Fully Decentralized Horizontal Autoscaling for Burst of Load in Fog Computing

EunChan Park*, KyeongDeok Baek, Eunho Cho and In-Young Ko

Korea Advanced Institute of Science and Technology, South Korea
E-mail: eunchan.park@kaist.ac.kr; blest215@kaist.ac.kr; ehcho@kaist.ac.kr; iko@kaist.ac.kr
**Corresponding Author*

Received 07 August 2023; Accepted 29 September 2023;
Publication 23 December 2023

Abstract

With the increasing number of Web of Things devices, the network and processing delays in the cloud have also increased. As a solution, fog computing has emerged, placing computational resources closer to the user to lower the communication overhead and congestion in the cloud. In fog computing systems, microservices are deployed as containers, which require an orchestration tool like Kubernetes to support service discovery, placement, and recovery. A key challenge in the orchestration of microservices is automatically scaling the microservices in case of an unpredictable burst of load. In cloud computing, a centralized autoscaler can monitor the deployed microservice instances and make scaling actions based on the monitored metric values. However, monitoring an increasing number of microservices in fog computing can cause excessive network overhead and thereby delay the time to scaling action. We propose DESA, a fully DEcentralized Self-adaptive Autoscaler through which microservice instances make their own scaling decisions, cloning or terminating themselves through self-monitoring. We evaluate DESA in a simulated fog computing environment with different numbers of fog nodes. Furthermore, we conduct a case study with the 1998 World Cup website access log, examining DESA's performance

Journal of Web Engineering, Vol. 22_6, 849–870.
doi: 10.13052/jwe1540-9589.2261

in a realistic scenario. The results show that DESA successfully reduces the scaling reaction time in large-scale fog computing systems compared to the centralized approach. Moreover, DESA resulted in a similar maximum number of instances and lower average CPU utilization during bursts of load.

Keywords: Web services in edge clouds, microservice autoscaling, service elasticity, container orchestration.

1 Introduction

Today, an unprecedented number of Web of Things (WoT) devices are deployed in our everyday environment. WoT devices such as smart cameras and traffic sensors in smart cities constantly generate huge amounts of data that require processing and analysis to extract useful information. Since resource-constrained WoT devices alone cannot handle such heavy computational tasks, attempts have been made to transfer the data to a distant global cloud server or local fog computing nodes, mitigating the communication overhead by placing computational resources closer to the end devices. as shown in Figure 1. Even though the distributed fog nodes have less computing power than the cloud, they effectively mitigate the communication overhead and congestion in the cloud by placing computational resources closer to the end devices [2]. In this system, along with the increase of end devices, the number of fog nodes in between the cloud and the end devices may also increase.

Meanwhile, in such fog computing systems, MicroService Architecture (MSA) are increasingly adopted and utilized for operating web applications, As an evolution of the conventional service-oriented architecture style, MSA

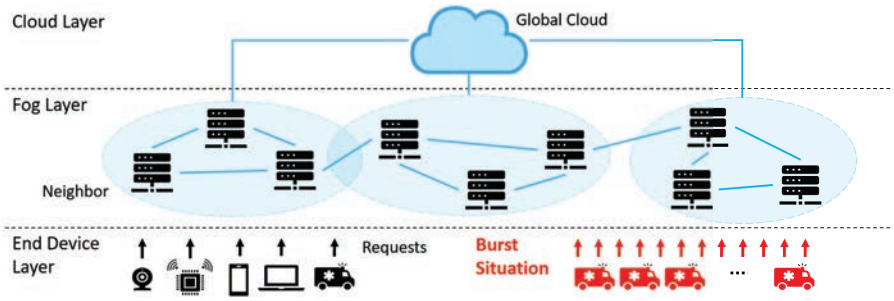


Figure 1 A fog computing system during a burst of load.

emphasizes dividing a system into small and lightweight services that each run its own process, which collaborate with each other with different roles. These microservices are often deployed in lightweight and portable containers [7, 8], which are monitored and managed through container orchestration tools such as Kubernetes.

A key challenge in the orchestration of microservices is service elasticity. Service elasticity indicates the ability of the microservices to be automatically scaled up or down corresponding to the computational demand [16]. For example, in a disaster situation, requests for emergency medical services may suddenly increase, leading to bursts of computational load at the related microservices, as shown in Figure 1. To handle such bursts of load, the load should be distributed across enough instances of the microservices, which means that the number of instances across the fog nodes should be temporarily increased in bursts of load [10, 14].

There mainly have been two approaches for the scaling of microservices: proactive and reactive scaling. In proactive scaling, prediction models predict the bursts of load in advance and perform scaling actions prior to the actual burst. However, these prediction models require reliable historical data for training and cannot predict unprecedented situations. On the other hand, reactive scaling approaches maintain global controllers that monitor the active microservice instances to make scaling decisions according to predefined rules. Although reactive scaling may be less efficient than proactive scaling for predictable bursts of load [14], reactive scaling is inevitably used for unpredictable bursts of load [14, 23].

