2017 29th International Teletraffic Congress

Monitoring Container Services at the Network Edge

Marcel Großmann and Clemens Klug
Computer Networks Group, University of Bamberg
Bamberg, Germany
Email: {marcel.grossmann | clemens.klug}@uni-bamberg.de

***Abstract*—Recent developments induced by the Internet of Things (IoT) force a paradigm shift to deploy on demand services to a broad range of different computing architectures. Mainly single board computers (SBCs) gained a lot of attraction in recent years, shifting highly available processing power to service consumers and IoT devices. Simultaneously, container virtualization achieved a breakthrough by the well known Docker environment that became a key competitor to the utilization of virtual machines. Nevertheless, spreading containers at small scale needs a proper allocation of resources available on SBCs.**

As a first step, we developed a multi-architecture framework "PyMon" to monitor different computing architectures with a small footprint. PyMon itself is based on the recent version of "monit" and a "Django" application to collect monitoring data. It is delivered by several Docker images and allows monitoring with a reasonable processing overhead. Our demonstration shows the statistics of either processes or containers running on a cluster of SBCs, where each member is monitored. Furthermore, we evaluated the resource usage of the two key competitors for container cluster management on SBCs, Kubernetes and Docker Swarm, with PyMon.

***Keywords*—Container Virtualization; Cluster Computing; Distributed Computing; Mesh Networking; Privacy; Reliability; Redundancy; QoS;**

I. Introduction

The foundation of everything as a service (XaaS) is mainly based on the uprising of virtualization technology utilized in cloud data centers. Nowadays, virtual machines are the workhorses to effectively provide services and modern communication infrastructures running on top of physical servers, which are aggregated to powerful computing clusters. Within the software virtualization scalability, redundancy, availability, and reliability issues are satisfied.

However, rapid developments in the Internet of Things (IoT), which include larger structures, e.g., smart homes, smart cars, and smart cities, are challenging this traditional approach of service deployment [cf. 1]. By using traditional distributed data center approaches for IoT purposes, transmission of mainly unnecessary data burdens the core network. Fog and Edge Computing proposed by Yi *et al.* [2] is one proposal to shift virtualized service of cloud data centers towards the network edge. Due to recent advances in designing energy-efficient, cheap, and low cost single board computers (SBC), positioning computing power near to the consumers of services is easy to obtain. By empowering the edge network in this manner, the core network drops its bottleneck state.

But, what about virtualization? Virtualization is undergoing a paradigm shift due to the success of container virtualization approaches, particularly with regard to the rise of Docker's community and its corresponding open source framework. Besides, the Docker Inc. established a viable ecosystem around the container technology to make it easily accessible for everyone. In addition, due to significant efforts of the Hypriot Team[1], the Docker engine was ported to the ARM platform and is officially supported by Docker since release 1.12.1[2]. It allows to run processes efficiently at highest speed on SBCs without the overhead of a hypervisor using lightweight kernel namespaces derived from Linux. This development strengthens the feasibility to realize clustered edge computing infrastructures, with nearly the same advantages as data center provided services offer. Therefore, it is no wonder that orchestration tools like Docker Swarm and Kubernetes are already realized on ARM based SBCs. Until now, a skillful placement of containerized services on small devices is challenging due to the lack of monitoring approaches for containers on SBCs. Of course, the ecosystem of container monitoring solutions is rapidly changing[3], but by executing tools like Google's cAdvisor[4] attached to Prometheus[5] on small devices utilizes too much resources to be a feasible SBC monitoring solution.

In this paper, we show a tinier monitoring solution, especially designed for SBCs, called *PyMon*, which is made publicly available on Github[6]. PyMon extends the host-based monitoring tool *monit*[7] with capabilities to inspect running Docker containers. It is delivered by Docker images and they run on architectures supporting Docker, namely x86_64, ARM and AARCH64.

II. Structure of Monitoring Framework PyMon

The framework PyMon is structured in a modular manner and is packaged into several Docker images. Figure 1 shows the linkage of all modules, which may be defined in a *docker-compose* file. In a brief summary, PyMon can be separated in two main parts: The server application, consisting of *Caddy*[8] as a load-balancing proxy server, *PyMon* to collect and analyze data, and a *Postgres* database to save all captures; and a *monit* instance for each monitored host.

