Internet of Things Gateways Meet Linux Containers: Performance Evaluation and Discussion

Alexandr Krylovskiy
Fraunhofer FIT, Sankt Augustin, Germany
Email: alexandr.krylovskiy@fit.fraunhofer.de

***Abstract*—Internet of Things (IoT) applications often involve deployment of gateways at the network edge for integrating physical devices, pre-processing sensor data, and synchronizing it with the cloud. The deployment, configuration, and maintenance of the software running on the gateways in large-scale deployments is known to be challenging. In this work, we analyze the deployment requirements of IoT gateways and evaluate containerized deployment with linux containers as a potential approach addressing them. We perform several synthetic and application benchmarks providing an insight in the performance overhead introduced by linux containers and how they affect typical applications running on IoT gateways.**

I. Introduction

Internet of Things applications often need to integrate heterogeneous IoT devices, collect and pre-process sensor data, and synchronize it with the cloud. These tasks are commonly implemented in *IoT gateways* - embedded devices of varying computational power deployed at the network edge and implementing functionality according to the requirements of the application at hand. With the ubiquity of affordable System-on-Chip (SoC) computers and the growth of their computational power, it becomes feasible to empower the gateways with more functionality in a cost-effective manner.

Considering the dynamics of the IoT landscape and large diversity of IoT applications and use cases, implementing such *edge intelligence* [1] is challenging [2]. One of the challenges is the operations involved in managing IoT gateways in a large-scale infrastructure, including deployment, configuration, and upgrading their software. Considering the hardware capabilities of modern SoC computers and the functionality they provide serving as IoT gateways, their operations become comparable to that of the low-end cloud servers.

In the industry, container-based solutions are being progressively adopted for development and deployment of portable distributed applications. The underlying linux containers provide a lightweight virtualization technology allowing to gain the common benefits associated with virtualization at a low cost in performance overhead. After the success of Docker [3] - a tool for building, managing, and provision application containers, a new standardization effort[1] has been started by major cloud providers and leading industrial companies to unify the format and runtime for application containers.

In this work, we summarize the requirements to software deployment on IoT gateways and explore the feasibility of using linux containers as a base technology for this task. We evaluate the performance of Docker on two generations of Raspberry Pi SoC computer [4] performing several synthetic micro-benchmarks testing CPU, memory, disk, and network I/O as well as stress-test two services often offered by IoT gateways.

II. Related Work

Several OpenSource projects offer software for IoT gateways employing different technologies. Mihini[2] is an embedded runtime on top of linux exposing a high-level LUA API and integrating with an execution environment that allows for remote deployment and debugging. Similarly, Kura[3] is an OSGi-based framework providing a runtime and a set of services for common gateway tasks involving working with I/O, networking, cloud, and data processing. Both Mihini and Kura are tied to specific platforms for runtime, configuration and deployment, which limits them to the software developed in these technologies (C/LUA and Java). LinkSmart Device Gateway [5] avoids this limitation by integrating IoT devices via technology-independent components called *device agents*. It does not, however, provide any means for deploying device agents or managing their dependencies, leaving this task to the application infrastructure.

Several approaches to provisioning large-scale IoT infrastructures covering software deployment on IoT gateways has been proposed recently. rtGovOps [2] is a runtime framework for governance of large-scale software-defined IoT systems such as industrial fleet monitoring applications. In rtGovOps, the functionality of IoT gateways is managed centrally as *capabilities* provisioned by *rtGovOps agents* running on the gateways. The capabilities are packaged in images and provisioned on demand at runtime. In the prototype evaluation described in [2], the agents implemented as shell scripts provision MQTT and CoAP protocols as capabilities on the simulated IoT gateways.

Similarly to rtGovOps, LEONORE [6] provides an infrastructure for large-scale provisioning of resource-constrained IoT deployments. LEONORE is motivated by a use case of large-scale deployments in Building Management Systems with resource-constrained gateways of varying hardware configurations and changing functional requirements. To accommodate for these restrictions and heterogeneity, it implements custom build and packaging systems with dependencies management and builds tailored to different hardware configurations. LEONORE provides server-side building and packaging

[1] http://www.opencontainers.org/

[2] http://www.eclipse.org/mihini/

[3] http://www.eclipse.org/kura/

978-1-5090-0366-2/15/$31.00 ©2015 IEEE

of software and automatic push- and pull-based provisioning on the gateways.

