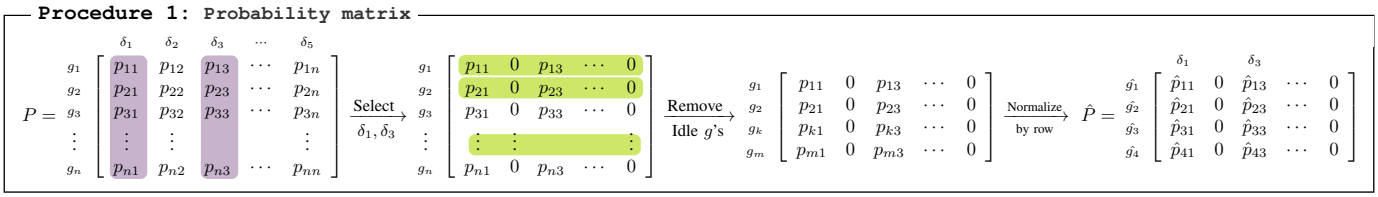
$$\mathcal{E}\% = 100\% \times \frac{\text{Slow}}{\text{Total}} = 100\% \times (V_{l_o} + V_{c_o}) / \sum_{i=1}^{n} g_i.load$$

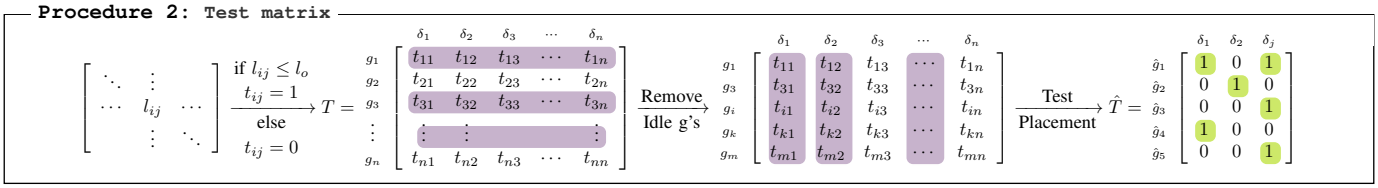
Selecting a suitable replica placement consists in finding ω_i such that $\mathcal{E}(\omega_i) \leq \mathcal{E}_o$.

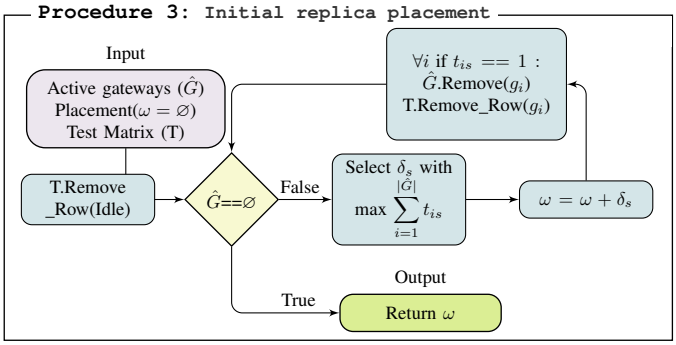
C. Initial replica placement

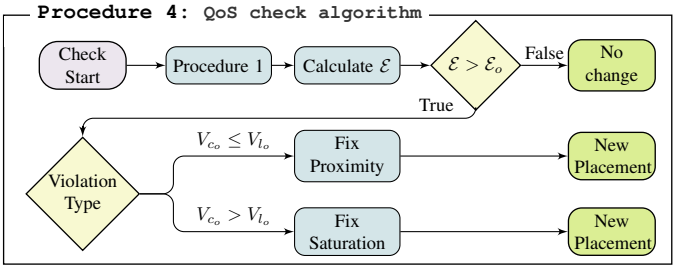
When a new application is deployed in the fog computing platform, no information is available yet about its traffic characteristics. Instead, Voilà uses the set of active gateways from other deployed applications (regardless of their actual workload) to define an initial set of replica locations.

