Container Deployment Strategy for Edge Networking

Walter Wong
University of Helsinki
Helsinki, Finland
walter.wong@helsinki.fi

Aleksandr Zavodovski
University of Helsinki
Helsinki, Finland
aleksandr.zavodovski@helsinki.fi

Pengyuan Zhou
University of Helsinki
Helsinki, Finland
pengyuan.zhou@helsinki.fi

Jussi Kangasharju
University of Helsinki
Helsinki, Finland
jussi.kangasharju@helsinki.fi

Abstract

Edge computing paradigm has been proposed to support latency-sensitive applications such as Augmented Reality (AR)/ Virtual Reality(VR) and online gaming, by placing computing resources close to where they are most demanded, at the edge of the network. Many solutions have proposed to deploy virtual resources as close as possible to the consumers using virtual machines and containers. However, the most popular container orchestration tools, e.g., Docker Swarm and Kubernetes, do not take into account the locality aspect during deployment, resulting in poor location choices at the edge of the network. In this paper, we propose an edge deployment strategy to tackle the lack of locality awareness of the container orchestrator. In this strategy, the orchestrator collects information about latency and the real-time resource consumption from the current container deployments, providing a bird's-eye view of the most demanded locations and the best places for deployment to cover the largest number of clients. We evaluated the proposed model using 16 AWS regions across the globe and compared to the standard deployment strategies. The experimental results show our edge strategy reduces the average latency between serving container to the clients by up to 4 times compared to the standard deployment algorithms.

CCS Concepts • **Computer systems organization** → **Cloud computing**; • **General and reference** → *Empirical studies*; *Evaluation.*

Keywords Edge computing, Containers, Deployment, Scheduling

ACM Reference Format:
Walter Wong, Aleksandr Zavodovski, Pengyuan Zhou, and Jussi Kangasharju. 2019. Container Deployment Strategy for Edge Networking. In *4th Workshop on Middleware for Edge Clouds & Cloudlets (MECC '19), December 9–13, 2019, Davis, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3366614.3368101

1 Introduction

Smartphones bring a broad variety of applications to the end user including online gaming, image recognition, video streaming, and other interactive applications. However, those devices are not always well suited to all range of applications due to their limits on the amount of processing, storage, and battery. Solutions like offloading tasks to the cloud [8] were proposed to alleviate this problem, but they are not applicable for delay-sensitive applications such as online gaming and augmented reality (AR) and virtual reality (VR), which depend on low network latency and high bandwidth to operate seamlessly. To even aggravate the situation, the Internet is going to have even more pressure on the bandwidth: according to [6], there are currently 4.3 billion smartphone subscriptions, and the mobile data traffic has grown 54% since last year. By 2023, it is expected there will be 3.5 billion IoT connections and over 1 billion 5G subscriptions where 20% is going to be all mobile data traffic. This expected growth will put a lot of pressure on the cloud services in order to meet the increasing delay-sensitive applications.

The edge computing paradigm has been discussed as an emerging solution for low latency applications. At the core of the concept, the main idea is to bring the computing resources closer to where the users are, at the edge of the network, reducing the latency to the end-users and the transit bandwidth consumed in the path. This concept has been explored previously [7], and some solutions were proposed, e.g., Cloudlets [13] and Foglets [14] and even a small cluster of limited devices [9]. Many studies have already shown the benefits of edge computing, e.g., [15, 17] and most proposed solutions use some kind of virtualization, either virtual machines [13] or containers [11, 16] in their solutions. Despite the fact of both virtualization and containerization are used as a technological option to deploy edge networks, it is seldom discussed how they are used to deploy and orchestrate multiple virtual machines and containers at the edge of the network. The standard deployment of container management frameworks such as Docker Swarm [2] and Kubernetes [5] do not take into account the locality issue, which is a critical part and one of the main drivers of deploying at the edge. One of the issues is that the container management systems were built to be run in datacenters rather than at the edge of the network, thus, locality is not that critical as all containers run in the same datacenter site. However, the impact of a poor location selection can result in a high impact on the latency to end-users at the edge.

