Proximity-Aware Replicas Management in Geo-Distributed Fog Computing Platforms

Ali Fahs

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par **« Ali Jawad FAHS »**

« Proximity-Aware Replicas Management in Geo-Distributed Fog Computing Platforms »

Thèse présentée et soutenue à « Rennes », le « 16 décembre 2020»
Unité de recherche : IRISA (UMR 6074)
Thèse N° :

Rapporteurs avant soutenance :

Ivona Brandic	Professeure, Vienna University of Technology
Etienne Rivière	Professeur, Université catholique de Louvain

Composition du Jury :

Président :		
Examinateurs :	Ivona Brandic	Professeure, Vienna University of Technology
	Etienne Rivière	Professeur, Université catholique de Louvain
	Erik Elmroth	Professeur, Umeå University
	David Bromberg	Professeur, Université de Rennes 1
	Shadi Ibrahim	Chargé de recherches, Inria
Dir. de thèse :	Guillaume Pierre	Professeur, Université de Rennes 1

“Achieve your dreams, but remember when you visit home taking out the garbage is one of your duties.”

Jawad Fahs, *My Father*

ACKNOWLEDGMENTS

First and foremost, I would like to thank my supervisor, **Guillaume Pierre**, for all his contributions to this work. Guillaume has helped me reach my maximum potential. His guidance and professional advice have driven this work to where it is now. Here I would like to highlight his work ethic, his attention to details, and his continuous technical and personal support. I will always be grateful for the time he invested in my development as a researcher, as an engineer, and as a student.

I would also like to express my appreciation to the members of the jury: **Ivona Brandic**, **Etienne Rivière**, **Erik Elmroth**, **David Bromberg**, and **Shadi Ibrahim**. Thank you for taking the time to read and evaluate my thesis, and for all the valuable feedback.

I owe a deep sense of gratitude to **Erik Elmroth** for welcoming me as a visiting researcher at the department of computing science, Umeå University. This fruitful experience has given me an insight into the research culture in another well-established research team. Erik's knowledge and care have impacted this work positively and for that, I am always grateful.

I would like also to thank the members of the CSID committee **Cédric Tedeschi** and **Shadi Ibrahim** for their guidance over the years of the Ph.D. Their advice were very helpful in steering this work in the right direction.

Thanks also go to the members of the Myriads team and the FogGuru project with whom I have great memories. During the time I spent in the team, I have made a lot of friends who helped me with technical and non-technical issues, and with whom I enjoyed our coffee breaks. Here I have to specifically thank **Millian Poquet**, **François Lemercier**, **Dorra Boughzala**, **Loic Guegan**, and **Mehdi Belkhiria** for their unmatched help and for our random discussions that I savor.

I would like to profusely thank my friends in France, Sweden, and Lebanon for staying by my side through the highs and the lows of my Ph.D. I want to specifically thank **Marina Kukurtçu**, **Rahaf Rahal**, **Ali Hamdan**, **Mohammad Ghousein**, **Ahmad Shokair**, **Ali Mokh**, **Salma Melliti**, **Houda Hammami**, **Raghed Bejjani**, and **Bahaa Mazloum**. Thank you for giving this experience the best flavor with all the memories we had together.

Finally and most importantly, I want to thank my family for their unconditional love and support. For my father **Jawad Fahs**, thank you for motivating me to be curious and ambitious, thank you for instilling the love of education and science in me and in my siblings, and thank you for providing me with all that I need to excel. For my mother **Nadia Fawzi**, my teacher, and the warmth that reached me even when I was thousands of miles away, I am a better version of myself because of your wisdom, patience, love, and care. For my siblings **Imad, Israa, Ghadeer, and Assil**, thank you for being there for me and for supporting your little brother since the first moment. I wish this achievement well make you smile and only by then I would consider that I have achieved my goals.

RÉSUMÉ

Le Cloud Computing offre aux fournisseurs d'applications des ressources de calcul abordables et évolutives, du stockage et des services réseau. Traditionnellement, une platforme de Cloud Computing consiste en un ensemble de quelques data centers hébergeant un grand nombre de serveurs puissants, ces derniers étant reliés par des réseaux dédiés. Cette architecture centralisée garantit une faible latence et une bande passante élevée entre chaque noeud installé dans un data center.

Cependant, dans une telle organisation, l'utilisateur final doit soumettre ses requêtes par le biais d'une connexion internet longue distance pour atteindre les ressources applicatives dans le cloud. Par conséquent, la requête de l'utilisateur peut être impactée par une latence réseau plus élevée et une bande passante limitée. De nombreuses applications peuvent tolérer une latence élevée et fonctionner correctement dans ces conditions. Cependant, de nouvelles applications réclamant des latences très faibles imposent que leurs requêtes soit traitées dans un temps très court. Ce type de requêtes doit nécéssairement être traité localement plutôt qu'à distance dans le cloud.

Les limitations du cloud computing ont favorisé la création de plates-formes de Fog Computing pour faire le lien entre le cloud et l'utilisateur final. Le Fog répond à ce problème par l'ajout de ressources localisées à proximité des utilisateurs finaux. Ces ressources sont limitées en capacité, mais elles permettent de fournir aux applications sensibles à la latence de courts chemins réseau entre les utilisateurs et les ressources fog. Dans le même temps, les data centers dans le cloud peuvent offrir une vaste réserve de ressources pour les applications réclamant de grandes puissances de calcul.

Localiser toutes les ressources du Fog au même endroit est souvent insuffisant pour garantir à tous les utilisateurs connectés un accès rapide aux ressources. Par conséquent, le Fog Computing est conçu de manière géo-distribuée où les noeuds de calcul sont éparpillés dans la zone de couverture. Il en va de même pour les instances applicatives, où l'accès à une instance d'application en bordure de réseau peut réduire la latence perçue par l'utilisateur. Ainsi, les applications situées dans une plate-forme Fog sont

organisées comme un ensemble de répliques fonctionnellement équivalentes et placées de façon à satisfaire au mieux les demandes des utilisateurs.

Sachant que le principal objectif du Fog computing est de garantir une faible latence d'accès, il est indispensable de pouvoir maîtriser la proximité entre les utilisateurs et les répliques des applications. Créer un moteur d'orchestration capable de gérer cette proximité requiert l'exploitation de données relatives à la proximité entre les noeuds. Cela impose également de concevoir des algorithmes à tous les niveaux de gestion des ressources où une décision doit être prise concernant la latence entre l'utilisateur et la ressource.

Nous pouvons résumer les niveaux de gestion de ressources en l'exploitation de proximité lors du routage de requêtes, le placement et replacement de répliques d'application, et le contrôle dynamique du nombre des répliques.

La première étape consiste à router les requêtes de l'utilisateur vers l'une des répliques de l'application. Le routage avec connaissance de proximité, dans ce cas, doit router les requêtes vers des répliques proches tout en maintenant un équilibre de charge parmi celles-ci.

Cependant, pour respecter les contraintes de latence définies par les applications, il faut que des répliques suffisamment proches existent. En conséquence, le placement des répliques doit être également basé sur l'emplacement des utilisateurs. Le caractère non-stationnaire des charges du Fog Computing nécessite en outre de résoudre le problème du placement de répliques de manière dynamique.

Lorsque l'application fait face à une charge de requêtes variable, les ressources de l'application peuvent également se retrouver en surcharge. Dans ce cas, l'application doit s'adapter pour offrir à tous les utilisateurs une qualité de service satisfaisante. Outre la nécessité d'un routage et d'un placement avec connaissance de proximité, cela nécessite une mise à l'échelle automatique avec également connaissance de la proximité. La mise à l'échelle automatique doit être capable de détecter les surcharges et les emplacements d'où proviennent cette charge de sorte que les nouvelles répliques soient placées à proximité des nouvelles sources de trafic.

Cette thèse propose un ensemble complet d'algorithmes qui constituent une plate-forme de Fog Computing avec tous les outils de gestion de la proximité nécessaires. Nos contributions visent chaque niveau de la gestion de ressources discutée jusqu'ici. En guise de preuve de concept, ces travaux ont été implémentés au dessus de Kubernetes, et ont été évalués sur une plate forme expérimentale réelle.

Première contribution : Proximity-Aware Routing

La première contribution appelée “Proxy-mity” est un système de routage avec prise en compte de proximité pour les plates-formes de Fog Computing. Il s’intègre de manière transparente à Kubernetes, et fournit un mécanisme de contrôle simple qui permet aux administrateurs système pour choisir le meilleur compromis entre la latence utilisateur et l’équilibre de charge entre les répliques. Les résultats des expérimentations montrent que Proxy-mity peut réduire la latence moyenne entre les utilisateurs et les répliques jusqu’à 90%, tout en permettant aux administrateurs système de contrôler le déséquilibre de la charge de leurs systèmes.

Deuxième contribution : Tail-Latency-Aware Fog Application Replica Placement

Router toutes les requêtes utilisateur vers une réplique proche de cet utilisateur est possible seulement dans le cas où une réplique proche existe effectivement. Cela nécessite de la plate-forme de Fog Computing qu’elle positionne les répliques sur des ressources disponibles en fonction de leur proximité des sources de trafic.

Pour traiter ce problème de placement de répliques, nous proposons “Hona,” un ordonnanceur tenant compte de la latence intégré au moteur d’orchestration de Kubernetes. Hona maintient une vue à grain fin des volumes de trafic générés à différents emplacements des utilisateur. Cet algorithme utilise une heuristique pour identifier les meilleurs placements des répliques. Il met également à jour ces placements dynamiquement en fonction des évolutions du trafic généré par les utilisateurs. Les résultats de nos évaluations montrent que Hona identifie efficacement un placement des instances, ce qui réduit de manière significative les derniers percentiles de la latence. Dans le même temps, cet algorithme conserve une complexité de calcul basse et maintient un équilibre de charge raisonnable entre les répliques.

Troisième Contribution : Tail-Latency-Aware Fog Application Replicas Autoscaler

Tout nombre fixe de répliques peut faire face à une saturation des ressources dans le cas d’une forte augmentation de la charge. Inversement, lorsque la charge baisse cela peut engendrer un gaspillage de resources inutilisées. Par conséquent, le nombre de répliques doit être contrôlé dynamiquement pour assurer des performances satisfaisantes à un prix raisonnable.

Nous proposons “Voila,” un algorithme de mise à l’échelle intégré au moteur d’orchestration de Kubernetes. Voilà utilise Proxi-mity pour router les requêtes et Hona pour surveiller le volume de trafic ainsi que leurs provenances. Il utilise une heuristique pour prendre les décision de placement des réplications et choisir le nombre de répliques. Nos évaluations basées sur un cluster de 22 noeuds ainsi que des données réelles de trafic montrent que Voilà assure que 98% des requêtes sont routées vers une réplique proche et non surchargée. Cette solution montre également de bons résultats en passant à l’échelle sur des sytèmes de taille plus importante.

ABSTRACT

Cloud computing offers application providers scalable and affordable computing, storage, and networking resources. A cloud computing platform typically consists of a small number of data centers that host a huge number of powerful servers connected using dedicated network links. This centralized architecture ensures a low-latency and high-bandwidth connectivity between the co-located nodes.

However, in this organization, end users must submit their requests over long-distance Internet links to reach the application resources hosted in the cloud. As a result, the end-users requests may suffer from high network latency and limited bandwidth. Many applications tolerate high latency and operate well within these conditions. On the other hand, a family of emerging latency-sensitive applications requires their requests to be returned within tight latency bounds. Such requests must be processed locally rather than in a remote cloud.

The drawbacks introduced by cloud computing have motivated the creation of fog computing platforms to bridge the gap between the cloud and its end users. The fog is an extension of cloud data centers with additional resources located in the end-users vicinity. These resources are limited in capacity, but they provide latency-sensitive applications low user-to-resource network latency. Meanwhile, the cloud data centers offer a vast resource pool that can serve compute-intensive applications.

Placing all the fog resources in a single location is often insufficient to provide all the connected users with nearby resources. Consequently, fog computing is designed as a geo-distributed platform where the computing nodes are scattered across the coverage area. The same can be said for the application instances, as providing a single application instance at the edge of the network may as well result in high user-perceived latency. As a result, the applications in a fog platform are typically organized as a set of functionally-identical replicas placed in well-chosen locations.

Since delivering low latency is one of the main objectives of fog computing, proximity-awareness is an essential feature that must be implemented in fog computing platforms. The creation of a proximity-aware fog orchestration engine requires one to exploit information about the inter-node proximity, and to design

algorithms in the different levels of resource management that make decisions based on user-to-resource latency.

We can summarize the proximity-aware resource management levels as request routing, application replicas placement and re-placement, and autoscaling of the size of the replica set.

The first step is routing the requests from the end users to one of the application replicas. Proximity-aware routing, in this case, should route requests toward nearby replicas while maintaining an acceptable load balance between them.

However, respecting the latency bounds defined by the applications requires the existence of replicas reachable by the users within those bounds. As a result, placing the replicas should also be done based on the location of the users. The non-stationary nature of fog computing workloads requires one to solve the placement problem in a dynamic fashion.

When the application is facing a surge in load, the application's resources may become overloaded. In this case, the application should scale to offer all the users an excellent quality of experience. This requires not only a proximity-aware request routing and replicas placement, but also a proximity-aware autoscaler. The autoscaler should be capable of detecting the surge in load and the location from which this load is emitted such that new replicas are placed next to the new sources of traffic.

This thesis proposes a complete set of algorithms that provide fog computing platforms with the necessary proximity-awareness. Our contributions target every level in the discussed resource management levels. The contributions are implemented on top of Kubernetes as a proof of concept and evaluated using a realistic testbed.

First contribution: Proximity-Aware Request Routing

The first contribution is Proxy-mity, a proximity-aware traffic routing system for distributed fog computing platforms. It seamlessly integrates with 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 across replicas. The evaluation shows that Proxy-mity can reduce the average user-to-replica latency by as much as 90% while allowing the system administrators to control the level of load imbalance in their system.

Second contribution: Tail-Latency-Aware Fog Application Replica Placement

Routing every user request to a nearby replica is possible only in case a nearby replica exists. This requires the fog platform to place replicas in available resources according to their proximity from the sources of traffic.

To address the replica placement problem, we propose Hona, a latency-aware scheduler integrated into the Kubernetes orchestration system. Hona maintains a fine-grained view of the volumes of traffic generated from different user locations. It then uses simple yet highly-effective heuristics to identify suitable replica placements and to dynamically update these placements upon any evolution of user-generated traffic. Our evaluations show that Hona efficiently identifies instance placements which significantly reduce the tail latency. At the same time, it keeps its own computation complexity low and maintains reasonable load balancing between the replicas.

Third contribution: Tail-Latency-Aware Fog Application Replicas Autoscaler

Any fixed number of replicas may reach over-saturation in the case of a load surge. Conversely, it may also waste valuable resources when the load drops. Therefore, the size of a replica set should be controlled dynamically to ensure optimal performance at a reasonable cost.

We propose Voilà, a tail-latency-aware autoscaler integrated into the Kubernetes orchestration system. Voilà uses Proxy-mity to route requests and Hona periodic checks to monitor the traffic volumes and their locations. It then implements heuristics to make decisions on the replicas placement and size of the replica set. The evaluations based on a 22-nodes cluster and a real traffic trace show 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.

TABLE OF CONTENTS

LIST OF FIGURES

List of Tables

List of Abbreviations

AI	Artificial Intelligence
AR	Augmented Reality
DNAT	Destination Network Address Translation
DNS	Domain Name System
F-RAN	Fog Radio Access Network
IoT	Internet of Things
ISP	Internet Service Providers
K8s	Kubernetes
MCC	Mobile Cloud Computing
MEC	Mobile Edge Computing
OSI	Open Systems Interconnection
QoE	Quality-of-experience
QoS	Quality of Service
RAN	Radio Access Network
SOA	Service-Oriented Architecture
VANET	Vehicular Ad hoc NETworks
VM	Virtual Machine
VR	Virtual Reality
WAN	Wide Area Network

Chapter 1

Introduction

Cloud computing provides on-demand computing, storage, and networking resources that liberate the application developers from maintaining traditional IT servers, and also guarantees them access to resilient, scalable, affordable, and secure resources. A cloud computing infrastructure typically includes a limited number of data centers. Each data center is composed of a large number of co-located computing nodes connected to each other using high-bandwidth/low-latency network links [1].

Cloud applications are designed to exploit the cloud nodes which are their first building block. Through virtualization, an application is hosted on one or more cloud nodes where compute, storage, and network resources are granted. The application is then exposed using an ingress that is configured to provide the application an externally-reachable address. This allows end users located outside the data center to communicate with the cloud application resources using the ingress address. As a result, end users are invited to transmit their load over long-distance Internet links to be processed in one of the cloud data centers.

This centralized paradigm however induces two drawbacks. First, the end-user requests often incur high network latency to reach a data center, estimated as 20-40 ms over wired networks, and up to $150\,ms$ over 4G mobile networks [2]. Such latency is acceptable for many applications, but a family of emerging latency-sensitive applications such as virtual reality and autonomous vehicles require the end-to-end communication and processing latency to be kept within 10-20 ms [3, 4]. In consequence, these applications' requests cannot be processed in the cloud and must instead get handled locally to avoid the high cost of network latency introduced by remote clouds.

A second limitation of centralized clouds stems from the rise of *Internet of Things* (*IoT*) technologies. IoT applications generate their input data at the edge of the network, and far from the data centers. The volume of data they produce is steadily increasing, and it is projected that by 2025, $75\,\%$ of all enterprise data will be generated

far from the data centers [5]. Transferring these data to the cloud before processing them would place a significant stress on the *Internet Service Providers* (*ISPs*) links. An obvious alternative consists of processing the data where they are produced, and of sending only the compute-intensive tasks towards the cloud.

These limitations of traditional cloud platforms have motivated the creation of fog computing to bridge the gap between the cloud and the end users. Fog computing extends the cloud with additional compute, network, and storage resources located at the edge of the network. When compared to cloud resources, the edge resources can range from moderate to small in terms of computing power, but they serve an essential role in reducing the user-to-resource network latency.

The difference between cloud and fog architectures is not limited to the size of the resource pool. It also extends to the geographical distribution of nodes and the network topology between them. A cloud computing infrastructure is organized as a small number of extremely powerful data centers. For instance, Google's infrastructure is composed of 20 data centers that are connected via dedicated links [6]. In contrast, fog computing aims at delivering ultra-low latency with its end users not only by placing nodes at the edge of the network, but also by distributing them across the coverage area (a campus, a city, or possibly even a region or a whole country). The fog nodes are connected to each other and to the Internet using a variety of networking protocols such as WiFi, 4G, and Ethernet. The fog resources are then reachable using access points where the end users connect and transmit load.

One of the fundamental advantages of fog computing is delivering low user-to-resource network latency. However, the distribution of the nodes places them next to the end users, but necessarily far away from each other. This combination of distributed nodes, unconventional network topology, limited resources, and dynamic workload creates a set of challenges that must be tackled for this concept to see the light of day.

Moving from cloud computing where all the physical nodes are functionally equivalent to fog computing where nodes are heterogeneous in terms of their proximity to the end users, requires one to implement mechanisms to detect this proximity and to manage resources accordingly. Proximity-awareness is one of the main features that differentiate the fog from other on-demand resource platforms, and creating proximity-aware mechanisms enables the fog to deliver on its promises.

Creating a proximity-aware fog platform is, therefore, a hot research topic. An application hosted on a fog platform that lacks proximity awareness may not profit from the availability of ultra-low latency resources and by consequence may result in a degraded *Quality-of-experience* (*QoE*). This can be caused by the absence of proximity awareness at different levels of the resource management that range from request routing, application instances placement, and to the size of the reserved application resource pool.

An end user or a group of end users transmitting load using the same access point are considered a single source of traffic. The network latency between the sources of traffic and the chosen application resource varies according to the network link established between the access point and the fog computing node that holds the resource. As a result, locating one application instance at the network edge does not necessarily provide a low-latency access to all of its end users. In consequence, fog applications are compelled to deploy multiple functionally-equivalent service replicas in nodes that are adjacent to the sources of traffic. In this case, the purpose of replication is not only fulfilling availability and processing capacity constraints, but also providing nearby replicas to all the sources of traffic dispersed across the coverage area.

This thesis specifically addresses the needs of latency-sensitive service-oriented applications. This type of application is organized as a set of replicated services responsible for processing/handling a single type of task. Every request issued toward one of the services should result in the same outcome regardless of the choice of replica to process it. The only difference in this case lies in the network latency between the access point where the end user is connected, and the fog node where the selected replica is hosted. Service-oriented applications are often elastic in order to facilitate horizontal scalability [7]. In this thesis, we consider data consistency between the application replicas as a responsibility handled by the application provider, and transparent to the fog computing platform itself. We also make the hypothesis that application replication process requests without issuing queries to other services.

Choosing the best set of fog nodes where an application should deploy its replicas requires one to chose a set of replica placements that minimize the network latencies between end-user devices and their closest application replica. To deliver outstanding QoE to the end users it is important that every single request gets processed within

tight latency bounds. We, therefore, follow best practice from commercial content delivery networks [8] and aim to minimize the *tail request latency* rather than its mean, for example, defined as the fraction of requests incurring a latency greater than some threshold.

Non-stationary load affects applications hosted on various platforms [9]. However, in geo-distributed fog platforms, this non-stationarity incurs both along the time and the space dimensions. Unlike online applications where the workload is an aggregation of all the incoming requests from an unbounded user population, fog applications have to handle requests according to their origin and their proximity to the available replicas. The geo-distributed nature of the fog combined with end-users mobility creates load variations not only as a function of time but also as a function of the access point locations where end users are connected. As a result, the replica set size must depend on the workload volume and the locations from which this workload is transmitted.

Fog computing, like cloud computing, enables a clean separation of concerns. Developers focus on implementing and improving their applications, while the infrastructure automates the deployment and management of these applications. Establishing this separation was made possible mainly by virtualization and orchestration. In virtualization, the application is encapsulated with all its dependencies in a virtual instance that facilitates the deployment of the application regardless of the target environment. For fog platforms, containers and specifically Docker containers [10] are widely used as the *de facto* virtualization technology since they offer a lightweight implementation, quick boot-up, and they can run on very limited machines. Kubernetes [11], Docker Swarm [12], Apache Mesos [13] are container orchestration engines that automate the deployment and management of the containers. *Kubernetes* (*K8s*) is considered as the most used container orchestration engines for fog computing [14]. Portability, flexibility, and lightness are among the K8s features that make it a potential fog orchestration engine. Yet, Kubernetes was designed with cluster and cloud environments in mind. It, therefore, lacks some key features to make it fully suitable in fog environments. For these reasons, we chose to extend K8s with the fog-specific features it still misses. However we can argue that our algorithms may be easily adapted to integrate into other container orchestration engines.

A fog computing infrastructure requires information about the inter-nodes proximity and their proximity to the end users. We identify three levels of resource management where this proximity must be exploited to deliver low user-to-resource latency:

- **Request routing:** the network latency between an access point and the application replicas varies according to the network link between the access point and the node that hosts the replica. Proximity awareness is then essential to distinguish nearby replicas from far-away ones. Using proximity estimations as input, the orchestration engine must route each request to a nearby replica rather than one of the far-away ones, in a way that does not distort the load balance between the replicas.
- **Replica placement/re-placement:** proximity-aware routing can ensure low latency only if nearby replicas exist. An access point which is receiving load without having any nearby replicas where it can route traffic signals a proximity-*unaware* replica placement. The main challenge in proximity-aware replica placement lies in detecting the sources of traffic and finding a placement that offers each one of them at least one nearby replica. Since the number of possible replica placements is very large, a heuristic should be implemented to effectively find a satisfactory solution. A placement is considered a solution if it can provide a low tail user-to-replica latency. Moreover, any static placement may provide the promised *Quality of Service* (*QoS*) for only a short period of time before the workload characteristics change. As a result, the placement should be updated dynamically to cope with the workload variations.
- **Autoscaling:** The number of replicas which should be deployed cannot remain constant for extended periods of time. First, a surge in traffic may overload the replicas and may lead to performance degradation. Second, the day/night patterns in workload distribution show a drop in traffic during nighttime which may lead to a waste of valuable resources. This is suboptimal in any multi-tenancy platform, and especially in fog platforms where the edge resources can be very limited. As a result, the number of replicas and their placement should change following the workload volume and the locations from which this non-stationary load is transmitted. Autoscaling ensures that none of the replicas are overloaded and that the placement and scale are capable of delivering a low tail latency.

In this thesis, we propose three contributions to transform a cloud orchestration into a proximity-aware one, qualified to handle the emerging requirements imposed by the fog applications. The contributions introduce proximity awareness at every level of the discussed resource management levels.

1.1 Contributions

1. **Proxy-mity: Proximity-Aware Traffic Routing**
   A geo-distributed system such as a fog computing platform must necessarily choose a suitable trade-off between resource proximity and load-balancing. Systems like Mesos [13], Docker Swarm [12] and Kubernetes [11] implement location-*un*aware traffic redirection policies which deliver excellent load-balancing between application replicas, but very suboptimal user-to-resource network latencies. At the same time, any system which would route every request to the closest replica would face severe load imbalance between replicas if some users create more load than the others [15].

   In this contribution, we propose Proxy-mity, a proximity-aware request routing plugin for Kubernetes. Proxy-mity exposes a single easy-to-understand configuration parameter α which enables system administrators to express their desired trade-off between load-balancing and proximity. It integrates seamlessly within Kubernetes and introduces very low overhead.

   In our evaluations, Proxy-mity reduces 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.
2. **Hona: Tail-Latency-Aware Fog Application Replica Placement**
   Proxy-mity can ensure low user-to-resource latency only if nearby replicas are available. Therefore, proximity awareness should also be considered when placing application replicas in the vicinity of the end users. However, choosing which specific resources from a large-scale fog computing infrastructure should be allocated to place the application instances remains a difficult problem.

   We propose Hona, a tail-latency-aware application replica scheduler which integrates within the Kubernetes container orchestration system. Hona makes

use of Kubernetes to monitor the system resource availability [11], Vivaldi coordinates to estimate the network latency between nodes [16] and Proxy-mity to monitor the traffic sources and to route end-user traffic to nearby instances. It uses a variety of heuristics to efficiently explore the space of possible instance placement decisions and select a suitable one upon the initial replica placement. Finally, it constantly monitors the performance of the current placement and automatically takes corrective re-placement actions when the characteristics of the end-user traffic change. The Hona re-placement algorithm is able to maintain the tail latency under a certain threshold while inducing minimal changes to the placement.

Our evaluations based on a 22-node testbed show that the multiobjective heuristics used by Hona identify placements with a tail latency very close to the theoretic optimal placement, but in a fraction of the computation time, while preserving an acceptable load distribution between application instances. The re-placement algorithm efficiently maintains a very low tail latency despite drastic changes in the request workload or the execution environment. We further demonstrate the scalability of our algorithms with simulations up to 500 nodes.

3. **Voilà: Tail-Latency-Aware Fog Application Replicas Autoscaler**
   Hona dynamically updates the placement of a fixed-size replica set to achieve low tail latency. Yet, a surge in traffic may overload the replicas and lead to a degraded QoE; similarly, a drop in traffic may lead to a waste of valuable resources. Therefore, 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 [17], which motivates the need for using an auto-scaler to dynamically adjust the number and locations of a fog application's replicas.

   We propose Voilà, a tail-latency-aware fog application replica autoscaler for Kubernetes. 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.

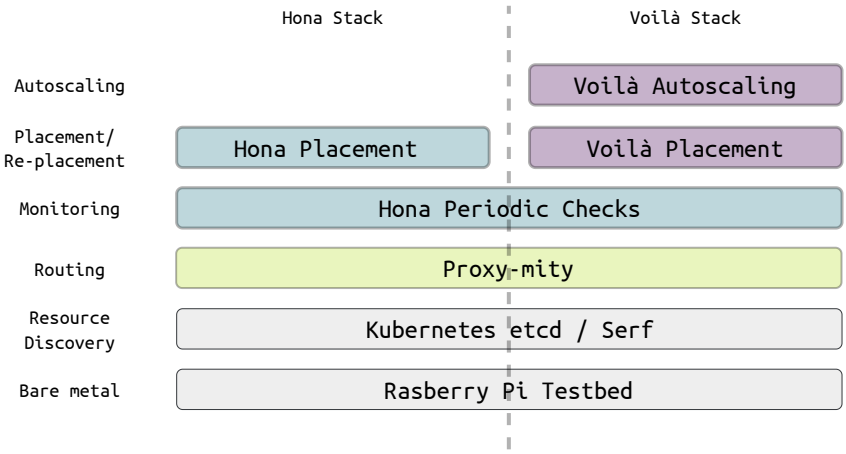


Figure 1.1 – Our contributions in the abstract layers of resource management.

Our evaluations based on a 22-node cluster and a real traffic trace show 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.

Figure 1.1 shows the resource management layers used in the second and third contributions. We refer to the monitoring system as Hona periodic checks. This system extracts the cluster and workload characteristics every period of time, then makes them accessible to the upper layers. Both Hona and Voilà use Proxy-mity for routing requests and Hona periodic checks to collect cluster and traffic info. However, Hona makes decisions on placements of a fixed-size replica set while Voilà has the capabilities to scale the replica set according to the state of the application workload. Note that Voilà makes use of Hona periodic checks but replaces Hona's replica placement with its own algorithms.

The aggregation of Proxy-mity, Hona periodic checks, and Voilà constitute a coherent set of plugins that work on top of Kubernetes. These plugins introduce proximity awareness at different levels of resource management and can arguably turn Kubernetes into a mature proximity-aware fog platform. We used Kubernetes-based implementations as a proof of concept. But, in principle, the same algorithms may also be implemented in other orchestration engines.

1.2 Published papers

The following papers have been published as part of this thesis:

1. *"Proximity-Aware Traffic Routing in Distributed Fog Computing Platforms"*, **Ali J. Fahs** and Guillaume Pierre, 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2019), Larnaca, Cyprus. [18]
2. *"Tail-Latency-Aware Fog Application Replica Placement"*, **Ali J. Fahs** and Guillaume Pierre, 18th International Conference on Service Oriented Computing (ICSOC 2020), Dubai, UAE. [19]
3. *"Voilà: Tail-Latency-Aware Fog Application Replicas Autoscaler"*, **Ali J. Fahs**, Guillaume Pierre, and Erik Elmroth, 28th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2020), Nice, France. [20]

1.3 Organization of the thesis

This thesis is organized in 6 chapters:

Chapter 2 presents a theoretical and practical overview of fog computing platforms and the systems used to achieve the thesis' objectives. In this chapter we discuss cloud computing and the emergence of fog computing, its architecture, challenges, and use cases. Then we present a practical overview of the Kubernetes architecture and highlight the need of network proximity for addressing application requirements. Finally, we describe the complete fog computing architecture which constitutes the foundation of this thesis.

Chapter 3 describes the current state of the art according to the different levels of resource management: workload routing, placement and re-placement, and finally autoscaling. At the end of this chapter we position our contributions with respect the current state of the art.

In Chapter 4, 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. The evaluation shows that Proxy-mity can reduce the 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.

