Tail-Latency-Aware Fog Application Replica Placement

Ali Fahs, Guillaume Pierre

Tail-Latency-Aware Fog Application Replica Placement

Ali J. Fahs Guillaume Pierre
Univ Rennes, Inria, CNRS, IRISA (Rennes, France)

Abstract. Latency-sensitive applications often use fog computing platforms to place replicas of their services as close as possible to their end users. A good placement should guarantee a low tail network latency between end-user devices and their closest replica while keeping the replicas load balanced. We propose a latency-aware scheduler integrated in Kubernetes which uses simple yet highly-effective heuristics to identify suitable replica placements, and to dynamically update these placements upon any evolution of user-generated traffic.

Introduction

Predictable low response time is an essential property for a large range of modern applications such as augmented reality and real-time industrial IoT [1]. When such applications are hosted in Cloud platforms, their response time depends on the provisioned processing capacity and the network characteristics between the end users and the cloud servers. However, users are often dispersed across a broad geographic area far from the cloud data centers. This motivates the need for Fog computing platforms which extend Cloud platforms with additional computing resources located in the vicinity of the end users, where distributed applications may deploy one or more replicated VM or container instances [11].

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 [19] 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 [24]. Hona uses Kubernetes to monitor the system resource availability, Vivaldi coordinates to estimate the network latency between nodes [5] and `proxy-mity` to monitor traffic sources and to route end-user traffic to nearby replicas [7]. 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 automatically takes corrective re-placement actions when the characteristics of the end-user workload changes.

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.

Background

Kubernetes

We base this work on the Kubernetes platform which automates the deployment, scaling and management of containerized applications in large-scale computing infrastructures [24]. 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 replicas and constitute the system's computing, network and storage resources.

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

Network traffic routing: User requests addressed to a Kubernetes service are first routed to a *gateway* node within the Kubernetes system. Every worker node can act as a gateway: the fog computing platform is in charge of routing incoming traffic to any one of them using networking technologies such as WiFi and LTE, possibly in combination with SDN/NFV. Second, the request is further routed internally to the Kubernetes system. Kubernetes services are composed of iptables or IPVS rules installed in every worker node.

Pod scheduling: When a new set of pods is created, the Kubernetes scheduler is in charge of deciding which worker nodes will be in charge of executing them. The scheduler selects a list of nodes that are capable of executing the new pods,

[1] Hona (هُنَا) means "here" in Arabic.

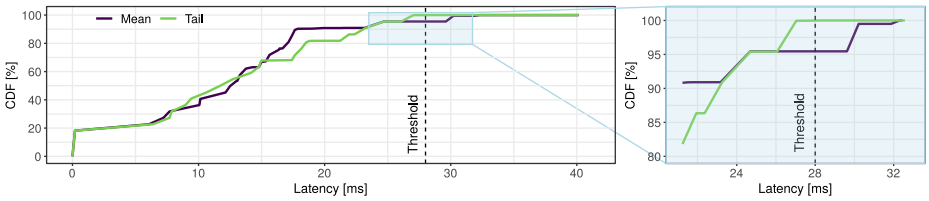

Fig. 1: Optimizing the mean or the tail latency.

and stores this decision in an object store called *etcd*. In every worker node, a *kubelet* daemon periodically checks *etcd* and deploys the assigned pods.

Kubernetes' limitations: Kubernetes was designed to manage cluster-based or cloud-based platforms. In consequence, it considers all worker nodes as functionally equivalent to one another, and it does not have any notion of node proximity to the end users. To make it suitable for fog computing scenarios, we aim to modify its scheduling components to proactively place pods in worker nodes located close to the main sources of network traffic. This allows one to considerably reduce the network latencies between the end-user devices and the nodes serving them, while keeping replicas reasonably load-balanced.

Network proximity

In fog computing platforms, servers are located close to the end users but necessarily far from each other. Choosing a replica placement in such environment requires an accurate estimation of network latencies across the full system.