Several enthusiast and start-up projects have successfully adopted linux containers for software deployment on affordable pocket-sized SoC computers [7]. resin.io[4] uses linux containers and Docker to offer an infrastructure for building software in the cloud and deploying it on the remote devices automatically. To this end, we are not aware of any studies analyzing the performance overhead and trade-offs of using linux containers on the SoC computers.

The studies evaluating the performance of linux containers compared to native hardware and other virtualization technologies [8], [9] show that containers have almost negligible overhead and equal or exceeding performance of hypervisors. Using the typical application load for benchmarking on high-grade server hardware, the authors in [8] demonstrate that containers result in a significant (20%) performance decrease in I/O intensive tasks compared to the native hardware. We use similar methodology to evaluate the performance of linux containers on lower-end hardware employed for IoT gateways.

III. Background

A. Deployment requirements

The diversity of IoT applications results in a variety of use cases that need to be supported by IoT gateways in different hardware and software configurations. Furthermore, many applications require management of large-scale deployments, which requires automated deployment and configuration infrastructure up to the network edge. Taking into account these considerations and previous studies, we emphasize the following deployment requirements on IoT gateways:

- **modularity** for support of heterogeneous IoT applications with varying and changing requirements to the gateway functionality, where functional modules can be installed, upgraded, and retired over time;
- **automation** for provisioning of large-scale IoT deployments, including initial deployment and configuration as well as future updates and maintenance;
- **vendor and technology independence** for avoiding vendor lock-in and simultaneous deployment of software from different vendors and/or implemented using different technologies;
- **dependencies management** for portable deployment of the functional modules with all of their dependencies, independent of the dependency management support of the vendor and technology they use;
- **security** for accommodating varying security requirements of applications and end-users to the deployment infrastructure;
- **low performance overhead** for efficient utilization of resources of the network and resource-constrained devices;
- **usability** for getting a wider adoption of the deployment approach and toolchain by a larger developers community.

[4]http://resin.io/

The deployment approaches described in Section II demonstrate the trade-offs involved in building deployment infrastructures addressing these requirements. E.g., focusing on the provisioning of large-scale deployments employing resource-constrained devices, LEONORE and rtGovOps minimize the footprint of the provisioning system sacrificing its functionality and usability for lower performance overhead. They create custom packaging and deployment solutions that require additional efforts and limit the available off-the-shelf software that can be deployed on the gateways.

Considering the diversity of IoT applications and use cases together with the growing capabilities of affordable SoC computers, it is tempting to adopt more resource-demanding deployment approaches. Offering more functionality and higher usability at the cost of increased performance overhead, they would provide more versatility and lower the entry barrier for developers. We believe that the containerized deployment based on linux containers, which has been adopted in the industry for similar benefits, has the potential of becoming the foundation for such deployment approaches in IoT.

B. Linux containers

Container-based virtualization [10] is a lightweight virtualization technology offering isolation of processes at the operating system level. It does not involve expensive hardware emulation as hypervisors [11], but comes at the cost of the shared kernel, which provides less isolation and limits virtualization to the host operating system. Compared to hypervisors, container-based virtualization offers several advantages including more dense deployments and smaller image sizes.

Linux containers are built on the *kernel namespaces* [12] feature, which allows creating isolated *containers* prohibiting the processes running in them to see and access resources outside of their environment. Furthermore, system resources available to individual containers can be controlled using the control groups (*cgroups*) [13] kernel feature that allows to group individual processes and control their aggregate resources consumption. Because of the little overhead involved in instantiating new containers and efficient sharing of resources between them, it becomes practical to create containers hosting individual applications – *application containers*.

