

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/301670224>

Design of an IoT Cloud System for Container Virtualization on Smart Objects

Chapter · January 2016

DOI: 10.1007/978-3-319-33313-7_3

CITATION

1

READS

45

5 authors, including:



[Maria Fazio](#)

Università degli Studi di Messina

89 PUBLICATIONS 356 CITATIONS

[SEE PROFILE](#)



[Antonio Celesti](#)

Università degli Studi di Messina

96 PUBLICATIONS 557 CITATIONS

[SEE PROFILE](#)



[Massimo Villari](#)

Università degli Studi di Messina

151 PUBLICATIONS 1,061 CITATIONS

[SEE PROFILE](#)



[Antonio Puliafito](#)

Università degli Studi di Messina

395 PUBLICATIONS 4,019 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



Stack4Things [View project](#)



CloudforEurope [View project](#)

All content following this page was uploaded by [Davide Mulfari](#) on 10 July 2016.

The user has requested enhancement of the downloaded file.

Design of an IoT Cloud System for Container Virtualization on Smart Objects

Davide Mulfari, Maria Fazio, Antonio Celesti^(✉), Massimo Villari,
and Antonio Puliafito

DICIEAMA, University of Messina,
Contrada Di Dio, 98166 Sant'Agata, Messina, Italy
{dmulfari,mfazio,acelesti,mvillari,apuliafito}@unime.it
<http://mdslab.unime.it>

Abstract. Nowadays, container virtualization is a lightweight alternative to the hypervisor-based approach. Recent improvements in Linux kernel allow to execute containers on smart objects, that are, single board computers running Linux-based operating systems. By considering several IoT application scenarios, it is crucial to rely on cloud services able to deploy and customize pieces of software running on target smart objects. To achieve this goal, in this paper, we focus our attention on a Message Oriented Middleware for Cloud (MON4C), a system designed to compose cloud facilities by means of a flexible federation-enabled communication system. Its objective is to provide Internet of Things (IoT) services in a complex smart environment, such as a smart city, where smart objects interact each others and with the cloud infrastructure. More specifically, we discuss how MOM4C can be extended to support container virtualization on Linux embedded devices in order to easily deploy IoT applications in a flexible fashion and we present the design of related software modules.

Keywords: Cloud computing · Container based virtualization · IoT · Embedded systems · Linux

1 Introduction

Resource virtualization is one of the key concepts in cloud computing and it refers to the act of creating a virtual (rather than physical) version of “something”, including but not limited to a virtual computer hardware platform, operating system (OS), storage device, or computer network resources. Virtualization consists of using an intermediate software layer on top of an underlying system in order to provide abstractions of multiple virtual resources. The latter software components are known as Virtual Machines (VMs) and they can be viewed as isolated execution contexts. Nowadays, several virtualization techniques are available. One of the most popular is the hypervisor-based virtualization, which requires a Virtual Machine Monitor (VMM) software module on top of a “host” OS that provides a full abstraction of VMs. In this case, each VM has its own

“guest” OS that is completely isolated from others. This enables us to execute multiple different OSs on a single physical host OS. Examples of such software solutions include: Xen, VMware, Oracle VirtualBox and KVM.

Recently, a lightweight alternative technology is the container-based virtualization, also known as OS level virtualization. This kind of virtualization partitions the physical machines resources, creating multiple isolated user-space instances [1]. Figure 1 depicts the key difference between the aforementioned virtualization technologies. While the hypervisor based virtualization provides a full abstraction for guest OS(s) (one per VM), the container based virtualization works at the OS level, providing abstractions directly for the “guest” processes. In essence, hypervisor solutions work at the hardware abstraction level and containers operate at the system call layer.

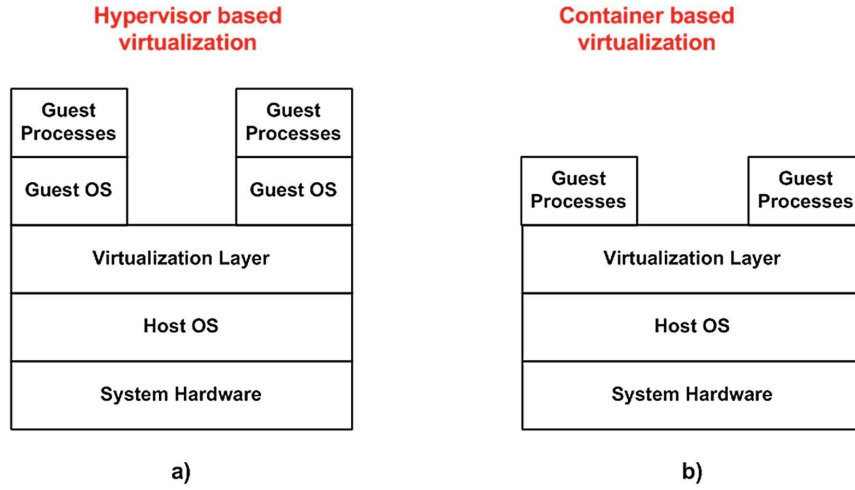


Fig. 1. Difference between (a) hypervisor and (b) container based virtualization.