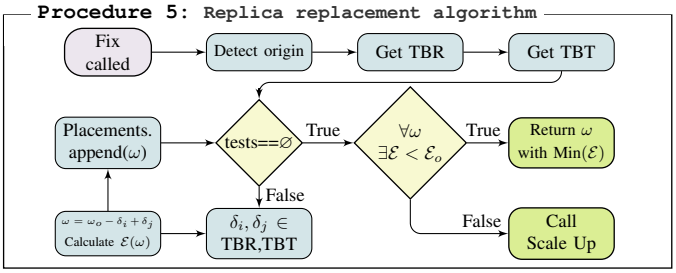
Finding the optimal placement of r replicas among n worker nodes requires in principle one to fully explore the set of all











possible solutions Ω. However, this set is extremely large even for a modest value of n and r:

$$|\Omega| = \binom{r}{n} = \frac{n!}{r!(n-r)!}$$

For example, placing 10 replicas out of 50 server nodes yields a set of 10,272,278,170 possible placements. Exploring them all is obviously infeasible. Instead, we explore only a small subset of promising placements, and choose the first one which satisfies that all active gateways have a nearby replica to which they may route incoming requests.

Procedure 2 takes the latency matrix L as an input, and labels all the possible routes as suitable (with value 1 if $l_{ij} \le l_o$) or unsuitable (with value 0 otherwise). The objective of the resulting Test matrix $\hat{T}$ is to check whether every active gateway is covered by at least one nearby replica. this condition is determined by the fact that each row corresponding to an active gateway $\hat{g}_i$ has at least one $\hat{t}_{ij} = 1$.

The algorithm to identify a suitable replica placement is illustrated in Procedure 3. Starting from an empty placement ω, the set of active gateways $\hat{G}$, and the Test Matrix T, the algorithm starts by removing the idle gateways from T, then iteratively adds new replicas until all active gateways are covered by at least one suitable nearby replica. Every new host node δ_s is chosen with a greedy heuristic as the one which covers the greatest number of gateways. The loop continues until $\hat{G}$ becomes empty, which indicates that all active gateways are covered. The procedure finally returns ω.

Note that this initial placement is only a starting point when a new application is deployed in the platform. It is determined based on latency requirements only. Depending on the request workload, it may or may not satisfy the processing capacity requirement as well. Also, user-generated traffic is expected to vary over time, which mandates the usage of re-placement and autoscaling techniques, as we discuss next.

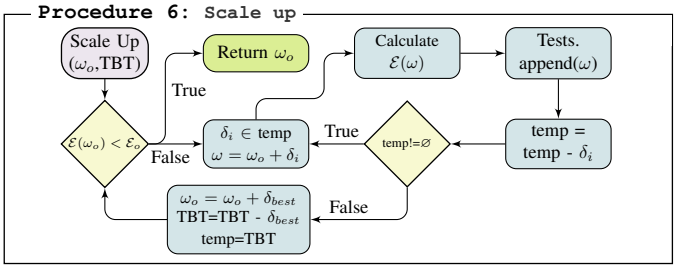
D. Replacement and autoscaling

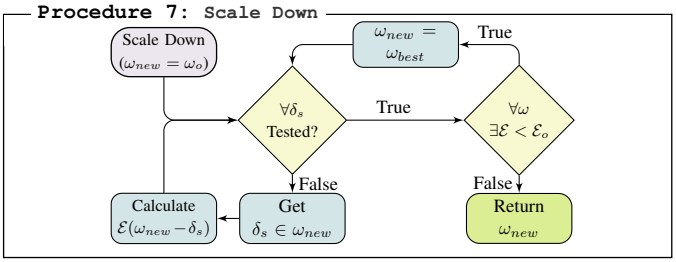
After an application has started, Voilà periodically checks whether the latency and the processing capacity requirements are still met. In case of violation, it implements corrective actions to bring the QoS within its desired bounds. Voilà also periodically checks whether fewer replicas may be sufficient.

1) Checking for potential violations: Voilà periodically checks whether the QoS constraints are still respected. As shown in Procedure 4, the QoS check algorithm starts by calculating the percentage $\mathcal{E}$ of slow requests in the last cycle for ω_o. If $\mathcal{E}(\omega_o) > \mathcal{E}_o$ a violation is declared and the violation type determines which corrective function must be called.

