A Decentralized System for Load Balancing of Containerized Microservices in the Cloud

Marian Rusek(✉), Grzegorz Dwornicki, and Arkadiusz Orłowski

Faculty of Applied Informatics and Mathematics, Warsaw University of Life Sciences, Nowoursynowska 166, 02–787 Warsaw, Poland
marian_rusek@sggw.pl

Abstract. Microservice architecture is a cloud application design pattern which shifts the complexity away from the traditional monolithic application into the infrastructure. Each microservice is a small containerized application that has a single responsibility in terms of functional requirement, and that can be deployed, scaled and tested independently using automated orchestration systems. We propose a simple swarm-like decentralized load balancing system for microservices running inside OpenVZ containers. It can potentially offer performance improvements with respect to the existing centralized container orchestration systems.

Keywords: Container orchestration · Swarm algorithm · Microservices

1 Introduction

A typical monolithic enterprise systems we build today are difficult to scale, difficult to understand and difficult to maintain. Written in a monolithic way, these systems tend to have strong coupling between the components in the service and between services. A system with the services tangled and interdependent is harder to write, understand, test, evolve, upgrade and operate independently. Strong coupling can also lead to cascading failures: one failing service can take down the entire system, instead of allowing you to deal with the failure in isolation. Popular application servers (e.g., WebLogic, WebSphere, JBoss or Tomcat) encourage this monolithic model.

Microservices-based architecture is free of these problems [2,16,18,24,28]. It advocates creating a system from a collection of small, isolated services, each of which owns their data, and is independently isolated, scalable and resilient to failure. Services integrate with other services in order to form a cohesive system thats far more flexible than a typical monolithic system. One of the key principles in employing a microservices-based architecture is the decomposition of the system into discrete isolated subsystems communicating over well defined asynchronous protocols and decoupled in time (allowing concurrency) and space (allowing distribution and mobility – the ability to move services around).

Some developers and researches believe that the concept of microserices is a specific pattern of implementation of service-oriented architecture (SOA). However the microservice pattern has the following unique specifics: microservices use lightweight HTTP mechanisms for communication, they are independently deployable by fully automated machinery, and there is only a bare minimum of centralized management [24]. Enterprise service bus (ESB) is a typical software model used for designing and implementing communication between mutually interacting software applications in SOA. ESB provides all of the routing and data transformation required to get the parts of an application talking to each other. In the microservices-based architecture there is no central unit like ESB which does the routing. The accidental complexity is shifted from inside of an monolithic application into the infrastructure. It is possible because now we have many more ways to manage that complexity: programmable infrastructure, infrastructure automation, and the movement to the cloud [28].

Today we have a much more refined foundation for isolation of services, using virtualization, Linux Containers (LXC), Docker, and Unikernels [15,19]. This has made it possible to treat isolation as a necessity for resilience, scalability, continuous delivery and operations efficiency. It has also paved the way for the rising interest in microservices-based architectures, allowing you to slice up the monolith and develop, deploy, run, scale and manage the services independently of each other. The value of microservices and containers lies in how they enable smaller, faster, more frequent change [5,14]. While cloud computing changed how we manage "machines," it didnt change the basic things we managed. Containers, on the other hand, promise a world that transcends our attachment to traditional servers applications and application components. One might claim that represent the fruition of the object-oriented, component-based vision for application architecture.

So how do you build a smart system from a data center filled with dumb servers? This is where tools like Google Kubernetes [4] and open source Apache Mesos [13] data center operating system come in. Also of note is Dockers platform, using its Machine, Swarm and Compose tools [26]. The role of orchestration and scheduling within these container platforms is to match applications to resources. Google developed Kubernetes for managing large numbers of containers. Instead of assigning each container to a host machine, Kubernetes groups containers into pods. For instance, a multi-tier application, with a database in one container and the application logic in another container, can be grouped into a single pod. The administrator only needs to move a single pod from one compute resource to another, rather than worrying about dozens of individual containers. Apache Mesos is a cluster manager that can help the administrator schedule workloads on a cluster of servers. Mesos excels at handling very large workloads, such as an implementation of the Spark or Hadoop data processing platforms. Docker Swarm is a clustering and scheduling tool that automatically optimizes a distributed applications infrastructure based on the applications lifecycle stage, container usage and performance needs. All these container orchestration systems are monolithic applications running as daemons on dedicated nodes of the cloud. They orchestrate containers in a centralized fashion.

