# Proximity-Aware Traffic Routing in Distributed Fog Computing Platforms

Ali Fahs, Guillaume Pierre

## To cite this version:

Ali Fahs, Guillaume Pierre. Proximity-Aware Traffic Routing in Distributed Fog Computing Platforms. CCGrid 2019 - IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing, May 2019, Larnaca, Cyprus. pp.1-10, 10.1109/CCGRID.2019.00062 . hal-02048965

**HAL Id: hal-02048965**

**https://hal.inria.fr/hal-02048965**

Submitted on 26 Feb 2019

# Proximity-Aware Traffic Routing in Distributed Fog Computing Platforms

Ali J. Fahs
*Univ Rennes, Inria, CNRS, IRISA*
Rennes, France
ali.fahs@irisa.fr

Guillaume Pierre
*Univ Rennes, Inria, CNRS, IRISA*
Rennes, France
guillaume.pierre@irisa.fr

***Abstract*—Container orchestration engines such as Kubernetes do not take into account the geographical location of application replicas when deciding which replica should handle which request. This makes them ill-suited to act as a general-purpose fog computing platforms where the proximity between end users and the replica serving them is essential. We present `proxy-mity`, a proximity-aware traffic routing system for distributed fog computing platforms. It seamlessly integrates in Kubernetes, and provides very simple control mechanisms to allow system administrators to address the necessary trade-off between reducing the user-to-replica latencies and balancing the load equally across replicas. `proxy-mity` is very lightweight and it can reduce average user-to-replica latencies by as much as 90% while allowing the system administrators to control the level of load imbalance in their system.**

***Index Terms*—Fog Computing, Proximity-Awareness, Load-Balancing, Kubernetes.**

## I. Introduction

Fog computing extends datacenter-based cloud platforms with additional resources located in the immediate vicinity of the end users. By bringing computation where the input data was produced and the resulting output data will be consumed, fog computing is expected to support new types of applications which either require very low network latency to their end users (e.g., augmented reality applications [1]) or produce large volumes of data which are relevant only locally (e.g., IoT-based data analytics).

Geographical proximity of the computing resources with the end users makes fog computing platforms very different from traditional datacenter-based clouds: when cloud platforms aim to concentrate huge amounts of computing resources in a small number of data centers, fog computing rather aims to distribute resources as broadly as possible across some geographical area so some resources are always located in the immediate vicinity of every end user's devices.

A large range of fog computing applications such as fog-assisted social networks and autonomous driving systems will need to serve numerous users or devices at the same time. To maintain proximity, these applications deploy multiple instances in relevant locations and provide a homogeneous interface to their users through the use of classical data partitioning and/or (partial) replication techniques. In this model, from a functional point of view any interaction with the application may be addressed to any instance of the application, but performance-wise it is highly desirable that interactions are addressed to nearby nodes.

A geo-distributed system such as a fog computing application must necessarily choose a suitable trade-off between resource proximity and load-balancing. A system which would always route every request to the closest instance may face severe load imbalance between instances if some users create more load than others [2]. On the other hand, systems like Mesos [3], Docker Swarm [4] and Kubernetes [5] implement location-*un*aware traffic redirection policies which deliver excellent load-balancing between application instances but very suboptimal user-to-resource network latencies.

In this paper we propose `proxy-mity`, a proximity-aware request routing plugin for Kubernetes. We chose Kubernetes as our base system because it matches many requirements for becoming an excellent fog computing platform: it can exploit even very limited machines thanks to its usage of lightweight containers rather than VMs, while remaining highly scalable and robust in highly dynamic and unstable computing infrastructures. Our approach can however easily be adapted to integrate in other container orchestration systems.

`proxy-mity` exposes a single easy-to-understand configuration parameter $\alpha$ which enables system administrators to express their desired trade-off between load-balancing and proximity (defined as a low user-to-instance network latency). It integrates seamlessly within Kubernetes and introduces very low overhead. In our evaluations, it can reduce the end-to-end request latencies by up to 90% while allowing the system administrators to control the level of load imbalance in their system.

This paper is organized as follows. In Section II, we discuss the background and related work. In Section III, we present the design of `proxy-mity`, and in Section IV we evaluate its performance. Finally, in Section V we conclude.

## II. Related work

### A. Background

Kubernetes is an open-source container orchestration platform which automates the deployment, scaling and management of containerized applications on large-scale computing infrastructures [5]. Kubernetes deploys every application instance in a *Pod*, which is a tight group of logically-related containers running in a single worker node. The containers which

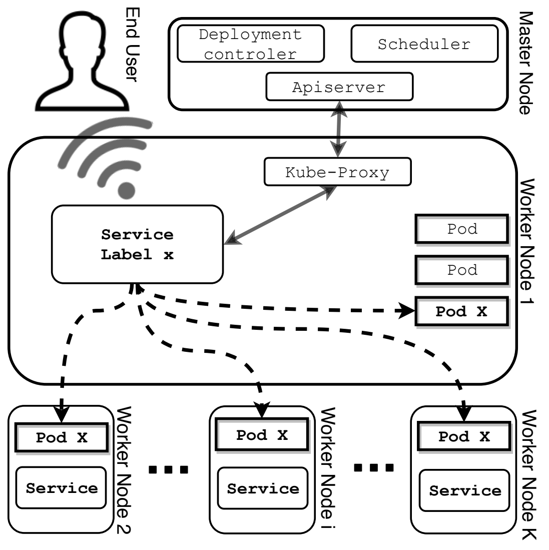


Fig. 1: Organization of a Kubernetes service.

belong to the same pod expose a single private IP address to the rest of the Kubernetes system, and they can communicate with one another using an isolated private network.

Pods are usually not managed directly by the developers. Rather, application developers are expected to create a *Deployment Controller* which is in charge of the creation and management of a set of identical pods providing the expected functionality. A Deployment Controller can dynamically add and remove pods to/from the set, for example to adjust the processing capacity according to workload variations or to deal with end-user mobility.

As illustrated in Figure 1, a set of identical pods can be made publicly accessible to external end users by creating a *Service* which exposes a single stable IP address and acts as a front end for the entire set of pods.

Although a Kubernetes service is conceptually a single component, it is implemented in a highly distributed manner. When an end user sends a request to an application's service IP address, this request is routed to one of the application's pods following a two-step process.

**First step: routing external requests to the Kubernetes system.** A variety of mechanisms such as DNS redirection and software-defined networking must be used to route the request to any node belonging to the Kubernetes system. This means in particular that every node in Kubernetes (which may or may not contain a pod of the concerned application) can actually act as a *Gateway node* between the end users and the Kubernetes system.

