A Containerized Big Data Streaming Architecture for Edge Cloud Computing on Clustered Single-board Devices

Remo Scolati, Ilenia Fronza, Nabil El Ioini, Areeg Samir and Claus Pahl
Free University of Bozen-Bolzano, Bolzano, Italy

Keywords: Edge Cloud, Container Technology, Cluster Architecture, Raspberry Pi, Docker Swarm, Big Data, Data Streaming, Performance Engineering.

Abstract: The constant increase of the amount of data generated by Internet of Things (IoT) devices creates challenges for the supporting cloud infrastructure, which is often used to process and store the data. This work focuses on an alternative approach, based on the edge cloud computing model, i.e., processing and filtering data before transferring it to a backing cloud infrastructure. We describe the implementation of a low-power and low-cost cluster of single board computers (SBC) for this context, applying models and technologies from the Big Data domain with the aim of reducing the amount of data which has to be transferred elsewhere. To implement the system, a cluster of Raspberry Pis was built, relying on Docker to containerize and deploy an Apache Hadoop and Apache Spark cluster, on which a test application is then executed. A monitoring stack based on Prometheus, a popular monitoring and alerting tool in the cloud-native industry, is used to gather system metrics and analyze the performance of the setup. We evaluate the complexity of the system, showing that by means of containerization increased fault tolerance and ease of maintenance can be achieved, which makes the proposed solution suitable for an industrial environment. Furthermore, an analysis of the overall performance, which takes into account the resource usage of the proposed solution with regards to the constraints imposed by the devices, is presented in order to discuss the capabilities and limitations of proposed architecture.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

In the Internet of Things (IoT) almost everything generates data, of which a major part is stored or processed in a cloud environment. Furthermore, the amount of data generated or collected is growing exponentially, according to the IDC 2014 report on the Digital Universe (Turner, 2014). In this kind of situation, local (pre-)processing of data might be a better alternative to the current centralised cloud processing model. Our objective is to present a lightweight infrastructure based on the edge cloud computing model, which aims to provide affordable, low-energy local clusters at the outer edge of the cloud, possibly composed of IoT devices themselves. The proposal explored here is thus a small low-power, low-cost cluster of single board computers in a local network, which is able to pre-process generated data at the outer edge. The aim is to present a model which is scalable and affordable, with a low power footprint, suitable for industrial applications in an IoT environment.

The proposed edge cloud architecture (Pahl et al., 2018b) makes use of technologies commonly used in processing of Big Data, which has similar characteristics of high speed, high variety, and high volume, when scaled down to the constraints imposed by IoT devices, i.e., Apache Hadoop and Apache Spark. The system is based on the containerization software Docker, which provides the means of orchestrating services on a device cluster. To analyze the performance of the system (Heinrich et al., 2017), with regards to the constraints imposed by the Raspberry Pi, system metrics are collected through a monitoring stack based on Prometheus, a monitoring and alerting tool, deployed on the cluster.

Raspberry Pi-based architectures have already been investigated for IoT and edge settings (Tso et al., 2013; Hentschel et al., 2016; von Leon et al., 2019; Pahl et al., 2017), but a performance exploration for an industry-relevant setting is still missing. Furthermore, the limits of a flexible, i.e., software-defined, virtualised architecture for software management need to be investigated.

Our paper details how the proposed solution can be implemented, using a cluster of Raspberry Pis, a

Scolati, R., Fronza, I., El Ioini, N., Samir, A. and Pahl, C.
A Containerized Big Data Streaming Architecture for Edge Cloud Computing on Clustered Single-board Devices.
DOI: 10.5220/0007695000680080
In *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER 2019)*, pages 68-80
ISBN: 978-989-758-365-0
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

single board computer (SBC). We show how the proposed system can be implemented, using Docker to deploy and orchestrate lightweight containers which run an Apache Hadoop and Apache Spark cluster, used to process data with a test application. Furthermore, we demonstrate how the cluster can be used as test bed for such applications, gathering meaningful system metrics through a monitoring system based on Prometheus. We conclude with an evaluation, taking into account the complexity of managing the cluster as well as the overall performance.

2 REQUIREMENTS AND APPLICATION SCENARIOS

In a typical cloud computing scenario, the usual approach is to collect data and to send it to a centralized system where storage space and computational power is available, without any selection or analysis. With increasing complexity and also having to deal with increasing amounts of data generated in IoT environments, a centralised cloud infrastructure is not ideal from a latency, cost and reliability perspective.

The aim of this work is to evaluate the feasibility of an alternative low-cost and lightweight approach based on the fog or edge cloud computing model. The main requirement is to be able to collect, process, and aggregate data locally, using affordable devices, such that the overall amount of traffic and the need for a backing cloud infrastructure can be reduced. The costs for acquisition, maintenance and operation of such a system and its suitability in an industrial setting will also be considered as well as the overall performance of the system with regards to the limited power at disposal. The chosen software platform needs to reflect a industry-relevant setting in terms of software deployment or big data processing.

The main use cases for such an application range from the IoT domain, where large amounts of data are generated and have to be processed, to autonomous monitoring and automation systems, like remote localised power grids. Additionally, the ability to process data locally with a low-cost and low-power system opens up use cases in environments without large amounts of processing power at disposal on premise, which would benefit from a decreased traffic, like systems in remote areas. Possible application scenarios include autonomous power generation and distribution plants, such as smaller local energy grids, or equivalent distributed systems in rural areas.

Such systems could benefit from the use of single board computers like the Raspberry Pi (Johnston et al., 2018) due to the possibility of connecting sensors to the devices GPIO paired with the capability to perform more complex computations, thus creating a network of smart sensors capable of recording, filtering, and processing data. The nodes can be joined together to a cluster to distribute the workload in the form of containers, possibly having separate nodes responsible for different steps of the data pipeline from generation and collection to evaluation of the data in a big data streaming and analytics platform. This would lead towards a microservices-style architecture (Jamshidi et al., 2018), (Taibi et al., 2018) allowing for flexible software deployment and management.

The relevant technologies for a lightweight cluster infrastructure (Raspberry Pis), a container-based software deployment and orchestration platform (Docker swarm) and a big data streaming application architecture are introduced in the next section.

3 BACKGROUND INFORMATION

The three platform technologies – Raspberry Pis, Docker containers and Hadoop – shall be introduced.

3.1 Raspberry Pi

The Raspberry Pi is a single-board computer, first introduced in early 2013. The computer was initially developed as an educational device, but soon attracted attention from developers due to the small size and relatively low price (RPi Foundation, 2018). Since the first model was well received, there have been multiple updates of the platform. In this project, the Raspberry Pi 2 Model B, released in 2015, is used. The specifications are shown in Table 1.

Table 1: Specification of the Raspberry Pi 2, Model B.