In this paper, we present a new scheduling strategy for container orchestration called *edge scheduling strategy (ESS)*. The ESS edge scheduling strategy aims to assist the container orchestrators to take a better decision on where to deploy containers by taking additional metrics such as response time or underlying network latency as input for the deployment decision. ESS adds two new components to the Docker Swarm scheduling process, the *classification* and the *edge scheduling* steps. The classification component is responsible for going through all container deployment requests and prioritizing them according to the metrics defined by the user. The edge scheduling component is responsible for receiving monitoring information and scheduling containers to be deployed in the best available location closer to the user. The benefits are twofold: first, it prioritizes requests that are most sensitive to the user according to the metrics defined previously, e.g., the user can define that she wants to scale out the next container where there is more demand; and second, the latency between the client and the node where the container is deployed to is going to be the lowest possible. As a result, the proposed model will always allocate the containers as close as possible to the end-user in the edge of the network.

We implemented a prototype of the proposed ESS scheduling strategy and monitoring components and evaluated them against the standard deployment strategies on Docker Swarm in two scenarios: (i) average latency vs. deployment strategy and (ii) percentage of requests that was addressed in the lowest latency possible. The experimental results show that (i) the overall latency using the new strategy is up to 5 times lower than other strategies in the best scenario and (ii) the percentage of requests handled in the lowest latency possible (under 25ms) is 50% higher compared to other strategies.

2 Background

The container lifecycle management is composed of two phases: (i) *selection* phase and *scheduling* phase. First, all scaling requests are placed in a FIFO queue. Then, in the selection phase, Docker Swarm selects all nodes that meet the user resource requirements, e.g., CPU, memory, storage, and places in a node candidate list and passes it to the filtering phase. Upon completing the filtering phase, Docker Swarm has a list of candidates to where the containers can be deployed and applies one of the following container deployment strategies to place containers: *binpack*, the scheduler will select the most loaded node from the list to place the container; *spread*, the scheduler will select the node with the least number of replicas of the container; and *random*, the scheduler will randomly pick one node to deploy the container.

Although these strategies offer some flexibility for the user to choose where to deploy the container, they are limited to provide the best deployment location in the context of edge computing. For example, consider a multiplayer game with low-latency requirements (e.g., AV/VR based) at the edge of the network, and it needs to scale in order to keep up with the increasing number of users at the edge. If the developer selects *spread*, the container with the application will end up in a different location as the policy is to provide high-availability. If the developer selects *binpack*, the container will be deployed in the busiest location, but we cannot assume that at that moment the player is in the busiest location.

We list the following reasons for those limitations: first, the main aim of those strategies are either (i) minimize costs by reducing the number of running nodes (binpack) or (ii) availability to have multiple replicas of the same service (spread). Second, those deployment strategies were devised in the context of running in datacenter, where the locality component is not a problem as all nodes will be running in the same datacenter site or region. Third, all container scheduling requests are placed in a FIFO queue, thus, the containers are deployed based on the arrival order without any classification or prioritization of the requests. Although the user can request the Docker engine to place certain containers to selected locations, it is done manually with labels, and not automated and as part of the scheduling and deployment strategy. In contrast to the current deployment strategies, we propose a mechanism to take the locality and classification of the request as input for the deployment decision for the edge networking, which we will describe in the next section.

3 System Design

In this section, we present the *ESS* strategy to assist the Docker container orchestrator in overcoming the current limitations on the edge deployment. The first add-on to the scheduler is the *classification* component shown in Fig. 1. This component is responsible for receiving all scaling requests and ranking them according to the user metrics. The ranking process starts with the definition of metrics to be used as criteria by the user, e.g., CPU level or latency between server and client, then retrieving the container's monitored metric and, lastly, ordering them according to the metric defined previously.

