Towards Multi-container Deployment on IoT Gateways

Koustabh Dolui
imec-Distrinet
Katholieke Universiteit Leuven
Email: koustabh.dolui@kuleuven.be
ORCID: 0000-0002-2690-4038

Csaba Kiraly
OpenIoT Research Unit
FBK CREATE-NET
Email: kiraly@fbk.eu
ORCID: 0000-0002-6839-5024

Abstract—Stringent latency requirements in advanced Internet of Things (IoT) applications as well as an increased load on cloud data centers have prompted a move towards a more decentralized approach, bringing storage and processing of IoT data closer to the end-devices through the deployment of multi-purpose IoT gateways. However, the resource constrained nature and diversity of these gateways pose a challenge in developing applications that can be deployed widely. This challenge can be overcome with containerization, a form of lightweight virtualization, bringing support for a wide range of hardware architectures and operating system agnostic deployment of applications on IoT gateways. This paper discusses the architectural aspects of containerization, and studies the suitability of available containerization tools for multi-container deployment in the context of IoT gateways. We present containerization in the context of AGILE, a multi-container and micro-service based open source framework for IoT gateways, developed as part of a Horizon 2020 project. Our study of containerized services to perform common gateway functions like device discovery, data management and cloud integration among others, reveal the advantages of having a containerized environment for IoT gateways with regard to use of base image hierarchies and image layering for in-container and cross-container performance optimizations. We illustrate these results in a set of benchmark experiments in this paper.

Index Terms—Docker, containerization, Internet of Things, gateway, edge computing, cloud computing, fog computing

I. INTRODUCTION

Over the recent years, the concept of Internet of Things has gradually evolved from a paradigm to create a network of objects connected to the Internet to an interconnected network of data producers and data consumers. Regular day-to-day objects equipped with sensors act as data producers generating data sensed from the surrounding environment while software applications as well as end-devices equipped with actuators act as data consumers performing actions based on the data gathered. This notion of IoT has led to its application to various domains like health-care, autonomous transport, smart cities, among others which leverage the data generated from end devices to gain meaningful insights on the generated data. Due to the resource-constrained nature of these IoT end-devices, the scope of storage and data processing on these devices is limited. Thus, the data processing and business analytics are performed on cloud platforms and software services running on the cloud. However, with the number of connected devices predicted to grow up to 75 billion by 2025 [1], the data processing and storage architecture based solely on the cloud is facing a few challenges.

The majority of IoT applications are heavily dependent on cloud storage and processing which affects the end-to-end latency and results in inefficient utilization of resources. These challenges, together with privacy and security considerations, have prompted a move away from the centralized architecture of storage and processing on the cloud to bring the processing and storage closer to the end-devices with the edge computing model. The edge computing model leverages a set of devices in the architecture between the end devices and the cloud. These devices can be legacy devices present in the network with storage and processing capabilities [2] or dedicated devices deployed to serve this purpose like IoT gateways and cloudlet devices[3].

The presence of the edge computing layer aids offloading of data processing and validation to the edge layer. Moreover, it facilitates implementation of collaborative computing among end-devices as well as implementation of device and data management policies. However, the edge layer devices are not usually resource enriched; thus following the cloud oriented approach of hypervisor-based virtualization can prove to be cumbersome on these devices. Moreover, the edge layer devices are heterogeneous in terms of their hardware specifications, processor architecture and operating systems running on the devices. Thus, light-weight virtualization in the form of containerization offers a suitable solution to the concerns addressed above. Containerization allows virtualization at the OS-level, leveraging the kernel of the operating system to offer isolated user-spaces to each container. Thus, containerization facilitates the implementation of a microservice architecture, where each functionality on the edge device is developed as a service running inside each container. Containerization offers the flexibility to develop different services in different programming languages and communication among the containers using well-defined APIs.

In this paper, we present AGILE, an open-source framework for IoT gateways offering services including device and

protocol management, data storage, security and access control. AGILE is designed based on a microservice architecture with each of the services above deployed in separate containers. Such containerization provides multiple advantages, but comes with a performance overhead. We use AGILE as a case-study to observe the overhead associated with containerization over a conventional approach and discuss improvements achieved by applying techniques like cross-container optimization and in-container optimization.

