Evaluating Container Deployment Implementations for Foglets

Michael Mutkoski
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, USA
mutkoski@gatech.edu

Abstract

In recent years, the number of devices connected to local networks has rapidly expanded to create a new internet known as The Internet of Things. The applications run on these devices often require lower latency solutions than cloud computing can provide in order to perform time-sensitive interactions with other devices near the network's edge. One solution to this problem is fog computing, a geo-distributed architecture that provides computational resources closer to the edge of the network. This proximity yields low-latency connections among such devices. In order to implement a powerful fog computing network, applications must be able to deploy and migrate quickly throughout the geo-distributed resources. In the Foglets project, containers are used to efficiently deploy applications. The Foglets project currently contains two platforms that handle container deployment: one that utilizes system calls, and another that uses the well-established Docker API. In this work, we evaluate the latency and throughput of the two deployment platforms, as well as the impact of container commands and size on these metrics. We found that while serving many simultaneous deployments through multithreading, the Docker API yields lower latency and higher throughput. We also found that the size of the container and commands run on the container had a negligible impact on the deployment's latency and throughput.

Introduction

The increase of internet connected hardware devices, commonly known as The Internet of Things, has led to a need for lower latency connectivity in a technological landscape that is dominated by cloud computing. While methods such as compression, local caching, and TCP window sizing have dramatically increased the speed at which internet connected devices can

communicate with the Cloud, the physical limitations of transferring data over hundreds of miles to these cloud-based data centers still impose severe limitations on situationally aware applications that require increasingly low latency computation. (Belli 2015)

The fog computing model is a simple, low latency solution to the expanding hardware infrastructure behind the Internet of Things. The idea behind fog computing is to extend the cloud computing model out to the edges of the network. This involves using devices close to the user's local network for utility computing, allowing for shared computational power and data storage, accompanied by the low latency of communication close to the network's edge. (Bonomi et al. 2012)

The current programming model used to implement fog computing is known as Foglets. This model first identifies resources within the network topology, deploying application components to the identified resources that meet the latency thresholds of each component. Foglets also allows each fog node, or computational resource, in the topology to handle multi-application collocation. It provides APIs to allow application components to communicate over the network. Finally, Foglets supports situationally aware applications by allowing for latency and workload-driven resource adaptation, as well as geographic and temporal state migration. (Saurez et al. 2016)

The Foglets project initially required an efficient way to deploy and migrate applications across fog nodes. The simple, but greedy, solution was virtual machines, which would allow for full operating systems containing all necessary dependencies to be wrapped around an application

and distributed to be run on fog nodes quickly with reliable results. However, virtual machines require the transfer of relatively large files, and necessitate a new guest operating system to be booted up on top of the node's existing operating system on each deployment, making it a very bulky and suboptimal solution. Containers, on the other hand, are software packages containing applications and all of their dependencies, without the added size of a guest operating system. These containers are lightweight, fast to deploy, and run reliably on different computing environments. In addition, many containers can run simultaneously on a single engine that functions on top of the node's operating system. This made containers the clear solution for the scalable deployment and migration of applications in the Foglets project. (Saurez et al. 2016)

The Foglets project currently implements two methods of deploying containers, one through system calls and another using the Docker API wrapped in a custom Docker Manager class. In the system calls implementation, the Foglets code makes a call to the host's operating system, which spawns a command line process. This process then sends a deployment command to the Docker Daemon, which spawns a container. Once this command has been sent, there is no longer an open connection between the Foglets project and the Docker Daemon.

The second deployment method is a custom Docker Manager class, which establishes a connection directly with the Docker Daemon through the Docker API. The connection established by the Docker Manager is kept open, allowing for subsequent API calls to be made over the same connection. This research is designed to prove whether or not the properties of the Docker API, wrapped by the Docker Manager, affects the latency and throughput of deploying containers. The results of this research will determine which implementation is better for the

future of Foglets, as well as other Edge Computing projects that utilize mass deployment of containers.