In Chapter 5, we propose Hona, a latency-aware scheduler integrated in the Kubernetes orchestration system. Hona maintains a fine-grained view about the volumes of traffic generated from different user locations. It then uses simple yet highly-effective heuristics to identify suitable replica placements, and to dynamically update these placements upon any evolution of user-generated traffic. Our evaluations show that Hona efficiently identifies instance placements which reduce the tail request latency. At the same time, it keeps computation complexity low and maintains reasonable load balancing between the replicas.

In Chapter 6, we propose Voilà, a tail-latency-aware auto-scaler integrated in the Kubernetes orchestration system. Voilà uses Proxy-mity to route requests and Hona periodic checks to monitor the traffic volumes and their locations. It then implements heuristics to make decisions on the replicas placement and size of the replica set. The 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.

Finally, Chapter 7 presents the conclusions of the thesis and discusses directions for future work.

Chapter 2

Background

In this chapter, we present a theoretical and practical overview of fog platforms and the systems used to achieve the thesis objectives. First, we discuss cloud computing and how it did not fully satisfy new application requirements. We then detail fog computing, its architecture, challenges, and use cases. Third, we present a practical overview of Kubernetes architecture, and its different abstraction layers. The fourth section highlights the need of network proximity for addressing application requirements. Finally, we present a complete fog computing architecture which constitutes the foundation of this thesis.

2.1 Cloud computing

Cloud computing drove a shift in the IT industry from traditional in-house servers toward cloud-based services. In 2019, Flexera surveyed 786 technical professionals across a broad cross-section of organizations and found that 94% of respondents use the cloud one way or another [21]. This can be attributed to a lower cost to obtain resources, avoiding the maintenance efforts of an in-house server, the scalable approach clouds present such that an application provider can easily increase or decrease the utilized resources, and finally minimizing the aftermath of server failures as the application can be easily spawned over a different server in the same cloud provider.

2.1.1 Cloud architecture

Cloud servers are typically powerful nodes co-located in huge data centers that are based in different locations across the globe. Figure 2.1 shows a map of the 20 data centers of Google [6]. Each of these data centers can hold hundreds of exabytes in

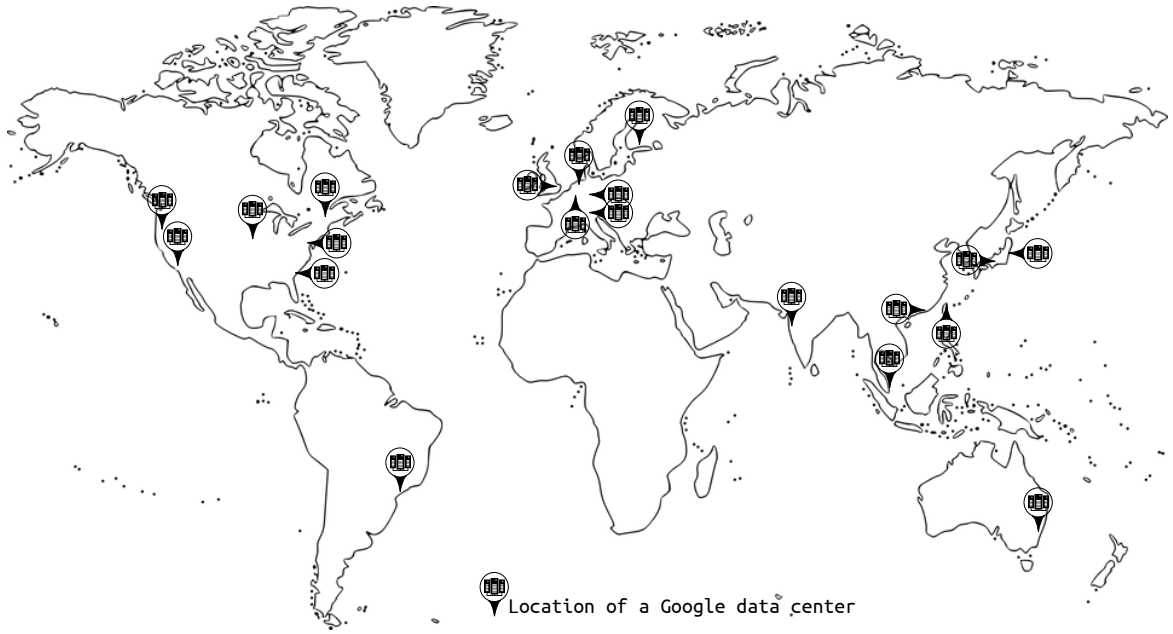


Figure 2.1 – A map of Google cloud data centers.

disk space. The total number of servers in all the data centers owned by Google was estimated in 2016 as 2.5 million servers [22].

The location and the size of the data centers follows an economical model that decreases the cost of operation by concentrating a big number of nodes, and by targeting locations with lower electricity price [23]. The nodes inside the data centers are connected via low-latency/high-bandwidth links that handle the ever-rising internal traffic. Nowadays fiber optics technology is making its way to data centers to meet this demand [24]. Thus, the inter-node latency within data centers can be considered negligible.

However, this centralized architecture (illustrated in Figure 2.2) constrains end users to communicate with the application services over long-distance Internet links. For a wide range of applications, communicating over the backhaul network does not affect the user-perceived QoE. For example, storage and web hosting applications can offer decent performance when deployed on a cloud platform since they can tolerate relatively long network latencies. In contrast, a new emerging class of latency-sensitive applications require the response time to be lower than a strict threshold, and will perform poorly under such conditions.

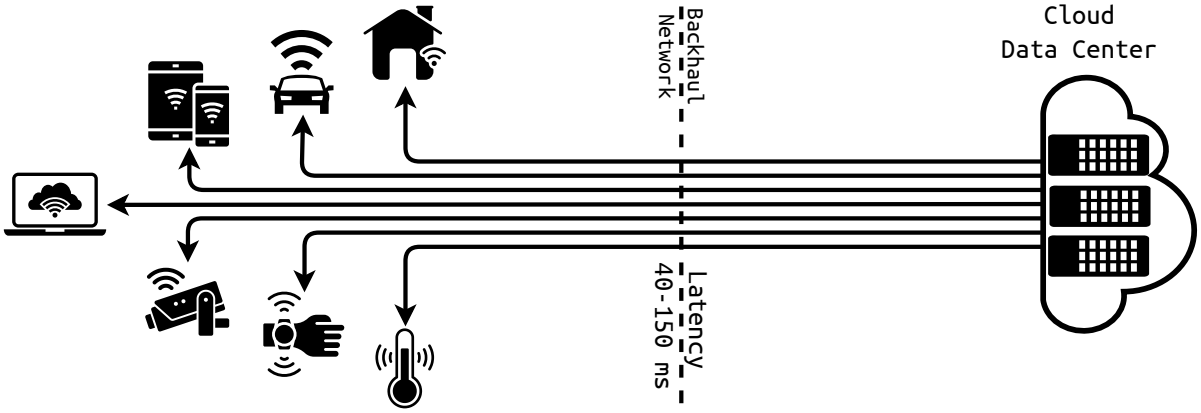


Figure 2.2 – Cloud computing architecture.

2.1.2 Cloud limitations and the emergence of fog computing

The upsurge in the fields of *Artificial Intelligence* (*AI*), autonomous vehicles, and most prominently IoT have changed the nature of end users. Rather than having people behind devices transmitting requests to the cloud, it is estimated that by 2025 55% of all the data will be generated by 21.5 billion IoT devices [25] (approximately 63% of all the connected devices [26]). The shift of end users' nature was combined with an emergence of new set of IoT applications and other applications like stream processing and *Virtual Reality* (*VR*). Such applications are demanding in terms of latency and bandwidth.

This growth in the number of connected devices and the introduction of a new requirements have created new challenges for the cloud architecture:

- **Network round trip time:** the round-trip time needed for an end user to access a cloud application is estimated as $40\,ms$ for a wired connection and as high as $150\,ms$ for a 4G connection [2]. Meanwhile, applications like VR and gaming can only tolerate end-to-end response times (including network and computatuion delays) of $20\,ms$ maximum [3]. Such low latency cannot be supported even by 5G where the user-to-Internet latency is reported as $30 - 40\,ms$ [27, 28].
- **Bandwidth:** The majority of IoT devices use very little bandwidth, however the massive number of IoT connected devices can easily congest network links, specifically those of the *Wide Area Network* (*WAN*) [29]. Since the number of devices is ever increasing, and the technologies that use them is advancing it is evident that the WAN bandwidth should increase to accommodate them. Yet,

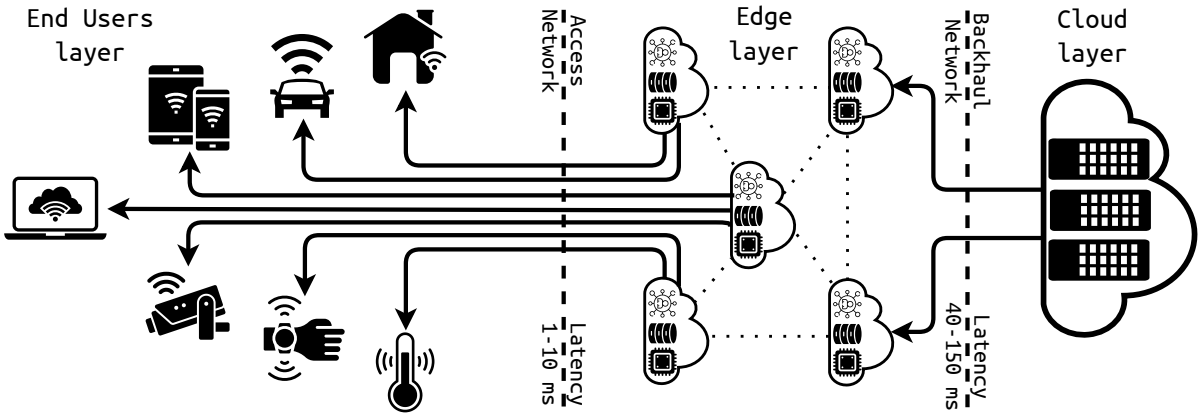


Figure 2.3 – Fog computing architecture.

the improvements in the field of network technology are slow compared to the growth of Internet traffic produced by the IoT devices.

We can conclude that the drawbacks of cloud platforms are derived from the network topology, where all the requests have to traverse long and possibly congested routes to reach the services. In order to address the issues of network latency and bandwidth, there is a need for a new computing paradigm that provides resources in the vicinity of the end users, while preserving the advantage of unlimited cloud resources. Fog computing is not meant to replace the cloud but rather to extend it with additional resources located closer to the end-user devices [23].

2.2 Fog computing

Fog computing aims to bridge the gap between the cloud and the connected devices, providing the end users with resources accessible within their near-range communication links (see Figure 2.3). This approach guarantees a low user-to-resource latency derived from the fact that the application can be placed in a near-range resource. In addition, the bandwidth in the fog layer is managed by the fog platform and does not suffer an ISP bandwidth limitation.

However, the precise definition and standardization of fog computing technologies remains a hot research topic. In the research community slightly different definitions have been proposed [30–34]. These definitions share common characteristics and

diverge mainly on the definition of the fog nodes. We can say that most of the researchers in fog computing will agree on the following statements:

- Fog computing provides compute, network, and storage resources.
- A part of the resources is located at the edge, within network low latency from the end users.
- The fog nodes are not co-located in a single place but rather geographically distributed over the coverage area.
- A fog computing platform is composed of resources located at the edge as well as resources located in traditional cloud data centers.

In contrast, different researchers have proposed different types of fog nodes [34]. The most common are:

- **Cellular base stations:** commonly referred to as *Fog Radio Access Network* (*F-RAN*) nodes. This architecture takes advantage of the overlapping between the fog and the *Radio Access Network* (*RAN*) at the extreme edge. The fog nodes in this case are relatively powerful and well connected. Yet, such approach has a dependency on the telecommunication companies which own the cellular base stations [35, 36].
- **Network elements:** specifically WiFi access points and routers. Similar to the cellular base stations, the access points are distributed at the extreme edge. Adding some resources to these network elements would make them very convenient fog nodes but with less resources due to space limitations [37–39].
- **Intermediate compute nodes:** such as small to medium size servers distributed on the edge without dependencies on other devices. This approach provides a tunable resource pool that is relatively capable, but we argue the ease of placing and connecting such nodes in safe places over the geographical area [29, 40].
- **Points of presence (PoPs):** some researchers found the convenience of using very small devices like single board computers, since they are cheap, do not require a lot of energy, and can be placed practically anywhere. A platform using such nodes would require large number of them to provide a decent resource pool [41–43].

In our work, we have chosen to base our efforts on PoPs since they are accessible, and are commonly used to prototype a distributed fog system. At the same time, an

implementation running on top of such nodes proves the lightness and the feasibility of the system. In other words, if this implementation can run on the limited resources of the PoPs, then in principle it should also work on more capable nodes.

2.2.1 Fog applications

The number of applications that intend to exploit fog computing platforms is growing. Fog applications are driven by innovative design that requires additional platform characteristics which can only be delivered if the application instances were deployed close to the source of traffic [4]. An application implemented in the edge layer will not only benefit from low latency and bandwidth optimization but also will take advantage of content caching at a finer granularity. Fog computing offers cost cutting when implementing a private fog is cheaper than a pay-as-you-go cloud model. IoT devices can reduce their energy consumption since they have to wait less to receive their responses. A set of promising use cases for fog computing can then be categorized by the following categories:

- **IoT applications:** fog computing was conceptualized with IoT applications in mind. It can deliver all the requirements of IoT including mobility support, geo-distribution, and location awareness. As a result, many IoT applications intend to run on fog platforms [44–50].
- **Entertainment applications:** this category includes gaming and virtual/augmented reality applications where a high latency can severely affect the QoE. Many fog applications require low latencies, although the definition of "low" varies a lot. Some applications require a low latency in terms of seconds (*e.g.* IoT), while other applications require latencies in terms of milliseconds. We categorize the applications that requires latency in terms of milliseconds as *latency-sensitive applications*. The interesting factor of such applications that they need bounds on their response time where they define a threshold latency. If a request response time exceed this threshold the QoE gets impacted [47, 51–53].
- **Stream processing applications:** a stream processing application is a collection of operators, all of which process streams of data. Applications like video broadcasting and surveillance are stream processing applications and they have a dependency on the network bandwidth. The bandwidth optimizations offered by the fog presents an attraction for such applications [54–58].

- **Others:** such as autonomous vehicle [59–61], data storage [62, 63], wearable technology [51], and smart grids [64].

In the design of fog computing platforms, nodes are close to the end users but necessarily far from each other. Unlike cloud where the nodes are connected using high-bandwidth/low-latency network links, fog computing nodes are typically far away from each other. An application deployed in the fog layer must therefore provide several replicas to keep a low latency between each end user and at least one of the application replicas.

This was evident in the survey [4] that shows most of the surveyed applications are deployed using software distribution to provide a nearby instance for all the connected devices. This is enabled using software replication where several application replicas are spawned in different nodes in the fog layer. Other trends were also detected among the surveyed applications for IoT and stream processing applications that are based on widely-distributed user-base. Such applications require a congruent software distribution of the fog nodes and application replicas.

Many fog applications are organized following a *Service-Oriented Architecture* (*SOA*), which involves the deployment of services as units of logic that runs in a network. As a result each service is responsible for processing a single task, and the user device contacts these services to get the desired response. The attractive aspect of this architecture lies in the independence of the service's replicas, which separates the concerns between the service's internal design and the client making use of this service.

This thesis specifically focuses on implementing a proximity-aware fog computing orchestration engine for replicated service-oriented applications. We target latency-sensitive applications which define a certain latency threshold and run on top of request/reply services.

2.2.2 Fog architecture

As previously discussed, fog computing extends the cloud with additional resources located in the immediate vicinity of the end users. The architecture of a fog platform (shown in Figure 2.3) therefore adds an intermediary layer between the cloud nodes and the end users. This layer is often referred to as the edge layer. The aggregation of edge and cloud layers allows the system to process latency-sensitive end-users requests

on a nearby node. At the same time this paradigm does not prohibit developer from utilizing the unlimited cloud resources for non-sensitive tasks.

We define the fog as an aggregation of three main layers:

1. **End-user layer:** This layer consists of the end-user devices such as IoT sensors and actuators, mobile phones, wearable devices, and autonomous vehicles. These devices are the origin of the requests that will later be processed in the fog. The end-user devices are connected to the fog layer through network links using a variety of mechanisms such as *Domain Name System* (*DNS*) redirection and software-defined networking (we discuss the organization of this access network in Section 2.5.1).
2. **Edge layer:** This layer is composed of the fog nodes. It is connected to the end users using access network and to the cloud using backhaul network. It is composed of limited resources that are distinguished by their low latency from the end users. As previously mentioned, we consider the fog nodes as points of presence that can be as small as single-board computers. The PoPs are distributed across a potentially large coverage area like a city center or a university campus. Each end-user device connects to one of the PoPs in order to reach the desired application. In other words the PoPs are the gateways where the end users connect and submit their requests. The PoPs are connected to each other using a "backhaul" network which also gives them an Internet access for relaying requests toward the cloud.
3. **Cloud layer:** This layer consists of the traditional cloud where one or more powerful data centers are connected to each other and to the rest of the Internet. The cloud layer accommodates extensive compute and storage resources that are capable of supporting compute-intensive applications.

The differences between the edge layer and the cloud layer can be summarized by:

- **User-to-resource latency:** The fog nodes communicate with the end users devices using access networks such as LAN, WiFi, and the last mile of the LTE networks. This maintains a low user-to-fog latency compared to a high user-to-cloud latency where the requests traverse long-distance Internet links.
- **Architecture:** The main difference here is the distribution of the nodes in the edge layer, compared to co-located nodes in the cloud data centers.

- **Resource capacity:** The cloud resources can be considered unlimited, compared to moderate compute and storage capacities provided by the edge.

However these differences do not deprive the application providers from taking advantages of both layers. The fog layer targets the latency-sensitive tasks and offers them nearby compute resources, while the cloud layer may handle storage and compute-intensive tasks.

2.2.3 Challenges of fog computing

Fog computing remains a new concept and no current platform can claim to solve the wide range of issues it creates. Reaching the point where fog computing attains the desired results requires one to address a number of challenges:

- **Orchestration engine:** to implement a fog computing orchestration engine a number of features have to be supported. Some of these features are already present in cloud computing, the likes of availability, scalability and fault tolerance. However the distribution of fog nodes and the limitation in resources make supporting these features more difficult. On the other hand, a healthy operation of the fog, and more specifically the edge layer requires features that support node heterogeneity, light deployment, and proximity-awareness. We discuss this challenge further in Section 2.3.
- **Proximity-awareness:** a key feature for a fully-functional fog is the ability to estimate and integrate the inter-node latencies in the platform design. A set of technical issues have to be addressed to first implement such measurement without inducing a network overhead, and to utilize the measurements in different abstraction layers in the resource management. we discuss this challenge further in Section 2.4.
- **Privacy and security concerns:** privacy and security should be addressed in every layer of the fog. The concerns are often focused on the gateways and the access control, and their vulnerability to attacks like man-in-the-middle and IP spoofing. Other security and privacy concerns in authentication, and data protection have been identified [65, 66]. Preserving a secure platform for the users' data and privacy remains a priority and any platform that will be available commercially must abide by high security standards.

- **Edge layer scalability:** the edge layer is widely distributed in a way that provides every end user with a nearby node. This architecture brings many advantages but it can be a challenging in terms of scale. In the scenarios of a fog platform like that of a city center, the nodes will be scattered allover the city. Providing all the nodes with the means of communication and power can be very challenging, and designing a scalable platform to control them is still not solved.

This thesis focuses on implementing proximity-awareness in fog computing orchestration engine. In other words, we focus on the first and second challenges. Proximity awareness is a key feature for latency-sensitive applications. As a result, implementing the tools to exploit user-to-resource latency is a requisite for designing a fully functional fog orchestration engine.

2.3 Kubernetes

Orchestration is the process of managing different applications which coexist in the same platform. The management of these applications ensures their QoS requirements by providing the proper resources, whether it is a computing resource measured by the number of cores and memory usage, a storage resource measured by the disk space, or a network resource measured by the assigned bandwidth. The duties of an orchestrator include the treatment of the workloads and creating the routes for these workloads, health checks on the applications state, and handling application provider requests to change the state of the deployment. Orchestration is also referred to as automation since all the orchestrator's tasks are done in an automated fashion.

A number of mature orchestration engines are available for cloud computing, including Kubernetes [11], Docker Swarm [12], Mesos [13], and OpenStack [67]. These engines have stood the test of time and have proven that they are efficient and reliable in the context of a cluster or a data center. In contrast, fog computing is still in its early stages, and it does not yet have a dedicated orchestration engine that can support its specific requirements. In this section we will discuss the motivation behind choosing Kubernetes as the basis of our efforts, then we continue by explaining how Kubernetes operates.

2.3.1 Why Kubernetes?

In the state of the art there exist two main directions to design future fog computing orchestration engines. The first is starting from scratch and creating an engine that is specific for fog computing [68]. In the second direction, researchers are remodeling one of the cloud orchestration engines to support fog computing [69–71]. Since our work is targeting proximity-awareness and the set of mechanisms to support it rather than the engineering task of creating a new orchestrator, we took the second direction where we implemented proximity-awareness on top of Kubernetes as a proof-of-concept. We may argue that a future dedicated fog orchestrator will eventually outperform a remodeled one, however the journey for such orchestrator to reach production is a long one.

Kubernetes is an open-source container orchestration platform which automates the deployment, scaling and management of containerized applications on large-scale computing infrastructures [11]. It is one of the most popular container orchestration engines [72]. The Cloud Native Computing Foundation has found in a survey of respondents from different business sectors that $78\,\%$ of the respondents are using Kubernetes in production [73]. Moreover, Kubernetes is getting recognized as the new standard of computing. Many industries were able to use Kubernetes at the edge, proving its flexibility [74]. This drove new efforts for creating extended versions of Kubernetes specifically targeting the needs of edge/fog computing [75, 76].

A Kubernetes cluster consists of a master node which is responsible for scheduling, deploying and monitoring the applications, and a number of worker nodes which actually run the application's replicas and constitute the system's computing, network and storage resources.

We can differentiate Kubernetes from its contenders by its flexible implementation. Kubernetes is designed as a set of plug-ins which work together to maintain the system state using feedback control loops. As well, Kubernetes as a software does not provide application-level services, such as middlewares, databases, caches, nor cluster storage systems. This approach makes Kubernetes very lightweight. The developer can then implement the application-level services as plug-ins. This makes Kubernetes very attractive for fog computing, since application providers may select what tools they want to be included, rather than having a monolithic platform that can do everything.

Another factor that motivated the use of Kubernetes for our case scenario was the ease of modifying the system just by adding plug-ins, which makes it easier for the

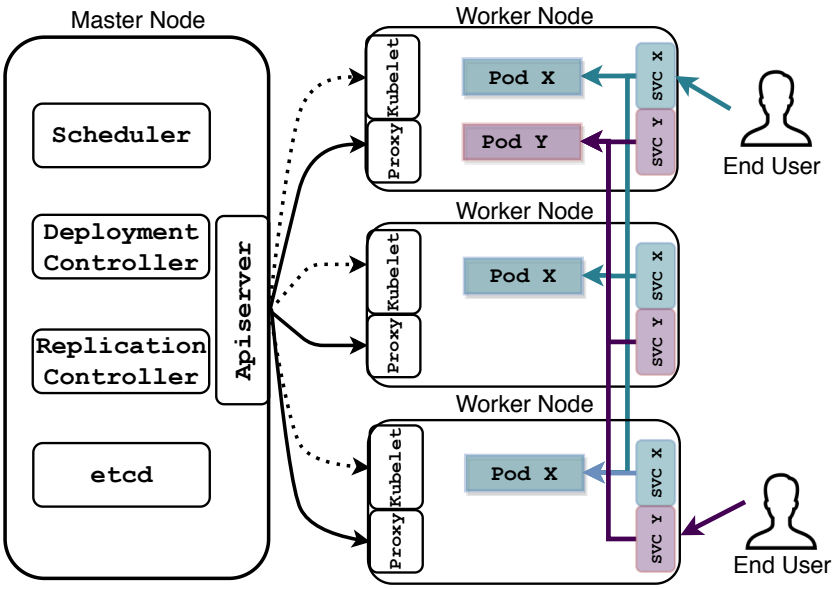


Figure 2.4 – Organization of a Kubernetes service. "Service X" forwards requests towards three pods located in three different nodes, whereas "Service Y" serves only one pod.

developer to update the system during runtime, rather than getting a new source code with the modification and then compiling and implementing everything. Several other researchers made the same choice of basing experimental fog platforms on Kubernetes [69–71, 77].

2.3.2 Application model

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 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. Maintaining data consistency between

pods is under the responsibility of the application developers. Kubernetes does not provide any support for implementing this data consistency.

Pods can be created and destroyed dynamically. This leads to a problem where an end user who wants to communicate with the pods must keep track of the IP addresses of the current running pods. In Kubernetes, a *Service* is an abstraction which defines a logical set of Pods and a policy by which incoming requests are routed to them. As illustrated in Figure 2.4, 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 a worker node receives a request for an application's service IP address, the request is further routed internally to the Kubernetes cluster using kernel-level *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. Unlike the most common load balancers for cluster computing which run in layer 7 of the OSI model, the Kubernetes services route packets using iptables in layer 3.

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 IP routing tables (see Figure 2.4). 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 2.5, 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

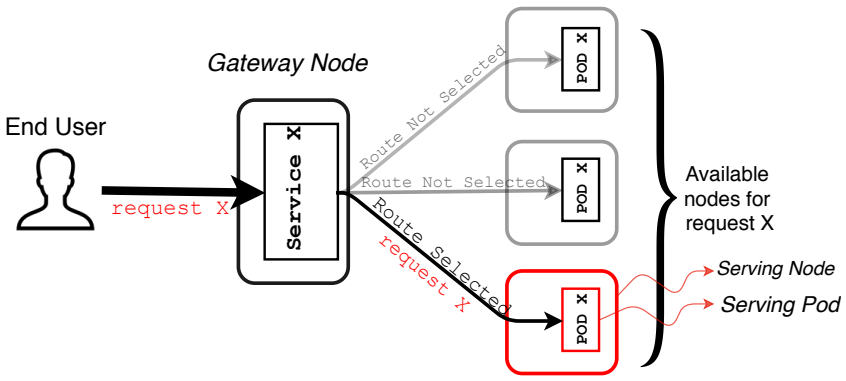


Figure 2.5 – Gateway node and serving nodes.

computing platform. one 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.

2.3.3 Pod scheduling

When a new pod or group of pods is created, the Kubernetes scheduler is in charge of deciding which worker nodes will be in charge of executing them. Figure 2.6 illustrates the two phases of the scheduling process. The scheduler first builds a list of nodes which are capable of executing the new pod (e.g., because they have sufficient available resources to accommodate the new pod). Second, it ranks the "feasible" schedules according to some policy, chooses the schedule with the greatest score, and stores this decision in an object store. In every worker node, a *Kubelet* daemon periodically checks this object store and deploys any pod assigned to it which was not created yet.

2.3.4 Resource discovery

Kubernetes uses etcd to store the state of the cluster which is defined as a distributed, reliable key-value store for the most critical data of a distributed system [78]. It presents a reliable way to store data that needs to be accessed in a distributed system. Any application can use etcd to save and update its data as

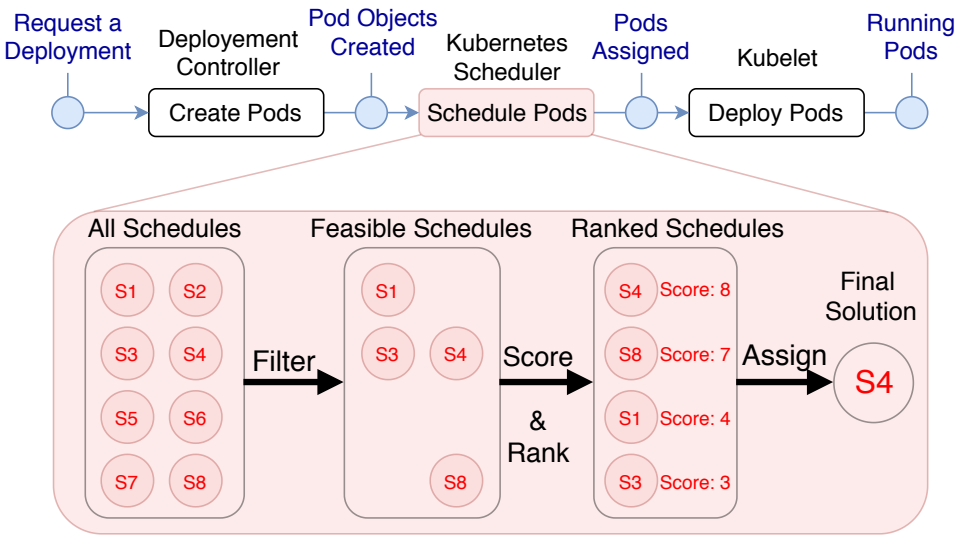


Figure 2.6 – Kubernetes' scheduling process.

key-value pairs. The key-value store preserves the previous versions of a key-value pair when its value is superseded with new data.

In Kubernetes, etcd is deployed as a pod or a set of pods. The state of the cluster includes the current system configuration, the status of the nodes and pods, and the specification of the deployments. As a result, etcd is the main tool Kubernetes uses for resource discovery.

Kubernetes was designed to manage computing resources in cluster-based or cloud-based environments. As a consequence, it considers all worker nodes as functionally equivalent since it have no notion of node proximity to the end users. For Kubernetes to become suitable for fog computing scenarios, we aim in this work to modify its application and scheduling models to make them fog-aware.

2.4 Network Proximity

In the edge layer of fog computing platforms, nodes are located close to the end users but necessarily far from each other. This distribution requires one to accurately estimate the network latency between the nodes. Such measurement is essential for healthy operation of the fog platform.

Figure 2.7 depicts an application deployed in the edge layer. Each circle represents a fog node located over the map of France, which represents the coverage area. For the

sake of simplicity, we consider in this example that the inter-node network latency is proportional to the geographical distance in the map. A subset of the nodes hold a replica of the application, these nodes are identified by the blue color. A user connected to the edge layer gets connected to one of the nodes using the access network. Requests submitted by the user are then further routed toward an application replica by the Kubernetes services.

As discussed in Section 2.2.1, latency-sensitive applications must process their requests within a certain deadline otherwise the users will suffer from a degraded QoE. This condition cannot be achieved without an accurate estimation of the network latency and proper detection of the sources of load.

In a replicated service-oriented application, the measurements of the inter-node latencies can be essential over three different levels of resource management:

- **Level 1: Routing**
  Any node in the edge layer may receive and relay requests toward the application replicas. As a result, the node should make the decision of which replica should handle which request. In Kubernetes, the simple approach of equally balancing the load over the different replicas would lead to suboptimal performance in terms of latency. A node should therefore have the means to distinguish between nearby nodes and far away ones, such that a request should have a higher probability in getting routed to a replica hosted in a nearby node. We address this challenge in Chapter 4.