However, reactively autoscaling during an unpredictable burst of load requires an expensive process of monitoring. Especially in fog computing, where a vast number of microservice instances are deployed over geographically distributed fog nodes, centralized monitoring-based autoscalers lack scalability as the number of fog nodes and microservices increases, due to the excessive networking cost of monitoring [20]. Furthermore, the scaling decisions may be delayed by the monitoring latency, which is aggravated as more fog nodes and microservices are deployed in the system. Therefore, rapidly reacting to unpredictable bursts of load has remained a challenge in realizing service elasticity in fog computing with a myriad of nodes.

As an extension of [17], we present DESA, a fully DEcentralized Self-adaptive Autoscaler, for handling bursts of load at microservices in fog computing environments. Instead of utilizing a centralized controller, each microservice instance regularly monitors its own metric value and analyzes the need for scaling by itself. When a burst of load occurs, the metric

value of each instance exceeds the upper threshold, and instances clone themselves to the current or a neighboring fog node. Then, when the burst fades away, cloned instances gradually terminate themselves as the metric values fall below the lower threshold. We design DESA based on the Monitor-Analyze-Plan-Execute (MAPE) model [6] for self-adaptiveness, with the MAPE control loop running in each microservice instance.

We evaluated the performance of DESA through a widely adopted network simulator, iFogSim2 [11]. We compared DESA with Horizontal Pod Autoscaler (HPA) as the baseline, which is the default autoscaler provided by Kubernetes. The simulation results show that DESA successfully reduces the scaling reaction time compared to the baseline approach by eliminating the delay caused by centralized controllers. Moreover, even without a centralized controller, DESA resulted in a lower average CPU utilization during bursts of load, generating similar numbers of microservice instances during bursts of load in general.

In addition, to evaluate DESA in a realistic scenario, we conducted a case study using the 1998 World Cup website access log. Compared to HPA, DESA more closely followed the needed amount of microservice instances to handle a burst of load, showing that DESA is more effective than HPA in handling a burst of load.

The main contributions of this work are summarized as follows:

- We extend the service elasticity problem to consider bursts of load in fog computing systems with increasing amounts of distributed fog nodes.
- We propose DESA, a fully decentralized self-adaptive autoscaler that reactively scales microservices across geographically distributed fog nodes in fog computing systems. DESA applies a fully decentralized MAPE loop for reactively scaling microservices.
- We evaluate the performance of DESA through iFogSim2, and experimental results show that our proposed autoscaler reacts faster to unpredictable bursts of load than Kubernetes' HPA, the de facto standard reactive autoscaler, reducing the overall network overhead.

The remainder of this paper is organized as follows. In Section 2, we discuss related works. In Section 3, we present our approach along with the problem definition. In Section 4, we show and discuss the results of our experimental evaluation and the case study. Finally, we present some concluding remarks in Section 5.

2 Related Works

Autoscaling and self-adaptive systems have been studied extensively by researchers of microservices. In this section, we discuss some existing proactive and reactive autoscaling approaches and introduce some works that utilize a decentralized self-adaptive system to handle autoscaling of microservices.

2.1 Proactive Autoscaling Approaches

Previous works propose proactive approaches in horizontally autoscaling microservices, predicting the load of microservices using machine-learning-based methods [9,13,23]. Ming et al. go further to create a hybrid of proactive and reactive approaches, handling the decision conflicts between the two approaches to optimizing the elastic expansion [23]. However, prediction models used in these works require reliable historical metric data for training. Moreover, proactive approaches heavily assume that bursts of load are predictable and can be handled prior to the burst, meaning that they can be domain-specific and not applicable to unexpected and unpredictable bursts of load.

2.2 Reactive Autoscaling Approaches

Kubernetes' HPA is the de facto standard for reactively autoscaling microservices [14]. For scaling purposes, a metric server collects resource metric data every 15 seconds, querying each node in the cluster. Then, a central controller fetches the metric data from the metric server and makes scaling decisions based on the threshold values that are manually set by the developer prior to deployment. While HPA is the most popular tool available on the market, the centralized metric server lacks scalability and suffers significant performance degradation as the number of fog nodes in the system increases.

Rossi et al. suggest a hierarchical autoscaling control model in which a centralized controller collects proactive or reactive scaling actions requested by decentralized microservice managers [19]. While this approach can successfully distribute the monitoring component of the autoscaling system, the centralized controller can still act as a bottleneck in fog computing systems, where instances of microservices are geographically distributed to multiple fog nodes.

2.3 Decentralized Self-Adaptive Systems

Brogi et al. propose a fully decentralized management of services, showing the effectiveness of decentralized orchestration of applications [3]. However, the work does not consider the reaction time in handling bursts of load in a large number of fog nodes, as their experiment involves only 85 fog nodes.