The rest of the paper is structured as follows. In section II, we discuss about the state of the art in this research area highlighting the work on the performance measurements on containerized approaches as well as work on containerization in the IoT context. In section III, we discuss the services offered by the AGILE framework and the corresponding architecture. Section IV highlights the different approaches for optimizing the performance of containers for edge layer devices. In section V, we present our observations from the performance tests carried out on AGILE, concluding our work in section VI.

II. RELATED WORK

The growth in both the number of IoT devices and IoT applications in a wide array of domains has brought about new requirements to the IoT ecosystem which include location awareness, geo-distribution of processing nodes and low latency in device-cloud communication. This has led to a burden on the traditional resources for IoT including networking, storage and processing resources. Consequently, considerable amount of literature has been published on the use of Single Board Computers (SBCs) like Raspberry Pi as an intermediate processing layer [4][5] and an enabler for various IoT applications[6][7]. Moreover, the suitability of networking resource virtualization for IoT including Software Defined Networks (SDNs) and Network Virtualization among others [8][9] have also been discussed in existing work. Our study on the existing literature focuses on virtualization of storage and processing resources at the OS-level in the form of containerization. We study two aspects of containerization in the IoT context, first, the existing studies on the suitability and performance of containerization on resource-constrained devices and second, the application of containerization to different use-cases of IoT.

A. Suitability of Containerization

Previous studies have focused on the tradeoffs in applying hypervisor based virtualization and lightweight containerization to edge devices. The authors of [10] illustrate the advantages of containerization over hypervisor based hardware virtualization in terms of size of the resources, flexibility and portability. The authors state that hypervisor based virtualization is more suited for Infrastructure-as-a-Service on the cloud than containerization, which offers a portable runtime, easier deployability on multiple servers and interconnectivity among containers. These advantages, on the other hand, make containerization more suitable for the edge

layer in a Platform-as-a-Service scenario [11]. Pahl et. al [4] leverages the resources of Raspberry Pi devices to further build a cluster of containers running on multiple devices in the PaaS context. The cluster is designed to perform computationally intensive tasks including cluster and data management, overcoming the resource-constrained nature of each device.

A significant amount of research is aimed at conducting performance tests to analyze the behavior of edge devices with the implementation of containerization. The authors of [12] study the performance of VM based virtualization and containerization against a native approach in terms of CPU cycles, disk throughput and network throughput. The results show that containerization outperforms VM based virtualization for memory I/O, network throughput and CPU cycles. Morabito et. al [13] studies the processing resource and power consumption for performing different tasks on Raspberry Pi for wired and wireless connectivity. These tasks include running a containerized CoAP server for processing data, as well as for performing sensing actuation and video analytics. The author of [14] performs benchmark tests on Raspberry Pi B+ and Raspberry Pi model B in terms of Disk and Network I/O for native and containerized approaches. While the overhead is very high for the Raspberry Pi B+, significant performance improvements are observed for Raspberry Pi 2.

In the above studies, the benchmarks clearly show that containerization is feasible on System on Chip (SOC) devices like Raspberry Pi 2 and it is more optimized than using VM based virtualization. However, there is a lack of studies on how the process of containerization itself can be optimized and benchmarks tests for the same. Existing literature shows approaches to deploy containers in a cluster of devices, however, there exists a gap in terms of multi-container deployment and optimization on a single device.

B. Containerization in IoT use cases

Several articles in existing literature present applications of IoT based on containerization in different use cases. The authors of [15] demonstrate a distributed and layered architecture for Industrial IoT based applications with deployment of docker containers on end devices, the gateway and the cloud. Chesov et. al [16] present a multi-tier approach to containerize different functionalities in the smart cities context like data aggregation, business analytics and user interaction with data and deploy the containers on the cloud. Kovatsch et. al simplify programmability of IoT applications by exposing scripts and configurations using a RESTful CoAP interface from a deployed container [17].