- **Level 2: Placement**
  Routing a request to a nearby replica requires two conditions: the first is proximity-aware routing, and second is the availability of a replica hosted in a nearby node. In consequence, placing an application replica in such geo-distributed layer should depend on the location of the sources of traffic, and their proximity to the applications replica such that a latency-sensitive application can meet its latency threshold. Since the load received by a fog platform may vary over time, a change in the sources of traffic and/or inter-node latencies may change during the operation. The placement problem then has to be treated dynamically, and a re-placement should occur once a change in the load and platform characteristics is detected. Two different proximity-aware placement solutions are detailed in Chapters 5 and 6.

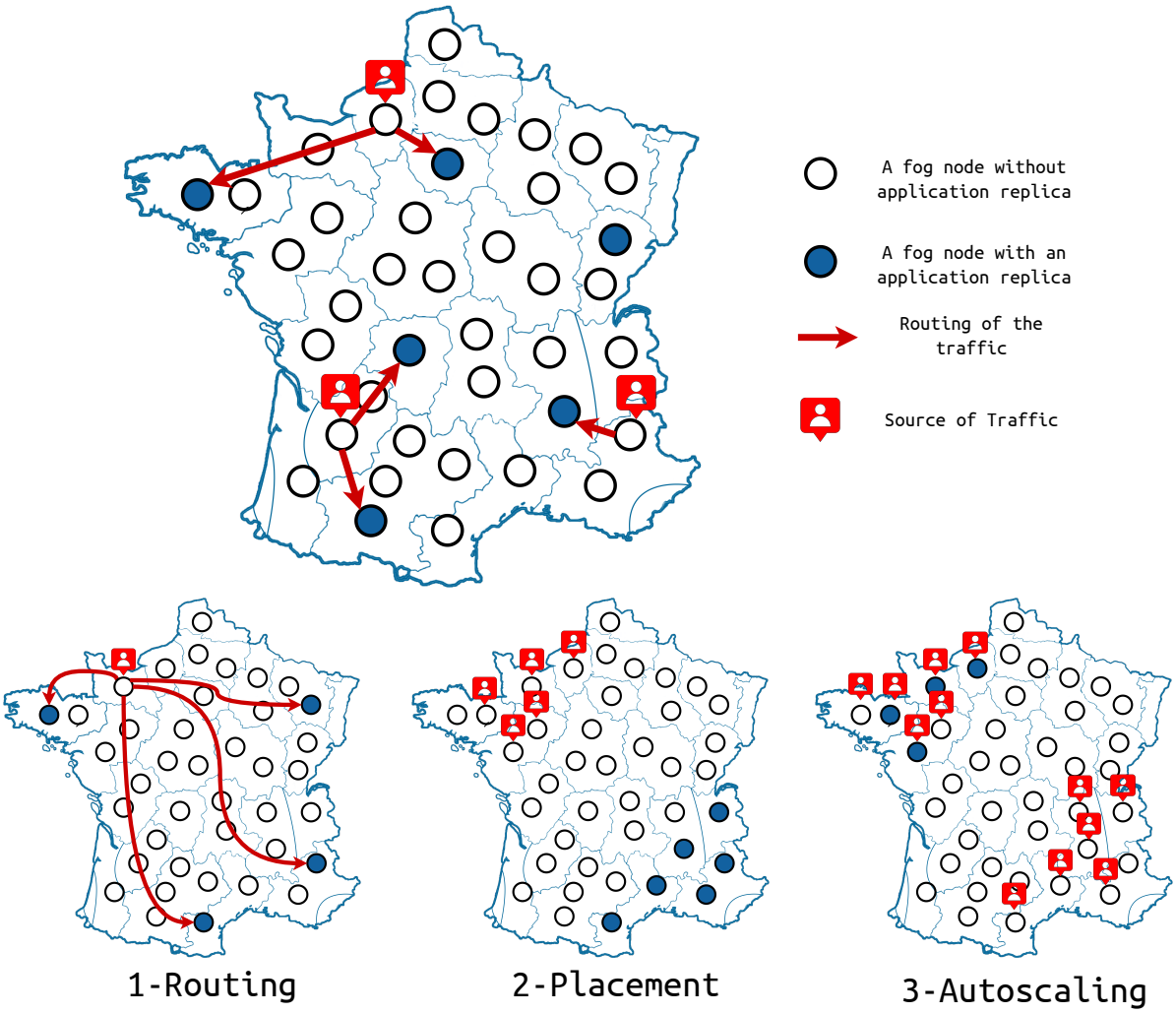


Figure 2.7 – A visual reference of a deployed application and the different layers of resource management.

- **Level 3: Autoscaling**
  The resources in the edge layer are both precious and limited. The number of deployed replicas per applications must therefore be optimized to avoid over provisioning while at the same time to provide a nearby unsaturated replica for all the end users. As a result autoscaling should not only depend on the size of the current load but also on the location from where this load was transmitted, and the application latency threshold. Similar to the placement challenge, autoscaling should be done in a dynamic matter to take the location and the

intensity of load into account before changing the size of the placement. We present an autoscaling algorithm in Chapter 6.

From these three challenges we can conclude that a replicated service-oriented application implemented in fog computing requires information regarding the proximity between the fog nodes. This information serves the purpose of meeting the network latency threshold required by the applications. At the same time, the dynamicity of the load requires the resource management to be dynamic as well. To achieve proximity-aware routing, placing, and autoscaling in the fog platform we therefore have to provide an accurate estimation of the inter-nodes latency (discussed in Section 2.4.1), process the end-user requests within a certain threshold (discussed in Section 2.4.2), and take into account the load variation (discussed in Section 2.4.3).

2.4.1 Latency estimation and Vivaldi coordinates

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, which were designed for data center environments, do not make any attempt at routing end user requests to nearby nodes.

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.

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 network latencies in large-scale network infrastructures [79]. 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, our work relies on Vivaldi coordinates for modeling the latencies between nodes [16]. Vivaldi is a distributed, lightweight algorithm to accurately predict the latency between hosts without contacting them. Using Vivaldi, a node in the cluster

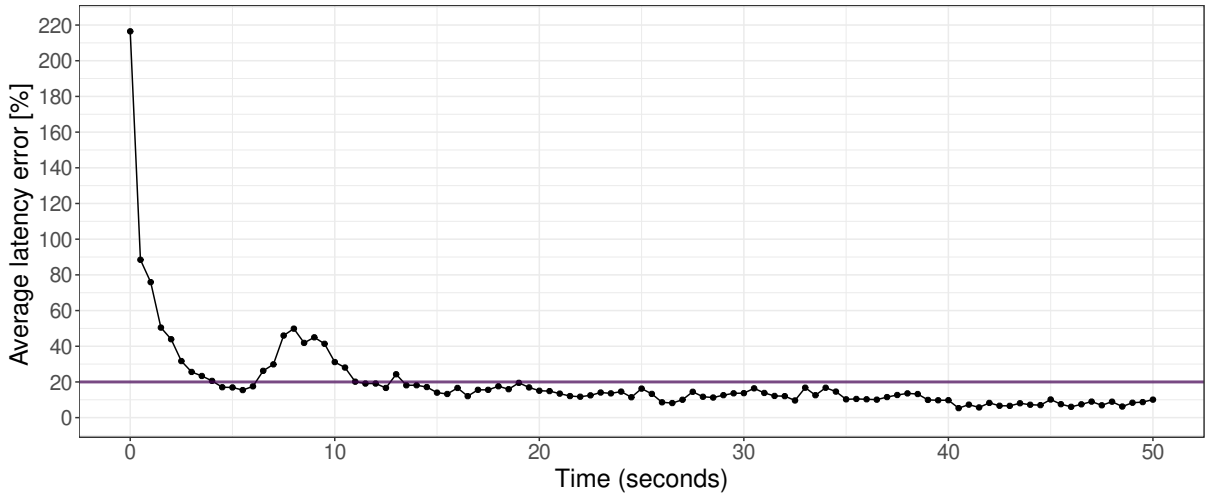


Figure 2.8 – Accuracy of Vivaldi latency predictions for a newly joined node in a 12-node cluster.

can easily compute the latency with all the nodes by communicating with only a few of them.

We specifically use Serf, a mature open-source tool which maintains cluster membership, detects failures, and offers a robust implementation of Vivaldi coordinates [80]. 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 [1].

Figure 2.8 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 wide range of values, this level of prediction accuracy is largely sufficient to distinguish a nearby node from a further away one.

1. `https://www.serf.io/docs/internals/simulator.html`

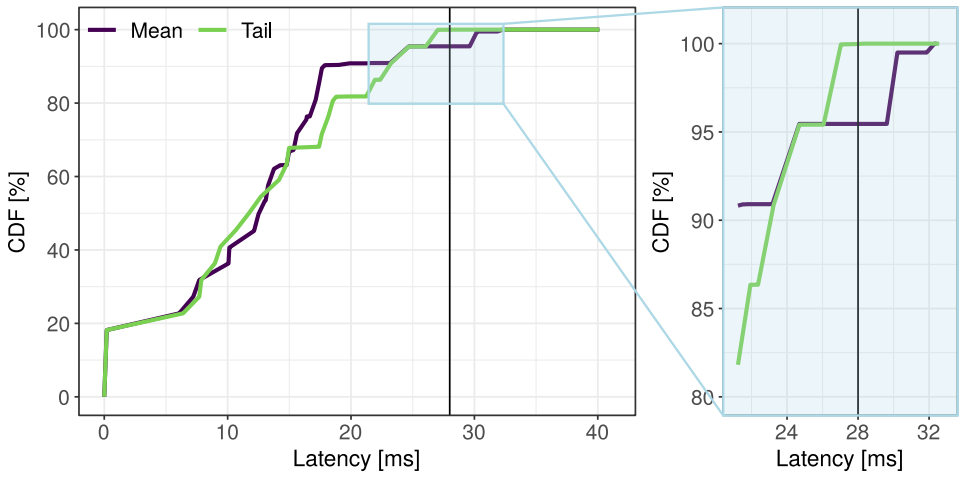


Figure 2.9 – Optimizing the mean or the tail latency.

2.4.2 Optimizing the mean or the tail latency

Fog computing platforms were created for scenarios where the network distance between the user devices and the application instances must be minimized. For instance, virtual reality applications usually require a response times under $20\,ms$. Such applications *"need to consistently meet stringent latency and reliability constraints. Lag spikes and dropouts need to be kept to a minimum, or users will feel detached* [3]*."* 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.

To illustrate the difference between placements which optimize the mean or the tail latency, we explore 50 randomly-chosen placements of 4 replicas within a 22-nodes testbed (further described in Chapter 5). We then select the two placements which respectively minimize the mean ("Mean") and the number of requests with device-to-closest-replica latencies greater than a threshold $L = 28\,ms$ ("Tail"). Figure 2.9 compares the cumulative distribution functions of the obtained latencies delivered by the two placements. Mean delivers very good latencies overall, and it can process many more requests under $20\,ms$ compared to Tail. However, when zooming at the end of the distribution, we see that roughly 5% of requests incur a latency greater than $28\,ms$, and up to $32\,ms$. The users who incur such latencies are disadvantaged compared to the others, and are likely to suffer from a bad user experience.

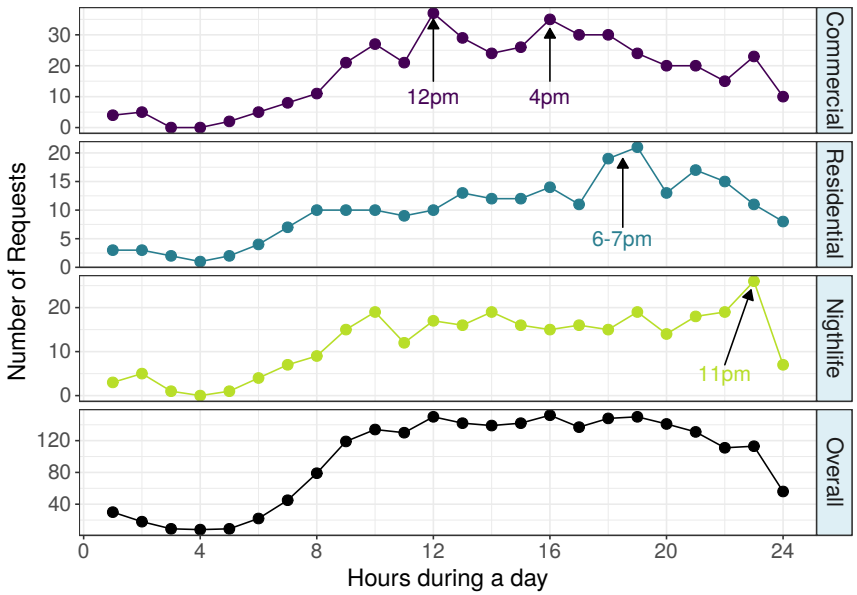


Figure 2.10 – Load variation according to time and space.

On the other hand, with the same number of replicas, Tail guarantees that 100% of requests incur latencies under $27\,ms$. Although the mean latency delivered by this placement is slightly greater than that of the Mean placement, this configuration is likely to provide a much more consistent experience to all the application's users.

In this thesis, we therefore aim to find replica placements which minimize the tail device-to-closest-replica latency, while maintaining acceptable load balancing between the replicas.

2.4.3 Non-stationary traffic properties

Any online application which processes incoming requests from an unbounded user population notoriously experiences significant workload variations across time [17]. 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 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 [81].

Figure 2.10 shows the aggregated requests of users in 10 different areas of the city of Trento, 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 $9\,am$ and $11\,pm$. However, different zones observe workload peaks at different times of the day. Commercial districts with shopping malls and offices peak at $12\,pm$ and $4\,pm$, whereas residential areas peak at $7\,pm$ and nightlife neighborhoods peak at $11\,pm$.

For a replicated service-oriented fog application, 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.

2.5 A complete fog computing architecture

In this section we summarize the conditions and assumptions we took to define the scope of this work. First we dive deep in the network model suggested for the edge layer. Then we present the components of an application running on top of Kubernetes and their characteristics.

2.5.1 Network model

A fog computing network is responsible for carrying packets from the end user to the application's resources. This includes resources both located at the edge layer and at the cloud data centers. As explained in Section 2.2.1, we target latency-sensitive applications. For these applications all the resources will be located at the edge. As a result, we focus in our network model on the routes from the end users toward the edge layer application resources.

The journey of a packet in fog computing starts at the end users layer, where a user submits a request in the form of a network packet toward the application's replicas. As explained in Section 2.2.2, an end user device can be an IoT device, a mobile phone, a wearable technology, or an autonomous vehicle. Our work is agnostic to the nature of end users an it rather focuses on the applications they are communicating with.

After the request is transmitted it will pass through multiple routing levels presented in Figure 2.11:

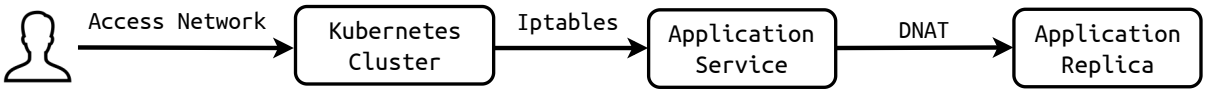


Figure 2.11 – Routing a request from the end user to the application replica.

- **First step: access network.**
  A variety of mechanisms such as DNS redirection and software-defined networking must be first 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 platform 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 may rely on a wide variety of technologies such as LTE and *Software-defined networking* (*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: Iptables routing.**
  In Kubernetes, a packet directed toward an application holds the application service IP as its destination. The packet passes through the chains of Iptables, where it will be forwarded to the application-specific chain which represent a Kubernetes application service.

- **Third step: internal request routing to one of the application's replicas.**
  The packet is further routed internally to the Kubernetes cluster using DNAT which chooses one of the service pods' private IP address as a destination. When the application comprises more than one pod, the Kubernetes service is responsible for selecting one of the replicas to handle the incoming packet (see Figure 2.5).

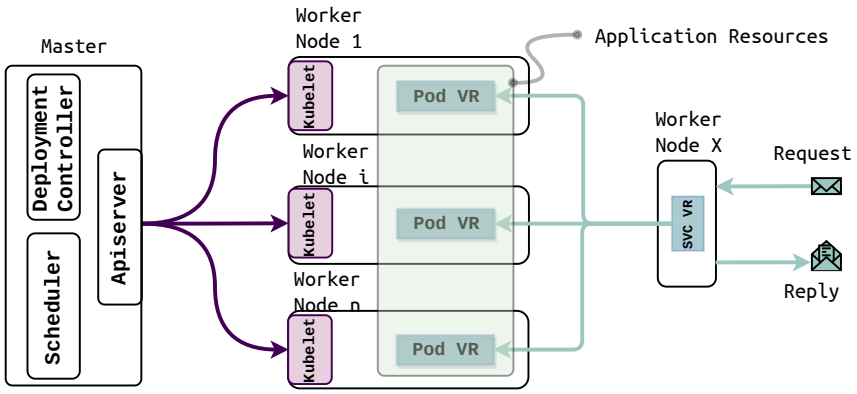


Figure 2.12 – The application model in our scope.

2.5.2 Replicated service-oriented applications in Kubernetes

As explained previously, an application in Kubernetes runs as a deployment consisting of a set of pods. In our case we consider that all the application's pods are functionally identical and represent the application's resources. Each pod in the deployment is a service replica which can receive a request, execute the appropriate function and respond with an answer. We assume that the application replicas process the entire request with no additional calls to other services.

Although the time needed to process the request is application-specific, in this work we decided to ignore this part of the end-to-end request latency. We rather try to minimize the **network** round trip time, and consider the execution time of the request as out of our scope.

We target latency-sensitive applications like VR, AR, and gaming. Latency-sensitive applications have threshold that need to be respected, otherwise a violation in latency can lead to a degraded QoE. Elasticity is another feature that we consider in our work, such that the application replicas either have their own means of reaching consistency, or they are elastic and are not affected by inconsistent replicas. In general, we consider the matter of replica's consistency maintenance out of our scope.

The resources of a replicated service-oriented applications have to be handled in a different manner when implemented in fog computing. Rather than focusing only on the number and size of the resources, in a geo-distributed fog the location of the resources can as well affect the performance of such applications. Resource management in the edge layer should also be influenced by the threshold of

latency-sensitive application. The deployment of the replicas should guarantee a round-trip latency lower than the application's threshold.

As a result, the ultimate objective of this thesis is creating algorithms that can provide such demanding requirements. Respecting such threshold requires one to first enable a proximity aware routing, which we have done by implementing ***Proxy-mity*** presented in Chapter 4. However, routing alone is not sufficient to guarantee all users' request will be returned within the latency threshold. We thereafter implemented placements algorithms, ***Hona*** in Chapter 5 and ***Voilà*** in Chapter 6 that creates a dynamic placement based on the location of the sources of traffic. To amount for the changing load size and the non-stationary load we as well implemented an autoscaling algorithm in Voilà.

Chapter 3

State of the art

In a fog computing platform, edge-layer resources are limited in numbers and in computing capacity while remaining essential for a wide array of applications. This has made resource management in fog computing and specifically in the edge layer a hot research topic. Numerous resource management mechanisms have therefore been proposed to address various aspects of fog computing resource management [82, 83].

Different authors have chosen different approaches to target a multitude of resource scheduling challenges in the fog. The many combinations of system model, application model, virtualization technique, type of requests, and optimization objective have produced different strategies for resource management in fog computing:

- **System model:** numerous platforms like MEC [84–87], F-RAN [35, 36, 88, 89], and Multi-tier [43] computing can be considered one way or another as a form of fog computing. However, each one of these platform features a unique set of requirements to be delivered through resource management. For example, MEC resource management focuses on the layer where the users tasks are executed. F-RAN resource management in the other hand divides the incoming requests according to the cellular base station where the end users are connected.
- **Application model:** as explained in Section 2.2.1, various classes of applications are intended to make use of the fog platforms. However, different types of applications require different resource management mechanisms [4, 90]. Most notably, stream processing and latency-sensitive applications' resource management should be carried away as application-centric, since the objective is to guarantee the application's QoS. In contrast, resource management for less demanding applications is typically handled as platform-centric where the objective is to improve the platform performance in terms of energy consumption and/or resource provisioning.

- **Virtualization techniques:** Much like cloud platforms, fog platforms' edge-layer 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 rather than virtual machines [41], enabling one to envisage fog computing platforms making use of edge resources as limited as Raspberry Pis [85, 91, 92]. The choice of virtualization technique may influence the scheduling algorithms. For example, the time needed for a *Virtual Machine* (*VM*) to boot up should be taken into account in the scheduling algorithm. In contrast, containers boot up is much quicker. Although containers deployment on modest machines may be painfully slow, a variety of techniques have been proven to significantly speed up this operation [93]. In such cases, the deployment time of containers can be considered relatively negligible compared to that of VMs.

- **Type of requests:** the type of expected workload influences the choice of resource management techniques. In the literature, some researchers consider an application request as an encapsulated task which must execute in the edge layer without previously reserved resources. In other cases, like in our approach, we consider the request as a set of parameters that will be executed in an already deployed application resource. This distinction drives two very different sets of requirements for the resource management.

- **Optimization objective:** different authors follow different objectives for the resource management [4, 33, 82]. The objective depends mainly on the type of target application and the proposed system model. Resource management objective can be: latency minimization [94–96], energy optimization [97–99], resource utilization [100], network usage [70], etc.

As discussed in Section 2.4, we consider three levels of resource management. Although the literature targets different aspects and objectives of resource management in fog computing, we can still categorize the contributions according to these levels. The first level deals with assigning user requests and tasks to applications resources. We define service placement and re-placement as the second level of resource management. Finally, the third level concerns the autoscaling of the provisioned resources. We discuss these three levels in turn.

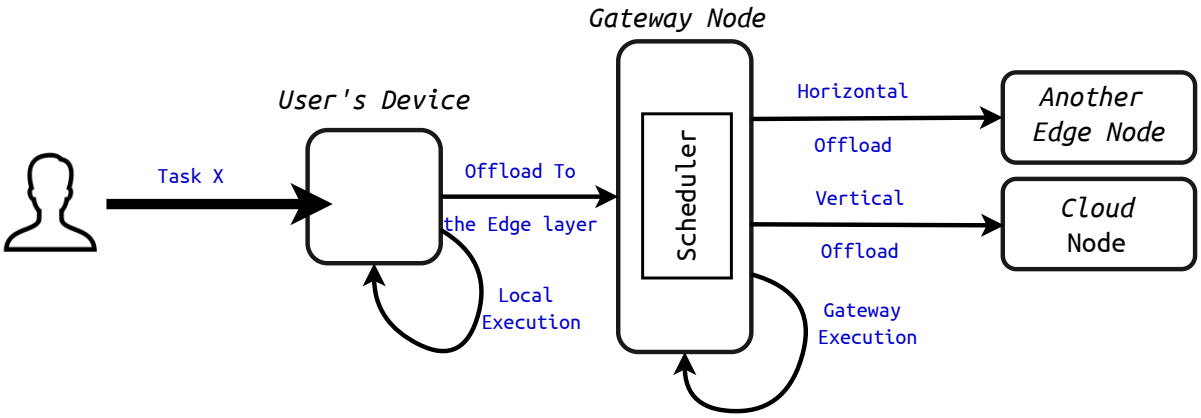


Figure 3.1 – Offloading in fog computing.

3.1 Workload routing

Workload routing controls the flow of the users' workload toward the fog resources. Two directions in workload routing can be distinguished. The first one considers the users' requests as encapsulated tasks that will provision a resource, execute, and then release the resource. Another direction considers the applications as a set of replicated service-oriented server processes that receives operation parameters via user's request and then return the output of the operation. In this case the resources are provisioned prior to receiving the users' requests. Throughout this section we will refer to the workload as a set of service-oriented *requests*. When the applications work by provisioning and releasing a resource it will be refereed to as a set of *tasks* that may be offloaded to the fog.

3.1.1 Task offloading

Offloading is the process of relocating a task from one node to another. Offloading techniques are designed to optimize resource utilization in terms of cost, energy consumption, or response time. This concept is popular in *Mobile Cloud Computing* (*MCC*), where the application decides whether to execute on the user device or to be offloaded to a cloud data center [98]. With the introduction of *Mobile Edge Computing* (*MEC*), this concept gained a similar traction, however with an access to two different types of resources, the edge resources and the cloud data centers [87].

Figure 3.1 showcases a typical offloading scenario where an end user device can either process the task locally, otherwise the task will be offloaded to the fog platform.

In the fog platform also a gateway node can decide to process the task locally or offload it horizontally to another edge node, or vertically to a cloud node.

Among the many papers that proposed an offloading technique, a number of them have worked on reducing application response time. *Mukherjee et al.* reduce the overall task completion latency by solving a quadratically constraint quadratic programming optimization problem [94]. *Vu et al.* offload services in fog radio access networks to optimize the energy consumption and offload latency [89]. *Yousefpour et al.* propose a delay-minimizing offloading policy for fog nodes to reduce service delay for the IoT nodes [101]. *Sun et al.* propose the ETCORA algorithm to solve the energy and time cost minimization problem for IoT-fog-cloud architecture [99]. *Basic et al.* propose a fuzzy handoff control to avoid tasks being offloaded horizontally between edge nodes. The algorithm takes the application response time as an indicator whether to move the task to another node [102].

Although these works optimize tasks execution latency, most of the presented articles are designed for IoT applications and the deadlines associated with such applications. Also in the majority of the presented work, the edge layer's network latencies are either not considered [94, 99] or defined as a constant between all the nodes [89, 101].

In contrast, our work targets latency-sensitive applications with latency thresholds as low as $20\,ms$. A design based on transmitting a task and waiting for the resource to be ready before the execution is not appropriate for such applications. Also, choosing some tasks to be processed in the backend cloud does not allow one to reduce the tail execution latency. In this thesis, we thus aim at processing all the requests in the fog layer, without offloading to a backend cloud.

3.1.2 Request routing

A replicated service-oriented application hosted in the cloud often uses a proxy to balance incoming requests over the available replicas. In fog computing however, the responsibilities of the proxy are not limited to load balancing between a set of functionally equivalent replicas, but rather extend to routing the requests toward the nearest available replica. In the literature, a number of routing techniques have been suggested to optimize latency, energy consumption, and backhaul network traffic while maintaining a balanced load [103, 104].

In the field of *Vehicular Ad hoc NETworks* (*VANETs*), a moving vehicle requires a dynamic update of the nearby resources to establish routes between the vehicle and an edge node. *Lu et al.* propose the IGR routing scheme [105]. Based on the street map and the position of vehicles, IGR selects the routing path according to the packet error rate of each link and vehicle density of each street. SFIR takes advantage of SDN and fog computing technologies to improve data forwarding in vehicle-to-vehicle communication [106]. *Kadhim et al.* presented an energy-efficient multicast routing protocol based on SDN and fog computing [97]. This protocol exploits a mathematical model to select the optimal multicast path with minimum energy and take into account deadline and bandwidth constraints. Yet, routing in VANETs is designed with mobility in mind and aims at providing a reliable link to the destination resources. In contrast, our approach focuses on delivering a low user-to-resource latency by exploiting the inter-nodes latencies.

PiCasso is a container orchestration platform that specifically targets edge clouds with a focus on lightness and platform automation [68]. The developers of PiCasso intend to develop a service proxy that will redirect user's requests to the closest node. However, no technical detail is provided in the paper.

Puthal et al. propose a secure and sustainable load balancing technique which aims to distribute the load to the less-loaded edge data centers [107]. 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 [108]. Although these algorithms improve the application's performance by preventing over saturation, 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 [109]. 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.

Table 3.1 – Literature classification for application placement algorithms.

Type	Ref.	Dyn.	Rep.	Obj.	Eval.	Type	Ref.	Dyn.	Rep.	Obj.	Eval.
Data	[113]	✗	✗	NU	Sim	Service Oriented	[121]	✗	✗	RT,RU	Sim
	[114]	✗	✗	RT	Sim		[122]	✗	✓	DT	Testbed
	[115]	✗	✗	RT	Sim		[123]	✗	✗	PX,DT	Sim
	[116]	✓	✓	RT	Sim		[124]	✗	✗	PX	Sim
	[117]	✓	✓	RT	Sim		[125]	✗	✗	PX,RU	Sim
VM	[118]	✗	✗	NU	Sim		[126]	✓	✗	PX,DT	Testbed
	[119]	✗	✓	NU	Sim		[127]	✓	✓	PX,RU	Testbed
	[120]	✓	✓	NU	Sim		Hona	✓	✓	PX,LB	Testbed+Sim
							Voilà	✓	✓	PX,ST	Testbed+Sim

We present Proxy-mity, our solution for requests routing in fog platforms. Proxy-mity estimates the inter-node latencies using Vivaldi coordinates then automates the pod selection process by implementing a trade-off between proximity and fair load balancing. The request routing take place in the IP layer using kernel-level iptables, which ensures fast redirection.

3.2 Placement and re-placement

As mentioned previously, in fog platform users' requests are routed toward already deployed application instances. The request can be a database query, a set of parameters to be executed on a service, or a request for image/video processing. Unlike the encapsulated tasks for which the resources are reserved and released, requests are routed toward a set of unique application instances that can handle the request. This approach requires one to place the application instances in nodes that improve the overall performance of the application. The performance optimization is tied to the choice of nearby nodes that have enough resources to support the application's instance.

The instances placement problem has been extensively studied since the creation of the first geo-distributed environments such as content delivery networks [110–112], and a very large number of papers have been published on this topic.

Table 3.1 presents the most relevant recent publications on replica placement in fog and edge computing systems [113–127]. The papers are classified along multiple dimensions:

Type describes *what* is being placed. Data placement focuses on decreasing the download delay of cached items by placing specific cache in a specific location [114–117]. In contrast, VM placement typically aims to reduce network usage [118–120], while service placement optimizes mostly network proximity and resource utilization [123–127].

Dynamicity (Dyn) is important in systems which may experience considerable workload variations over time. Many papers focus on the initial placement problem only, without attempting to update these placements when the workload changes.

Replication (Rep) indicates whether the proposed systems aim at placing a single object instance, or a set of replicas.

Objective (Obj) represents the metric(s) that the placement algorithms aim to optimize: Response Time (RT) represents the overall response latency including network and processing latency; Network Usage (NU) is the volume of backhaul traffic; Resource Utilization (RU) is the effective use of the available resources; Deployment Time (DT) is the time needed for the algorithm to find and deploy a solution; Availability (AV) is the probability that a system is operational at a given time; Proximity (PX) is the latency between end-user and the closest application instance; Load Balancing (LB) is the distribution of the load over application's replicas; and Saturation (ST) is the performance degradation induced by overloaded replicas.

Evaluation (Eval) of placement algorithms is often done using simulators such as CloudSim [128] and iFogsim [129] to evaluate their solutions. However, some authors also use actual implementations and evaluate them in a real environment or a testbed.

Few papers in Table 3.1 propose dynamic placement algorithms for replica sets. Yu *et al.* study the placement of replicated VMs in order to minimize the backhaul network traffic [120]. The algorithm considers the proximity of end users to the fog nodes, but does not take the proximity between distributed fog nodes into account.