Estimating network latencies: Hona models network latencies using Vivaldi to accurately predict the latency between hosts without contacting all of them [5]. Hona specifically uses Serf, a mature open-source tool which maintains cluster membership and offers a robust implementation of Vivaldi coordinates [9].

Routing requests to a nearby node: By default, Kubernetes gateways route every incoming request to any node holding a pod of the application regardless of its location. To serve end user requests by nearby replicas, we use `proxy-mity` which redefines the network routing rules to route requests with high probability to a nearby application replica [7]. To avoid overloading certain nodes while others are underutilized, `proxy-mity` allows one to define a tradeoff between proximity and load-balancing.

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 [6]."* Aiming to minimize the mean latency between the user devices and their closest replica does not allow one to satisfy such extremely demanding type of requirements.

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 Section 36). We then select the two placements which respectively minimize the mean ("Mean") and the number of

Table 1: State of the art.

Type	Ref.	Dyn.	Rep.	Obj.	Eval.	Type	Ref.	Dyn.	Rep.	Obj.	Eval.
Data	[18]	✗	✗	RT	Sim	Service	[21]	✗	✗	PX,DT	Sim
	[17]	✗	✗	RT	Sim		[22]	✗	✗	PX,RU	Sim
	[14]	✗	✗	NU	Sim		[12]	✗	✗	RT,RU	Sim
	[2]	✓	✓	RT	Sim		[26]	✗	✗	PX	Sim
	[20]	✓	✓	RT	Sim		[23]	✓	✗	PX,DT	Testbed
VM	[15]	✗	✗	NU	Sim		[10]	✗	✓	DT	Testbed
	[28]	✗	✓	NU	Sim		[16]	✓	✓	PX,RU	Testbed
	[27]	✓	✓	NU	Sim		Hona	✓	✓	PX,LB	Testbed+Sim

requests with device-to-closest-replica latencies greater than a threshold $L = 28\,ms$ ("Tail"). Figure 1 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.

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

State of The Art

The replica placement problem has been extensively studied since the creation of the first geo-distributed environments such as content delivery networks [13], and a very large number of papers have been published on this topic. Table 1 classifies the most relevant recent publications along multiple dimensions:

Type describes *what* is being placed. Data placement [17] focuses on the download delay of cached items by placing caches in specific locations. VM placement aims to reduce network usage [15, 27, 28] while service placement optimizes mostly network proximity and resource utilization [16, 21–23, 26].

Dynamicity (Dyn) matters in systems which may experience considerable workload variations over time. Many papers focus on the initial placement problem only, without trying to update the placements upon workload changes.

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

Objective (Obj) represents the optimized metrics: Response Time (RT) is 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; Proximity (PX) is

the latency between end-user and the closest application replica; and Load Balancing (LB) is the equal distribution of load across replicas.

Evaluation (Eval) of placement algorithms is often done using simulators such as CloudSim [4] and iFogsim [8]. However, some authors also use actual prototypes and evaluate them in a real environment or a testbed.

Few papers in Table 1 propose dynamic placement algorithms for replica sets. Yu *et al.* study the placement of replicated VMs to minimize the backhaul network traffic [27]. 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.* [2] and Shao *et al.* [20] propose dynamic replica placement algorithms for data services in edge computing. Similarly, Li *et al.* [16] 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 1, optimizing the mean latency does not necessarily imply an improvement in the human-perceived quality of service. These papers also do not consider load balancing between replicas. Finally, only [16] has implemented and tested its proposed algorithms in a real testbed.

In contrast, to our best knowledge, Hona presents the first dynamic replica placement algorithm which aims to maintain the tail latency and the load imbalance within pre-defined bounds. Hona solves the placement problem based on the network routes as well as the origin of traffic, and has been implemented in a mature container orchestration system.

System design

The objective of this work is to dynamically choose 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).