Tangari et al. apply a self-adaptive decentralized framework for monitoring network resources for Software-Defined Networking (SDN) [21]. Through this decentralization of the monitoring module, resources are dynamically reallocated to the services in the system. However, their approach requires a synchronization interface that may have significant overhead in fog computing environments. Moreover, since the work focuses on the specific domain of SDN, their method does not consider the characteristics of microservices that are required for scaling container-based microservices across fog nodes.

Zhang et al. propose a multi-level self-adaptation model for microservice scaling, extending the Kubernetes infrastructure to demonstrate its ability [25]. However, their work is only partially decentralized and can lack scalability in fog computing systems as more fog nodes enter the system.

3 Decentralized Self-Adaptive Horizontal Autoscaler

3.1 Problem Definition

Figure 2 shows the overview of our target environment. Our target environment is a fog computing system in which geographically distributed fog

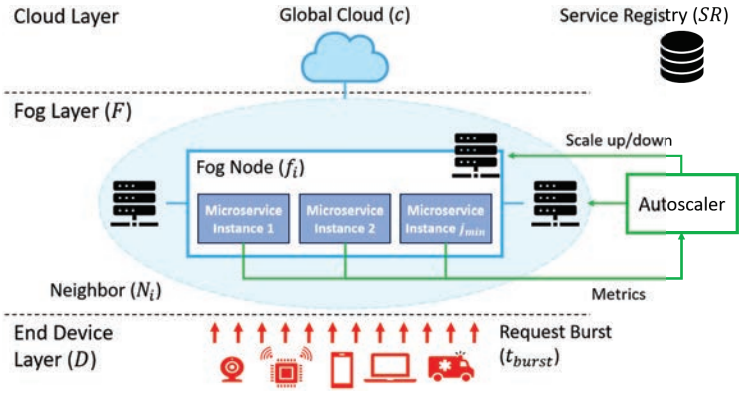


Figure 2 Overview of the target environment.

nodes, $f_i \in F$, are connected to the central cloud, c, with various network latencies, l_i, depending on their geographical location. Fog nodes are connected to each other, and each fog node maintains a list of its neighboring fog nodes ($N_i \subseteq F$) that have short communication latencies due to geographical locality. Instances of the microservice, $\mu_j \in I$, are placed across fog nodes upon deployment. The developer of the microservice sets the minimum number of instances, $j_{min} > 0$, before deployment, and j_{min} instances are deployed initially by the system. We define the total number of instances at a given time t as $j(t)$, with $j(0) = j_{min}$.

Various end devices, D, send data-processing requests, r, to the instances, I, that are placed across the system. Following the service-oriented architecture, a device initially queries a global service registry to establish the connection with an available microservice instance. The service registry assigns end devices to microservice instances in a round-robin manner to equally distribute the requests to the instances. Note that there is no single-point load balancer that dynamically balances the requests. Once the assignment is established, the end device sends requests to the assigned microservice instance until the processing is done or the device is handed over to another instance. Also, when scaling occurs, the service registry is updated accordingly to add the newly generated instances or delete the terminated instances.

A burst of load happens at an unpredictable point in time, t_{burst}, and stops at t_{end}. During the burst, the active microservice instances suffer from heavy networking and computing loads. We use CPU utilization of an instance at a given time t, $CPU_j(t)$, as the performance metric of the instances, which is the most common performance metric of computation-intensive applications [4].

The goal of this problem is to stabilize the performance metric of the system during the burst by dynamically scaling the microservices shortly after a burst. The autoscaler in the system makes the scaling decisions by periodically monitoring the metric values of the microservice instances. During a burst, the autoscaler should notice the burst and increase the number of microservice instances as soon as possible. Then, after the burst ends, the autoscaler should decrease the number of instances to reduce the cost of maintaining an excessive number of instances.

There are two concerns for the autoscaler: (1) minimizing the scale-up reaction time after a burst occurs, RTU, and (2) minimizing the average CPU utilization, CPU_{avg}, in an acceptable range during a burst. Firstly, we define the scale-up reaction time, RTU, as the time at which the burst of load has

been successfully handled. RTU is measured as the first time after a burst at which the average metric value of the entire microservice instances, I, fall below the upper threshold α, while each instance does not exceed the upper threshold by 10%, as follows:

$$RTU = \min\left(\left\{ t \mid \frac{1}{j(t)} CPU_j(t) \leq \alpha \text{ and } \forall \mu_j, CPU_j(t) \leq \alpha \times 1.1 \right\}\right)$$

where $t_{burst} < t \leq t_{end}$. Secondly, to measure the stabilization of the metric value resulting from scaling, we define the average CPU utilization of all microservice instances during the burst, CPU_{avg}, as follows:

$$CPU_{avg} = \frac{1}{t_{end} - t_{burst}} \sum_{t=t_{burst}}^{t_{end}} \frac{1}{j(t)} \sum_{\mu_j} CPU_j(t).$$