When using Kubernetes as a fog computing platform, we assume that the fog system is somehow able to route end user traffic to a nearby gateway node. In our implementation every fog compute node also acts as a WiFi hotspot to which end user's devices may connect to access the system. This organization naturally routes every request to a Kubernetes node in a single wireless network hop. Other implementations

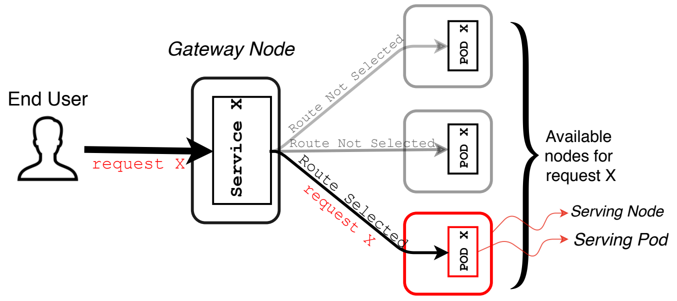


Fig. 2: Gateway and Serving Nodes.

may rely on a wide variety of technologies such as LTE and SDN to provide the same functionality. Note that every fog computing platform must necessarily implement such a mechanism, otherwise it would not be able to provide any form of network proximity between the end users and the fog resources serving them.

**Second step: internal request routing to one of the application's pods.** The request is further routed internally to the Kubernetes cluster using DNAT *(Destination Network Address Translation)* which chooses one of the service pods' private IP address as a destination. When the application comprises more than one pod, Kubernetes load-balances requests equally between all available pods.

Internal request routing is implemented by a daemon process called *kube-proxy* which runs in every Kubernetes node. When the Kubernetes master node detects a change in the set of pods belonging to a service, it sends a request to all kube-proxy daemons to update their local routing tables. Such changes may be caused by an explicit action from the system such as starting or stopping a pod, or by a variety of failure scenarios.

The kube-proxy daemons are in charge of updating the kernel-level routing configuration using iptables or IPVS. For each service's IP address, kube-proxy creates rules which load-balance incoming connections among the service pods' private IP addresses. As shown in Figure 2, network traffic is addressed toward a *Serving node* which contains the actual *Serving pod* that will process the incoming request.

In the standard Kubernetes implementation, every incoming connection has an equal probability to be processed by each of the service pods. This load-balancing strategy is very sensible in a cluster-based environment where all service pods are equivalent in terms of their functional and non-functional properties. However, it clearly does not fit our requirement of proximity-based routing in the context of a fog computing platform. The objective of our work is therefore to re-design the internal network routing in Kubernetes such that every gateway node creates specific local routes that favorize nearby pods. In a broadly geo-distributed system such as a fog computing platform this will significantly decrease the mean and standard deviation of network latencies experienced by the end users, thereby providing them with a better and more predictable user experience.

## B. State of the art

Fog computing aims at providing compute, storage and networking resources that are geographically distributed across a geographical area such as a building, a neighborhood and a city [6], [7]. Much like cloud platforms, fog platforms rely on virtualization techniques to support multi-tenancy and increase resource utilization. For resource-efficiency reasons, most fog computing platforms rely on lightweight container technologies [8] rather than virtual machines, enabling one to envisage fog computing platforms making use of resources as limited as Raspberry Pis for example [9], [10].

The main purpose of fog computing is to allow applications to use resources located in the immediate vicinity of the end users. This for example enables applications which can offload parts of their tasks to the fog infrastructure. Optimizing the trade-off between latency and using the limited edge resources is relevant for example for video analytics applications [11]. Such applications must carefully control the trade-off between user-perceived latency and energy consumption [12].

When multiple fog nodes provide equivalent functionality, it is the task of a load balancer to choose the best available node which should process each end user request. *Puthal et al.* propose a secure and sustainable load balancing technique which aims to distribute the load to the less-loaded edge data centers [13]. Similarly, *Beraldi et al.* propose a cooperative load balancing technique to distribute the load over different edge data centers to reduce the blocking probability and the task overall time [14]. However, these techniques do not aim to reduce the network latency between the end user and the fog node serving them. Also, they were evaluated using simulations only, with no actual system implementation.

*Kapsalis et al.* propose a fog-aware publish-subscribe system which aims to deliver messages to the best possible node according to a combination of network latency, resource utilization and battery state [15]. To our best knowledge this is the only proposed fog system which aims, similarly to our work, at implementing a trade-off between proximity and fair load balancing. However, this approach was implemented and evaluated only in simulation. It is unclear how network latencies, resource utilization and battery states would be measured in a real implementation nor how messages would be routed to their destination without being dispatched by a single central broker node.

PiCasso is a container orchestration platform that specifically targets edge clouds with a focus on lightness and platform automation [16]. The developers of PiCasso intend to develop a service proxy that will redirect user's request to the closest node. However, no technical detail is provided. PiCasso is still under development and not publicly available.

OpenStack++ is a set of OpenStack extensions that enable the deployment of Cloudlets in an edge computing environment [17]. However, applications in OpenStack++ are implemented as a single (migratable) VM so requests addressed to any application always have only a single possible destination.

Fogernetes is an extension of Kubernetes which, similarly to our work, aims to extend Kubernetes for fog computing scenarios [18]. Fogernetes and `proxy-mity` rely on the same application model based on pods and replication controllers. However, in Fogernetes, the selection of which pod should be used to process which request is done manually through the use of node labels. In contrast, `proxy-mity` aims to automate the pod selection to implement any trade-off between proximity and fair load balancing defined by the platform's administrator.

## III. System design

`proxy-mity`[1] is a plug-in designed to integrate in a Kubernetes system and implement proximity-aware traffic routing. It however has very few dependencies with Kubernetes and may arguably be adapted to work in different platforms.

Similarly to the standard kube-proxy Kubernetes component, `proxy-mity` is deployed in every worker node of the system. It continuously monitors network latencies with the other worker nodes using Serf [19], a lightweight implementation of Vivaldi coordinates [20]. When a `proxy-mity` daemon detects a change in the set of pods belonging to any service, it recomputes a new set of traffic routing rules (with their weights determining the probability that a request follows each route) according to preferences expressed by the system administrator, and injects them in the local Linux kernel using *iptables*.