Usually decentralized orchestration systems offer performance improvements. For example decentralized orchestration of composite web services yields increased throughput, better scalability, and lower response time [6]. In this paper a decentralized system for load balancing of containerized microservices is proposed. In Sect. 2 the internals of a virtualization container are analyzed. In addition to cloud application it can run an additional process implementing the mobile agent intelligence. In Sect. 3 the swarm-like algorithm of container migration in the cloud is introduced. The number of containers on each host plays a role analogous to a pheromone in colonies of insects or simple transceivers mounted on autonomous robots. In Sect. 4 some preliminary experimental results for a simple cloud consisting of 18 hosts are presented. We finish with a summary and brief remarks in Sect. 5.

2 Container Internals

The startup time for a container is around a second. Public cloud virtual machines take from tens of seconds to several minutes, because they boot a full operating system every time. Thus recently the cloud industry is moving beyond self-contained, isolated, and monolithic virtual machine images in favor of container-type virtualization [17,25]. Containers introduce autonomy for applications by packaging apps with the libraries and other binaries on which they depend. This avoids conflicts between apps that otherwise rely on key components of the underlying host operating system. Containers do not contain a operating system kernel, which makes them faster and more agile than virtual machines. Container-type virtualization is an ability to run multiple isolated sets of processes, each set for each application, under a single kernel instance. Having such an isolation opens the possibility to save the complete state of (in other words, to checkpoint) a container and later to restart it. This feature allows one to checkpoint the state of a running container and restart it later on the same or a different host, in a way transparent for running applications and network connections [11,17].

In this paper container-type virtualization is used to build a swarm of tasks in a cloud. Each container in addition to the application and their libraries contains an separate process representing the mobile agent [1,12,27]. It deals with sensing the neighboring containers and initiating live migration of its container to another host. A typical modern server can run only about 10 virtual machines or about 100 containers. Therefore the density of container based mobile agents in a cloud can be 10 times higher. Typical migration time of a virtual machine is about 10 s—container can be migrated in time of order of 1 s [29]. Thus container migration is about 10 times faster. Containers can call system functions of the operating system kernel running on the server. Therefore in principle they can initiate container migration without help of a separate daemon process running on the host server.

The years from 2002 to 2010 represented a time of experimentation, when two projects in particular moved the needle on virtualization containers in Linux.

VServer project patched the Linux kernel in order to split things up into virtual servers, an early version of what today we would call containers. The second project was OpenVZ, which transformed the Linux kernel, so that you could run containers in production. Despite its success, OpenVZ never managed to get the containerization technology merged into the stock Linux kernel and always required a custom patch to make it possible. At later time, control groups and namespaces [22] were introduced, making LXC containers a functionality available within the stock Linux kernel. It thus became possible to use something that looked like a container without patching the kernel. At the time, Salomon Hykes was leading dotCloud, an infrastructure platform as a service (PaaS) company that was committed to applying standards in the deployment of distributed architecture for applications. They spent three years running a cloud platform production using LXC, so they had a lot of operational experience. They learned that this technology was not practical, so they wrote a tool that was more stable and allowed to deliver container-based application deployment for large-scale hybrid cloud environments. In this way a popular Docker technology based on a libcontainer format was born.

Docker containers cannot be live migrated between hosts—they can only be snapshotted and restored on the same or other host. The generally accepted method for managing Docker container data is to have stateless containers running in the production environment that store no data on their own and are purely transactional. Stateless containers store processed data on the outside, beyond the realm of their container space, preferably to a dedicated storage service that is backed by a reliable and available persistence backend. Another class of container instances are these that host storage services, like upon pattern is to use data containers. The runtime engines of these stateful services get linked at runtime with the data containers. In practical terms, this would mean having a database engine that would run on a container, but using a "data container" that is mounted as a volume to store the state. Therefore to run a cloud hosting environment, it is important to have a distributed storage solution, like Gluster and Ceph, to provide shared mount points. This is useful if the container instances move around the cloud based on availability.