Architecture	ARMv7
SoC	Broadcom BCM2836
CPU	900 MHz quad-core 32-bit ARMCortex-A7
Memory	1 GB
Ethernet	10/100Mbit/s

3.2 Docker

3.2.1 Architecture

Docker, first released in 2013, is an open source software project, which allows to run containerized applications (Docker, 2018). A container is a runnable instance of a Docker image, a layered template with instructions to create such a container. A container holds everything the application needs to run, like

system tools, libraries and resources, while keeping it in isolation from the infrastructure on which the container is running, thus forming a kind of virtualisation layer. This is achieved by compartmentalizing the container process and its children, using the Linux containers (LXC) and libcontainer technology provided by the Linux kernel via kernel namespacing and control groups (cgroups), in fact isolating the process from all other processes on the system, while using the hosts kernel and resources. The major difference between containers and virtual machines is that containers, sharing the hosts kernel, do not necessitate a separate operating system, resulting in less overhead and minimizing the needed resources. Figure 1 illustrates Dockers architecture.

The access to system resources can be individualized for each container, thus allowing access, among other devices, to storage, or, in the case of the Raspberry Pi, to the general-purpose input/output pins (GPIO) for interaction with the environment using sensors or actuators.

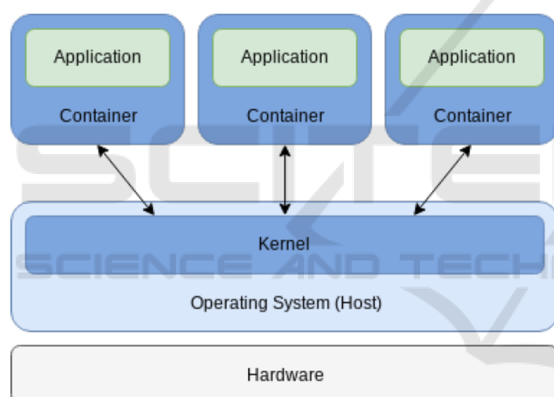

Figure 1: Docker container architecture.

3.2.2 Docker Engine

Docker Engine[1], the Docker application, is built using a client-server architecture. The Docker daemon, which acts as the server, manages all Docker objects, i.e., images, containers, volumes and networks. The client, a command line interface (CLI), communicates with the daemon using a REST (representational state transfer) API (application programming interface).

3.2.3 Docker Swarm

Docker 1.12, released in mid 2016, and all subsequent releases, integrate a swarm mode[2], which allows natively to manage a cluster, called swarm, of Docker engines. Docker swarm mode allows to use the Docker CLI to create and manage a swarm, and to deploy application services to it, without having to resort to additional orchestration software. A swarm is made of multiple Docker hosts, running in swarm mode and acting as manager nodes or worker nodes. A host can run as manager, worker, or both. When a service is created, the number of replicas, available network and storage resources, exposed ports, and other configurations are defined. The state of the service is being maintained actively by Docker, which means that if, for instance, a worker node becomes unavailable, the tasks assigned to that node are scheduled on other nodes, thus providing fault-tolerance. A task here refers to a running container that is run and managed by the swarm, as opposed to a standalone container. Figure 2 illustrates an example of such a Docker swarm configuration.

[1]Docker Engine, https://docs.docker.com/engine
[2]Docker Swarm, https://docs.docker.com/engine/swarm/

3.3 Hadoop

3.3.1 MapReduce

MapReduce is a programming model for processing big data sets, which allows to use a distributed, parallel algorithm on clustered devices (Dean and Ghemawat, 2004). A MapReduce program consists of a map method, which performs sorting and filtering of the data, and a Reduce method, which performs an associative operation. Although inspired by the map and reduce methods commonly used in functional programming, the main concern of the MapReduce framework is the optimization of the underlying engine, achieving scalability and fault tolerance of the applications implementing it.

Both the Map and Reduce operation of MapReduce are performed on structured data, which takes the form of (key, value) pairs. The Map function is applied to every pair (k1, v1) of the input in parallel, producing a list of pairs (k2, v2). After this first step, all pairs with the same key k2 are collected by the MapReduce framework, producing one group (k2, list(v2)) for each key. Then, the Reduce function is applied to each group in parallel, producing a list of values v3. The whole process is illustrated below:

```
1   Map(k1,v1) list(k2,v2)
2   Reduce(k2, list(v2)) list(v3)
```

3.3.2 Hadoop

The Apache Hadoop framework is an open source software library, which allows to process large datasets in a distributed way, using the MapReduce programming model (Apache, 2018). It is designed

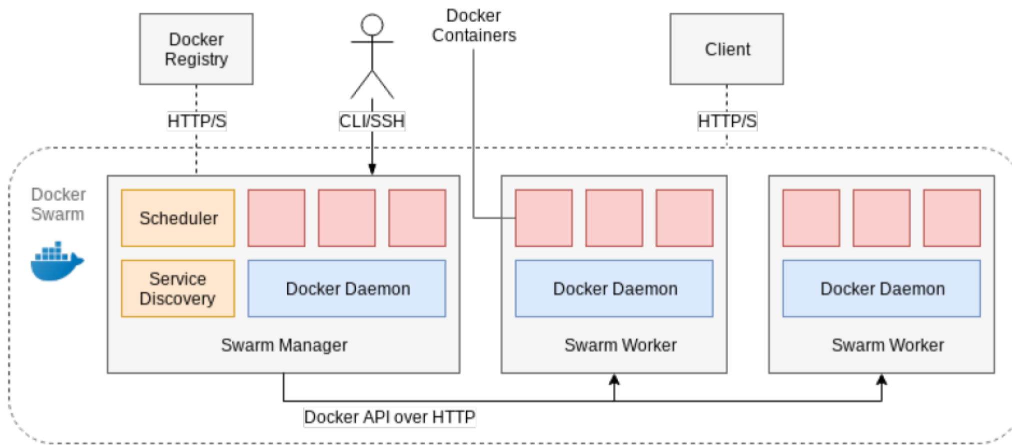

Figure 2: Docker Swarm Configuration Sample.

to scale from single nodes to clusters with multiple thousands of machines (Baldeschwieler, 2018). The following modules make up the core Apache Hadoop library:

- Hadoop Common – common libraries & utilities;
- Hadoop Distributed File System (HDFS) – distributed file system to store data on cluster nodes;
- Hadoop YARN – managing and scheduling platform for computing resources and applications;
- Hadoop MapReduce – large scale data processing implementation of the MapReduce programming model.

A typical, small Hadoop cluster includes a single master node and multiple worker nodes. The master node acts as a task and job tracker, NameNode (data index), and DataNode (data store) , while the worker nodes act as task tracker and DataNode. Figure 3 illustrates such a cluster.