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 $P\%$:

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

$$\sigma_{req} = \sqrt{\frac{1}{r} \times \sum_{i=1}^{r} (\varphi_i.req - \mu_{req})^2}$$

$$I\% = \frac{\sigma_{req}}{\sum_{i=1}^{r} \varphi_i.req} \times 100\%$$

Likewise, the Imbalance metric $I\%$ evaluate the load balancing between replicas. Lower Imbalance values depict a better system. We define Imbalance as the standard deviation of the workloads of individual replicas for a given application.

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. We use $\alpha = 0.95$ to favorize Proximity over Imbalance improvements. This function can easily be extended to integrate other metrics such as financial cost and energy consumption.

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 re-placement problem the application is already deployed so we can rely on the specific traffic addressed to it.

Evaluating the two metrics requires three types of input data:

Cluster information including the nodes, their resources, the pods, and their hosting node is maintained by Kubernetes itself. We can access it with simple calls to its `etcd` service.

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

Traffic information can be obtained from `proxy-mity` which logs the source and destination of each request transmitted. `proxy-mity` makes this information available to Hona's scheduler via a call to its local Serf agent.

Initial replica placement

When deploying an application for the first time, finding the optimal placement for r replicas among n nodes requires in principle one to explore the full set Ω of possible placements and choose the one which optimizes the objective function Θ. Unfortunately, this space is extremely large even for modest values of r and n, so exploring it in its entirety is not feasible.

We however note that it is not 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. In practice we define a Proximity threshold which represents a sufficiently good solution. Our heuristics stop the search as soon as they find a solution which exceeds the threshold, or when the time quota allocated to the search expires.

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 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 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 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 fraction of total number of system nodes to the desired number of replicas, multiplied by a user-defined variable p.

Algorithm 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
```

Algorithm 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
```