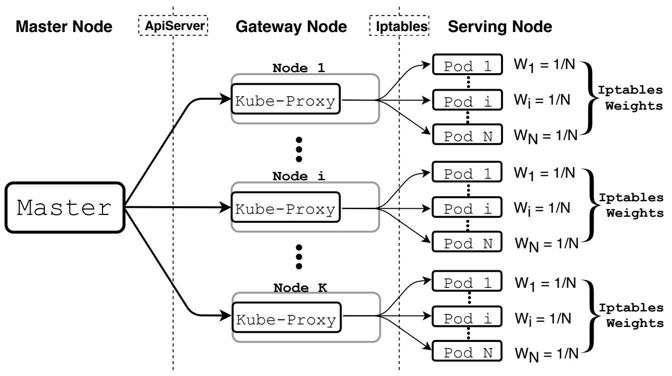
In the next sections we respectively discuss the overall system architecture, the representation and measurement of proximity between nodes, the calculation of weights to be associated with each route, and the injection of new routes in the local Linux kernel.

### A. Architecture

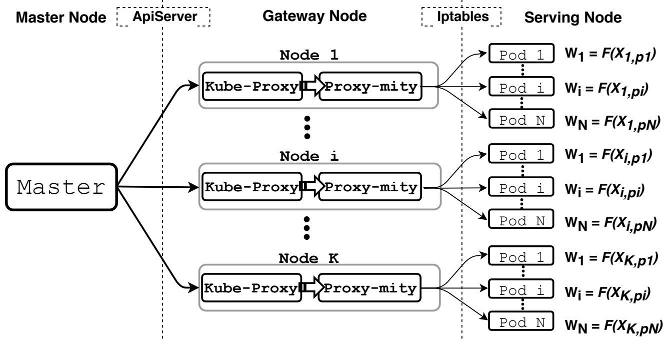
Kubernetes is designed as a set of control loops. It therefore continuously monitors itself, and takes corrective actions when the state it observes deviates from the specification of the desired system state. This organization makes it highly dynamic and robust against a wide range of situations. The master node maintains a view of the current system state which can be queried by other components.

As shown in Figure 3a, in unmodified Kubernetes a kube-proxy daemon is started in every worker node to maintain its local *iptables* routes. When a change is detected in the set of pods belonging to a service (caused by a pod start or stop operation or by any kind of failure), all kube-proxy daemons re-inject new routes in their local *iptables* system. All kube-proxy daemons inject the same set of rule which ensures that every pod from the application receives an equal $1/N$ share of the load (where $N$ is the number of pods of the application). This ensures excellent load balancing between the pods. However, in a fog computing scenario where nodes are broadly geo-distributed, it actually routes significant amounts of end user requests to pods located far away from them. This results in unacceptably high mean network latencies, and also in very high standard deviations.

[1]https://github.com/alijawadfahs/FOG-aware



(a) kube-proxy architecture.



(b) `proxy-mity` architecture.

Fig. 3: Architectures of kube-proxy and `proxy-mity`.

The architecture of `proxy-mity`, presented in Figure 3b, is very similar to that of kube-proxy. It receives the same notifications as kube-proxy upon a change in the set of pods belonging to a service. However, each kube-proxy daemon computes a specific set of weights to be attached to each route according to the measured network latencies. Different worker nodes therefore compute different sets of rules, and the weights attached to different routes in each gateway node are explicitly biased to send more load to nearby nodes.

These sets of rules are recomputed and re-injected every time a modification is detected in the set of pods which constitute an application, and also periodically to account for possible variations in the measured network latencies between nodes.

## B. Measuring proximity

Data center networks often follow very complex topologies to provide cloud users with many interesting properties such as excellent bisection bandwidth and resilience toward a wide range of possible disruptions. They often have excellent performance which means that the inter-node latencies within a data-center are usually low enough to be ignored in practice. Nodes in a data center may be located far from the end users, but they are very close from each other. This is the reason why orchestration systems such as Kubernetes that were designed for data center environments do not make any attempt at routing end user requests to nearby nodes.

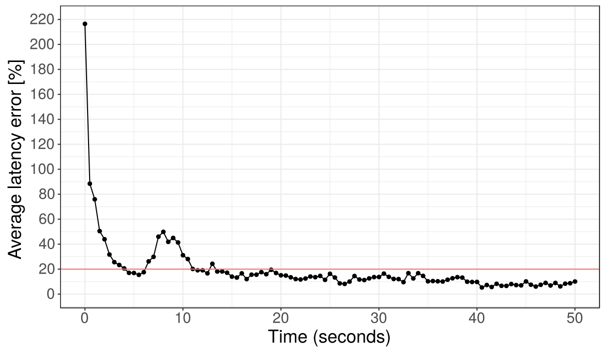


Fig. 4: Accuracy of Vivaldi latency predictions for a newly joined node in a 12-node cluster.

However, in fog computing environments resources are physically distributed across some geographical area in order to provide compute, storage and networking resources in the immediate vicinity of the end users. Nodes in this case will be close to the end users, but necessarily far from each other.

There are many ways to represent proximity. For example, the broad availability of inexpensive GPS receivers makes it easy to measure the geographical distance between nodes. However, geographical distance is known to be a poor predictor for route lengths or latencies in large-scale network infrastructures [21]. We therefore prefer directly relying on network latency as the measure of proximity.

To avoid the overhead of periodically measuring $N^2$ pairwise latencies between $N$ nodes, `proxy-mity` relies on Vivaldi coordinates [20] for modeling the latencies between nodes. Vivaldi is a distributed, lightweight algorithm to accurately predict the latency between hosts without contacting them. Using Vivaldi, a node in the cluster can easily compute the latency with all the nodes by communicating with a few of them.

We specifically use Serf [19], a mature open-source tool which maintains cluster membership, detects failures, and offers a robust implementation of Vivaldi coordinates. Serf is based on a gossiping protocol where each node periodically contacts a set of randomly-selected other nodes, measures latencies to them, and adjusts their Vivaldi coordinates accordingly. Latency between any pair of nodes is modeled as the Euclidean distance between their respective Vivaldi coordinates. The end result is a lightweight and robust system which can produce accurate predictions of inter-node latencies.

Figure 4 depicts the accuracy of latency predictions produced by Serf. Immediately after a fresh node joins a 12-node cluster, its latency predictions are highly inaccurate. However, the system converges very quickly. Roughly 20 seconds after startup, the prediction error consistently remains below 20%, and stabilizes in the order of 10%. In a fog computing system where latencies between nodes are expected to belong to a very wide range of values, this level of prediction accuracy is largely sufficient to distinguish a nearby node from a further away one.

## C. Weight calculation

In Kubernetes, a set of identical pods is called a *Deployment*. A Kubernetes service associates a single IP address to such a set of pods to which incoming requests are distributed.

Consider a deployment $\Phi$ composed of $N$ functionally identical pods:

$$\Phi = \{\varphi_1, \varphi_2, ..., \varphi_N\}$$

where each $\varphi$ represent one pod in this deployment.