3.2 System Architecture & Algorithm

As a solution to the defined problem, we propose DESA, as shown in Figure 3. Instead of placing a global and centralized controller over the microservices, we place a self-adaptive and decentralized controller for each microservice instance. Through this controller, each microservice instance individually decides whether to scale up (cloning) or scale down (cooling). To implement the self-adaptiveness of DESA, we design DESA based on the Monitor-Analyze-Plan-Execute (MAPE) model [6] , with the MAPE control loop running in each microservice instance. Each step of the MAPE loop is represented as an independent component within an instance, with each component passing commands to the next component in a loop.

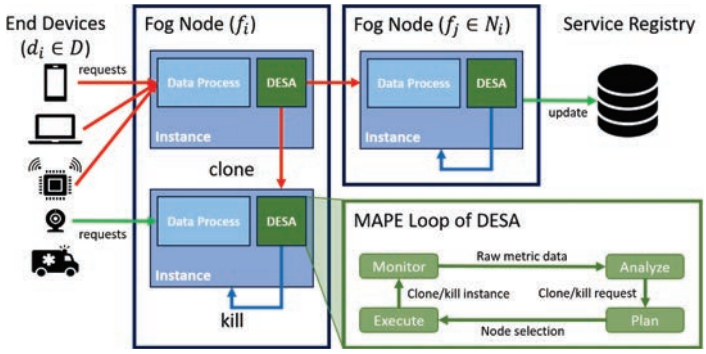


Figure 3 Overview of DESA's architecture.

Monitor In the monitor component, each microservice instance periodically collects the metric value (e.g. CPU utilization, memory utilization, workload) for every custom interval. The monitor component only fetches the metric value within the container holding the instance, which means that no network communication is involved in this monitoring process.

Analyze The analyze component receives the metric value from the monitor component and makes cloning or cooling decisions using threshold-based policies. DESA clones the instance as a child instance when the metric value exceeds the upper threshold (Algorithm 1). Then, when the burst of load has been relieved and the metric values fall below the lower threshold, child instances terminate themselves.The metric data provided by monitor are analyzed through threshold-based policies, making decisions to clone the microservice instance when the metric value exceeds the upper threshold. DESA label all newly generated instances as children, and each child instance holds its parent container's ID.

Algorithm 1 shows the cloning algorithm of DESA. The cloning algorithm of DESA is as follows, For every custom interval, the metric value is fetched from the monitor (line 4), and a cloning decision is made if the metric value exceeds the upper threshold, α (line 5). For cloning, the number of new child instances to be cloned, $newChildNum$, is decided similarly to HPA, using the ratio of the current and desired metric values to estimate the number of instances that are needed to lower the metric value below α (line 7–12). In case the instance already has child instances, however, the floor operation is used instead of the ceiling operation to limit the number of child instances newly generated by DESA. Then, actual cloning commands are sent to the plan component (line 14–15). Note that without any buffers, microservices may repeatedly alternate between cloning and cooling in a short period of time if the metric value is on the border of the threshold. Therefore, a different $cooltime$ is assigned to each child instance to make the cooling process gradual (line 16).

Plan When the plan component receives a cloning command from the analyze component, the fog node for deploying the clone is selected. If the current node has enough computing resources to place the cloned instance, the current node is selected. If the resource is insufficient, then the cloning command is forwarded to a neighboring node in a round-robin manner. When a neighboring node receives this forwarded command, it restarts this node selection process. Once a node with enough resources is found, the loop proceeds to the execute component.

Algorithm 1: Cloning algorithm

```
   Input  : α, monitoringInterval, monitor
 1 Procedure cloningAnalysis
 2     childNum ← 0
 3     for every monitoringInterval do
 4         metric ← monitor.getMetricValue()
 5         if metric > α then
 6             cooltime ← 0
 7             if childNum > 0 then
 8                 newChildNum ← ceil(metric ÷ α)
 9             end
10             else
11                 newChildNum ← floor(metric ÷ α)
12             end
13             childNum ← childNum + newChildNum
14             for steps ← 1 to newChildNum by 1 do
15                 this.clone(cooltime)
16                 cooltime ← cooltime + monitoringInterval
17             end
18         end
19     end
20 end
```

Execute With the selected node received from the plan component, the execute component makes the actual cloning or terminating request to the selected node. In the case of cloning, as the child instances are placed, some of the end devices connected to the parent instance are directly handed over and distributed across newly generated child instances without the end devices having to revisit the service registry. Finally, the service registry is updated with the network address of the newly generated instances.