Literature Review

Within the research field of networking and pervasive systems, there has been a notably dramatic increase in the number of network-connected devices over the past decade. The expansion of the Internet of Things (IoT) has pushed researchers to study new ways of supporting and utilizing the computing power of network connected devices in order to create and improve on technologies within the field of computer science. (Bonomi et al. 2012)

Currently, the most commonly supported network structure for employing computing power and data storage to edge IoT devices is known as cloud computing. Within the cloud computing paradigm, edge devices are connected through a series of routers and switches, commonly known as the internet, to computers outside the device's local network. These "cloud computers" are often located hundreds of miles away and allocate computing and storage resources to the processes of the original device. This model allows for the edge devices to have limited local computing power and storage, making it possible for each device to be made cheaper and smaller. (Belli 2015)

There has been a wealth of research in the field of cloud computing in order to make it a viable backend system for edge devices. There have been significant breakthroughs in the implementation of data reduction algorithms on the Cloud's backend, which alleviates data loads on edge devices and allows for the transfer of data to be faster and more robust. (Papageorgiou et

al. 2015) A recent advancement in the field of cloud computing is the use of virtual machine-based Cloudlets for mobile computing. In the Cloudlets model, containers and virtual machines are used in the Cloud to create dynamic environments that have transient customization. This allows for faster set-up and teardown of environments, as well as a more modular and customizable structure within the machines in the Cloud. The end result of the Cloudlets model is lower latency cloud computing specifically engineered for supporting a multitude of edge devices. (Satyanarayanan et al. 2009)

Even with the abundance of research intended to optimize cloud computing to support the Internet of Things, there is an inherent latency associated with data traveling hundreds of miles to cloud computing centers. There are certain devices and use cases that cannot function as intended with these levels of latency. The next field of research that is meant to support the abundant need for low latency computation on edge devices is known as fog computing. The concept of fog computing was introduced in 2012 as a form of distributed computation that takes place at a network level and would be able to handle the low latency needs of network connected devices. (Bonomi et al. 2012)

The first published implementation of a fog computing platform was released in 2013 under the name Mobile Fog. This implementation was rudimentary and incomplete, with many systems still utilizing greedy algorithms. Although many details such as dynamic scaling and software evaluation were addressed in the publication, much of the research into algorithmic efficiency and implementation at scale was underdeveloped (Hong et al. 2013)

The next significant advancement to the field of distributed systems also took place in 2013 with the concept of service-oriented heterogenous resource sharing in the Mobile Cloud. The publication of this concept discussed how resource sharing within a mobile cloud could be accomplished with optimally low latency. The paper then gave a comparative analysis of the process. (Nishio et al. 2013)

Eventually, in 2015, the concept of using containers and clusters was introduced to the fog computing paradigm. It was presented as a lightweight solution for deploying programs onto fog nodes and allowed for low latency and high customizability. (Pahl, Lee 2015) This idea evolved into the implementation of the Foglets project. This program focused on the creation of a more advanced fog computing system which would allow developers to incrementally deploy and migrate geo-distributed, situationally aware applications in a fog network structure. Foglets also supports the deployment of multiple containers to each fog node, and supports monitoring and migration of the applications and nodes, creating a constantly shifting optimal fog. (Saurez et al. 2016)

In 2018, a paper was published on the inefficiencies of Docker container deployment. It detailed how the Docker system deploys containers without parallelization, thereby wasting the hardware resources of the deploying devices. The researchers then proposed three optimizations: sequential image layer downloading, multi-threaded layer decompression, and I/O pipelining. This research was focused around fog computing, and the single-board devices that it commonly takes place on, such as Raspberry Pis. On such devices, the combination of these three

techniques resulted in container deployment times reduced by a factor of up to four. (Ahmed et al. 2016)

The intent of this paper is to extend the current research of the Foglets project. The efficiency of the deployment and migration of containers in the Fog is fundamental to the competence of Foglets. Up to this point, no research has been published comparing the efficiency of system calls and the Docker API as used within the Foglets platform. The purpose of this paper is to fill this gap and provide an optimal deployment method for the future of Foglets.