As motivated in [1], all the containers share a single kernel; so the container based virtualization is supposed to have a weaker isolation when compared to hypervisor based virtualization. However, from the point of view of the users, each container looks exactly like a stand-alone OS. Additionally, by considering a cloud computing scenario, developers can deploy higher densities with containers than with VMs on the same physical host. Another advantage of containers over VMs is that starting and shutting down a container is much faster than starting and shutting down a traditional VM equipped with a guest OS.

Recent technological developments have allowed container-based virtualization technology to support Single Board Computer (SBC) devices equipped with a modern Linux kernel supporting a suitable virtualization layer. In these scenarios, container based software seems to be an interesting approach to deploy and to customize software applications running on a SBC. More specifically,

in the present paper, we focus on Internet of Things (IoT) application scenarios and we define “smart object” a SBC embedded device equipped with a Linux based OS that runs specialized pieces of software in order to grab and process data from external sensors. We intend to distribute multiple smart objects in a complex environment, such as a smart city, where it is crucial to rely on a cloud service able to deploy and to customize pieces of software running on target smart objects.

In order to pursue our goals, we consider a Message Oriented Middleware for Cloud (MOM4C) [2], a piece of middleware able to arrange customizable Cloud facilities by means of a flexible federation-enabled communication system. The considered middleware has very innovative features, that make efficient, scalable and versatile the Cloud service provisioning. In addition, MOM4C enables the development of distributed services over an asynchronous instant-messaging architecture, which can be used for intra/inter-domain communications. In Cloud computing environments, MOM4C allows to compose Cloud facilities according to client requirements. MOM4C has been designed to act as a “planetary system model”, where the central star includes the core, i.e., all the basic communication functionalities of the piece of middleware and the planets are the Cloud utilities that can be used. Such a service provisioning model guarantees high scalability and customization of the required service. In addition, besides the basic communication functionalities, the core includes security mechanisms for guaranteeing secure data exchange.

More specifically the main contribution of this paper is to discuss how MOM4C can support Linux based smart objects in order to allow software architects to dynamically deploy pieces of software on them by means of container-based virtualization techniques. The proposed hardware/software infrastructure uses Docker as containers engine platform; within the last year, such a software has emerged as a standard runtime, image format, and “build system” for containers on several distributed Linux environments.

The remainder of this paper is organized as follows. In Sect. 2, we discuss related works. In Sect. 3, we provide an overview about the container based virtualization for Linux environments and IoT devices. In Sect. 4, we discuss how we extended MOM4C in order to support the container virtualization in IoT devices. A system prototype with implementation highlights is discussed in Sect. 5. Finally, Sect. 6 concludes the paper.

2 Related Work

Nowadays, containers represents an interesting alternative to VMs in the Cloud scenarios [3]. Although the concepts underlying containers such as namespaces are very mature, only recently containers have been adopted and standardized in mainstream OS(s), leading to a renaissance in the use of containers to provide isolation and resource control. Linux is the preferred OS for cloud environments due to its zero price, large ecosystem, good hardware support, good performance, and reliability. The kernel namespace feature needed to implement containers in

Linux has only become mature in the last few years since it was first discussed in 2006 [4]. Several articles have focused on container based virtualization technologies by considering Cloud computing scenarios. Docker [5] is a lightweight virtualization based on Linux Containers (LXC) that can completely encapsulate an application and its dependencies within a virtual container. In [6], the authors discuss the design and the implementation of Cloud system based on Docker, especially intended for a Platform as a Service (PaaS). As motivated in [7], Docker has been deployed within a platform for bioinformatics computing that exploits advanced Cloud services. Authors investigate the security level of Docker by considering two main areas: (1) the internal security of Docker, and (2) how Docker interacts with the security features of the Linux kernel, such as SELinux and AppArmor, in order to harden the host system. The proposed analysis shows that Docker provides a high level of isolation and resource limiting for its containers using namespaces, cgroups, and its copy-on-write file system, even with the default configuration. It also supports several kernel security features, which help in hardening the security of the host [8].

Nowadays, Cloud computing has emerged in different application fields including energy efficiency [9], storage [10], Assistive Technology [11], dataweb [12] and so on. Several manuscripts deal with the development of Cloud pieces of middleware, addressing specific issues and exploiting different technologies. To support application execution in the Cloud, in [13], authors present Cloud-Scale. It is a piece of middleware for building Cloud applications like regular Java programs and easily deploy them into IaaS Clouds. It implements a declarative deployment model, in which application developers specify the scaling requirements and policies of their applications using the Aspect-Oriented Programming (AOP) model. A different approach is proposed in [14], which presents a low latency fault-tolerance piece of middleware for supporting distributed applications deployment within a Cloud environment. It is based on the leader/follower replication approach for maintaining strong replica consistency of the replica states. If a fault occurs, the reconfiguration/recovery mechanisms implemented in the middleware ensure that a backup replica obtains all the information it needs to reproduce the actions of the application. The piece of middleware presented in [15] has been designed aiming at mission assurance for critical Cloud applications across hybrid Clouds. It is centered on policy-based event monitoring and dynamic reactions to guarantee the accomplishment of “end-to-end” and “cross-layered” security, dependability and timeliness. In [16], the authors present a piece of middleware for enabling “media-centered” cooperation among home networks. It allows users to join their home equipments through a Cloud, providing a new content distribution model that simplifies the discovery, classification, and access to commercial contents within a home networks. Mathias and Baude [17] focus on the integration of different types of computational environments. In fact, they propose a lightweight component-based piece of middleware intended to simplify the transition from clusters, to Grids and Clouds and/or a mixture of them. The key points of such a system are a modular infrastructure, that can adapt its behaviour to the running environment, and application connectivity requirements. The problem of integrating multi-tenancy into the Cloud

