Orchestration of Containerized Microservices for IIoT using Docker

João Rufino*, Muhammad Alam*, Joaquim Ferreira*,‡, Abdur Rehman†, Kim Fung Tsang**
* Instituto de Telecomunicações, Campus Universitário de Santiago, 3810-193 Aveiro, Portugal
‡ ESTGA - Universidade de Aveiro, 3754-909 Águeda, Portugal
†Department of Internetworking, Faculty of Engineering , Dalhousie University, Halifax, NS, Canada.
**City University of Hong Kong – HK.
{joao.rufino, alam, jjcf}@ua.pt, abdur.rehman@dal.ca, ee330015@cityu.edu.hk

***Abstract*—Industrial Internet of things (IIoTs) relies on different devices working together, gathering and sharing data using multiple communication protocols. This heterogeneity becomes a hindrance in the development of architectures that can support applications operating independently of the underlying protocols. Therefore in this paper, we proposed a modular and scalable architecture based on lightweight virtualization. The modularity provided by the proposed architecture combined with lightweight virtualization orchestration supplied by Docker simplifies management and enables distributed deployments. Availability and fault-tolerance characteristics are ensured by distributing the application logic across different devices where a single microservice or even device failure can have no effect on system performance. The proposed architecture is instantiated and tested on a simple time-dependent use case. The obtained results validates that the proposed architecture can be used to deploy services on demand at different architecture layers.**

***Index Terms*—Software Defined Network; Industrial Internet of Things; Wireless Communication;**

I. Introduction

Industry 4.0 was presented as the next high-tech initiative by the German Government and since then, various companies and research institutes have been working to contribute on this topic. Industry as always relied on Machine to Machine (M2M) systems where devices exchange information and perform actions without the manual assistance of human services. These devices called Cyber Physical Systems (CPS) have been evolving from identification technologies (like RFID tags) and sensors or actuators with limited functions to more complex systems.

The Internet of Things (IoT) has brought new cards to the table by presenting a layered architecture that separates application layer from the sensing layer. It worked as an enabler for internet connected devices to cooperate and achieve common goals [1]. The generation of large volumes of data, which have to be processed, stored and presented, lead to the concept of Fog Computing. Thus, in order to attain better overall performance and reduce latency, a mediation layer between cloud and end-devices, capable of hosting small applications, was introduced by Cisco [2].

In fact, scalability was an high motivator for the application of IoT to Industry. It became intuitive that using a layered architecture and leveraging a cloud computing platform could improve asset performance and efficiency by providing a central virtual infrastructure for monitoring and analytic tools to be deployed. This results in an end-to-end service-based system that can be managed by businesses and accessed by end-users on demand.

To achieve modularity, the micro-service architecture [3] has been explored. Micro-service is a relatively new term in software architecture patterns. Contrary to the modern monolithic architecture where an application self-contains all the components, micro-services present an approach to develop applications as a set of small independent services. Consequently, different parts of the same application can be deployed in their own processes, making it possible to decompose huge applications in single use cases or services with a specific objective. These different parts are centrally managed by a completely separate service. Since each microservice is relatively small, it makes easier to deploy new service versions and dynamically adapt to changes.

Container-based virtualization is not a new concept. In fact, Docker [4] defines a standard for a group of well-known Linux functions creating a lightweight virtualization technique. Moreover, Docker is an open platform for developers and system administrators to build, share, and run distributed applications. Docker containers have two different layers, a Read-Only called image and a Read/Write normally named container. An image can be composed of other images, for example, we can have a Linux based OS as a base image and install the required dependencies, this compilation creates our service image. The instantiation of an image starts a container. Therefore, it is possible to create the minimal environment to have a service running.

With the infrastructure and service model defined the next step is orchestration. Having a centralized control over containers which enables starting, migrating and terminating services is of uttermost importance. Moreover, orchestration allows service policies and permissions to be defined and ensured on the different architecture layers. Hence, orchestration can be used by both developers and operations for an agile service development, integration and implementation, to improve security and even to schedule updates and tasks across layers.