Parallels®Virtuozzo is another widely deployed container-based virtualization software for Linux and Windows operating systems. As opposed to Docker Virtuozzo allows for live migration of containers. The results presented in this paper were obtained using an open source version of this software called OpenVZ [17]. OpenVZ is available for Linux operating system only and runs on a custom kernel. There have been several studies on various optimizations of container migration algorithms [17]. Two best known examples are lazy migration and iterative migration. Lazy migration is the migration of memory *after* actual migration of container, i.e., memory pages are transferred from the source server to the destination on demand. In the case of an iterative migration iterative migration of memory happens *before* actual migration of container. In our experiments with stripped down OpenVZ containers with a size of 50 MB in a test system consisting of two nodes connected by a 100 Megabit Ethernet network we measured the migration time seen by the host $T = 6.61\,\mathrm{s}$ and the migration time seen by the container $\tau = 2.25\,\text{s}$. The later is three times smaller than the former due to optimizations described above.

We have altered the OpenVZ kernel by adding a system function allowing the container to ask the host to migrate it to another host. By calling this function a container is placed in a queue in the kernel - a dedicated daemon reads this queue and migrates all the containers waiting in it. Thus containers can leave the host only in a sequence. Our studies indicate that a parallel migration is possible, but the performance gain is negligible—migration speed increases only by 8 %. In addition as shown in our previous paper dealing with two hosts only sequential migration helps to stabilize the swarm algorithm [23].

Processes running inside an OpenVZ container have their own disk with partitions and file systems. In reality this is a virtual disk and its image is stored as a file on the physical disk of the host. This solution makes the migration of the container's data to another host is very easy—only a single file needs to be copied between servers. The network of the container is isolated in a way that allows the container to have they own IP address on the network. This is not the IP address of host but it can be reached from the other containers and hosts. Each container maintains its own state: network connections, file descriptors, memory usage etc. Containers share only the kernel with the host operating system. Thanks to state isolation from other containers a container can be migrated to another system and resumed.

When we launch our container for the first time it does not know on what host it was started. However it can use an ICMP echo/reply mechanism to detect the IP address of the host. Each ICMP packet has a TTL (Time-to-Live) value. When this packet is routed trough router this value is decreased. When it reaches 0 the packet is destroyed and an error ICMP packet is send back. This ICMP error packet will have last router IP address. Thus to detect the IP address of the host our container can send an ICMP echo packet with value 1 of TTL to some arbitrary external IP address. The host system acts as a router for container's network. When this special packet is sent by the container it will never reach the destination but the host system will send back an ICMP error packet with its IP address.

Host system keeps all the containers filesystems mounted on its local file system. Each container's file system is visible as a folder located in `/var/lib/vz/root/CID` where CID is an unique container identification number. The location can be exported trough an network file system like NFS. Our container will mount it locally. To do this it needs to know the export path `/var/lib/vz/root` and the IP address of its host system. By counting the number of entries in this folder it can detect other containers running on the same hosts and count their number N. By calling a custom system function written by us it can also check for the number Q of containers queued for migration in the kernel. A container can also log in via ssh to another host and ask for these parameters there. Each container knows how many hosts we have H and knows their IP addresses. It also knows how many other containers are there C in the cloud. These numbers can be updated dynamically at runtime by probing other containers and hosts using ICMP echo/reply protocol either by the container itself or by the host.