is addressed in [18]. The authors propose a Cloud architecture for achieving multi-tenancy at the Service Oriented Architecture (SOA) level by virtualizing the middleware servers running SOA artifacts and allowing a single instance to be securely shared between tenants or different customers. The key idea of the work is that the combination between virtualization, elasticity and multi-tenancy makes it possible an optimal usage of data center resources (i.e., CPU, memory, and network). A piece of middleware designed for monitoring Cloud resources is proposed in [19]. The presented architecture is based on a scalable data-centric publish/subscribe paradigm to disseminate data in multi-tenant Cloud scenarios. Furthermore, it allows to customize both granularity and frequency of received monitored data according to specific service and tenant requirements. The work proposed in [20] aims to support mobile applications with processing power and storage space, moving resource-intensive activities into the Cloud. It abstracts the API of multiple Cloud vendors, thus providing a unique JSON-based interface that responds according to the REST-based Cloud services. The current framework considers the APIs from Amazon EC2, S3, Google and some open source Cloud projects like Eucalyptus. In [21], the authors present a piece of middleware to support fast system implementation and ICT cost reduction by making use of private Clouds. The system includes application servers that run a Java Runtime Environment (JRE) and additional modules for service management and information integration, designed according to a Service Oriented Architecture (SOA).

3 Container Virtualisation for Linux Environments

Container-based virtualization can be considered as an approach in which the virtualization layer runs within an application on top of the OS. In this approach, the OS's kernel runs on the hardware node with several isolated guest virtual environments called containers. In this Section, we describe the pieces of software needed to support the container virtualization by considering a generic Linux system. Looking at Fig. 2, a Linux host OS is normally deployed on the top of system hardware layer (including CPU, RAM, peripherals, etc.) and its kernel needs to work with a suitable virtualization layer. In this way, the OS-level virtualization does not require an additional hypervisor layer since the virtualization capabilities are part of the host OS. This technique allows to virtualize applications on top of the host OS itself. Therefore, the overhead produced by the hypervisor mediation is eliminated enabling near native performances. In addition, the host kernel provides process isolation and performs resource management. This means that even though all the containers are running under the same kernel, each container is a virtual environment that has its own file system, processes, memory, devices, etc. There are different host applications located on top of the Linux kernel. In particular, we focus our attention on the containers engine component that automates the deployment of any application as a lightweight, portable, self-sufficient container that will run virtually anywhere.

By considering several IoT services and applications, in this paper we mainly focus our attention on considering Linux-based Single Board Computers (SBCs)

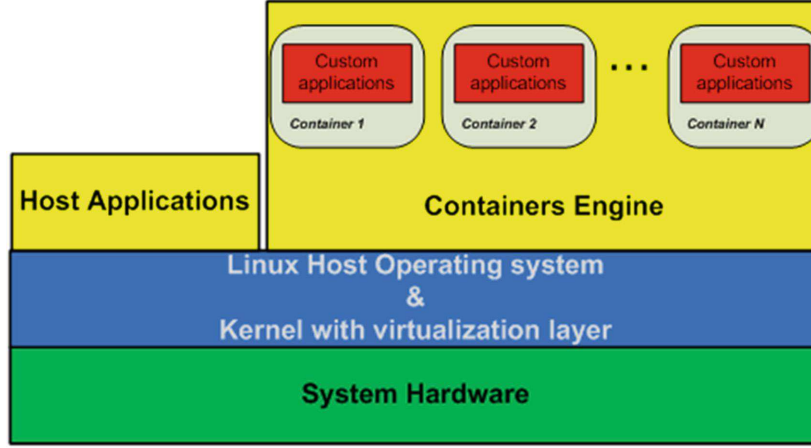


Fig. 2. Container-based virtualization.

that include several General Purpose Input Output (GPIO) extensions allowing our IoT device to interact with many different external sensors and actuators. More specifically, we consider the software structure shown in Fig. 3. Starting from bottom, our system hardware consists of a Raspberry Pi B+ model [22]. While the latter board is, in essence, a very inexpensive Linux computer, there are a few things that distinguish it from a general purpose machine. One of the main differences is that the Raspberry Pi can be directly used in electronics projects because it includes GPIO pins on the board. These GPIO hardware extensions can be accessed for controlling hardware such as LEDs, motors, and relays, which are all examples of outputs. As for inputs, the used Raspberry Pi can read the status of buttons, switches, and dials, or it can read data coming from sensors like temperature, light, motion, or proximity [23]. Our Raspberry Pi board is equipped with the Raspbian distribution that is the most popular OS for the considered piece of hardware; it also includes customizations that are designed to make the Raspberry Pi easier to use and includes many different software packages out of the box. In particular, in this paper, we are considering Raspbian 3.18.8 Linux kernel version that comes with the LXC extensions. As discussed in [24], this extension represents container-based OS virtualization and one of its major benefits is that it can run multiple Linux instances on a single physical host. With reference to the Fig. 3, host applications are deployed on the top of Raspbian OS and Linux kernel.

