Edge Resource Utilization Using OS-Level Virtualization on Smart Devices*

Thomas Renner**, Marius Meldau
and Andreas Kliem

*Technische Universität Berlin (TU Berlin),
Complex and Distributed IT Systems (CIT),
10587 Berlin, Germany*

***Corresponding Author: thomas.renner@tu-berlin.de*

Received 19 August 2016; Accepted 17 October 2016;
Publication 2 November 2016

Abstract

Following the Internet of Things (IoT) paradigm, more and more connected and embedded devices like sensors, smart phones, or entertainment devices are becoming available surrounding us. This increasing number of IoT devices leads to an increasing amount of resources made available to the users. For instance, if devices like smart phones or smart TVs are considered, this includes computing and storage resources. In this paper we propose a container-based resource allocation scheme to increase the utilization of these IoT resources and made them shareable between different users due to cloud-centric IoT deployments. The approach allows various applications and users to dynamically allocate resources offered by edge devices and process IoT data close to the source. The approach is evaluated regarding its feasibility in terms of performance on a exemplary resource constrained IoT device.

Keywords: Internet of Things, Edge Computing, OS-level Virtualization, Resource Management, Resource Sharing.

*An earlier version of this paper appeared in the Proceedings of International Conference on Software Networking (ICSN), IEEE, 2016.

1 Introduction

The Internet of Things (IoT) paradigm [1] is based on the assumption that uniquely identifiable things can be integrated into information networks. Therefore, a virtual representation of the physical things needs to be created. This task can be accomplished by circuits embedded or attached to the things or by IoT Integration Platforms, such as a middleware running on a gateway device. In general, a virtual representation can be described as an interface that exposes the capabilities of the thing using a set of operations (e.g. getters or setters allowing to access certain properties). Once virtual representations are available, they can be integrated into information networks, such as the Internet, by means of communication. Thus, enabling developers to integrate the things into business, information and social processes (i.e. workflows and applications). As a result, the IoT paradigm can be envisioned as an enabler for a variety of applications found in consumer and industry related application domains. Popular examples are Smart Homes, Smart Cities, (Factory) Automation, E-Health, or Logistics [19, 32].

In order to allow application developers to properly take advantage of all the interconnected things, providing the IoT Integration Platform is the key challenge for most of the IoT deployments. Due to its various application domains, the IoT subject to a huge heterogeneity in terms of things, their virtual representations and the components required to create these representations (e.g. device drivers, data formats, configurations). These often follow different or even no standards [26]. As a result, IoT Integration Platforms need to hide the process of establishing physical integration and spontaneous interoperability from the application developers. This challenge is intensified by the rapid growth of things deployed to our environment, which is motivated by the proliferation of low cost, small size and more powerful devices [28] and by the increasing attention IoT based solutions gain.

Many solutions for IoT Integration Platforms currently available and discussed in literature follow a Cloud-centric approach. A Cloud-centric approach, sometimes referred to as Sensor Virtualization, basically consists of an edge node acting as a gateway (e.g. a router or a smart phone) for the things and a set of Cloud nodes providing the resources to perform data analytics and host applications. Thereby, the gateway node physically integrates the things and dispatches the resulting data streams to the Cloud nodes (i.e. takes care of Machine-to-Machine (M2M) [20] related aspects of the solution). Application logic and major parts required to achieve interoperability (e.g. data transformation, unification of interfaces) are entirely shifted to the Cloud. As a result, Cloud-centric solutions separate data acquisition and processing from each

other. This causes the underlying network infrastructure to meet high requirements regarding non-functional properties. Low latency, high bandwidth, and efficient security and privacy measures are required to meet the Quality of Service expectations of IoT applications, especially in terms of applications related to the Industrial IoT domain [18]. Furthermore, the network infrastructure needs to provide sufficient resources in order to avoid congestion in the presence of the continuously growing amount of connected things.