Aral *et al.* [116] and Shao *et al.* [117] propose dynamic replica placement algorithms for data services in edge computing. Similarly, Li *et al.* [127] present a replica placement algorithm to enhance data availability. All these papers use the mean latency as their metric for response time evaluation. However, as discussed in Section 2.4.2, optimizing the mean latency does not necessarily imply an improvement in the human-perceived quality of service as it does not give guarantees on individual

request response time [130]. These papers [116, 117, 127] also do not consider the impact of load distribution over the replicas on the application performance. Finally, only [127] has implemented and tested its proposed algorithms in a real testbed.

In contrast, to our best knowledge, our contributions Hona and Voilà present the first dynamic replica placement algorithms which aim to maintain the tail latency within pre-defined bounds. Hona (presented in Chapter 5) finds a solution based on tail latency and load balancing to ensure a fair distribution of the load over a fixed size replicaset [19]. Voilà (presented in Chapter 5) on the other hand has the capability to scale the replicaset's size according to the tail latency and the pods' saturation [20]. Hona and Voilà solve the dynamic placement problem based on the network routes as well as the distribution of traffic, and have been implemented in a mature container orchestration system.

3.3 Autoscaling

Fog computing applications are expected to face workload variations throughout the day. Such variations can take the form of variations as function of time–for example, the typical day/night patterns– as well as variation in terms of the sources of traffic induced by the non-stationary nature of the end users. As a result, an autoscaling scheme must be implemented to scale the application resources to adapt for the perceived changes in the workload.

Although autoscaling has been extensively studied over the previous decades, its application in the context of fog computing platforms remains nascent. This can be attributed to the prevalent spread of vertical offloading as an alternative for autoscaling [82]. If an application is facing saturation in the provisioned computing resources, then the straightforward solution would be offloading a part of the requests to the cloud. Although such approach is valid for certain types of applications, this approach does not offer a feasible solution for latency-sensitive applications, for all the reasons mentioned in Section 3.1.1.

Nevertheless, the literature includes a wide range of autoscaling techniques in similar platforms. Autoscalers designed for general-purpose Kubernetes clusters aim at providing a seamless service for the application users [131–133]. Geo-distributed computing environments autoscalers aim at selecting the nearest available resource [134, 135]. Furthermore, countless autoscaling algorithms for cloud

computing which have various objectives like cost reduction, performance optimization, etc. [136].

Few papers propose auto-scaling systems designed for fog computing platforms. *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 [77].

On the other hand, ENORM aims to reduce latency between users and computing devices, and the network traffic to the cloud [137]. However, it chooses the resources regardless of their location, and essentially considers every fog node as equivalent to one another. ORCH proposes to dynamically add compute nodes in the system to resolve workload surges [96]. 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 [20]. It aims at optimizing the tail request latency rather than its mean while avoiding replica overloads. Voilà's autoscaling and placement algorithms, and their evaluations, are presented in Chapter 6.

3.4 Conclusion

Figure 3.2 places our contributions in a visual representation of the discussed state of the art. Each of our contributions targeted a problem and provided a lightweight solution that was implemented on top of Kubernetes. Our contributions were evaluated both using a realistic testbed to validate the feasibility, and simulation to validate the scalability. The figure highlights our contributions that focused on latency-aware resource management. Each of the contribution include a solution for specific resource management challenges. In contrast, the accumulation of our three contributions presents a complete solution for all three levels of resource management. Our contributions were implemented on top of Kubernetes, although one may argue that the same algorithms can be easily adapted for any other orchestration engine.

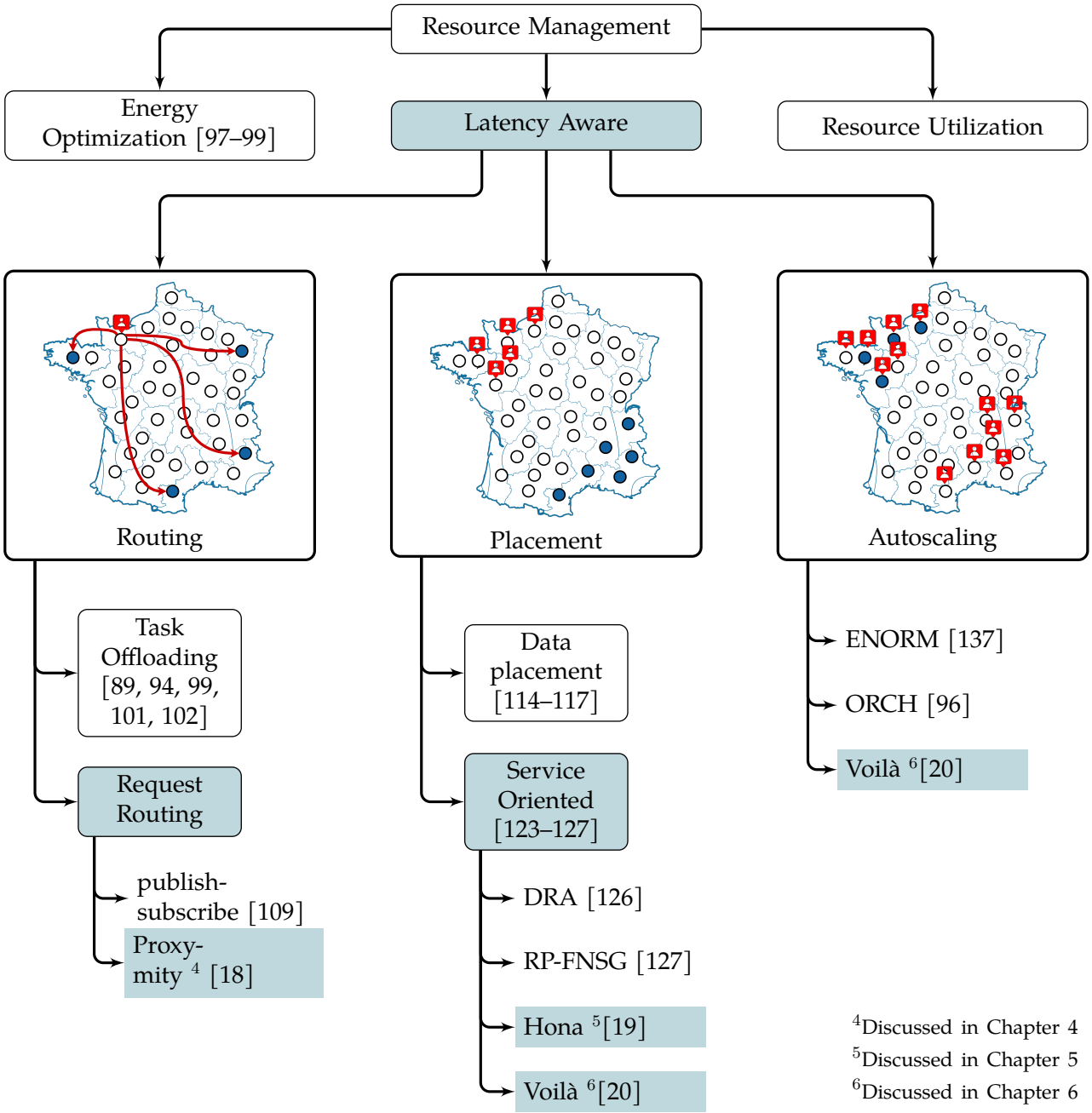
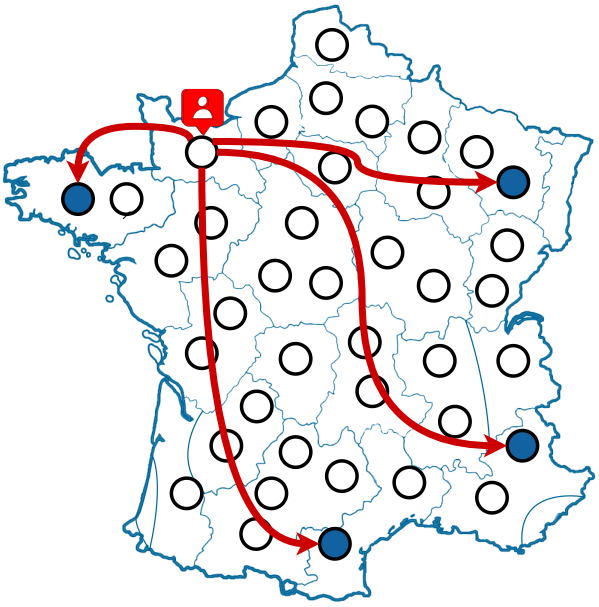


Figure 3.2 – State of the art.

Chapter 4

Proximity-aware request routing



In this chapter, 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. The evaluation shows that Proxy-mity 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.

4.1 Introduction

A large range of emerging fog computing applications such as augmented reality and autonomous driving systems 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 platforms 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 [15]. On the other hand, systems like Mesos [13], Docker Swarm [12] and Kubernetes [11] 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 chapter, 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 be adapted to integrate in other container orchestration systems.

Proxy-mity exposes a single easy-to-understand configuration parameter α 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 chapter is organized as follows. In Section 4.2, we showcase the design of Proxy-mity, how proximity is measured using Vivaldi coordinates, how the trade-off

between proximity and load balance is implemented, and how the algorithm was implemented in the iptables routes. Section 4.3 evaluates the performance of Proxy-mity using a realistic testbed and simulations. Finally, in Section 4.4 we conclude.

4.2 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 [80], a lightweight implementation of Vivaldi coordinates [16]. 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*.

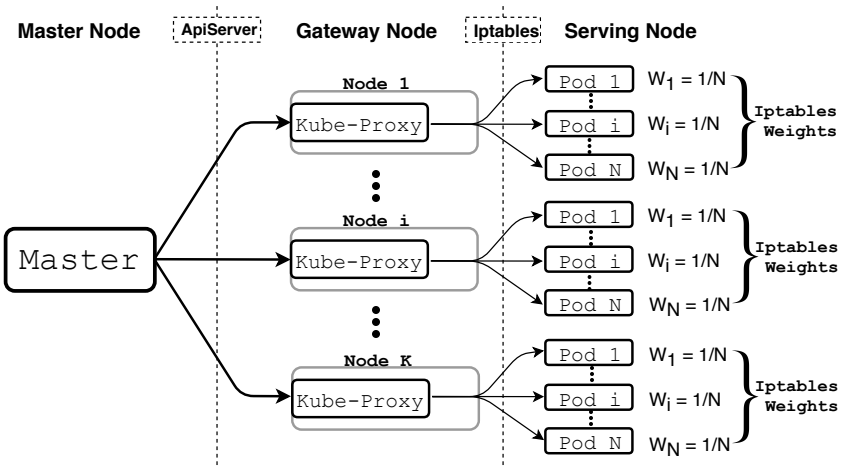
In the next sections we respectively discuss the overall system architecture, the representation and measurement of proximity between nodes, the calculation of weights to be associated with each route, and the injection of new routes in the local Linux kernel.

4.2.1 Architecture

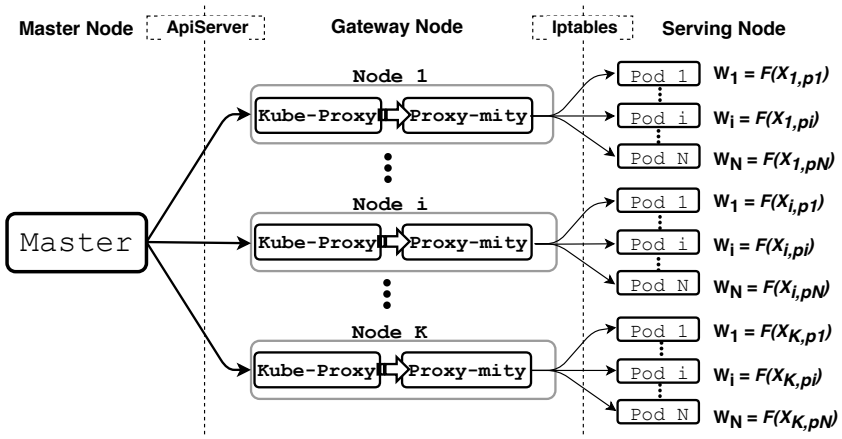
Kubernetes is designed as a set of control loops. It therefore continuously monitors itself, and takes corrective actions when the state it observes deviates from the specification of the desired system state. This organization makes it highly dynamic and robust against a wide range of situations. The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

As shown in Figure 4.1a, 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

1. `https://github.com/alijawadfahs/FOG-aware`



(a) kube-proxy architecture.



(b) Proxy-mity architecture.

Figure 4.1 – Architectures of kube-proxy and Proxy-mity.

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 rules 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.

The architecture of Proxy-mity, presented in Figure 4.1b, 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.

4.2.2 Measuring proximity

As mentioned in Section 2.4.1, to avoid the overhead of periodically measuring N^2 pairwise latencies between N nodes, Proxy-mity relies on Vivaldi coordinates [16] 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 [80], 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.

4.2.3 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 Φ composed of N functionally identical pods:

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

where each φ 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_α:

$$F_\alpha(\varphi) = \alpha.P(\varphi) + (1-\alpha).L(\varphi) \tag{4.1}$$

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 φ_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:

$$P(\varphi_i) = \frac{f_\beta(l_i)}{\sum_{j=1}^{N} f_\beta(l_j)} \tag{4.2}$$

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

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

$$f_\beta^{inverse}(l) = \frac{1}{\beta l} \tag{4.3}$$

$$f_\beta^{power}(l) = \frac{1}{l^\beta} \tag{4.4}$$

$$f_\beta^{exponential}(l) = e^{-\beta l} \tag{4.5}$$

As we will discuss in Section 4.3, different decay functions have different levels of aggressiveness in selecting nearby pods.

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

$$F_{\alpha,\beta}(\varphi_i) = (1-\alpha).\frac{1}{N} + \alpha.\frac{f_\beta(l_i)}{\sum_{j=1}^{N} f_\beta(l_j)} \quad \forall \varphi_i \in \Phi \tag{4.6}$$

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 4.3, 4.4 or 4.5 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.

4.2.4 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

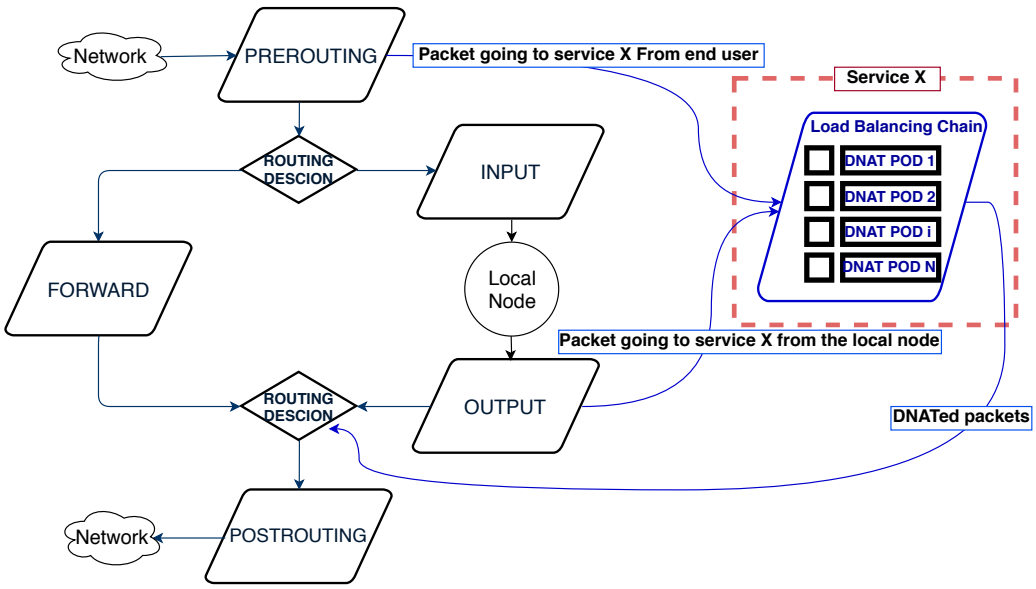
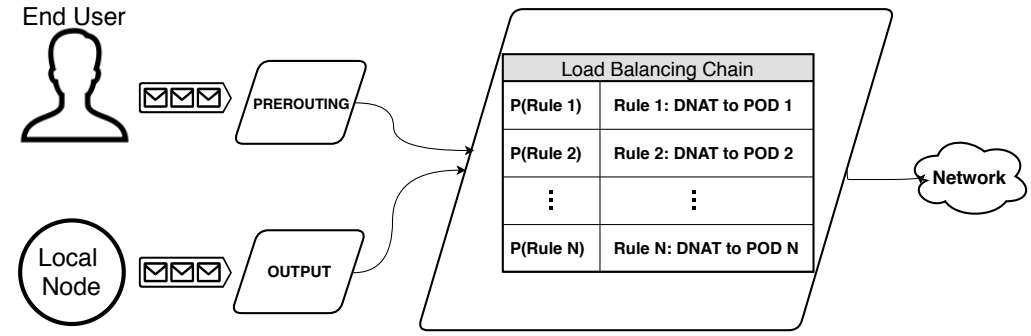


Figure 4.2 – Iptables chains and load balancing.

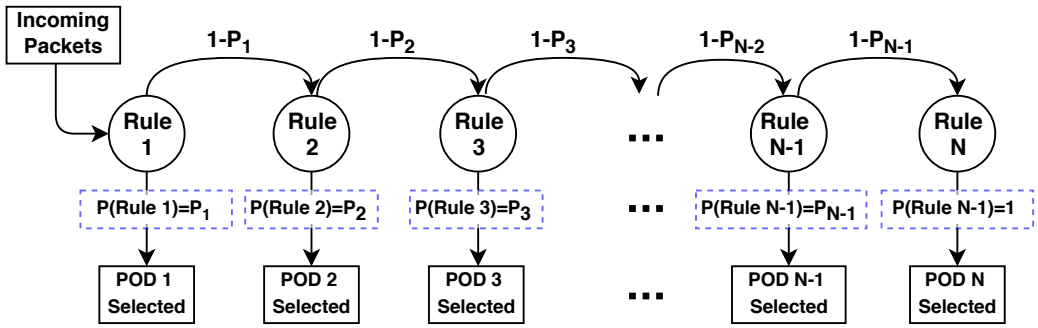
firewall in the form of *iptables* rules. *iptables* defines chains of rules for the treatment of packets where every chain is associated with a different kind of packet processing. Packets are processed by sequentially traversing the rules in their chains.

As illustrated in Figure 4.2, 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 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 4.3a, 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.



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



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

Figure 4.3 – Load balancing rules in iptables.

Rules are executed in a predefined sequential order, so the probabilistic load-balancing system actually implements a Markov chain, as shown in Figure 4.3b. 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 $P(Pod_i)$:

$$P(Pod_i) = \begin{cases} P_1 & \text{if } i = 1 \\ P_i \times \prod_{j=1}^{i-1}(1 - P_j) & \text{if } 1 < i < N \\ \prod_{j=1}^{N-1}(1 - P_j) & \text{if } i = N \end{cases} \tag{4.7}$$

Injecting a set of weights $\mathbb{W} = \{w_1, w_2, ..., w_N\}$ as computed in Equation 4.6 for a deployment Φ 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:

$$P_i = \begin{cases} w_1 & \text{if } i = 1 \\ w_i \times \frac{1}{\prod_{j=1}^{i-1}(1-P_j)} & \text{if } 1 < i < N \\ 1 & \text{if } i = N \end{cases} \tag{4.8}$$

Upon every detected modification in the set of pods belonging to a Kubernetes service, Proxy-mity therefore recomputes the weights w_i using Equation 4.6 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 4.8 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.

4.3 Evaluation

4.3.1 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 4.4. 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 [138, 139].

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 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

2. `https://blog.hypriot.com/downloads/`

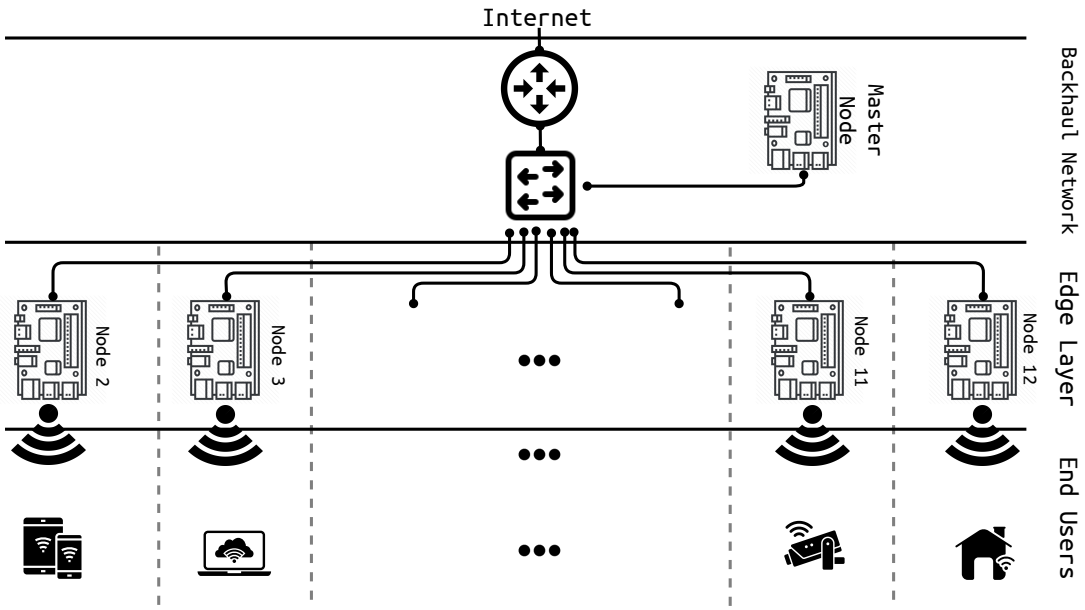


Figure 4.4 – Experimental testbed organization.

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 4.1. 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.

4.3.2 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.

3. `https://wondernetwork.com/`

Table 4.1 – Inter-node network latencies (in ms).

	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

Figure 4.5 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.

4.3.3 Service access latency

We first evaluate the effectiveness of Proxy-mity in distributing load according to a given proximity/load-balancing trade-off α and decay function $f_\beta(l)$. We deploy Proxy-mity with a Kubernetes service which contains a small Web server that simply returns

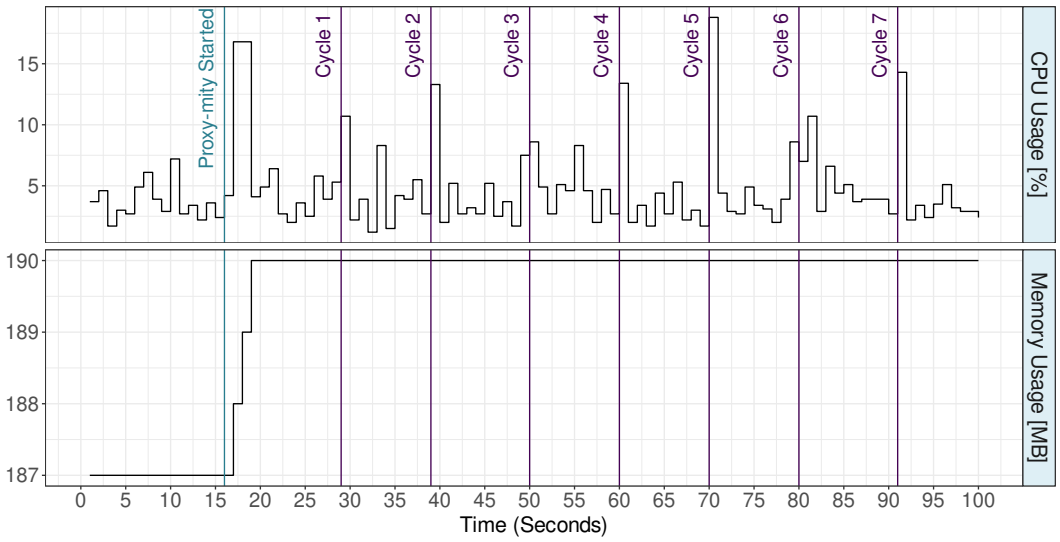


Figure 4.5 – CPU and memory usage for Proxy-mity.

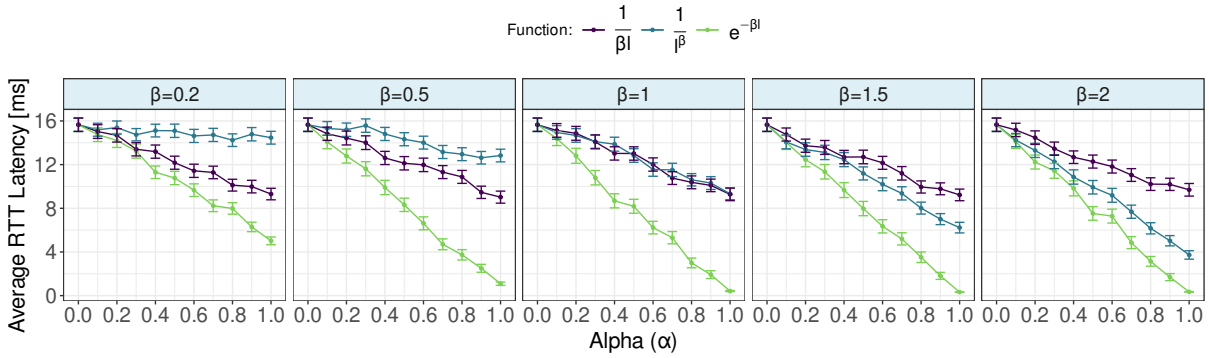
Table 4.2 – Proxy-mity evaluation parameters.

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

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.

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 4.2.

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



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

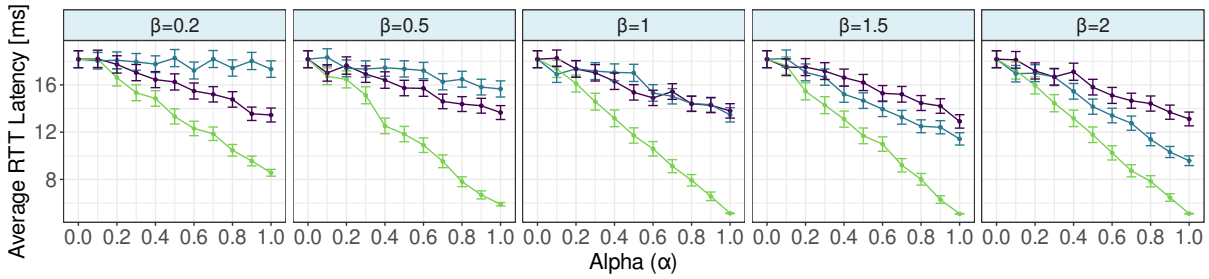


Figure 4.6 – Average service access latency.

Figure 4.6 shows the average measured end-to-end latency for various values of α, choices of decay function, and values of β: Figure 4.5a shows the results with a deployment of 11 pods, and Figure 4.5b shows the results with a deployment of 5 pods.

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

Effect of parameter β and the choice of decay function All evaluated decay functions achieve similar results where greater values of α 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 β 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 β to a single "medium" value per decay function.

4.3.4 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 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 4.7. 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 α 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

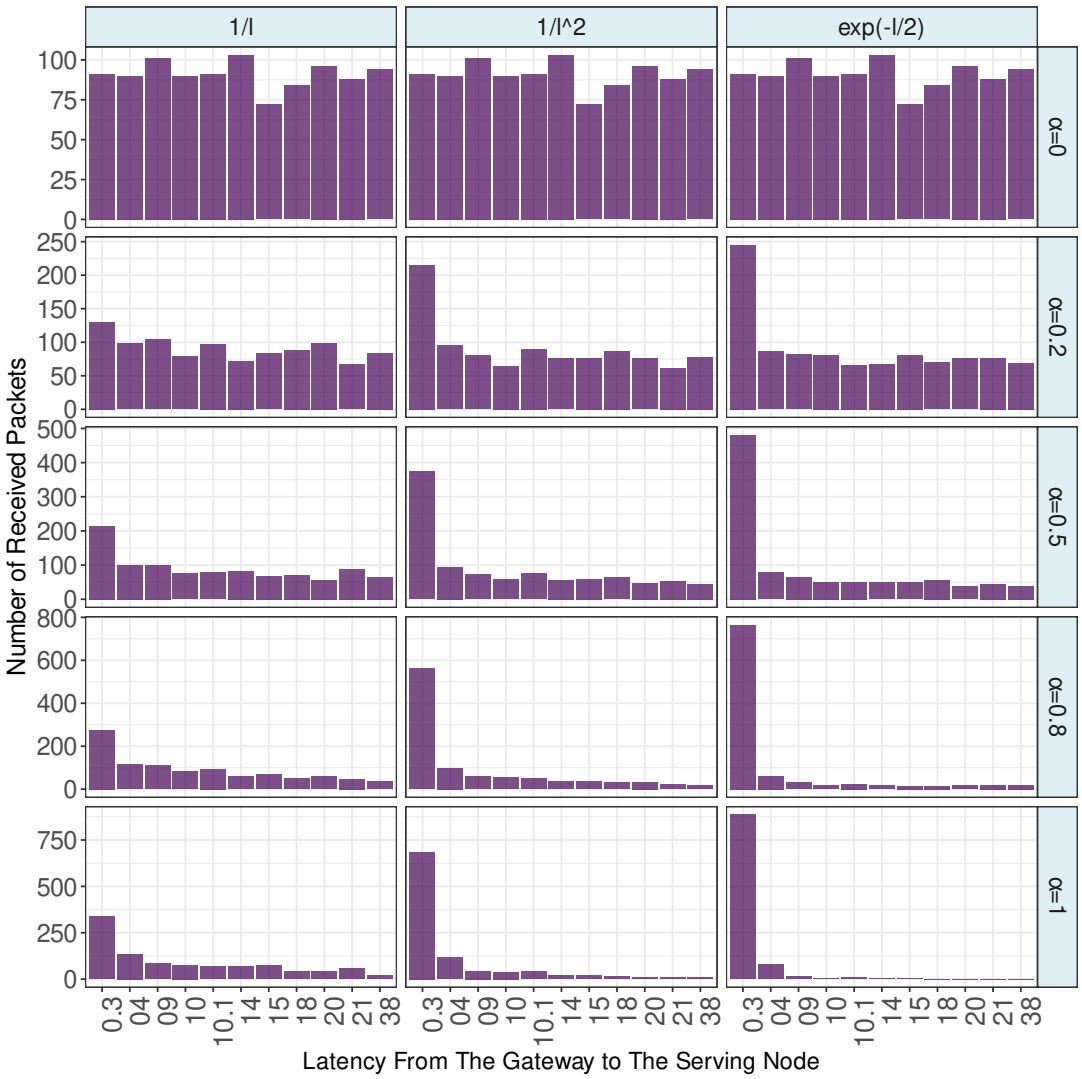


Figure 4.7 – Load distribution as a function of α and $f(l)$.

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.