A Kubernetes service essentially implements a map function which determines the probability that an incoming connection gets routed to each of these pods. Kubernetes' kube-proxy component implements a very simple mapping function:

$$F(\varphi_i) = 1/N \quad \forall \varphi_i \in \Phi$$

We can however generalize this formula to any function $F$ which respects:

$$F : \Phi \longrightarrow [0,1] \quad | \quad \sum_{i=1}^{N} F(\varphi_i) = 1$$

As previously discussed, a request routing system for fog computing environments must necessarily implement a trade-off between proximity and load balancing. A system which optimizes based on proximity only risks severe load imbalances between pods in case different numbers of requests are generated in different geographical areas of the system. On the other hand, balancing the load equally among pods will result in larger means and standard deviations of the latencies between the users and the pods serving them.

We address this challenge by proposing two mapping functions $P$ (which aims for proximity regardless of load balancing) and $L$ (which aims at load balancing regardless of proximity). These two functions can be combined in a single function $F_\alpha$:

$$F_\alpha(\varphi) = \alpha.P(\varphi) + (1-\alpha).L(\varphi) \tag{1}$$

Here $\alpha \in [0,1]$ is a parameter chosen by the system administrator which represents the desired trade-off between pure load-balancing (when $\alpha = 0$) and pure proximity-based routing (when $\alpha = 1$).

Function $L$, which aims to balance the load, is the same as the original Kubernetes one:

$$L(\varphi_i) = 1/N \quad \forall \varphi_i \in \Phi$$

Function $P$, which aims at maximizing proximity, takes into account the estimated network latencies between the local node and all the possible serving nodes in the system. These latencies are represented by the set $\mathbb{L} = \{l_1, l_2, ..., l_N\}$ where $l_i$ represents the network latency to the physical node which holds pod $\varphi_i$. In fact, any function where nodes with lower latencies are given greater weight than further away nodes may act as the proximity-maximizing decay function:

$$P(\varphi_i) = \frac{f_\beta(l_i)}{\sum_{j=1}^{N} f_\beta(l_j)} \tag{2}$$

where $f_\beta(l_i)$ is a weight determined from the estimated latency to every node. We use the secondary parameter $\beta$ to determine how aggressive the proximity-oriented function should be to favorize nearby nodes.

We propose three possible decay functions to determine the weights $f_\beta(l)$:

$$f_\beta^{inverse}(l) = \frac{1}{\beta l} \tag{3}$$

$$f_\beta^{power}(l) = \frac{1}{l^\beta} \tag{4}$$

$$f_\beta^{exponential}(l) = e^{-\beta l} \tag{5}$$

As we will discuss in Section IV, different decay functions have different levels of aggressiveness in selecting nearby pods.

The final weight function $F_{\alpha,\beta}(\varphi)$ is therefore:

$$F_{\alpha,\beta}(\varphi_i) = (1-\alpha).\frac{1}{N} + \alpha.\frac{f_\beta(l_i)}{\sum_{j=1}^{N} f_\beta(l_j)} \quad \forall \varphi_i \in \Phi \tag{6}$$

A special case in the computation of weights relates to fact that the node which computes new weights may also hold a pod of the concerned application. In this case, the latency attached to the localhost interface may be as low as 0.3 ms. When applying formulas 3, 4 or 5 this results in giving the localhost route an extremely high probability compared to the other pods of the application. To avoid this effect, we artificially increase the localhost latencies in the weight calculation by a parameter *localrtt* that is set to be slightly lower than the lowest inter-node latencies observed in the deployed system.

## D. Updated routes injection

Once every node in `proxy-mity` has computed the fraction of requests it should route to every other node, the last step is to inject the corresponding routes in the Linux kernel firewall in the form of *iptables* rules. *iptables* defines chains of rules for the treatment of packets where every chain is associated with a different kind of packet processing. Packets are processed by sequentially traversing the rules in their chains.

As illustrated in Figure 5, iptables rules are organized in five chains. Incoming packets first traverse the PREROUTING chain, then they get split between two chains. The packets whose destination IP address is locally available are sent to the INPUT chain for immediate delivery. Other packets traverse the FORWARD chain which decides where they should be sent next. On the other hand, the packets issued from the local node traverse the OUTPUT chain. Finally, all outgoing packets traverse the POSTROUTING chain before being actually sent to the network. Kubernetes implements its internal network routing system by defining rules in the PREROUTING and

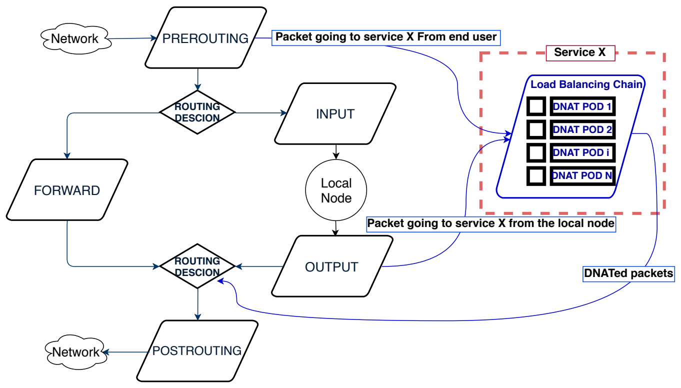


Fig. 5: Iptables chains and load balancing.

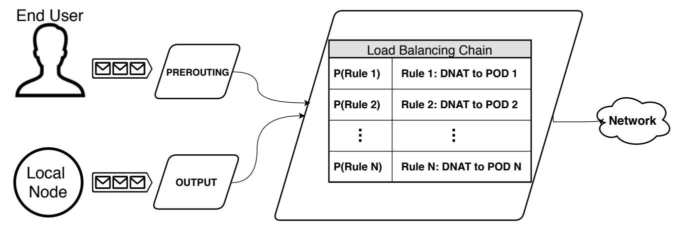
OUTPUT chains: incoming network packets whose destination address matches the IP address of a Kubernetes service are redirected using rules in the PREROUTING towards the load-balancing chain. On the other hand, the OUTPUT chain redirects the packets sent to the service by the local node itself.

As depicted in Figure 6a, every Kubernetes service is actually implemented as a separate chain which redirects packets to the respective pods using DNAT. To load-balance incoming requests among the service pods, each iptables rule $i$ defines a probability $P_i$ for incoming requests to exit the iptables chain and get routed to the corresponding pod $Pod_i$.