2) Replica replacement: When a QoS violation is detected, Voilà first tries to fix it by moving a replica from one server node to another. Procedure 5 starts by selecting a number of replicas To-Be-Replaced (*TBR*) and a number of server nodes To-Be-Tested (*TBT*). The TBR and TBT are chosen according to the nature of the QoS violation:

- In the case of a Proximity violation, TBR is the set of current replicas which participate the least to the gateways-to-replica





proximity metric. An active gateway which depends on a single nearby replica with latency under l_o defines this replica as "vital." On the other hand, all non-vital replicas are included in TBR. Similarly, server nodes are included in TBT if they are located close enough from one gateway which does not have access to a suitable replica.

$$\hat{T} = \begin{array}{c} \\ \hat{g}_1 \\ \hat{g}_2 \\ \hat{g}_3 \\ \hat{g}_4 \end{array} \begin{array}{c} \begin{array}{ccc} \delta_1 & \delta_2 & \delta_3 \end{array} \\ \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \end{array} \quad (1) \qquad T = \begin{array}{c} \\ \hat{g}_1 \\ \hat{g}_2 \\ \hat{g}_3 \\ \hat{g}_4 \end{array} \begin{array}{c} \begin{array}{cccccc} \delta_1 & \delta_2 & \delta_3 & \delta_4 & \delta_5 & \delta_6 \end{array} \\ \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix} \end{array} \quad (2)$$

In (1) we see an example where the test matrix $\hat{T}$ indicates that gateway $\hat{g}_3$ does not have any nearby replica, which is the source of the QoS violation. Gateway $\hat{g}_4$ has only one nearby replica at δ_2, so δ_2 is not included in TBR. TBR finally contains δ_1 and δ_3 . In (2) we see the full test matrix T (including server nodes not currently hosting a replica). TBT then contains δ_4 and δ_6 as these two server nodes are considered as close enough from gateway $\hat{g}_3$.

- In the case of a capacity saturation violation, TBR contains the list of replicas currently receiving a low workload. TBT is the set of nodes located in close proximity from the currently overloaded replicas.

Once the sets TBR and TBT have been defined, Procedure 5 takes replica re-placement decisions in the same way for both types of QoS violations. It iteratively chooses a pair of nodes $\delta_i \in$ *TBR*, $\delta_j \in$ *TBT*, and evaluates $\mathcal{E}$ in case node δ_i was replaced with δ_j in the current replica placement ω. If at least one replacement decision delivers an acceptable QoS with $\mathcal{E}(\omega) < \mathcal{E}_o$ within some pre-defined computation time, then the procedure returns the best replacement decision it has found. Otherwise, it considers that replacing replicas is unlikely to address the QoS violation, so it calls the Scale Up procedure to create an additional replica.

3) Scaling up: To choose the node where an additional replica should be created, Procedure 6 first defines a set TBT in the same way as previously. It then iteratively considers every node from this set and tests whether adding it to ω (without replacing the existing replicas) would solve the QoS violations. If no single new replica is found to be able to solve the violation, it tries to add two new replicas, and so on until the violation is solved or all nodes from TBT have been added.

4) Scaling down: Scaling down does not take place upon any QoS violation. Rather, if the system did not occur any violation for a predefined period of time, it checks whether it may reduce the number of replicas (and thereby reduce its resource usage) without introducing violations.

Procedure 7 iteratively tries to decrement the number of replicas and to identify one replica that can be removed without violating the QoS constraint. The algorithm stops when no more replicas can be removed.

When Voilà decides to change the number or location of replicas, it asks Kubernetes to create/delete pods accordingly.

V. Evaluation

A. Experimental setup

We evaluate Voilà with both experimental measurements and simulations. The experimental setup consists of 22 Raspberry Pi (RPi) model 3B+ single-board computers acting as fog computing servers. Such machines are frequently used to prototype fog computing infrastructures [3], [21], [24]. They run HypriotOS Linux v1.9.0 with kernel 4.14.34, Docker v18.04.0 and Kubernetes v1.9.3. We implemented Voilà on top of Serf v0.8.2.dev and the development version of Proxy-mity.