3.3.3 HDFS

The base of the Hadoop architecture consists of the Hadoop Distributed File System (HDFS) and a processing part, which implements the MapReduce programming model. Files are split into blocks of data and distributed across the DataNodes. Transferring a packaged application on the same nodes, Hadoop takes advantage of the principle of data locality. Since the nodes manipulate the data they have access to, Hadoop allows for faster and more efficient processing of the dataset than more traditional supercomputer architectures (Wang et al., 2014).

3.3.4 Apache Spark

Apache Spark is an open source framework for distributed computing, which provides an interface for executing applications on clusters (Apache, 2018). The framework uses as its foundation a resilient distributed dataset (RDD), a distributed and fault-tolerant set of read-only data items. As an extension to the MapReduce paradigm, Sparks RDDs offer a limited form of Hadoop distributed shared memory for distributed programs. Allowing a less forced dataflow compared to the MapReduce paradigm, Apache Spark facilitates the implementation of programs which can reduce the latency, compared to an Apache Hadoop implementation, by several orders of magnitude.

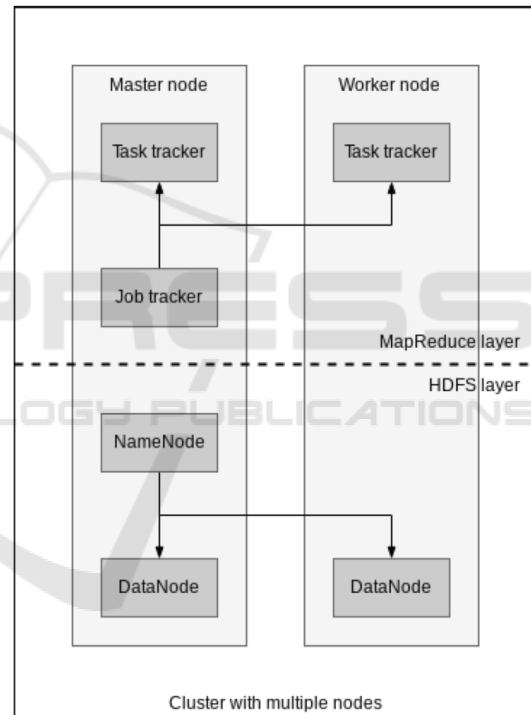

Figure 3: Hadoop cluster.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Apache Spark requires a cluster manager – supported implementations are Spark native, Apache Hadoop YARN, and Apache Mesos clusters – and a distributed storage system. Supported systems are, among others, HDFS, Cassandra, and Amazon S3, or a custom solution might be implemented. Furthermore, in extension to Apache Hadoop's batch processing model, Apache Spark provides an interface to perform streaming analytics, where data is continuously processed in smaller batches. Spark supports distributed storage systems, TCP/IP sockets, and a variety of data feed providers such as Kafka and Twitter as streaming sources.

4 STATE OF THE ART REVIEW

Lightweight virtualization solutions based on containers have continuously gained in popularity during the last years. Their lightweightness make them in particular suitable for computing at the outer edge, where limited resources are available, but still data originating from the IoT layer needs to be processed:

- Overhead: Compared to native processes, Docker container virtualization has been shown to not add significant overhead by leveraging kernel technologies like namespaces and control groups, allowing the isolation of processes from each other and an optimal allocation of resources, such as I/O device access, memory and CPU (Renner et al., 2016), (Morabito, 2016).
- IoT/Edge Applicability: Recent research has proved that Docker is an extremely interesting solution for IoT and Edge Cloud applications, where, due to the constraints imposed by low-power devices, it allows lightweight virtualization and facilitates the creation of distributed architectures (Pahl and Lee., 2015), (Morabito et al., 2017).
- Big Data: In addition, Docker is suitable for provisioning Big Data platforms, for instance Hadoop and Pachyderm, helping overcome difficulties in installation, management, and usage of large data processing systems (Naik, 2017). Such systems, as for instance Hadoop, have been shown to be a convenient solution for pre-processing large amounts of data also on small clouds with limited networking and computing resources (Femminella et al., 2016).

The recent development of affordable single board computers allows to create low-power and low-cost clusters for IoT environments, offering the capability of pre-processing sensor data, while keeping acquisition and maintenance costs low (von Leon et al., 2018). Devices such as the Raspberry Pi, which offer the possibility to attach sensors and actuators to its GPIO pins, are particularly interesting for any system which gathers, processes, and reacts to environment data of some form, such as smart infrastructures and smart sensor networks, as has been shown by the research done at the University of Glasgow in the Smart Campus project (Tso et al., 2013), (Hentschel et al., 2016). The team behind HypriotOS, a Linux based operating system created with the aim of making the Docker ecosystem available for ARM and IoT devices, showed that container orchestration tools like Docker Swarm and Kubernetes can be used on Raspberry Pis to create highly available and scalable clusters (Renner, 2016a), (Renner, 2016b).

However, an exploration of the limits of putting a containerised big data streaming application on a lightweight cluster architecture is still lacking. Thus, we have built such an architecture and evaluated it in terms of cost, configuration and performance aspects. An important concern for us was the construction of industry-relevant container management (Docker) and monitoring (Prometheus) tools.

5 LEIGHTWEIGHT EDGE DATA PROCESSING PLATFORM

The proposed system is built on top of a Raspberry Pi cluster. The devices are part of a Docker swarm, leveraging the ease of container orchestration on multiple devices provided by the same. All applications, i.e., the Apache Hadoop and Apache Spark cluster, the Prometheus monitoring stack and the applications used to simulate data collection, are executed inside Docker containers, simplifying the deployment and management of the services, even in case of a hardware failure. The Hadoop distributed file system (HDFS) is used as data source for an Apache Spark streaming application. The data is provided by a Nodejs[3] application, which writes files to the HDFS via its API. An overview of the architecture is shown in Figures 4 and 5. Figure 4 shows the architecture of the implementation and the data flow during the experiments, while Figure 5 shows an overview of the distribution of the services on the Raspberry Pis in the configuration used for the experiments.

[3]Nodejs, https://nodejs.org

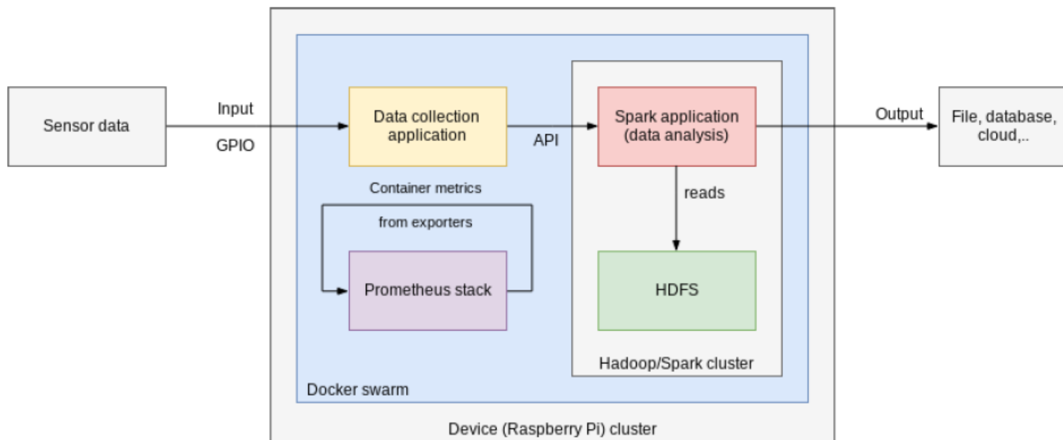

Figure 4: System Architecture.

