Investigation of Containerized-IoT Implementation Based on Microservices

Jayanshu Gundaniya and Chung-Horng Lung(✉)

Department of Systems and Computer Engineering, Carleton University, 1125 Colonel By Drive, Ottawa, ON K1S 5B6, Canada
jayanshugundaniya@carleton.ca, chlung@sce.carleton.ca

Abstract. Due to a rapid increase in potential scales of Internet of Things (IoT) systems, it becomes challenging to deploy, patch and upgrade applications effectively. Breaking these applications down into small and independent microservices and isolating them by using container techniques leads to faster development, deployment and integration as compared to the traditional monolithic development approach. This paper introduces one such experimental implementation of simulated generic IoT services deployed separately as Docker containers on different machines using an open-source container orchestration platform Docker Swarm. Such a distributed implementation enables easier deployment and expansion of the system by adding and modifying only a few services in need. The objective of the approach is to conduct rapid prototyping of container-based simulated IoT microservices and investigate the management of multiple containers using open-source Docker Swarm. We validated the concept of developing and integrating containerized IoT microservices and evaluated the effectiveness of the concept. The experimental result shows that container-based IoT applications can be efficiently developed as microservices and multiple IoT containers can be effectively managed with an orchestrator. The experience can be applied to real IoT applications using actual IoT sensors and devices.

Keywords: Internet of Things · Microservices · Containers · Docker · Docker Swarm

1 Introduction

Internet of Things (IoT), one of the most popular topics of research these days, is constantly being updated with new features. All these features lead to an increase in its scale and thus we need an architecture and design that enables us to dynamically add these new features and scale them according to the needs without undergoing a system redesign at each stage. Moreover, according to predictions conducted by Gartner, the total number of connected IoT devices would reach about 21 billion in 2020 [1]. Deployment of large IoT applications is still in the beta stage. As the number of devices is increasing, a decentralized approach is needed for solving issues regarding the communication and management of devices. To make devices and services work independently would

H. Song and D. Jiang (Eds.): SIMUtools 2020, LNICST 370, pp. 104–115, 2021.
https://doi.org/10.1007/978-3-030-72795-6_9

involve decoupling them in development and deployment. Service Oriented Architecture (SOA) is one of the widely used solutions for achieving loose coupling between services, but it lacks support for the constant patching and integration of new services.

Though the concept of microservices has been in use for years, the term "microservices" was coined in 2014 [2]. In microservices, applications consist of small and independent services that communicate using lightweight methods like HTTP API [3]. The authors in [3] have explained it as: "In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies". This allows individual services to be tested or patched independently without disturbing the rest of the deployment. Microservices Architecture is already in use heavily by companies like Netflix and Amazon, and it is attracting more attentions in both academia and industry.

These services, if running on the same machine, need to be isolated from each other to separate the environment variables used by a service from being modified by other services. This is usually realized by containerizing those services. Application containerization is an OS-level virtualization method [14]. It can be used to run distributed applications without the overhead of launching and running a fresh operating system as an entire virtual machine (VM) for each app. Multiple isolated applications can be deployed on a single host and can utilize the API of the same OS kernel. The invention of an open-source containerizing engine called Docker [15] has made containerizing easy and popular as Docker containers are designed to run on everything from physical computers to virtual machines, bare-metal servers, and more.

Microservices and distributed applications go together with containerization, as each container operates independently of others. Microservices communicate via APIs with the container virtualization layer. Microservices could be scaled up to distribute the load and meet the rising demand for an application component, thus making the system flexible.

The main objective of this paper is to experimentally validate the effectiveness of applying microservices using containers for IoT systems which have become popular and in high demand. Specifically, this paper investigates an approach consisting of simulated IoT applications bundled as microservices designed and implemented as containers. The main evaluation criteria include the deployment of containerized-IoT applications, management of multiple containers, and extensibility of the system for adding more sensors and devices. This approach is deployed on multiple machines by using an orchestration platform Docker Swarm [16]. The experimental result demonstrates that the simulated IoT application can be effectively integrated using microservices for continuous development with container techniques.

The rest of the paper is organized as follows. Section 2 describes the background information and related work. Section 3 presents the design and implementation of containerized microservices. Section 4 describes the conclusion and future directions.

