Container-based Support for Autonomic Data Stream Processing through the Fog

Antonio Brogi, Gabriele Mencagli, Davide Neri, Jacopo Soldani and Massimo Torquati

Department of Computer Science, University of Pisa
Largo B. Pontecorvo 3, I-56127 Pisa, Italy
{brogi,mencagli,davide.neri,soldani,torquati}@di.unipi.it

Abstract. We present a container-based architecture for supporting autonomic data stream processing application on Fog computing infrastructures. Our architecture runs applications as Docker containers, and it exploits Docker's native features to dynamically scale up/down the resources of a Fog node assigned to the applications running on it. Preliminary results demonstrate that Docker containers are appropriate for building migratable autonomic solutions in the Fog.

Keywords: Data Stream Processing, Autonomic Computing, Fog, IoT, Docker

1 Introduction

Fog computing [21] aims at distributing computing, storage and networking resources along the Cloud-to-IoT continuum, closer to the edge of the network where millions of connected devices produce huge data flows. Many applications (e.g., intelligent transportation, emergency management or e-health) need to process such data flows by meeting compelling time requirements which cannot be satisfactorily met by traditional Cloud+IoT solutions, typically because of latency and/or bandwidth limitations [5].

To suitably host autonomic data stream parallel applications on Fog infrastructures, new solutions for the dynamic management of resources within and across Fog nodes are needed. Container-based virtualization can help solving this need [17], and the objective of this paper is precisely to investigate how to use it to dynamically manage autonomic applications on Fog infrastructures.

We present a container-based architecture for supporting autonomic data stream processing applications one Fog infrastructures. The architecture exploits containerization to dynamically scale the resources assigned to each deployed application. Each Fog node hosts a Fog Node Controller, which interacts with the controllers of the autonomic applications deployed on such node. The objective of the interaction is to dynamically scale up and down the resources assigned to hosted applications. Fog Node Controllers of different nodes also interact to support the migration of deployed applications. Fog Node Controllers and applications are deployed as Docker containers.

The rest of this paper is structured as follows. We first discuss (Section 2) two motivating examples that illustrate need and benefits of dynamic resource management within/across different Fog nodes. After introducing Docker (Section 3), we describe the proposed system architecture (Section 4). We also present the results of two experiments (Section 5) that show the feasibility of the proposed container-based support for autonomic data stream processing in the Fog. We finally discuss related work (Section 6) and we draw some concluding remarks (Section 7).

2 Motivating examples

In this section we describe two basic examples that motivate the development of our architecture. The first example describes a scenario of *intra-fog node* resource management and orchestration, through the synergical interaction between the Fog Node Controller (`FNC`) and the Application Controllers (`AC`s) running the autonomic logic of streaming applications within a fog node. The second focuses on a more complex and challenging case of *inter-fog node* adaptation, where our architecture is exploited in order to take full advantage of its potential.

Intra-fog node scenario. Each fog node, besides being interconnected to various data providers (e.g., sensors, IoT and edge devices), can be connected to an overlay of fog nodes and eventually to a traditional cloud system. Fig. 1(left) exemplifies our vision by highlighting the role of a fog node. Within a fog node, various streaming applications can run. Each streaming application is characterized by: *i)* a set of data providers that feed the application with a continuous flow of data items to be processed; *ii)* a set of data consumers that will retrieve real-time data analytic produced by the application. We also envision that *each application should be designed with an autonomic logic inside*, responsible for scaling up/down the resources utilized by the application and/or other application-dependent configuration knobs (e.g., load balancing policies, scheduling disciplines). While some kinds of reconfigurations are executed transparently to the Fog infrastructure others needs a proper interaction with the `FNC` (e.g., resource scaling).