Algorithm 3: Hona's replica replacement 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)
```

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.

Replica re-placement

Online systems often observe significant variations over time of the characteristics of the traffic they receive [25]. 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 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.

Evaluation

We evaluate this work using a combination of experimental measurements and simulations. The experimental setup consists of 22 Raspberry Pi (RPi) model 3B+ single-board computers acting as fog computing servers. Such machines are frequently used to prototype fog computing infrastructures [3]. 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`.

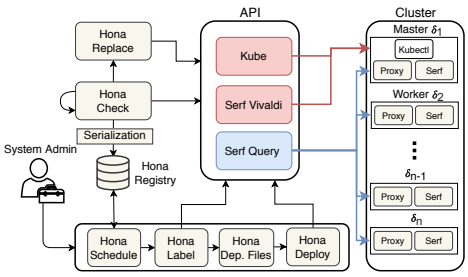

Fig. 2: System architecture.

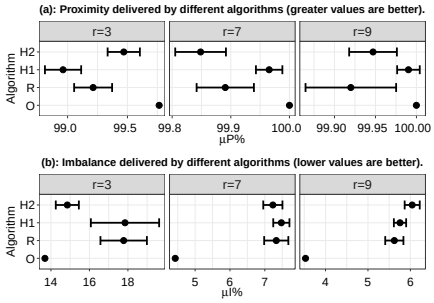

Fig. 3: Initial replica placement analysis (testbed, $n = 21$).

As shown in Figure 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.

We emulate realistic network latencies between the worker nodes using the Linux `tc` [2]command. We specifically use latency values measured between European cities [3]. Network latencies range from 3 ms to 80 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.

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.

Overall performance (testbed experiments): Figure 3 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.

[2] https://linux.die.net/man/8/tc
[3] https://wondernetwork.com/

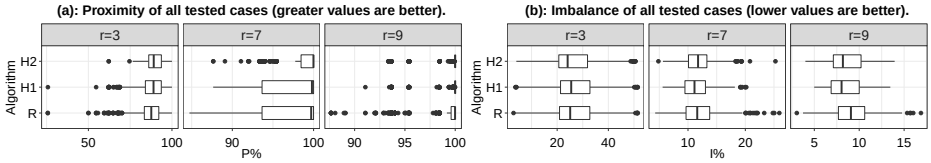

Fig. 4: Individual test cases analysis (testbed, $n = 21$).

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.

To better understand the differences between the Random and the Hona heuristics, Figure 4 depicts the 5th/25th/50th/75th/95th percentiles of *all the tested placements* during the same experiment. In contrast, Figure 3 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 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 5a and 5b, 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 requirements (Figure 5c). We observe that H2 finds solutions much quicker than the other heuristics.

Finally, Figure 5d 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.

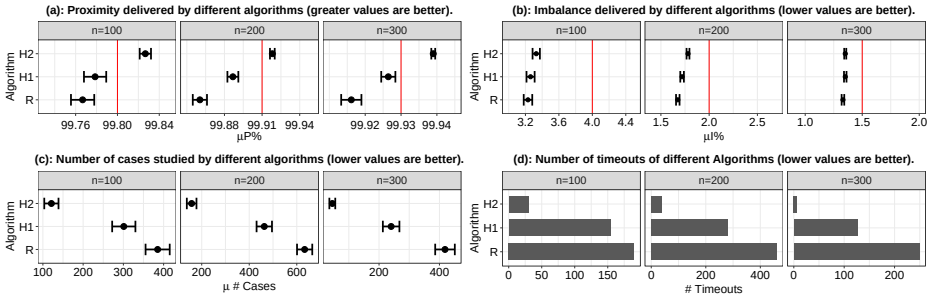

Fig. 5: Initial replica placement with various system sizes (simulator, $r = n/10$).

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 paper we therefore use this heuristic for the initial replica placements.

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, 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 6 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 6a 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 6b 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

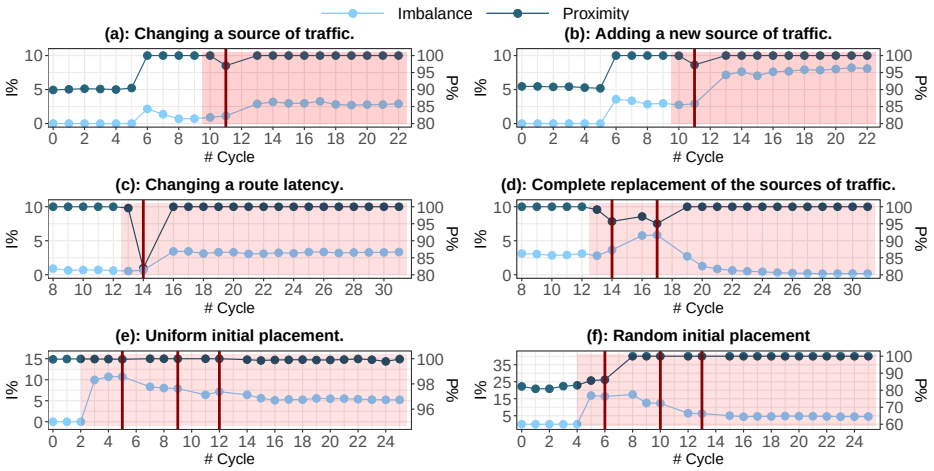

Fig. 6: Replica re-placement analysis (testbed, $n = 21$).

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.

(c) Changing a route latency: Figure 6c 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 6d 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 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 6e 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 6f shows a case where the initial replica placement was chosen randomly. When traffic starts in cycle 4,

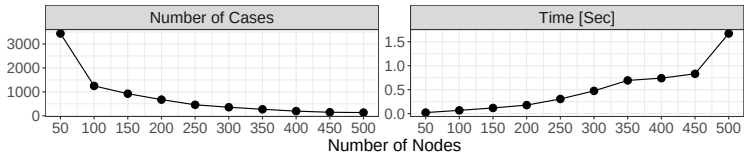

Fig. 7: Complexity of the H2 heuristic (simulator).

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.

Computational complexity