Building and packaging applications with their full dependencies into application containers deployable on a variety of infrastructures is the core idea behind Docker [3]. Docker is a linux containers management tool and an infrastructure for building, shipping, and running application containers. Compared to other containers management tools, Docker offers several automation features and layered filesystem images with AUFS. The latter significantly reduces the size of container images by sharing common filesystem layers among images and storing only incremental changes of individual containers.

To provide network connectivity between containers and the host, Docker creates a bridge interface that connects all containers and the host into a virtual network and enables communication from containers to the outside network using SNAT. Placing containers in a separate network stack provides several security benefits, but to enable applications and services running inside containers to accept network connections, their ports need to be exposed on the host with DNAT. Docker also

TABLE I. HARDWARE UNDER EVALUATION

Device	Abbrev.	Year	CPU Arch/Freq/Cores	RAM
Raspberry Pi model B+	RPI B+	2014	ARMv6 / 700Mhz / 1	512 Mb
Raspberry Pi 2 model B	RPI 2	2015	ARMv7 / 900MHz / 4	1 Gb

allows to provide containers full access to the host network stack and eliminate NAT, but this enables containers to open low-numbered ports and access local network services, which is a security threat in many scenarios.

IV. PERFORMANCE EVALUATION

In this section, we evaluate the performance of Docker and underlying linux containers in several synthetic and application benchmarks and compare it to the native hardware on two generations of Raspberry Pi [4] that is often employed in IoT applications to implement the IoT gateway functionality [14].

A. Configuration and methodology

Raspberry Pi [4] is an affordable (US $35) and a vastly popular hardware platform. The devices selected for evaluation correspond to two generations of the Raspberry Pi platform representing the evolution of the IoT gateway hardware in the last few years. Their main characteristics relevant in the scope of this evaluation are summarized in Table I.

Raspberry Pi model B+ (RPI B+) is the improved revision of the Raspberry Pi model B board dating back 2012, which itself is an upgraded version of the original board with additional 256 Mb of RAM. The B+ revision includes several minor improvements, but is based on the same Broadcom BCM2835 SoC used in the original Raspberry Pi and has 512 Mb RAM as the model B from 2012. For the scope of this evaluation, the B+ model is effectively identical to the earlier model B and represents the first generation of the Raspberry Pi.

Raspberry Pi 2 model B (RPI 2) is the second generation of the Raspberry Pi platform featuring a Broadcom BCM2836 SoC with a quad-core ARM Cortex-A7 CPU and 1 Gb of RAM. The new board keeps the affordable price of its predecessor while significantly increasing its capabilities. Released in the early 2015, the new Raspberry Pi board represents the next generation of affordable SoC computers that will drive IoT applications in the next years.

For all tests, a 16 Gb SanDisk Extreme® PLUS microSDHC™ UHS-I Card was used on both boards. Docker 1.6.0 was used as the linux containers management tool provided by the Hypriot image [7] running Raspbian Jessie with Linux 3.18.11. The benchmarks involving network communication were performed from an Intel Core 2 Duo PC running Linux 3.13.0 with a Broadcom BCM5787M Gigabit Ethernet card. The PC was directly connected to the NIC of the board under test providing a 100BASE-TX Fast Ethernet connection.

If not stated otherwise, all results are averaged over 10 runs and presented in tables as follows. For *native* tests (running natively on the hardware), the tables show both means and standard deviations in absolute values. For *docker* tests (running in Docker containers), the tables show percentage deviations from the corresponding means of the *native* tests and standard deviations in absolute values. The plots depict mean values and show standard deviations as bars.

TABLE II. CPU BENCHMARKING: NBENCH

DEV	CONF	MEMORY	INTEGER	FLOAT
RPI 2	native	4.27 (0.04)	5.63 (0.03)	4.75 (0.02)
	docker	**+0.75% (0.04)**	**+0.2% (0.02)**	**+0.29% (0.02)**
RPI B+	native	2.42 (0.03)	3.15 (0.01)	2.09 (0.04)
	docker	**+0.45% (0.04)**	**+0.34% (0.01)**	**+1.62% (0.01)**