Consider an application consuming a data stream generated by a set of mobile devices localized near to a fog node, and processing the most recent data items using a sliding-window model [2] according to a feasible parallel pattern like the ones in [7]. To keep up with the arrival rate, the `AC` of the considered application may decide to increase the parallelism degree of such application in order to process input data faster. While the `AC` is in charge of reconfiguring the application to exploit additional resources (e.g., by spawning new processes/threads on-demand), the `FNC` is responsible for making the resources available to respond to the dynamic need of applications. To this end, the `FNC` is in charge of maintaining a complete vision of the node status (e.g., cores and cpu time available, memory utilization [3]), and of processing the requests of `AC` by finding feasible agreements. For example, if the `AC` requires the exclusive utilization of eight additional cores, the `FNC` can serve such request completely, if enough physical

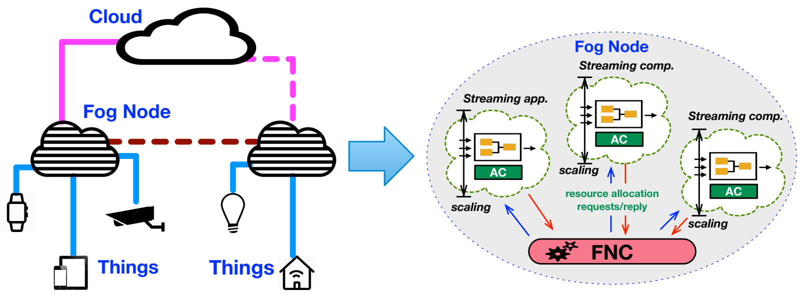

Fig. 1: Fog computing architecture and internal behavior of a fog node, managing several streaming applications or their parts.

resources are available. Otherwise, the `FNC` can partially serve the request of the `AC` by allocating fewer cores. As *extrema ratio*, the `FNC` may *unilaterally* release some cores previously assigned to other running applications to serve completely the request, by informing the corresponding `ACs` of the decision taken. This scenario is depicted in Fig. 1(right).

Inter-fog node scenario. Suppose that an application is a composition of two communicating components. The first (called *Filtering*) is a small graph of operators processing items produced by a set of data providers, by discarding inputs that are deemed to be irrelevant to the rest of the application. This component processes data items at high speed, thus it must exploit geographical proximity [19] with the data providers in order to leverage a reduced network cost. Instead, the *Selection* component runs a computationally demanding preference query like a skyline or a top-k query [23], in order to extract the best objects among the most recent data items received from the preceding phase.

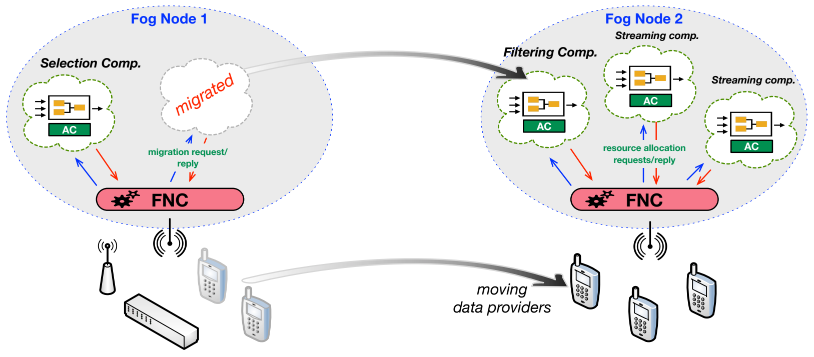

Fig. 2: Example of migration of a data stream processing component between Fog Nodes.

The infrastructure should be able to support the migration of streaming components from a fog node to another one properly chosen. This can be the result of an *internal* decision of the application itself, or *externally* triggered by the resource management control of the fog platform. As in the example of Fig. 2, the data providers feeding our *Filtering* component, which is initially deployed on `FN1`, are mobile devices that may enter in the proximity of `FN2` at a certain time instant. The corresponding `AC` that continuously monitors the component's QoS may experience too high network latency and/or insufficient network bandwidth. Therefore, the `AC` may opportunistically decide to ask the `FNC` of `FN1` to start the migration to a suitable fog node (e.g, `FN2` in this case). As a second case, the decision can be triggered by the infrastructure itself, for example if the `FNC` is unable to meet the resource utilization requests of the applications running in the first fog node, and some of them must be migrated to make further local resources available. In both cases, the underlying infrastructure should provide mechanisms for seamless migration with minimal intrusion and downtime in the processing flow.

3 Background: Docker

Container-based virtualization technology offers a lightweight virtualization which provides near-native performances [22]. Container-based virtualization leverages directly on the kernel features of the host OS for running multiple isolated user-space instances (called *containers*). Since containers share the same kernel of the host OS, container-based virtualization adds minimal overhead to the guest applications.