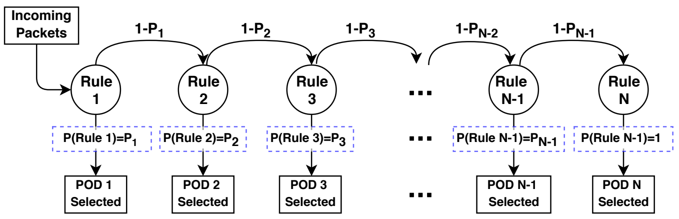
Rules are executed in a predefined sequential order, so the probabilistic load-balancing system actually implements a Markov chain, as shown in Figure 6b. Every incoming packet sent to the service first undergoes rule 1 with a probability $P_1$ of exiting the chain and of being redirected to $Pod_1$. With probability $1 - P_1$ the packet continues to the next rule. The same mechanism is used for all rules in the chain, except the last one which routes all remaining messages to the last pod with probability 1. Based on the individual probabilities $P_i$, an incoming packet will therefore eventually get routed to $Pod_i$ with probability $P(Pod_i)$:

$$P(Pod_i) = \begin{cases} P_1 & \text{if } i = 1 \\ P_i \times \prod_{j=1}^{i-1} (1 - P_j) & \text{if } 1 < i < N \\ \prod_{j=1}^{N-1} (1 - P_j) & \text{if } i = N \end{cases} \quad (7)$$

Injecting a set of weights $\mathbb{W} = \{w_1, w_2, ..., w_N\}$ as computed in Equation 6 for a deployment $\Phi$ therefore requires us to compute the probabilities $P_i$ which should be defined in the iptables Markov chain such that the resulting probabilities $P(Pod_i)$ match the desired weights $w_i$:



(a) iptables rule chains for both types of packets.



(b) Markov chain implementing the load-balancing rules for one Kubernetes service.

Fig. 6: Load balancing rules in iptables.

$$P_i = \begin{cases} w_1 & \text{if } i = 1 \\ w_i \times \frac{1}{\prod_{j=1}^{i-1} (1-P_j)} & \text{if } 1 < i < N \\ 1 & \text{if } i = N \end{cases} \quad (8)$$

Upon every detected modification in the set of pods belonging to a Kubernetes service, `proxy-mity` therefore recomputes the weights $w_i$ using Equation 6 in every worker node of the system based on its estimated latencies to the other nodes, and converts them into iptables rule probabilities using Equation 8 before injecting them in the local Linux kernel. The same process is also applied periodically to account for possible modifications in the estimated inter-node network latencies.

We evaluate the performance of these newly applied rules in the next section.

## IV. Evaluation

### A. Experimental setup

We evaluate `proxy-mity` using an experimental testbed composed of 12 Raspberry Pi 3 B+ single-board computers (*Rpi's*), as depicted in Figure 7. Despite their obvious hardware limitations, Raspberry PIs offer excellent performance/cost/energy ratios and are well-suited to fog computing scenarios where the devices' physical size and energy consumption are important enablers for actual deployment [22], [23].

All Rpi's are installed with HypriotOS 1.9.0 Linux distribution[2], Linux 4.4.50 kernel, and Kubernetes v1.9.3. In this setup, one machine acts as the Kubernetes master node

[2]https://blog.hypriot.com/downloads/

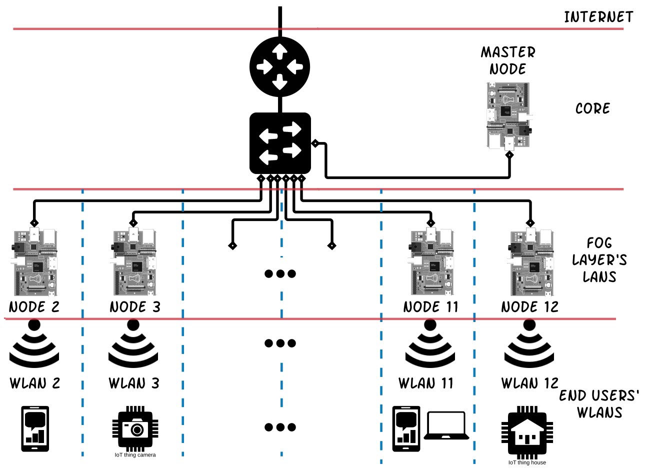


Fig. 7: Experimental testbed organization.

| | Amsterdam | Brussels | Copenhagen | Düsseldorf | Geneva | London | Lyon | Marseille | Paris | Strasbourg | Edinburgh |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Amsterdam | 0.3 | 14 | 18 | 12 | 20 | 9 | 24 | 40 | 26 | 13 | 19 |
| Brussels | 14 | 0.3 | 16 | 14 | 20 | 10 | 14 | 16 | 8 | 24 | 17 |
| Copenhagen | 18 | 16 | 0.3 | 15 | 30 | 20 | 25 | 35 | 22 | 27 | 31 |
| Düsseldorf | 12 | 14 | 15 | 0.3 | 15 | 15 | 25 | 20 | 10 | 22 | 22 |
| Geneva | 20 | 20 | 30 | 15 | 0.3 | 18 | 12 | 10 | 36 | 20 | 28 |
| London | 9 | 10 | 20 | 15 | 18 | 0.3 | 14 | 38 | 4 | 21 | 10 |
| Lyon | 24 | 14 | 25 | 25 | 12 | 14 | 0.3 | 24 | 10 | 16 | 25 |
| Marseille | 40 | 16 | 35 | 20 | 10 | 38 | 24 | 0.3 | 25 | 30 | 27 |
| Paris | 26 | 8 | 22 | 10 | 36 | 4 | 10 | 25 | 0.3 | 12 | 13 |
| Strasbourg | 13 | 24 | 27 | 22 | 20 | 21 | 16 | 30 | 12 | 0.3 | 30 |
| Edinburgh | 19 | 17 | 31 | 22 | 28 | 10 | 25 | 27 | 13 | 30 | 0.3 |

TABLE I: Inter-node network latencies (ms).

while the eleven remaining nodes act as worker nodes. These machines are connected to each other using a dedicated Gigabit Ethernet switch. Every worker node also acts as a WiFi hotspot which allows end users and external IoT devices to connect to a nearby node. Any request addressed by an end-user device to a Kubernetes service therefore reaches one of the worker nodes in a single WiFi network hop, before being further routed via the wired network to one of the service's pods using the iptables rules created by `proxy-mity`.

We create artificial network latencies between every pair of nodes using the Linux *tc* command. We use actual measurements of city-to-city network latencies as a representation of realistic pairwise latencies between geo-distributed nodes. These pairwise latencies were obtained from the WonderNetwork[3] GeoIP testing solution, and are presented in Table I. In this configuration, network latencies range from 4 ms to 40 ms and can arguably represent a typical situation for a geo-distributed fog computing infrastructure.