In this paper, we propose a way of combining Docker and micro-services strategies while using a distributed, modular

978-1-5090-5320-9/17/$31.00 ©2017 IEEE

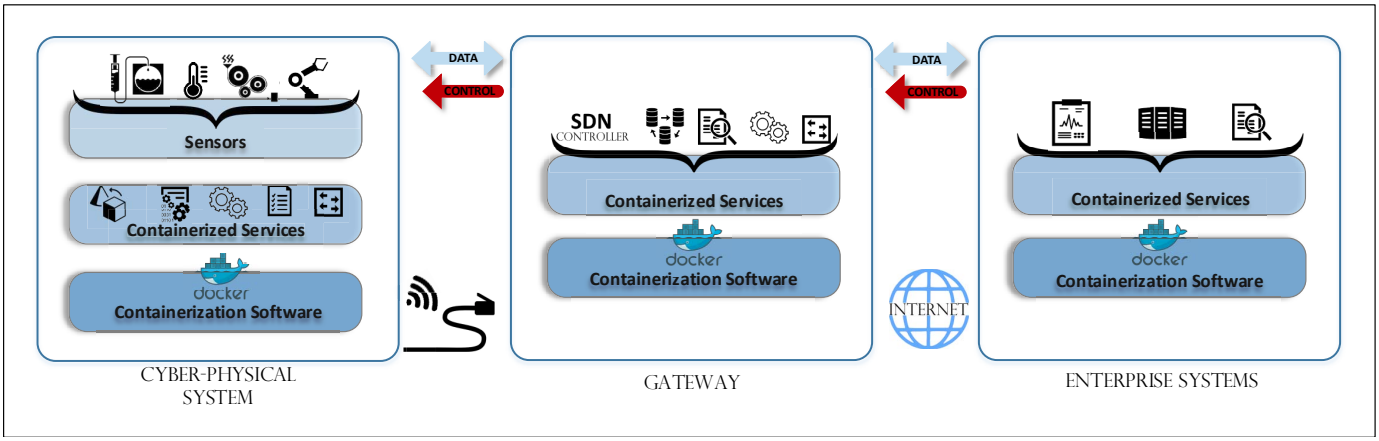

Fig. 1. Proposed Architecture.

and easily scalable architecture that satisfies key requirements for the execution of IIoT applications. Containerization enables service isolation, and different Docker tools allow scaling containerized services. Modularity and decentralization are achieved trough dividing applications in independent micro-services and deploy them across different components of the system. Moreover, interoperability between devices and machines can be abstracted using REST-based protocols and/or a distributed database at the gateway for communication mediation. Developing new services become easy as the architecture is decoupled from technical specifications and complexities. Additionally, combining these two features with proper testing provide a faster way of development and orchestration.

The rest of this paper is organized as follows: Section II presents a review of the related work. Following that, section III describes the proposed architecture. Section IV depicts the implementation of the proposed architecture using SBC as both gateway and end-devices and section on a use case scenario. Furthermore, while section V presents the experimental tests and validation procedures, as well as the obtained results. Finally, section VI summarizes the conclusions and future work.

II. Related Work

In recent years, Docker has been emerging in the Cloud-Computing (CC) world where virtual-machines (VM) used to dominate. A number of research studies have been conducted to compare their overall performance. In [5] an in-depth comparison of different virtualization methods was made, it was concluded that Docker presents minimal overhead in terms of CPU, memory, storage and network performance. This has motivated researchers to study Docker suitability in different scenarios. For instance, Docker was tested as a Platform as a Service for CC [6]. Docker was also analysed in terms of deployment, management and fault tolerance of services [7]. Both authors concluded that Docker could be a suitable candidate for edge computing.

Docker has also been used in [8] to create a Gateway for IoT. In this study, gateway functionalities are extended from the normal protocol conversion and traffic handling. In particular, this study shows how to efficiently use Docker containers to customize the IoT platform, by offering data processing services at the Edge.

In [9] the authors have created a docker based CPS with real-time constraints. A Single Board Computer (SBC) running Docker was combined with a micro-controller to create an automated guided vehicle. In this work, only the end-devices were contemplated, although it became clear that docker could be used alongside physical systems.