4.3.5 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

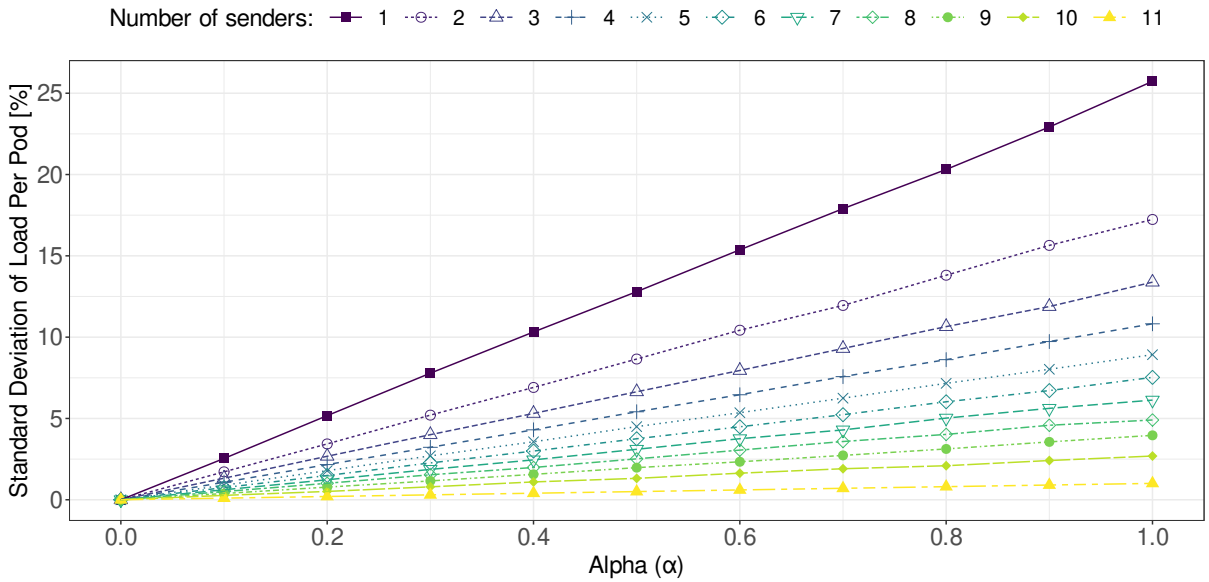


Figure 4.8 – Overall system load imbalance as a function of α and the number of senders.

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 4.6 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 4.1. 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 4.8 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 α 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 α takes intermediate values between 0 and 1, the load imbalance varies linearly between these two extremes. This means that the criteria for choosing a good value for α can be explained to system administrators in a precise yet very intuitive manner: *"α 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"*.

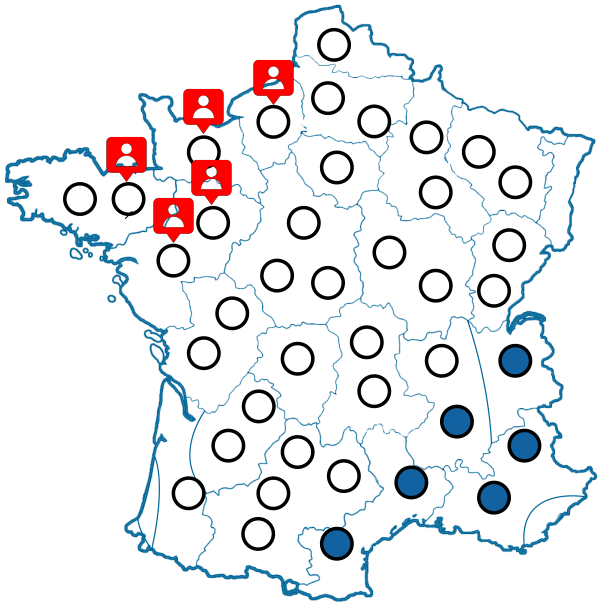
4.4 Conclusion

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.

The work in this chapter is the first step toward a proximity-aware orchestration engine. Solving the routing problem is a very important step to solve the placement and autoscaling problems as we are going to see in the next two chapters.

Chapter 5

Tail-latency-aware placement/re-placement



In this chapter, we propose Hona, a latency-aware scheduler integrated in the Kubernetes orchestration system. Hona maintains a fine-grained view about the volumes of traffic generated from different user locations. It then uses simple yet highly-effective heuristics to identify suitable replica placements, and to dynamically update these placements upon any evolution of user-generated traffic. Our evaluations show that Hona efficiently identifies instance placements which reduce the tail latency. At the same time, it keeps computation complexity low and maintains reasonable load balancing between the replicas.

5.1 Introduction

Choosing the best set of fog servers where an application should deploy its replicas requires one to follow two objectives. First, the chosen placements should minimize the network latencies between end-user devices and their closest application replica. To deliver outstanding Quality-of-Experience to the users it is important that each and every issued request gets processed within tight latency bounds. We therefore follow best practice from commercial content delivery networks [8] and aim to minimize the *tail latency* rather than its mean, for example, defined as the fraction of requests incurring a latency greater than some threshold. Second, a good placement should also allow the different replicas to process reasonably well-balanced workloads. When application providers must pay for resource usage, they usually cannot afford to maintain replicas with low resource utilization, even if this may help in reducing the tail device-to-replica latency.

Selecting a set of replica placements within a large-scale fog computing infrastructure remains a difficult problem. We first need to monitor the usage of the concerned applications to accurately identify the sources of traffic and their respective volumes. Then, we must face the computational complexity of the problem of choosing r nodes out of n such that at least $P\%$ of end-user requests can be served in less than $L\,ms$ by one of the chosen nodes, and the different application replicas remain reasonably load-balanced. Replica placements must then be updated when the characteristics of end-user requests change. Finally, we need to integrate these algorithms in an actual fog orchestration platform.

We propose Hona[1], a tail-latency-aware application replica scheduler which integrates within the Kubernetes container orchestration system [11]. Hona uses Kubernetes to monitor the system resource availability, Vivaldi coordinates to estimate the network latency between nodes [16] and Proxy-mity to monitor traffic sources and to route end-user traffic to nearby replicas [18]. Hona uses a variety of heuristics to efficiently explore the space of possible replica placement decisions and select a suitable one upon the initial replica placement. Finally, it constantly monitors the performance of the current placement and automatically takes corrective re-placement actions when the characteristics of the end-user workload changes.

1. Hona (هُنَا) means “here” in Arabic.

Our evaluations based on a 22-node testbed show that Hona's heuristics can identify placements with a tail latency very close to the theoretic optimal placement, but in a fraction of the computation time. Hona's placements also deliver an acceptable load distribution between replicas. The re-placement algorithm efficiently maintains a very low tail latency despite drastic changes in the request workload or the execution environment. Finally, we demonstrate the scalability of our algorithms with simulations of up to 500 nodes.

This chapter is organized as follows. Section 5.2 presents the system model, how Hona collects its information, and the initial replica and re-placement algorithms. Then, Section 5.3 evaluates this contribution and Section 5.4 concludes.

5.2 System design

Hona [2] dynamically chooses the placement of fog application replicas in a fog computing infrastructure to substantially reduce the user-experienced tail latency (thereafter referred to as Proximity) while keeping replicas load-balanced (thereafter referred to as minimizing Imbalance).

To realize a concrete implementation of this idea, we address the following questions in turn:

1. How should we define objective functions to represent the Proximity and Imbalance of a replica placement decision? (§ 5.2.1);
2. How should we monitor a fog computing infrastructure to precisely identify the sources of traffic? (§ 5.2.2);
3. How should we explore the space of possible choices to select an initial replica placement? (§ 5.2.3);
4. How should we update an prior replica placement choice in reaction to changes of the user-generated traffic patterns? (§ 5.2.4);
5. How can we integrate these different techniques in Kubernetes with minimum modifications to its source code? (§ 5.2.5).

2. `https://gitlab.inria.fr/afahs/hona-code`

5.2.1 System model

We define a fog computing infrastructure as a set of n server nodes $\Delta = \{\delta_1, \delta_2, \ldots, \delta_n\}$, where each δ_i is an object of class `Node` which holds information on the status of the node, its Vivaldi coordinates, and its current request workload. Similarly, we define a deployed application as a set of r replicas $\Phi = \{\varphi_1, \varphi_2, \ldots, \varphi_r\}$ (with $r \leq n$). A `Replica` object φ_i holds information on the status of the replica, its hosting node, its current request workload and the locations from which this workload originates.

The replica placement problem can be formulated as the mapping of every replica $\varphi_i \in \Phi$ to a server node $\delta_j \in \Delta$ to optimize some pre-defined utility metrics. It can be solved in principle by exploring the set of all possible placement decisions $\Omega = \{c_1, c_2, \ldots, c_k\}$ where $c_i \subset \Delta$ and $|c_i| = r$. However, the number k of possible placements is extremely large even for modest values of r and n, so the usage of a heuristic is necessary to efficiently identify interesting placement decisions.

We evaluate the quality of a potential replica placement decision according to two metrics. The Proximity metric $P\%$ represents the tail latency experienced by the application users. Specifically, it measures the percentage of network packets which reached their assigned replica with a latency lower than the target L. Greater Proximity values depict a better system. Every replica object φ_i holds two member variables which respectively estimate the total number of packets received by the replica ($\varphi_i.req$) and the number of received packets with a latency greater than the target L ($\varphi_i.sreq$). Using these variables we can compute the Proximity $P\%$:

$$P\% = \left[1 - \frac{\sum_{i=1}^{r} \varphi_i.sreq}{\sum_{i=1}^{r} \varphi_i.req} \right] \times 100\%$$

Likewise, the Imbalance metric $I\%$ represents the quality of load balancing between the replicas, and therefore the effectiveness of each provisioned replica. Lower Imbalance values depict a better system. We define Imbalance as the standard deviation of the workloads of individual replicas for a given application:

$$I\% = \frac{\sigma_{req}}{\sum_{i=1}^{r} \varphi_i.req} \times 100\% \qquad \text{Where:} \quad \sigma_{req} = \sqrt{\frac{1}{r} \times \sum_{i=1}^{r} (\varphi_i.req - \mu_{req})^2}$$

Our heuristics aim to optimize an objective function Θ which is a linear combination of $P\%$ and $I\%$. For each case $c_i \in \Omega$ they evaluate the objective function Θ, and eventually select the evaluated case which maximizes the function:

$$\Theta_\alpha(c_i) = \alpha \frac{c_i.P\%}{P_{max}\%} + (1-\alpha)\frac{I_{min}\%}{c_i.I\%}$$

The value α represents the desired tradeoff between Proximity and Imbalance, and $P_{max}\%$ and $I_{min}\%$ respectively represent the greatest and lowest observed values of $P\%$ and $I\%$ in the set of evaluated cases.

In our evaluations we use $\alpha = 0.95$ to favorize the reduction of tail latency over the reduction of load imbalance. Note that this function can easily be extended to integrate other optimization metrics such as financial cost and energy consumption.

5.2.2 System monitoring

To evaluate the $P\%$ and $I\%$ metrics, Hona relies on measured data about the sources of traffic addressed to different nodes. The initial replica placement problem must be solved before the application gets deployed, so it cannot rely on information related to this specific application. Instead, we rely on information from other applications, as an approximation of the future traffic of the concerned application. In the replica replacement problem the application is already deployed so we can rely on the specific traffic addressed to it.

Evaluating the two metrics requires access to three types of data: (i) cluster information with the list of nodes in the system, their available resources and the list of running pods with their hosting node; (ii) latency information between any pair of nodes in the system; and (iii) network traffic information with the volumes of traffic exchanged between every gateway node and every replica.

Cluster information is maintained by Kubernetes itself. We can access it with simple calls to its `etcd` service.

Latency information is maintained by Serf using an efficient gossiping protocol. We can obtain an accurate up-to-date estimate of the latency between any pair of worker nodes with a simple call to the `rtt` interface of Serf's agent at the master node.

Traffic information can be obtained from Proxy-mity. In Kubernetes, every Service is assigned a distinct private IP address which uniquely identifies a single application. Proxy-mity, in turn, creates in every worker node an IP-layer route between this

service's IP address and every pod which belongs to this application. Gateway nodes therefore have access to detailed information regarding the volume of traffic exchanged with every pod in the system thanks to kernel-level counters. Proxy-mity makes this information available to Hona's scheduler via a local call to its local Serf agent.

5.2.3 Initial replica placement

When deploying an application for the first time, finding the optimal placement decision for r replicas among n worker nodes requires in principle one to fully explore the set Ω of all possible placements to identify the placement which optimizes the objective function $P\%$. However, the space of possible placements is extremely large even for modest values of r and n:

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

For instance, placing 10 replicas in a 100-node system produces a space of $\frac{100!}{10!(100-10)!} = 17,310,309,456,440$ possible solutions. Exploring this space in its entirety is obviously not feasible. However, it is not strictly necessary for us to identify the exact optimal placement. In most cases it is largely sufficient to identify an approximate solution which delivers the expected quality of service to the end users. We can therefore define heuristics which explore only a small fraction of Ω and select the best placement out of the explored solutions.

To partially explore Ω, the first step is to restrict the search to the worker nodes which have sufficient available resources to host a pod. This reduces the set of nodes to $\Delta' \subset \Delta$ which contains only the suitable nodes.

We define two heuristics to explore the space of initial replica placements: a random search heuristic, and a heuristic which exploits Vivaldi's geometric model of network latencies.

Random search heuristic: This heuristic is presented in Algorithm 5.1. The *RandomCases* function first computes the load distribution per node (LPN) using the information collected from the nodes. It then initializes the set of evaluated cases with a first randomly-selected configuration, and iteratively draws additional randomly-selected configurations until a solution is found or the time quota allocated

Algorithm 5.1: Random-search initial placement heuristic.

```
   Input: Δ, Lat, QoS, Traf, t, r, L
   Output: c_sol
1  Function RandomCases(Δ, Lat, QoS, Traf, t, r, L)
2      Δ' ← GetFeasibleNodes(Δ)
3      LPN← CalculateLoadPerNode(Traf)
4      SN ← GetRandomSet(Δ',r)
5      c_i ← CaseStudy(Δ, Δ', SN, Lat, Traf, LPN, L)
6      Cases.append(c_i)
7      while Test(c_i,QoS,t) != True do
8          c_i ← CaseStudy(Nodes, Lat, Traf, SN, LPN, L)
9          Cases.append(c_i)
10     c_sol ← GetBest(Cases)
11     return c_sol
```

to the search expires. The *GetBest* function then selects the best studied configuration and the function returns.

In our experience, this heuristic provides good solutions when the Latency threshold is relatively high as many placements can fulfill this QoS requirement. A short random search identifies at least one of them with high probability. However, in more difficult cases with a lower latency threshold, the number of solutions reduces drastically and this heuristic often fails to find a suitable one. We therefore propose a second heuristic which uses Vivaldi's geometric model to drive the search toward more promising solutions.

Vivaldi-aware heuristic: Vivaldi models network latencies by assigning each node an 8-dimensional coordinate. The latency between two nodes is then approximated by the Euclidean distance between their coordinates.

Hona introduce an efficient search heuristic which exploits this simple geometric model. As shown in Algorithm 5.2, the heuristic starts by computing the load distribution per node before grouping the nodes into small groups according to their location in the Vivaldi Euclidean space.

The main idea of this heuristics is to identify groups of nearby nodes and to select a single replica among them to serve the traffic originating from all of them. The grouping of nearby nodes is done using the *CreateGroups* function which randomly selects a first node and creates a group with all nodes in its neighborhood. The size of each group is determined by the *ND* (*Nodes Density*) variable. This variable is computed as the

Algorithm 5.2: Hona's initial replica placement heuristic.

Input: Δ, Lat, r, QoS, t, Traf, tech, p, Change, L
Output: c_{sol}

```
1  Function CreateGroups(Δ, Lat, r, L, Tech, p)
2      ND ← len(Δ) / (r * p)
3      Temp ← Δ
4      while len(Temp) > 0 do
5          if len(Temp) < ND then
6              Groups.append(group(Temp, Δ, L, Tech))
7          else
8              MainNode ← Random.choice(Temp)
9              Nearby ← GetNearby(MainNode, Temp)
10             GN ← [MainNode] + Nearby
11             Temp.remove(GN)
12             Groups.append(group(GN, Δ, L, Tech))

13 Function group(GN, Δ, L, Tech)
14     group.nodes ← GN
15     if Tech == 0 then
16         group.leader ← GetLeaderRequests(GN)
17     if Tech == 1 then
18         group.leader ← GetLeaderNeighbors(GN, Δ, L)

19 Function HonaCases(Δ, Lat, r, QoS, t, Traf, Tech, p, Change, L)
20     Count = 0
21     LPN ← CalculateLoadPerNode(Traf)
22     Groups ← CreateGroups(Δ, Lat, r, L, Tech, p, Traf)
23     Leaders ← GetLeaders(Groups)
24     SN ← GetRandomSet(Leaders,n)
25     c_i ← CaseStudy(Δ, Lat, Traf, SN, LPN, L)
26     Cases.append(c_i)
27     while Test(c_i,QoS,t) != True do
28         Count++
29         c_i ← CaseStudy(Δ, Lat, Traff, SN, LPN, L)
30         Cases.append(c_i)
31         if Count%Change == 0 then
32             Groups ← CreateGroups(Δ, Lat, r, L, Tech, p)
33             Leaders ← GetLeaders(Groups)
34         SN ← GetRandomSet(Leaders,n)
35     c_sol ← GetBest(Cases)
36     return c_sol
```

fraction of total number of system nodes to the desired number of replicas, multiplied by a user-defined variable p. Larger values of p create smaller groups. The algorithm periodically re-generates new groups and group leaders, until a solution is found or the deadline is reached.

Once a group has been identified, a single node within the group is chosen as the group leader which will receive a replica while the others are excluded as potential replica locations.

We propose two possible criteria for the final selection of the group leader, which result in two variants of this heuristic:

H1 selects the node which generates the greatest number of end-user requests. This increases the number of requests that will be processed by their gateway node, with a gateway-to-replica latency of approximately 0.

H2 selects the node with the greatest number of neighbors. Neighborhood is established as an enclosure of the nodes with a latency lower than the threshold latency L. A replica placed in a node with high number of neighbors will offer a nearby replica for all its neighbors.

Similar to the random placement heuristic, this algorithm randomly chooses r group leaders to produce a replica placement which gets evaluated using function Θ. The algorithm evaluates as many such placements as possible until a solution is found or the deadline expires, and terminates by returning the best placement.

5.2.4 Replica re-placement

Online systems often observe significant variations over time of the characteristics of the traffic they receive [17]. To maintain an efficient replica placement over time, it is important to detect variations when they occur, and to update the replica placement accordingly.

Hona periodically recomputes the Proximity and Imbalance metrics with monitored data collected during the previous cycle. When these metrics deviate too much from their initial values, it triggers the *Replace* function which is in charge of updating the replica placement. To avoid oscillating behavior, and considering that re-placing a replica incurs a cost, Hona re-places at most one replica per application and per cycle.

Algorithm 5.3: Hona's replica re-placement heuristic.

```
   Input: Δ, Φ, QoS, Lat, Reason, Traf
   Output: SelectedSolution
1  Function Replace( Δ, Φ, QoS, Lat, Reason, Traff)
2      for ∀φ_i ∈ Φ do
3          Slow[φ_i ] ← CalculatePercentageSlow(Φ, Traf, Lat)
4          ReqPerPod[φ_i] ← GetRequestPerPod(Φ, Traf)
5      if Reason=="Proximity" then
6          SortedPods ← Sort(Slow)
7          PotentialNodes ← SlowSources(Traf,Φ,Lat)
8      if Reason=="Imbalance" then
9          SortedPods ← ISort(ReqPerPod)
10         PotentialNodes ← NearbyTraffic(Traff,Φ,Lat)
11     for φ_i in SortedPods do
12         for δ_i in PotentialNodes do
13             SN ← Nodes(Φ) - Node(φ_i) + δ_i
14             c_i ← CaseStudy(Δ, Lat,Traff,SN,LPN,AL)
15             Cases.append(c_i)
16             if c_i is a solution then
17                 solutions.append(c_i)
18                 found ← True
19         if found==True then
20             Return GetBest(solutions)
21     if Solutions==NULL then
22         Return GetBest(Cases)
```

Algorithm 5.3 presents the re-placement heuristic. It first sorts the application replicas to identify the least useful ones according to the current conditions, and then tries to find them a better location out of a filtered set of nodes.

The identification of the least useful replica depends on the nature of the performance violation. If the Proximity metric has degraded significantly, then the heuristic will attempt to re-place one of the replicas with the greatest observed tail latency. On the other hand, if the re-placement is triggered by an increase of the Imbalance metric, the heuristic will select one of the replicas which process the lowest amount of load.

Likewise, the set of potential nodes available to host the pod is selected according to the violation type. If the violation was caused by a lack of proximity, the potential nodes will consist of the gateway nodes that are suffering from high tail latency. On the

other hand, if the violation was caused by load imbalance, the potential nodes are those located close to the main sources of traffic.

The replacement function then iterates through the list of least useful replicas, and tries to find a better node to hold them. It stops as soon as it finds a suitable solution which improves Θ by at least some pre-defined value. In case no improvement can be obtained by re-placing one replica, the system keeps the current placement unmodified. A potential solution in this case would be to increase the number of replicas. We leave this topic for future work.

5.2.5 Implementation

We implemented the replica placement and re-placement algorithms in Kubernetes by changing the way it schedules application pods on the available resources. This can be done without changing the scheduler's implementation by exploiting the standard *NodeSelector* functionality which allows entities to define constraints regarding the placement of pods onto nodes.

Hona assigns labels to server nodes and to application deployments such that the application's pods can only be placed by the Kubernetes scheduler on the set of nodes which was chosen by the replica placement algorithm. Hona adds an anti-affinity rule which forces the replicas to be located in different nodes, resulting in Kubernetes necessarily placing replicas in the chosen locations.

When executing node re-placement, Hona must not only identify the node where a new replica will be started. To avoid creating unnecessary downtimes, it must also make sure to delete the previous replica only after the new one has been started. This procedure is illustrated in Figure 5.1. In the initial state, the application has two replicas running in δ_1 and δ_2. The replica re-placement process starts when Hona detects a QoS violation. It then chooses a solution: the pod in Node 2 must be replaced with another in δ_3. The first step is labeling δ_3, then requesting the creation of an additional pod by scaling up the deployment. Once the new pod has been started, Hona removes the label at δ_2, orders the deletion of the pod located at this node, and scales the deployment down to its initial number of replicas. This changes the replica location without incurring a temporary down time during which the application would have one less replica to handle the request workload.

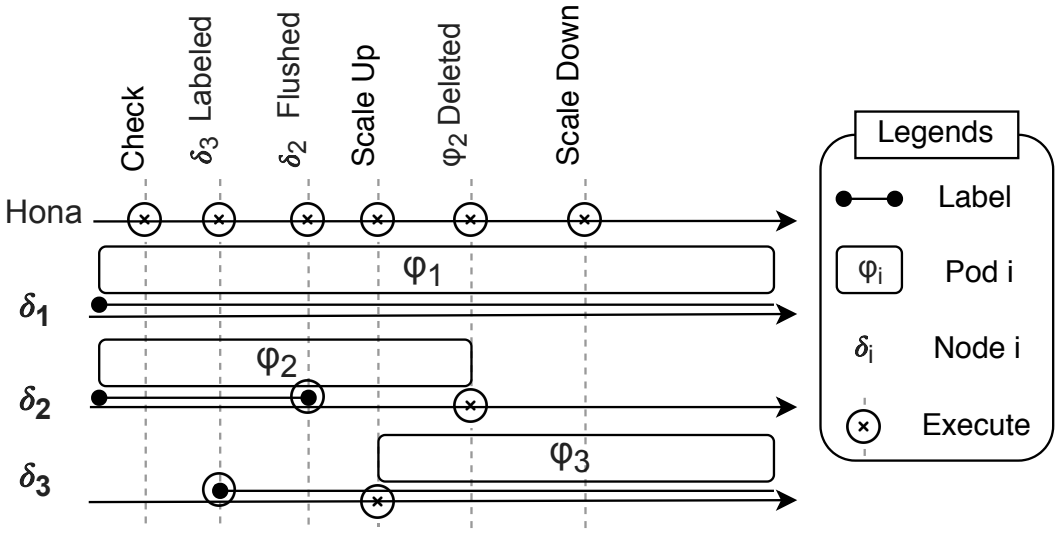


Figure 5.1 – Execution of a node re-placement operation.

5.3 Evaluation

We evaluate this work using a combination of experimental measurements and simulations. The experimental setup consists of 22 RPis model 3B+ single-board computers acting as fog computing servers. Such machines are frequently used to prototype fog computing infrastructures [92]. They run the HypriotOS v1.9.0 distribution with Linux kernel 4.14.34, Docker v18.04.0 and Kubernetes v1.9.3. We implemented Hona on top of Serf v0.8.2.dev and the development version of Proxy-mity.

As shown in Figure 5.2, Hona is implemented as a daemon running in the Kubernetes master node. It fetches information from Kubernetes and Serf, and expresses its placement decisions by attaching labels to the concerned nodes.

In our cluster, one RPi runs the Kubernetes master and the Hona scheduler, while the remaining RPIs act as worker nodes capable of hosting replicas. Every worker node is also a WiFi hotspot and a Kubernetes gateway so end users can connect to nearby worker nodes and send requests to the service.

Similar to the evaluation of Proxy-mity, we emulate realistic network latencies between the worker nodes using the Linux `tc`[3] command. We specifically use latency values measured between European cities[4]. In the evaluation of Proxy-mity the

3. `https://linux.die.net/man/8/tc`
4. `https://wondernetwork.com/`

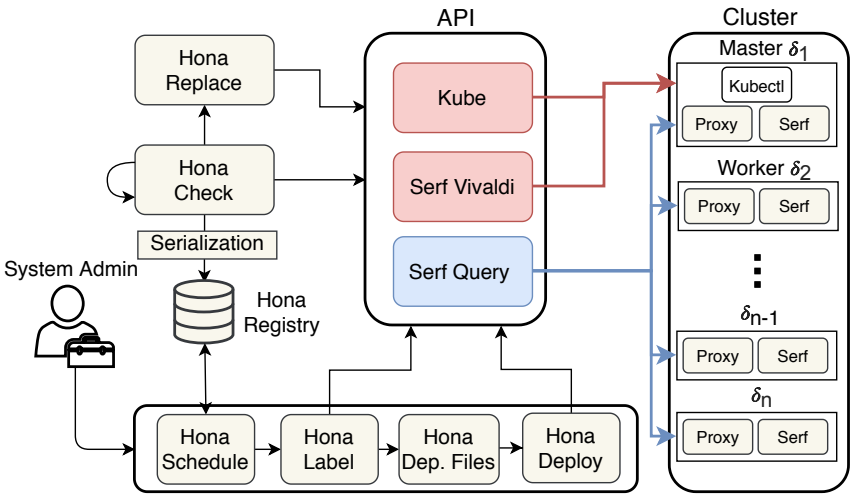


Figure 5.2 – Hona's architecture.

testbed consisted of 12 nodes where each node emulated one of the cities shown in Table 4.1. In Hona, the tested was extended to 22 nodes and the cities that are emulated are depicted in Figure 5.3. Network latencies range from 3 ms to 80 ms with an average latency of $\approx$ 28 ms and arguably represent a typical situation for a geo-distributed fog computing infrastructure.

The application is a web server which simply returns the IP address of the serving pod. We generate workloads either by equally distributing traffic among all gateway nodes, or by selecting specific gateways as the only sources of traffic. The threshold latency is $L = 28\,ms$ (the median inter-node latency in our system), the trade-off between Proximity and Imbalance is $\alpha = 0.95$, and the deadline to find a placement is $10\,s$.

We perform the scalability analysis using a simulator which randomly creates up to 500 virtual nodes in the Vivaldi Euclidean space, and use the same heuristics implementation as in Hona to select replica placements.

5.3.1 Initial replica placement

We first evaluate Hona's initial placement algorithms and compare them with the unmodified Kubernetes scheduler and the optimal solution found using a brute-force approach. In the following graphs, each algorithm is denoted by a letter: **O** for the optimal solution found using brute-force search, **R** for the random heuristic, **H1** and **H2** for the first and second versions of Hona heuristic.

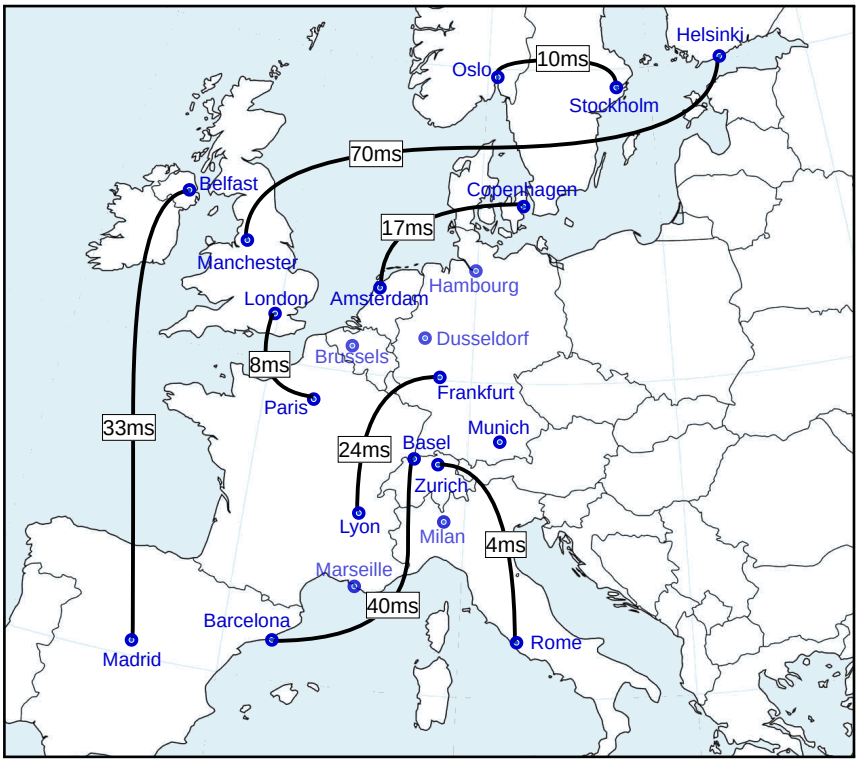


Figure 5.3 – Selected European cities and some examples of network latencies between them.

Overall performance (testbed experiments): Figure 5.4 compares the Proximity and Imbalance of solutions found by the different algorithms for various numbers of replicas within the 21 worker nodes in the testbed. We run each experiment 100 times, and evaluate 200 configurations per experiment.

Increasing the number of replicas to be placed makes the search easier, and it delivers better results. More replicas can better cover the different regions of the system, and the probability for any node to have a replica nearby increases. Similarly, increasing the number of replicas makes load balancing easier.

The three Hona heuristics perform well in this case with results very close to the brute-force optimal in a fraction of the time (for $r = 9$, O required ≈ 48 minutes compared to 0.55 seconds for the heuristics). We however notice that in the relatively difficult case of $r = 3$ the H2 heuristic outperforms the others according to both metrics since it was designed to find solutions when the number of replicas is relatively very small compared to the number of available nodes. This advantage becomes more evident when testing over a large scale cluster.