[3]https://wondernetwork.com/

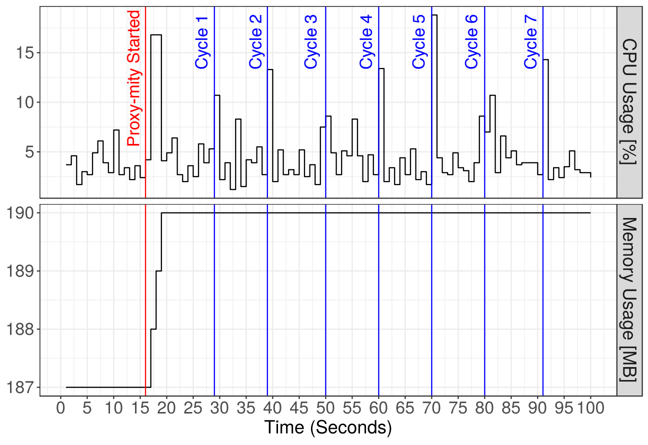


Fig. 8: CPU and Memory Usage For `proxy-mity`.

## B. Performance overhead

The `proxy-mity` load-balancing system must carry additional tasks compared to the standard kube-proxy component of Kubernetes: it must execute Serf on every worker node (which creates periodic CPU and network activity), recompute weights and inject updated routes periodically. When the fog computing platform is composed of limited devices such as Raspberry PIs it is important to keep this performance overhead as low as possible.

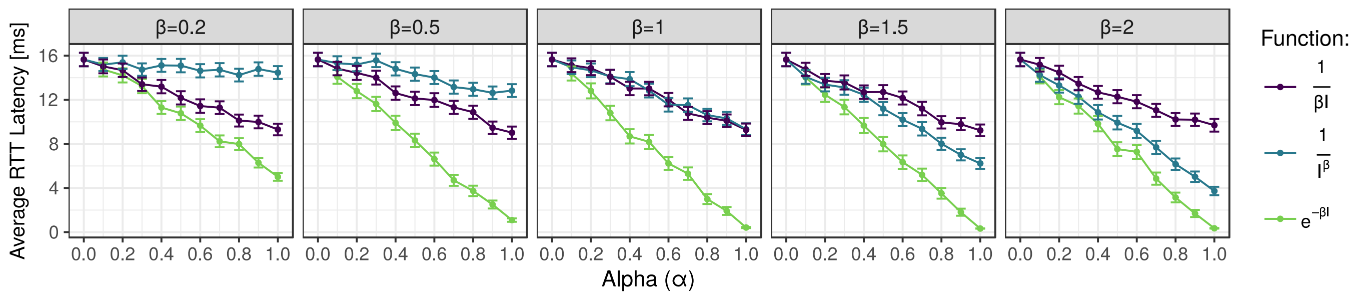
Figure 8 shows the the total node's CPU and memory usage before and after starting `proxy-mity` on one of the cluster's nodes. `proxy-mity` is configured to check for changes every 10 seconds (this is the default value in our implementation).

Before `proxy-mity` starts at time 15 s, the monitored node is already acting as an (idle) Kubernetes worker node. It uses on average 3% of CPU and 187 MB of memory. After `proxy-mity` is started the memory usage grows by only 3 MB and the average CPU usage grows by $\approx$ 2-4%. We conclude that the performance overhead, although not totally negligible, remains sufficiently low not to disturb the good behavior of worker nodes in their operations.

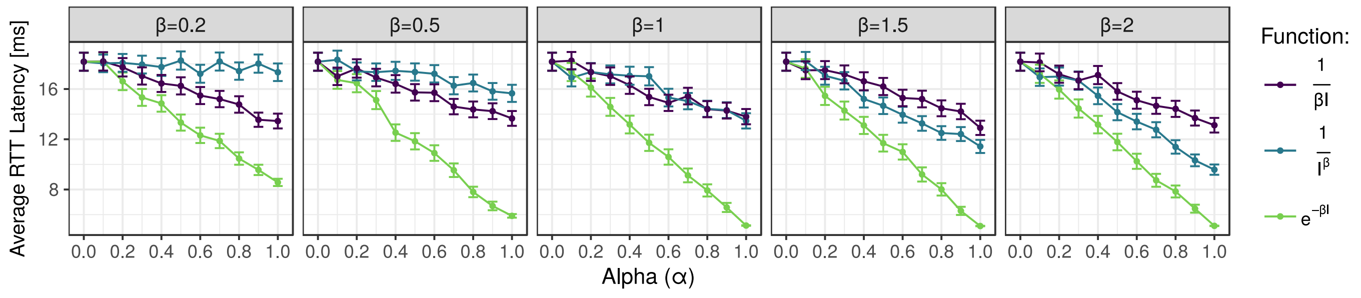
This low performance overhead also indicates that the introduction of `proxy-mity` will not significantly affect the scalability or fault-tolerance properties of Kubernetes. Besides the introduction of the very lightweight, scalable and robust Serf system, `proxy-mity` does not require a re-organization of Kubernetes processes, and simply creates iptables rules with different weights at every node.

## C. Service access latency

We first evaluate the effectiveness of `proxy-mity` in distributing load according to a given proximity/load-balancing trade-off $\alpha$ and decay function $f_\beta(l)$. We deploy `proxy-mity` with a Kubernetes service which contains a small Web server that simply returns the IP address of the serving pod to every client. The execution time of the service function itself is extremely short, so any end-to-end latency measured at the client side accurately represents the network latency that was experienced by every request.



(a) Deployment with 11 pods located in all the nodes.



(b) Deployment with 5 pods (none of them in the London gateway node).

Fig. 9: Average service access latency.

In every experiment in this section, we issue 1000 HTTP requests originating from a single node of the system (the London node), and observe the distribution of latencies experienced by these requests. The requests address a Kubernetes service which is deployed either across 11 pods (one in every worker node of the system), or only 5 pods (with none of these pods running in the London node). Having all traffic originating from a single node can be seen as a worst-case scenario for load-balancing among pods, and therefore allows us to closely observe the behavior of `proxy-mity`. The parameters for this experiment are summarized in Table II.

| | |
|---|---|
| $\alpha$ | $\{0.1, \ \ldots, \ 1\}$ |
| $\beta$ | $\{0.2, \ 0.5, \ 1, \ 1.5, \ 2\}$ |
| $f_\beta(l)$ | $\{1/\beta l, \ 1/l^\beta, \ exp(-\beta l)\}$ |
| $localrtt$ | $3\,ms$ |
| Number of pods ($\lvert\Phi\rvert$) | $\{5, \ 11\}$ |
| Transmitted requests/experiment | 1000 |

TABLE II: Evaluation parameters.