2 Background and Related Work

This section explains the research performed on the application of microservices in IoT. A great deal of research has been conducted to integrate the application design areas since microservices was coined. Key issues and design choices like orchestration, communication protocols, and microservices are the primary areas of research.

2.1 Internet of Things (IoT)

IoT is a term used for connected computing devices which may be mechanical or digital systems that communicate by transferring data over a local or global network with little or no human-to-human or human-to-computer interactions. It can be an automobile with proximity sensors, a volume measurement unit in an industry, etc.

An IoT system could consist of numerous devices that have embedded processors coupled with intelligent sensors. The sensors collect data, communicate with a gateway for data transfer and the gateway can also act on data to achieve an objective. These devices may even communicate with other related devices and act on the data they get from one another.

The deployment of IoT applications mostly is still in the development stage. Prediction released by Gartner in 2018 [1] indicates that almost half the cost of implementing IoT solutions would consist of efforts spent on integrating them with one another and the backend. Other challenges may include concurrent operations and communications of the connected devices and their security on the network.

2.2 Microservices and Containerization

The microservice architecture allows for the continuous delivery/deployment of large-scale applications having high complexity levels. It is a unique approach for developing a large single application as a collection of small independent services, each running separately and communicating with one another using lightweight mechanisms, such as a REST API [17]. A REST (Representational State Transfer) API (application program interface) defines a way in which a developer can design an application (client) to request services from a server. It usually uses HTTP GET, POST, PUT and DELETE requests for communications. A large application is broken down into small services based on their business capabilities. Their deployment could also be automated like the original monolith. Centralized management of these services is kept at a minimum to avoid tight coupling. Alshuqayran et al. [4] presented a mapping study on the research conducted for the challenges and quality requirements of microservices.

One of the ways these small independent services are deployed is to use containers [14]. A container packs the application/service under its consideration with all dependencies with it, thus making it deployable on any environment running a container engine. Although containers are memory efficient, portable, and scalable, some of its disadvantages include the inability of security monitoring tools to protect the containers as most of them are designed for hypervisors, virtual machines (VMs), and natively run OSes.

2.3 Microservices and IoT

Butzin et al. [2] introduced a microservices approach to IoT applications. They mentioned the importance of the Circuit-Breaker pattern [5] which will monitor the health of services and prevent a broken service from receiving calls. Another important concept they discussed that goes well with the Circuit-Breaker is the Load Balancer pattern [6] that would distribute workload amongst equal services and works serially with the Circuit-Breaker. Petrenko et al. [7] described the semantic workflows that are useful in IoT and discussed an orchestration approach to dynamic service-oriented systems.

Campeanu [8] published a mapping study on the progress of microservice architecture in IoT. The paper states that "the number of publications in 2016 is four times larger than to the previous year. Moreover, the number of publications in 2017 is higher (i.e., 62%) than the previous year." Datta et al. [9] provided an end-to-end skeleton architecture for deployment of microservices suitable for IoT systems both on an edge server and on cloud. Khazaei et al. [10] presented a self-managing autonomic containerized IoT platform called SAVI-IoT for various use cases, such as Big data compatibility, in-place data processing, etc. Khanda et al. [11] also investigate implementing IoT as microservices in university buildings by using Jolie, a programming language built for microservices.

2.4 Docker Containers

Docker containers have been widely used in industries these days [15]. Docker is an open source containerizing engine designed to deploy and run applications. Docker allows the developer to isolate the applications from the infrastructure or system it is being deployed on for faster delivery of the software.

Docker isolates the environment in which a container runs, thus providing an ability to deploy it on any host OS which has a Docker Engine. Each container also packs the required dependencies required by an application. Thus, multiple containers having different dependencies can separately run on a single host machine without the overhead of using a hypervisor or a VM manager. Some key features of Docker containers are highlighted as follows:

- Services: A Docker service is a bunch of containers in production. All those containers run on one image but could have different ports exposing them. A service can contain multiple replicas of a single container depending on the load required. This number could be increased or reduced, thus leading to dynamic scaling of the system.
- Swarm: Docker swarm [16] consists of a swarm manager employing various strategies to run containers on multiple nodes. Strategies such as the emptiest node, global container, etc., could be used. Some crucial features available from Docker Swarm include service discovery, load balancing, multi-host networking, etc. [2].
- Stacks: Docker stack consists of a group of interrelated services that share the same dependencies. In production, these could be orchestrated and scaled together. In this paper, only one stack is used but complex applications usually require multiple stacks.