The existing literature applies containerization to different use-cases and specific areas of IoT. However, there is a lack of a framework based on containerization which can be applicable to multiple use cases and applications of IoT. We try to address this gap with AGILE, to build a modular framework based on containerization to offer services abstracted from various use cases and applications of IoT.
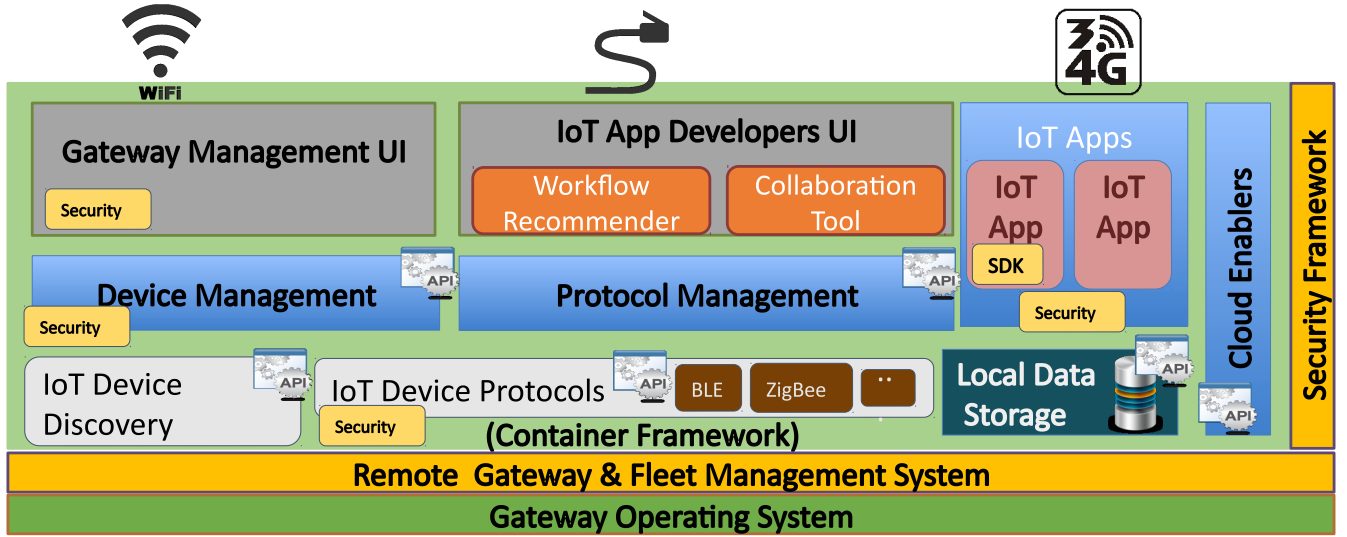
Fig. 1. Overview of the AGILE modular architecture

III. AGILE FRAMEWORK

The AGILE gateway, which stands for an Adaptive & Modular Gateway for the IoT, was conceived to design and implement a truly modular gateway in terms of hardware and software and to be adaptive to handle various types of devices, networking interfaces, communication protocols, and use cases.

A. AGILE Microservices

AGILE is aimed at developing an open source software and hardware framework for IoT development. The hardware framework involves development of two separate versions of the gateway, a maker's version supporting fast prototyping based on the Raspberry Pi 3 board while the industrial version is being developed for use cases requiring rugged hardware and being ready for production. In the following subsection, we elaborate the software framework for AGILE as illustrated in Fig. 1.

1) Device Management: The device management handles addition and removal of new devices to the gateway. The device manager supports a set of devices which offer interfaces like reading and writing to the device as well as execution of methods offered by the device. The devices communicate using one or more protocols supported by the protocol manager.

2) Protocol Management: The protocol manager offers interfaces to add and remove protocols as well as support for implementations of underlying methods for each protocol including device discovery. The protocol manager supports a set of protocols which offer interfaces like read, write, connect and disconnect with the devices implementing the protocol.

3) Local Data Storage: The data storage component provides timeseries based storage for data generated by IoT sensors, with support for retention policies and encryption.

4) Gateway Management UI: The gateway management UI allows the user to manage and access functionalities on the gateway, to start and stop other services like device discovery as well as access to data and visualization of the stored data.