Figure 9 shows the average measured end-to-end latency for various values of $\alpha$, choices of decay function, and values of $\beta$: Figure 9a shows the results with a deployment of 11 pods, and Figure 9b shows the results with a deployment of 5 pods.

*a) Effect of parameter $\alpha$:* In all presented figures, we observe that configurations where $\alpha = 0$ experience high average latencies. This is due to the fact that requests are distributed equally among all the pods, so a significant fraction of requests gets routed over long-latency routes. This is the default Kubernetes behavior.

When $\alpha$ increases, requests experience much lower latencies, which indicates that the closest pods receive more load than the others. For example, in the case of $\{|\Phi| = 11, \ \beta = 0.5, \ f_\beta(l) = exp(-\beta l)\}$, the overall average request latency is 15.7 ms for $\alpha = 0$ but only 1.09 ms for $\alpha = 1$ (a 92% reduction of latency).

The parameter $\alpha$ therefore effectively allows the system administrator to control the latency/load-balancing trade-off: low values of $\alpha$ produce equal load balancing whereas high values of $\alpha$ favorize proximity.

*b) Effect of parameter $\beta$ and the choice of decay function:* All evaluated decay functions achieve similar results where greater values of $\alpha$ produce lower average service latencies. However, they differ in their level of aggressiveness. Unsurprisingly, the exponential function $exp(-\beta l)$ produces the fastest decay whereas the other two functions produce slower decay. The exponential function may therefore be used in scenarios where we want to strongly skew the request routing system toward proximity, whereas the other two functions may be used for implementing less skewed load distribution.

Interestingly, the choice of parameter $\beta$ does not significantly influence the end results, except for the $f_\beta(l) = \frac{1}{l^\beta}$ decay function. This is due to the fact that the shape of the chosen decay function matters more than its own parameter. In future experiments we therefore fix $\beta$ to a single "medium" value per decay function.

## D. Load distribution

We now focus more closely on the statistical distribution of request latencies. We execute the same experiment as in the previous section over a deployment of 11 pods, and measure

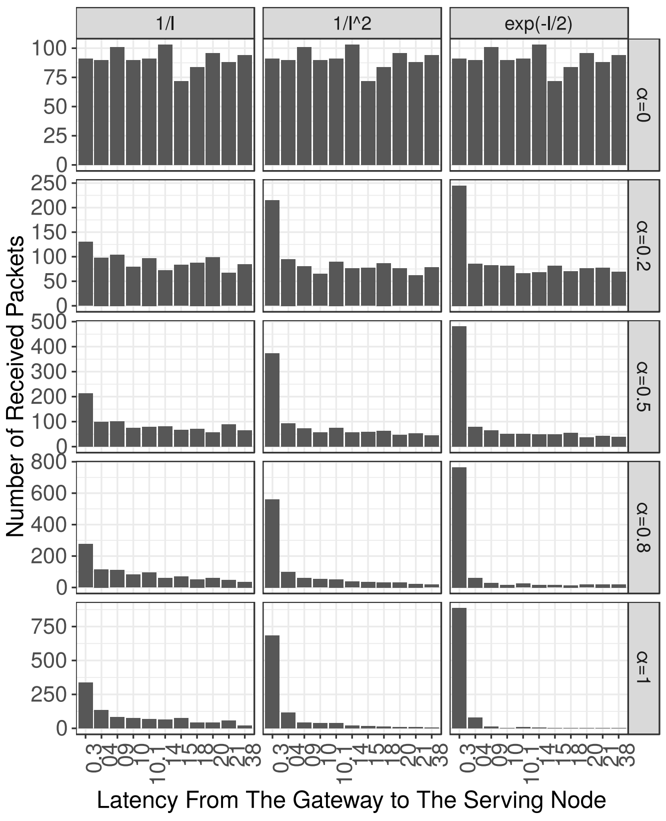


Fig. 10: Load distribution as a function of $\alpha$ and $f(l)$.

the number of requests which get routed to each pod (sorted by their latency to the gateway node).

The experiment results are presented in Figure 10. Each bar in the figure indicates the number of requests processed by a pod with the associated latency to the gateway node. We can see when $\alpha = 0$ that all the pods receive roughly the same number of requests regardless of their distance to the gateway node. The load per pod fluctuates slightly because the routing system is probabilistic and therefore experiences some amount of noise.

As the value of $\alpha$ increases, more packets get routed toward the pods with a lower latency. Finally, with $\alpha = 1$ the load is balanced only based on the proximity function, which leads to extreme skew between nodes (in particular in the case of $f(l) = exp(-l/2)$, where a single pod receives more than 90% of the total load).

The obvious possible drawback in the extreme case of $\alpha = 1$ is that a single pod which processes most of the incoming traffic might become overloaded as a result of the load imbalance. Remember however that we are producing incoming traffic at a single node only. In a setup where traffic is being generated in multiple locations, we would observe much less load imbalance between the pods, as we discuss next.

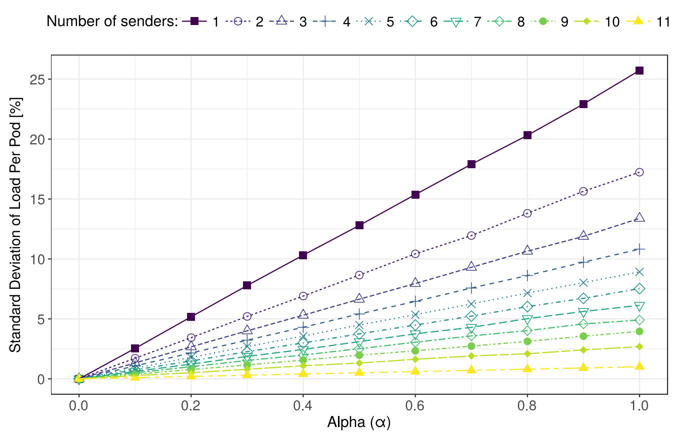


Fig. 11: Overall system load imbalance as a function of $\alpha$ and the number of senders.

### E. Load (im)balance in the presence of multiple senders

A fog computing platform has very few reasons to deploy pods in regions where no user is accessing the considered service. Evaluations where all the traffic originates from a single sender therefore represent a worst-case scenario in terms of the load imbalance it creates. A more realistic scenario where traffic originates from multiple senders in various regions of the system would arguably experience a much lower load imbalance.

To study this effect, we simulated a system where a randomly-chosen subset of the worker nodes act as traffic senders. All senders issue the same number of requests, while the other nodes do not send any request at all. Using Equation 6 with $f(l) = exp(-l/2)$ (where $\beta = 0.5$) we can compute the weights that each sender node would assign to each of the pods, based on the inter-node latencies from Table I. In the presence of multiple senders, each pod serves a large number of requests originating from nearby senders, and lower numbers of requests originating from senders located further away. We can therefore add these numbers together to compute the total load that each of the pods is expected to receive.