3 Design and Implementation

In this paper, the usage of microservice architecture for a generic IoT system is analyzed. Our system architecture and design are inspired by [10] but the implementation is primarily written in JavaScript with the use of containers for isolation. We briefly discuss the design and then describe the use cases that suit such a development.

The deployment of containers in our approach was carried out using Docker on a machine running Ubuntu 16.04 with 4 GB RAM and Pentium 4 processor.

3.1 System of Containers

The applications used in this implementation are loosely based on a workshop conducted by Nearform [13]. Our system involves services like frontend, database, serializer, broker, and simulated IoT applications. The services were written in JavaScript with Nodejs runtime. The experiment carried out in this paper preserves the implementation of some of those services and adds some more to it. Moreover, we use Docker Swarm and deploy containers on multiple machines with the help of its orchestrator instead of the regular implementation as standalone Docker containers on a single machine. A generic model of the containers used in this paper and their communications is shown in Fig. 1.

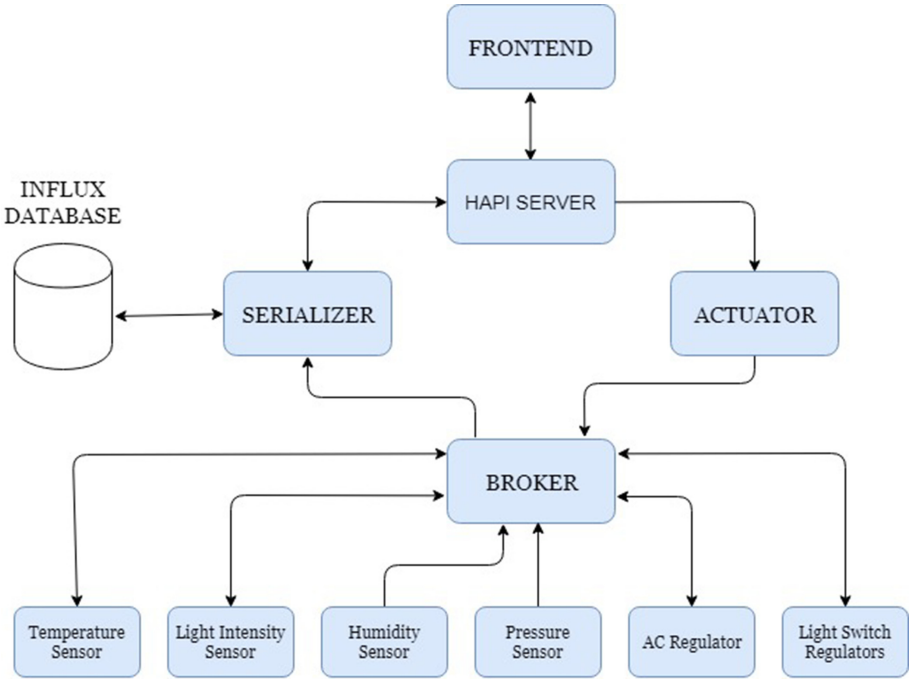

Fig. 1. Block diagram of the proposed implementation

A brief description of the containers shown in Fig. 1 is presented as follows:

- *Frontend*: This is the user interface (UI) of the system. It is a Hapi server to poll the *Database* for retrieval of values that are to be displayed.
- Serializ*er*: It writes to the *Influx Database* using the Nodejs APIs and it receives the poll from the *Frontend* and commands from the *Broker*.
- *Influx Database*: The database is a container implementation, which is a time-series database.
- *Broker*: In this implementation, the message broker is implemented using Mosca, which is an MQTT broker written in JavaScript. It receives messages on topics and publishes them to subscribers.
- *Actuator*: The component acts as an intermediary between the *Frontend* and the *Broker*. The *Broker* has a specific API used to communicate with the *Actuator*. The *Actuator* takes the overhead off the *Frontend* from calling the API.
- *Temperature, Humidity, Pressure and Light sensors*: Each of these IoT sensors randomly generates a value for its respective category every few seconds and sends it as the {topic, value} pair to the *Broker*.
- *AC and Light Regulators*: Both are subscribed to the *Broker* for receiving messages on appropriate topics and turn devices, e.g., AC or light, on or off depending on the received values and current threshold values via the *Broker.*