We consider the Docker Platform as container engine, which is an open platform for developers and system administrators to build, ship, and run distributed applications. Being the Docker Engine, a lightweight portable, runtime, and packaging tool it represents a valuable solution to implement a cloud service for sharing applications and automating workflows. In fact, Docker Hub enables apps to be quickly assembled from components and fulfil the gap between development

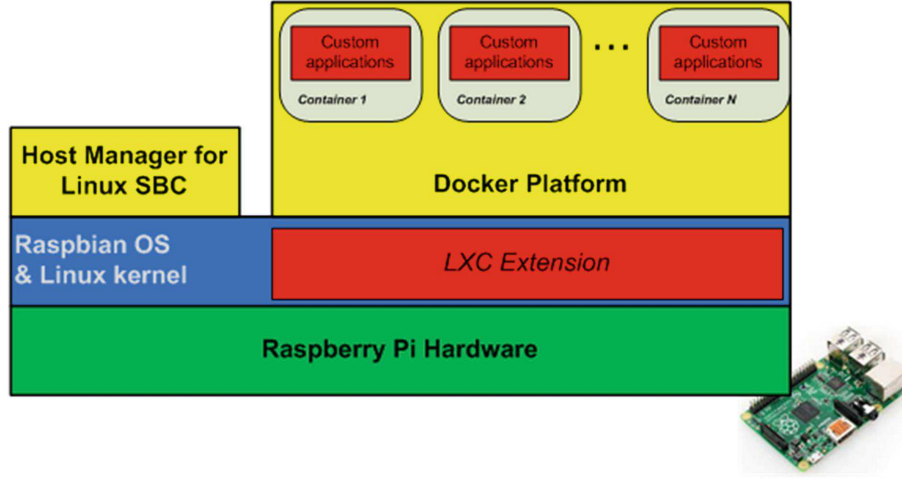


Fig. 3. Software architecture for container-based virtualization deployed on a Raspberry Pi board.

and production environments. As a result, cloud providers can fast ship and run the same application and service on VMs and IoT devices. Docker is also an open-source implementation of the deployment engine which powers dotCloud, a popular Platform-as-a-Service (PaaS). It directly benefits from the experience accumulated over several years of large-scale operation and support to hundreds of thousands of applications and databases. It relies on a different sandboxing method known as containerization. Most modern OS kernels now support the primitives necessary for containerization, including Linux with openvz, vserver and recently LXC containment features. Through a powerful API and simple tools, it lets Linux users to create and manage system or application containers.

In our context, at the same Docker's level, we can consider several services and applications allowing IoT devices to interact with the cloud, as it will be discussed in the next Section.

4 MOM4C Extension for IoT and Container Support

The MOM4Cloud architecture and its design choices have been already discussed in [2]. In this paper, our major contribution is to extend the piece of middleware's functionalities in order to support the management of container based environment on SBCs Linux devices, also known as smart objects or IoT devices. We can consider the container-based virtualization as a method for making available services and applications on IoT systems. For these reasons, our reference scenario includes a set of physical hardware resources i.e., embedded systems, where several types of container images are dynamically loaded according to their workload and other parameters. In this way, we aim to provide services

into a complex smart environment, like a smart city where the objects can also interact with each others. Such environments are often pictured as constellations of instruments across many scales that are connected through multiple networks which provide continuous data regarding the movements of people and materials in terms of the flow of decisions about the physical and social form of the city. Cities however can only be smart if there are intelligent functions that are able to integrate and synthesise this data to some purposes, with the aim of improving the efficiency, equity, sustainability and quality of life in cities [25]. From a technical point of view, our cloud system has to guarantee the following basic operations:

- Monitoring the container environments behaviour and performance, in terms of CPU, memory and storage usage.
- Managing the container images, providing functions to destroy, commit, migrate and set network parameters.
- Managing the container resources, i.e., images discovery, uploading and downloading via a FTP repository.

Figure 4 summarizes our reference scenario and it shows a cluster of two kinds of nodes. Blade servers execute a cluster level management module, called Cluster Manager (CM), while each SBC piece of hardware supports both a host level management module, the Host Manager (HM), and a specialized Containers Engine component, like Docker. All these entities interact exchanging information by means of the communication system based on the Extensible Messaging and Presence Protocol (XMPP). The dataset necessary to enable the middleware functioning is stored within a specific Database deployed in a distributed fashion such as MongoDB; in addition, the depicted software infrastructure is equipped with a container repository that works with the FTP protocol. More