Figure 11 depicts the standard deviation among the predicted loads per pod, in a scenario where every node holds a pod of the service and a random subset of $k$ nodes act as traffic senders.

We observe two interesting phenomena. The first one relates to the fact that a greater number of senders naturally creates a better-balanced system. Using one sender among 11 nodes, with $\alpha = 1$, the standard deviation among predicted pods' loads is as high as 25% of the mean load among pods. When moving to two randomly-chosen senders, this standard deviation drops to 17% of the mean. The same trend continues until the scenario where all nodes act as senders: here the standard deviation drops to a mere 1% of the mean. This indicates that, although $\alpha = 1$ requires `proxy-mity` to aggressively favorize proximity and low end-to-end request latencies, geographical distribution of the traffic sources naturally helps to balance the load among pods. The more uniform the distribution of traffic sources is, the better-balanced the resulting system will be (without sacrificing the objective of proximity and low service latencies).

The second phenomenon concerns the relation between $\alpha$ and the system load imbalance, measured as the standard deviation between predicted pods' load. When $\alpha = 0$ the predicted load imbalance is obviously 0, as each sender equally distributes the load it creates between all the pods. When $\alpha = 1$ the predicted load imbalance is a function of the distribution of traffic sources. Interestingly, when $\alpha$ takes intermediate values between 0 and 1, the load imbalance varies linearly between these two extremes[4]. This means that the criteria for choosing a good value for $\alpha$ can be explained to system administrators in a precise yet very intuitive manner: *"$\alpha$ linearly controls the system imbalance between 0 when $\alpha = 0$ and some value when $\alpha = 1$ which is determined by the geographical heterogeneity of the traffic senders"*.

## V. Conclusion

Container orchestration engines such as Kubernetes do not take the geographical location of service pods into account when deciding which replica should handle which request. This makes them ill-suited to act as general-purpose fog computing platforms where the proximity between end users and the replica serving them is essential. We presented `proxy-mity`, a proximity-aware traffic routing system for distributed fog computing platforms. It seamlessly integrates in Kubernetes, and gives very simple mechanisms to allow system administrators to control the necessary trade-off between reducing the user-to-replica latencies and balancing the load equally across replicas. When the pods are geographically distributed close to the sources of traffic, `proxy-mity` drastically reduces the end-to-end service access latencies without creating major system load imbalances.

This work demonstrates that orchestration systems which were originally designed for cluster-based environments can be extended to become proximity-aware with no need for major structural changes. It paves the way toward extending some of the major orchestration systems to become mainstream, general-purpose platforms for future fog computing scenarios.

## References

[1] M. S. Elbamby, C. Perfecto, M. Bennis, and K. Doppler, "Toward low-latency and ultra-reliable virtual reality," *IEEE Network*, vol. 32, Apr. 2018.

[2] Q. Fan and N. Ansari, "Towards workload balancing in fog computing empowered IoT," *IEEE Transactions on Network Science and Engineering*, 2018. Early access.

[3] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. Usenix NSDI Symposium*, 2011.

[4] Docker Inc., "Swarm mode overview." Docker Documentation. https://docs.docker.com/engine/swarm/.

[4]This can be explained analytically but detailing the proof would exceed the scope of the current paper.

[5] Cloud Native Computing Foundation, "Kubernetes." https://kubernetes.io/.

[6] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, *Big Data and Internet of Things: A Roadmap for Smart Environments*, ch. Fog Computing: A Platform for Internet of Things and Analytics. Springer, 2014.

[7] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey." arXiv:1808.05283v2 [cs.NI], Sept. 2018.

[8] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee, "A container-based edge cloud PaaS architecture based on Raspberry Pi clusters," in *Proc. Future Internet of Things and Cloud Workshops (FiCloudW)*, 2016.

[9] F. P. Tso, D. R. White, S. Jouet, J. Singer, and D. P. Pezaros, "The Glasgow Raspberry Pi cloud: A scale model for cloud computing infrastructures," in *Proc. ICDCS Workshops*, 2013.

[10] A. van Kempen, T. Crivat, B. Trubert, D. Roy, and G. Pierre, "MEC-ConPaaS: An experimental single-board based mobile edge cloud," in *Proc. IEEE Mobile Cloud Conference*, 2017.

[11] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li, "Lavea: Latency-aware video analytics on edge computing platform," in *Proc. ACM/IEEE Symposium on Edge Computing*, 2017.

[12] R. Deng, R. Lu, C. Lai, T. H. Luan, and H. Liang, "Optimal workload allocation in fog-cloud computing toward balanced delay and power consumption," *IEEE Internet of Things Journal*, vol. 3, May 2016.

[13] D. Puthal, M. S. Obaidat, P. Nanda, M. Prasad, S. P. Mohanty, and A. Y. Zomaya, "Secure and sustainable load balancing of edge data centers in fog computing," *IEEE Communications Magazine*, vol. 6, May 2018.

[14] R. Beraldi, A. Mtibaa, and H. Alnuweiri, "Cooperative load balancing scheme for edge computing resources," in *Proc. Conference on Fog and Mobile Edge Computing (FMEC)*, 2017.

[15] A. Kapsalis, P. Kasnesis, I. S. Venieris, D. I. Kaklamani, and C. Z. Patrikakis, "A cooperative fog approach for effective workload balancing," *IEEE Cloud Computing*, vol. 4, Apr. 2017.

[16] A. Lertsinsrubtavee, A. Ali, C. Molina-Jimenez, A. Sathiaseelan, and J. Crowcroft, "PiCasso: A lightweight edge computing platform," in *Proc. IEEE Conference on Cloud Networking (CloudNet)*, 2017.

[17] K. Ha and M. Satyanarayanan, "OpenStack++ for cloudlet deployment," Tech. Rep. CMU-CS-15-123, Carnegie Mellon University, Aug. 2015.

[18] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge, "Fogernetes: Deployment and management of fog computing applications," in *Proc. IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2018.

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

[20] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: A decentralized network coordinate system," in *Proc. ACM SIGCOMM Conference*, 2004.

[21] L. Subramanian, V. N. Padmanabhan, and R. H. Katz, "Geographic properties of internet routing," in *Proc. Usenix ATC conference*, 2002.

[22] A. Ahmed and G. Pierre, "Docker Container Deployment in Fog Computing Infrastructures," in *Proc. IEEE EDGE*, 2018.

[23] W. Hajji and F. P. Tso, "Understanding the performance of low power Raspberry Pi cloud for big data," *Electronics*, vol. 5, June 2016.