The *Frontend* uses the Hapi server, which uses a polling service. The polling service polls the *Influx Database* periodically through the *Serializer* using Seneca actions. The *Frontend* is also connected to the *Actuator* and calls its actions via Seneca, which are then converted to appropriate messages by the *Actuator* and passed to the *Broker* which will publish it for the subscribers to read. Every read or write or database creation done by the database is carried out by the *Serializer*. The *Broker*, on receipt of a 'write' message from any of the sensors, triggers the *Serializer* using Seneca APIs and passes the values to be written into the database. The sensors are all connected to the *Broker* and subscribed to relevant topics except humidity and pressure sensors as they neither receive offset values nor commands.

3.2 Use Cases

We adopt some of key use cases presented in [11, 12] in our evaluation of the architecture and design. There are:

- *Measurement of conditions in a smart home*: Temperature and light measurements in rooms are measured. This is a basic and yet critical requirement for many IoT applications, as new devices or sensors may need to be added or replaced. The idea can be easily applied to other similar types of sensors, e.g., humidity, fire alarm, etc.
- *Management of home appliances*: Applications for AC and Light Regulators have been tested in the experiment. Other appliances having configurable interfaces can also be more easily added in our implementation than in a usual monolithic approach, as only a small number of relevant containers may be related.
- *Service replicas*: A service can contain multiple replicas of a single container depending on configuration and the number of replicas can be changed dynamically. This feature is supported by Docker, which can improve system performance by deploying

more instances and can increase reliability in case of a failure of one instance. The failover will be handled automatically by Docker Swam.

3.3 Description of Workflow

Figure 2 shows the administrator UI of the *InfluxDB* container which is exposed on a port, i.e., 8083, (administrator interface port for the container implementation) of localhost (or any other IP of a connected swarm machine). The top right corner as depicted in Fig. 2 provides an option to select the database. In this experiment, five databases are used: temperature, light, pressure, humidity, and status. The first four databases are for storing values from their respective sensors, whereas the last one is for storing the status of AC or Lights as provided by their containers. Figure 2 displays some timed temperature values (random values in a range for illustration) obtained from sensor 1. *InfluxQL* is a SQL-like query language for interacting with *InfluxDB* and providing features specific for storing and analyzing time series data [7].