[1] https://blog.hypriot.com
[2] https://github.com/docker/docker/releases/tag/v1.12.1
[3] http://rancher.com/comparing-monitoring-options-for-docker-deployments/
[4] https://github.com/google/cadvisor
[5] https://prometheus.io/
[6] https://github.com/whatever4711/PyMon
[7] https://mmonit.com/monit/
[8] https://caddyserver.com/

978-0-9883-0453-6/17 $31.00 © 2017 ITC
DOI 10.1109/ITC.2017.47

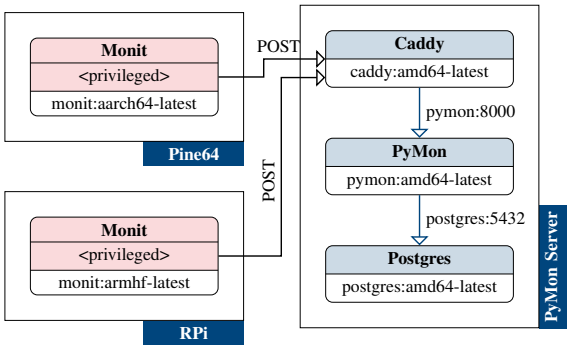


Figure 1. Architecture of the PyMon framework

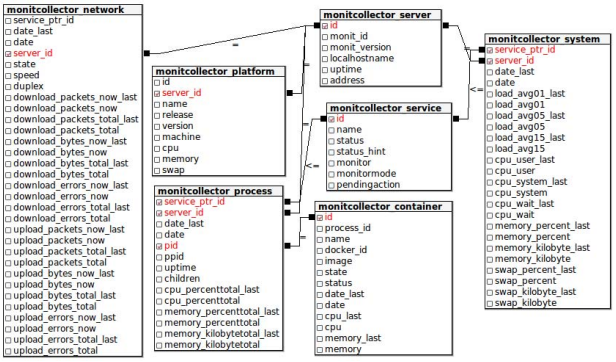


Figure 2. Structure of database

A virtual network is established to interconnect the server application modules, in a way that Caddy, PyMon, and Postgres can communicate. Additionally, Caddy is attached to an externally reachable network, such that it accepts all HTTP traffic and forwards all incoming traffic to PyMon.

A. Monit

We extended monit with an API to inspect running Docker containers and monitor their resource utilization. Therefore, it should run as a privileged container to be able to access the Docker socket. An instance of this container monitors all containers on the respective host system, including the monit container itself.

Listing 1 Code to capture container statistics

```
def update_stats(self, data):
 old=containers[self.id]["stats"]
 new={
  "cpu": container_cpu_percent(data),
  "memory": data["memory_stats"]["usage"]
 }
 for m in self.__metrics:
  self.history[m].append(new[m])
  new[m]=statistics.mean(self.history[m])
 containers[self.id]["stats"] = new
```

The extension wraps collected data about containers into a JSON element and transmits it to the PyMon server. Due to the streaming API of Docker statistics, a dedicated service is started through monit on each monitored host. The CPU usage reported by Docker is specified as active CPU time of a container. To derive the percentage of CPU consumption we have to intersect container and system CPU times. A monit service uses threads for all running containers to process all statistic streams in parallel. The service includes a basic HTTP server, offering a JSON-encoded dictionary of container metrics with aggregated measurements since the last poll. This is requested regularly via a monit check progress, where the JSON object is passed to `stdout` and therefore included in the report by monit. Listing 2 shows an example of a statistic summary of a recently started Kubernetes dashboard container (gcr.io/google_containers/kubernetes-dashboard-arm:v1.5.1).

Listing 2 Example of container statistics

```
{"59f0b9d4752..": {
"image": "kubernetes-dashboard-arm:v1.5.1",
"status": "Up 28 seconds",
"name": "/k8s_kubernetes-dashboard.314..",
"state": "running",
"stats": {
  "cpu": 0.00310019557823129,
  "memory": 5283840
}}}
```

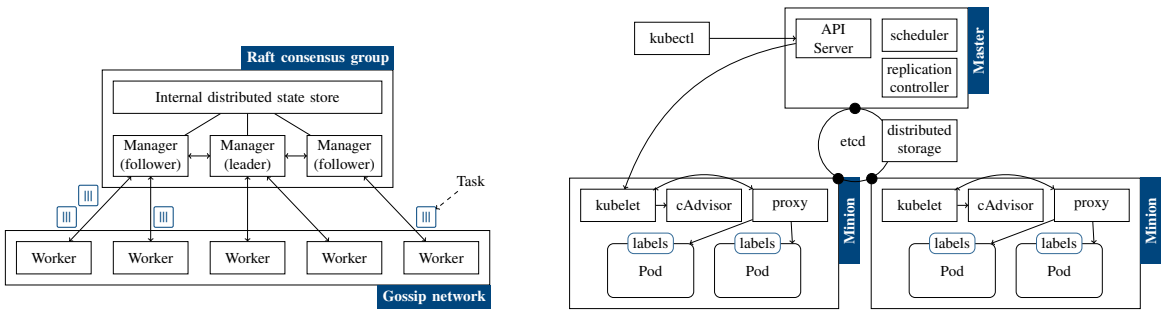
Additionally, various other host-specific metrics are configured to be measured (CPU, memory, network traffic). All measurements are collected in a monit XML report and are pushed to the Caddy HTTP server.