4 Experimental Evaluation

We evaluated DESA by comparing the RTU, $max(j(t))$, and CPU_{avg} with the baseline in a simulated fog computing environment. For the simulations, we used iFogSim2, which is the most commonly used simulator of fog computing environments [11, 16]. The simulation code used for evaluation is available online.[1]

[1] https://github.com/pec9399/DESA

4.1 Simulation Setting

Table 1 depicts the settings of the simulations. The simulations were performed on a Windows 11 Pro machine with Intel i7-12700F 2.10GHz CPU and 32GB RAM. We chose HPA as the baseline since it is the de facto standard of centralized monitoring-based reactive autoscalers [14]. HPA utilizes a metric server in the cloud that collects the metric values from each fog node's monitor component. The central controller, also located in the cloud server, fetches the metric value from the metric server and makes scaling decisions for the microservices.

Generally, we set the parameters of the simulations based on the default settings of HPA. We set the upper threshold, α, of the CPU utilization metric to 50%, following a common threshold value of HPA. We set the monitoring interval of the autoscalers to 15 seconds, the default setting of HPA. The initial and minimum number of microservice instances was set to 4. Finally, the network latencies, l_i, from node f_i to cloud c, were set between 50 to 900ms, considering the distributed nature of fog nodes. We performed 50 independent simulations for the number of fog nodes from 100 to 5000 since the metric server used by HPA officially supports up to 5000 nodes. Note that the number of fog nodes in the system remained fixed for each simulation. Each simulation lasted for 15 minutes, and the simulations were repeated ten times.

In the simulations, the number of requests to a microservice suddenly increases at t_{burst}, which is set as a random time within the first five minutes of the simulation. As a result, the CPU of the initial instances exceeds α, and the autoscalers eventually scale the microservice in order to lower the CPU below α. We assumed that the fog nodes in our system, F, have enough computing resources to host the additional microservice instances generated by scaling.

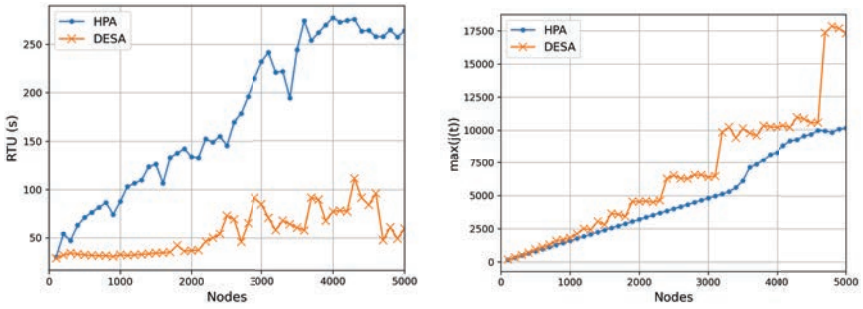
Table 1 Simulation parameters

Parameter	Value
Number of nodes	$100 \sim 5000$
Simulation time	15 minutes
Burst time (t_{burst})	Random within first 5 minutes
Burst duration ($t_{end} - t_{burst}$)	5 minutes
Min instances (j_{min})	4
Upper threshold (α)	50%
Monitoring interval	15 seconds
Network latency (l_i)	$50 \sim 900$ milliseconds

4.2 Result and Analysis

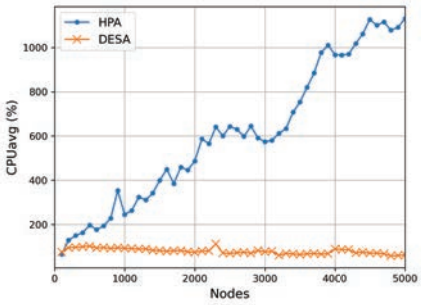
Figure 4a shows the scale-up reaction time, RTU, of HPA and DESA as the number of fog nodes increases. Specifically, DESA has 48.03% faster RTU from N = 100 to 1000, 71.38% for 1100 to 2000, 64.48% for 2100 to 3000, 71.17% for 3100 to 4000, and 71.55% for 4100 to 5000. The result shows that DESA reacts faster than HPA to bursts of load overall, especially as the number of fog nodes increases.

Figure 4b shows the maximum number of instances, $max(j(t))$, created during the burst of load as the number of fog nodes increases. The result shows that DESA produces similar $max(j(t))$ compared to the baseline, generating 23.92% more instances on average. Note that while $max(j(t))$ of DESA remained similar to that of HPA at the beginning, $max(j(t))$ of DESA had sudden increases at N = 1900, 2400, 3200, and 4700. This is because DESA's cloning algorithm nearly doubles the number of instances to react faster to bursts of load. However, without the exceptional cases, the



(a) Scale-up reaction time

(b) Max instances created



(c) Avg. CPU utilization

Figure 4 Experimental Result of DESA and HPA.

maximum number of instances generated during the burst does not overly exceed that of the baseline in general.