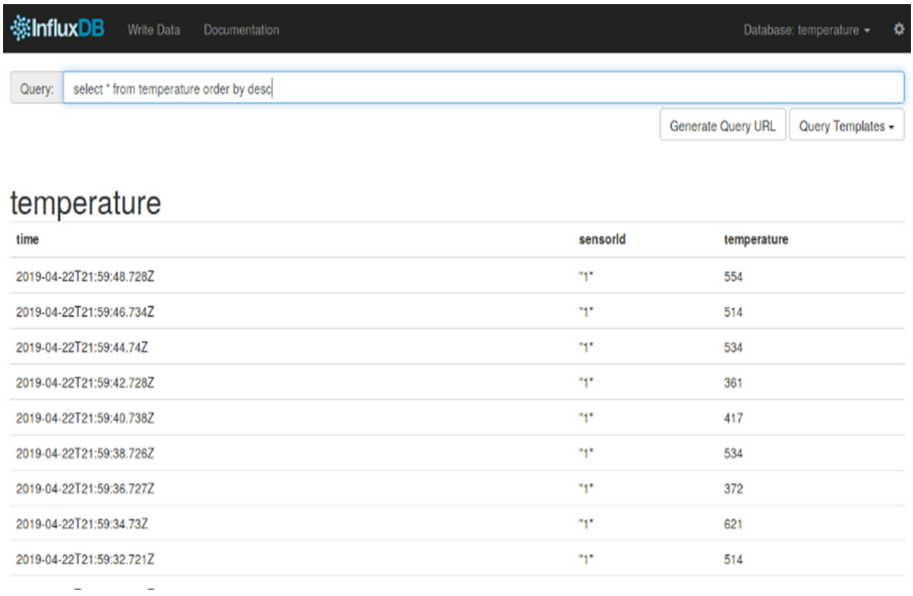

Fig. 2. Screenshot of an *InfluxQL* query run on administrator UI of *InfluxDB*

The top five components, as shown in Fig. 1, are designed using containers. They are all functioning in an orchestrated manner. When sensor values are to be plotted on the graph, the *Frontend* will call appropriate actions on the *Serializer* which will retrieve the values from the *InfluxDB* and provide them to the *Frontend*. A similar workflow occurs when a command from the *Frontend* needs to be sent to a sensor or regulator. The *Frontend* will first call actions from the *Actuator* which will then act on behalf of the *Frontend* to convert commands into appropriate messages for the *Broker*.

When a sensor value needs to be written to the *InfluxDB*, the sensor sends the value to the *Broker* as a message and the *Broker* will then call appropriate actions from the *Serializer* to write to *InfluxDB*.

The communications between sensors and *Actuator* are supported by message passing via the *Broker* and they work independently. On the other hand, *Frontend* needs to call appropriate actions from *Serializer* or *Actuator* to get its work done. Thus, when a new sensor is added (as a container), it is easy for it to subscribe to messages from other sensors or the *Actuator* and start working as it should.

Figure 3 shows the output of various sensors, i.e., temperature, humidity, pressure, and light being logged on the *Frontend* using the rickshaw charting framework. The *Frontend* pulls the output data values registered by the sensors in the past 20 min from the database via a Seneca [18] action by calling appropriate functions in the *Serializer*. Seneca supports microservice system by using a messaging mechanism based on pattern matching. It hides where services are located, how many of services there are, or what services do. Everything external to the microservice, e.g., databases, is hidden behind microservices. The decoupling facilitates system expandability for continuously adding new features.

The values in those graphs are the values randomly generated, for concept demonstration purpose, by the sensors based on a pre-configured offset using simulation. The offset is different for different sensors to make it easy to differentiate while testing. Temperature and Light sensors even allow the offset to be modified for better testing as the offset values affect the status of AC and lights, respectively. The offsets are not registered in the database as they are just for simulation.

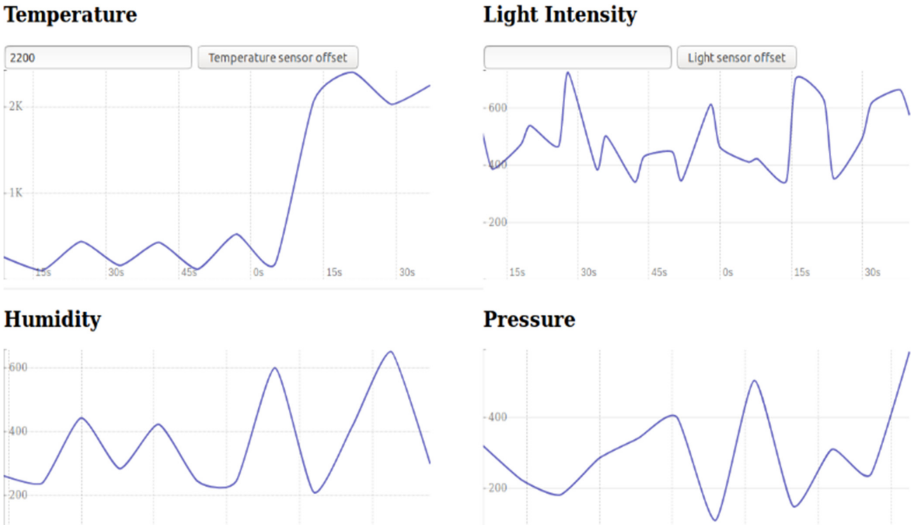

Fig. 3. Graphs based on sensor-generated values: an illustration

As an example in IoT, in Fig. 4, the upper and lower thresholds of the AC (top left) are displayed. When the Temperature offset is set, an appropriate action from the *Actuator* is

invoked using a Seneca action. The *Actuator* then converts it into a message format with a topic intended to the AC Regulator and sends the message to the *Broker*. The *Broker* receives it and publishes it for the corresponding subscriber(s) to read. The temperature sensor, being subscribed to receive the offset for it, does so and starts generating values above the new offset (2200 as shown in Fig. 4). The next generated value, as before, is sent in a message with a topic to the *Broker* so that it can act on it and can also let other subscribers read it.