3 Swarm Algorithm

There many examples in biology how complex global behaviors can arise from simple interactions between large numbers of relatively unintelligent agents. Examples of self-organized processes of natural aggregation are nest construction, foraging, brood sorting, hunting, navigation, and emigration. All involve only local interactions between individuals and between individuals and their environment. For example the ants rely only on physical contact and pheromone communication, but simple individual ant behaviors result in group behaviors that are thought to be optimal for the entire colony. Emerging technologies are making it possible to cheaply manufacture small robots with sensors, actuators and computation. Swarm approaches to robotics, involving large numbers of simple robots rather than a small number of sophisticated robots, has many advantages with respect to robustness and efficiency. Such systems can typically absorb many types of failures and unplanned behavior at the individual agent level, without sacrificing task completion [3,7–9,20,21]. These properties make swarm intelligence an attractive solution also for other problem domains. In this paper we use this approach for task scheduling in a complex distributed system—the cloud.

Let us now propose a swarm-like decentralized algorithm for container migration inspired by pheromone robots [20,21]. The proposed approach threats the containers as mobile agents and is also capable of automatic self-repair; the system can quickly recover from most patterns of agent death and can receive an influx of new agents at any location without blocking problems. Each host is described by a pheromone p which can be either repulsive $0 < p < 1$ or attractive $p < 0$. The complete algorithm executed by a dedicated process running inside each container reads as follows:

```
 1: Use the method described in Sect. 2 to get the IP address of the host.
 2: Mount the /var/lib/vz/root folder exported by it.
 3: loop
 4:    repeat
 5:       Obtain the pheromone value $p$ of the host.
 6:       Generate a random number $0 < r < 1$.
 7:    until $r < p$
 8:    Randomly choose a host with an attractive pheromone $p < 0$.
 9:    Ask the host to migrate to it.
10:    repeat
11:       Get the IP address of the host.
12:    until It's different from the previous one
       {Migration to another host is complete}
13:    Unmount /var/lib/vz/root from the old host.
14:    Mount it from the new host.
15: end loop
```

Thus the pheromone p can be viewed as a migration probability of a container. The simplest choice for p is the fraction of the number of containers on a host:

$$p = \frac{N - Q - n}{N - Q} \tag{1}$$

above the equilibrium value where the containers are equally distributed between the hosts:

$$n = \frac{C}{H} \tag{2}$$

In this case the tasks are migrated between the nodes until the number of tasks on each node is the same and equal to n. Using an analogy with physics the nodes can be imagined to be gas containers, and the tasks running on them—gas molecules. The network links between the nodes are tubes connecting the containers between each other. The pressure of the gas equilibrates until it is the same in each container. Similar analogy is used in a self-repairing formation of mobile agents [8]. Note that the subtraction of Q in Eq. (1) is necessary in order to avoid oscillations of the containers [23].

In realistic cloud environments, tasks often differ regarding CPU and I/O load. Hence, optimization towards an equal distribution of tasks across available hosts does not seem to be optimal in each case. Instead of using Eq. (2) the desired number of containers of a given type n could be computed on each server separately using Dominant Resource Fairness (DRF) algorithm [10]. For example consider a server with 9 CPU cores and 18 GB of RAM. Container of type A needs 1 CPU core and 4 GB of RAM, and container of type B—3 CPU cores and 1 GB od RAM. DRF gives $n_A = 3$, and $n_B = 2$. The pheromone value p from Eq. (1) needs to be calculated separately for each container type. There are separate migration queues Q for containers of different types.

4 Experimental Results

The experiments were performed on $H = 18$ servers equipped with Intel®i5-3570 Quad-Core CPU, 8 GB of RAM each connected by a dedicated 100 Megabit Ethernet network. All servers were running Debian GNU/Linux operating system with OpenVZ software installed. The Linux kernel was modified by adding new system functions as described in Sect. 3. At the initial time $C = 18 \cdot 17 = 306$ identical containers are launched on the first 17 hosts, and the last one was empty:

$$N_i = 18, \quad i = 1, \dots 17, \qquad N_{18} = 0 \tag{3}$$

Each container has a size of about 100 MB and its migration to another host takes $T = 16$ s. Each container is running a Python script implementing the algorithm from Sect. 3. It starts by scanning the network using `nmap` to find the number of containers C, and the number of hosts $H = 18$ and their IP addresses. Than the mean number of containers $n = 17$ is calculated and the script enters a loop in which it periodically checks the pheromone value p, and decides with probability p whether to migrate to another host. In addition on the host server a monitor program was started which periodically (period 5 s) checked the number of containers N in the filesystem and the number of containers queued for migration Q in the kernel queue (access to this data from a user process was possible by a custom system function added to the kernel).