Various initiatives and approaches like Fog Computing [5] or Mobile Edge Computing [25] have recognized the demand for more flexible IoT Integration Platforms and propose utilizing resources at the edge of the network. The objective is to reduce the latency and the amount of data that need to be transmitted between data sources and sinks (e.g. applications or consumers). The gap between data acquisition and processing resulting from Cloud-centric approaches can be bridged by moving application logic (e.g. (pre-) processing) towards the gateway nodes of the IoT Integration Platform or the things themselves. Several applications are proposed to benefit from the reduced latency (e.g. traffic control systems). A further motivation of this approach is given considering the growing amount of so called Smart Devices. Examples for Smart Devices include smart phones, smart TVs, game consoles, or routers, which are already used as gateways in IoT integration platforms. Basically, a Smart Device can be distinguished from a usual sensor or thing by its capability to execute custom software components. In addition to its role as a data source in terms of the IoT, a Smart Device offers computing and storage capabilities. Hence, internal resources like CPU or storage can be made available to external components such as Apps or software components for data (pre-) processing. By facilitating these resources, the IoT Integration Platform can be enriched with data processing capabilities available at the edge of the network, which means close to the data sources (and possibly consumers as well). However, most of these Smart Devices are currently used for stand-alone purposes and need to be enriched with capabilities to share their processing power for instance. A suitable example is given by a smart TV. Current smart TVs are based on a fully featured computing platform to offer services like video streaming, internet access or other entertainment services. However, often these TVs are only used for a few hours per week and remain idle the meantime.

With this motivation, we propose to apply the popular Cloud Computing paradigm [21] to the IoT domain. We consider things as resources that may offer their computing or storage capabilities. These can be allocated on demand independent from the owning user. This introduces the notion of sharing IoT devices and the capabilities they offer. As a result, the utilization of

compute and storage capabilities can be increased and the communication demand and latency can be reduced, because edge devices are used for data (pre-) processing and aggregation. In addition, a resource oriented view on the available IoT infrastructure enables new ways of mobility, scalability, and upcoming paradigms like community based interaction and sharing of IoT resources [12, 14].

This paper presents an approach towards a container-based resource management system for connected, embedded IoT devices. The approach is based on the Device Cloud [29], an IoT architecture that focuses the application of Cloud Computing paradigms to the IoT domain by enabling users to share and allocate physical IoT devices using a resource-pool abstraction. The resource management capabilities of the presented approach are based on Docker [22]. We propose a container-based virtualization approach leveraging the advantages of the underlying Operating-system-level (OS-level) virtualization. OS-level virtualization provides sufficient resource isolation capabilities and provides less overhead compared to a virtual-machine based approach [31]. Appropriate isolation is a mandatory requirement, since the proposed approach assumes that resources of a single Smart Device can be allocated and shared by multiple users. In particular, we introduce an architecture that takes advantage of Docker's automated deployment model and allows sharing resources within containers between different users and devices. The presented approach focuses the feasibility of OS-level virtualization for the isolation of user code and the dynamic allocation of resources offered by Smart Devices.

The remainder of the paper is structured as follows. Section 2 discusses approaches for the isolation of user code by means of virtualization on resource constrained devices. Section 3 explains our approach and system architecture. Section 4 presents first results, including our prototype and an evaluation of it. Section 5 discusses further related work Section 6 concludes this paper.

2 Background

Virtualization can be considered as one of the core enablers leading to the success of the Cloud Computing paradigm. It allows providers of computing infrastructure to provision currently idle resources and thus, can significantly benefit the utilization of resources. Its main advantage is the high level of isolation achieved between tasks belonging to different users. However, with respect to mobile devices, which usually suffer from resource constraints and regard battery life-time as one of their most expensive resources [4], the efficiency of the virtualization technique employed becomes of crucial importance.

The most comprehensive level of isolation is achieved by full (hardware) virtualization. A hypervisor provides a full set of virtual devices to guest operating systems. As a result, guest operating systems usually do not need to be aware of the virtualization and can be operated in the virtual environment without modification. However, because full virtualization requires to pass all hardware accesses through virtualized drivers, these solutions are constrained by reduced I/O performance and considerable virtualization overhead. Since resource constraints play an important role for embedded devices, paravirtualization has been considered as a virtualization approach for such environments. Paravirtualization replaces the virtual I/O-stack with a more efficient and lightweight API.