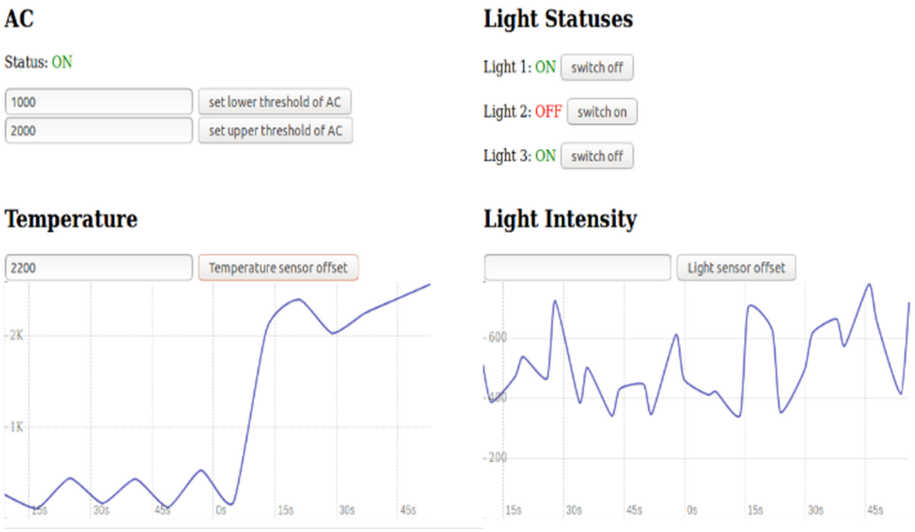

Fig. 4. AC and Light status: an illustration

When the AC container receives this message, it will act on it to turn on the AC. After switching the AC on, it sends out a message to the *Broker* with an appropriate topic intending that it needs to write the new status of the AC into the *Influx Database,* so that *Frontend* can poll and update itself. The *Broker* gets the message and uses a Seneca action to call the corresponding *Serializer* function which writes to the *Influx Database.* When the *Frontend* polls the next time, it gets a new status and updates the respective output.

In Fig. 5, it can be observed that the AC is being turned OFF, because the temperature has dropped below the lower threshold. The flow of messages here is the same as it was before when the AC was turned ON due to increasing offset.

A similar workflow or explanation can be applied to the lights except the lack of threshold setting.

3.4 Monitoring and Managing Containers

Figure 6 shows the Portainer UI which is a Docker management interface. Portainer supports monitoring and management of multiple containers by displaying the node on which the container is deployed, container ID and also the metrics of the container,