Methods and Procedures

The objective of this study was to test the latency and throughput of the two deployment methods implemented in the Foglets project. We constructed the Docker Benchmark program within Foglets, which used both system calls and the Docker API to deploy containers, meanwhile recording the respective latencies and throughput.

Since the Foglets project was originally implemented in C++, the Docker Benchmark program followed this convention. The resulting program was executed on a machine running Ubuntu 16.04.6 LTS, with the x86-64 architecture and the Linux 4.4.0-112-generic kernel. The machine had 24 CPU cores and 50 gigabytes of RAM.

The deployment functionality of the Foglets project is multi-threaded and can potentially serve many simultaneous deployments of containers. Therefore, the first objective of the Docker Benchmark program was to implement a thread pool service that would allow for multithreading

of the container deployments. We wrote the locked queue class, which implemented a mutex locked generic queue data structure that we used to store processes that would be run when a thread became available. We then wrote a thread pool class, which stored an input number of threads, using them to execute tasks automatically as jobs were added to the locked queue. We implemented functions to enqueue system calls as well as Docker Manager calls. These functions also measured and stored, in a member variable vector, the latencies of each deployment executed in order for them to be output and reviewed later.

Next we implemented the functions that measured the execution time of large-scale deployments, which made up the main class of the Docker Benchmark program. These functions accepted an input number of threads and containers to deploy, and then queued the deployments in the thread pool. As the thread pool handled the calculation of individual deployments for latency, these functions measured the overall throughput of the simultaneous deployments. They also accept the container and command to be run.

Our first series of tests evaluated whether or not the size of the container and the command executed affects the latency and throughput of the deployment on the system call and Docker API implementations. For these tests, we used a constant 24 threads and 24 deployments to match the number of cores on the machine we were testing on. For each deployment, we recorded latency and throughput.

We ran the following commands:

- ls
- ping localhost -c 100
- watch -n 1 ls

We ran these commands on the following container images:

- CentOS – 193 MB
- Debian – 125MB
- Ubuntu – 118 MB
- Alpine – 4 MB

Next, after finding that image size and command had a negligible effect on the latency and throughput of the system, as shown in the Results section of this paper, we tested the latency of deploying the Alpine image with the “ls” command. Once again, this was run with both the system call and Docker API implementations. For this test, we used a variable number of threads, with the number of containers staying constant. Starting at one thread and twenty-four containers, we incremented the number of threads until we’d reached twenty-four threads with twenty-four containers deployed, measuring the latency of each deployment.

Finally, we tested the throughput of both the system call and Docker API implementations. We chose the lowest latency number of threads and deployments, as found in the latency tests above.

We then ran this combination thirty times, recording the throughput of each implementation over each iteration. For this test we used the Alpine image with the "ls" command once again.

Results

As explained in the Methods and Procedures section, our first evaluations tested the effect of container size and commands on latency and throughput. For this experiment, we used four container images of different sizes, and three different commands to be executed on those containers. We deployed twenty-four containers on twenty-four threads for each combination of container image and command. This was repeated using the system calls implementation and the Docker API. The results are recorded in the tables below. The top number in each white cell is the average latency of each container deployed, and the bottom number is the overall throughput after deploying the 24 containers.

System Calls	Alpine 4 MB	Ubuntu 118 MB	Debian 125MB	CentOS 193 MB
ls	13.832 sec 1.015 containers/sec	13.043 sec 1.041 containers/sec	13.517 sec 1.005 containers/sec	13.010 sec 1.038 containers/sec
ping localhost -c 100	13.480 sec 1.035 containers/sec	13.587 sec 1.056 containers/sec	13.190 sec 1.021 containers/sec	13.182 sec 1.047 containers/sec
watch -n 1 ls	13.813 sec 1.024 containers/sec	13.587 sec 1.028 containers/sec	13.422 sec 1.015 containers/sec	13.458 sec 1.026 containers/sec