The most popular paravirtualization solution is the open source software Xen [2]. The concept is to run a privileged virtualization domain called *dom0* that has access to the underlying hardware and multiple limited domains called *domU* that access the hardware via *dom0*. While the *dom0* can rely on native device drivers to integrate the underlying hardware, the *domU* domains require Xen-specific paravirtualization drivers to access the hardware through the *domU*. Xen is lightweight enough to run on a smart device [8]. Based on the Xen Project, embeddedXEN [30] is a project specifically aimed at running the Xen hypervisor on smart devices. It uses a multi-kernel binary image which can be flashed for example on a smartphone. This image makes deployment relatively easy. However, the specific architecture employed by the embeddedXen project limits Xen's capabilities to only a single *domU* besides the obligatory *dom0*. In general, the installation and deployment of Xen on smart devices introduces some difficulties. The main challenge is to provide the required paravirtualization drivers. While sensors may be integrated using a trusted gateway *domU*, each *domU* would still need to be able to access the hardware resources offered by the host devices (e.g. storage). Since the amount of different hardware platforms available for smart devices is considerably greater than for regular server or PC platforms, providing specialized paravirtualization drivers for each possible platform may not be feasible.

Another virtualization solution available for smart devices is the VMware Mobile Virtualization Platform (MVP) [3], which is a type-2 paravirtualizing hypervisor hosted on Android and an ARM processor. The major objective of MVP is to enhance corporate IT-infrastructures with Bring Your Own Device (BYOD) capabilities. BYOD allows employees to use their private devices and integrate them securely into a corporate IT environment following certain policies and conditions (e.g. isolation of user and company data). The MVP

does not place any restrictions on the host system, but creates virtual guest environments that can be secured by the IT department. However, MVP still suffers from the requirement to provide paravirtualization drivers.

2.1 OS-Level Virtualization

Operating System (OS) Containers or Operating-System-Level Virtualization is a completely different isolation concept. It does not provide virtual hardware environments like full and paravirtualization. Rather, it adds an additional layer of isolation within the Kernel of the host OS. This yields multiple virtualized userland environments, each of which has a view on the system as if it was running alone on the machine [16].

Recently, OS-Level virtualization is gaining attention as the concept behind the scenes of the Docker platform [7]. Although recently replaced by another API, Docker was initially developed on top of the Linux Containers (LXC) library. Thus, Docker allows isolating applications by means of Container-based OS-level virtualization. A Container provides all necessary dependencies an application requires. Furthermore, the Linux Kernel features control groups (cgroups) and namespaces allow controlling the resource consumption of a particular Container and isolating the Containers from each

Figure 1 Multiple containers isolated by an OS-Level based virtualization stack.

other. In comparison to full and paravirtualized approaches, OS-level virtualization (i.e. Container virtualization) features are provided by the Kernel without the need to operate a distinct hypervisor. Therefore, it is considered as more lightweight and performant. Although providing less resource isolation capabilities than full hardware or paravirtualization, we consider the level of isolation as sufficient for the IoT resource management scenario.

Docker defines a scripted build process for container images similar to the classical build process of software. It also uses an online registry where those images can be published. Other images can be build based on those images, and finally these images can be started as actual containers on a Docker host. By using differential-based / copy-on-write file systems, a fast distribution of updates can be achieved. Thus, additionally to its virtualization capabilities, Docker can be considered as a version control system for binary images. Therefore, Docker features an efficient distribution scheme appropriate to implement and deploy *Task Execution Containers* among the smart device platforms that may be used in the IoT resource management scenario.

The efficiency of Docker's facilities for storing and distributing images of Containers is based on a flexible and exchangeable storage backend. By default Docker uses the overlay file system AUFS. AUFS combines several directories into a virtual file system with multiple layers, called branches. The lower layers are read only, but files from these layers can logically be changed by (re)writing a file on the topmost branch [23]. Also, there are backends available for devicemapper which uses virtual block devices [17], and the copy-on-write file system btrfs [27]. This means that even large amounts of *Task Execution Containers* can be maintained with little overhead compared to a single one.

Besides deploying images, Docker provides a registry, which basically acts as a pool of repositories containing Docker images. It allows developers to publish new and updated images and therefore offers an already exiting approach for managing the set of *Task Execution Containers* required to set up the resource management scenario on different smart device platforms.

3 Resource Sharing Architecture

This section describes the design of the Container-based approach we propose to manage and share resources of IoT devices between different users and applications. As outlined in Section 1, scalability is an important requirement to meet the growing demands of the users and handle the growing amount of deployed IoT devices and their related resources. Therefore, the IoT Resource

Management Framework presented in this section is designed relying on a federated approach, introducing several collaborating domains. After describing the overall concept of the domain network, the composition of a single domain network in terms of its nodes is discussed. Finally, life cycle operations required to deploy, initialize and release nodes and *Task Execution Containers* are introduced.