Figure 4c shows the CPU_{avg} during the burst as the number of fog nodes increases. Specifically, CPU_{avg} of DESA is 78.30% lower than that of HPA on average. This indicates that DESA more successfully handles the burst of load than HPA by reacting faster to the burst of load.

In summary, by reducing the network overhead caused by a centralized controller, DESA reacted faster to bursts of load while resulting in similar maximum number of instances and lower average CPU utilization during bursts of load.

4.3 Case Study: 1998 World Cup Website Access Log

For the simulations, the bursts of load were statically given to the microservice such that the metric value exceeds the upper threshold, as stated in the problem definition. However, to further evaluate DESA in a real-world scenario, we used the 1998 World Cup website access log to represent the burst of load. Specifically, we used the requests received by the website between 18:00 to 24:00 of July 7, 1998 (GMT+2 timezone). This time was chosen since the website received the highest number of requests per minute during this period.

To speed up the simulation, the number of requests made per minute to the 1998 World Cup website was instead sent every second to the microservice instances in our system, lasting for a duration of six minutes instead of six hours in total. During the six-minute period, we measured the number of instances generated by both autoscalers and compared the results with the number of instances actually needed to keep the metric value below the upper threshold. The actually needed number of instances is estimated by linearly regressing HPA's $max(j(t))$. For the case study, we set the number of nodes to 1000, j_{min} to 4, and α to 50%.

Figure 5 shows the number of microservice instances generated by each autoscaler during the simulation, and the green line represents the number of actually needed instances to keep the metric value below α. Note that cooling does not occur during the six-minute period.

The results show that DESA generated instances closer than HPA to the actually needed amount of instances. Specifically, up to t = 222, the time at which the number of requests reaches the maximum, DESA generated on average 70.64% of the actually needed amount of instances while HPA only generated 37.17%. This means that DESA is more effective in reactively handling unpredictable bursts of load in the real world.

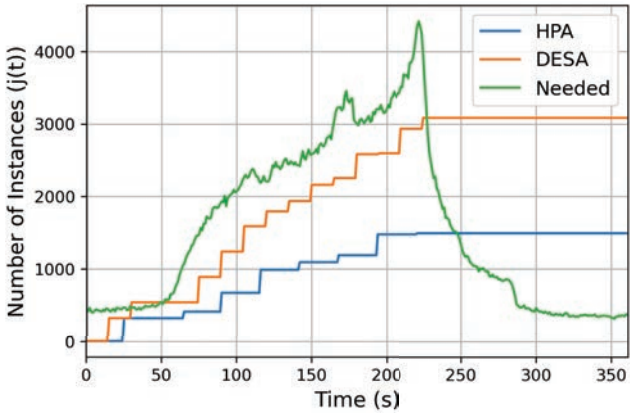


Figure 5 Case study using 1998 World Cup access log.

4.4 Threats to Validity

We used iFogSim2 [11, 16] to establish the external validity of the simulations. Because there is no implementation of HPA in iFogSim2, we implemented both DESA and HPA in iFogSim2 by ourselves, translating the components of DESA and HPA as application modules in iFogSim2. Our implementation includes all the components in HPA so that the scaling behavior closely follows the actual behavior of HPA.

DESA requires developers to choose appropriate parameter settings such as the upper (α) and lower (β) thresholds of the metric values, and the performance of DESA can be affected by such parameters. While we empirically chose reasonable values of these parameters for our simulations, we plan to perform in-depth experiments to analyze the influence of these parameters on DESA's performance in future work.

Note that we disregard the container startup time in measuring RTU. Therefore, in an actual system, RTU is expected to be slightly higher than the simulation result for both DESA and HPA. However, this increase is expected to be constant for both autoscalers, which means that disregarding container startup time does not influence the comparison of the two approaches.

Furthermore, for DESA, the MAPE loop in an instance should begin once the instance is cloned. This means that although the monitoring interval is set to equal values for all instances, instances that are cloned at different times should have different starting points for monitoring. However, due to the limitation of the simulator, the MAPE loop begins from the same point in time for all the microservice instances. However, we expect that having

different starting points of monitoring will have a minor influence on the scaling reaction time and overall performance of scaling.

Also due to the limitation of the simulator, we disregard the computational overhead of the MAPE control loop running in each microservice instance. This overhead that exists in DESA, however, is similar in amount to the computational overhead of HPA. The difference exists in that DESA distributes this overhead across multiple instances while HPA handles the computation in the centralized controller. Therefore, we expect that this overhead does not influence the result of our work.

4.5 Discussion and Future Work

Although DESA successfully reduces the RTU after a burst occurs, scaling down after the burst is designed to be gradual, which means that the scale-down process requires more computing resources to be occupied briefly after the burst has ended. This is to ensure that child instances are not terminated simultaneously, which may result in repeated fluctuation of the metric value such that instances are repeatedly scaled up and down within a short period of time.