5) IoT App Developers UI and SDK: The IoT App Developers UI is aimed at offering a user-friendly graphical interface to create application logic by wiring together AGILE specific and generic nodes in an application workflow. The applications can e.g. include data collection and storage, rules applicable on the data to implement sensing-actuation use-cases, as well as analytics on sensor data or on the data stored on the gateway. A separate software development kit (SDK) facilitates development of IoT applications written directly in JavaScript, while apps written in other languages can also directly use the APIs provided by gateway microservices.

6) Cloud Enablers: The AGILE framework supports multiple cloud providers including Xively and Google Drive to process and push the data collected from the device and store them on these cloud platforms. This allows seamless end-to-end connectivity from the peripheral devices up to the cloud platforms.

B. Containerized Architecture for AGILE

The software framework for AGILE is designed using a microservice architecture. The rationale behind following the microservice oriented approach are the following: (i) The services offered are split into components which can interact with each other over required and provided interfaces, (ii) Using this approach, the components are easily scalable and adaptable to changing requirements in the system.

The implementation of the software framework is achieved by containerizing the offered microservices. The containerization engine we have used for AGILE is Docker due
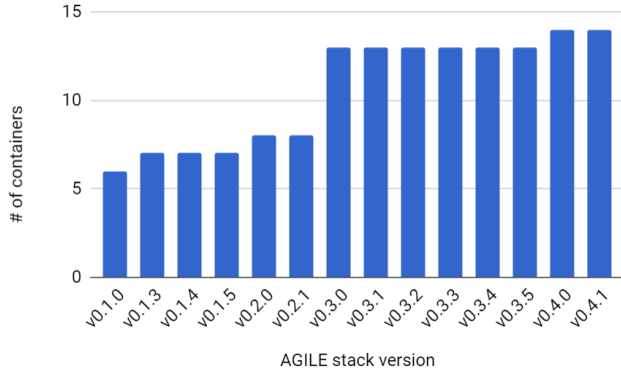
Fig. 2. Number of containers for each AGILE stack version

to its wide-scale adoption, documentation and support for multiple architectures. The functionalities mentioned in the previous subsection are implemented individually in different Docker containers. The framework is language-agnostic, allowing development in any language, and thus, a wider choice of open-source code to be reused. Moreover, software dependency conflicts are easy to overcome since each containerized service has its own file system namespace.

The core functionality of the gateway which includes device management and protocol management is containerized exposing the interfaces for the HTTP REST API endpoints. The protocols are implemented in individual containers which include ZigBee and BLE. These core containers communicate with each other over the DBus which is implemented in a containerized form as well.

The Gateway management UI is implemented using OS.js, a JavaScript based Web Desktop platform, in a containerized environment. This container depends on agile-core to offer access to the other microservices. The developers UI leverages the Node-RED tool deployed in a separate container. The cloud integration modules are provided as nodes inside the Node-RED tool. The number of containers used for implementing incremental versions of the AGILE stack is illustrated in Figure 2; which acts as a premise for our experimental setup.

IV. CONTAINERIZATION AND IoT

Containerization of the micro-services provides several advantages in the design, development, deployment, security, and management of the gateways, however it also brings non-negligible overhead in other areas, most notably in resource consumption.

From the design perspective, the containerized architecture maps well to the micro-service principle, providing clear boundaries between individual services. Containerization also allows fine-grained control over access to system resources as well as restricted interactions between containers based on access policies. For example, in case of AGILE, only protocol adapters dealing with network interfaces are

allowed access to system devices specifically only to those interfaces they require. Containerization facilitates version management of individual components, while also simplifying the whole micro-service composition. Finally, we should note that due to the large number of Edge/GW class hardware platforms, support of Linux distributions and package repositories is often lagging behind their server and desktop counterparts. Containerization overcomes this by requiring only an up-to date kernel and the container framework to be installed on the host. The rest is containerized and available on all CPU compatible platforms.

Containerization can also simplify the development process if the build process is also containerized. In this case, tools in the host file system are not involved in the build process, ensuring that the service is built from source in an entirely reproducible way.