3.1 Device Cloud Domain Network

The Device Cloud follows a federated approach to tackle the scalability challenge. It consists of a set of Domains, whereas each Domain may entirely operate on its own without being required to collaborate with any other Domain. This is based on the assumption that resource sharing interactions usually stay within a certain group of users and therefore follow the locality principle. Therefore, each Domain provides a set of management capabilities, that enable nodes to interact with each other (e.g. directory services or the *Task Execution Containers* repository). These management capabilities are hosted by an independent Domain Operator not participating in the resource sharing processes (similar to the DNS system). However, resource sharing is not restricted to the set of devices contained in one Domain. As described by the general concept of sharing devices for the purpose of increasing resource utilization [12], interactions crossing Domain boundaries are possible. Therefore, the Domain Operators need to negotiate access to so called remote resources on behalf of the users be part of their Domain. As shown in Figure 2, a Domain basically consists of different types of nodes types, whereas no assumptions on the topology are made:

- **Aggregation Nodes (ANs):** ANs represent the smart devices that build the foundation of the proposed resource sharing approach. We use the term Aggregation Node because these nodes are able to conduct generic computational tasks like, for instance, aggregation or pre-processing of data. In general, ANs are devices that provide sufficient compute- and storage resources to run multiple containers without compromising the device main functionality, such as the ability to receive phone calls. Typically, a Domain consists of many interconnected ANs.
- **Backend Nodes (BNs):** BNs represent nodes that are used to host the management capabilities of a Domain. Thus, BNs are usually operated by the Domain Operator and are likely to be deployed on dedicated servers in a data center. BNs provide directory, monitoring and management

services to ANs. For instance, BNs are responsible for scheduling containers and distribute Docker images among ANs.

- **Device Nodes (DNs):** DN are sensors or things that do not fulfill the AN characteristics and thus only act as data sources (e.g. sensors). DN are usually integrated into the Domain information network by ANs. However, since DN do not apply for the Docker based resource management, these nodes are not shown in Figure 2 and their integration is not further discussed.

3.2 Aggregation Node Architecture

According to the node classification given in the previous section, ANs constitute the target for the OS-level based virtualization that is required to allow sharing resources of an AN among multiple users. This section will describe the architecture of a single AN. We assume that the platform of the

Figure 2 Federated device cloud domains consisting of interconnected Aggregation Nodes (AN) and Backend Nodes (BN).

AN (e.g. Android) is Linux based and supports all Kernel features required to run Docker.

Figure 3 illustrates the internal architecture of an AN. Besides the Linux Kernel and the corresponding Operating System hosted on the AN, following components and containers are defined for the purpose of making the (computational) resources of the node available:

- **Bootstrap Code:** The bootstrap code is necessary to set up the Docker based resource sharing environment on an AN. It retrieves the most recent container images from its corresponding BN and allocates containers inside the Docker Daemon. The Docker Daemon can be also referred to as the Aggregation Node Runtime, which manages our *Task Execution Containers*.
- **Management ANR:** The Management ANR is a singleton container that provides the management facilities required within the Aggregation Node Runtime. This includes life-cycle operations required to start, stop, or update containers. Furthermore, the Management ANR registers the AN with the Domain Network by contacting the BN and starting a neighbor AN discovery. As a result, the resources of the AN can be

Figure 3 Overview of the aggregation node architecture.

considered for all future resource allocation attempts. In order to keep track of the available resources, the Management ANR monitors the resource consumption of its running containers and reports the results to its corresponding BN for the purpose of container scheduling decisions.