TABLE III. MEMORY I/O BENCHMARKING: STREAM

DEV	CONF	SCALE, Mb/s	ADD, Mb/s	TRIAD, Mb/s
RPI 2	native	709.22 (0.29)	621.5 (1.55)	518.01 (1.08)
	docker	**+0.57% (3.21)**	**+1.08% (5.31)**	**+2.97% (2.58)**
RPI B+	native	212.48 (0.08)	302.81 (0.12)	289.96% (0.12)
	docker	**-0.08% (0.07)**	**+0.07% (0.15)**	**-0.07% (0.16)**

TABLE IV. NETWORK I/O BENCHMARKING: IPERF

DEV	CONF	BW_TCP, Mbps	BW_UDP, Mbps
RPI 2	native	51.62 (0.4)	94.88 (2.87)
	docker	**-7.65% (0.07)**	**-0.51% (2.47)**
	dockerhost	**+0.43% (0.53)**	**+0.97% (0.0)**
RPI B+	native	78.51 (0.31)	94.66 (2.75)
	docker	**-48.73% (0.38)**	**-92.79% (2.04)**
	dockerhost	**-16.58% (0.53)**	**-0.54% (3.75)**

B. Synthetic benchmarks

Synthetic benchmarks evaluate the CPU, memory, disk, and network I/O performance using benchmarking tools to stress individual subsystems in order to isolate and understand the overhead introduced by the linux containers and Docker.

CPU. nbench [15] is a single-threaded benchmarking tool for measuring the CPU, FPU, and memory system performance. It runs several algorithmic tests and calculates three indexes: *Integer*, *Floating Point*, and *Memory*. The results of running nbench on the selected platforms are shown in Table II. The results demonstrate the expected difference in performance between different hardware generations (ARMv6 700Mhz vs. ARMv7 900Mhz) and show that the linux containers have a negligible impact on the CPU performance, even outperforming native tests up to 1.6% in some cases.

Memory I/O. STREAM [16] measures memory performance using simple vector operations: *Copy*, *Scale*, *Add*, and *Triad*. The results of the latter three are shown in Table III (the Copy operation has shown to be the fastest and its results are omitted for brevity). Similarly to the CPU, memory tests show almost 2x speed difference between the two generations of the Raspberry Pi and almost negligible overhead of linux containers and Docker on both platforms.

Network I/O. To evaluate the network I/O performance, we focus on two main measures relevant for most practical applications: bandwidth and latency. As described in Section III, Docker employs NAT, which has been shown to have a significant performance impact in [9]. To localize this factor, we include an additional configuration *dockerhost* in each experiment where Docker containers use the host interface directly by starting containers with the `--net=host` option.

To measure the network bandwidth, we use unidirectional TCP and UDP tests of the iperf [17] network benchmarking tool. The results of iperf bandwidth tests are shown in Table IV, where *BW_TCP* and *BW_UDP* are the measurements of the bandwidth using the TCP and UDP protocols correspondingly.

TABLE V. NETWORK I/O BENCHMARKING: NETPERF

DEV	CONF	RTT_UDP, μs	RTT_TCP, μs
RPI 2	native	523.61 (10.41)	496.74 (9.9)
	docker	**+0.92% (1.49)**	**+1.23 (0.97)**
	dockerhost	**+3.25% (2.17)**	**+3.51% (3.87)**
RPI B+	native	827.89 (1.20)	778.35 (2.63)
	docker	**+30.97% (2.96)**	**+28.84 (3.38)**
	dockerhost	**+0.99% (4.13)**	**+1.19% (7.38)**

TABLE VI. DISK I/O BENCHMARKING: BONNIE++

DEV	CONF	BCK_IN, Kb/s	BCK_OUT, Kb/s	REWRITE, Kb/s
RPI 2	native	23207.4 (29.66)	18613.0 (876.39)	8991 (230.31)
	docker	**+0.75% (26.48)**	**-2.58% (1157.25)**	**-1.15% (209.17)**
RPI B+	native	22290.70 (84.12)	16736.5 (831.11)	8584.8 (395.76)
	docker	**+0.04% (18.1)**	**+1.4% (767.95)**	**-0.58% (456.47)**