The RPis are organized with one master node and 21 worker nodes capable of hosting replicas. Every worker node also acts as a WiFi hotspot and a Kubernetes gateway so end users can connect to the WiFi network and send requests to the service.

We emulate a realistic workload based on a trace of geo-distributed Internet requests in the province of Trentino in Italy [2]. Every request is tagged with a location at 1 km granularity of the base station it was addressed to. We randomly select 22 1 km^2 cells and inject the load of each cell in a different testbed gateway. The application is a simple web server which returns the IP address of the serving pod, such that the request processing time is almost zero.

We emulate realistic inter-node latencies using the Linux `tc` command. Latency values are defined as a linear function of the geographical distance between the cells. They range from 4 ms to 80 ms with a median of 26 ms, which arguably represents a typical situation for a fog computing infrastructure.

We also perform scalability analysis using a simulator which simulates up to 500 virtual nodes using the same latencies and workload distributions, as well as the same algorithm implementations as in Voilà to select replica placements.

B. Autoscaling behavior

We first evaluate Voilà on the testbed based on parameters shown in Table II. Figure 3 shows 28 hours of workload from the Trentino trace, and Voilà's autoscaling behavior when confronted to this workload. We sped up the trace so every hour in the trace is replayed over 2 min in the experiment.

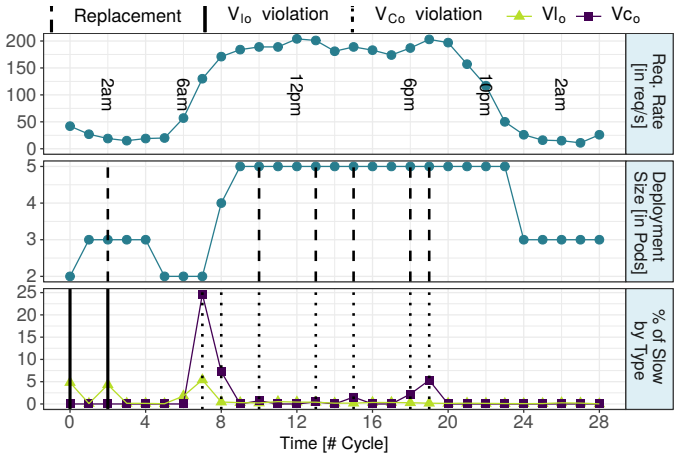


Fig. 3. Autoscaling over a 28-hour workload trace (testbed experiment).

TABLE II
TESTBED EVALUATION PARAMETERS.

Variable	Value	Variable	Value
l_o	15 ms	n	22 nodes
c_o	50 req/pod/s	$\|\hat{G}\|$	18 nodes
$\mathcal{E}_o$	0.5%	Cycle duration τ	120 s

When the application is deployed at 12am on the first day, the initial replica placement algorithm creates two replicas. We however notice that, although the workload intensity is fairly low, about 5% of requests are being slow, mainly because of network latency between the gateways and the replicas. At the next cycle Voilà creates a third replica, which fixes this QoS violation. At 2am another violation occurs, but a replica replacement is sufficient to solve this issue. Between 7am and 9am we observe a strong workload increase. Voilà detects a capacity saturation violation and reacts by bringing the number of replicas to 5 such that the violation is resolved and the number of slow requests gets back to almost zero.

From 10am until 8pm the global load stabilizes close to its daily peak. However, as discussed in Section II-C this "stable" workload still observes many changes in the users' locations. We observe that Voilà adjusts to these changes by issuing a number of replica relocation operations. Finally, when traffic decreases in the evening, Voilà scales the system down to three replicas after observing three cycles with no QoS violations.

We conclude that Voilà effectively controls the number and location of replicas. Only $\mathcal{E}_T = 2.6\%$ of all requests were categorized as slow: 0.59% because of proximity violations, and 2.01% because of capacity saturation violations.