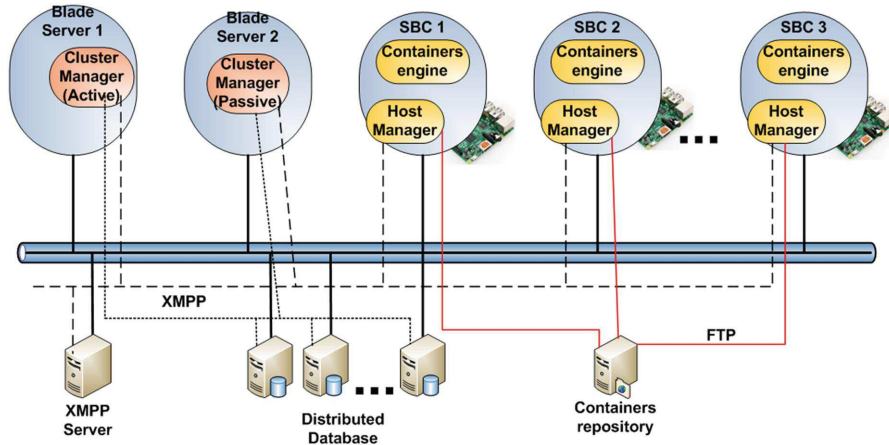


Fig. 4. Reference scenario. MOM4C architecture extended for IoT devices and container support.

specifically, core components of our infrastructure can be split into two logical categories: the software agents (typical of the architecture itself) and the tools they exploit. To the former set belong both the Host Manager and the Cluster Manager: The CM consists in as an interface between administrators (software entities, which can exploit the cloud services) and the HM agents. A CM receives commands from administrators, performs operations on the HM agents (or on the database) and finally sends information to administrators. It also performs the management of container images and the monitoring of the overall state of the cluster. According to our idea, at least one CM has to be deployed on each cluster but, in order to ensure higher fault tolerance, many of them should exist. A master CM will exist in active state while the other ones will remain in a monitoring state, although admin messages are listened whatever operation is performed. The HM performs the operations needed to monitor the physical resources and the instantiated container images: it interacts with the containers engine, the SBC's operating system and the FTP repository where the images are stored.

4.1 Architecture Overview

In this part, we focus our attention on the design of CM and HM software modules. Regarding CM, Fig. 5 highlights its functional blocks and their organization: the main components are described as follows:

- Database Manager: such a component interacts with the database employed to store information needed to the cluster handling. Database Manager must maintain the data strictly related to the cluster state.
- Performance Estimator: it analyses the performance dataset collected from physical assets (physical IoT devices), in order to provide a trend of performance estimation.
- Image Manager: it manages both registrations and uploads within the Cluster Storage System of the Docker images.
- Storage Manager: it manages the internal cluster distributed file system.

As previously mentioned, HM modules are deployed on each SBC piece of hardware. The HM's architecture is shown in Fig. 6. Its main components include:

- Monitor: it provides resource usage monitoring for each SBC. The pieces of information are organized and made available to the HM coordinator.
- Container engine interface: it is the middleware back-end of the container engine running on the SBC, for example the Docker Platform.
- Image Manager: it supplies to the container engine interface the needed container images by means of the FTP protocol.
- Network Manager: it gathers information about the host network state and it manages host network (at OS level) according to the guidelines provided by the HM Coordinator.

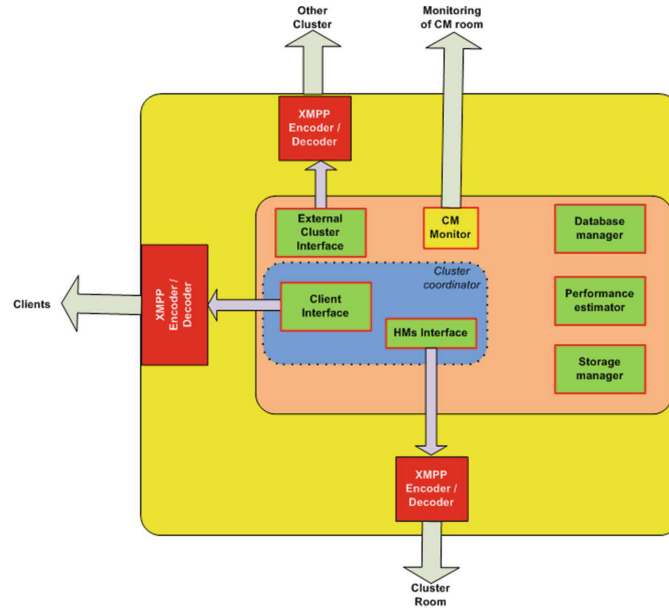


Fig. 5. Cluster Manager architecture.

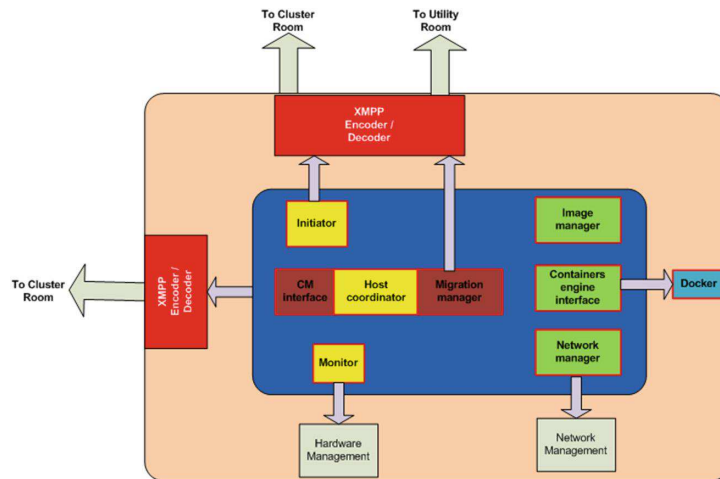


Fig. 6. Host Manager architecture.