- **Device Driver Engine (DDE) ANR:** Similar to the Management ANR, the DDE ANR is a singleton Container providing fundamental system services to the Aggregation Node Runtime. It is responsible for the physical integration and management of IoT devices and sensors and makes them available to other Containers. This includes local sensors embedded into the AN (e.g. a GPS sensor integrated into a smart phone) as well as devices connected through the communication interfaces of the AN (e.g. via Bluetooth). In general, the DDE is a framework for on-demand composition of device control logic and allows loading required integration knowledge (e.g. device drivers) from BNs at runtime. The DDE has been described in [13]. Given the DDE, application containers do not need to take care of the physical integration process of sensors and other devices. The DDE exposes them using a Device Category based interface format as described in [24]. Therefore, the BN is able to match the set of Device Categories available on an AN to the requirements of application containers and consider the results during its scheduling decisions.
- **Service ANR:** The Service ANR represents one instance of a *Task Execution Container*. Service ANRs are Docker containers that execute external application logic (i.e. user code) on the AN. Hence, Service ANRs make the resources of an AN available. By instantiating new Service ANRs, available resources can be allocated on-demand. Based on the type of the user code, different images for Service ANRs can be deployed. as outlined, each Service ANR can access external data sources, such as sensors, through the IPC interfaces offered by the DDE ANR.

Since the resources offered by a single smart device are often limited and may not be sufficient to fulfill the overall requirements of an application, several Service ANRs hosted on different ANs may be allocated for one application. Since these Service ANRs are likely to communicate with each other, a virtual network for each application (i.e. service) is established. As a result, two kinds of virtual networks exist within one Domain Network:

- **Management Network:** Each Domain includes one management network. The management network connects the Management ANRs and is used to exchange state information and control commands.

- **Service Network:** Each Domain includes a number of Service Networks, whereas the quantity depends on the number of applications active within the Domain. Each application is assigned its own virtual network. Using the Service Network, Service ANRs belonging to the same application can exchange application data.

3.3 Resource Sharing Life-Cycle

In order to allocate and release resource properly, certain life-cycle operations are defined for allocation and utilization of AN resources by applications. The operations basically describe the process of ANR container deployment, instantiation and release as well as the respective state transitions. As shown in Figure 4, the node and container initialization process consists of following steps:

Figure 4 Node and container initialization. A new AN joins the device cloud network and an application is started.

1. **Pull Management ANR:** Upon initialization of an AN, the bootstrap code loads the most recent image of the Management ANR from the BN of its corresponding Domain Network.
2. **Start Management ANR:** Afterwards, the bootstrap code starts the Management ANR container, which now takes over management of the Aggregation Node Runtime.
3. **ANR Discovery:** On startup the Management ANR registers itself with the BN and starts a neighbor discovery to find all other ANs. Therefore, it uses the BN as an anchor node and retrieves a list of existing Management ANRs.
4. **Pull Configuration:** After having registered with the Domain Network, the Management ANR queries the network for a current set of policies and configurations. These can be retrieved from the BN or from adjacent ANs, which reduces possible delay due heavy load of the BN.
5. **Start DDE ANR:** Next, the Management ANR starts the DDE ANR in order to discover and integrate available DNs (e.g. sensors).
6. **Service Registration:** Application developers willing to facilitate the resources of a Domain, must upload their Service ANR containers to the BN. The BN takes care of scheduling the application by identifying possible ANs through the Management Network.
7. **Service Deployment:** Management ANRs being elected to host Service ANRs pull the recent container image from the BN and start up the container. The resulting Service ANR will announce itself to its corresponding Service Network.
8. **Shutdown:** Upon shutdown of the AN, the Management ANR asks all Service ANRs and the DDE ANR to stop and release allocated resources as well as terminate possible connections to other nodes. It then stops all running containers. Finally, the Management ANR requests the bootstrap code to terminate itself.

4 Evaluation

This section presents our experimental setup, the test workload, and benchmark results for a container-based Resource Management Platform. Since this paper focuses on the feasibility of the OS-level virtualization based resource allocation and isolation, it particularly targets performance measures of life cycle operations required to deploy and release containers. In addition, the computational overhead of executing applications in containers is measured by using a test application. All experiments were done using on a Raspberry

Pi 2 Model B with a 900MHz quad-core ARM Cortex-A7 CPU and 1GB RAM. The Raspberry Pi runs on Hypriot Docker for Raspberry Pi version 0.2¹, a Debian based operating system including a Docker package for ARM architectures.