The second component is the *edge scheduling*, and it is responsible for selecting the closest available locations to the clients to deploy containers. This component periodically probes the Swarm nodes to get the relative latency between the cluster nodes and stores the information in a *neighbor table*. The neighbor table contains a list of all nodes and the corresponding latency to each other node, sorted by the lowest

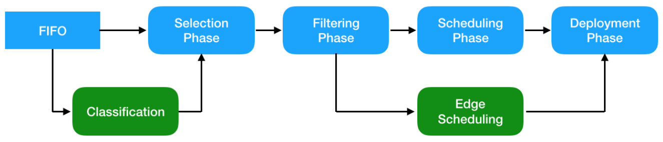


Figure 1. Docker container creation pipeline.

latency (closest neighbor) to the farthest. During the scheduling step, the edge scheduler will select the node's location that is closer to the client to deploy a container. However, in case the node is already full, it will look at the neighbor table to get the second closest node to the original location and so forth. Therefore, the scheduler will always try to deploy to the closest available location to the client based on the latency.

Algorithm 1: Container deployment strategy

```
1  Input: Multiple scaling out request for apps
2  for each scaling request for $app_i$ do
3      $metric_i$ ← Get monitoring metric for $app_i$
4  ordered_app_list ← sort(get_app($metric_i$))
5  ordered_location ← sort(ordered_app_list)
6  selected_node_list ← get all nodes from the Swarm
     cluster
7  filtered_node_list ← filter(selected_node_list)
8  not_deployed_container ← True
9  while not_deployed_container do
10     candidate_list ← ordered_location ⋈
         filtered_node_list
11     for each $location_i$ in candidate_list do
12         if $container\_count_i < node\_limit_i$ then
13             Deploy container at $location_i$
                 not_deployed_container ← False
14             break
15         else
16             for each $neighbor_j$ in neighbor table do
17                 if $container\_count_j < node\_limit_j$ then
18                     Deploy container at $neighbor_j$
                         not_deployed_container ← False
                         break
```

Algorithm 1 shows all steps involved in the scaling out process. First, the classifier component receives all scaling out requests and holds up for *t* seconds. This time is set to be equal to the cluster resource monitoring and at Docker Swarm is set to 15s. The *withholding* interval is defined as the same as the monitoring interval, and the default value is set to 10s. For each incoming request, the classifier will query the monitoring database to get the metric defined by the user and sort it from the most to the less violated compared to the base metric (steps 2-4). After that, it will get the container list and the underlying node list that has the application with the violated metric (*ordered_location*) (step 5). Then, the request goes to the usual select and filter phases from Docker until it reaches the *edge scheduler* component (steps 6-8). This component gets the list of nodes with most metric violations and executes an inner join with the filtered list from Docker (step 11) and sends it over to the container scheduler to deploy the container at that location (step 12-15). In case the preferred location is already full, it will use the *neighbor table* to find the ordered list of locations to deploy the container (steps 17-20).

4 Implementation

Fig. 2 shows the implementation components of the *ESS* strategy, dividing into the monitoring and scheduling subsystems. The monitoring system has three components *cAdvisor*[3], *Prometheus*[4] and *AlertManager*[1]. cAdvisor is a data collecting tool from Google to collect both container and host information. Under the hood, it is a daemon that collects all stats such as CPU usage, networking, and storage from all containers and also machine-wide and parses it to be exported to Prometheus. Prometheus is a monitoring tool that collects resource data from all monitored endpoints and applies a set of user-defined rules on them, such as triggering alerts in case some metrics are passed through. The last component is AlertManager, which main goal is to aggregate and hold back a series of alerts triggered by the same event, e.g., degraded response time due to service overload. Prometheus will receive this information and send a continuous series of alerts informing that this given threshold has been broken. Without the AlertManager filtering, these alerts will trigger multiple scaling out requests for the same root cause and also without giving time for the system to react to the scaling process.