Recent micro-service based architecture has been tested in IoT scenarios. For example, [10] presents an architecture for Smart Buildings where sensors share data with micro-services running on SBC. Authors concluded that the flexibility and scalability enabled by the isolated services can benefit IoT deployments.

From previous presented work, it is reasonable to conclude that Docker is a suitable candidate for an IIoT virtualization technique. Different architectures were presented but none of these provide a solution considering end-devices as CPS capable of running containerized services. It is reasonable to assume that with the evolution of CPS the sensing layer will also have enough computational resources to host lightweight virtualization.

III. Proposed Architecture

Figure 1 shows the proposed architecture. As can be seen in the figure, each component is embedded with a Containerization Software (Docker). Applications are divided in small services and implemented inside containers. Each service depicts a different use case and runs as a independent containerized micro-service. Containers can be grouped, managed and scaled by a manager running in the Enterprise System.

The architecture is composed of three different layers, each growing in computational resources. The first layer can interact with the surrounding environment by sensing or monitoring physical systems and/or communicating with the end-user. All collected data is temporarily stored in the border gateway defining a new layer. And therefore, Gateways are having a

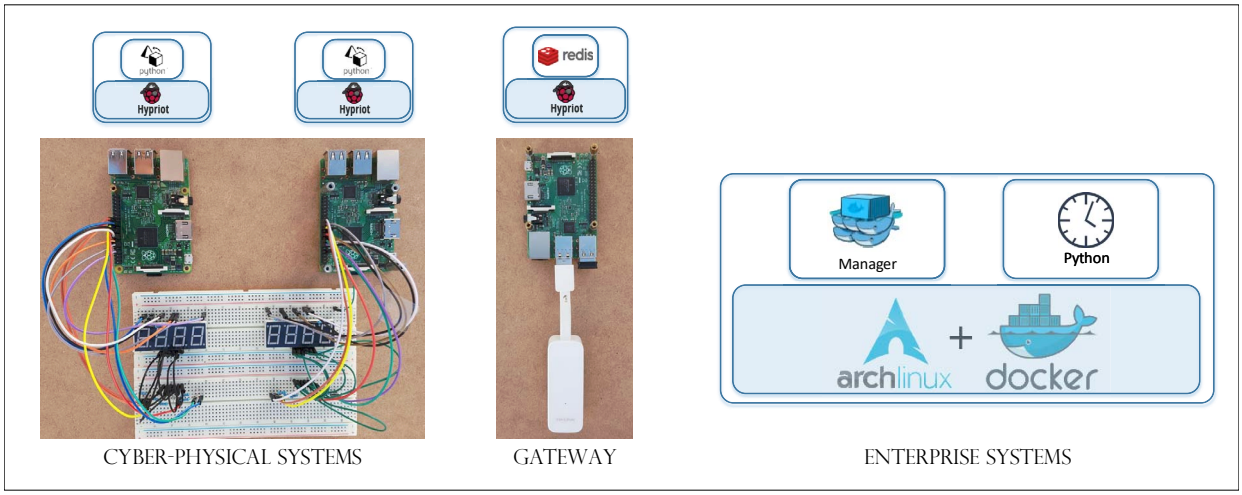

Fig. 2. Test Scenario for Orchestration.

strategic position that is ideal for local data processing and simple analytics. Moreover, they are used to mediate communication between layers, thus reducing network complexity by working as a single entry/exit point. Furthermore, the enterprise tier works as the union point for different local domains. In addition, this layer is for central storage and a controlling point for the subsequent layers.

A. Sensing layer

End-devices are presented as CPS capable of performing lightweight virtualization. This extra layer of computing power allows micro-services to be hosted inside containers. Since containers are managed by the gateway and orchestrated at the enterprise tier, a new level of adaptability is achieved. Services can be deployed on demand or scaled to ensure system performance. Although service monitoring take place on the mediation layer, the enforcement of sensor correct behaviour is made locally by the CPS. To achieve a most efficient response data transformation services have to be combined with behaviour monitoring and sensor management. Also, containerized software defined switches (SD-SW) can be deployed for packet forwarding optimization.