4.1 Performance of Life-Cycle Operations

An important performance aspect of the resource management platform arises from life-cycle actions, required to manage the deployment of single ANR containers (e.g. init, start, stop). Naturally, these operations take some time and can have serious impact on the overall performance of the platform. This is because a Domain Network is supposed to exhibit a dynamic nature, which means frequent changes of service allocations and container deployments are subject to single ANs. Therefore, it is important to observe how fast the life-cycle transitions can be conducted on actual hardware. It is worth mentioning that we assume that the device on which the containers are running has already registered with the domain. Thus, the evaluation focuses on the duration of the following life-cycle operations:

- **Initialize:** Initializing an ANR requires to create a data-container and a corresponding runtime container.
- **Start:** After initialization, the runtime container is automatically started by invoking the entry-point defined in the Dockerfile corresponding to the runtime-container (e.g. a command to start user code).
- **Stop:** Stops the runtime container.
- **Update:** Stops a runtime container, loads the most recent image from a private Docker Hub and starts the container again.
- **Reset:** Stops and removes the runtime and data container.

Table 1 shows the time in seconds that a life-cycle action took per container. Three series of tests have been conducted, whereas one 1, 5 and 15 containers have been started in a sequential order. Every experiment was executed

Table 1 Execution times of different life cycle action in seconds for successive starting of 1, 5 and 15 containers

# Containers	Init	Start	Stop	Update	Reset
1	1.76	0.87	0.83	2.00	1.33
5	2.03	1.01	0.93	1.94	1.37
15	2.18	1.02	0.82	2.08	1.37

¹<http://blog.hypriot.com/>, accessed 2016-04-01

seven times. We report the median runtime. The results shows that all life-cycle operations complete within an acceptable amount of time. With an increasing load, the average life cycle execution time per container increases by about 20%.

4.2 Computation Overhead

This subsection evaluates the computational overhead in terms of CPU utilization and the resulting impact on tasks running inside the containers. Therefore, the CPU performance was measured using the Scimark 2.0 benchmark². This benchmark was developed for scientific and numerical computing. It measures several computational kernels and reports a composite score in approximate Mflop. To measure the performance of a task running in Docker, the Scimark jar file is treated as a piece of user code to be deployed inside a Service ANR. Therefore, the benchmark was executed upon startup on the respective container. For every run, the number of simultaneously running Scimark containers has been increased to simulate high system load. Table 2 shows the results of these experiments. As mentioned, the Raspberry Pi 2 used in our evaluation has four CPU cores. Scimark 2.0 is a single threaded application. Therefore, a relative score variable, expressing the score per instance by $\frac{\text{no. of instances}}{\text{no. of cores}}$, was introduced in Table 2. Thus, a value of 1.0 stands for an approximately equal performance result compared to a single benchmark run without using a Docker based container. The results show no relevant difference between native and dockerized benchmark processes, nor do they show a relevant drop in performance due to the scheduling overhead introduced due to many benchmark instances running simultaneously. As a result, the computational overhead can be assessed as low. Even 20 instances with heavy CPU usage slow down the overall performance only by 4%.

Table 2 Results of Scimark 2.0 in Mflop. Relative score is normalized on the number of cores, and relative to the native performance

# Containers	Composite Score	Relative Score
Bare Metal (Baseline)	14.19	–
1 Container	14.22	1.00
2 Containers	14.21	1.00
4 Containers	14.14	1.00
8 Containers	6.98	0.98
20 Containers	2.73	0.96

²<http://math.nist.gov/scimark2/>, accessed 2016-04-01

5 Related Work

This section presents related approaches that target the utilization of computational resources offered by edge devices.

Cloud computing paradigms like pooling and allocating a subset of the resources to certain tasks on-demand have recently been applied to sensor networks. Motivated by the increasing set of capabilities of single nodes, Virtual Sensor Networks (VSNs) assume that the overall set of nodes in a Wireless Sensor Network (WSN) can be virtually grouped into subsets and linked to specific applications or tasks [9, 10]. Unlike traditional WSNs, where usually all nodes perform similar tasks, a subset of nodes can be allocated for a given time period to perform specific tasks based on the observations of the environment (e.g. a critical event was recognized).

Another approach exploiting virtualization techniques for sensor nodes is called Cyber-Physical Cloud Computing [6, 11]. By applying lightweight virtualization capabilities to sensor nodes, sensors basically act as servers that move in space and execute virtual sensors. Virtual sensors can migrate between physical ones, which is referred to as cyber-mobility (i.e. moving between sensor hosts). Additionally, virtual devices can move with their current sensor host (if the sensor node is mobile), which is referred to as physical mobility.