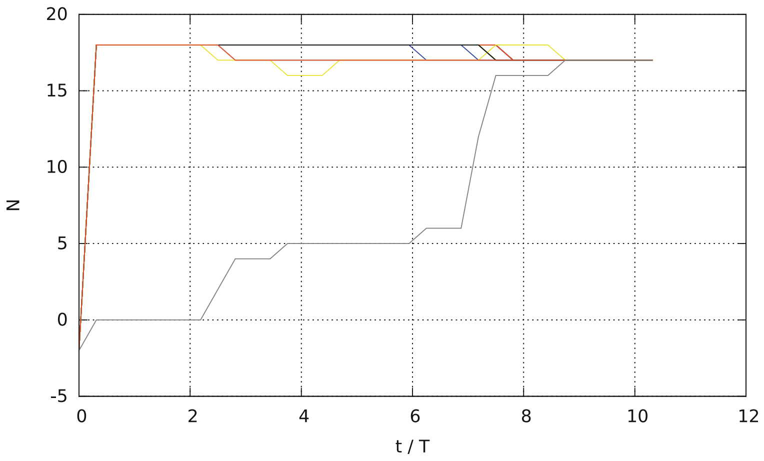

Fig. 1. Number of containers on each host versus time.

In Fig. 1 we have the numbers of containers N on each host plotted versus time t. It is seen from inspection of this plot that the containers can arrive to the destination host in parallel thus network bandwith was apparently not a problem during this experiment. Notice that around $t \simeq 4\,T$ the yellow line drops below the equilibrium value of $N = 17$—this happens because the migration process is inherently a probabilistic one. The migration probability is $p = 1/18$ but sometimes more than $p\,N = 1$ container can decide to jump to another host. Also the containers do not move independently but interact with each other. If more than one excess container asks the host for migration, then one of them must wait in a queue until the first one leaves the host. The system reaches equilibrium and migration stops around $t \simeq 9\,T$:

$$N_i = 17, \quad i = 1, \ldots 18 \tag{4}$$

Thus at average two containers arrive to the destination host during time T.

Containers startup is not instantaneous. The experiment was arranged in such a way that the migration agent processes inside the containers were started in a loop—therefore some were started later than the other. In Fig. 2 we have the numbers of containers waiting for migration Q on each host plotted versus time t. Indeed we see a small delay in entering the queue. The first container leaves the migration queue around $t \simeq 1.5\,T$ (red line) but is deleted from the file system with some delay only after $t > 2\,T$ (c.f., Fig. 1).

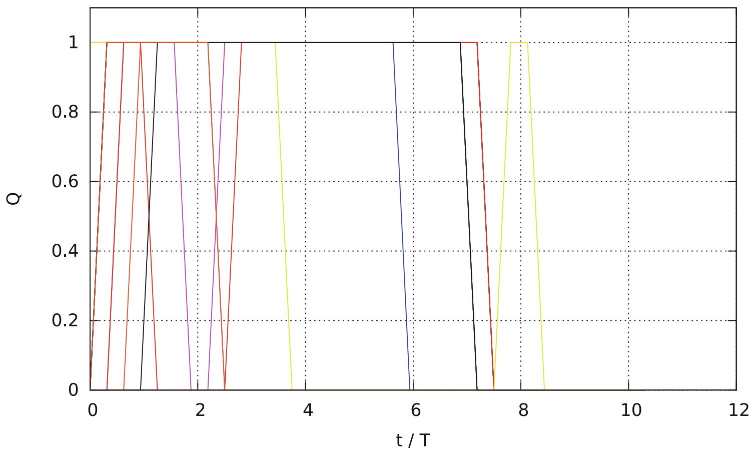

Fig. 2. Number of containers waiting for migration on each host versus time. (Color figure online)