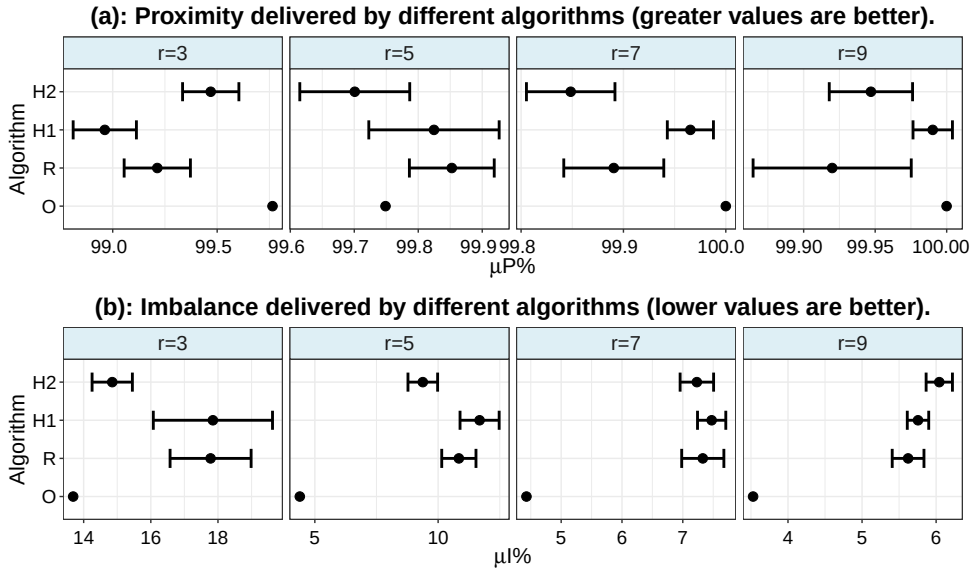


Figure 5.4 – Initial replica placement analysis (testbed, $n = 21$).

To better understand the differences between the Random and the Hona heuristics, Figure 5.5 depicts the 5th/25th/50th/75th/95th percentiles of *all the tested placements* during the same experiment. In contrast, Figure 5.4 shows only the best solutions found by every run of the heuristics. We can clearly see the differences between heuristics; the Random heuristic evaluates placement options across a wide range of quality, whereas the H1 and H2 heuristics better focus their search on promising placement options.

Effect of system size (simulator evaluations): We now explore Hona's placement algorithms in systems up to 300 nodes. Figure 5.6 depicts the results obtained from 1000 runs of every evaluation. We chose the latencies between nodes by randomly selecting Vivaldi coordinates for every node within a distance of at most $80\,ms$ between nodes. To make the placement problem equally difficult with different system sizes, we also scaled the number of requested replicas accordingly: $r = n/10$. The red lines indicate the target values. We do not plot the brute-force optimal placements which would require extremely long executions.

In Figures 5.6-a and 5.6-b, we observe greater differences between the three Hona heuristics with larger system sizes. In particular, the H2 heuristic delivers better Proximity for large-scale systems. This is due to the fact that it selects group leaders with respect to the number of neighbors they can serve with low latency.

The H1 and H2 heuristics also outperform the Random heuristic in the number of cases they need to evaluate before finding a solution which meets the user's

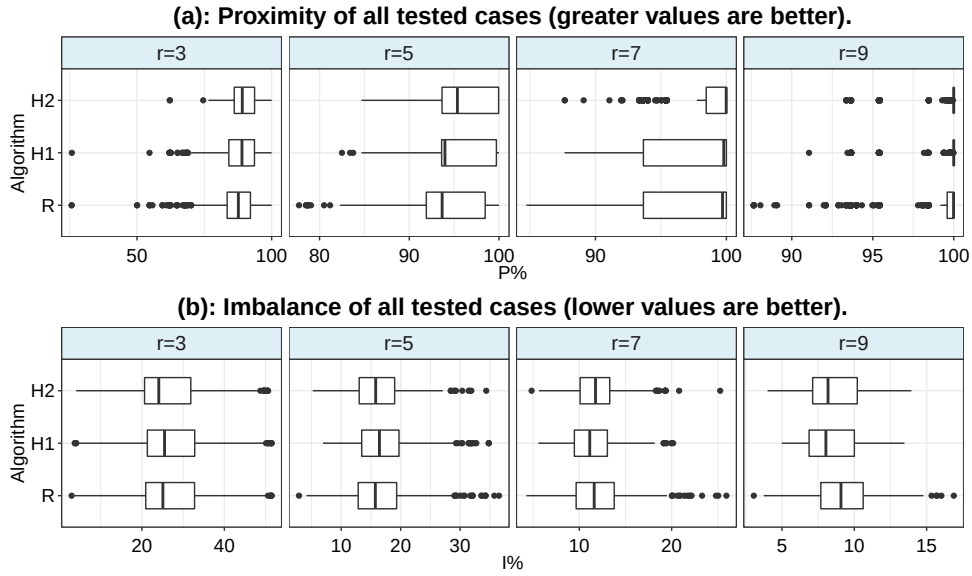


Figure 5.5 – Individual test cases analysis (testbed, $n = 21$).

requirements (Figure 5.6-c). We observe that H2 finds solutions much quicker than the other heuristics.

Finally, Figure 5.6-d shows the number of heuristic executions which reached the timeout without finding a suitable solution. Here as well, the H2 heuristic significantly outperforms the others because it targets its search to cases which have a greater probability of delivering high-quality results.

We conclude that the H2 heuristic delivers better-quality results than the others, in less time, and with a lower probability of a failed search. In the rest of this chapter we therefore use this heuristic for the initial replica placements.

5.3.2 Replica re-placement

After the initial deployment of an application, Hona monitors the network traffic it handles and periodically recomputes its performance metrics $P\%$ and $I\%$. When these metrics deviate too much from their expected values, it tries to re-place replicas within the system to address the new situation.

We evaluate the behavior of Hona in our 22-nodes testbed with a variety of scenarios. We define the Proximity target as $P\% = 99.5\%$ of requests with a latency under $L = 28\,ms$, with a tolerance of 0.5% before triggering re-placement. Similarly,

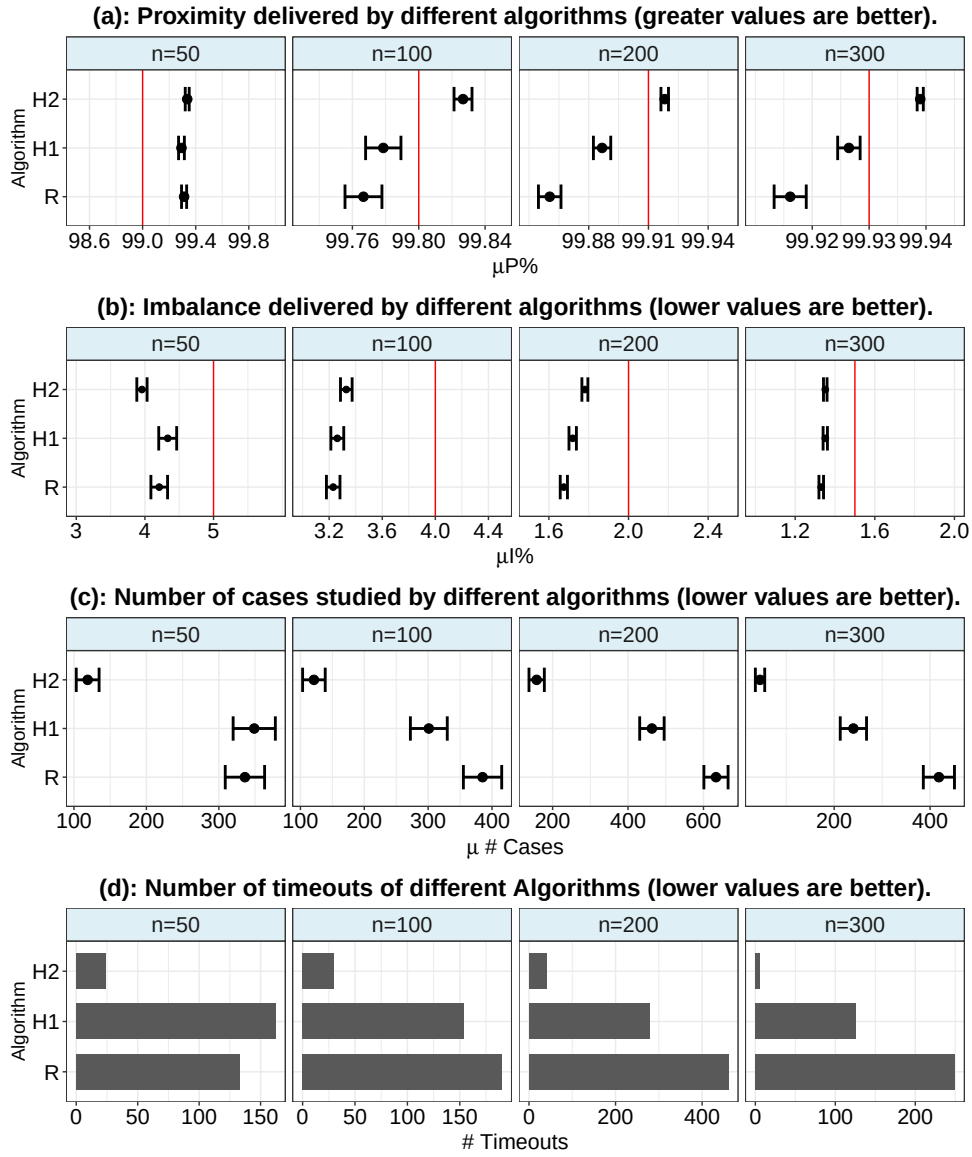


Figure 5.6 – Initial replica placement with various system sizes (simulator, $r = n/10$).

the Imbalance target is $I\% = 5\%$, with a tolerance of 1% before re-placement. These metrics are evaluated at a periodicity of $30\,s$.

Figure 5.7 depicts increasingly difficult re-placement scenarios. We plot the Proximity and Imbalance metrics as calculated at the end of every cycle. The red area depicts the period during which the new situation is introduced, and the vertical red line(s) represents the time(s) at which the re-placement algorithm actually changes the placement of replicas. We do not plot the $P\%$ and $I\%$ metrics in the cycle

immediately after a re-placement: these metrics capture the transient state during which a new replica is created while another one is deleted, and therefore do not represent accurate information.

(*a*) *Changing a source of traffic:* Figure 5.7-a shows a case where one source of traffic gets replaced with another one. During the first five cycles, no load is issued to the studied application so the Imbalance metric remains at $I\% = 0$. Proximity is calculated according to the background traffic of other applications, which explains its initial value of 90%. Some load is then generated starting from cycle 6. The two metrics reach very good values: almost 100% for $P\%$, and about 2% for $I\%$. At cycle 9, however, we replace one of the main sources of traffic with another one located far away from any current replica. This event is detected quickly and, at cycle 11, the system moves the useless replica close to the next source of traffic, which effectively repairs the Proximity degradation.

(*b*) *Adding a new source of traffic:* Figure 5.7-b shows a scenario where a new source of traffic is added far away from the current set of replicas. This results in a Proximity violation which is quickly detected by the system. However, in this situation there is no solution that would bring both metrics within their expected bounds. Since we favorized Proximity over Imbalance in the objective function Θ, the system moves one replica close to the new source of traffic, which fixes the Proximity violation at the expense of a degraded imbalance. The only solution in this case to solve both QoS violations is scaling up the replica set, as we discuss in Chapter 6

(*c*) *Changing a route latency:* Figure 5.7-c shows the case where the load distribution remains unmodified, but the latency between a gateway node and its closest replica changes suddenly from $10\,ms$ to $50\,ms$. In this case, Serf must first detect the change of network latencies before Hona can react and re-place the concerned replica accordingly. We see in the figure that these two operations take place quickly. One cycle after the latency change, Hona triggers a re-placement operation which brings performance back to normal.

(*d*) *Complete replacement of the sources of traffic:* Figure 5.7-d depicts a dramatic situation where the entire workload changes at once: in cycle 11 we stop all the sources of traffic, and replace them with entirely different ones. In this case, the replica re-placement takes place in two steps. A first re-placement is triggered at cycle 14: this operation improves Proximity but at the expense of an increase in the load

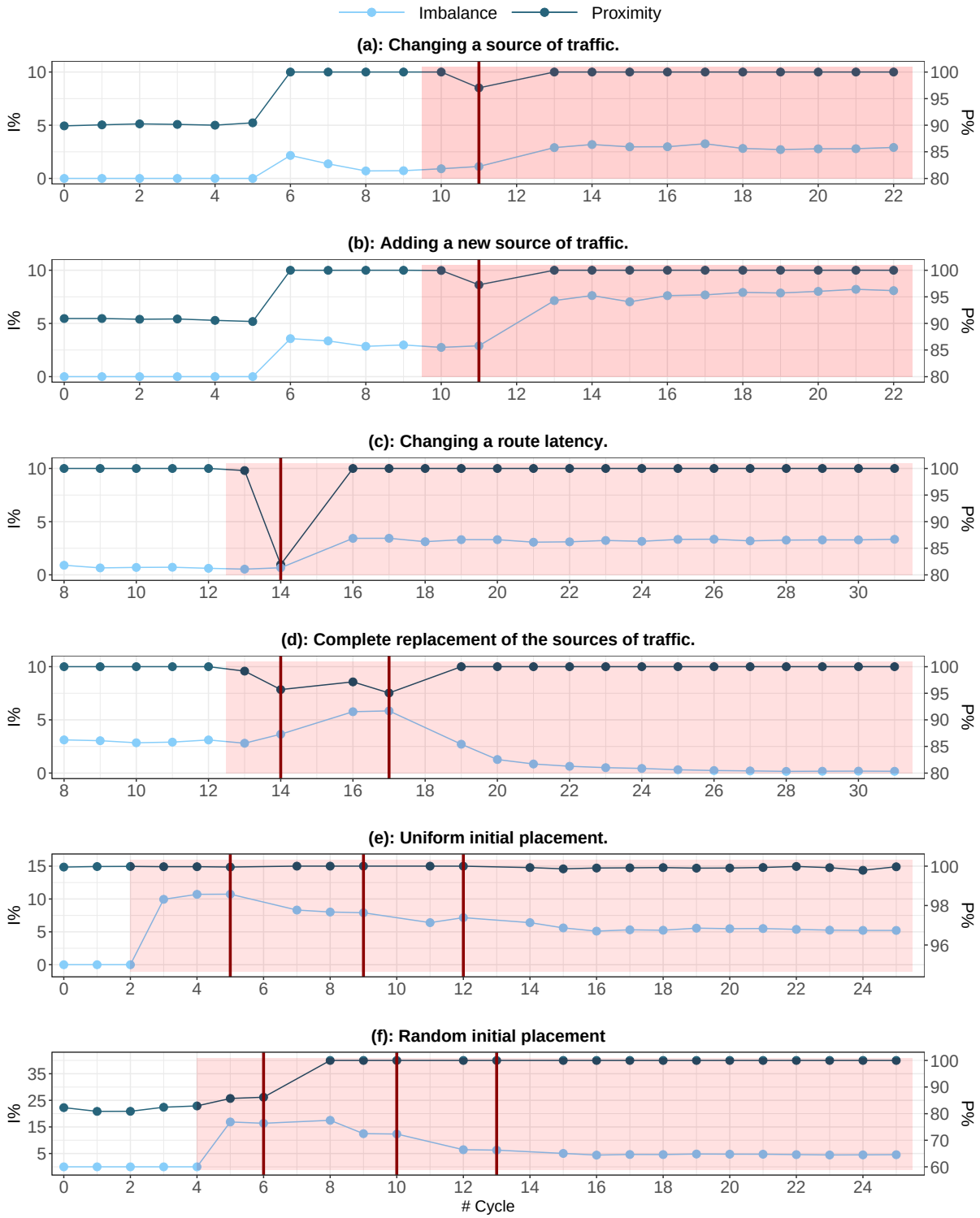


Figure 5.7 – Replica re-placement analysis (testbed, $n = 21$).

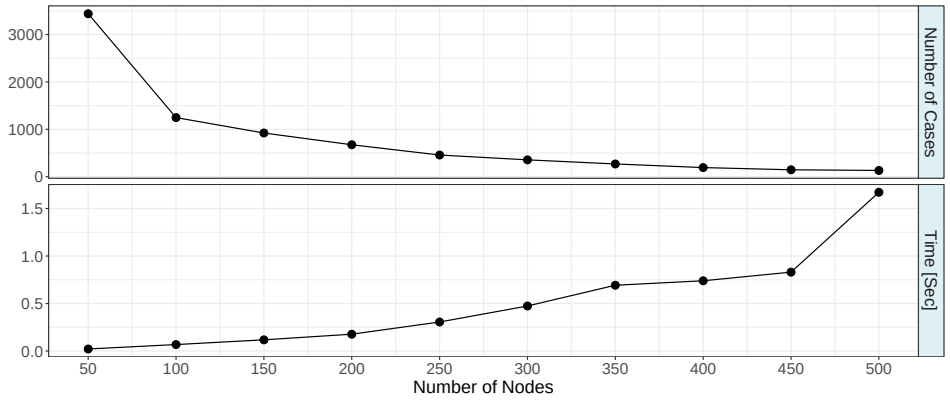


Figure 5.8 – Complexity of the H2 heuristic (simulator).

Imbalance. At cycle 17 a second re-placement is triggered which brings both metrics back within their expected values.

(*e*) *Starting from a uniform replica placement:* Figure 5.7-e shows a difficult situation created by a sub-optimal initial replica placement. We initially placed replicas with no information whatsoever about the future workload. In this case replicas get placed uniformly across the system. The Proximity is not affected thanks to the uniform distribution of replicas. On the other hand, once actual traffic is produced, an important Imbalance is detected. The system repairs it (without significantly affecting Proximity) in three re-placement operations.

(*f*) *Starting from a random replica placement:* Figure 5.7-f shows a case where the initial replica placement was chosen randomly. When traffic starts in cycle 4, both metrics are far from their expected values. The desired performance is obtained after three re-placement operations.

Hona addresses a wide variety of QoS violations, and provides effective solutions to solve them. In our experiments we never observed oscillating behavior in which the system would not very quickly reach a new stable state.

5.3.3 Computational complexity

Figure 5.8 shows the computation time of the H2 heuristic for placing 10 replicas with QoS bounds of $P\% = 99.5\%$, $L = 25\,ms$ and $I\% = 4\%$. We used a mid-range machine with a quad-core Intel Core i7-7600U CPU @2.80GHz. The current implementation is single-threaded, but parallelizing it should in principle be easy as different placements can be evaluated independently from each other.

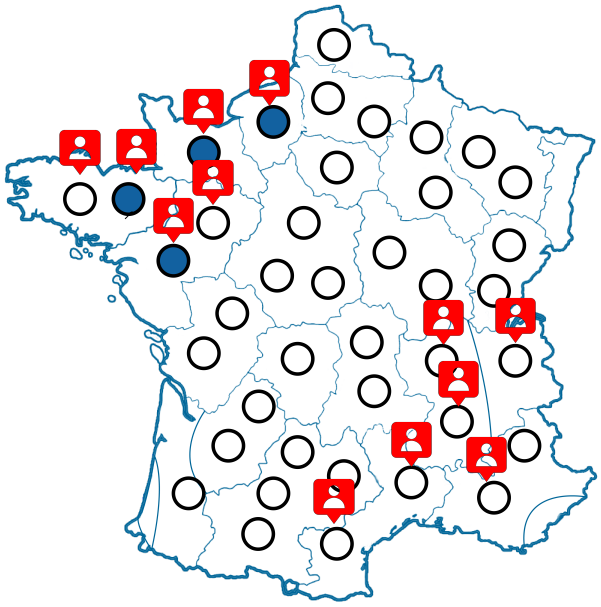
The top part of the figure depicts the number of cases which can be evaluated within $10\,s$. Clearly, the complexity of evaluating any single case increases with system size as the metric evaluation function needs to iterate through a greater number of potential traffic sources. However, as shown in the bottom part of the figure, even for large system sizes, the computation time until a satisfactory solution is found remains under $2\,s$ of computation. This comes from the fact that, with larger system sizes, the number of acceptable solutions grows as well, and a solution can be found with a lower number of evaluated cases.

5.4 Conclusion

Replica placement is an important problem in fog computing infrastructures where one can place computation close to the end-user devices. When many sources can generate traffic it is often not affordable to deploy an application replica close to every traffic source individually. One rather needs to limit the number of replicas, and to choose their location carefully to control the tail latency and the system's load balance. Replica placement decisions must also be updated every time a significant change in the operating conditions degrades the QoS metrics. We have shown that, despite the huge computational complexity of searching for the optimal solution, simple and effective heuristics can identify sufficiently good solutions in reasonable time. We have implemented Hona in Kubernetes, thereby bringing it one step closer to becoming one of the mainstream, general-purpose platforms for future fog computing scenarios.

Chapter 6

Tail-latency-aware autoscaling



In this chapter, 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. The 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.

6.1 Introduction

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 additional replicas to serve workloads originating from major sources of traffic.

Fog computing resources are precious in a multi-tenant environment, so fog applications should carefully adjust their deployments to 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 [17], 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. Similarly to the previous chapter, 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 [72]. 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.

This chapter is organized as follows. Section 6.2 first presents the Voilà system, then Section 6.3 evaluates it. Finally, Section 6.4 concludes.

6.2 System Design

Voilà[1] dynamically scales and places an application's replicas in a cluster of geo-distributed fog nodes to 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.

To realize a concrete implementation of this idea, we address the following questions in turn:

(A) What are the variables for evaluating the cluster and application status and how we obtain them?(§ 6.2.1);

(B) How to evaluate a placement? (§ 6.2.2);

(C) How to place a new application? (§ 6.2.3);

(D) What are the violations and how to act upon them?(§ 6.2.4);

(E) When and how to scale down the replica size?(§ 6.2.4).

6.2.1 System model and monitoring

Table 6.1 summarizes the main variables used to describe Voilà's 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.

1. `https://gitlab.inria.fr/afahs/voila_code`

Table 6.1 – Voilà 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 $g_i \to \varphi_j$ 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.

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

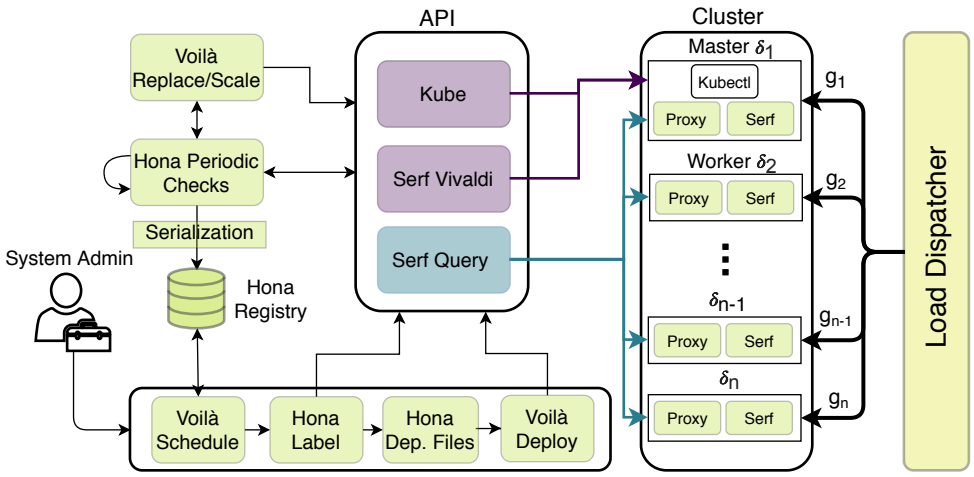


Figure 6.1 – Voilà system architecture

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.

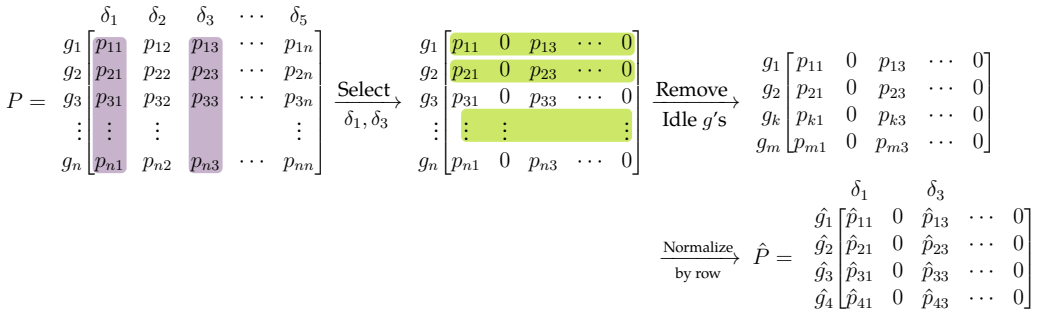
Figure 6.1 shows the system architecture of Voilà. Hona periodic checks is used by Voilà to collect all the cluster, latency, and traffic information by contacting serf and Kubernetes etcd. These information are saved in Hona registry and can be accessed by Voilà when needed.

6.2.2 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

Procedure 6.1: Probability matrix



once per placement cycle, and then exploits it to evaluate the quality of any replica placement.

Procedure 6.1 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:

$$V_{l_o}(\hat{G}, \hat{P}, L) = \sum_{i=1}^{|\hat{G}|} \sum_{j=1}^{n} \hat{p}_{ij} \times \hat{g}_i.load \times f_1(l_{ij})$$

$$\text{where } f_1(l_{ij}) = \begin{cases} 1 & \text{if } l_{ij} > l_o \\ 0 & \text{else} \end{cases}$$

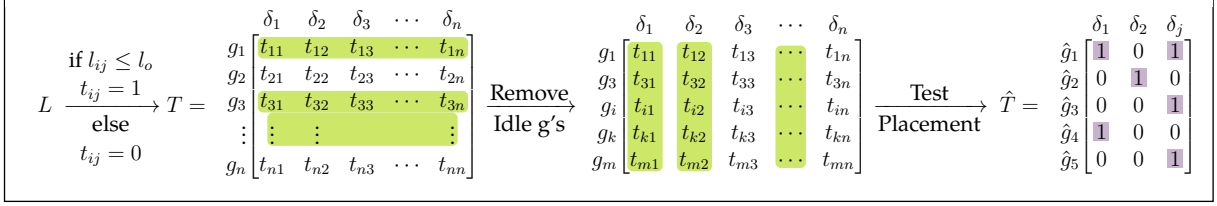
Similarly, function V_{c_o} counts the requests which would be routed to an overloaded replica φ_j hosted at δ_k:

$$V_{c_o}(\Phi, \hat{P}) = \sum_{j=1}^{r} f_2(\varphi_j) \qquad \varphi_j.load = \sum_{i=1}^{|\hat{G}|} \hat{g}_i.load \times \hat{p}_{ik}$$

$$\text{where } f_2(\varphi_j) = \begin{cases} \varphi_j.load - c_o \times \tau & \text{if } \varphi_j.load > c_o \times \tau \\ 0 & \text{else} \end{cases}$$

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

Procedure 6.2: `Test matrix`



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

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

6.2.3 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.

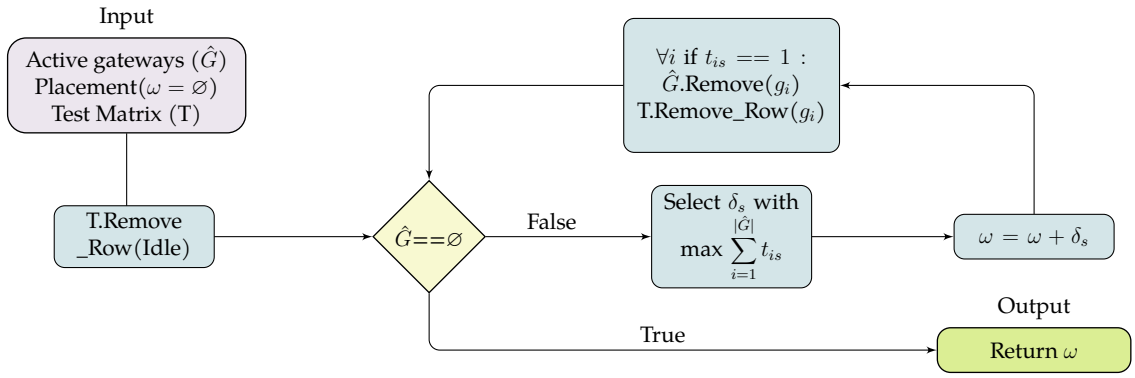
Finding the optimal placement of r replicas among n worker nodes requires in principle one to fully explore the set of all possible solutions Ω. However, this set is extremely large even for a modest value of n and r:

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

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 6.2 takes the latency matrix L as an input, and labels all the possible routes as suitable (with value 1 if $l_{ij} \leq l_o$) 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 $\hat{t}_{ij} = 1$.

Procedure 6.3: `Initial replica placement`



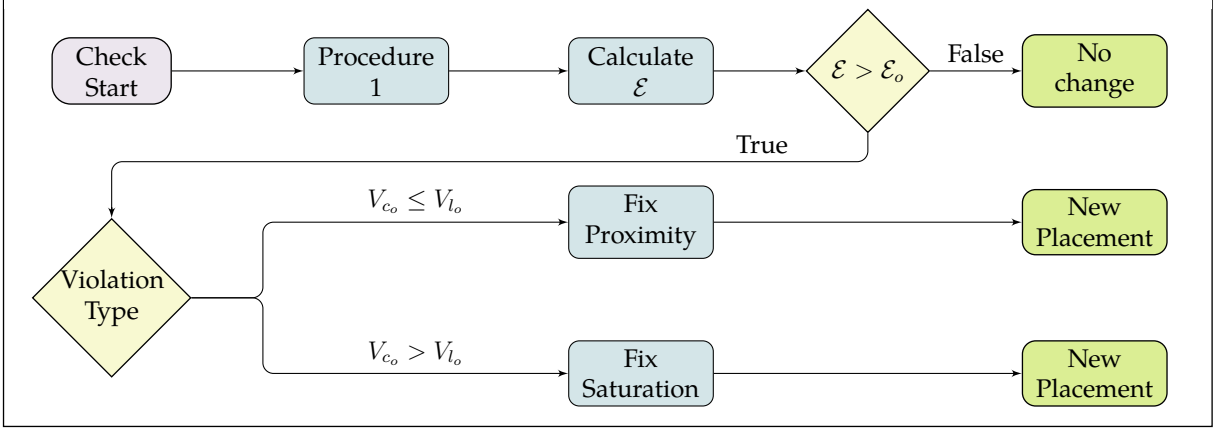
The algorithm to identify a suitable replica placement is illustrated in Procedure 6.3. Starting from an empty placement ω, the set of active gateways $\hat{G}$, and the Test Matrix T, the algorithm starts by removing the idle gateways from T, then iteratively adds new replicas until all active gateways 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.

6.2.4 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.

Procedure 6.4: QoS check algorithm



Checking for potential violations

Voilà periodically checks whether the QoS constraints are still respected. As shown in Procedure 6.4, 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.