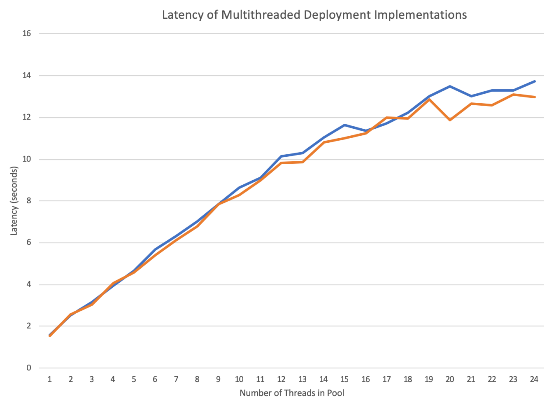
Docker API	Alpine 4 MB	Ubuntu 118 MB	Debian 125MB	CentOS 193 MB
ls	12.474 sec 1.049 containers/sec	12.873 sec 1.051 containers/sec	12.218 sec 0.980 containers/sec	12.73 sec 1.040 containers/sec
ping localhost -c 100	12.621 sec 1.046 containers/sec	12.891 sec 1.030 containers/sec	13.137 sec 1.023 containers/sec	12.764 sec 1.053 containers/sec
watch -n 1 ls	12.635 sec 1.034 containers/sec	12.574 sec 1.066 containers/sec	12.367 sec 1.066 containers/sec	12.828 sec 1.060 containers/sec

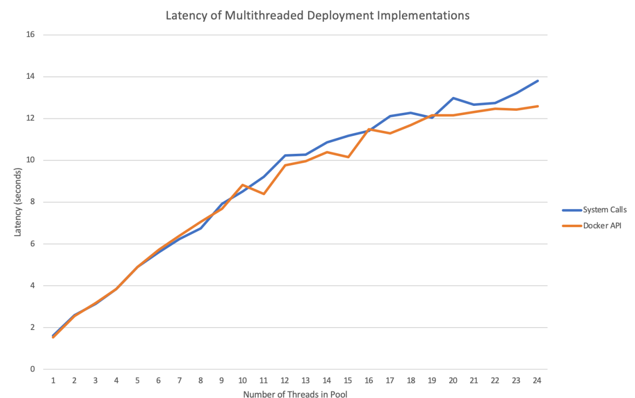
Next, we tested latency by deploying twenty-four containers on a variable number of threads, and tracking the latencies of the system call and Docker API implementations. The values in the white boxes of the table below are latency measurements, and therefore represent the number of seconds it took to deploy each container.

Threads	1	2	3	4	5	6	7	8	9	10	11	12
System Calls	1.598	2.582	3.147	3.850	4.917	5.619	6.149	6.749	7.935	8.521	9.220	10.241
Docker API	1.523	2.571	3.165	3.859	4.899	5.729	6.374	7.065	7.702	8.832	8.375	9.746

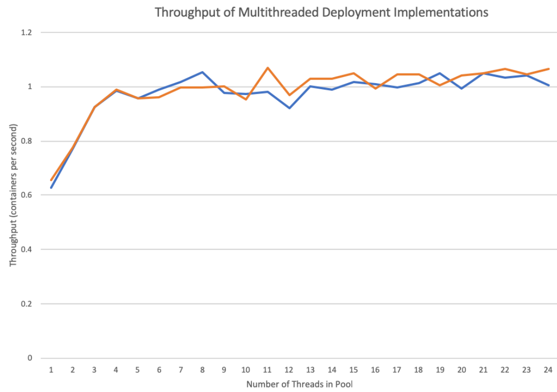
Threads	13	14	15	16	17	18	19	20	21	22	23	24
System Calls	10.261	10.867	11.173	11.422	12.102	12.276	12.016	12.956	12.657	12.733	13.217	13.796
Docker API	9.941	10.377	10.142	11.503	11.281	11.688	12.157	12.143	12.315	12.452	12.409	12.594