Docker [8] is the de-facto standard technology exploiting container-based virtualization. It provides the ability to package any application with all its dependencies (e.g., libraries, binaries, data files, etc.) into an isolated Docker *container*. Docker also *i)* permits limiting the resources assigned to a container in term of memory and CPU (by default, a container has no resource constraints), ans *ii)* it provides functionalities for checkpointing and restoring a running container by exploting *CRIU* [6, 9, 18].

A Docker container is created from a Docker *image*. From a single Docker image one or more Docker containers can be started. Docker also permits to look for existing images instead of building them from scratch. The images can be stored into *Docker registries* (e.g., Docker Hub [12]) where other users can retrieve and use them. Docker registries ease the distribution of images across different environment.

Docker containers can communicate by using Docker container networking [11]. Two containers attached to the same network can communicate with all other containers attached to the same network. Docker offers various network drivers depending if the containers reside on a single host or across a cluster of hosts. Standard sockets can also be used as low-level mechanisms for implementing a communication channel between containers.

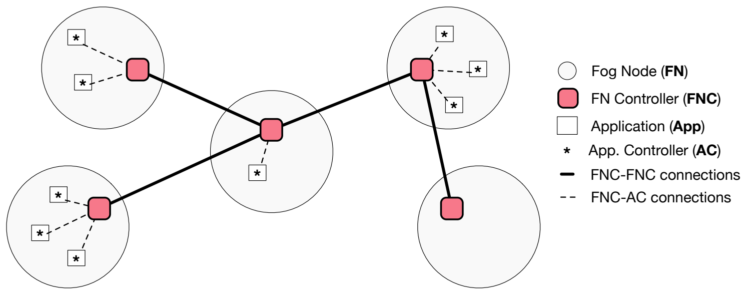

Fig. 3: An example of instance of the proposed architecture.

Docker has also built-in orchestration tools that allow to deploy multi-container applications. For instance, *Docker compose* [10] permits creating and managing Docker containers on a single host or in a cluster of hosts.

4 System architecture

We hereby illustrate the main concepts of the high-level architecture we envision. Such architecture is composed by four main components: Fog nodes (FNs), fog node controllers (FNCs), autonomic applications (Apps), and autonomic application controllers (ACs). A sample instance of our proposal is depicted in Fig. 3.

FNs are devices (e.g., smartphones, laptops, routers) with limited amounts of available computational resources, and they are in charge of running containerized Apps. Therefore, FNs must be able to decide whether an App can run on a FN, and how many computational resources to assign to such App (e.g., cores, CPU time, memory, bandwidth). This is why FNs are equipped with FNCs that are in charge of scheduling containerized Apps on FNs and of assigning to each App a certain amount of resources available in the hosting FN.

Each App runs in an independent container, or alternatively it can be split into various interacting components, each running in an independent container. Each AC running within a Docker container is also in charge of running the autonomic control loop of the corresponding App or component, and of interacting with the FNC of the corresponding FN to dynamically scale up/down the set of resources assigned to the Docker container, and/or to support the migration to another FN (or to the cloud).

Accordingly, FNCs will have to support both FNC-FNC and FNC-AC communications. FNC-FNC communications are inter-node, hence requiring to be network communications. FNC-AC communications are instead intra-node, hence they can be implemented with reduced communication latency by exploiting shared memory or domain sockets mechanisms. The latter seems to be more suitable, as FNCs and ACs run in Docker containers, and different Docker containers can communicate using shared socket files (see Sect. 3).

In the following, we detail the behaviour of the architecture during the execution of the scenarios sketched in Sect. 2, by distinguishing those only concerning fog nodes from those also including autonomic applications[1].

Fog nodes. Our architecture is designed to account for FNs freely joining or detaching from the system. Whenever a new FN is willing to join the system, its FNC must connect to one or more of the FNCs already available in the system (e.g., those of the "geographically closest" FNs, or those that can guarantee a desired response time). It must then communicate the computational resources available in the new FN, and this information will be taken into account (by all FNCs) when deciding how to schedule containerized Apps within/across FNs. At this point, the new FN is considered to be part of the overlay of FNs, hence being eligible for deploying containerized Apps on it.

Whenever a FN wishes to detach from the system, its FNC should communicate to the other FNCs that such FN is going to detach. This will result in disconnecting FNC of the detaching FN from the overlay of FNs, and in migrating all Apps running on the detaching FN to the other FNs in the system.