Krainer et al. [15] introduce an idea for a platform as a service (PaaS) solution that uses physical vehicle swarms in a virtualized manner to provide information acquisition services. The authors propose a virtual vehicle monitor (VVM), similar to a virtual machine monitors for server virtualization. This VVM creates a set of virtual vehicles (VV) that can be configured via a web interface. At the same time the VVM controls the physical vehicles and manages the virtual vehicle software agents. It is also responsible for the migration of software agents between physical vehicle nodes. Clients can get access to several virtual vehicles via the web interface, and program a complex information acquisition task. A use case is given by surveying the humidity and temperature in a forest to assess the risk for fires. This task is then mapped by the VVM to the physical vehicles considering the requests from other VVs.

6 Conclusion

This paper proposes a container-based approach for an Internet of Things Resource Management Platform. The approach is evaluated regarding its feasibility to run on resource-constrained Smart Devices. Using OS-level based virtualization, applications can dynamically allocate containers on available

smart devices such as smart phones or smart TVs connected to their network and utilize the offered resources more efficiently. Furthermore, the approach allows sharing the resources between multiple users and applications. The presented prototype and evaluation based on Docker shows that OS-level virtualization is a reasonable choice in terms of computational overhead and deployment time.

Future work will primarily target the improvement of our approach regarding memory and storage resources. In addition, container monitoring and scheduling strategies that take the characteristics of our presented Device Cloud Network, such as high mobility, into account, need to be investigated.

References

- [1] Ashton, K. That internet of things thing. (2009). *RFiD J.* 22, 97–114.
- [2] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. *ACM SIGOPS Operati. Syst. Rev.* 37, 164–177.
- [3] Barr, K., Bungale, P., Deasy, S., Gyuris, V., Hung, P., Newell, C., Tuch, H., and Zoppis, B. The vmware mobile virtualization platform: is that a hypervisor in your pocket? *SIGOPS Oper. Syst. Rev.* 44, 124–135.
- [4] Bell Media. *Battery Life Trumps All Other Smartphone Features for Consumers: Survey*. Available at: <http://www.ctvnews.ca/sci-tech/battery-life-trumps-all-other-smartphone-features-for-consumers-survey-1:1821084> (Accessed September 20, 2016).
- [5] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. (2012). “Fog computing and its role in the internet of things,” in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, MCC '12*, 13–16 (New York, NY: ACM).
- [6] Craciunas, S. S., Haas, A., Kirsch, C. M., Payer, H., Röck, H., Rottmann, A., Sokolova, A., Trummer, R., Love, J., and Sengupta, R. (2010). “Information-acquisition-as-a-service for cyber-physical cloud computing,” in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 14–14. (Berkeley, CA: USENIX Association).
- [7] Docker Inc. (2015). “Understanding Docker,” in *Docker Documentation* (online February 24, 2015).
- [8] Hwang, J.-Y., Suh, S.-B., Heo, S.-K., Park, C.-J., Ryu, J.-M., Park, S.-Y., and Kim, C.-R. (2008). “Xen on ARM: system virtualization using Xen hypervisor for ARM-based secure mobile phones,” in *2008 5th*

- IEEE Consumption Communication Network Confence*, 257–261 (Rome: IEEE).
- [9] Islam, M. M., Hassan, M. M., Lee, G.-W., and Huh, E.-N. (2012). A survey on virtualization of wireless sensor networks. *Sensors*, 12, 2175–2207.
- [10] Jayasumana, A. P., Han, Q., and Illangasekare, T. H. (2007). “Virtual sensor networks – a resource efficient approach for concurrent applications,” in *Fourth International Conference on Information Technology (ITNG '07)*, 111–115.
- [11] Kirsch, C., Pereira, E., Sengupta, R., Chen, H., Hansen, R., Huang, J., Landolt, F., Lip-pautz, M., Rottmann, A., Swick, R., Trummer, R., and Vizzini, D. (2012). “Cyber-physical cloud computing: the binding and migration problem,” in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 1425–1428.
- [12] Kliem, A., and Kao O. (2015). “The internet of things resource management challenge,” in *2015 IEEE International Conference on Data Science and Data Intensive Systems*, 483–490 (Rome: IEEE).
- [13] Kliem, A., Koener, M., Weissenborn, S., and Byfield, M. (2015). “The device driver engine-cloud enabled ubiquitous device integration,” in *2015 IEEE Tenth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, 1–7 (Rome: IEEE).
- [14] Kliem, A., and Renner, T. (2015). “Towards on-demand resource provisioning for iot environments,” in *Intelligent Information and Database Systems*, 484–493 (Berlin: Springer).
- [15] Krainer, C., and Kirsch, C. M. (2014). “Cyber-physical cloud computing implemented as paas,” in *Proceedings of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems*, 15–18. (New York, NY: ACM).
- [16] Laadan, O., and Nieh, J. (2010). “Operating system virtualization: practice and experience,” in *Proceedings of 3rd Annual Haifa Exp. System Conference* (New York, NY: ACM).
- [17] Larsson, A. (2016). *devicemapper – a storage backend based on Device Mapper*.
- [18] Laukkarinen, T., Suhonen, J., and Hannikainen, M. (2013). An embedded cloud design for internet-of-things. *Int. J. Distribut. Sensor Netw.* 9, 790130.
- [19] Libelium Comunicaciones Distribuidas S.L. (2014). *Top 50 Internet of Things Applications – 50 Sensor Applications for a Smarter World*. Available at: http://www.libelium.com/top_50_iot_sensor_applications_ranking/ (accessed April 01, 2016).