However, there is considerable overhead involved in containerization, especially if we consider a multi-container installation with dozens of containers deployed on a single gateway class machine. Most prominent is the overhead in the overall size of deployment, which translates into an overhead in cost since higher reliability in the form of Embedded MultiMediaCard (eMMC) memory is more expensive than Secure Digital (SD) cards. Due to the possibility of limited and intermittent connectivity, download or update times affect the performance of the system. Due to the resource constrained nature of the gateways in comparison to the cloud, high resource utilization and build times for containers add to the overhead.

As mentioned earlier AGILE relies on Docker for containerization. In a Docker based multi-container environment each micro-service has its own file system image generated using a "Dockerfile": the shell script like "recipe" to build, install, and run the service. To allow sharing content between containers and to speed up generation of images, Docker uses layered images where each layer contains a set of files adding, overwriting, or deleting files from the union of previous layers. The image generation starts from an image referred to as the "base image", followed by addition of build dependencies, build tools, build artifacts, runtime dependencies, and language runtimes. The overall size for all the images of all micro-services can be significant.

To reduce the overall size of the distribution and to address the aforementioned overhead we define a novel taxonomy to propose the following optimization techniques.

A. In-container image layering optimizations

In container image layering optimization is aimed at reducing overhead during the generation and deployment of individual micro-services. Techniques illustrated on Fig. 3 are:

1) self-contained Dockerfile: In this case, which serves as our baseline for the following optimizations, each of the above mentioned steps in the process of generating the final image from the base image generates one layer or more in
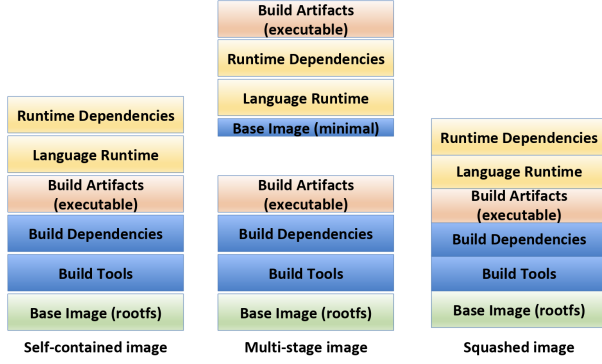
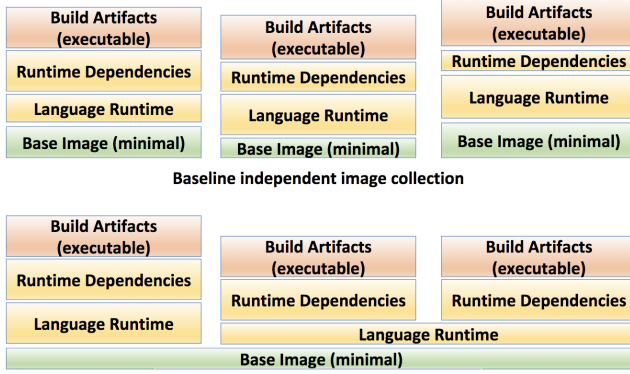Fig. 3. Layered image stack for in-container optimization

Fig. 4. Layered image stack for cross-container optimization

the layered file-system of the container depending on the complexity of each step.

2) multi-stage Dockerfile: In case of the multi-stage build, first a dockerized "build image" is created comprising of the base image, build dependencies and build steps. Leveraging the build image, a special "deployment image" is created containing only the runtime environment and the actual executables[1].

3) image squashing: The image squashing technique creates a single-layered or monolithic image from a multi-layered image. In a multi-layered image, the generation of a new layer makes all previous layers immutable. Thus, if a file is modified or deleted, the old content is residual, increasing overall image size. This overhead of larger image sizes is mitigated by the image squashing technique.

B. Cross-container optimizations

When multiple containers are deployed on the same device, a further optimization is possible as shown on Fig. 4: containers from the set of images can rely on common layers.

1) baseline image collection: Our baseline is a multi-container setup where each image is generated by developers choosing their base images independently. In practice, this

can result in a lack of common layers among the images, and the overall distribution size becomes the sum of individual images.