5.1 Hardware Architecture

For the cluster, eight Raspberry Pi 2 Model B are used, each fitted with an 8 GB micro SDcard for the installation of the OS. The Raspberry Pis are connected to a Veracity Camswitch 8 Mobile switch[4], which is also used to power the devices through POE (Power over Ethernet). The switch features ten 10/100Mbit/s ports, eight of which are 802.3 at POE outputs. The Raspberry Pis are connected to the switch using category 5E SFTP cables.

POE has the advantage of the setup being cleaner, since a separate power supply is not needed for the Raspberry Pis and as such less cables run across the system, although this is neither necessary nor cost efficient. Since the Raspberry Pi does not provide the necessary connectivity for POE, the devices need to be outfitted with an additional POE module, connected to the Raspberry Pis through the GPIO pins.

5.2 Software Architecture

5.2.1 Operating System

The Raspberry Pis run Hypriot OS[5], a specialized Debian distribution, as their operating system. The distribution comes with Docker pre-installed, and is a minimal operating system optimized to run Docker on ARM devices. Hypriot OS is available as an image which can be flashed, ready to use, on a micro SD card. The OS provides a pre-boot configuration file, allowing, among others, to set a host name, which is used for Avahi local host name resolution.

The OS comes with a pre-installed SSH service, accessible through the configured credentials. Since password authentication is not secure, all nodes are set up to use public key authentication. To automate the setup of the nodes, like configuring automatic mounting of drives and authentication, or installing additional software, a configuration management tool, namely Ansible[6], is used since it allows to define the hosts and automate configuration tasks via SSH.

5.2.2 Swarm Setup and Management

The Docker engine CLI is used to initialize and setup the swarm. To initialize it, a single node swarm is created from one of the nodes, which becomes the manager for the newly created swarm. The manager stores join tokens for manager and worker nodes, which can be used to join other machines to the swarm. Furthermore, different tags can be set through the Docker CLI on each node, which are used to constrain the deployment of services to specific nodes. To perform other swarm management tasks, for instance promotions, demotions and to manage the membership of nodes, Docker engine commands can be issued to any one of the swarm managers.

5.2.3 Service Deployment

To deploy services to the swarm, Docker stack deployment is used, which allows to deploy a complete application stack to the swarm. To describe the stack, Docker uses a stack description in form of a Compose

[4]Veracity Global Camswitch 8, http://www.veracityglobal.com/products/networked-video-integration-devices/camswitch-mobile.aspx

[5]Hypriot OS, https://blog.hypriot.com/about/

[6]Ansible, https://www.ansible.com/

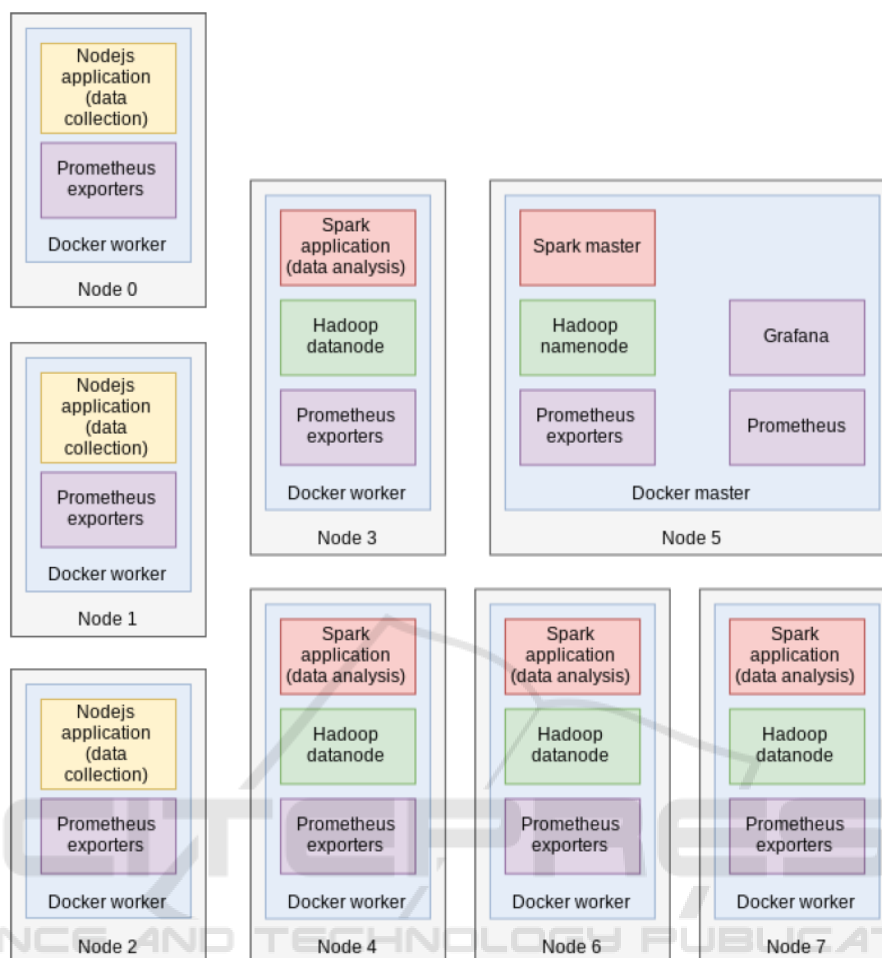

Figure 5: Overview of the distribution of the service containers on the nodes.

Table 2: Service allocation on the cluster nodes.

Node	Container
Node 0	Data collection, Prometheus exporters[7]
Node 1	Data collection, Prometheus exporters
Node 2	Data collection, Prometheus exporters
Node 3	Spark worker 2, Prometheus exporters
Node 4	Spark worker 1, Prometheus exporters
Node 5	Spark master, Prometheus exporters, Prometheus, Grafana
Node 6	Spark worker 3, Prometheus exporters
Node 7	Spark worker 4, Prometheus exporters

file (version 3), where multiple services are defined. For each service that is part of the stack, the origin registry, ports and networks, mounted volumes, service name and replicas as well as deployment constraints, for example Docker node tags, can be specified. When deployed on a manager node, Docker will deploy each service in the stack to the nodes of the swarm, according to the constraints and definition, balancing out the containers on the available nodes.