It is worth noting that a FN may detach from the system without priorly advertising the FNCs of the other FNs (e.g., because the corresponding device unexpectedly crashes or shuts-down), and this should also result in migrating all Apps that were running on the crashed FN to the other FNs in the system. To enable this, the availability of each FN will have to be monitored (e.g., with watchdogs or heartbeat services connected to its FNC).

Autonomic applications. Data stream processing applications will be deployable on the proposed architecture after being properly containerized as (possibly multi-container) Docker applications. The images of the containers forming an application will have to be available on a remote, publicly accessible Docker registry (e.g., Docker Hub [12]).

The administrator of an application can issue the deployment of her application by connecting to one of the FNCs in the system, and by indicating the Apps to be executed. The administrator indicates the Docker images used to run the Apps along with the deployment constraints of each App. For example, the administrator can constraint the App to be deployed on a certain subset of FNs, or she can specify that the App must be migrated to cloud whenever all the FNs do not satisfy the requested resources by the App.

The FNCs will then coordinate themselves to identify a FN satisfying the deployment constraints of an App, and they will inform the corresponding FNC to enact the deployment of such App. The FNC will then download the image of the App from the remote registry, it will start the App by running a Docker container from the downloaded image, it will assign an initial set of computational resources to the App, and it will start interacting with the AC to scale the resources assigned to the App (when necessary).

[1] Due to space limitations, we hereafter abstract from the actual policies to be employed for coordinating FNCs and for deciding how to schedule containerized Apps within/across FNs depending on available resources.

A FNC can scale up and down the set of resources assigned to an App (e.g., by decreasing/increasing the cores, CPU time, and bandwidth assigned to such App) by simply changing the resources assigned to the corresponding Docker container (see Sect. 3). This may be driven by exploiting reactive or predictive control policies [16], and it happens: when a FNC needs to remove some of the resources that were assigned to an App and to re-assign such resources to other Apps, or when an AC realizes that the App it is controlling requires less/more resources (e.g., to change the parallelism degree and adapt it to the data rate of the input stream). In the latter case, an AC sends a request to the FNC of the hosting FN, which decides how/whether to scale the resources assigned to the corresponding App.

It may happen that the computational resources available in a FN are no more capable of satisfying the requirements of all Apps running on it. If this is the case, the FNC of the overloaded FN will interact with the other FNCs in the system to decide which Apps can be migrated and on which FNs. To migrate them, it then send a migration request to the AC of each App to be migrated. The AC will then start preparing the migration by storing the current state of the App, and it will answer to the FNC by returning it the current state of the App. The FNC of the FN where the App must be migrated will then initiate the procedure for deploying such App, by exploiting the stored state of App as the initial application state.

It may also happen that no FN is capable of satisfying the requirements of a to-be-migrated App. If this is the case, the FNCs can decide to migrate an App to the cloud (with a migration approach very similar to that described above), or to reduce the resources assigned to an App as much as possible (if such App does not support fog-to-cloud migration).

Finally, an App can be undeployed from the system by simply informing the FNC of the FN where such App is running. This can either be done by the AC (if it realizes that the App has ended its tasks), or by the administrator of the App. The FNC will then just have to remove the corresponding Docker container, hence freeing the resources assigned to it.

5 Preliminary results

In this section we show two preliminary results aimed at illustrating that Docker can help deploying autonomic data stream processing applications in the Fog. First, we illustrate how Docker can be exploited by a FNC for limiting the physical resources (i.e., CPUs) assigned to a containerized App running on a FN. Second, checkpoint and restore features offered by Docker (version *17.03.1-CE*) are used to freeze and restore a containerized App on a FN[2].

Intra-fog node test. In the first test, we considered a FNC and an App running in Docker containers on a FN. The goal of the experiment is twofold: we show

[2] The source code of the experiments is available on *GitHub* https://github.com/di-unipi-socc/ffdocker.

how a `FNC` and the `AC` of an `App` can communicate on the same `FN`, and how the `FNC` can exploit Docker for limiting the CPUs assigned to such `App`. More precisely, the `App` and the `FNC` work as follows:

- the `App` is an autonomic application equipped with its `AC` that consumes the CPUs of the `FN` running the *cpuburn* application (`https://patrickmn.com/projects/cpuburn/`). The `AC` periodically sends a request to the `FNC` asking for increasing or decreasing a random number of the CPUs assigned;
- the `FNC` waits for incoming requests from the `AC` and (if available) increases or decreases the assigned CPUs to the `App`.