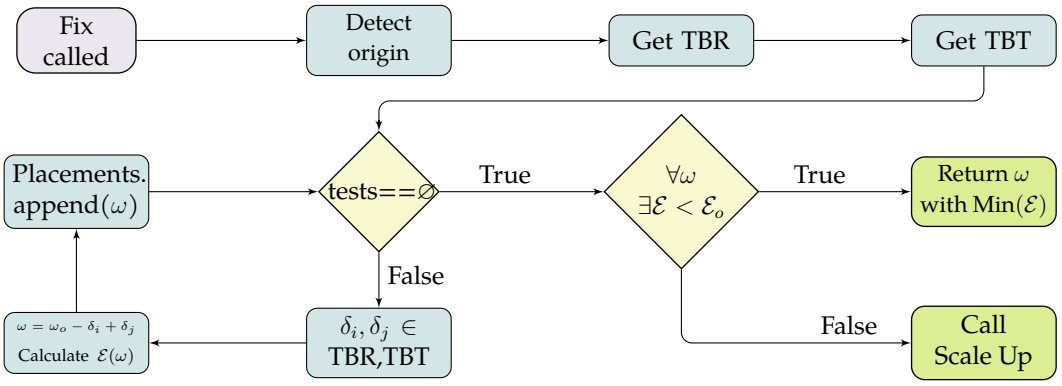
Replica replacement

When a QoS violation is detected, Voilà first tries to fix it by moving a replica from one server node to another. Procedure 6.5 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.

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

Procedure 6.5: Replica replacement algorithm



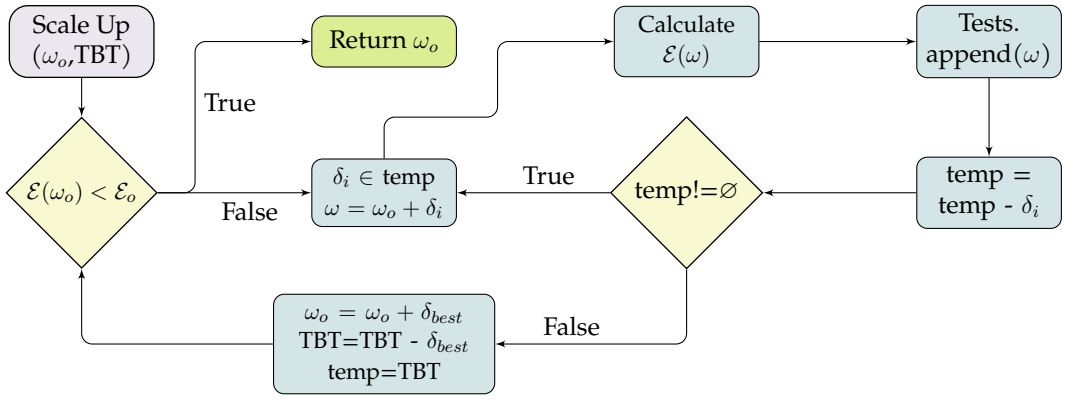
In (1) we see an example where the test matrix $\hat{T}$ indicates that gateway $\hat{g}_3$ does not have any nearby replica, which is the source of the QoS violation. Gateway $\hat{g}_4$ has only one nearby replica at δ_2, so δ_2 is not included in TBR. TBR finally contains δ_1 and δ_3 . In (2) we see 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 6.5 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.

The fix functions limit the space of possible placement by detecting the origin of the violations and by testing placement modifications that target these origins. Both functions execute similar strategies to find a corrective replacement explained in Procedure 6.5, the procedure starts by detecting the origins of the violation, in the case of proximity violation, the origins are gateways that lack a nearby replica, meanwhile for saturation they are overloaded pods. In the next step, Voilà limits the search space to a set of To-Be-Replaced (TBR) and To-Be-Tested (TBT) host nodes, a number of

Procedure 6.6: `Scale up`



tests is created by drawing a node from TBR and replacing it with one from TBT. The Procedure executes a loop to perform all the tests and finally, the best solution will be returned if it respects $\mathcal{E}_o$, otherwise Voilà declares that the violation can't be revised without scaling up the size r and as a result `Autoscale()` is called.

The difference between the fix functions lies in TBR and TBT, for proximity TBR is the serving nodes that are considered non-vital. A gateway that depends exclusively on one host node makes it vital. If this node was removed then we are certain that this gateway will suffer form low latency. on the other hand, TBT are the non-host nodes that can provide the origins of the violation with a nearby replica.

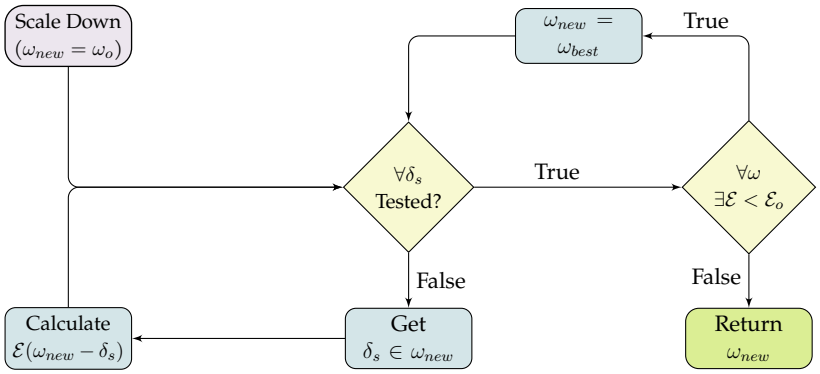
For saturation the search space is determined in a different manner, TBR includes all nodes that are receiving a small volume of load, each node in TBR will be associated with a set TBT of nodes located near the overloaded pods and that will ensure no gateway will be deprived of a vital node.

Scaling up

To choose the node where an additional replica should be created, Procedure 6.6 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.

The replacement approach helps the system maintaining a small number of replicas before moving toward an autoscaling approach. However, a surge in users'

Procedure 6.7: Scale Down



requests and emerging gateways can restrict the system to spanning new replicas. Voilà handles autoscaling as a continuation of the already studied violation (Procedure 6.6). The current placement ω_o alongside the set of promising nodes TBT are passed to `Autoscale()` such as a node $\delta_i \in$ TBT is appended to the ω_o. When all the promising placement are tested, the placement of the best value of $\mathcal{E}$ is selected as the best placement and the replica size will increase by 1. However, increasing the size by 1 is not always sufficient to meet the $\mathcal{E}_o$ constraint, the size will keep going up until the constraint are met.

The replacement and autoscaling algorithms are programmed in a way that maintain a minimal replica size by effectively exploring the search space. As equally important, the algorithms induce minimal changes on the placement since such change incur a high cost.

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 6.7 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.

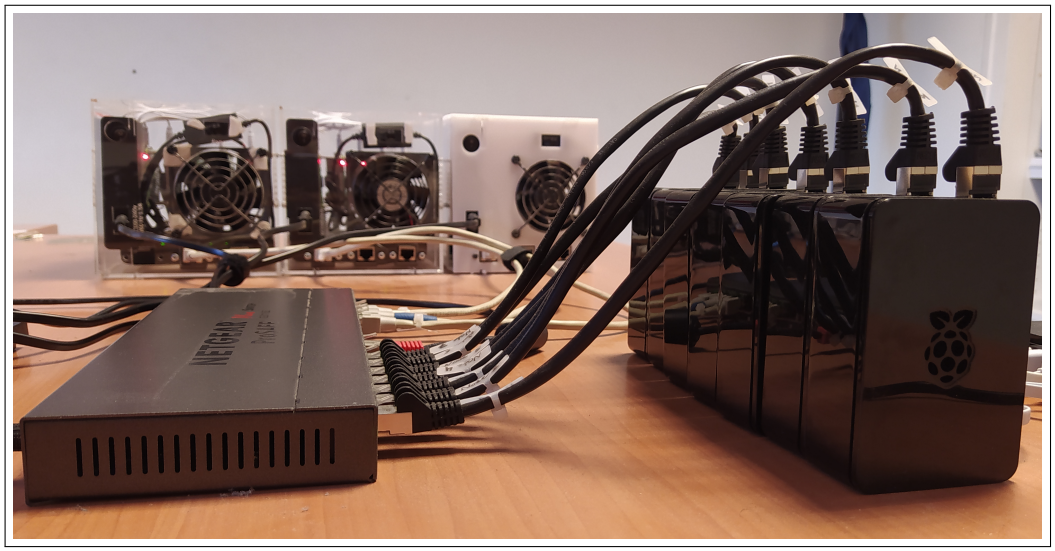


Figure 6.2 – A photo of the testbed.

6.3 Evaluation

6.3.1 Experimental setup

We evaluate Voilà with both experimental measurements and simulations. The experimental setup consists of 22 RPis model 3B+ single-board computers acting as fog computing servers. 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 (see Figure 6.2). 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 [81]. Every request is tagged with a location at 1 km granularity of the base station it was addressed to. We randomly select 22 1km^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.

6.3.2 Hona performance compared to Voilà

Voilà redefines Hona's placement algorithms mainly because the evaluation metrics in Hona are slightly different than the metrics in Voilà. Both algorithms try to reduce the network tail latency by finding a placement that offers nearby replicas to all of the detected sources of traffic. However, Hona tries to optimize the load imbalance between the replicas defined as the standard deviation of each replica's number of requests. This metric was used since Hona has no notion of replica capacity. As a result, it tries to minimize the variation between the fixed number of replicas.

On the other hand, Voilà improves on Hona by defining a metric that directly calculates the replica saturation using the maximum number of requests that can be handled by a replica during a specific time period. Voilà has the capability of increasing the number of replicas to avoid saturation, whereas Hona optimizes the proximity and try to solve the saturation violations using a best-effort approach. This was evident in Figure 5.7-(b) where Hona fixed the proximity violation but was not able to scale up the replica set to fix a possible saturation violation.

Another reason for creating new placement algorithms from scratch lies in their compatibility with the autoscaling algorithms. In Voilà, we considered the autoscaling approach as a continuation of an already-studied violation in the Voilà's re-placement. This, however, would not be possible using Hona's algorithms.

The Voilà objective function was optimized to be processed using the matrices $\hat{P}$ and $\hat{T}$ which calculate the placement and autoscaling solutions faster by calculating general P and T matrices and then conducting a case-specific $\hat{P}$ and $\hat{T}$. Also, the non-necessary nodes and idle gateways are removed from the objective function calculation, which leads to much faster processing.

We illustrate the speedup obtained by replacing Hona's algorithms with Voilà's in Figure 6.3, where we compared the average and standard deviation of the time needed to calculate the objective function of Voilà compared to Hona using the

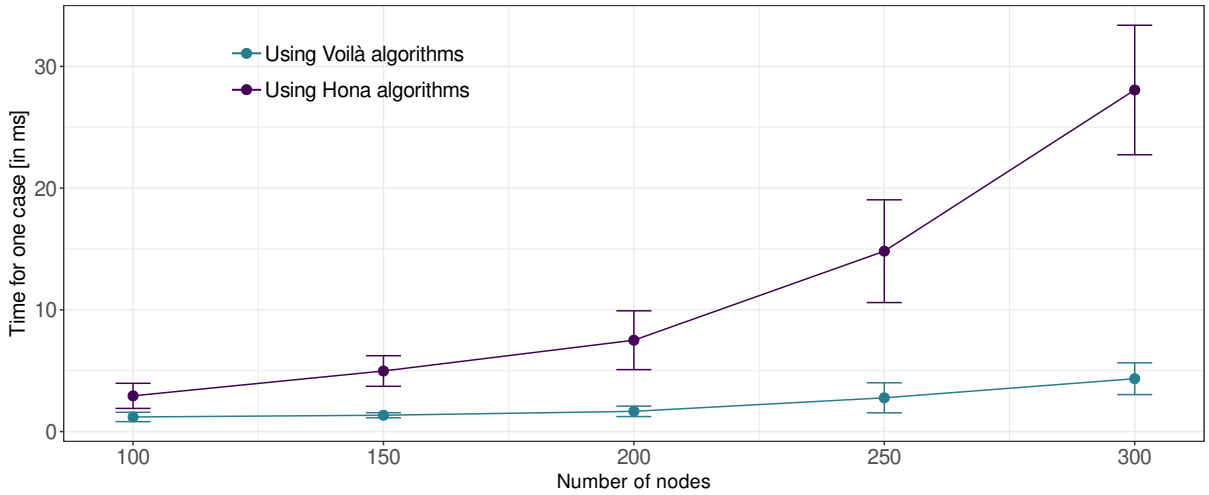


Figure 6.3 – Time needed to compute the objective function for both Hona and Voilà as a function of the number of nodes.

simulator over the same network and cluster conditions. Voilà can clearly evaluate the objective function much faster than Hona (6 times faster for 300 nodes). The average time for Hona follows a more aggressive slope as the number of nodes increases. Similarly, the standard deviation for Hona increases at a faster rate than that of Voilà, which means that Voilà is more consistent in the objective function calculation.

6.3.3 Autoscaling behavior

We first evaluate Voilà on the testbed based on parameters shown in Table 6.2. Figure 6.4 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.

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.

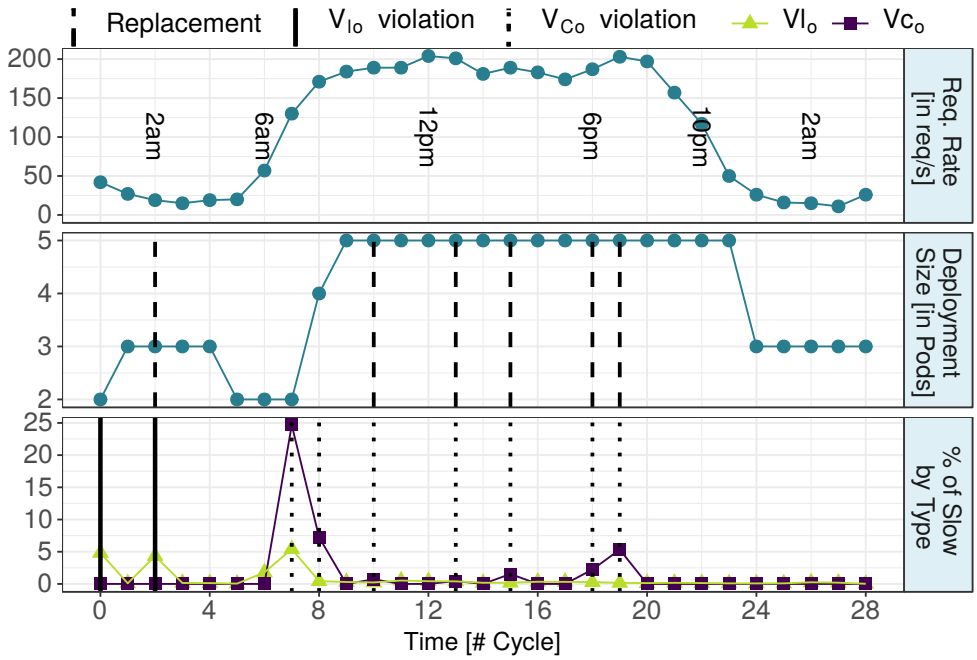


Figure 6.4 – Autoscaling over a 28-hour workload trace (testbed experiment).

Table 6.2 – Testbed evaluation parameters.

Variable	Value
l_o	15 ms
c_o	50 req/pod/s
$\mathcal{E}_o$	0.5%

Variable	Value
n	22 nodes
$\lvert\hat{G}\rvert$	18 nodes
Cycle duration τ	120 s

From 10am until 8pm the global load stabilizes close to its daily peak. However, as discussed in Section 2.4.3 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.

6.3.4 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

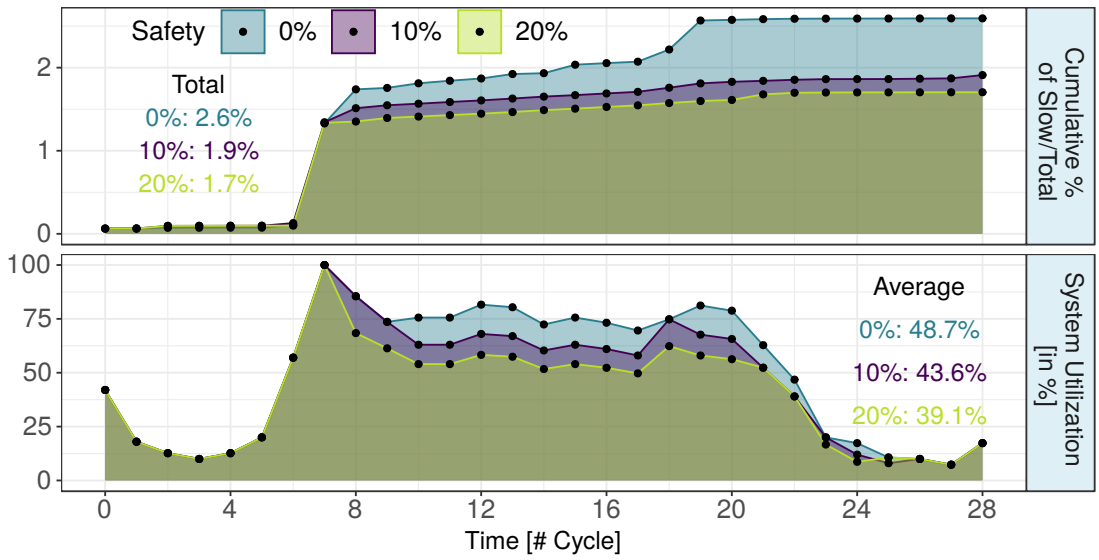


Figure 6.5 – Triggering scale-up early (testbed experiment).

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.

Figure 6.5 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.

6.3.5 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 $safety = 20\%$.

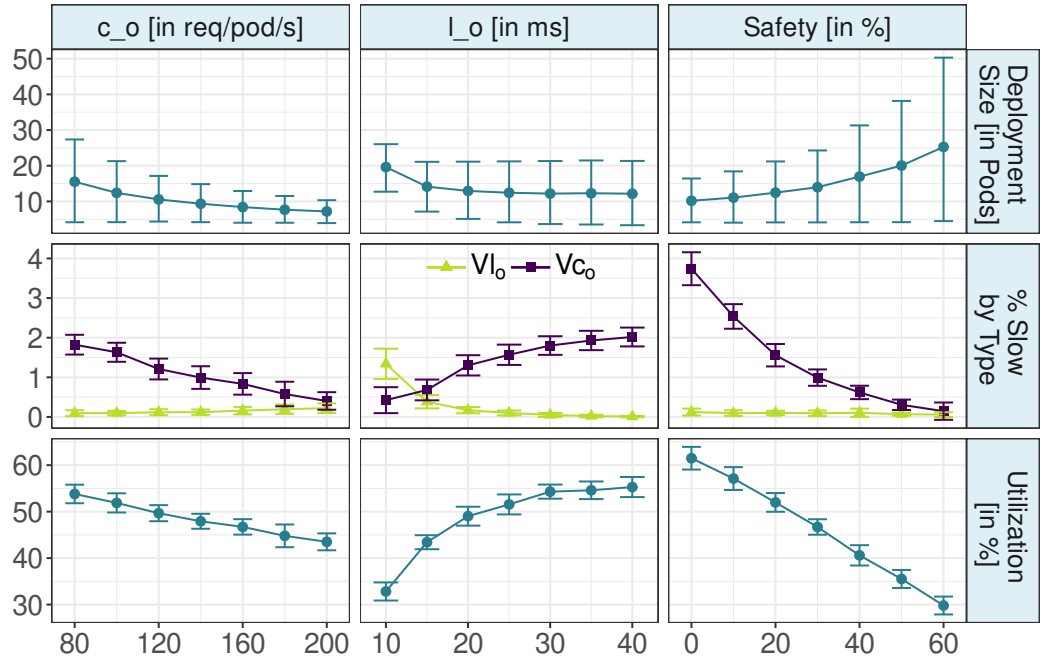


Figure 6.6 – Sensitivity analysis (simulator).

Figure 6.6 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.

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.

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

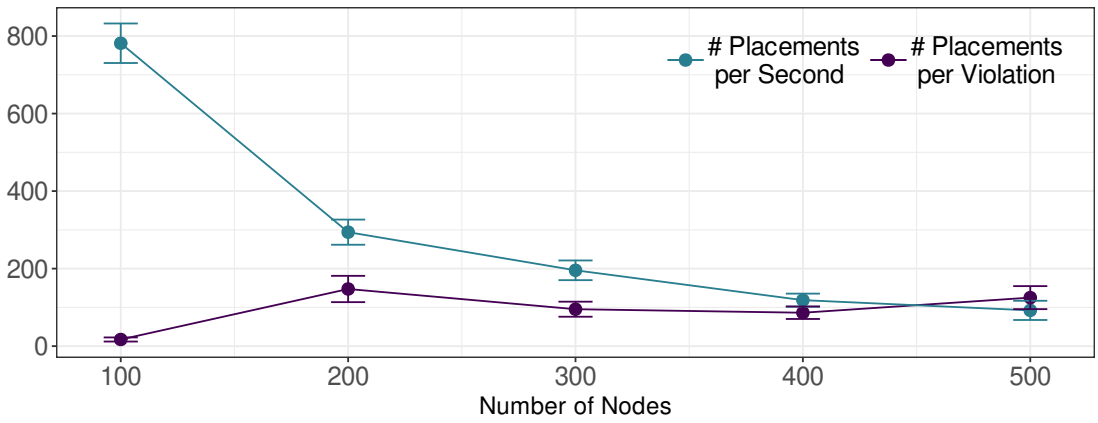


Figure 6.7 – Scalability (simulator).

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.

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.

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.

6.3.6 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 6.7 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 ≈ 100 placements per second. Voilà's algorithms typically need about 100-150 placement evaluations to repair a violation, regardless of system size.

6.4 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.

Chapter 7

Conclusion

7.1 Summary

Although cloud computing offers a reliable solution for a wide range of applications, this paradigm presents limitations in fulfilling all the requirements of a family of emerging applications. Most notably, latency-sensitive applications require their requests to be returned within tight latency bounds. On the other hand, fog computing extends the cloud data centers with additional resources located in the vicinity of the end users. This enables latency-sensitive applications to process their requests without sending them to the cloud and, as a result, to receive a reply with very low network latency. Fog computing platforms are an aggregation of three layers: the end users' devices located at the edge of the network, the edge layer composed of fog nodes which offer ultra-low latencies for the end users and, finally, the traditional data centers.

Providing low user-to-resource latency is one of the fundamental objectives of fog computing. This is achieved by carefully placing hardware and software resources at the edge of the network. Starting from a geo-distributed user base, the fog nodes are geo-distributed as well to grant each user resources in their immediate vicinity. Similarly, fog applications should place their replicas carefully so each user has access to a nearby application replica.

This thesis addresses the specific needs of replicated service-oriented applications. These applications may create functionally-equivalent service replicas that are scattered across the fog nodes, allowing a consistently low user-to-replica latency.

Optimizing network latency in the edge layer for such application model requires one to implement proximity-aware mechanisms within a mature container orchestration engine. One needs to estimate the inter-node latencies between the fog nodes, route the end-users' requests to nearby replicas, detect the sources of traffic and provide them with a nearby replica, update the placement of the replicas when

the workload changes and, finally, scale the replica set to guarantee consistent performance at the lowest possible cost.

We targeted the challenge of proximity-aware resource management for replicated latency-sensitive service-oriented applications in order to control the tail user-perceived latency and to account for the workload non-stationarity in both time and space. This was done over the three levels of resource management: routing, placing, and autoscaling.

In the first contribution, we proposed Proxy-mity, a proximity-aware request routing plugin for Kubernetes. Proxy-mity is able to identify the nearby replicas using Vivaldi coordinates. It can then route requests to nearby replicas. Proxy-mity exposes a single variable α which allows system administrators to control the trade-off between proximity and load imbalance between replicas. The evaluations show the effectiveness of this system in lowering the average user-to-replica latency compared to the traditional load balancing mechanisms used by major cloud orchestration engines.

In the second contribution, we presented Hona, a set of algorithms for replica placement/re-placement. Hona uses Proxy-mity for routing requests and Vivaldi coordinates for estimating the inter-node latencies. It then implements periodic checks to detect the volumes and locations of the sources of traffic. Hona uses heuristics to find a replica placement that is capable of reducing the tail latency while preserving a good load balance between the replicas. Hona dynamically identifies changes in the workload characteristics and updates the placement to maintain performance if a QoS violation is detected. The evaluations show that the Hona heuristics are capable of finding a placement that respects the defined latency bound, and that the re-placement algorithm can cope with a wide variety of changes in the workload.

In the third contribution, we designed Voilà, a tail-latency-aware autoscaler. Voilà relies on Vivaldi coordinates, Proxy-mity, and Hona's periodic checks. Voilà's algorithms dynamically control the number and placement of the replicas to reduce the tail latency, potential replica saturations, and the placement cost. The evaluations show 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.

All of the presented contributions were implemented on top of Kubernetes and have been tested using a real testbed of RPi nodes. The aggregation of Proxy-mity, Hona's

periodic checks, and Voilà represents a complete proximity-aware solution for a mature cloud orchestration engine.

7.2 Future directions

We presented and evaluated a complete set of algorithms that target proximity-aware resource management in fog computing platforms. However, this work allows only a certain type of application and system model defined in Section 2.5.2. As a result, a sensible direction for future research would be to investigate similar algorithms for fog computing platforms that differ in terms of application and/or system model.

7.2.1 Extending the fog with spare nodes

As explained in Section 2.4.3, fog workloads may experience surges as a function of both time and space. To cope with such variations one needs to implement a robust autoscaling algorithm like Voilà, capable of scaling the replica set according to the demand. Yet, some extreme cases may result in the consumption of all the available resources in the proximity of the surge. Consequently, the end users may suffer from a degraded quality of service induced by a necessary best-effort approach to offload their requests to farther fog nodes, or even cloud data centers if no other fog nodes are available.

A straightforward solution for this problem would consist of increasing the size of the available resources at the edge layer. As an analogy, cellular networks are usually provisioned with enough resources to ensure an excellent QoE even under peak traffic conditions [28]. This leads to a low energy efficiency and resource utilization during non-peak hours. For cellular companies like Vodafone Germany, this necessary over-provisioning has resulted in a $6\,\%$ annual decrease in revenue during the period of 2000 to 2009, one of the main reasons being the energy cost [140].

An alternative solution would be the capability of leveraging another source for spare resources [96]. This can be done, for example, by renting nearby resources from another fog infrastructure provider. In some cases, like fog gaming where the users' Sega consoles are used as fog nodes [141], the platform can also rent resources from

currently inactive users in order to process requests originating from nearby active users.

This approach defines a trade-off between performance and cost, where additional resources increase the operational cost but may help in ensuring a better user-perceived QoE [142]. In addition to the proximity metrics that define the QoS for latency-sensitive applications, another type of platform-based requirement emerges. The new end goal would be providing the applications' QoS at the lowest cost.

Creating a proximity-aware resource scheduling system which may dynamically provision and integrate spare nodes would require mechanisms for defining the set of available spare nodes, control mechanisms to rent a spare node, and mechanisms to release them as soon as possible to cut down on costs. This approach might be defined as the fourth level of resource management that controls the scale of the resource pool.

7.2.2 Fog federations

The introduction of cloud computing concepts moved application developers and businesses from owning private in-house data centers toward accessing multi-tenant data centers. This architecture have proven to be successful in terms of cost by enabling the businesses to increase their resource pool in a couple of clicks.

The emerging fog computing industry is following the same footsteps. A number of private fog platforms have already been implemented [50, 141]. As the demand for fog computing technologies is increasing, a new type of fog infrastructure providers will rise to outsource the demand of ultra-low latency resources in the immediate proximity of end users [143, 144]. Multiple fog infrastructure providers are anticipated to appear to serve users located in urban centers or rural areas [143, 145]. In such cases the resources provided by different fog infrastructure providers will be similar in terms of their user-perceived latency.

Instead of directly competing with each other for the same group of end-users, multiple small- or medium-sized fog providers may decide to cover non-overlapping areas. Creating a federation of such fog infrastructure providers would therefore allow the providers to join forces and increase their profits by enabling users to access a bigger and more diverse resource pool.

However, there is currently no fog federation framework similar to the ones implemented in cloud computing [33]. One of the reasons is the absence of resource sharing models and pricing models for the edge layer resources from different

providers. In consequence, resource management in fog federations may become a promising research topic.

We believe that our work may be extended to consider the nodes' provider as a parameter in the process of node selection, whether for routing, placement, or autoscaling. The objective functions used in both Proxy-mity and Voilà permit one to easily add new metrics, which allows one to evaluate placements according to arbitrarily complex requirements.

7.2.3 Fog node heterogeneity

As discussed in Section 2.3, the research community proposes many different types of fog nodes that vary in their size, location, and network connectivity. This is combined with a niche for specialized fog nodes that serve specific types of needs. Most notably, applications making use of AI algorithms have an interest in locating hardware-accelerated devices such as NVIDIA Jetson TX2 in the edge layer [146]. This system-on-module is equipped with special hardware designed to process AI tasks much faster than traditional CPUs and GPUs.

Other examples of specialized fog nodes are gaming consoles and autonomous vehicles. In gaming, Sega is trying to use its gaming consoles in Japan to form a "fog gaming" infrastructure [141]. These consoles are fine tuned to handle gaming tasks. In autonomous vehicles as well, road-side units may be deployed to allow better communication with the vehicles [147]. These devices should have the capabilities to handle an end user moving at high speeds.

In these different use cases, we can expect that a future fog platform composed of regular fog nodes plus specialized nodes must route, place, and autoscale not only according to the proximity and availability of the fog nodes but also according to the node specialization. In this case, AI requests for example have to be specifically routed toward AI fog nodes. AI application replicas should therefore be placed/replaced/autoscaled specifically in the AI nodes. Similarly to fog federations, evaluating possible solutions in such system model require new system metrics that can be added in the system's objective functions.

7.2.4 Fog resource management for microservices

In fog computing platforms, application instances must be distributed over the coverage area to offer nearby resources to all the end users. These instances can take at least two forms: functionally-equivalent replicas like the one discussed in this thesis, or partitioned microservices where different microservices may have different roles.

The main distinction between service replication and partitioning can be summarized in the way requests behave with each application model. With replication, a request is routed toward a monolithic instance that is capable of processing the request and returning the answer. On the other hand, microservice requests are routed from a front-end usually running on or close to the end user's devices toward back-end services located in the edge or the cloud using light communication protocols like Representational State Transfer (REST) over HTTP [148, 149]. Moreover, end-users requests may incur complex invocation graphs involving multiple microservices.

In the literature, most of the work has considered the edge layer as a single entity where all the edge nodes are capable of delivering low latency [149–151]. An interesting approach would be taking the resource management of microservices to the next level where the inter-node latencies are accounted for, rather than considering all the fog nodes equivalent in terms of their latency to the end users.

Proximity-aware resource management for microservices needs to take another approach compared to monolithic service replication. Proximity-aware routing in microservices can be split into two parts: how to route toward the back-end, and how to route between the different services located in the back-end. Similarly, proximity-aware placement for microservices does not merely require a mapping between a set of equivalent instances and suitable nodes, but a set of inter-related placement problems where the location of each microservice instance can affect the operation of the whole system. Another interesting research question in this context is whether it is more suitable to scale horizontally by creating replicas for an overloaded microservice, or vertically by increasing the amount of resources granted to this microservice.

7.3 Closing statement

This work demonstrates that orchestration systems which were originally designed for cloud-based and 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.

BIBLIOGRAPHY

[1] Shubhika Taneja and Yang Liang. « Google cloud Networking in-depth: Series digest ». http://bit.ly/Cloud_networking. 2019.