B. PyMon

The core is PyMon, a Django application, which runs inside a container and receives all data from monitored hosts (see subsection II-A). All collected data is stored in the Postgres DB for seven days worth of regular updates to allow retrospective analysis while the accumulation of data is limited. These stored measurements include the total CPU and memory consumption of the server, as well as the network traffic of its interfaces and the metrics of running containers. The web interface of the Django application offers graphs and tabular overviews of the monitored systems' states, while additional plotting options are available through the management interface.

C. Postgres DB

The database structure is derived from the Django ORM mapping of monit XML reports. Figure 2 shows the resulting structure, where our Docker statistics aggregator is a service with a running process. Each monitored container refers to this process and thus allows identification of the server it is running on.



(a) Docker Swarm (b) Kubernetes

Figure 3. Simplified versions of architectures for cluster management.

III. Measurements for a Fog Computing Cluster

Among a few data center orchestration engines, for instance OpenStack, Pacemaker, Marathon, etc., we selected ARM compatible ones as trial environment. By evaluating four possible candidates, we figured out that Mesos and Nomad are either to complex or with limited support for ARM architectures. Thus, we chose Kubernetes, which is unofficially supported for ARM, and Docker Swarm mode. This being said, Docker and Kubernetes became two potential competitors as tools for high availability on small platforms. It is interesting to evaluate their performance against each other, and we compare these two technologies according to their basic resource consumption.

A. Docker Swarm on ARM

Within the release of Docker 1.12 all clustering features are implemented in the Docker engine. The "Swarm Mode"[9] activates its cluster capabilities and features for cluster management are complete for orchestration purposes, even if the feature set is not as large as the one of Kubernetes. The Docker engine carries all mandatory features for master nodes and worker nodes as depicted in Figure 3a. The manager nodes, starting in the follower state, use the Raft Consensus Algorithm to handle the global cluster state[10] and elect one leader node for the cluster. In combination with the *internal distributed state store*, the cluster state, health checks, and eventually fault tolerance capabilities are provided. On manager nodes *services* are created, which are the blueprints of tasks to be executed on worker nodes; normally, manager nodes possess worker capabilities too. Services can be scaled, thus they provide several replicated tasks across the cluster. Moreover, *stacks*, a combination of several services, can be defined and deployed against a master node.

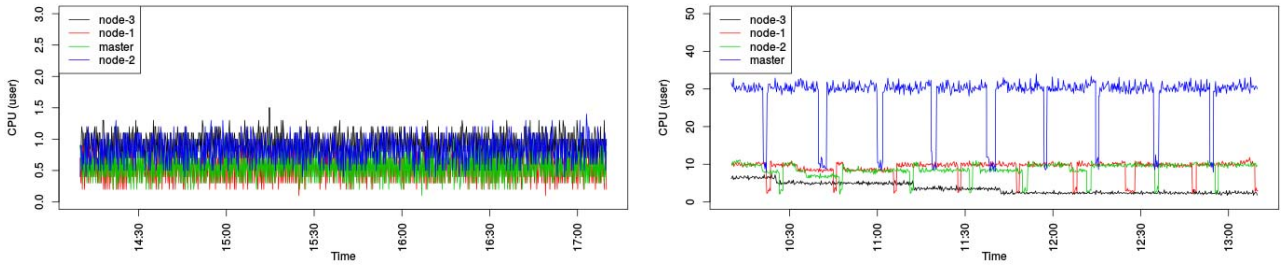
B. Kubernetes on ARM

Kubernetes has been open sourced in 2014 and it provides the experience that Google gathered with their internal orchestration platform "Borg". Thus, Kubernetes is seen as very reliable and well tested. In contrast to Docker, it provides much more abstractions that enable provisioning highly tailored deployments. Some of the features that Docker Swarm Mode does not provide, are: centralized logging, identity and authorization or resource monitoring and more[11]. Kubernete's main concepts have much in common with Docker Swarm Mode: *Pods* are the smallest deployable units, which are created, scheduled, and managed. Internally, they consist of a logical collection of containers that belong to an application. Worker nodes are also called *Minions*, which are managed by the Master node. "Minions can run one or more pods. It provides an application-specific "virtual host" in a containerized environment" [3]. The command line tool `kubecfg` and its related tool set is used to control overall settings of the cluster, such as replication controllers and the scheduler. Kubernetes uses *etcd* as key-value-store and offers a central API that takes management commands from the operator, as depicted in Figure 3b. Obviously, this implementation is very similar to Docker Swarm. Nevertheless, one difference exists in the leader election, where Kubernetes uses its own implementation.