Also, it is typical for developers to set an upper limit for the number of instances to negotiate the operational costs during bursts. To apply this limit, the autoscaler should be aware of the total number of instances that are currently active. However, since DESA is fully decentralized, the total number of active instances can only be heuristically estimated, which is the reason that DESA makes more instances than HPA in general. We leave the development of the method for limiting the number of total instances as future work.

We plan to expand DESA to consider the geographic locality of the requests from the end devices. To fully exploit the advantage of geographically distributed fog nodes, DESA may detect a burst of load in a specific region and clone microservice instances to the fog nodes close to the requesting end devices in the region. In that case, neighboring fog nodes would be defined in terms of geographical locations and may cooperate to make scaling decisions systematically.

In order to expand DESA to consider the geographic locality, the node selection process needs to be carefully considered. Currently, when the current node cannot place the cloned instance, DESA uses a simple round-robin method to select an alternative neighboring node. However, if all the resources in N_i are occupied and cannot host any additional microservice instance, the cloning command may repeatedly hover around the fog layer

before finally reaching the cloud. While this limitation was not focused on in this work, we plan to consider such resource-constrained scenarios in selecting the appropriate neighboring node in future work.

5 Conclusion

Establishing service elasticity when microservices experience unpredictable bursts of load has been a challenge that must be addressed, especially for large-scale fog computing systems. Since these bursts of load are unpredictable, existing proactive approaches are insufficient. Furthermore, current reactive autoscalers lack scalability in fog computing systems due to their centralized components. In this work, we present DESA, a fully decentralized self-adaptive autoscaler that rapidly scales microservice instances across geographically distributed fog nodes. With a MAPE loop running in each microservice instance, DESA performs scaling of microservices without a centralized controller, reducing the network overhead of existing reactive autoscalers. Evaluation results show that DESA has a faster scale-up reaction time to bursts of load, especially as the number of fog nodes in the system increases. Moreover, DESA leads to a similar maximum number of instances and a lower average CPU utilization during a burst compared to the baseline, showing a comparable performance of scaling despite its decentralized nature. The case study also demonstrates DESA's effectiveness compared to HPA in handling a realistic burst of load.

We plan to expand DESA to consider the geographic locality of the requests from the end devices. To fully exploit the advantage of geographically distributed fog nodes, DESA may detect a burst of load in a specific region and clone microservice instances to the fog nodes close to the requesting end devices in the region. In that case, neighboring fog nodes would be defined in terms of geographical locations and may cooperate to make scaling decisions systematically. We will also expand upon the node selection process to consider scenarios in which regional clusters are resource-constrained.

Acknowledgements

This research was supported by the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program (IITP-2023-2020-0-01795) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation)

References

[1] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51, 2016.

[2] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, page 13–16, New York, NY, USA, 2012. Association for Computing Machinery.

[3] Antonio Brogi, Stefano Forti, Carlos Guerrero, and Isaac Lera. Declarative application management in the fog: A bacteria-inspired decentralised approach. *Journal of Grid Computing*, 19(4):45, 2021.

[4] Emiliano Casalicchio and Vanessa Perciballi. Auto-scaling of containers: The impact of relative and absolute metrics. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 207–214, 2017.

[5] Byungkwon Choi, Jinwoo Park, Chunghan Lee, and Dongsu Han. Phpa: A proactive autoscaling framework for microservice chain. In *5th Asia-Pacific Workshop on Networking (APNet 2021)*, APNet 2021, page 65–71, New York, NY, USA, 2022. Association for Computing Machinery.

[6] Autonomic Computing et al. An architectural blueprint for autonomic computing. *IBM White Paper*, 31(2006):1–6, 2006.

[7] Corentin Dupont, Raffaele Giaffreda, and Luca Capra. Edge computing in iot context: Horizontal and vertical linux container migration. In *2017 Global Internet of Things Summit (GIoTS)*, pages 1–4, 2017.

[8] Bukhary Ikhwan Ismail, Ehsan Mostajeran Goortani, Mohd Bazli Ab Karim, Wong Ming Tat, Sharipah Setapa, Jing Yuan Luke, and Ong Hong Hoe. Evaluation of docker as edge computing platform. In *2015 IEEE Conference on Open Systems (ICOS)*, pages 130–135, 2015.

[9] Li Ju, Prashant Singh, and Salman Toor. Proactive autoscaling for edge computing systems with kubernetes. In *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, UCC '21, New York, NY, USA, 2022. Association for Computing Machinery.

[10] Manuel Ramírez López and Josef Spillner. Towards quantifiable boundaries for elastic horizontal scaling of microservices. In *Companion*

Proceedings of the10th International Conference on Utility and Cloud Computing, pages 35–40, 2017.

[11] Redowan Mahmud, Samodha Pallewatta, Mohammad Goudarzi, and Rajkumar Buyya. Ifogsim2: An extended ifogsim simulator for mobility, clustering, and microservice management in edge and fog computing environments, 2021.