[2] CLAudit project. *Planetary-scale cloud latency auditing platform*. http://claudit.feld.cvut.cz/. 2017.

[3] Mohammed S Elbamby, Cristina Perfecto, Mehdi Bennis, and Klaus Doppler. « Toward low-Latency and ultra-reliable virtual Reality ». In: *IEEE Network* 32.2 (2018).

[4] 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 ». In: *arXiv preprint arXiv:1907.11621* (2019).

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

[6] GuGuss. « Cloud provider locations ». http://bit.ly/Cloud_Locations. 2018.

[7] Zhou Wei, Jiang Dejun, Guillaume Pierre, Chi-Hung Chi, and Maarten van Steen. « Service-oriented data denormalization for scalable web applications ». In: *Proceedings of the 17th international conference on World Wide Web (WWW)*. 2008.

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

[9] Ahmed Ali-Eldin, Oleg Seleznjev, Sara Sjöstedt-de Luna, Johan Tordsson, and Erik Elmroth. « Measuring cloud workload burstiness ». In: *Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*. 2014.

[10] Docker Inc. *Docker: Empowering app development for developers*. Docker Documentation. https://docs.docker.com/. 2013.

[11] The Kubernetes Authors. « Kubernetes ». https://kubernetes.io/. 2019.

[12] Docker Inc. *Swarm mode overview*. Docker Documentation. `https://docs.docker.com/engine/swarm/`. 2013.

[13] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. « Mesos: A platform for fine-grained resource sharing in the data center ». In: *Proceedings of the Usenix Symposium on Networked Systems Design and Implementations* (*NSDI*). 2011.

[14] Saiful Hoque, Mathias Santos de Brito, Alexander Willner, Oliver Keil, and Thomas Magedanz. « Towards container orchestration in fog computing infrastructures ». In: *Proceedings of 41st IEEE Annual Computer Software and Applications Conference* (*COMPSAC*). 2017.

[15] Qiang Fan and Nirwan Ansari. « Towards workload balancing in fog computing empowered IoT ». In: *IEEE Transactions on Network Science and Engineering* 7.1 (2020).

[16] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. « Vivaldi: A decentralized network coordinate system ». In: *Proceedings of the ACM SIGCOMM Conference*. 2004.

[17] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. « Wikipedia workload analysis for decentralized hosting ». In: *Computer Networks* 53.11 (2009).

[18] Ali J. Fahs and Guillaume Pierre. « Proximity-aware traffic routing in distributed fog computing platforms ». In: *Proceedings of the ACM/IEEE Cluster, Cloud and Internet Computing Conference* (*CCGrid*). 2019.

[19] Ali J. Fahs and Guillaume Pierre. « Tail-latency-aware fog application replica placement ». In: *Proceedings of the 18th International Conference on Service Oriented Computing*(*ICSOC*). 2020.

[20] Ali J. Fahs, Guillaume Pierre, and Erik Elmroth. « Voilà: Tail-latency-aware fog application replicas autoscaler ». In: *Proceedings of the IEEE Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems* (*MASCOTS*). 2020.

[21] Flexera. « RightScale state of the cloud report 2019 ». `https://bit.ly/RightScaleReport`. 2019.

[22] Data Center Knowledge. « Everything you ever wanted to know (and didn't) about Google data centers but were afraid to ask ». http://bit.ly/Data_Center_Knowledge. 2017.

[23] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. « Fog computing and its role in the Internet of Things ». In: *Proceedings of the workshop on Mobile computing*. 2012.

[24] Benoit Chevarie. « The importance of managing data center fiber ». Belden white paper. http://bit.ly/FiberDataCenter. 2020.

[25] Intel. « Intelligent decisions with Intel Internet of Things ». https://intel.ly/32ybEs2. 2018.

[26] IoT Analytics. « State of the IoT 2018 ». http://bit.ly/State_IoT. 2018.

[27] Verizon. « What is the Latency of 5G? » https://vz.to/2EdT5Sa. 2020.

[28] Ali El Amine. « Radio resource allocation in 5G cellular networks powered by the smart grid and renewable energies ». PhD thesis. Ecole nationale supérieure Mines-Télécom Atlantique, 2019.

[29] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. « Big data and Internet of Things: A roadmap for smart environments ». In: ed. by Nik Bessis and Ciprian Dobre. Springer, 2014. Chap. Fog Computing: A Platform for Internet of Things and Analytics.

[30] Alan Sill. « Standards at the edge of the cloud ». In: *IEEE Cloud Computing* 4.2 (2017).

[31] Subhadeep Sarkar and Sudip Misra. « Theoretical modelling of fog computing: a green computing paradigm to support IoT applications ». In: *Iet Networks* 5.2 (2016).

[32] Luis M Vaquero and Luis Rodero-Merino. « Finding your way in the fog: Towards a comprehensive definition of fog computing ». In: *ACM SIGCOMM Computer Communication Review* 44.5 (2014).

[33] 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 ». In: *Journal of Systems Architecture* 98 (2019).

[34] Eva Marín-Tordera, Xavi Masip-Bruin, Jordi García-Almiñana, Admela Jukan, Guang-Jie Ren, and Jiafeng Zhu. « Do we all really know what a fog node is? Current trends towards an open definition ». In: *Computer Communications* 109 (2017).

[35] Mugen Peng, Shi Yan, Kecheng Zhang, and Chonggang Wang. « Fog-computing-based radio access networks: Issues and challenges ». In: *IEEE Network* 30.4 (2016).

[36] Shao-Chou Hung, Hsiang Hsu, Shao-Yu Lien, and Kwang-Cheng Chen. « Architecture harmonization between cloud radio access networks and fog networks ». In: *IEEE Access* 3 (2015).

[37] Vangelis Gazis, Alessandro Leonardi, Kostas Mathioudakis, Konstantinos Sasloglou, Panayotis Kikiras, and Raghuram Sudhaakar. « Components of fog computing in an industrial Internet of Things context ». In: *Proceedings of the 12th Annual IEEE International Conference on Sensing, Communication, and Networking-Workshops (SECON Workshops)*. 2015.

[38] Ibrahim Abdullahi, Suki Arif, and Suhaidi Hassan. « Ubiquitous shift with information centric network caching using fog computing ». In: *Computational intelligence in information systems*. Springer, 2015.

[39] Niko Mäkitalo, Aleksandr Ometov, Joona Kannisto, Sergey Andreev, Yevgeni Koucheryavy, and Tommi Mikkonen. « Safe, secure executions at the network edge: coordinating cloud, edge, and fog computing ». In: *IEEE Software* 35.1 (2017).

[40] Bo Tang, Zhen Chen, Gerald Hefferman, Tao Wei, Haibo He, and Qing Yang. « A hierarchical distributed fog computing architecture for big data analysis in smart cities ». In: *Proceedings of the ASE BigData & Social Informatics*. 2015.

[41] Claus Pahl, Sven Helmer, Lorenzo Miori, Julian Sanin, and Brian Lee. « A container-based edge cloud PaaS architecture based on Raspberry Pi clusters ». In: *Proceedings of the FiCloud Workshop*. 2016.

[42] Badraddin Alturki, Stephan Reiff-Marganiec, and Charith Perera. « A hybrid approach for data analytics for Internet of Things ». In: *Proceedings of the 7th International Conference on the Internet of Things*. 2017.

[43] Jianhua He, Jian Wei, Kai Chen, Zuoyin Tang, Yi Zhou, and Yan Zhang. « Multitier fog computing with large-scale iot data analytics for smart cities ». In: *IEEE Internet of Things Journal* 5.2 (2017).

[44] OpenFog Consortium. *Real-time subsurface imaging*. `http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip`. 2018.

[45] OpenFog Consortium. *Patient monitoring*. `http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip`. 2018.

[46] Cisco. *Enabling MaaS through a distributed IoT data fabric, fog computing and network protocols*. White paper. `https://bit.ly/3nYkqcM`. 2018.

[47] Bruce Thomas, Ben Close, John Donoghue, John Squires, Phillip De Bondi, Michael Morris, and Wayne Piekarski. « ARQuake: An outdoor/indoor augmented reality first person application ». In: *Proceedings of the International Symposium on Wearable Computers*. 2000.

[48] Pengfei Hu, Sahraoui Dhelim, Huansheng Ning, and Tie Qiu. « Survey on fog computing: architecture, key technologies, applications and open issues ». In: *Journal of Network and Computer Applications* 98 (2017).

[49] Gangyong Jia, Guangjie Han, Aohan Li, and Jiaxin Du. « SSL: Smart street lamp based on fog computing for smarter cities ». In: *IEEE Transactions on Industrial Informatics*. Vol. 14. 11. 2018.

[50] Hamidreza Arkian, Dimitrios Giouroukis, Paulo Souza Junior, and Guillaume Pierre. « Potable water management with integrated fog computing and LoRaWAN technologies ». In: *IEEE IoT Newsletter* (2020).

[51] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. « Towards wearable cognitive assistance ». In: *Proceedings of the MobiSys Conference*. 2014.

[52] Chao Zhu, Giancarlo Pastor, Yu Xiao, and Antti Ylajaaski. « Vehicular fog computing for video crowdsourcing: Applications, feasibility, and challenges ». In: *IEEE Communications Magazine*. Vol. 56. 10. 2018.

[53] Yuhua Lin and Haiying Shen. « CloudFog: Leveraging fog to extend cloud gaming for thin-client MMOG with high Quality of Service ». In: *IEEE Transactions on Parallel and Distributed Systems* 28.2 (2017).

[54] OpenFog Consortium. *Visual security and surveillance scenario (3.2)*. https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf. 2017.

[55] OpenFog Consortium. *Out of the fog: Use case scenarios (live video broadcasting)*. http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip. 2018.

[56] Utsav Drolia, Katherine Guo, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. « Cachier: Edge-caching for recognition applications ». In: *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2017.

[57] Ge Ma, Zhi Wang, Miao Zhang, Jiahui Ye, Minghua Chen, and Wenwu Zhu. « Understanding performance of edge content caching for mobile video streaming ». In: *IEEE Journal on Selected Areas in Communications* 35.5 (2017).

[58] Hamidreza Arkian, Guillaume Pierre, Johan Tordsson, and Erik Elmroth. « An experiment-driven performance model of stream processing operators in fog computing environments ». In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC)*. 2020.

[59] OpenFog Consortium. *Out of the fog: Use case scenarios (high-scale drone package delivery)*. http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip. 2018.

[60] OpenFog Consortium. *Autonomous Driving*. http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip. 2018.

[61] Swarnava Dey and Arijit Mukherjee. « Robotic SLAM: A review from fog computing and mobile edge computing perspective ». In: *Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing Networking and Services (MOBIQUITOUS)*. 2016.

[62] Hadeal Abdulaziz Al Hamid, Sk Md Mizanur Rahman, M. Shamim Hossain, Ahmad Almogren, and Atif Alamri. « A security model for preserving the privacy of medical big data in a healthcare cloud using a fog computing facility with pairing-based cryptography ». In: *IEEE Access* 5 (2017).

[63] Pengzhan Hao, Yongshu Bai, Xin Zhang, and Yifan Zhang. « Edgecourier: An edge-hosted personal service for low-bandwidth document synchronization in mobile cloud storage services ». In: *Proceedings of the 2nd ACM/IEEE Symposium on Edge Computing*. 2017.

[64] Yan-Da Chen, Muhammad Zulfan Azhari, and Jenq-Shiou Leu. « Design and implementation of a power consumption management system for smart home over fog-cloud computing ». In: *Proceedings of the 3rd International Conference on Intelligent Green Building and Smart Grid (IGBSG)*. 2018.

[65] Shanhe Yi, Zhengrui Qin, and Qun Li. « Security and privacy issues of fog computing: A survey ». In: *Proceedings of the International conference on wireless algorithms, systems, and applications*. Springer. 2015.

[66] Arwa Alrawais, Abdulrahman Alhothaily, Chunqiang Hu, and Xiuzhen Cheng. « Fog computing for the Internet of Things: Security and privacy issues ». In: *IEEE Internet Computing* 21.2 (2017).

[67] OpenStack. « The OpenStack project ». `https://www.openstack.org/`. 2010.

[68] Adisorn Lertsinsrubtavee, Anwaar Ali, Carlos Molina-Jimenez, Arjuna Sathiaseelan, and Jon Crowcroft. « PiCasso: A lightweight edge computing platform ». In: *Proceedings of the IEEE CloudNet Conference*. 2017.

[69] Michael Chima Ogbuachi, Anna Reale, Péter Suskovics, and Benedek Kovács. « Context-aware Kubernetes scheduler for edge-native applications on 5G ». In: *Journal of Communications Software and Systems* 16.1 (2020).

[70] José Santos, Tim Wauters, Bruno Volckaert, and Filip De Turck. « Towards network-aware resource provisioning in Kubernetes for fog computing applications ». In: *Proceedings of the IEEE Conference on Network Softwarization (NetSoft)*. 2019.

[71] Cecil Wöbker, Andreas Seitz, Harald Mueller, and Bernd Bruegge. « Fogernetes: Deployment and management of fog computing applications ». In: *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*. 2018.

[72] Udi Nachmany. *Kubernetes: Evolution of an IT revolution*. `https://bit.ly/3fX763x`. 2018.

[73] Cloud Native Computing Foundation. « 2019 CNCF Survey results are here: Deployments are growing in size and speed as cloud native adoption becomes mainstream ». `http://bit.ly/CNCFSUR`. 2020.

[74] The New Stack. « Kubernetes is the new standard for computing, including the edge ». `https://bit.ly/3lHJ2on`. 2020.

[75] The K3s authors. « The certified Kubernetes distribution built for IoT and Edge computing ». https://k3s.io/. 2019.

[76] Hrishikesh Barua. « CNCF approves Kubernetes edge computing platform KubeEdge as incubating project ». https://bit.ly/3lMziZQ. 2020.

[77] Wei-Sheng Zheng and Li-Hsing Yen. « Auto-scaling in Kubernetes-based fog computing platform ». In: *Proceedings of the International Computer Symposium*. Springer. 2018.

[78] The Linux Foundation. « etcd: A distributed, reliable key-value store for the most critical data of a distributed system ». https://etcd.io. 2020.

[79] Lakshminarayanan Subramanian, Venkata N Padmanabhan, and Randy H Katz. « Geographic properties of Internet routing ». In: *Proceedings of the Usenix Annual Technical Conference (ATC)*. 2002.

[80] HashiCorp. « Serf: Decentralized Cluster Membership, Failure Detection, and Orchestration ». https://www.serf.io/. 2013.

[81] 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 ». In: *Scientific data* 2.1 (2015).

[82] Klervie Toczé and Simin Nadjm-Tehrani. « A taxonomy for management and optimization of multiple resources in edge computing ». In: *Wireless Communications and Mobile Computing* 2018 (2018).

[83] Mostafa Ghobaei-Arani, Alireza Souri, and Ali A Rahmanian. « Resource management approaches in fog computing: a comprehensive review ». In: *Journal of Grid Computing* (2019).

[84] Vincenzo De Maio and Ivona Brandic. « Multi-objective mobile edge provisioning in small cell clouds ». In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. 2019.

[85] Alexandre van Kempen, Teodor Crivat, Benjamin Trubert, Debaditya Roy, and Guillaume Pierre. « MEC-ConPaaS: An experimental single-board based mobile edge cloud ». In: *Proceedings of the 5th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. 2017.

[86] Ye Yu, Xin Li, and Chen Qian. « SDLB: A scalable and dynamic software load balancer for fog and mobile edge computing ». In: *Proceedings of the ACM Workshop on Mobile Edge Communications*. 2017.

[87] Liang Huang, Xu Feng, Luxin Zhang, Liping Qian, and Yuan Wu. « Multi-server multi-user multi-task computation offloading for mobile edge computing networks ». In: *Sensors* 19.6 (2019).

[88] Xiang Wang, Supeng Leng, and Kun Yang. « Social-aware edge caching in fog radio access networks ». In: *IEEE Access* 5 (2017).

[89] 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 ». In: *Transactions on Emerging Telecommunications Technologies* 30.4 (2019).

[90] Atakan Aral, Ivona Brandic, Rafael Brundo Uriarte, Rocco De Nicola, and Vincenzo Scoca. « Addressing application latency requirements through edge scheduling ». In: *Journal of Grid Computing* 17.4 (2019).

[91] Fung Po Tso, David R White, Simon Jouet, Jeremy Singer, and Dimitrios P Pezaros. « The Glasgow Raspberry Pi cloud: A scale model for cloud computing infrastructures ». In: *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems Workshops (ICDCS)*. 2013.

[92] Paolo Bellavista and Alessandro Zanni. « Feasibility of fog computing deployment based on Docker containerization over RaspberryPi ». In: *Proceedings of the 18th international conference on distributed computing and networking (ICDCN)*. 2017.

[93] Arif Ahmed. « Efficient cloud application deployment in distributed fog infrastructures ». PhD thesis. Université de Rennes 1, 2020.

[94] 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 ». In: *IEEE Transactions on Industrial Informatics* (2019).

[95] Qiang Li, Jin Lei, Jingran Lin, and Xiaoxiao Wu. « Latency minimization for multiuser computation offloading in fog-radio access networks ». In: *arXiv preprint arXiv:1907.08759* (2019).

[96] Klervie Toczé and Simin Nadjm-Tehrani. « ORCH: Distributed orchestration framework using mobile edge devices ». In: *Proceedings of the 3rd IEEE International Conference on Fog and Edge Computing (ICFEC)*. 2019.

[97] Ahmed Jawad Kadhim and Seyed Amin Hosseini Seno. « Energy-efficient multicast routing protocol based on SDN and fog computing for vehicular networks ». In: *Ad Hoc Networks* 84 (2019).

[98] Sura Khalil Abd, Syed Abdul Rahman Al-Haddad, Fazirulhisyam Hashim, Azizol BHJ Abdullah, and Salman Yussof. « Energy-aware fault tolerant task offloading of mobile cloud computing ». In: *Proceedings of the 5th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. 2017.

[99] Huaiying Sun, Huiqun Yu, Guisheng Fan, and Liqiong Chen. « Energy and time efficient task offloading and resource allocation on the generic IoT-fog-cloud architecture ». In: *Peer-to-Peer Networking and Applications* 13.2 (2020).

[100] Lin Gu, Deze Zeng, Song Guo, Ahmed Barnawi, and Yong Xiang. « Cost efficient resource management in fog computing supported medical cyber-physical system ». In: *IEEE Transactions on Emerging Topics in Computing* 5.1 (2015).

[101] Ashkan Yousefpour, Genya Ishigaki, Riti Gour, and Jason P Jue. « On reducing IoT service delay via fog offloading ». In: *IEEE Internet of Things Journal* 5.2 (2018).

[102] Fani Basic, Atakan Aral, and Ivona Brandic. « Fuzzy handoff control in edge offloading ». In: *Proceedings of the IEEE International Conference on Fog Computing (ICFC)*. 2019.

[103] Feyza Yildirim Okay and Suat Ozdemir. « Routing in fog-enabled IoT platforms: A survey and an SDN-based solution ». In: *IEEE Internet of Things Journal* 5.6 (2018).

[104] Harikrishna Pydi and Ganesh Neelakanta Iyer. « Analytical review and study on load balancing in edge computing platform ». In: *Proceedings of the 4th IEEE International Conference on Computing Methodologies and Communication (ICCMC)*. 2020.

[105] Ting Lu, Shan Chang, and Wei Li. « Fog computing enabling geographic routing for urban area vehicular network ». In: *Peer-to-Peer Networking and Applications* 11.4 (2018).

[106] Naserali Noorani and Seyed Amin Hosseini Seno. « Routing in VANETs based on intersection using SDN and fog computing ». In: *Proceedings of the 8th IEEE International Conference on Computer and Knowledge Engineering (ICCKE)*. 2018.

[107] Deepak Puthal, Mohammad S Obaidat, Priyadarsi Nanda, Mukesh Prasad, Saraju P Mohanty, and Albert Y Zomaya. « Secure and sustainable load balancing of edge data centers in fog computing ». In: *IEEE Communications Magazine* 6.5 (2018).

[108] Roberto Beraldi, Abderrahmen Mtibaa, and Hussein Alnuweiri. « Cooperative load balancing scheme for edge computing resources ». In: *Proceedings of the 2nd International Conference on Fog and Mobile Edge Computing (FMEC)*. 2017.

[109] Andreas Kapsalis, Panagiotis Kasnesis, Iakovos S Venieris, Dimitra I Kaklamani, and Charalampos Z Patrikakis. « A cooperative fog approach for effective workload balancing ». In: *IEEE Cloud Computing* 4.2 (2017).

[110] Magnus Karlsson, Christos Karamanolis, and Mallik Mahalingam. *A framework for evaluating replica placement Algorithms*. Tech. rep. HPL-2002-219. HP Labs Palo Alto, 2002.

[111] Jad Darrous, Thomas Lambert, and Shadi Ibrahim. « On the Importance of Container Image Placement for Service Provisioning in the Edge ». In: *Proceedings of the 28th International Conference on Computer Communication and Networks (ICCCN)*. 2019.

[112] Michał Szymaniak, Guillaume Pierre, and Maarten van Steen. « Latency-driven replica placement ». In: *IPSJ Journal* 47.8 (2006).

[113] Isaac Lera, Carlos Guerrero, and Carlos Juiz. « Comparing centrality indices for network usage optimization of data placement policies in fog devices ». In: *Proceedings of the 3rd International Conference on Fog and Mobile Edge Computing (FMEC)*. 2018.

[114] Juan Liu, Bo Bai, Jun Zhang, and Khaled B Letaief. « Cache placement in Fog-RANs: From centralized to distributed algorithms ». In: *IEEE Transactions on Wireless Communications* 16.11 (2017).

[115] Mohammed Islam Naas, Philippe Raipin Parvedy, Jalil Boukhobza, and Laurent Lemarchand. « iFogStor: an IoT data placement strategy for fog infrastructure ». In: *Proceedings of the 1st IEEE International Conference on Fog and Edge Computing (ICFEC)*. 2017.

[116] Atakan Aral and Tolga Ovatman. « A decentralized replica placement algorithm for edge computing ». In: *IEEE Transactions on Network and Service Management* 15.2 (2018).

[117] Yanling Shao, Chunlin Li, and Hengliang Tang. « A data replica placement strategy for IoT workflows in collaborative edge and cloud environments ». In: *Computer Networks* 148 (2019).

[118] Kangkang Li and Jarek Nabrzyski. « Traffic-aware virtual machine placement in cloudlet mesh with adaptive bandwidth ». In: *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 2017.

[119] Lei Zhao, Jiajia Liu, Yongpeng Shi, Wen Sun, and Hongzhi Guo. « Optimal placement of virtual machines in mobile edge computing ». In: *Proceedings of the IEEE Global Communications Conference (GLOBECOM)*. 2017.

[120] Ya-Ju Yu, Te-Chuan Chiu, Ai-Chun Pang, Ming-Fan Chen, and Jiajia Liu. « Virtual machine placement for backhaul traffic minimization in fog radio access networks ». In: *Proceedings of the IEEE International Conference on Communications (ICC)*. 2017.

[121] Fatma Ben Jemaa, Guy Pujolle, and Michel Pariente. « QoS-aware VNF placement optimization in edge-central carrier cloud architecture ». In: *Proceedings of the IEEE Global Communications Conference (GLOBECOM)*. 2016.

[122] Hua-Jun Hong, Pei-Hsuan Tsai, and Cheng-Hsin Hsu. « Dynamic module deployment in a fog computing platform ». In: *Proceedings of the 18th IEEE Asia-Pacific Network Operations and Management Symposium* (*APNOMS*). 2016.

[123] Alessio Silvestro, Nitinder Mohan, Jussi Kangasharju, Fabian Schneider, and Xiaoming Fu. « Mute: Multi-tier edge networks ». In: *Proceedings of the 5th Workshop on CrossCloud Infrastructures & Platforms*. 2018.

[124] Jinlai Xu, Balaji Palanisamy, Heiko Ludwig, and Qingyang Wang. « Zenith: Utility-aware resource allocation for edge computing ». In: *Proceedings of the IEEE international conference on edge computing* (*EDGE*). 2017.

[125] Olena Skarlat, Matteo Nardelli, Stefan Schulte, and Schahram Dustdar. « Towards QoS-aware fog service placement ». In: *Proceedings of the 1st IEEE international conference on Fog and Edge Computing* (*ICFEC*). 2017.

[126] Hengliang Tang, Chunlin Li, Jingpan Bai, JianHang Tang, and Youlong Luo. « Dynamic resource allocation strategy for latency-critical and computation-intensive applications in cloud-edge environment ». In: *Computer Communications* 134 (2019).

[127] Chunlin Li, YaPing Wang, Hengliang Tang, Yujiao Zhang, Yan Xin, and Youlong Luo. « Flexible replica placement for enhancing the availability in edge computing environment ». In: *Computer Communications* 146 (2019).

[128] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. « CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms ». In: *Software: Practice and Experience* 41.1 (2011).

[129] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K Ghosh, and Rajkumar Buyya. « iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments ». In: *Software: Practice and Experience* 47.9 (2017).

[130] Gil Tene. *How NOT to measure latency*. `https://bit.ly/2uyET1f`. 2012.

[131] Gourav Rattihalli, Madhusudhan Govindaraju, Hui Lu, and Devesh Tiwari. « Exploring potential for non-disruptive vertical auto scaling and resource estimation in Kubernetes ». In: *Proceedings of the 12th IEEE International Conference on Cloud Computing* (*CLOUD*). 2019.

[132] Salman Taherizadeh and Marko Grobelnik. « Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications ». In: *Advances in Engineering Software* 140 (2020).

[133] Thanh-Tung Nguyen, Yu-Jin Yeom, Taehong Kim, Dae-Heon Park, and Sehan Kim. « Horizontal pod autoscaling in Kubernetes for elastic container orchestration ». In: *Sensors* 20.16 (2020).

[134] Jieming Zhu, Zibin Zheng, Yangfan Zhou, and Michael R Lyu. « Scaling service-oriented applications into geo-distributed clouds ». In: *Proceedings of the 7th IEEE International Symposium on Service-Oriented System Engineering*. 2013.

[135] Chenhao Qu. « Auto-scaling and deployment of web applications in distributed computing clouds ». PhD thesis. University of Melbourne, 2016.

[136] Emanuel Ferreira Coutinho, Flávio Rubens de Carvalho Sousa, Paulo Antonio Leal Rego, Danielo Gonçalves Gomes, and José Neuman de Souza. « Elasticity in cloud computing: a survey ». In: *annals of telecommunications-annales des télécommunications* 70.7-8 (2015).

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

[138] Arif Ahmed and Guillaume Pierre. « Docker container deployment in fog computing infrastructures ». In: *Proceedings of the IEEE International Conference on Edge Computing (EDGE)*. 2018.

[139] Wajdi Hajji and Fung Po Tso. « Understanding the performance of low power Raspberry Pi cloud for big data ». In: *Electronics* 5.2 (2016).

[140] Albrecht Fehske, Gerhard Fettweis, Jens Malmodin, and Gergely Biczok. « The global footprint of mobile communications: The ecological and economic perspective ». In: *IEEE communications magazine* 49.8 (2011).

[141] Kris Holt. « Sega wants to turn Japanese arcades into 'fog gaming' data centers ». `https://engt.co/3cZfwaw`. 2020.

[142] Djawida Dib. « Optimizing PaaS provider profit under service level agreement constraints ». PhD thesis. Université de Rennes 1, 2014.

[143] Cosimo Anglano, Massimo Canonico, Paolo Castagno, Marco Guazzone, and Matteo Sereno. « A game-theoretic approach to coalition formation in fog provider federations ». In: *Proceedings of the 3rd IEEE International Conference on Fog and Mobile Edge Computing (FMEC)*. 2018.

[144] Schneider Electric. « Customer insight: Future-proofing your colocation business ». `https://bit.ly/3cZg7ZO`. 2017.

[145] Joe Weinman. « The economics of the hybrid multicloud fog ». In: *IEEE Cloud Computing* 4.1 (2017).

[146] Florin Manaila. « Moving AI from the data center to edge or fog Computing ». `https://ibm.co/36C4PcK`. 2020.

[147] Cunqian Yu, Bin Lin, Ping Guo, Wei Zhang, Sen Li, and Rongxi He. « Deployment and dimensioning of fog computing-based Internet of Vehicle infrastructure for autonomous driving ». In: *IEEE Internet of Things Journal* 6.1 (2018).

[148] Genc Tato, Marin Bertier, Etienne Rivière, and Cédric Tedeschi. « ShareLatex on the edge: Evaluation of the hybrid core/edge deployment of a microservices-based application ». In: *Proceedings of the 3rd Workshop on Middleware for Edge Clouds & Cloudlets*. 2018.

[149] Genc Tato, Marin Bertier, Etienne Rivière, and Cédric Tedeschi. « Split and migrate: Resource-driven placement and discovery of microservices at the edge ». In: *Proceedings of the Conference On Principles Of Distributed Systems (OPODIS)*. 2019.

[150] Ion-Dorinel Filip, Florin Pop, Cristina Serbanescu, and Chang Choi. « Microservices scheduling model over heterogeneous cloud-edge environments as support for IoT applications ». In: *IEEE Internet of Things Journal* 5.4 (2018).

[151] Samodha Pallewatta, Vassilis Kostakos, and Rajkumar Buyya. « Microservices-based IoT application placement within heterogeneous and resource constrained fog computing environments ». In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*. 2019.

UNIVERSITE BRETAGNE LOIRE MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Gestion des Répliques avec Prise en Compte de la Proximité dans les Plates-Formes Géo-Distribuées de Fog Computing.

Mot clés : Fog computing, Gestion de la proximité, Gestion des ressources, Kubernetes.

Résumé : L'architecture géo-distribuée de fog computing fournit aux utilisateurs des ressources accessibles avec une faible latence. Cependant, exploiter pleinement cette architecture nécessite une distribution similaire de l'application par l'utilisation de techniques de réplication. Par conséquent, la gestion de ces répliques doit intégrer des algorithmes prenant en compte la proximité aux différents niveaux de gestion des ressources du système.

Dans cette thèse, nous avons abordé ce problème à travers trois contributions. Premièrement, nous avons conçu un système de routage des requêtes entre les utilisateurs et les ressource prenant en compte la proximité. Deuxièmement, nous avons proposé des algorithmes dynamiques pour le placement des répliques prenant en compte les derniers percentiles de la latence. Enfin, nous avons développé un système de mise à l'échelle automatique qui ajustent le nombre des répliques de l'application en fonction de la charge subie par les applications fog computing.

Title: Proximity-Aware Replicas Management in Geo-Distributed Fog Computing Platforms.

Keywords: Fog computing, Proximity-awareness, Resource management, Kubernetes.

Abstract: Geo-distributed fog computing architectures provide users with resources reachable within low latency. However, fully exploiting the fog architecture requires a similar distribution of the application by the means of replication. As a result, fog application replica management should implement proximity-aware algorithms to handle different levels of resource management.

In this thesis, we addressed this problem over three contributions. First, we designed a proximity-aware user-to-replica routing mechanism. Second, we proposed dynamic tail-latency-aware replica placement algorithms. Finally, we developed autoscaling algorithms to dynamically scale the application resources according to the non-stationary workload experienced by fog platforms.