To deploy the compose file to the swarm manager and to set up the nodes, for example for the preparation of configuration files and mounted directories, Ansible scripts were used during this project, but the actions can be performed manually or with any other scripting language.

5.2.4 Hadoop and Apache Spark

Although it is possible to use Hadoop natively to create clusters of computers, both Hadoop as well as Apache Spark are deployed inside Docker containers here in order to streamline the deployment process and in order to avoid the installation of the software and the management of all the required dependencies on each device.

A custom Docker image is used to install both Hadoop and Apache Spark inside a container and for

[7]Prometheus exporters: Armexporter, Node exporter, cAdvisor.

the set-up of the environment. In the final set-up one master node and four separate worker nodes are deployed. Since the worker nodes act as DataNodes for the cluster, each worker node container is provided with sufficient storage space, by means of a standard 3.5 inch 1TB hard disk, mounted as a volume.

Hadoop and Apache Spark are deployed to the cluster, together with the data collection application used to evaluate the implementation, through a Docker Compose file.

6 EDGE DATA PROCESSING AND MONITORING

6.1 Data Collection and Analysis

The architecture will be evaluated using two applications, one for simulating data collection on part of the nodes and a second one is deployed to the Apache Spark cluster to process the collected data. The applications are also used to gather metrics related to the performance of the system, namely the processing time of submitted data.

Data Collection. In order to simulate data collection, a Nodejs application is used to send an HTTP PUT request with the data content to the HDFS API creating a file in a specified interval of time. The data collection application is deployed on the remaining three nodes of the cluster through a Compose file.

The experiment data are sample files exposed to and read by the application over the network allowing to change the size and the contents without having to modify the application and re-deploy it.

Data Analysis. For the analysis of the collected data, a Python application is used. The application polls for newly created files every second and performs a count of single occurrences of words in the files. The application is derived, with minimal changes, from a sample application provided with the Apache Spark engine[8].

The used implementation follows the classical MapReduce model, as described earlier, creating a list of (key, value) pairs of the form (word, 1) for each occurrence of a word. This list is then grouped by keys, in this case by words, and the Reduce step is then applied on each group, generating the sum of occurrences of each word. These last two steps are performed by one function, reduceByKey. The listing below illustrates the algorithm in pseudocode with the sample MapReduce implementation used as the test application.

```
1 lines
2 count :=
    lines.flatMap(lambda(line){line.split("")})
3     .map(lambda(word){(word, 1)})
4     .reduceByKey(lambda(val, acc){val + acc})
5 counts.pprint()
```

The application can be deployed to the Spark cluster using the Spark CLI tool sparksubmit to send it to the master node, which then distributes the application to the workers in the cluster.

6.2 Monitoring

In order to facilitate monitoring of the system, a Prometheus[9]-based monitoring stack is deployed to the cluster, composed of three different components.

Firstly, Prometheus is an open source monitoring and alerting system which uses a time series database, and provides many integrations with other software such as HAProxy, the ELK stack (Elasticsearch, Logstash and Kibana) or Docker.

Secondly, since the operational model of Prometheus is to pull metrics from services, the stack uses various so-called exporters to collect metrics and expose them to be collected by Prometheus. Many ready-to-use exporters exist, interesting for this immplementation are those for Docker and system metrics. The exporters used are cAdvisor[10], a daemon which collects and exposes container resource usage, and Node exporter[11], part of the Prometheus project, which exports hardware and system metrics exposed by the Linux kernel.

Thirdly, to visualize and review the collected data, Grafana[12] is used. Grafana is a tool for monitoring and analyzing metrics, which allows to create dashboards exposed via a web user interface (UI). This solution was chosen because it requires little configuration, for a use case involving the monitoring of Docker containers on multiple Linux machines. Thus an almost ready-to-use implementation, which can be deployed on the Raspberry Pis, was used. The implementation takes advantage of Docker containers, and the stack is deployed to the Docker cluster using a custom Compose file.

In the final setup, Prometheus and Grafana are both deployed on one of the nodes, while one instance

[8] Apache spark example applications, https://github.com/apache/spark/tree/master/examples

[9] Prometheus, https://prometheus.io/

[10] Google cAdvisor, https://github.com/google/cadvisor

[11] Node exporter, https://github.com/prometheus/node_exporter

[12] Grafana, https://grafana.com/

each of cAdvisor and Node exporter is deployed on every Raspberry Pi.

7 EVALUATION

We start with some observations on practical configuration and management concerns in Subsection 7.1 before addressing performance in the following performance evaluation in Section 7.2.

7.1 Set-up and Practical Considerations

We evaluate here the practical effort needed to set up and manage the described cluster architecture.

Both the hardware setup and the preparation for running the software do not require any special skills. The Micro SD cards can be flashed using a dedicated tool, which is available for any major platform. The Hypriot OS maintainers provide information on the pre-boot configuration file for the nodes operating system on their website, and from there, many guides are available for setting up a connection over SSH using public key authentication. Scripts for setting up, and later managing, the Docker swarm were prepared to automate time consuming tasks, but all the steps required can also be performed manually. Since the setup requires to run the same commands on all nodes, for example to install the required dependencies, or to join a newly created swarm, using a configuration management tool, or preparing some shell scripts, should be considered. Once the cluster is running, the Docker documentation provides support needed to manage the swarm or deploy services to the same. The services are deployed using Docker compose files, and running a command on one of the master nodes.

The following observations on the main evaluation criteria can be made:

- **Maintainability and Ease of Operation**. Using containerized services on a Docker swarm is crucial for achieving maintainability and ease of operation of the implementation.
- **Fault-tolerance**. Furthermore, the use of Docker requires only minimal overhead, but increases the fault tolerance of the system, since containers failing due to hardware or software problems can be easily restarted on any other node, which is taken care of by the cluster itself. Thus, Docker is ideal for achieving high availability and increasing the fault tolerance.
- **Image Building and Build Times**. It has to be kept in mind that for applications such as Apache Hadoop and Apache Spark, there might not always be a Docker image built for the ARM architecture. The continuous integration and continuous delivery (CI/CD) service provided by Gitlab[13] was used by us to build the required images. Since the containers used to build the images run on a different instruction set than the Raspberry Pis, a system emulator, QEMU[14], was used inside the Dockerfiles to cross-build images. Even though there are some problems with this approach due to the Java Virtual Machine (JVM) displaying bugs during the builds, making it necessary to run some preparatory commands manually on the already deployed services, it has been preferred to building the images on one of the cluster nodes due to the longer build times. Table 3 shows the build times of one of the used images on a common notebook[15], compared to the build times of the same image on a Raspberry Pi used in the cluster, to which the time to distribute the image on the nodes has to be added. The pull of the image from the Gitlab registry took around 6 minutes.