In the scheduling subsystem, the *classifier* component periodically queries the Prometheus database for usage metrics and keeps a sorted list of container locations with more requests in memory. The query period is set to be the same as the monitoring system (in our case, the default is 10s) so the component can keep the last data delayed by only 10 seconds and the overhead be low. The *edge scheduler* component periodically queries the nodes to retrieve the relative latency between each node and populates the *neighbor table.*

5 Evaluation

Fig. 3 shows the AWS inter-region latency (in milliseconds). The latency was collected by pinging each public endpoint on the AWS Dynamo database for over 1 month, and the average latency was taken. For all evaluation, we used 16 VMs to represent each AWS region and used Docker Swarm to cluster the nodes together. The AWS machine size was *t2.micro* with 1 vCPU and 1 GB of RAM. On the application level, we used a Docker image containing an HTTP server to be served as an endpoint for our data retrieval and latency computation experiment. The image required 35 MB of RAM to run, and

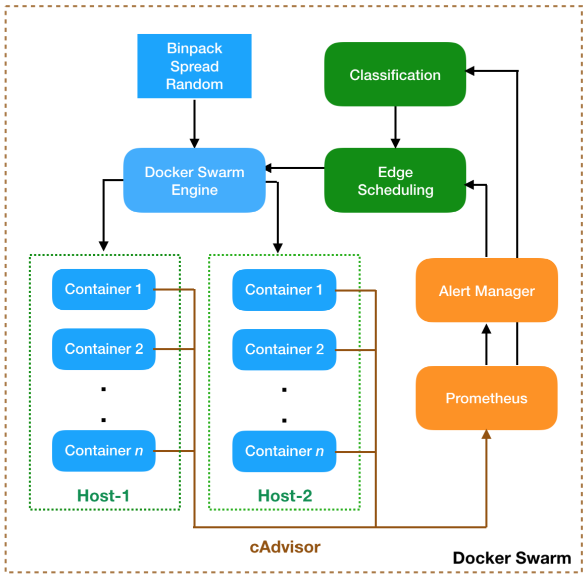
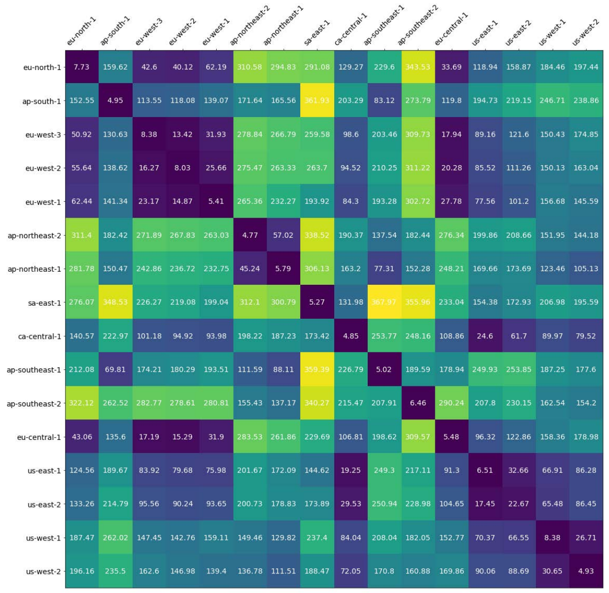


Figure 2. Implementation design of *edge* strategy with the *classifier* and *edge scheduler* components.