The `FNC` and the `App` reside on the same `FN` and they communicate using a *socket* file, where the `FNC` is the server and the `App` the client.

As we anticipated above, the `App` and the `FNC` are shipped in their own Docker containers and their images are stored in the *Docker Hub* registry[3]. The `App` is packaged into the `diunipisocc/app` image while the `FNC` is packaged in the `diunipisocc/fnc` image. In order to run the experiment, the `FNC` must be first executed by running the `diunipisocc/fnc` image with the following command:

```
docker run -v /tmp/ffsocket.sock:/tmp/ffsocket.sock
           -v /var/run/docker.sock:/var/run/docker.sock
           diunipisocc/fnc
```

When the `FNC` starts, it waits for requests listening on the `/tmp/ffsocket.sock` socket file. The `-v` option is used to mount a folder from the host into a container. Instead, the `/var/run/docker.sock` is the socket used by the `FNC` for interacting with Docker to update the CPUs assigned to the `App` container. The `App` can be launched by running the `diunipisocc/app` image:

```
docker run  -v /tmp/ffsocket.sock:/tmp/ffsocket.sock
               diunipisocc/app
```

The `App` mounts the `/tmp/ffsocket.sock` file for communicating with the `FNC`.

Fig. 4 (left) shows the result of the experiment executed on an Intel Linux machine with 48 cores. In the experiment, the `FNC` is configured to assign at most 20 cores to the `App` among the 48 cores available. The `App`, every 5 seconds, asks to the `FNC` to increase or decrease the cores assigned to it by a random number between 5 and 30. If the number of cores requested by the `App` are less or equal than 20, the `FNC` assigns to the `App` the cores requested, otherwise the `FNC` assign to the `App` at most 20 cores.

We measured the mean time required by the `FNC` to increase or decrease the cores assigned to a container. The time measured for updating the cores is about 80 microseconds with a standard deviation of 16 microseconds.

Inter-fog node test. In the second experiment we tested the possibility to use Docker for implementing containers live migration. The current version of

[3] The Docker images used to run the experiments are available in Docker Hub `https://hub.docker.com/u/diunipisocc/`

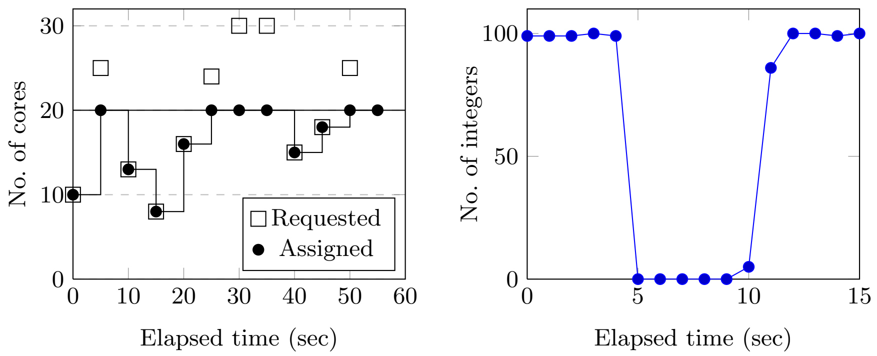

Fig. 4: Results obtained by running the *intra-fog node* experiment (left), and by running the *inter-fog node* experiment (right).

Docker only allows to checkpoint and restore a running container into the same host, whereas it does not support live migration across different hosts yet. There are other projects that implements live migration on top of CRIU [1], but they are not yet integrated with Docker.

The experiment reproduces a simplified version of the *inter-fog* scenario proposed in Sect. 2. The *Filtering* component sends an integer every 10 milliseconds (100 integers per second) to the *Selection* component that receives the stream of integers and prints them. *Selection*, *Filtering* and `FNC` run in their Docker container and they communicate via the default Docker bridge network (see Sect. 3). In our test we simulated the situation where the `FNC` checkpoints and restores the *Filtering* component in the same `FN`, evaluating the downtime experienced by the *Selection* component. This situation can happen, for example, if the `FNC` decides to temporarily suspend the execution of the *Filtering* component because it needs all the resources available on a Fog node to serve a higher priority request coming from another `App`.