  While these build times seem high, this is an activity that is not frequently needed and can generally be tolerated.

Table 3: Comparison of Docker image build times.

Target architecture	Build architecture	Build time
armv7	x86_64	6m 1s
armv7	armv7	14m33s

7.2 Performance

The main performance evaluation focus shall be on experimentally evaluating data processing time und resource consumption, based on input data file sizes that reflect common IoT scenarios with common small-to-midsize data producers.

Running the test applications required some fine tuning of the resources allocated to the Spark executors, influenced by the constraints imposed by the Raspberry Pis. The only relevant results were achieved by allocating 500MB of memory to each executor, since the Spark process would generate out-of-memory exceptions when using less, while on the other hand, if given more memory, the processes would starve one of the components necessary for the Hadoop/Spark cluster. This memory allocation can be set during the submission of a job.

[13]Gitlab, https://about.gitlab.com/

[14]QEMU, https://www.qemu.org/

[15]The model used is a HP 355 G2, with AMD A8-6410 2GHz CPU and 12GB memory.

7.2.1 File Processing Times

The delay between the submission of a new file and the end of the analysis process was measured by comparing the submission time with the time the output was printed to the standard output stream (stdout) of the submission shell, using time stamps for both.

Table 4 shows the results measured for files with around 500B and 1KB, respectively, submitted once a second, which reflects as explained a common small-to-midsize data production volume. In both cases, the data analysis application was polling for new files every second, and files were submitted at a rate of one per second.

The test cases aimed at exploring the limits of the RPi-based container cluster architecture for certain IoT and edge computing settings. The delays shown in Table 4 can be considered too high for some real-time processing requirements, e.g., for any application which relies on short times for immediate actuation, and are unexpectedly high considering the file size and the submission rates used during the test runs. However, if only storage and analysis without immediate reaction is required or the sensors produce a limited volume of data (such as temperature sensors), then the setting would be sufficient.

Table 4: File processing duration from time of file creation.

File size	Polling time	Delay
1.04KB	1s	236s
532B	1s	61s

7.2.2 Resource Consumption

Table 2 earlier showed how the various services were distributed among the nodes during the execution. The data recorded by the monitoring stack deployed shows no increase of the used resources during file submission and analysis.

CPU. Figure 6 shows the CPU time use (by node) during the execution of the test application, while Figure 7 shows the same data, divided by container. As could be expected, the graphs show an increase of the CPU workload on all Apache Spark nodes, in particular on the worker nodes of the cluster. The analysis application was submitted from node 3 (Spark worker 2), which explains the higher CPU use compared to the other nodes at point 0 in the graphs, while data collection was started two minutes in, at the 120 seconds mark, as can be seen by an increased CPU utilization by the Spark master node. Data collection and data analysis were stopped at 420 seconds and 480 seconds, respectively. The shown records are for the 532 Bytes test file, see also Table 4.

Memory. Similar results can be seen in Figure 8 and Figure 9, showing the memory use of the containers in Spark and the monitoring stack during the experiment divided by node and by container, respectively. As expected, we can observe that the memory usage by the Spark node used to submit the application, and which therefore acts as controller for the execution and collects the results from all other nodes, is higher than on the other nodes.

Not shown is the resource usage by the remaining system (e.g., the Docker daemons and system processes), which were constant during the test, with CPU time below 2% and memory use around 120MB on each node.

7.2.3 Analysis and Discussion

The graphs show that the system resources are not used optimally, with room to spare both on the CPU and in particular on the memory of the devices. The results suggest that the delay is due to how the distribution and replication of small files[16] is performed by HDFS, causing a suboptimal use of Sparks streaming capabilities and the systems resources.

Apart from the rather high delays and a suboptimal resource usage, the recorded data shows an uneven CPU utilization by the Spark master container, reaching 0% around the 250 seconds mark, which was consistently irregular throughout all the test runs. This might be due to the file system, or rather HDFS, starving the process due to high I/O times. Here, further experiments with alternative sources might explore and confirm reasons with more certainty.

Regarding possible hardware performance improvements, the Raspberry Pi 3 might here be a better option than a Pi 2. While in comparison the CPU only gains 300MHz, it also updates its architecture from a Cortex-A7 set to a Cortex-A53 one, which is an architecture boost from 32-bit to 64-bit, thus resulting in much better performance of a factor 2 to 3[17]. In terms of RAM, where the Raspberry Pi 2 has 450MHz, the Pi 3 has 900MHz RAM.

In conclusion, the aim of this investigation, was to determine some of the limits of the proposed infrastructure. Depending on the concrete application in question, our configuration might however still be sufficient or could be improved through better hardware or software configuration. Furthermore, a controller allowing self-adaptation (Jamshidi et al., 2016) to address performance and resource utilisa-

[16] A typical block size used to split files by HDFS is 64MB.

[17] https://www.jeffgeerling.com/blog/2018/raspberry-pi-3-b-review-and-performance-comparison

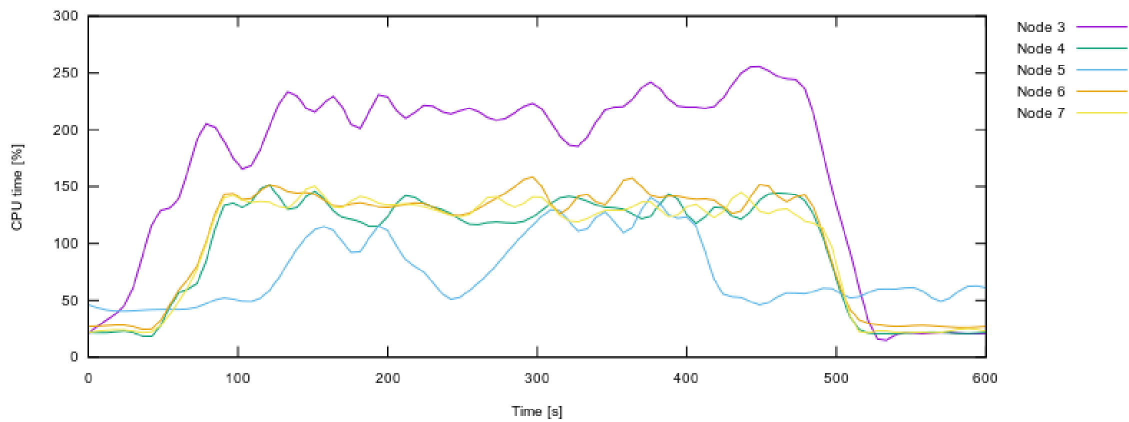

Figure 6: Container CPU time, by node.