C. Scaling up before saturation violations take place

The main reason for saturation violations is that any replica creation takes a few dozen seconds before the new replica becomes operational. If Voilà triggers a scale-up only after observing a saturation violation, then many requests may get penalized in the mean time. A practical solution to mitigate this effect consists of defining a safety margin and triggering scale-up operations to handle potential capacity saturation issues *before* saturation actually takes place.

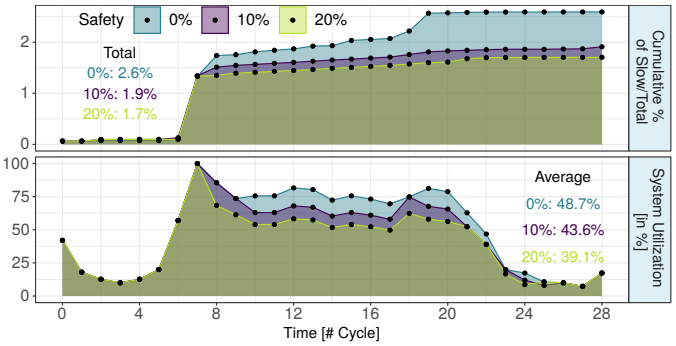


Fig. 4. Triggering scale-up early (testbed experiment).

Figure 4 shows the resource utilization of the busiest pod and the cumulative fraction of slow requests among the trace with *safety margins* 0%, 10% and 20% of actual pod capacity, which respectively trigger adaptation when any pod's workload reaches 100%, 90% and 80% capacity. Larger safety margins reduce the number of capacity saturation violations from $\mathcal{E}_T = 2.6\%$ to $\mathcal{E}_T = 1.7\%$ with *safety* $= 20\%$. However, because replicas are created sooner, the resource utilization through the day reduces slightly, from 48% to 39%. Better QoS comes at a greater cost in terms of resource usage.

Further reducing the number of violations would require predictive traffic models capable of anticipating the 9am traffic surge sufficiently early. We leave this topic for future work.

D. Sensitivity analysis

We now explore Voilà's performance by means of simulations in a 200-node system with 100 active gateways. We set $\mathcal{E}_o = 1\%$, and define default parameter values $c_o = 100\,req/pod/s$, $l_o = 20\,ms$ and *safety* $= 20\%$.

Figure 5 shows the system behavior when varying c_o, l_o and *safety*. Each test was repeated 50 times using different load distributions of 28 cycles from the Trentino grid. Every plot displays the average and the 95%-confidence interval, except the deployment size plot where the error bars depict the minimum and maximum sizes reached during the tests.

1) Deployment size: When c_o and l_o have low values, Voilà compensates by adding pods. Similarly, larger safety margins imply that nodes are less utilized, which requires more pods.

2) Slow Requests: The number of slow requests decreases when we increase the value of c_o: a smaller number of high-capacity pods can better absorb traffic intensity variations. Similarly, increasing the safety margin reduces the saturation violations. Varying l_o shows two effects: first, strict latency requirements with a low value of l_o makes latency-aware placement more difficult, which results in greater numbers of proximity violations. More surprisingly, larger values of l_o result in an increase in saturation violations. The reason is that when a gateway becomes active, its produced traffic increases over a short time period. When l_o is high, Voilà creates less replicas, so the violation occurs at a higher load rate which leads to more slow requests.

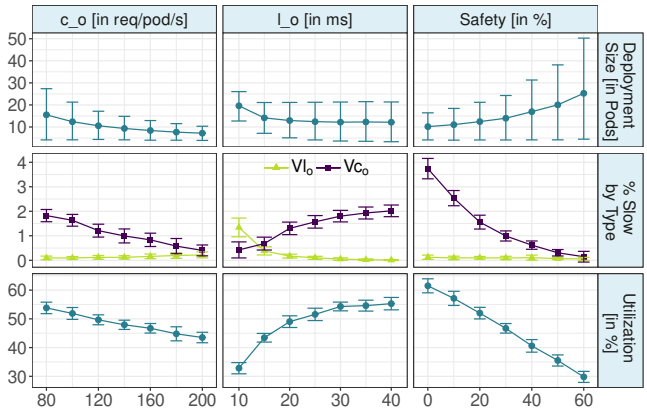


Fig. 5. Sensitivity analysis (simulator).