The results on the RPI 2 show 7.65% decrease of TCP bandwidth in the *docker* test, while UDP bandwidth is almost not affected. Disabling the NAT shows the performance equal to the *native* configuration. On the RPI B+, the NAT overhead is much more pronounced: in the *docker* configuration, the TCP bandwidth reduces almost by half, and the UDP bandwidth suffers a 93% decrease effectively reducing to < 7 Mbps. Disabling the NAT alleviates the problem, but the TCP performance remains almost 17% worse that the *native* configuration. Monitoring the CPU usage during the tests, we notice that the iperf TCP bandwidth tests saturate the CPU on RPI B+, while on RPI 2 the CPU usage remains below 10%.

To measure the latency, we use the *request-response* test of netperf [18], which measures round-trip latency from the number of request/response transactions and the test run time. The results of the netperf transaction latency tests are shown in Table V. In contrast to the previous results benchmarking high-grade server hardware using gigabit links in [9], [8], the results on RPI 2 show almost negligible impact on the latency caused by Docker in the tested scenario of 100 Mbps network. Disabling the NAT provides more volatile results, but does not impact the latency performance significantly either. The results on the RPI B+, on the other hand, support the previous findings and show over 30% increase in the round-trip latency in the *docker* test. Disabling the NAT improves the results significantly and allows to practically reach the *native* performance.

Disk I/O. Bonnie++ [19] is a disk I/O benchmarking tool performing a number of simple tests of the physical storage and file system performance. The tool was configured to use the test file of twice the size of the system memory on each device: 1Gb for RPI B+ and 2Gb for RPI 2. The results of the tests are shown in Table VI, where *BCK_IN* is the sequential block reading, *BCK_OUT* is the sequential block writing, and *REWRITE* is a sequence of read-modify-write operations testing the filesystem cache effectiveness. The results on both platforms show no significant overhead of Docker and AUFS on the disk I/O.

Summarizing the results of the synthetic benchmarks, the only significant performance overhead introduced by Docker has been identified in the network I/O tests. This overhead is evidently caused by using NAT and has a much larger impact on the RPI B+ where it quickly saturates the CPU and results in a significant performance degradation.

C. Application benchmarks

To evaluate the performance overhead caused by Docker and linux containers in practical scenarios, we perform benchmarks of two typical services running on IoT gateways: Mosquitto MQTT broker [20] and LinkSmart Device Gateway [5]. The former is a ubiquitously used service for transmitting sensor data streams, and the latter implements a typical gateway functionality communicating with physical sensors and providing higher-level APIs for accessing their data over the network by applications. In these tests, we focus on *throughput* and *latency*, which are measured similarly in both cases as described below.

MQTT Broker. Mosquitto [20] is an OpenSource message broker implementing the MQTT publish/subscribe protocol that is ubiquitously used in IoT applications to transmit sensor data streams. To measure its performance, we have implemented a simple benchmarking tool[5] that publishes messages to an MQTT broker with configurable message size and number of concurrent clients at an unbound rate. The *throughput* is measured as the average number of messages each client is able to publish per second. Using MQTT QoS 1 for all outgoing messages, *latency* is measured as the average time a message spends in processing from its submission to the outgoing queue on the client until receiving a publish acknowledgment from the broker. We use 100 bytes payload size varying the number of concurrent clients and test the same configurations as in the network I/O tests described in the previous section.