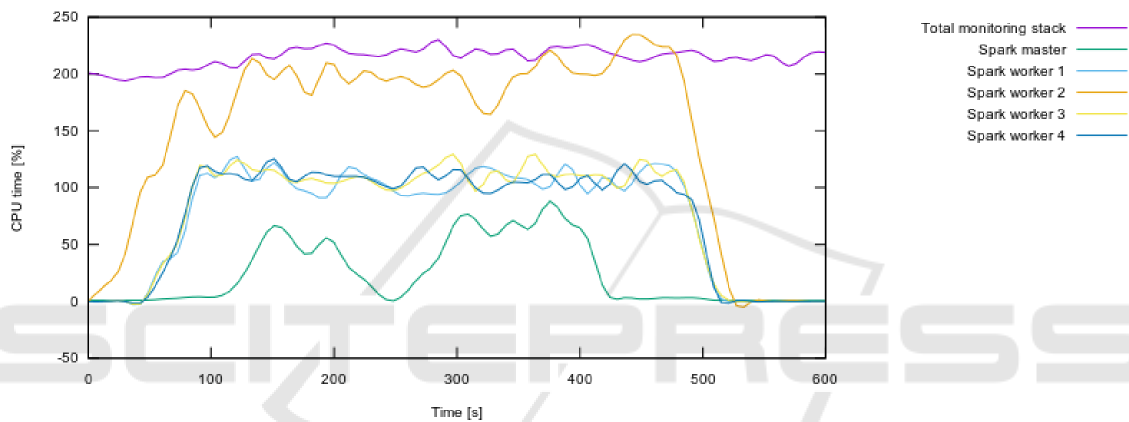

Figure 7: Container CPU time, by container.

tion anomalies of the platform configuration might be a solution for the future here that can address sub-optimal resource utilisation (Jamshidi et al., 2016; Jamshidi et al., 2015).

8 CONCLUSION

A lightweight containerised cluster platform can be suitable for a common range of IoT data processing applications at the edge. Leveraging the lightweight containerization and container orchestration capabilities offered by Docker, allows for an edge computing architecture that is simple to manage and has high fault-tolerance. Due to the Docker swarm actively maintaining the state of the services, Docker provides high availability of services, making the cluster self-healing, while also making scaling simple.

The low energy and cost impact of single board devices, while still being able to run complex infrastructures by means of clustering, are promising with regards to the overall reduction of infrastructure costs even in the presence of high volumes of data. Even though it has been shown that it is possible to implement a Big Data system on device clusters with strict constraints on networking and computing resources like the Raspberry Pis to create a low-cost and low-power cluster capable of processing large amounts of data, the actual performance of at least our prototype system has limits. For instance, a performance limitation of the prototype implementation was caused by the choice of the Hadoop distributed file system (HDFS) as source for data streams of small files.

In practical terms, for our platform configuration, most of the images used were created from scratch, or at least heavily customized from similar implementations to meet the requirements. Nonetheless, the number of projects targeting ARM devices are growing in popularity, for instance, the monitoring stack, which allowed us to use the system as a performance test bed for the implementation and applications, was available as a ready-to-use Docker compose file which could be deployed to the cluster with just some configuration changes.

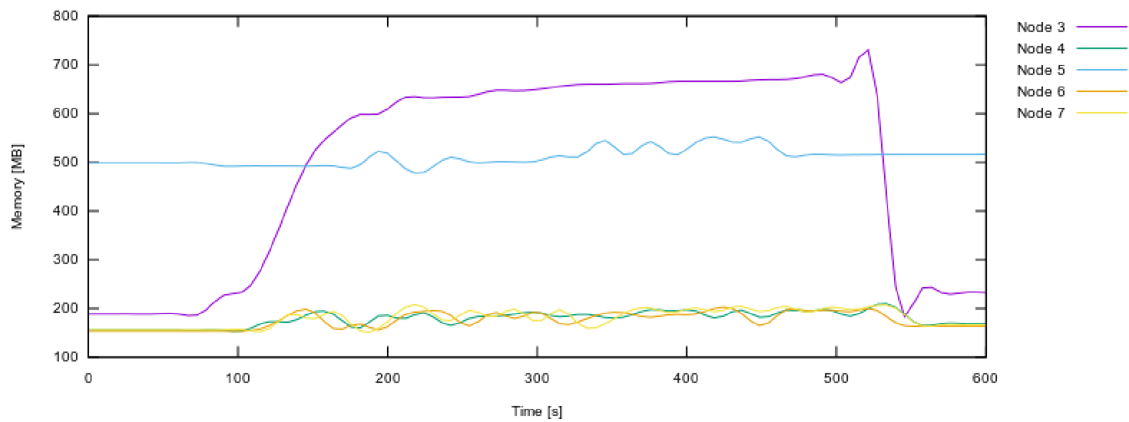

Figure 8: Container memory use, by node.

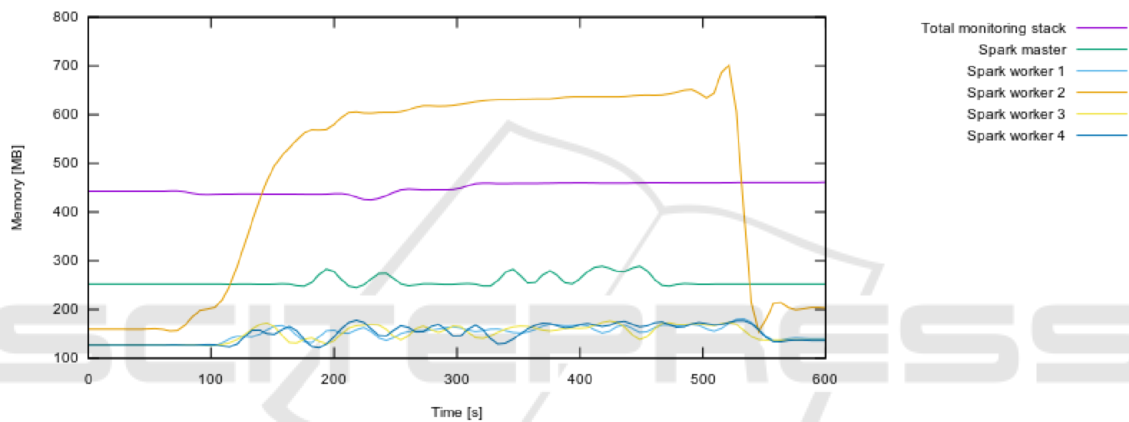

Figure 9: Container memory use, by container.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Lightweight edge architectures are important for many new application areas. Autonomous driving is an example where mobile edge clouds are required. Vehicles need to coordinate their behaviour, often using modern telecommunications technologies such as 5G mobile networks. This requires onboard computing capability as well as local edge clouds in addition to centralised clouds in order to guarantee the low latency requirements. SBC clusters are an example of computational infrastructure close the outer edge.