C. PyMon Measurements

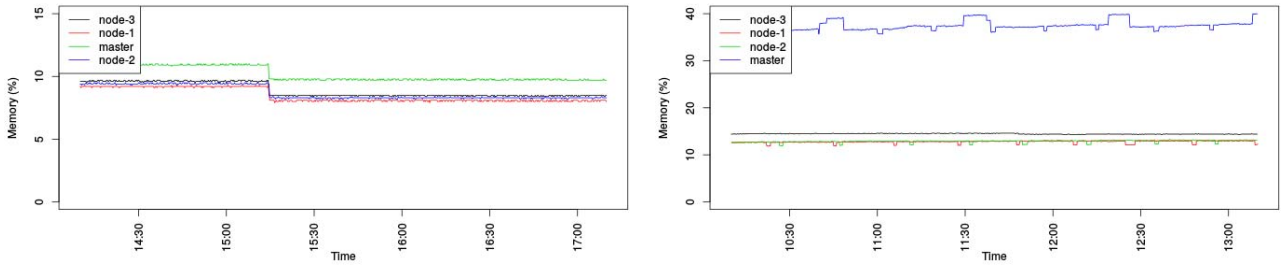
In our trial we use Docker in version 17.03.0-ce released in March 2017 for the measurements of Docker Swarm and Kubernetes. The hardware basis was provided by PicoCluster Ltd., which is a hardware stack of three RPis. Additionally, we provided a fourth RPi that acts as master node in our trials. We prepared four $16GB$ SD cards from SanDisk, with read speed up to $80^{MB}/_{s}$ and write speed up to $20^{MB}/_{s}$, with Hypriot OS and plugged them into all RPis. All devices are connected with an Asus RT-AC68U, which provides a $1^{Gbit}/_{s}$ switching fabric. Like depicted in Figure 1, we let *Monit* run natively on all RPis to be able to switch between Kubernetes and Docker Swarm without disrupting the data collection process. *PyMon* is installed on a *Pine64+*[12] AARCH64 SBC, which is connected to the network via its WiFi interface. The trial network is self-contained, such that no other entities, which may send broadcast messages into the network and disturb our measurements, have any influence.

[9]https://docs.docker.com/engine/swarm/
[10]https://docs.docker.com/engine/swarm/raft/
[11]http://kubernetes.io/docs/whatisk8s/
[12]https://www.pine64.org/



(a) Docker Swarm (b) Kubernetes

Figure 4. Comparison of CPU utilization



(a) Docker Swarm (b) Kubernetes

Figure 5. Comparison of memory utilization

D. Results

By comparing the results of the total CPU utilization of Figure 4 we see that Kubernetes uses around 30 percent CPU at the master node in its idle state and around 10 percent at the worker nodes. In both cases, Docker swarm requires less CPU utilization and should be the preferred solution for fog or edge clusters. Similar are the results for overall memory usage. As depicted in Figure 5, Kubernetes needs more memory compared to Docker running in Swarm Mode.

However, this comparison is not quite fair. As already mentioned, Kubernetes' capabilities are much more sophisticated and customizable than Docker Swarm Mode's. In addition, Kubernetes is much more feature rich than Docker and comes with essential services by default. Disabling them the hard way is probably risky for the stability of the cluster.

IV. Conclusion & Future Work

We provided a prototypical implementation of our lightweight monitoring framework PyMon, which is made available for all relevant Docker enabled IoT infrastructures - ARM, AARCH64 and x86_64 - on Dockerhub[13]. The system is easy to setup and collects container statistics. In the future, we would like to extend PyMon with an API that provides resource consumption and statistics of a container to be able to place new instantiations according to their needs. Furthermore, we like to integrate an alerting system to classify resource usage.

Acknowledgment

The authors would like to thank Nicor Lengert for his previous work on PyMon's Django instance, the Hypriot team for the portation of Docker to ARM platforms, and L. Kaldström for the project "Kubernetes on ARM"[14].

References

[1] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: a survey on enabling technologies, protocols, and applications", *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2347–2376, Jun. 2015, ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2444095.

[2] S. Yi, C. Li, and Q. Li, "A survey of fog computing: concepts, applications and issues", in *Proceedings of the 2015 Workshop on Mobile Big Data*, ser. Mobidata '15, Hangzhou, China: ACM, 2015, pp. 37–42, ISBN: 978-1-4503-3524-9. DOI: 10.1145/2757384.2757397.

[3] A. Gupta, *Miles to go 3.0*, (accessed 2016-Okt-14), 2015. [Online]. Available: http://blog.arungupta.me/key-concepts-kubernetes/.

[13] https://hub.docker.com/r/whatever4711/pymon/

[14] https://luxaslabs.com