The `FNC` triggers the migration of a component using the following steps:

1. The `FNC` sends a *migration request* to the *Filtering* component, notifying that the migration phase is willing to start.
2. The *Filtering* component receives the *migration request*, performs a clean up phase (e.g., it may notify the data sources to interrupt the data streaming), and sends a *migration reply* to the `FNC`.
3. The `FNC` receives the *migration reply* message and performs a checkpoint of the *Filtering* component,
4. Immediately after, the `FNC` restores the *Filtering* component into the same host and it continues to produce the stream of integers starting from the last checkpointed value.

The checkpoint of the *Filtering* saves both the application internal state (i.e. the last integer sent in the stream) and the sockets used for the communication. Fig. 4 (right) shows the result of the execution of the experiment in a single node. The *Selection* component receives 100 integers every second on average. After five seconds the *Filtering* component is forced to perform a migration by the `FNC`. The downtime experienced by the *Selection* component is about 5 seconds which is still significant though compliant with the measurements described in `https://criu.org/Performance_research`. However, the checkpoint and restore mechanisms of Docker are still under development and not yet officially released. We expect to see further optimizations in the next stable releases.

6 Related work

[19] proposes an architecture for processing streaming applications *near-to-the-edge*. The goal is to deploy latency-sensitive streaming operators near to the IoT devices that generate raw data streams. The infrastructure considers only two tiers: the first with traditional data centers and clouds, the second featuring *cloudlets* near to IoT devices. The application programmer defines which tier will preferably execute the distinct operators of a streaming application. With respect to our work, the distinction in two tiers seems restrictive, and the applications do not provide any elastic/autonomic support or capability.

Recently, techniques to map streaming applications onto IoT environments have received a considerable attention, because existing IoT platforms still lack of advanced features in terms of dynamic resource management and data privacy that are needed by the streaming context. IoT devices are often considered as mere data providers, at most enabled to filtering the data in order to save network bandwidth. [14] envisions an interesting approach that has several common points with our research. Container-based technologies are used to encapsulate streaming operators and to easily deploy them on a distributed environment. One of the aspects that distinguishes our approach is that each containerized application should have both the processing logic and the autonomic logic inside, the latter directly connected to our infrastructure management entities. This makes each running container an autonomous and adaptive entity, and not a static running code as in [14].

[20] presents *Foglets*, a programming infrastructure for managing geo-distributed awareness applications in the Fog. Based on the mobility of the sensors and the requirements of an application, the paper proposes both algorithms for deploying the application components on the fog nodes and techniques for handling the migration of these components between fog nodes. While, *Foglets* migrates applications whenever the resources they require are no more available in a Fog node, our approach tries to accomplish the application requirements by increasing or decreasing the resources available in a fog node before starting the migration phase.

A nice application scenario has been described in [4] for a urban video surveillance system deployed on a fog infrastructure. The approach follows a divide-and-conquer design, where raw data from IoT devices is filtered by applications running in Fog nodes and forwarded to a centralized cloud for processing. Although an interesting example, the utilization of the Fog infrastructure is limited and does not exploit the full potential of the paradigm.

Other recent papers mainly focus on extensions of the run-time support of existing and popular stream processing frameworks like Apache Storm and Flink, in order to make the frameworks able to deploy and run streaming applications in geographically distributed environments not limited to a single Cloud [13, 15]. Differently, our approach is focused around a two-level adaptation approach, where applications are themselves adaptive with their logic, interacting with our infrastructure for negotiating agreements in the resource utilization. Therefore, our approach is not limited to a single application running exclusively on the platform, and it is suitable to manage the execution of general applications and services, also outside the stream processing domain.

7 Conclusions

Fog computing is becoming a powerful enabler for IoT. Despite the growing interest, the implications and the advantages of Fog computing in streaming scenarios must still be explored and analyzed. Furthermore, the availability of new emerging virtualization concepts, like container-based technology, stimulates the research of new solutions for efficiently and flexibly deploy streaming applications in geographically distributed environments. In this paper we proposed a Docker-based architecture as an enabler for Fog deployment of autonomic applications. Besides the general overview of our idea, we presented also a concrete discussion of how the Docker technology can be exploited. Finally, first preliminary results confirmed our expectations about Docker as a viable approach for a new highly distributed and fog-oriented framework.