A possible future extension of the work should explore different possibilities enabled by Apache Spark, for example using the network as a data source. This approach could facilitate the use of common IoT data transmission protocols, like MQTT, while Hadoop and HDFS might be used on pre-processed and aggregated data. Furthermore, the model used to evaluate the implementation could be extended in order to verify the actual performance of the system in more scenarios. Here, alternative data stream sources could also be considered as part of future work.

Another direction would go beyond the performance engineering focus, addressing concerns such as security and trust (El Ioini and Pahl, 2018; Pahl et al., 2018a) for IoT and edge cloud settings.

ACKNOWLEDGEMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825012 – Project 5G-CARMEN.

REFERENCES

Apache (2018). Hadoop. Online: https://hadoop.apache.org. Accessed: September 2018.

Apache (2018). Spark. Online: https://spark.apache.org. Accessed: September 2018.

Baldeschwieler, E. (2018). Yahoo! launches worlds largest hadoop production application. Online:

http://yahoohadoop.tumblr.com/post/98098649696/ yahoo-launches-worlds-largest-hadoop-production. Accessed: September 2018.

Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In OSDI04: Sixth Symposium on Operating System Design and Implementation,

Docker (2018). Online: https://docs.docker.com/. Accessed: September 2018.

El Ioini, N. and Pahl, C. (2018). A review of distributed ledger technologies. OTM Confederated International Conferences.

Femminella, M., Pergolesi, M., and Reali, G. (2016). Performance evaluation of edge cloud computing system for big data applications. In 5th IEEE International Conference on Cloud Networking (Cloudnet), pages 170-175.

Fowley, F., Pahl, C., Jamshidi, P., Fang, D., and Liu, X. (2018). A classification and comparison framework for cloud service brokerage architectures IEEE Transactions on Cloud Computing 6 (2), 358-371.

Heinrich, R., van Hoorn, A., Knoche, H., Li, F., Lwakatare, L.E., Pahl, C., Schulte, S., and Wettinger, J. (2017). Performance engineering for microservices: research challenges and directions. Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion.

Jamshidi, P., Sharifloo, A., Pahl, C., Arabnejad, A., Metzger, A., and Estrada, G. (2016). Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures. 12th International ACM Conference on Quality of Software Architectures (QoSA).

Jamshidi, P., Sharifloo, A., Pahl, C., Metzger, A., and Estrada, G. (2015). Self-learning cloud controllers: Fuzzy q-learning for knowledge evolution. arXiv preprint arXiv:1507.00567.

Jamshidi, P., Pahl, C., Mendonca, N.C., Lewis, J., and Tilkov, S. (2018). Microservices: The Journey So Far and Challenges Ahead. IEEE Software 35 (3), 24-35.

Jamshidi, P., Pahl, C., and Mendonca, N.C. (2016). Managing uncertainty in autonomic cloud elasticity controllers. IEEE Cloud Computing, 50-60.

Johnston, S.J., Basford, P.J., Perkins, C.S., Herry, H., Tso, F.P., Pezaros, D., Mullins, R.D., Yoneki, E., Cox, S.J., and Singer, J. (2018). Commodity single board computer clusters and their applications. Future Generation Computer Systems, 89: 201-212.

Hentschel, K., Jacob, D., Singer, J., and Chalmers, M. (2016). Supersensors: Raspberry pi devices for smart campus infrastructure. IEEE Intl Conf on Future Internet of Things and Cloud (FiCloud).

Morabito, R. (2016). A performance evaluation of container technologies on internet of things devices. In IEEE Conference on Computer Communications Workshops.

Morabito, R., Farris, I., Iera, A., and Taleb, T. (2017). Evaluating performance of containerized iot services for clustered devices at the network edge. IEEE Internet of Things Journal, 4(4):1019-1030.

Naik, N. (2017). Docker container-based big data processing system in multiple clouds for everyone. In 2017 IEEE International Systems Engineering Symposium (ISSE), pages 1-7.

Pahl, C. and Lee. B. (2015). Containers and clusters for edge cloud architectures - A technology review. IEEE Intl Conf on Future Internet of Things and Cloud.

Pahl, C., El Ioini, N., Helmer, S., and Lee, B. (2018). An architecture pattern for trusted orchestration in IoT edge clouds. Third International Conference on Fog and Mobile Edge Computing (FMEC).

Pahl, C., Jamshidi, P., and Zimmermann, O. (2018). Architectural principles for cloud software. ACM Transactions on Internet Technology (TOIT) 18 (2), 17.

Pahl, C., Helmer, S., Miori, L., Sanin, J., and Lee. B. (2016). A container-based edge cloud paas architecture based on raspberry pi clusters. IEEE International Conference on Future Internet of Things and Cloud Workshops (FiCloudW).

Renner, T., Meldau, M., and Kliem, A. (2016). Towards container-based resource management for the internet of things. In International Conference on Software Networking (ICSN).

Renner, M. (2016). Testing high availability of docker swarm on a raspberry pi cluster. Online: https://blog.hypriot.com/post/high-availability-with-docker/, 2016. Accessed: September 2018.

Renner, M. (2016). Evaluation of high availability performance of kubernetes and docker swarm on a raspberry pi cluster. Highload++ Conference.

Raspberry Pi Foundation (2018). Online: https://www.raspberrypi.org/products/raspberry-pi-2-model-b/. Accessed: September 2018.

Taibi, D., Lenarduzzi,V., and Pahl, C. (2018). Architectural patterns for microservices: a systematic mapping study. Proc. 8th Intl Conf. Cloud Computing and Services Science.

Tso, F.P., White, D.R., Jouet, S., Singer, J., and Pezaros, D.P. (2013). The glasgow raspberry pi cloud: A scale model for cloud computing infrastructures. In IEEE 33rd International Conference on Distributed Computing Systems Workshops.

Turner, V. (2014). The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things. IDC Report.

von Leon, D., Miori, L., Sanin, J., El Ioini, N., Helmer, S., and Pahl, C. (2018). A Performance Exploration of Architectural Options for a Middleware for Decentralised Lightweight Edge Cloud Architectures. Intl Conf on Internet of Things, Big Data and Security.

von Leon, D., Miori, L., Sanin, J., El Ioini, N., Helmer, S., and Pahl, C. (2019). A Lightweight Container Middleware for Edge Cloud Architectures. Fog and Edge Computing: Principles and Paradigms, 145-170. Wiley & Sons.

Wang, Y., Goldstone, R., Yu, W., and Wang, T. (2014). Characterization and optimization of memory-resident mapreduce on hpc systems. In IEEE 28th Intl Parallel and Distributed Processing Symposium.