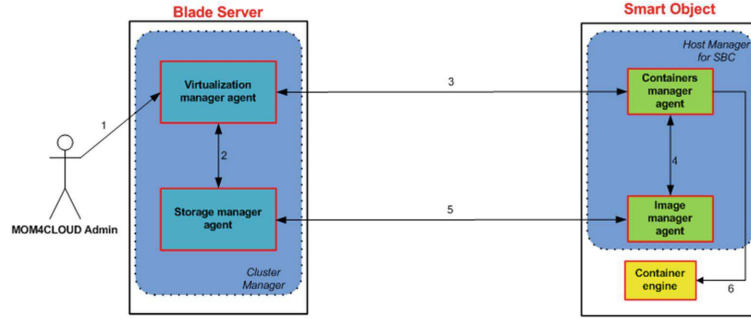


Fig. 7. Steps needed to load a container image on a smart object.

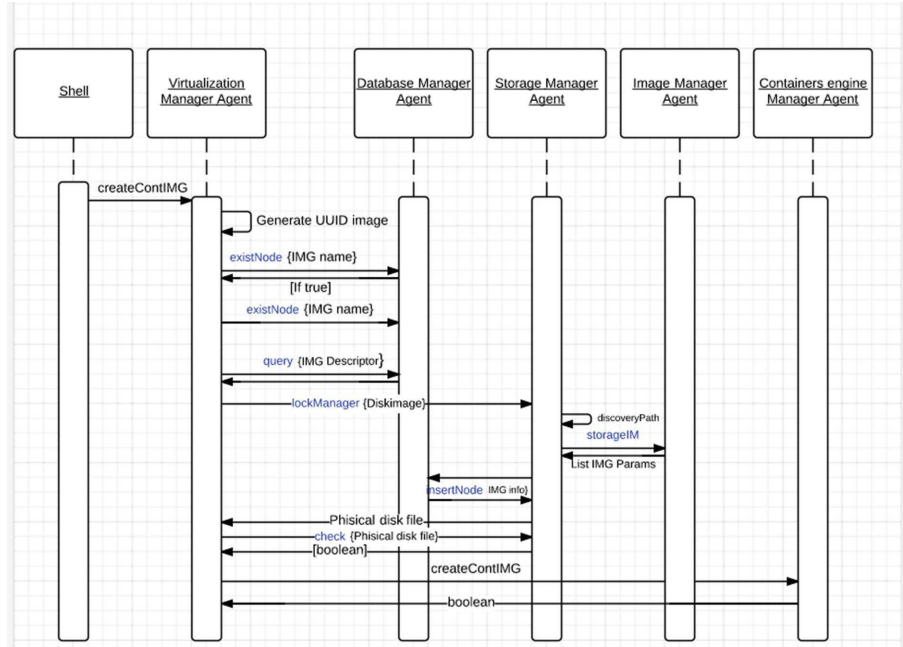
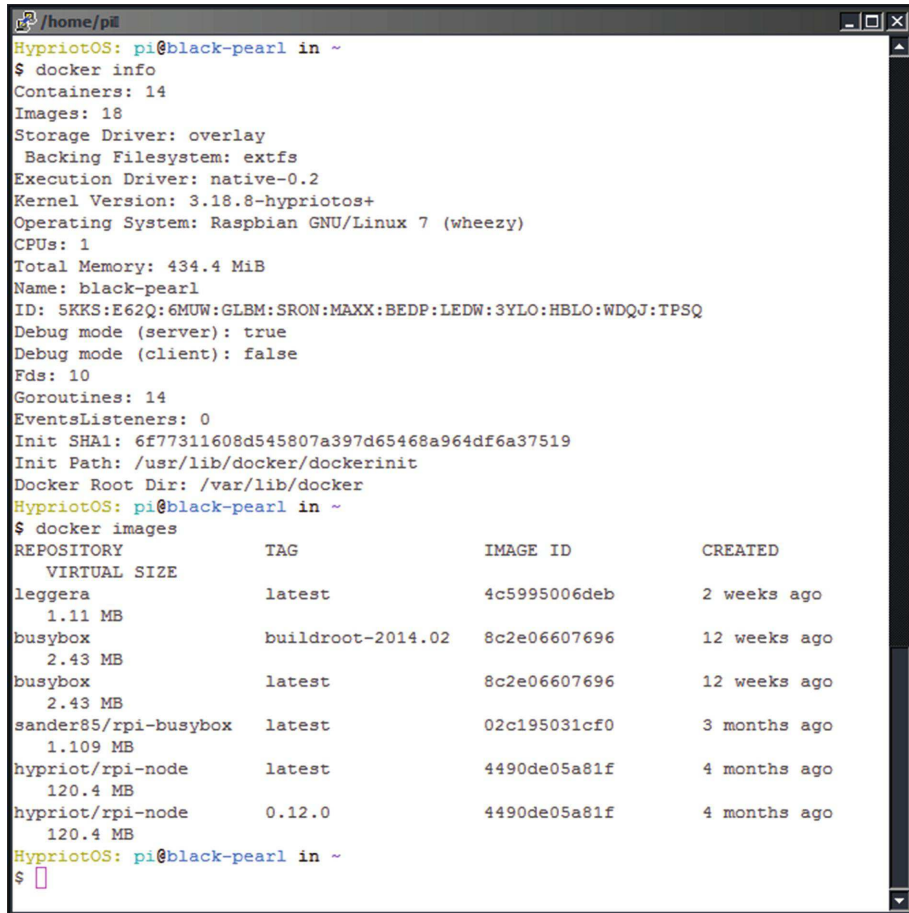


Fig. 8. Sequence diagram that shows the designed processes.