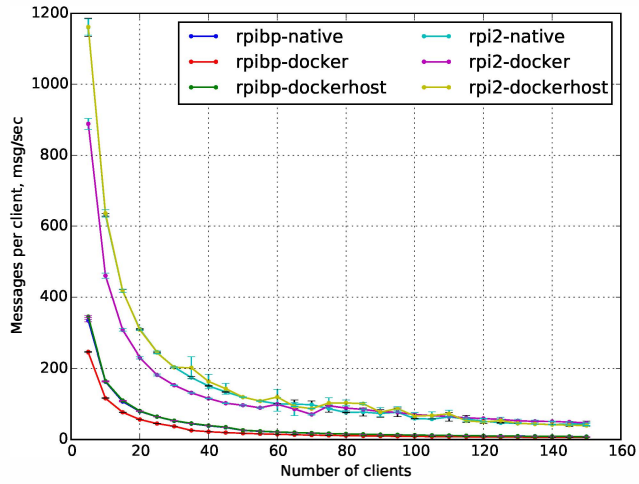
Figure 1a depicts the results for throughput and shows that it decreases dramatically with the increasing number of clients in all configurations. At the lower number of concurrent clients, Docker with NAT enabled results in up to 26% performance degrade, while with NAT disabled it shows practically the same performance as the native hardware on both RPI B+ and RPI 2. The NAT impact on the throughput decreases with increasing number of concurrent clients and converges at around 60 clients. This is evidently caused by the single-threaded mosquitto broker as it quickly saturates the CPU with the increasing number of concurrent clients independent of the NAT configuration. It is worth noting that despite not being able to take advantage of the multi-core CPU, the RPI 2 still shows more that 3x better performance compared to the older model.

Figure 1b depicts the latency results for the same experiment and shows its linear growth with the increasing number of clients. The results of RPI 2 show almost negligible difference of both Docker configurations compared to the native hardware with the NAT configuration showing slightly lower performance under smaller number of clients. The results of RPI B+, on the other hand, demonstrate a significant overhead of the NAT-enabled Docker configuration that is growing with the number of concurrent clients and reaches 30% under 150 clients.

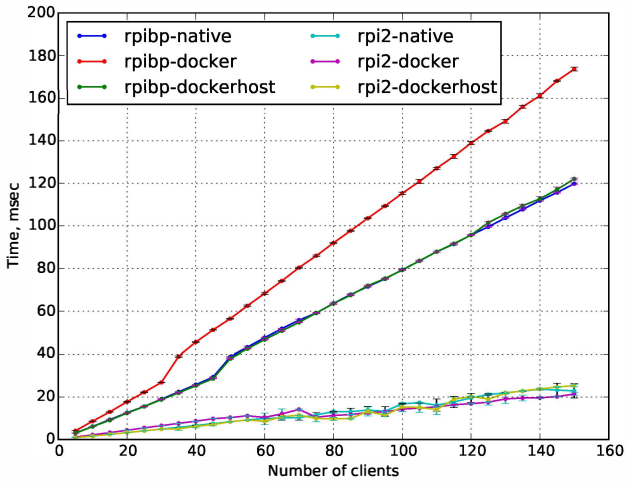
LinkSmart DeviceGateway. Device Gateway [5] is a LinkSmart middleware[6] component providing integration of IoT devices and translation of lower-level hardware protocols