B. Mediation layer

Exploiting the fog computing paradigm, the major role of the gateway is to reduce the gap between end-nodes and the cloud. Moreover, in data-driven systems, preventing data loss is a major concern. Thus, an effective network control and data-management system is required. In our proposal, GWs work has a P2P system, hosting three different distributed micro-services. An SDN controller, a database and a machine learning unit (MLU).

SDN is a networking paradigm that enables network devices to be centrally controlled. An independent communication channel is used to separate the control layer from the data layer. In our topology, this separation is achieved by defining a specific port, or by having an independent physical connection. Moreover, an SDN controller will manage all containerized SD-SWs which are either deployed on the GWs or end-devices. For mediating the communication between different services, controllers translate enterprise system rules to network flows which are inserted into switches. Following the rules instantiated by the controller, SD-SWs perform packet switching for the different micro-services running on the host. These containers will manage both connectivity and packet forwarding enhancing network control.

The MLU provides local intelligence to the gateway which can be used to meet and deploy real-time requirements. The MLU can communicate directly with the controller and the distributed database to perform behaviour analysis. This unit receives trained data from the enterprise layer and compares it to data on other components. It can be used to perceive local system erroneous behaviour, verify different components responsiveness and ensure that micro-services are complying with system politics. The micro-service architecture relies on fully independent services working together. In terms of scalability, using a database for communication between different services can be an hindrance. To allow simultaneously reading and writing by multiple micro-services a distributed database is used and deployed across different GWs. Consistency can be tuned for specific use-cases, but it can be eventually sacrificed to ensure minimal data loss and fast access to data.

C. Enterprise layer

In comparison to gateways where the primary requirement is speed, enterprise systems work with enormous quantities of unstructured data and therefore consistency is a key requirement. Composed by several large computers, the enterprise layer is used for more CPU/memory demanding services. Furthermore, it has a centralized position on the architecture, and consequently a functional viewpoint. Hence, it has three main objectives: data warehousing, data treatment and business logistics. In this regard, by consuming large volumes of data stored; data modeling, system optimization, and behaviour

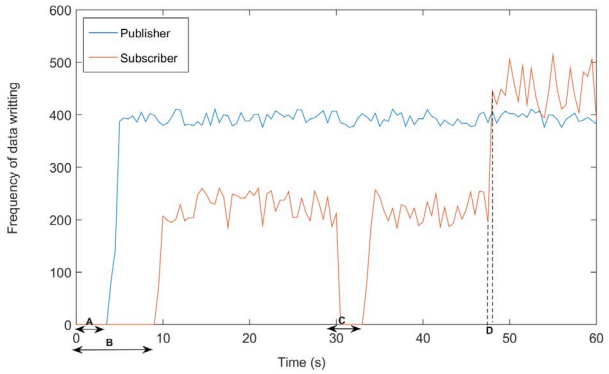

Fig. 3. Published and subscribed data per second

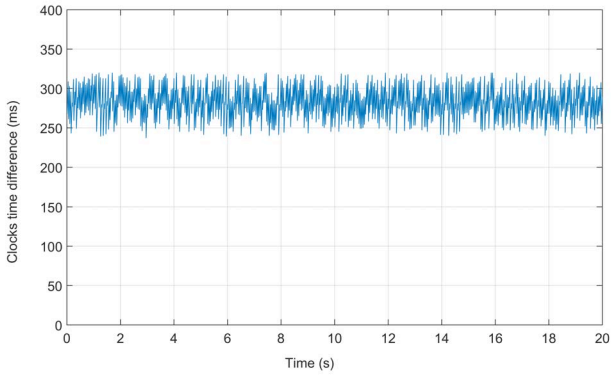

Fig. 4. Comparison between cloud and end-device time in milliseconds

prognosis can be achieved. In fact, through data-mining techniques, the behaviour of different system parts is modeled. Therefore, the results obtained can be shared with the MLUs for a supervised control of the running services. Enterprise Systems are also the main management and control entity. For a more granulated control enterprise politics are translated into orchestration directives, or/and SDN flows. This rules are defined, instantiated, disseminated and enforced across gateways. Thus, a manager instance ensures that business logistics and network policies are enforced. The MLU can tune the policies to increase performance.