	eu-north-1	ap-south-1	eu-west-3	eu-west-2	eu-west-1	ap-northeast-2	ap-northeast-1	sa-east-1	ca-central-1	ap-southeast-1	ap-southeast-2	eu-central-1	us-east-1	us-east-2	us-west-1	us-west-2
eu-north-1	7.73	159.62	42.6	40.12	62.19	310.58	294.83	291.08	129.27	229.6	343.53	33.69	118.94	158.87	184.46	197.44
ap-south-1	152.55	4.95	113.55	118.08	139.07	171.64	165.56	361.93	203.29	83.12	273.79	119.8	194.73	219.15	246.71	238.86
eu-west-3	50.92	130.63	8.38	13.42	31.93	278.84	266.79	259.58	98.6	203.46	309.73	17.94	89.16	121.6	150.43	174.85
eu-west-2	55.64	138.62	16.27	8.03	25.66	275.47	263.33	263.7	94.52	210.25	311.22	20.28	85.52	111.26	150.13	163.04
eu-west-1	62.44	141.34	23.17	14.87	5.41	265.36	232.27	193.92	84.3	193.28	302.72	27.78	77.56	101.2	156.68	145.59
ap-northeast-2	311.4	182.42	271.89	267.83	263.03	4.77	57.02	338.52	190.37	137.54	182.44	276.34	199.86	208.66	151.95	144.18
ap-northeast-1	281.78	150.47	242.86	236.72	232.75	45.24	5.79	306.13	163.2	77.31	152.28	248.21	169.66	173.69	123.46	105.13
sa-east-1	276.07	348.53	226.27	219.08	199.04	312.1	300.79	5.27	131.98	367.97	355.96	233.04	154.38	172.93	206.98	195.59
ca-central-1	140.57	222.97	101.18	94.92	93.98	198.22	187.23	173.42	4.85	253.77	248.16	108.86	24.6	61.7	89.97	79.52
ap-southeast-1	212.08	69.81	174.21	180.29	193.51	111.59	88.11	359.39	226.79	5.02	189.59	178.94	249.93	253.85	187.25	177.6
ap-southeast-2	322.12	262.52	282.77	278.61	280.81	155.43	137.17	340.27	215.47	207.91	6.46	290.24	207.8	230.15	162.54	154.2
eu-central-1	43.06	135.6	17.19	15.29	31.9	283.53	261.86	229.69	106.81	198.62	309.57	5.48	96.32	122.86	158.36	178.98
us-east-1	124.56	189.67	83.92	79.68	75.98	201.67	172.09	144.62	19.25	249.3	217.11	91.3	6.51	32.66	66.91	86.28
us-east-2	133.26	214.79	95.56	90.24	93.65	200.73	178.83	173.89	29.53	250.94	228.98	104.65	17.45	22.67	65.48	86.45
us-west-1	187.47	262.02	147.45	142.76	159.11	149.46	129.82	237.4	84.04	208.04	182.05	152.77	70.37	66.55	8.38	26.71
us-west-2	196.16	235.5	162.6	146.98	139.4	136.78	111.51	188.47	72.05	170.8	160.88	169.86	90.06	88.69	30.65	4.93

Figure 3. AWS Inter-region latencies (in ms). Rows represent the source AWS region, and columns are the destination regions.

each AWS VM could handle up to 30 containers (each VM had 1 GB of RAM). At the beginning of the experiment, one HTTP container application would be randomly deployed at one of the Docker cluster nodes. As more client requests arrive at the serving container, the monitoring system will detect the application overload (e.g., high CPU usage) and trigger Docker daemon to scale out and deploy a new container. The container deployment location is going to be based on the configured deployment strategy, e.g., binpack, spread, etc.

In the subsequent experiments, we used between 4 to 16 regions on AWS as the source region of the clients. The choice of the client region follows the gravity model[12] which states that the network traffic generated between two regions is proportional to the population of each region. We used Internet user demographics data[1] as a parameter to the probabilistic distribution to select one of the given AWS regions. Regions with higher population density have a higher probability to be chosen.

5.1 Experimental Evaluation

In the first experiment, we measured the latency using different deployment strategies shown in Fig. 4. The aggregated latency was calculated by getting the latency between each client to the serving container region on AWS and getting the 90*th* percentile of the measured values. The experiment was run 1,000 times, and the 90*th* percentile was calculated for each strategy. The results show that for a filling of for 25%, 50% and 75% in the Swarm cluster, the gains were 300%, 100% and 30% of improvement in the scenario with only 4 client regions, respectively. The gains increase with the number of clients as the probability of a client falling in the same region as the server increases with the number of regions.