3) Resource utilization: When the capacity c_o of every pod increases, their placement becomes increasingly dictated by latency considerations. Their average utilization therefore decreases. We observe a similar effect with low latency thresholds l_o where the many replicas that are created to cover the relevant areas of the network actually receive a modest workload each. Finally, as previously observed, increasing the safety margin decreases resource utilization.

4) Voilà under extreme values of l_o: Voilà performs well even with very strict values of l_o. For example a requirement of $l_o = 10\,ms$ is very challenging because in our experiments only 7.5% of the inter-node latencies are below $10\,ms$. In this case Voilà still maintains $\mathcal{E}_T < 1.7\%$, yet at the expense of large number of replicas with low resource utilization.

E. Scalability

We finally evaluate the execution time for various system sizes. All measurements are done on a quad-core Intel Core i7-7600U @2.80GHz laptop, $c_o = 100\,req/pod/s$, $l_o = 20\,ms$, $\mathcal{E}_o = 1\%$, half of the gateways transmitting load and over 10 runs with 28 cycles for each test.

Figure 6 compares the average number of placement that can be studied per second for various system sizes with the average number of placements that must be evaluated to repair a latency or capacity violation. When the cluster size increases, the time needed to study any single placement also increases. However, even for a large system with 500 nodes, Voilà evaluates $\approx$ 100 placements per second. Voilà's algorithms typically need about 100-150 placement evaluations to repair a violation, regardless of system size.

VI. Conclusion

Fog computing platforms must carefully control the number and placement of application replicas to ensure guaranteed proximity between the users and the replicas serving their requests, while avoiding replica overload and reducing resource consumption as much as possible. To our best knowledge, Voilà is the only proposed system which satisfies these three objectives even in challenging situations, and has been integrated in a popular container orchestration platform.

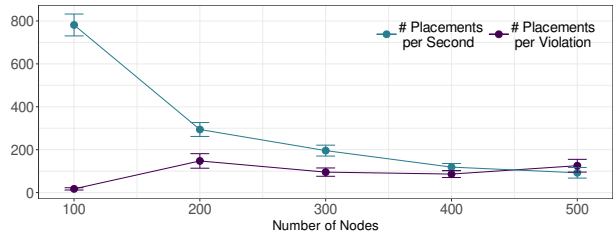


Fig. 6. Scalability (simulator).

References

[1] Arif Ahmed, HamidReza Arkian, Davaadorj Battulga, Ali J. Fahs, Mozhdeh Farhadi, Dimitrios Giouroukis, Adrien Gougeon, Felipe Oliveira Gutierrez, Guillaume Pierre, Paulo R. Souza Jr, Mulugeta Ayalew Tamiru, and Li Wu. Fog computing applications: Taxonomy and requirements. *CoRR*, abs/1907.11621. http://arxiv.org/abs/1907.11621.

[2] Gianni Barlacchi, Marco De Nadai, Roberto Larcher, Antonio Casella, Cristiana Chitic, Giovanni Torrisi, Fabrizio Antonelli, Alessandro Vespignani, Alex Pentland, and Bruno Lepri. A multi-source dataset of urban life in the city of Milan and the Province of Trentino. *Scientific data*, 2(1), 2015.

[3] Paulo Bellavista and Alessandro Zanni. Feasibility of fog computing deployment based on Docker containerization over RaspberryPi. In *Proc. ACM ICDCN*, 2017.

[4] Michael Chima Ogbuachi, Anna Reale, Péter Suskovics, and Benedek Kovács. Context-aware Kubernetes scheduler for edge-native applications on 5G. *Journal of Communications Software and Systems*, 16(1), 2020.

[5] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. In *Proc. ACM SIGCOMM*, 2004.