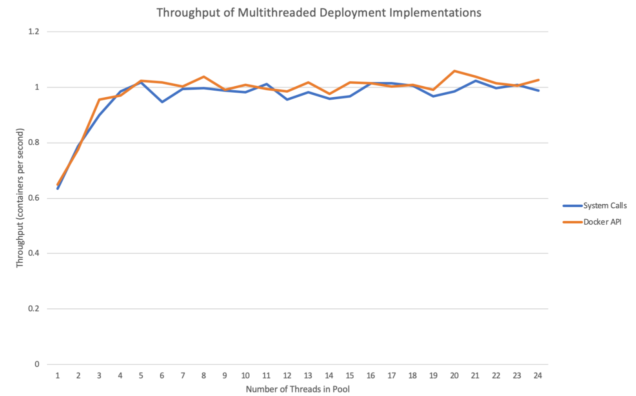
We repeated this process to ensure accurate data, and the second dataset matched the first. Line graphs of both trials are presented below.





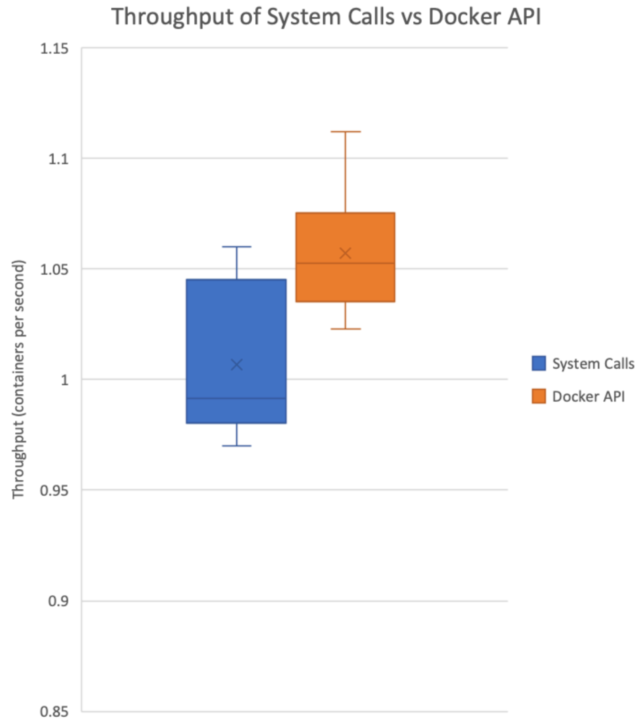
During these trials, we also collected data on the throughput of both the system calls and Docker API deployment implementations. In the graphs below, we can see how the addition of threads affects the two deployment methods over the course of the two trials.



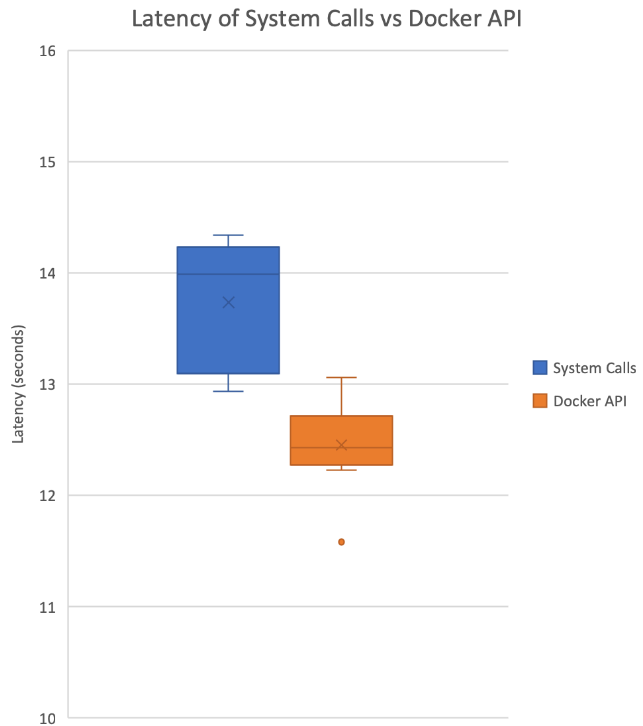


Next, we ran a new experiment to measure latency and throughput, keeping the number of threads and containers both constant at twenty-four each. We ran this test ten times, recording the latency and throughput of both the system calls and Docker API implementations. The data recorded from these trials is in the tables and plots below. The measures of throughput are in containers per second, and the measures of latency are in seconds.