2) base image hierarchy based optimization: To overcome the previous problem, we have introduced images based on a hierarchy of base images with the objective of maximizing the number of shared base layers. This hierarchy[2] of base images is built as follows: firstly, a series of lean CPU architecture specific images are created. Secondly, based on each of these images, a series of Linux distribution specific images are created. The significance of this second layer is to provide broad support for installation of both build and runtime dependencies. In the third layer of the hierarchy, images of the previous layer are leveraged to generate images for every relevant language build environment and runtime. By forcing the selection of images from this hierarchy, we achieve two important goals: (i) the number of layers that are shared between our deployed images is maximized, for example a Java Runtime Environment (JRE) will only be deployed once, and not in all images running Java code. (ii) Since all base images in the hierarchy exist for all CPU architectures, supporting multiple architectures becomes straightforward by generating CPU specific versions of our own images.

V. OBSERVATION

In this section, we present our study of the performance of our multi-container microservice based deployment on the AGILE gateway. We study the performance improvements in terms of the container sizes and container download/installation times.

A. Experimental setup

For the experiments, we have used the makers' version of the gateway which consists of an ARM Cortex A53 processor clocked at 1.2 GHz, 1 GB of RAM and a 32 GB Samsung EVO Plus Class 10 SD card. The installed operating system on the device was Raspbian Jessie, and it was connected to the Internet with a dedicated 50/10 Mbps FTTC/VDSL2 line. The following containers comprising the AGILE stack were used for the experiment. The updates to the overall stack were pushed and released monthly while individual containers were updated biweekly on average.

1) agile-core: : The agile-core container is built from a ZuluJDK base image and is dependent on the following containers, (i) agile-dbus, (ii) agile-devicemanager, (iii) agile-protocolmanager and (iv) agile-devicefactory. If the containers required by agile-core are already built and available on the device, they are reused, or they are built as well during the build process.

2) agile-ble: : The agile-ble container is also built from a ZuluJDK base image and depends on the agile-core docker image. The agile-ble container registers a dbus object which can be leveraged by the agile-protocolmanager. The dbus

---

[1]While previously multi-stage build required external tools, support was included in docker with version 17.05-ce. With this we have also migrated our containers to use the new framework.

[2]We rely on and contribute to the open source Resin.io image hierarchy.

object offers interfaces for methods supported by the BLE protocol including connection, discovery, read and write methods.

3) agile-nodered: : The agile-nodered container is built from a NodeJS base image and does not depend on other containers. This container runs the Node-Red application inside the container and exposes an endpoint to access the application. The Node-Red running inside the container also offers access to multiple other custom Node-Red nodes developed for the AGILE gateway which includes recommendation, cloud integration and device interfacing nodes.

B. Tests

The following tests are conducted to address two key issues for gateway devices mentioned in the state of the art. First, we study the sizes of the selected images as a function of in-container image optimization techniques, assessing their effectiveness. Second, we evaluate cross-container optimization by looking at the size and consequently download time of the AGILE stack. Time measurements are repeated ten times and average values are presented.

TABLE I
IMAGE SIZES OBTAINED BY IN-CONTAINER OPTIMIZATION

Image	Baseline	Multi-stage	Squashed
agile-ble	801MB	372MB	361MB
agile-nodered	613MB	294MB	291MB
agile-core	789MB	348MB	342MB

Table I shows the effect of in-container optimization techniques on images of the three selected components. Note that image sizes contain base layers, language runtimes for different languages (Java or JavaScript) and dependencies, hence the relatively large image sizes. Multi-stage build optimization reduces image sizes to as low as 45%, in case of agile-core, of the baseline. Squashing, if applied on top of the already optimized multi-stage image, improves image size only marginally.

On the contrary, as we show next, squashing can be counterproductive for updates. Fig. 5 shows the download times for images built using in-container optimization. Although in reality we have introduced various optimizations gradually as the stack and its components evolved in the recent years, for this experiment we have regenerated each version of the components in a baseline, a multi-stage, and a squashed version. First, the generated images are pushed to repositories on Docker Hub with incremental tags for changing versions. These images are then iteratively downloaded in the following manners: (i) *absolute*, where previous versions of an image are removed before download, and (ii) *incremental*, emulating software update, where previous image versions are kept to download just the updated layers.