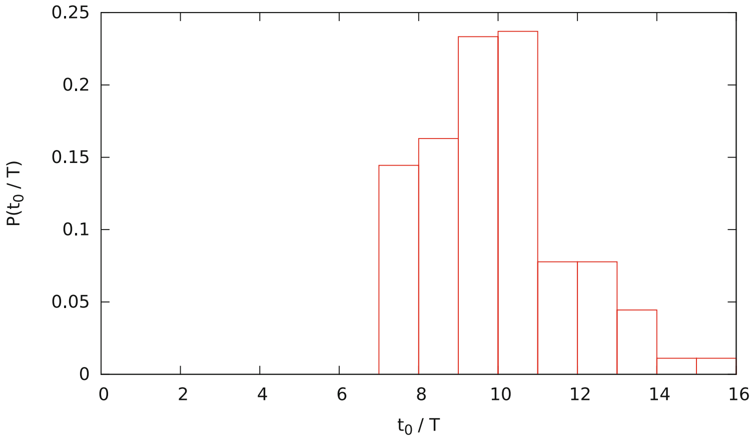

Fig. 3. Histogram of times needed to reach equilibrium.

To investigate the container self-organization process even further in Fig. 3 we have a histogram of times needed to reach equilibrium t_0 obtained from 300 runs of the algorithm. It is seen that the case discussed earlier is a fairly typical one.

5 Summary

In summary, the OpenVZ containerization software was used to implement a swarm of tasks executing in a cloud. Each task includes a mobile agent process which governs its migration to another nodes of the cloud. A variant of the Contained Gas Model known from self-repairing formations of autonomous robots is used. The tasks running on the nodes of the cloud self-organize to maintain a constant load among the servers. The system automatically adapts to creation and destruction of tasks as well as extension of the cloud by new servers. It can be easily adopted to react on server failures: a failing server can produce an artificial pheromone by creating entries in the `/var/lib/vz/root` directory of its filesystem. This will cause all the tasks running on it to migrate away from the pheromone. The performance of the swarm-like algorithm proposed to control the containers was experimentally tested on a simple "cloud" consisting of 18 nodes.

References

1. Aversa, R., Di Martino, B., Rak, M., Venticinque, S.: Cloud agency: a mobile agent based cloud system. In: 2010 International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), pp. 132–137. IEEE (2010)
2. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Migrating to cloud-native architectures using microservices: an experience report. In: Celesti, A., Leitner, P. (eds.) ESOCC Workshops 2015. CCIS, vol. 567, pp. 201–215. Springer, Heidelberg (2016). doi:10.1007/978-3-319-33313-7_15
3. Berman, S., Halász, A., Kumar, V., Pratt, S.: Bio-inspired group behaviors for the deployment of a swarm of robots to multiple destinations. In: 2007 IEEE International Conference on Robotics and Automation, pp. 2318–2323. IEEE (2007)
4. Brewer, E.A.: Kubernetes and the path to cloud native. In: Proceedings of the Sixth ACM Symposium on Cloud Computing, pp. 167–167. ACM (2015)
5. Calinciuc, A., Spoiala, C.C., Turcu, C.O., Filote, C.: Openstack and docker: building a high-performance iaas platform for interactive social media applications. In: 2016 International Conference on Development and Application Systems (DAS), pp. 287–290. IEEE (2016)
6. Chafle, G.B., Chandra, S., Mann, V., Nanda, M.G.: Decentralized orchestration of composite web services. In: Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters, pp. 134–143. ACM (2004)
7. Cheah, C.C., Hou, S.P., Slotine, J.J.E.: Region-based shape control for a swarm of robots. Automatica **45**(10), 2406–2411 (2009)
8. Cheng, J., Cheng, W., Nagpal, R.: Robust and self-repairing formation control for swarms of mobile agents. In: AAAI, vol. 5, pp. 59–64 (2005)
9. Dorigo, M., Trianni, V., Şahin, E., Groß, R., Labella, T.H., Baldassarre, G., Nolfi, S., Deneubourg, J.L., Mondada, F., Floreano, D., et al.: Evolving self-organizing behaviors for a swarm-bot. Auton. Robots **17**(2–3), 223–245 (2004)
10. Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., Stoica, I.: Dominant resource fairness: fair allocation of multiple resource types. In: NSDI, vol. 11, p. 24 (2011)
11. Hacker, T.J., Romero, F., Nielsen, J.J.: Secure live migration of parallel applications using container-based virtual machines. Int. J. Space Based Situat. Comput. **12**(1), 45–57 (2012)
12. Haichun, N., Yong, L.: A mobile agent-based task seamless migration model for mobile cloud computing. In: 2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA), pp. 241–246. IEEE (2014)

13. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R.H., Shenker, S., Stoica, I.: Mesos: a platform for fine-grained resource sharing in the data center. In: NSDI, vol. 11, p. 22 (2011)
14. Kratzke, N.: About microservices, containers and their underestimated impact on network performance. In: Proceedings of CLOUD COMPUTING 2015 (2015)
15. Madhavapeddy, A., Scott, D.J.: Unikernels: rise of the virtual library operating system. Queue **11**(11), 30 (2013)
16. Malavalli, D., Sathappan, S.: Scalable microservice based architecture for enabling DMTF profiles. In: 2015 11th International Conference on Network and Service Management (CNSM), pp. 428–432. IEEE (2015)
17. Mirkin, A., Kuznetsov, A., Kolyshkin, K.: Containers checkpointing and live migration. Proc. Linux Symp. **2**, 85–90 (2008)
18. Namiot, D., Sneps-Sneppe, M.: On micro-services architecture. Int. J. Open Inf. Technol. **2**(9), 39 (2014)
19. Pahl, C.: Containerisation and the paas cloud. IEEE Cloud Comput. **2**(3), 24–31 (2015)
20. Payton, D., Estkowski, R., Howard, M.: Progress in pheromone robotics. Intell. Auton. Syst. **7**, 256–264 (2002)
21. Payton, D., Estkowski, R., Howard, M.: Compound behaviors in pheromone robotics. Robot. Auton. Syst. **44**(3), 229–240 (2003)
22. Pike, R., Presotto, D., Thompson, K., Trickey, H., Winterbottom, P.: The use of name spaces in plan 9. In: Proceedings of the 5th Workshop on ACM SIGOPS European Workshop: Models and Paradigms for Distributed Systems Structuring, pp. 1–5. ACM (1992)
23. Rusek, M., Dwornicki, G., Orłowski, A.: Swarm of mobile virtualization containers. In: Świątek, J., Borzemski, L., Grzech, A., Wilimowska, Z. (eds.) Information Systems Architecture and Technology: Proceedings of 36th International Conference on Information Systems Architecture and Technology – ISAT 2015 – Part III. AISC, vol. 431, pp. 75–85. Springer, Heidelberg (2016). doi:10.1007/978-3-319-28564-1_7
24. Savchenko, D., Radchenko, G., Taipale, O.: Microservices validation: mjolnirr platform case study. In: 2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp. 235–240. IEEE (2015)
25. Scheepers, M.J.: Virtualization and containerization of application infrastructure: a comparison. In: 21st Twente Student Conference on IT, pp. 1–7 (2014)
26. Stubbs, J., Moreira, W., Dooley, R.: Distributed systems of microservices using docker and serfnode. In: 2015 7th International Workshop on Science Gateways (IWSG), pp. 34–39. IEEE (2015)
27. Thant, H.A., San, K.M., Tun, K.M.L., Naing, T.T., Thein, N.: Mobile agents based load balancing method for parallel applications. In: APSITT 2005 Proceedings. 6th Asia-Pacific Symposium on Information and Telecommunication Technologies, pp. 77–82. IEEE (2005)
28. Thönes, J.: Microservices. IEEE Softw. **32**(1), 116 (2015)
29. Zhao, M., Figueiredo, R.J.: Experimental study of virtual machine migration in support of reservation of cluster resources. In: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing, p. 5. ACM (2007)