The results show the aggregated latency is smaller using the edge strategy and the reasons are twofold: (i) ESS will select the best available AWS region to deploy the container (or any options in the neighbor list) reducing the overall latency and (ii) the edge strategy responds to the number of clients rather than being static as the other strategies. The latency curve also increases slowly due to the *neighbor table* as it provides the closest available locations to the client. Moreover, the aggregate latency in ESS also drops with the increase in the number of client regions because the fraction of the clients served with the best option is weighted by the number of clients. For the scenario with only 1 client region, the latency of edge strategy will also converge to the same value of the other strategies as the Swarm capacity is full.

The second experiment shows the latency distribution of all client requests vs. the scheduling strategy. The goal is to understand what ratio of the traffic is handled with low latency for each scheduling strategy. The experiment was run 1,000 times, and the data was divided into the intervals shown in Fig. 5. ESS is able to handle up to 27% of all requests within 25 ms with 4 client regions while the other three strategies can only handle up to 9% of all requests. This result is expected as the ESS strategy selects the most popular and closest node to the client. In addition to that, the percentage of requested handled within 25 ms increases with the number of clients due to the higher probability of a given client region being served in the same region by a container as the edge strategy

[1]https://www.internetworldstats.com/stats.htm

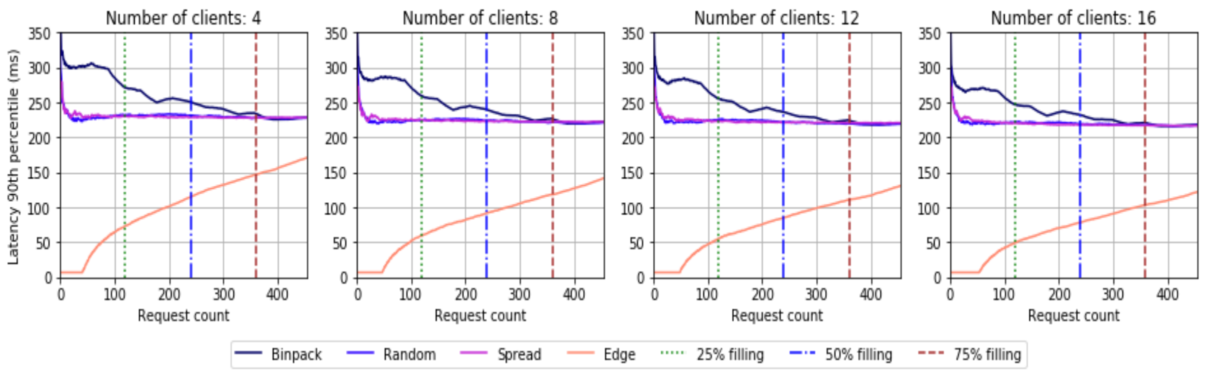


Figure 4. Client to server latency (90*th* percentile) vs. container deployment strategy. Vertical lines show the Swarm filling percentage.

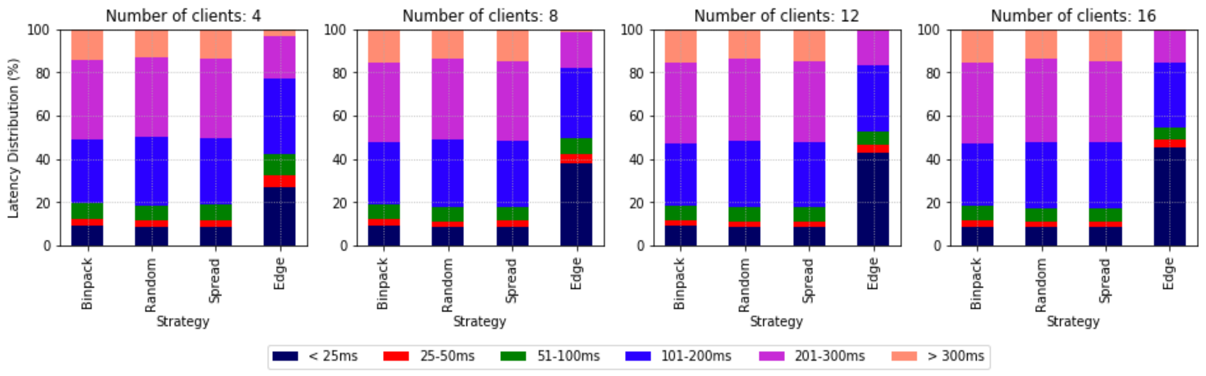


Figure 5. Latency distribution vs. deployment strategy. The edge strategy is able to handle more requests with low latency compared to the other algorithms due to the most frequently requested location metric.