4.2 Technical Details

By looking at the Fig. 7, the dynamic load management of container images on a smart object requires six separate steps. Starting from left, the MOM4C administrator is a person who interacts with our piece of middleware by using a computer console program. The shell program sends user requests to the active CM running on a specialized blade server. More specifically, at the first step the Virtualization manager agent works on the received commands (step 1) and forwards the query to the Storage manager agent (step 2). The latter software

module is responsible for managing the FTP repository that stores the required container image. If such an operation concludes successfully, an ACK message is sent to the Virtualization manager agent (step 3). After that, the active CM queries the Host Manager agent that executes on the smart object (step 4). The Virtualization manager agent sends suitable requests to the Containers manager agent to invoke the download of the required container images. Then, the SBC system connects to the Storage manager agent (step 5) in order to retrieve the needed data and information. Finally (step 6), the Containers manager agent calls the container engine (e.g., Docker) in a suitable way. In Fig. 8, we present the sequence diagram of the described process.



```

/home/pi
HypriotOS: pi@black-pearl in ~
$ docker info
Containers: 14
Images: 18
Storage Driver: overlay
  Backing Filesystem: extfs
Execution Driver: native-0.2
Kernel Version: 3.18.8-hypriotos+
Operating System: Raspbian GNU/Linux 7 (wheezy)
CPUs: 1
Total Memory: 434.4 MiB
Name: black-pearl
ID: 5KKS:E62Q:6MUW:GLBM:SRON:MAXX:BEDP:LEDW:3YLO:HBLO:WDQJ:TPSQ
Debug mode (server): true
Debug mode (client): false
Fds: 10
Goroutines: 14
EventsListeners: 0
Init SHA1: 6f77311608d545807a397d65468a964df6a37519
Init Path: /usr/lib/docker/dockerinit
Docker Root Dir: /var/lib/docker
HypriotOS: pi@black-pearl in ~
$ docker images

```

REPOSITORY	TAG	IMAGE ID	CREATED
leggera	latest	4c5995006deb	2 weeks ago
busybox	buildroot-2014.02	8c2e06607696	12 weeks ago
busybox	latest	8c2e06607696	12 weeks ago
sander85/rpi-busybox	latest	02c195031cf0	3 months ago
hypriot/rpi-node	latest	4490de05a81f	4 months ago
hypriot/rpi-node	0.12.0	4490de05a81f	4 months ago

```

HypriotOS: pi@black-pearl in ~
$

```

Fig. 9. Docker shell commands running on our embedded device.

5 System Prototype

A HM prototype was implemented on a Raspberry Pi B+ embedded system by using the Python high-level programming language. Our SBC device executes a custom Raspbian OS image with Docker 1.5 version, which adds support for IPv6, read-only containers and advanced statistics. Considering this environment, we relied on a standard Docker's command-line console as shown in Fig. 9 in order to monitor containers. In particular, our HM consists of a specialized XMPP client that accepts and processes container management messages coming from a CM deployed in blade server and that forward them to the container manager interface. This kind of communication has been managed through the XMPPY libraries. In particular, the container manager interface interacts with the underlying Docker engine. In this way, the following basic operations can be performed on container images:

- Download an image from a FTP repository available on a blade server;
- Upload an image to a FTP repository available on a blade server;
- Start, stop, delete a given container image available on the embedded device.

6 Conclusion

Nowadays, container-based virtualization is a kind of OS-level virtualization that allows us to run multiple instances of the same OS user workspace sharing the kernel of the host OS. Technological developments have allowed such a technology to support SBCs, i.e., smart IoT devices equipped with a modern Linux kernel supporting a suitable virtualization layer. Considering multiple application scenarios, it is important to rely on cloud services able to deploy and to customize pieces of software running on target smart objects. To achieve such a goal, in this paper, we focused our attention on MOM4C, a flexible solution able to arrange customizable cloud facilities by means of a federation-enabled communication system. In this way, we aim to provide services into a complex smart environment, like a smart city, where objects interact each others and with the cloud. Therefore, we have discussed how MOM4C can be extended to support container-based virtualization on Linux embedded IoT devices. More specifically, we designed the two main software modules constituting our software infrastructure.

Since our prototype implementation is still at an early stage, we are already working to further extend the system functionalities according to our reference architecture. In future work, we plan to perform a set of experiments in order to evaluate the behaviour of the piece middleware and its performance when managing multiple containers on the same IoT device.

Acknowledgments. The research leading to the results presented in this paper has received funding from the Project “Design and Implementation of a Community Cloud Platform aimed at SaaS services for on-demand Assistive Technology”.