The first column of figure 5 shows the absolute download times of a specific version of agile-core, considering the three aforementioned techniques. Download times are almost proportional to image sizes, although it is interesting to
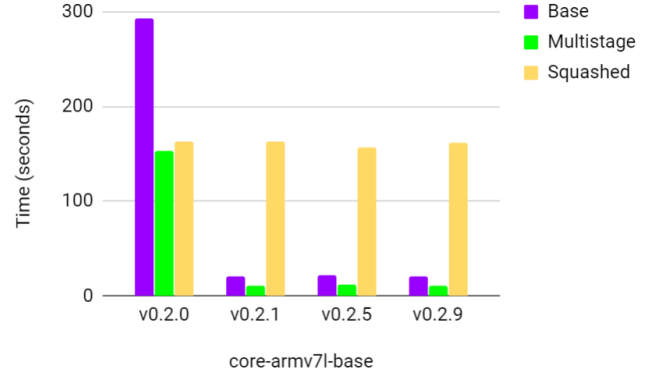

Fig. 5. Download times for in-container optimization techniques
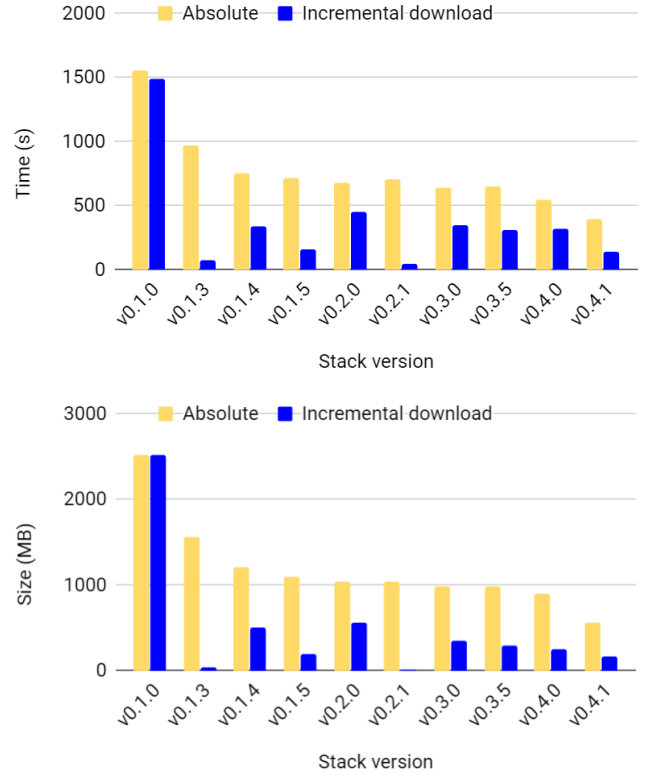

Fig. 6. Download times and data amounts for cross-container optimization techniques

note that the squashed image, even if smaller, downloads slightly slower than the multi-stage one. This is due to the way image download and decompression is parallelized in Docker. Subsequent columns show incremental download performance, where multi-stage, even baseline outperforms squashing. In fact, squashing forces the whole image content to be downloaded again, while multi-stage allows small changes to transform into the update of only a few small layers.

Finally, we consider multiple containers on the same gateway, i.e. the deployment of the AGILE stack. To simplify

discussion and avoid distortion from functionalities added during the evolution of the stack, we restrict deployment to the subset of images discussed before (agile-core, agile-ble, agile-nodered). Fig. 6 shows both the download time and the data amount as the stack evolved. In the initial versions, neither the base images were chosen considering common layers, nor in-container optimizations were used. Hence, the download amount was similar to the sum of baseline image sizes in Table I. The difference is due to the way data was obtained: in this experiment we use the original images forming a given version of the stack, while Table I contains the size of the newest version of each component regenerated using a given optimization technique. Incremental times and size also shows large values for v0.1.0, but only because this is the first stack deployed on a clean system. In the subsequent versions, absolute download times are reduced due to both cross-container and in-container optimizations. More specifically, multi-stage was introduced in v0.1.3 for agile-core and agile-ble, while only in v0.4.1 for agile-nodered. Reduction in the overall size can clearly be seen in these cases. The use of the base image hierarchy, instead, was introduced in v0.1.4 and v0.2.0, respectively. Since this involved a change in the base images, incremental downloads increased for the specific release, but the overall effect to stack size is positive.