spreads the containers following the client request pattern. In that scenario, edge strategy handles 37%, 43% and 46% for 8, 12 and 16 clients respectively. In the best scenario, the edge strategy can outperform by up to 5 times compared to the other algorithms. Comparing the percentage of traffic that different strategies can handle under 100 ms, the edge strategy can cover up to 40% (or up to 50% if more than 8 clients) while all other strategies can cover only up to 20% of the requests.

Note that the tests on AWS regions aim to get the latency between the regions to be used as a benchmark for our evaluation. Although the inter-region latency on AWS will be higher compared to a real edge deployment, we are only interested in the ratio between the inter-region latency to be used as input for the deployment algorithm comparison. We do not expect edge deployments to be done purely in the cloud, but use these numbers as indicative of the inter-region latencies.

6 Related Work

Kubernetes [5] scheduler starts selecting the set of *feasible placements*, which are the nodes that meet a set of resource constraints (e.g., CPU, memory requirements). Next, the scheduler filters the previous list to get the set of *viable placements*, which is the set of feasible placements with the highest score. Kubernetes adds the concepts of *taints, toleration* and *affinity/anti-affinity* in the node and pod selection. The main idea about taints and tolerance is to indicate to the scheduler that if a given node has already been tainted with a heavy-processing application such as Apache Spark, it will repel other heavy-processing applications that have a low tolerance to it. On the other hand, if a user wants to explicitly indicate that some pods should be together for performance, she can use *affinity* (or *anti-affinity*) to indicate to the scheduler that a given pod has an affinity to pods of the same type or class. Comparing Kubernetes to Docker Swarm and our approach, they work very similarly in the first phase (selection and filtering). However, Kubernetes does not support prioritization among multiple scaling requests and not take into account the client location to deploy the new pods (the assumption is that they will run in a data-center rather than as a distributed system).

Apache Mesos [10] supports both task and container scheduling, and it uses a two-level scheduling strategy to allocate resources. In the first level, Mesos manager retrieves all resources available on Mesos slaves and creates a unified view of the resource pool. This resource abstraction layer can be partitioned among different upper-level frameworks that are running on top of Mesos, such as Apache Hadoop and Marathon.

In the second level, the application-level scheduling framework schedules all tasks based on the application level policies to where to allocate and run the tasks. Compared to our work, the Docker Swarm has a similar role as Apache Mesos, where it gathers all resources from the nodes and provides a unified view on that. However, at the container scheduling step, Docker Swarm is directly responsible for scheduling and allocating containers to nodes, while Mesos outsources it to the upper framework to handle that.

7 Discussion

In this paper, we presented an orchestrator that can (i) prioritize requests and (ii) allocate containers to the closest region to the client as possible in the edge of the network. The main contribution of this paper is to provide a new orchestrator for Docker framework which is location-aware, i.e., the Docker orchestrator to take the scheduling decision based on locality and to be close to the consumers in the edge of the network. Unfortunately, current container orchestrators (Docker, Kubernetes, and Mesos) are not location-aware and just deploying containers in the wild for edge networks will result in poor performance as they may place far from the users.

The second takeaway from the paper is that a poorly configured container scheduler can end up scaling resources far from the most demanded region. In a scenario where the client region has a data-center deployed, it makes more sense to deploy directly to the cloud rather than having a closer edge deployment with a scheduler that can send your application container far away from the client (unless of course if all latency in the edge deployment in that region is lower than the latency to the cloud in the same region).