References

1. Xavier, M., Neves, M., Rossi, F., Ferreto, T., Lange, T., De Rose, C.: Performance evaluation of container-based virtualization for high performance computing environments. In: 2013 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 233–240 (2013)
2. Fazio, M., Celesti, A., Villari, M.: Design of a message-oriented middleware for cooperating clouds. In: Canal, C., Villari, M. (eds.) ESOC 2013. CCIS, vol. 393, pp. 25–36. Springer, Heidelberg (2013)
3. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 171–172 (2015)
4. Biederman, E.W., Networx, L.: Multiple instances of the global linux namespaces. In: Proceedings of the Linux Symposium, Citeseer (2006)
5. Bernstein, D.: Containers and cloud: from LXC to docker to kubernetes. *IEEE Cloud Comput.* **1**, 81–84 (2014)
6. Liu, D., Zhao, L.: The research and implementation of cloud computing platform based on docker. In: 2014 11th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP), pp. 475–478 (2014)
7. Kacamarga, M.F., Pardamean, B., Wijaya, H.: Lightweight virtualization in cloud computing for research. In: Intan, R., Chi, C.-H., Palit, H.N., Santoso, L.W. (eds.) *Intelligence in the Era of Big Data*. CCIS, vol. 516, pp. 439–445. Springer, Heidelberg (2015)
8. Bui, T.: Analysis of docker security. arXiv preprint [arXiv:1501.02967](https://arxiv.org/abs/1501.02967) (2015)
9. Giacobbe, M., Celesti, A., Fazio, M., Villari, M., Puliafito, A.: Towards energy management in cloud federation: a survey in the perspective of future sustainable and cost-saving strategies. *Comput. Netw.* **91**, 438–452 (2015)
10. Celesti, A., Fazio, M., Villari, M., Puliafito, A.: Adding long-term availability, obfuscation, and encryption to multi-cloud storage systems. *J. Netw. Comput. Appl.* **59**, 208–218 (2016)
11. Mulfari, D., Celesti, A., Villari, M.: A computer system architecture providing a user-friendly man machine interface for accessing assistive technology in cloud computing. *J. Syst. Softw.* **100**, 129–138 (2015)
12. Celesti, A., Tusa, F., Villari, M., Puliafito, A.: How the dataweb can support cloud federation: service representation and secure data exchange. In: 2012 Second Symposium on Network Cloud Computing and Applications (NCCA), pp. 73–79 (2012)
13. Leitner, P., Satzger, B., Hummer, W., Inzinger, C., Dustdar, S.: Cloudscale: a novel middleware for building transparently scaling cloud applications. In: SAC 2012, pp. 434–440 (2012)
14. Wenbing, Z., Melliar-Smith, P., Moser, L.: Fault tolerance middleware for cloud computing. In: IEEE 3rd CLOUD 2010, pp. 67–74 (2010)
15. Campbell, R., Montanari, M., Farivar, R.: A middleware for assured clouds. *J. Internet Serv. Appl.* **3**, 87–94 (2012)
16. Diaz-Sanchez, D., Almenarez, F., Marin, A., Proserpio, D., Cabarcos, P.A.: Media Cloud: an open cloud computing middleware for content management. *IEEE Trans. Consum. Electron.* **57**, 970–978 (2011)

17. Manias, E., Baude, F.: A component-based middleware for hybrid grid/cloud computing platforms. *Concurrency Comput. Pract. Exp.* **24**, 1461–1477 (2012)
18. Azeez, A., Perera, S., Gamage, D., Linton, R., Siriwardana, P., Leelaratne, D., Weerawarana, S., Fremantle, P.: Multi-tenant SOA middleware for cloud computing. In: *IEEE CLOUD 2010*, pp. 458–465 (2010)
19. Povedano-Molina, J., Lopez-Vega, J.M., Lopez-Soler, J.M., Corradi, A., Foschini, L.: Dargos: a highly adaptable and scalable monitoring architecture for multi-tenant clouds. *Future Gener. Comput. Syst.* **29**, 2041–2056 (2013)
20. Flores, H., Srirama, S.N.: Dynamic re-configuration of mobile cloud middleware based on traffic. In: *IEEE MASS 2012* (2012)
21. Nagakura, H., Sakurai, A.: Middleware for creating private clouds. *Fujitsu Sci. Tech. J. (FSTJ)* **47**, 263–269 (2011)
22. Maksimović, M., Vujović, V., Davidović, N., Milošević, V., Perišić, B.: Raspberry Pi as internet of things hardware: performances and constraints. *Des. Issues* **3**, 8 (2014)
23. Richardson, M., Wallace, S.: *Getting Started with Raspberry Pi*. O'Reilly Media, Inc., Sebastopol (2012)
24. Memari, N., Hashim, S.J.B., Samsudin, K.B.: Towards virtual honeynet based on LXC virtualization. In: *2014 IEEE Region 10 Symposium*, pp. 496–501 (2014)
25. Batty, M., Axhausen, K., Giannotti, F., Pozdnoukhov, A., Bazzani, A., Wachowicz, M., Ouzounis, G., Portugali, Y.: Smart cities of the future. *Eur. Phys. J.* **214**, 481–518 (2012)