VI. Conclusion

This paper summarizes the current state of the art on containerization techniques and suitability of containerization in the context of IoT systems. The existing gap in the current literature in terms of multi-container deployments on edge layer devices is addressed by illustrating a microservice architecture based container deployment and defining optimization techniques to improve the gateway performance.

The cross-container optimization technique proposed facilitates the reduction in build times for images and removes the redundancy among base images used in the stack. On the other hand, in-container optimization reduces the overall size of the AGILE stack and thus reduces download times for the images as well.

In future work, we will perform further measurements on multiple devices and device architectures to improve our proposed optimization techniques. We would also investigate optimization of pushing delta updates to the devices while minimizing the build and download times using the current study as a reference.

Acknowledgment

References

[1] "Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions)," https://www.statista.com/statistics/471264/iot-number-of-connected-dev ices-worldwide/.

[2] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog Computing and Its Role in the Internet of Things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13–16.

[3] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt, "Cloudlets: Bringing the cloud to the mobile user," in *Proceedings of the third ACM workshop on Mobile cloud computing and services*. ACM, 2012, pp. 29–36.

[4] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee, "A container-based edge cloud PaaS architecture based on Raspberry Pi clusters," in *Future Internet of Things and Cloud Workshops (FiCloudW), IEEE International Conference on*. IEEE, 2016, pp. 117–124.

[5] H. Chang, A. Hari, S. Mukherjee, and T. Lakshman, "Bringing the cloud to the edge," in *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*. IEEE, 2014, pp. 346–351.

[6] G. Senthilkumar, K. Gopalakrishnan, and V. S. Kumar, "Embedded image capturing system using raspberry pi system," *International Journal of Emerging Trends & Technology in Computer Science*, vol. 3, no. 2, pp. 213–215, 2014.

[7] R. Morabito, R. Petrolo, V. Loscrí, and N. Mitton, "Enabling a lightweight edge gateway-as-a-service for the internet of things," in *Network of the Future (NOF), 2016 7th International Conference on the*. IEEE, 2016, pp. 1–5.

[8] N. Bizanis and F. A. Kuipers, "SDN and virtualization solutions for the Internet of Things: A survey," *IEEE Access*, vol. 4, pp. 5591–5606, 2016.

[9] A. El-Mougy, M. Ibnkahla, and L. Hegazy, "Software-defined wireless network architectures for the Internet-of-Things," in *Local Computer Networks Conference Workshops (LCN Workshops), 2015 IEEE 40th*. IEEE, 2015, pp. 804–811.

[10] C. Pahl and B. Lee, "Containers and clusters for edge cloud architectures–a technology review," in *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*. IEEE, 2015, pp. 379–386.

[11] C. Pahl, "Containerization and the PaaS Cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.

[12] F. Ramalho and A. Neto, "Virtualization at the network edge: A performance comparison," in *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2016 IEEE 17th International Symposium on A*. IEEE, 2016, pp. 1–6.

[13] R. Morabito, I. Farris, A. Iera, and T. Taleb, "Evaluating Performance of Containerized IoT Services for Clustered Devices at the Network Edge," *IEEE Internet of Things Journal*, 2017.

[14] A. Krylovskiy, "Internet of Things gateways meet linux containers: Performance evaluation and discussion," in *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*. IEEE, 2015, pp. 222–227.

[15] J. Rufino, M. Alam, J. Ferreira, A. Rehman, and K. F. Tsang, "Orchestration of containerized microservices for IIoT using Docker," in *Industrial Technology (ICIT), 2017 IEEE International Conference on*. IEEE, 2017, pp. 1532–1536.

[16] R. Chesov, V. Solovyev, . Khlamov, and . Prokofyev, "Containerized cloud based technology for smart cities applications," *Journal of Fundamental and Applied Sciences*, vol. 8, no. 3S, pp. 2638–2646, 2016.

[17] M. Kovatsch, M. Lanter, and S. Duquennoy, "Actinium: A RESTful runtime container for scriptable Internet of Things applications," in *Internet of Things (IOT), 2012 3rd International Conference on the*. IEEE, 2012, pp. 135–142.