[12] Ismael Martinez, Abdallah Jarray, and Abdelhakim Senhaji Hafid. Scalable design and dimensioning of fog-computing infrastructure to support latency-sensitive iot applications. *IEEE Internet of Things Journal*, 7(6):5504–5520, 2020.

[13] Hoa X. Nguyen, Shaoshu Zhu, and Mingming Liu. Graph-phpa: Graph-based proactive horizontal pod autoscaling for microservices using lstm-gnn. In *2022 IEEE 11th International Conference on Cloud Networking (CloudNet)*, pages 237–241, 2022.

[14] João Nunes, Thiago Bianchi, Anderson Iwasaki, and Elisa Nakagawa. State of the art on microservices autoscaling: An overview. In *Anais do XLVIII Seminário Integrado de Software e Hardware*, pages 30–38, Porto Alegre, RS, Brasil, 2021. SBC.

[15] 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'19, page 71–81, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Samodha Pallewatta, Vassilis Kostakos, and Rajkumar Buyya. Microservices-based iot applications scheduling in edge and fog computing: A taxonomy and future directions, 2022.

[17] EunChan Park, KyeongDeok Baek, Eunho Cho, and In-Young Ko. Desa: Decentralized self-adaptive horizontal autoscaling for bursts of load in fog computing. In *3rd International Workshop on Big data driven Edge Cloud Services (BECS 2023) Co-located with the 23rd International Conference on Web Engineering (ICWE 2023)*, 2023.

[18] Fabiana Rossi, Valeria Cardellini, and Francesco Lo Presti. Hierarchical scaling of microservices in kubernetes. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 28–37, 2020.

[19] Fabiana Rossi, Valeria Cardellini, and Francesco Lo Presti. Self-adaptive threshold-based policy for microservices elasticity. In *2020 28th International Symposium on Modeling, Analysis, and Simulation*

of Computer and Telecommunication Systems (MASCOTS), pages 1–8, 2020.

[20] Salman Taherizadeh, Andrew C. Jones, Ian Taylor, Zhiming Zhao, and Vlado Stankovski. Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review. *Journal of Systems and Software*, 136:19–38, 2018.

[21] Gioacchino Tangari, Daphne Tuncer, Marinos Charalambides, Yuanshunle Qi, and George Pavlou. Self-adaptive decentralized monitoring in software-defined networks. *IEEE Transactions on Network and Service Management*, 15(4):1277–1291, 2018.

[22] Massimo Villari, Maria Fazio, Schahram Dustdar, Omer Rana, and Rajiv Ranjan. Osmotic computing: A new paradigm for edge/cloud integration. *IEEE Cloud Computing*, 3(6):76–83, 2016.

[23] Ming Yan, XiaoMeng Liang, ZhiHui Lu, Jie Wu, and Wei Zhang. Hansel: Adaptive horizontal scaling of microservices using bi-lstm. *Applied Soft Computing*, 105:107216, 2021.

[24] Jiawei Zhang, Xiaochen Zhou, Tianyi Ge, Xudong Wang, and Taewon Hwang. Joint task scheduling and containerizing for efficient edge computing. *IEEE Transactions on Parallel and Distributed Systems*, 32(8):2086–2100, 2021.

[25] Shuai Zhang, Mingjiang Zhang, Lin Ni, and Peini Liu. A multi-level self-adaptation approach for microservice systems. In *2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, pages 498–502, 2019.

Biographies

EunChan Park is enrolled in the Master's program at the Web Engineering and Service Computing Lab, School of Computing, Korea Advanced Institute of Science and Technology (KAIST). His research interests include service computing, edge computing, and container orchestration.

KyeongDeok Baek is enrolled in the integrated program for Master's and Ph.D. degrees of the Web Engineering and Service Computing Lab, School of Computing, Korea Advanced Institute of Science and Technology (KAIST). He received his Bachelor's degree from the School of Computing at KAIST. His research interests include human-centric service-oriented computing, Internet of Things services, service selection, and reinforcement learning.

Eunho Cho is Ph.D. Candidate of Web Engineering and Service Computing Lab, School of Computing, Korea Advanced Institute of Science and Technology (KAIST). He received his Master's degree from the School of Computing at KAIST. His research interests include AI4SE, simulation-based testing, autonomous driving system testing, self-adaptive system and cyber-physical system.

In-Young Ko is a professor at the School of Computing at the Korea Advanced Institute of Science and Technology (KAIST) in Daejeon, Korea. He received the Ph.D. in computer science from the University of Southern California (USC) in 2003. His research interests include services computing, web engineering, and software engineering. His recent research focuses on service-oriented software development in large-scale and distributed system environments such as the web, Internet of Things (IoT), and edge cloud environments. He is a member of the IEEE.