Acknowledgement. This work has been partially supported by the EU H2020-ICT-2014-1 project RePhrase (No. 644235).

References

1. Process HAULer. `https://criu.org/P.Haul`, last accessed: April 28th, 2017
2. Andrade, H., Gedik, B., Turaga, D.: Fundamentals of Stream Processing. Cambridge University Press (Sept 2014), cambridge Books
3. Bertolli, C., Mencagli, G., Vanneschi, M.: Analyzing memory requirements for pervasive grid applications. In: 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing. pp. 297–301 (Feb 2010)
4. Chen, N., Chen, Y., You, Y., Ling, H., Liang, P., Zimmermann, R.: Dynamic urban surveillance video stream processing using fog computing. In: 2016 IEEE 2nd International Conference on Multimedia Big Data (BigMM). pp. 105–112 (Apr 2016)
5. Chiang, M., Zhang, T.: Fog and iot: An overview of research opportunities. IEEE Internet of Things Journal 3(6), 854–864 (Dec 2016)
6. CRIU: Criu integration with docker. `https://criu.org/Docker`, last accessed: April 28th, 2017

7. De Matteis, T., Mencagli, G.: Parallel patterns for window-based stateful operators on data streams: An algorithmic skeleton approach. Int. J. Parallel Program. 45(2), 382–401 (Apr 2017)
8. Docker Inc.: Docker. `https://www.docker.com/`, last accessed: April 28th, 2017
9. Docker Inc.: Docker checkpoint command. `https://docs.docker.com/engine/reference/commandline/checkpoint/`, last accessed: April 28th, 2017
10. Docker Inc.: Docker compose. `https://docs.docker.com/compose/`, last accessed: April 28th, 2017
11. Docker Inc.: Docker container networking. `https://docs.docker.com/engine/userguide/networking/`, last accessed: April 28th, 2017
12. Docker Inc.: Docker hub. `https://hub.docker.com/`, last accessed: April 28th, 2017
13. Hochreiner, C., Vögler, M., Schulte, S., Dustdar, S.: Elastic stream processing for the internet of things. In: 2016 IEEE 9th International Conference on Cloud Computing (CLOUD). pp. 100–107 (June 2016)
14. Hochreiner, C., Vögler, M., Waibel, P., Dustdar, S.: Visp: An ecosystem for elastic data stream processing for the internet of things. In: 2016 IEEE 20th International Enterprise Distributed Object Computing Conference (EDOC). pp. 1–11 (Sept 2016)
15. Mehdipour, F., Javadi, B., Mahanti, A.: Fog-engine: Towards big data analytics in the fog. In: 2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech). pp. 640–646 (Aug 2016)
16. Mencagli, G., Vanneschi, M.: Qos-control of structured parallel computations: A predictive control approach. In: 2011 IEEE 3rd International Conference on Cloud Computing Technology and Science. pp. 296–303 (Nov 2011)
17. Pahl, C., Lee, B.: Containers and clusters for edge cloud architectures – a technology review. In: 2015 3rd International Conference on Future Internet of Things and Cloud. pp. 379–386 (Aug 2015)
18. Pickartz, S., Eiling, N., Lankes, S., Razik, L., Monti, A.: Migrating LinuX Containers Using CRIU, pp. 674–684. Springer International Publishing, Cham (June 2016)
19. Sajjad, H.P., Danniswara, K., Al-Shishtawy, A., Vlassov, V.: Spanedge: Towards unifying stream processing over central and near-the-edge data centers. In: 2016 IEEE/ACM Symposium on Edge Computing (SEC). pp. 168–178 (Oct 2016)
20. Saurez, E., Hong, K., Lillethun, D., Ramachandran, U., Ottenwälder, B.: 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. pp. 258–269. ACM (June 2016)
21. Shi, W., Dustdar, S.: The promise of edge computing. Computer 49(5), 78–81 (May 2016)
22. Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A.C., Peterson, L.L.: Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. SIGOPS Oper. Syst. Rev. 41(3), 275–287 (Mar 2007)
23. U, L.H., Mamoulis, N., Mouratidis, K.: Efficient evaluation of multiple preference queries. In: 2009 IEEE 25th International Conference on Data Engineering. pp. 1251–1254 (March 2009)