D. Key Characteristics

The industry is always adapting, software is being improved and customization is an hindrance for monolithic platforms. In fact, adding new features would normally imply completely re-building the whole system. The modularity provided by the proposed architecture combined with lightweight virtualization orchestration supplied by Docker simplifies management and enables distributed deployments.

In real-time scenarios where resilience is crucial and fault-tolerance essential, having independent services as base-blocks of applications help developers to meet these requirements. In fact, the application logic is distributed across different devices and therefore a single micro-service or even device failure can have no effect on system performance. Moreover, reliability can be achieved through the application of orchestration rules which can ensure service recovery in case of failure. Overall, resilience can be improved by providing redundancy at different layers.

IV. Test Case Scenario

Figure 2 depicts a test use case for the proposed architecture. The purpose of this example scenario is to demonstrate the capabilities of the architecture presented for IIoT deployments. More specifically, a time-sensitive application where end-devices are highly dependent on cloud input data. In this scenario, three Raspberry Pi B+ (RPi) were used with Hypriot Operating System [11]. RPis are SBCs with ARM CPUs and 1 GB RAM memory. Hypriot is a minimal operating system for RPi running Docker. Additionally, two SH5461AS 7-segment displays were used to display enterprise data. Each end-device is a RPi connected to one 7-segment display. End-devices are wired to the GW which communicates with the Cloud wirelessly using Wi-Fi. The Cloud is an Intel core i5 and 8 GB memory, laptop computer running Archlinux OS and Docker. Docker version 1.12 was used to create this topology. Docker 1.12 has a embedded orchestration tool named Swarm which has two types of roles, workers and managers. A worker can be used for hosting containers while a manager can terminate, deploy and manage running containers.

The service created for this experiment is publish/subscribe based. The service was decomposed in three different micro-services, publisher, storage and subscriber. The publisher is a script written in python for publishing the current epoch time to the database every 2 ms. Storage is a Redis distributed database, with a REST interface for publishing data. The subscriber is also written in python and uses a Redis client for subscribing to data changes. This data is then translated and presented in the 7-segment displays.

V. Results and Discussion

To evaluate the proposed architecture, we present and discuss the results obtained while orchestrating a new containerized micro-service to the scenario depicted in the last section. For this experiment, all the docker images used were already stored in the local repository and a Redis database instance was running on the gateway. The test had a duration of 1 minute and begins with the deployment of publisher and a subscriber instance. Twenty seconds after starting, the micro-services running in the CPS are stopped and started in the remaining end device. In order to have both clocks displaying the current time, the first container is restarted after 15 seconds.

To analyse the system behaviour, all data published to redis is also written to a file on the Enterprise System. Additionally after updating the 7-segment display, each subscriber writes the time displayed to a local file. By comparing the different files we can see the number of times data has been published, and the number of times the clock was updated. The results are depicted in figures 3 and 4.

Figure 3 presents the number of times data is written to each local file. We can see that on average the publisher writes 362 times per second while the subscriber updates the clock an average of 220 per second. The letters **A** and **B** mark the time interval taken to start the two micro-services. This time includes both building and deploying of the container. In short, this means that the docker image had to be downloaded from the registry, compiled and used to create a new container. In this context, and with a better computing power, the publisher started in 3.89 seconds. In contrast, the SBC took almost 9 seconds, to start a service with similar size. The interval depicted as **C** is the migration of service between CPS. It takes almost 1.8 seconds to terminate the service and due to the fact that the image was already built the time to start a new instance decreased significantly. Thus, restarting a micro-service (**D**), takes only 876 ms because both building and deployment were already made.