[6] Mohammed S. Elbamby, Cristina Perfecto, Mehdi Bennis, and Klaus Doppler. Toward low-latency and ultra-reliable virtual reality. *IEEE Network*, 32(2), 2018.

[7] Ali J. Fahs and Guillaume Pierre. Proximity-aware traffic routing in distributed fog computing platforms. In *Proc. ACM/IEEE CCGrid*, 2019.

[8] Ali J. Fahs and Guillaume Pierre. Tail-latency-aware fog application replica placement. In *Proc. ICSOC*, 2020.

[9] Ghobaei-Arani, Alireza Souri, and Ali A. Rahmanian. Resource management approaches in fog computing: a comprehensive review. *Journal of Grid Computing*, 18, 2019.

[10] HashiCorp. Serf: Decentralized cluster membership, failure detection, and orchestration. https://www.serf.io/.

[11] Mithun Mukherjee, Suman Kumar, Constandinos X. Mavromoustakis, George Mastorakis, Rakesh Matam, Vikas Kumar, and Qi Zhang. Latency-driven parallel task data offloading in fog computing networks for industrial applications. *IEEE Transactions on Industrial Informatics*, 16(9), 2019.

[12] Udi Nachmany. Kubernetes: Evolution of an IT revolution, 2018. https://bit.ly/3fX763x.

[13] Optimizely. The most misleading measure of response time. White paper, 2013. https://bit.ly/3boHgnZ.

[14] José Santos, Tim Wauters, Bruno Volckaert, and Filip De Turck. Towards network-aware resource provisioning in Kubernetes for fog computing applications. In *Proc. NetSoft*, 2019.

[15] Lakshminarayanan Subramanian, Venkata N. Padmanabhan, and Randy H. Katz. Geographic properties of Internet routing. In *Proc. Usenix ATC*, 2002.

[16] The Kubernetes Authors. Kubernetes. https://kubernetes.io/, 2019.

[17] Klervie Toczé and Simin Nadjm-Tehrani. A taxonomy for management and optimization of multiple resources in edge computing. *Wireless Comm. and Mobile Computing*, 2018.

[18] Klervie Toczé and Simin Nadjm-Tehrani. ORCH: Distributed orchestration framework using mobile edge devices. In *Proc. IEEE ICFEC*, 2019.

[19] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11), 2009.

[20] Rob van der Meulen. What edge computing means for infrastructure and operations leaders. Gartner white paper, 2018. https://gtnr.it/3euQbFh.

[21] Alexandre van Kempen, Teodor Crivat, Benjamin Trubert, Debaditya Roy, and Guillaume Pierre. MEC-ConPaaS: An experimental single-board based mobile edge cloud. In *Proc. IEEE Mobile Cloud*, April 2017.

[22] Duc-Nghia Vu, Nhu-Ngoc Dao Yongwoon Jang, Woongsoo Na, Young-Bin Kwon, Hyunchul Kang, Jason J. Jung, and Sungrae Cho. Joint energy and latency optimization for upstream IoT offloading services in fog radio access networks. *Transactions on Emerging Telecommunications Technologies*, 30(4), 2019.

[23] Nan Wang, Blesson Varghese, Michail Matthaiou, and Dimitrios S. Nikolopoulos. ENORM: A framework for edge node resource management. *IEEE transactions on services computing*, 2017.

[24] Cecil Wöbker, Andreas Seitz, Harald Mueller, and Bernd Bruegge. Fogernetes: Deployment and management of fog computing applications. In *Proc. IEEE/IFIP NOMS*, 2018.

[25] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P.Jue. All one needs to know about fog computing and related edge computing paradigms: a complete survey. *Journal of Systems Architecture*, 98, 2019.

[26] Ashkan Yousefpour, Genya Ishigaki, Riti Gour, and Jason P. Jue. On reducing IoT service delay via fog offloading. *IEEE Internet of Things Journal*, 5(2), 2018.

[27] Wei-Sheng Zheng and Li-Hsing Yen. Auto-scaling in Kubernetes-based fog computing platform. In *Proc. ICS*, 2018.