[5]https://github.com/krylovsk/mqtt-benchmark

[6]http://linksmart.eu

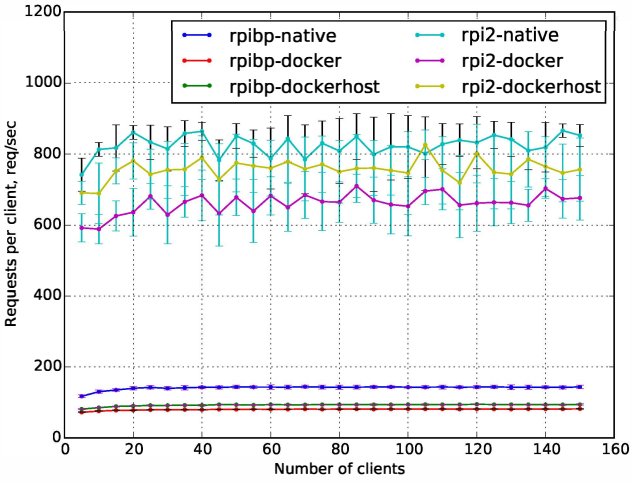


(a) Throughput

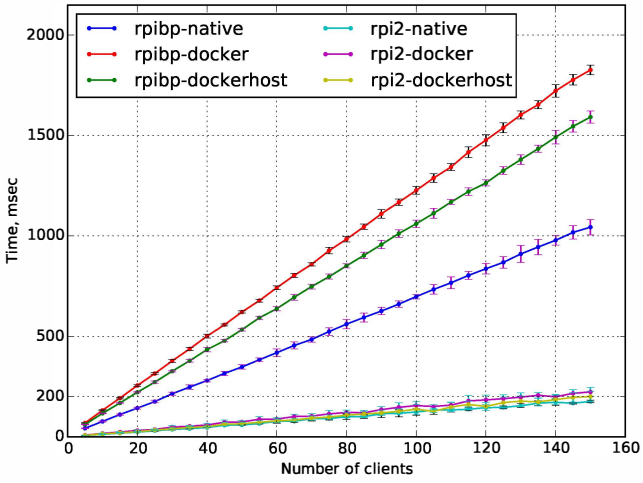


(b) Latency

Fig. 1. Mosquitto MQTT broker performance results



(a) Throughput



(b) Latency

Fig. 2. LinkSmart Device Gateway performance results

into common protocols used by IoT applications. For this benchmarking scenario, we use an iAM indoor air monitor sensor[7] connected to the USB port and integrated with the Device Gateway through a simple *device agent* that continuously reads the sensor measurements and writes them to `stdout`. The latest measurement from the device agent output is then cached by the Device Gateway and exposed via HTTP where it can be obtained by the applications in SenML/JSON format resulting in HTTP body of 52 bytes. To provide the Device Gateway an access to the USB port when running in a container, the Docker container is started with the `--device` option. For benchmarking, we use the Apache HTTP server benchmarking tool [21] varying the number of concurrent clients similar to the MQTT test. The throughput is measured as the average number of requests per second among all clients, and latency as the average request processing time among all requests.

[7] http://ams.com/eng/Products/Chemical-Sensors/Air-Quality-Sensors/iAM

The benchmarking results for throughput are shown in Figure 2a. The throughput in all experiments converges at around 15 clients and while showing volatility does not change significantly within the tested configurations of maximum 150 clients. The Device Gateway is implemented in Go with concurrent handling of HTTP requests by the `net/http` package, which allows it to take advantage of the multi-core RPI 2 processor resulting in up to 6x throughput per client compared to the single-core RPI B+. The impact of using Docker with and without NAT is similar to the MQTT results with NAT-enabled configuration showing the worst performance, especially on the RPI B+. Disabling the NAT improves performance, but does not allow it to reach the native hardware on both platforms.

Similar results demonstrate the latency measurements depicted in Figure 2b. In experiments on RPI 2, the native hardware has the edge in performance under almost all conditions with Docker showing about 20% and 10% higher latency with

NAT-enabled and -disabled configurations correspondingly. Compared to RPI 2, the results of RPI B+ show up to 5x higher latency on native hardware and a significantly worse performance when run in a Docker container, both with and without NAT enabled.

Summarizing the results of application benchmarks, it can be concluded that using linux containers and Docker on the older RPI B+ results in a significant performance degradation, which supports the earlier conclusions of synthetic benchmarks. Moreover, the HTTP testing with Device Gateway shows that NAT is not the only cause of the performance decrease and disabling it does not allow to reach the performance close to the native hardware. On the RPI 2, on the other hand, the results are much more promising. While using Docker results in a measurable overhead in some scenarios, it is clearly attributed to NAT and the results are predictable. Disabling the NAT allows to reach the performance that for most practical application is comparable to the native hardware.

V. Conclusion and future work

In this work, we have performed several synthetic and application benchmarks of Docker and underlying linux containers on two versions of the Raspberry Pi SoC platform representing two generations of IoT gateways hardware. While the results on the previous generation RPI B+ show a significant overhead that cannot be easily mitigated, the results of the recent RPI 2 platform are promising and comparable to the earlier findings evaluating linux containers on high-end servers in [8].

While there are clear benefits offered by containerized deployment in the cloud applications, this deployment approach has not yet been deeply investigated in IoT domain and on IoT gateways in particular. Demonstrating the overall feasibility of the approach and isolating its performance overhead, this work encourages further investigation of its possible adoption on the IoT gateways. While addressing many of the deployment requirements of common IoT applications, the performed evaluation shows that it also introduces measurable overhead and cannot substitute more light-weight deployment approaches. Moreover, for successful adoption in IoT, linux containers might require additional tooling beyond that offered by existing tools such as Docker.