Figure 7 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 left part of the figure depict 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 right 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.

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.

References

1. Ahmed, et al.: Fog computing applications: Taxonomy and requirements. arXiv preprint arXiv:1907.11621 (2019)
2. Aral, et al.: A decentralized replica placement algorithm for edge computing. IEEE Transactions on Network and Service Management **15**(2) (2018)

3. Bellavista, et al.: Feasibility of fog computing deployment based on Docker containerization over RaspberryPi. In: Proc. ACM ICDCN (2017)
4. Calheiros, et al.: CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Software: Practice and Experience **41**(1) (2011)
5. Dabek, et al.: Vivaldi: A decentralized network coordinate system. In: Proc. ACM SIGCOMM (2004)
6. Elbamby, et al.: Toward low-latency and ultra-reliable virtual reality. IEEE Network **32**(2) (2018)
7. Fahs, et al.: Proximity-aware traffic routing in distributed fog computing platforms. In: Proc. ACM/IEEE CCGrid (2019)
8. Gupta, et al.: iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, edge and fog computing environments. Software: Practice and Experience **47**(9) (2017)
9. HashiCorp: Serf: Decentralized cluster membership (2019), https://www.serf.io/
10. Hong, et al.: Dynamic module deployment in a fog computing platform. In: Proc. IEEE APNOMS (2016)
11. IEEE: IEEE standard for adoption of OpenFog reference architecture for fog computing (2018), https://bit.ly/38sXYjU
12. Jemaa, et al.: QoS-aware VNF placement optimization in edge-central carrier cloud architecture. In: Proc. IEEE GLOBECOM (2016)
13. Karlsson, et al.: A framework for evaluating replica placement algorithms. Tech. Rep. HPL-2002-219, HP Labs Palo Alto (2002)
14. Lera, et al.: Comparing centrality indices for network usage optimization of data placement policies in fog devices. In: Proc. IEEE FMEC (2018)
15. Li, et al.: Traffic-aware virtual machine placement in cloudlet mesh with adaptive bandwidth. In: Proc. IEEE CloudCom (2017)
16. Li, et al.: Flexible replica placement for enhancing the availability in edge computing environment. Computer Communications **146** (Oct 2019)
17. Liu, et al.: Cache placement in Fog-RANs: From centralized to distributed algorithms. IEEE Transactions on Wireless Communications **16**(11) (2017)
18. Naas, et al.: iFogStor: an IoT data placement strategy for fog infrastructure. In: Proc. IEEE ICFEC (2017)
19. Optimizely: The most misleading measure of response time. White paper (2013), https://bit.ly/3boHgnZ
20. Shao, et al.: A data replica placement strategy for IoT workflows in collaborative edge and cloud environments. Computer Networks **148** (2019)
21. Silvestro, et al.: Mute: Multi-tier edge networks. In: Proc. CrossCloud (2018)
22. Skarlat, et al.: Towards QoS-aware fog service placement. In: Proc. ICFEC (2017)
23. Tang, et al.: Dynamic resource allocation strategy for latency-critical and computation-intensive applications in cloud-edge environment. Computer Communications **134** (2019)
24. The Kubernetes Authors: Kubernetes, https://kubernetes.io/
25. Urdaneta, et al.: Wikipedia workload analysis for decentralized hosting. Elsevier Computer Networks **53**(11) (2009)
26. Xu, et al.: Zenith: Utility-aware resource allocation for edge computing. In: Proc. IEEE EDGE (2017)
27. Yu, et al.: Virtual machine placement for backhaul traffic minimization in fog radio access networks. In: Proc. IEEE ICC (2017)
28. Zhao, et al.: Optimal placement of virtual machines in mobile edge computing. In: Proc. IEEE GLOBECOM (2017)