Figure 4 presents the time gap between enterprise layer and the end-device layer. This interval is the difference between the data received by the publisher and the system time. It can be seen that, on average, the seven-segment displays were always delayed in 281 milliseconds. Although significantly delayed, for this use case, where only hours, minutes and seconds are presented on the displays it didn't affect the end-user experience.

VI. Conclusions

Going beyond the existing state-of-the-art work, in this paper, we proposed an architecture considering end-devices as CPS capable of running containerized services. In the proposed architecture, each component has embedded docker, applications (divided in small services) and implemented inside containers. The architecture is composed of three different layers named sensing, mediation and enterprise layer. Devices performing sensing and operations are represented as sensing layer, the intermediate layer representing GWs is characterized as the mediation layer while the enterprise layer is used for more CPU/memory demanding services. By adopting the proposed architecture, reliability can be achieved through the application of orchestration rules which can ensure service recovery in case of failure and system resilience can be improved by including redundancy at different layers. The proposed architecture was tested via a use case scenario in which a time-sensitive application was deployed where end-devices are highly dependent on cloud input data. The obtained results showed that the enterprise layer have management and control capabilities that ensure application deployment through orchestration tools. The results also prove that the implementation of the proposed architecture can deploy the time-dependent micro-services for IIoT.

The presented architecture can also be extended to resource-constrained devices, therefore, it is crucial to reduce system overhead. This can be achieved by reducing the docker images used and adopting different protocols for the inter-microservice communication. In this work, we used Hypertext Transfer Protocol (HTTP) as a Representational State Transfer (REST) interface for the database microservice. We are currently working on changing it to CoAP (Constrained Application Protocol) hoping to improve the communication. In addition, the programming language used was Python which is also being changed to C in order to reduce the docker image size.

Acknowledgments

This work is funded by National Funds through FCT - Fundação para a Ciência e a Tecnologia, Portugal, under the project UID/EEA/50008/2013.

References

[1] P. Bartolomeu, M. Alam, J. Ferreira, and J. Fonseca, "Survey on low power real-time wireless mac protocols," *Journal of Network and Computer Applications*, vol. 75, pp. 293–316, 2016.

[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*, ser. MCC '12. New York, NY, USA: ACM, 2012, pp. 13–16. [Online]. Available: http://doi.acm.org/10.1145/2342509.2342513

[3] D. Namiot and M. Sneps-Sneppe, "On iot programming," *International Journal of Open Information Technologies*, vol. 2, no. 10, 2014.

[4] C. Anderson, "Docker [Software engineering]," *IEEE Software*, vol. 32, no. 3, pp. 102–c3, 2015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7093032

[5] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An Updated Performance Comparison of Virtual Machines and Linux Containers," *Technology*, vol. 25482, pp. 171–172, 2014.

[6] D. Liu and L. Zhao, "The research and implementation of cloud computing platform based on docker," *2014 11th International Computer Conference on Wavelet Actiev Media Technology and Information Processing(ICCWAMTIP)*, pp. 475–478, 2014. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7073453

[7] B. I. Ismail, E. Mostajeran Goortani, M. B. Ab Karim, W. Ming Tat, S. Setapa, J. Y. Luke, and O. Hong Hoe, "Evaluation of Docker as Edge computing platform," *ICOS 2015 - 2015 IEEE Conference on Open Systems*, pp. 130–135, 2016.

[8] R. Morabito, R. Petrolo, and V. Loscr, "Enabling a lightweight Edge Gateway-as-a-Service for the Internet of Things," pp. 1–5, 2016.

[9] P. González-Nalda, I. Etxeberria-Agiriano, I. Calvo, and M. C. Otero, "A modular CPS architecture design based on ROS and Docker," *International Journal on Interactive Design and Manufacturing*, pp. 1–7, 2016.

[10] D. Salikhov, K. Khanda, K. Gusmanov, M. Mazzara, and N. Mavridis, "Microservice-based IoT for Smart Buildings," no. Ii, 2016. [Online]. Available: http://arxiv.org/abs/1610.09480

[11] Hypriot, "Docker Pirates ARMed with explosive stuff," https://blog.hypriot.com/, 2017, [Online; accessed 10-January-2017].