To make an informed decision and give specific recommendations on employing this deployment approach in a specific IoT application, more understanding of the overall IoT landscape and a case-by-case analysis are needed. Such analysis in a more generic sense might be feasible over time as the IoT field matures and more first-hand experience in different applications is gathered. Until then, linux containers appear as an attractive deployment technology for investigation offering a great deal of flexibility under relaxed resource constraints.

Acknowledgment

This research is funded by EU FP7 SMARTCITIES 2013 District Information Modelling and Management for Energy Reduction – DIMMER.

References

[1] M. Enescu. (2015) Paradigm Shift with Edge Intelligence. [Online]. Available: "http://blogs.cisco.com/openatcisco/paradigm-shift-with-edge-intelligence"

[2] S. Nastic, M. Vogler, C. Inzinger, H.-L. Truong, and S. Dustdar, "rtGovOps: A Runtime Framework for Governance in Large-Scale Software-Defined IoT Cloud Systems," in *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2015 3rd IEEE International Conference on*, March 2015, pp. 24–33.

[3] Docker: Build, Ship and Run Any App, Anywhere. Docker, Inc. [Online]. Available: https://www.docker.com/

[4] Raspberry Pi – Teach, Learn, and Make with Raspberry Pi. [Online]. Available: "https://www.raspberrypi.org/"

[5] LinkSmart Device Gateway. Fraunhofer FIT. [Online]. Available: https://linksmart.eu/redmine/projects/linksmart-local-connect/wiki/Device_Gateway

[6] M. Vogler, J. Schleicher, C. Inzinger, S. Nastic, S. Sehic, and S. Dustdar, "LEONORE – Large-Scale Provisioning of Resource-Constrained IoT Deployments," in *Service-Oriented System Engineering (SOSE), 2015 IEEE Symposium on*, March 2015, pp. 78–87.

[7] G. Fichtner, D. Reuter, S. Scherer, M. Renner, and A. Eiermann. (2015) Docker 1.6.0 is finally released into the wild. [Online]. Available: "http://blog.hypriot.com/post/docker-1-6-is-finally-released-into-the-wild/"

[8] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, March 2015, pp. 171–172.

[9] R. Morabito, J. Kjallman, and M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, March 2015, pp. 386–393.

[10] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 275–287.

[11] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974.

[12] E. W. Biederman and L. Networx, "Multiple instances of the global linux namespaces," in *Proceedings of the Linux Symposium*. Citeseer, 2006.

[13] P. Menage. CGROUPS. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt

[14] M. Jung, J. Weidinger, W. Kastner, and A. Olivieri, "Building Automation and Smart Cities: An Integration Approach Based on a Service-Oriented Architecture," in *Proceedings of the 2013 27th International Conference on Advanced Information Networking and Applications Workshops*, ser. WAINA '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1361–1367.

[15] U. F. Mayer. (2011) Linux/unix nbench. [Online]. Available: "http://www.tux.org/ mayer/linux/bmark.html"

[16] J. D. McCalpin. (2007) STREAM: Sustainable Memory Bandwidth in High Performance Computers.

[17] Iperf - The TCP/UDP Bandwidth Measurement Tool. [Online]. Available: "https://iperf.fr/"

[18] R. Jones. (2012) The Netperf Homepage. [Online]. Available: "http://www.netperf.org/"

[19] R. Coker. (2001) Bonnie++ now at 1.03e (last version before 2.0)! [Online]. Available: "http://www.coker.com.au/bonnie++/"

[20] Mosquitto: An Open Source MQTT v3.1/v3.1.1 Broker. "Eclipse Foundation". [Online]. Available: "http://mosquitto.org/"

[21] Apache HTTP server benchmarking tool. The Apache Software Foundation. [Online]. Available: "http://httpd.apache.org/docs/2.2/programs/ab.html"