Throughput (containers/sec)		
Trial	System Calls	Docker API
1	1.055	1.112
2	0.972	1.038
3	1.042	1.06
4	1.06	1.073
5	0.991	1.023
6	0.983	1.08
7	1.014	1.045
8	0.992	1.039
9	0.97	1.074
10	0.987	1.027



Latency (sec)		
Trial	System Calls	Docker API
1	13.055	11.581
2	14.219	12.603
3	13.102	12.446
4	12.937	12.324
5	14.082	12.537
6	14.279	12.289
7	13.349	12.419
8	13.898	13.042
9	14.340	12.225
10	14.075	13.060



Discussion

Our first series of evaluations sought to test whether or not the size of the container and the command run on the container has an impact on the deployment latency and throughput. From the results, we can clearly see that these variables have no effect on either metric. First, we can see that the command run has no effect on latency or throughput. For Alpine and Debian, when deployed by system calls, the "ls" command took slightly longer than "ping localhost -c 100" and "watch -n 1 ls," while on Ubuntu and CentOS "ls" was slightly faster than the alternatives. In every case, using either system calls or the Docker API, the latency was within 1 second between all commands on the same container, and the throughput was within 0.07 containers/second of all other commands run with the same container image. There is no discernable trend between the command deployed and latency or throughput of the deployment.

Next, we found that the size of the container also has a negligible impact on deployment speeds. Using system calls, Alpine, the smallest container, had higher latency deployments than any other container, while Debian, the second largest container, had the lowest throughput. Ubuntu, another mid-sized container had the highest throughput. When deployed through the Docker API, Ubuntu now has the highest latency deployments, while Alpine and Debian has the lowest. CentOS, the largest container, is neither the highest nor lowest for both latency and throughput. All latencies were once again within one second of each other across deployments of different container images, and throughputs were within 0.06 containers per second. There are no discernable trend between container size and deployment efficiency.

Next, we tested the effect of multithreading on the system calls and Docker API deployment implementations. As the number of threads was increased and the number of containers deployed remained constant, we observed an increase in latency. This was expected when multithreading. The computer's resources are divided among multiple processes, so each process takes longer. However, the work is being done synchronously, so the total deployment time of all containers decreased, resulting in an increasing throughput. This throughput rose sharply at first as more threads were added, but seemed to plateau at around six threads, with only small improvements when more threads were added.

These trends were present across both the system calls and Docker API deployment implementations. The system calls has a slightly higher latency and lower throughput across any number of threads. This would suggest that the Docker API was performing slightly more efficient deployments.

Our final series of tests were designed to hold the number of threads and containers constant, both at twenty-four, to get a direct comparison of the System Calls and Docker API implementations using the metrics of latency and throughput. Here we observed the largest differences across the study. The throughput of deployments using system calls was significantly lower than the deployments using the Docker API. Following a similar trend, the latency of deployments was much higher using system calls than the Docker API.

Conclusions and Future Work

The results of these evaluations suggest that the Docker API is a more efficient deployment implementation for multithreaded container deployments. The lower latency measurements in conjunction with higher throughput shows that the Docker API deploys individual containers faster during large-scale deployments, as well as finishing more deployments per second on average. The Docker API should be used in the Foglets system to synchronously deploy containers to fog nodes.

There is room for future work studying the monitoring implementations of each of these deployment methods. The Docker API maintains an established connection between the deployment machine and the Docker Daemon, which should allow for faster and easier monitoring of containers. However, the Foglets system allows for continuous migration of containers. Each time a container migrates to a new fog node, this connection must be reestablished. In environments with high migration rates, this added complexity could result in higher latency and less efficient monitoring.