Finally, ESS's deployment algorithm can use optimization techniques to provide even a better solution for edge placement. In the ESS design, we chose for fast response time as we expect that production system to have thousands of nodes while optimization techniques usually run in the order of minutes. We believe that the optimization values can be inputted to our system offline, thus, giving the scheduler better parameters and historical data to deploy future containers.

8 Conclusions

This paper presented the ESS edge deployment strategy, a new algorithm that selects the best container placement location at the edge of the network. The algorithm classifies the incoming scaling requests and prioritizes them according to the user-defined metrics and probes the monitoring components to get the most demanded container locations to take the deployment decision. We implemented a prototype, evaluated it using 16 different regions on AWS and compared the latency between different strategies on Docker Swarm. The experimental results show that the edge strategy can reduce the overall latency by up to 5 times compared to the other default strategies in the best scenario and the amount of client requests handled under 25ms is up to 1.5 times compared to the regular strategies.

References

[1] [n.d.]. AlertManager. https://prometheus.io/docs/alerting/alertmanager/. Accessed: 2019-01-30.

[2] [n.d.]. Docker Swarm Mode Overview. https://docs.docker.com/engine/swarm/. Accessed: 2019-01-30.

[3] [n.d.]. Google cAdvisor. https://github.com/google/cadvisor. Accessed: 2019-01-30.

[4] [n.d.]. Prometheus - Monitoring system & time series database. https://prometheus.io. Accessed: 2019-01-30.

[5] [n.d.]. The Kubernetes Scheduler. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/. Accessed: 2019-01-30.

[6] 2018. *Ericsson Mobility Report*. Technical Report. https://www.ericsson.com/assets/local/mobility-report/documents/2018/ericsson-mobility-report-june-2018.pdf.

[7] Rajesh Balan, Jason Flinn, M. Satyanarayanan, Shafeeq Sinnamohideen, and Hen-I Yang. 2002. The Case for Cyber Foraging. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop (EW 10)*. ACM, New York, NY, USA, 87–92. https://doi.org/10.1145/1133373.1133390

[8] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. 2012. COMET: Code Offload by Migrating Execution Transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 93–106. http://dl.acm.org/citation.cfm?id=2387880.2387890

[9] K. Habak, M. Ammar, K. A. Harras, and E. Zegura. 2015. Femto Clouds: Leveraging Mobile Devices to Provide Cloud Service at the Edge. In *2015 IEEE 8th International Conference on Cloud Computing*. 9–16. https://doi.org/10.1109/CLOUD.2015.12

[10] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, Berkeley, CA, USA, 295–308. http://dl.acm.org/citation.cfm?id=1972457.1972488

[11] C. Pahl and B. Lee. 2015. Containers and Clusters for Edge Cloud Architectures – A Technology Review. In *2015 3rd International Conference on Future Internet of Things and Cloud*. 379–386. https://doi.org/10.1109/FiCloud.2015.35

[12] Matthew Roughan. 2005. Simplifying the Synthesis of Internet Traffic Matrices. *SIGCOMM Comput. Commun. Rev.* 35, 5 (Oct. 2005), 93–96. https://doi.org/10.1145/1096536.1096551

[13] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. 2009. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing* 8, 4 (Oct 2009), 14–23. https://doi.org/10.1109/MPRV.2009.82

[14] Enrique Saurez, Kirak Hong, Dave Lillethun, Umakishore Ramachandran, and Beate Ottenwälder. 2016. Incremental Deployment and Migration of Geo-distributed Situation Awareness Applications in the Fog. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems (DEBS '16)*. ACM, New York, NY, USA, 258–269. https://doi.org/10.1145/2933267.2933317

[15] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (Oct 2016), 637–646. https://doi.org/10.1109/JIOT.2016.2579198

[16] Chih-Peng Wu, Mahima Agumbe Suresh, and Dilma Da Silva. 2017. Container Lifecycle Management for Edge Nodes: Poster. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing (SEC '17)*. Article 23, 2 pages.

[17] S. Yi, Z. Hao, Z. Qin, and Q. Li. 2015. Fog Computing: Platform and Applications. In *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*. 73–78.