- [20] M2M Alliance e.V. (2007). *Machine-to-Machine (M2M) - Whitepaper*. Available at: <http://www.m2m-alliance.de/uploads/media/Whitepaper.pdf> (accessed April 01, 2016).
- [21] Mell P., and Grance, T. (2011). The nist definition of cloud computing. Available at: <http://dx.doi.org/10.6028/NIST.SP.800-145>
- [22] Docker, D. M. (2014). *Lightweight linux containers for consistent development and deployment*, Linux J.
- [23] Okajima, J. R. (2015). *Aufs4 – advanced multi layered unification filesystem version 4.x*.
- [24] OSGi Alliance. (2016). *OSGi Compendium Release 5 - Device Access Service Specification*. Technical report (accessed September 22, 2016).
- [25] Patel, M., Naughton, B., Chan, C., Sprecher, N., Abeta, S., Neal, A., et al. (2014). “*Mobile-edge computing introductory technical white paper*,” in White Paper, Mobile-edge Computing (MEC) Industry Initiative.
- [26] Perera, C., Zaslavsky, A., Christen, P., and Georgakopoulos, D. (2014). Context aware computing for the internet of things: a survey. *IEEE Commun. Surv. Tutorials* 16, 414–454.
- [27] Petazzoni, J. (2013). “BTRFS support for Docker,” in *Docker Blog*.
- [28] Raj Rajkumar, R., Lee, I., Sha, L., and Stankovic, J. (2010). “Cyber-physical systems: the next computing revolution,” in *Proceedings of the 47th Design Automation Conference*, 731–736 (New York, NY: ACM).
- [29] Renner, T., Kliem, A., and Kao, O. (2014). “The device cloud-applying cloud computing concepts to the internet of things,” in *Proceedings of the 2014 IEEE 11th International Conference on Ubiquitous Intelligence and Computing*, 396–401 (Washington, DC: IEEE Computer Society).
- [30] Rossier, D. (2012). EmbeddedXEN: A Revisited Architecture of the XEN hypervisor to support ARM-based embedded virtualization.
- [31] Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. (2007). “Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors,” in *ACM SIGOPS Operating Systems Review*, Vol. 41, 275–287 (New York, NY: ACM).
- [32] Sundmaeker, H., Guillemin, P., Friess, P., and Woelfflé S (eds). (2010). “*Vision and challenges for realising the Internet of Things*”, in The Cluster of European Research Projects on the Internet of Things, Vol. 20 (Luxembourg: EUR-OP).

Biographies

T. Renner is postdoctoral student at the Complex and Distributed IT Systems Group of Technische Universität Berlin (TUB) in Berlin, Germany. His research interests include scalable data analytics, middleware-platforms for the Internet of Things, and building distributed systems.

M. Meldau achieved his master's degree in computer science at Technische Universität Berlin (TUB), Germany, in 2015. In his master's thesis, he designed and implemented a resource sharing concept for mobile device clouds.

A. Kliem is a postdoctoral researcher at the Complex and Distributed IT Systems Group of Technische Universitat Berlin (TUB) in Berlin, Germany. His current research interests center around middleware-platforms for the Internet of Things.