References

Belli, L. (2015). Big Stream Cloud Architecture for the Internet of Things. Proceedings of the 2015 on MobiSys PhD Forum. Florence, Italy, ACM: 5-6.

Bernstein, D. (2014). "Containers and Cloud: From LXC to Docker to Kubernetes." IEEE Cloud Computing 1(3): 81-84.

Bonomi, F., et al. (2012). Fog computing and its role in the internet of things. Proceedings of the first edition of the MCC workshop on Mobile cloud computing. Helsinki, Finland, ACM: 13-16.

Gedik, B., et al. (2008). SPADE: the system s declarative stream processing engine. Proceedings of the 2008 ACM SIGMOD international conference on Management of data. Vancouver, Canada, ACM: 1123-1134.

Ha, K., et al. (2017). You can teach elephants to dance: agile VM handoff for edge computing. Proceedings of the Second ACM/IEEE Symposium on Edge Computing. San Jose, California, ACM: 1-14.

Hong, K., et al. (2013). Mobile fog: a programming model for large-scale applications on the internet of things. Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing. Hong Kong, China, ACM: 15-20.

Hong, K., et al. (2011). Target container: A target-centric parallel programming abstraction for video-based surveillance. 2011 Fifth ACM/IEEE International Conference on Distributed Smart Cameras.

Koldehofe, B., et al. (2012). Moving range queries in distributed complex event processing. Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems. Berlin, Germany, ACM: 201-212.

Liu, Z., et al. (2010). Xen Live Migration with Slowdown Scheduling Algorithm. 2010 International Conference on Parallel and Distributed Computing, Applications and Technologies.

Nishio, T., et al. (2013). Service-oriented heterogeneous resource sharing for optimizing service latency in mobile cloud. Proceedings of the first international workshop on Mobile cloud computing & networking. Bangalore, India, ACM: 19-26.

Ottenw, B., et al. (2013). MigCEP: operator migration for mobility driven distributed complex event processing. Proceedings of the 7th ACM international conference on Distributed event- based systems. Arlington, Texas, USA, ACM: 183-194.

Pahl, C. and B. Lee (2015). Containers and Clusters for Edge Cloud Architectures -- A Technology Review. Proceedings of the 2015 3rd International Conference on Future Internet of Things and Cloud, IEEE Computer Society: 379-386.

Papageorgiou, A., et al. (2015). Real-time data reduction at the network edge of Internet-of- Things systems. Proceedings of the 2015 11th International Conference on Network and Service Management (CNSM), IEEE Computer Society: 284-291.

Satyanarayanan, M., et al. (2009). "The Case for VM-Based Cloudlets in Mobile Computing." IEEE Pervasive Computing 8(4): 14-23.

Saurez, E., et al. (2016). Incremental deployment and migration of geo-distributed situation awareness applications in the fog. Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems. Irvine, California, ACM: 258-269.

Simanta, S., et al. (2012). A Reference Architecture for Mobile Code Offload in Hostile Environments. Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, IEEE Computer Society: 282- 286.

Soltesz, S., et al. (2007). Container-based operating system virtualization: a scalable, high- performance alternative to hypervisors. Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007. Lisbon, Portugal, ACM: 275-287.

Urgaonkar, R., et al. (2015). "Dynamic service migration and workload scheduling in

edge- clouds." Perform. Eval. 91(C): 205-228.

Wang, S., et al. (2017). "Dynamic Service Placement for Mobile Micro-Clouds with Predicted Future Costs." IEEE Trans. Parallel Distrib. Syst. 28(4): 1002-1016.

Yao, H., et al. (2015). "Migrate or not? Exploring virtual machine migration in roadside cloudlet- based vehicular cloud." Concurr. Comput. : Pract. Exper. 27(18): 5780-5792.

A. Ahmed and G. Pierre, "Docker Container Deployment in Fog Computing Infrastructures," *2018 IEEE International Conference on Edge Computing (EDGE)*, San Francisco, CA, 2018, pp. 1-8.