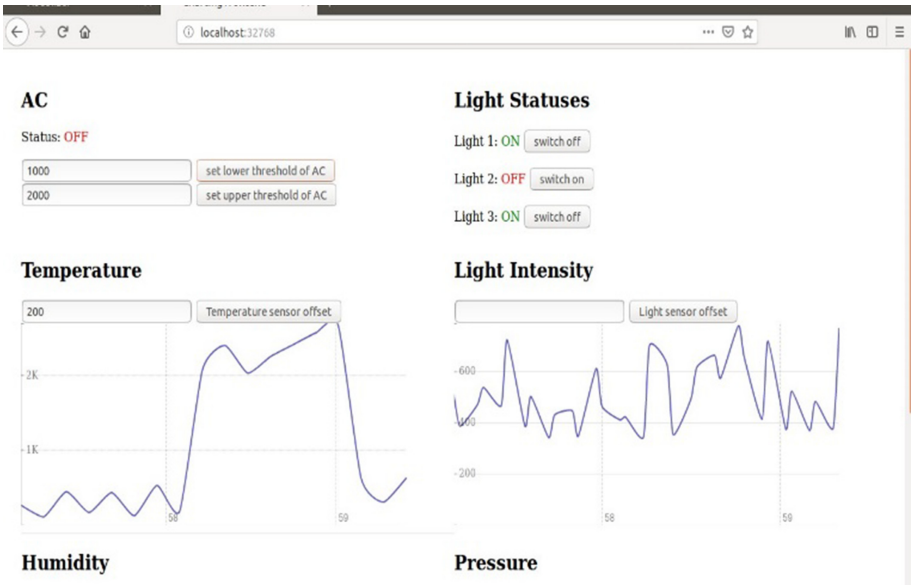

Fig. 5. Update on AC status: an illustration

such as the memory and CPU consumption of a container. Portainer also can present a network view and volume labels defined as pa,rt of a container. In addition, containers can be stopped or restarted from the management interface.

Moreover, scaling the container by creating replicas to improve reliability and possibly performance can be easily realized via the UI itself with the 'Scale' option under the 'Scheduling Mode' (the 3rd column as depicted in Fig. 6). If a containerized replication fails, the Docker management will automatically take an appropriate action, e.g., load balancing among remaining replicas or move the workload to the last running container, when the failure is detected.

The 'Swarm' option on the left can display the nodes in the swarm, as shown in Fig. 6, including the memory, total cores, hostname, and also collectively displaying this information for the entire swarm. Our deployment was on three machines. Figure 6 shows that the *Broker* is running on the node laptop-3. The 'Stacks' option shows a list of services that are deployed to a part of that stack as in the screenshot. We deployed one stack containing all the containers called 'myproject'. So, the 'Stacks' shows one entry named 'myproject'. When it is checked and expanded, it shows the list of services that are deployed to a part of that stack as in the screenshot. When a specific service is expanded, e.g., 'myproject_broker' in Fig. 6, the container (and replicas) running under the service is (are) displayed.

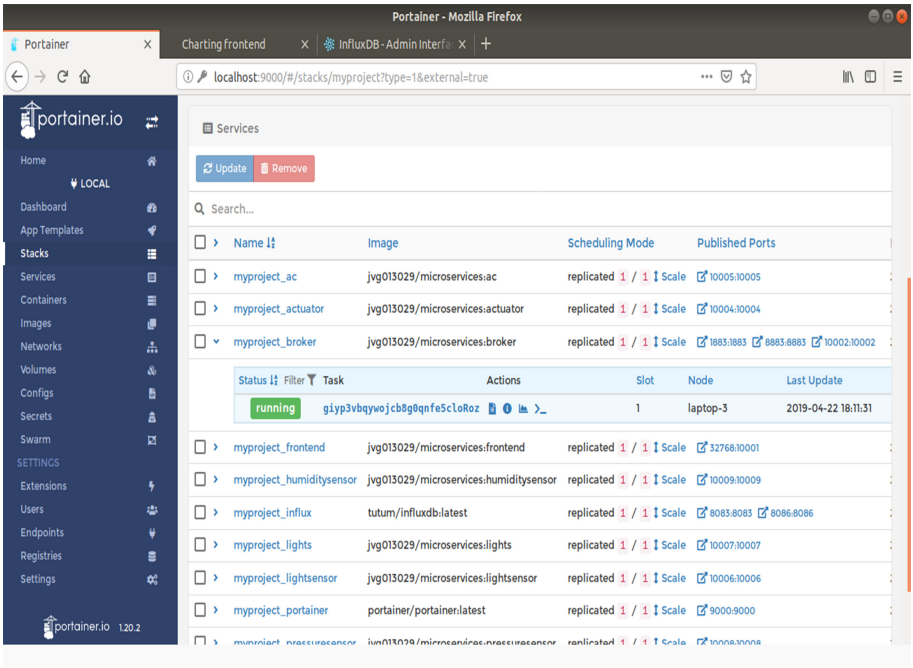

Fig. 6. Portainer UI for docker management

4 Conclusion and Future Work

IoT applications have attracted a great deal of attention and the deployment of large scale or high scalability IoT applications becomes crucial. On the other hand, containers and microservices have been popular in practice due to a number of advantages, particularly independent development, incremental deployment, high scalability. Management of multiple containers based on the microservice architecture becomes an important task, as the number of containers can be high. The objective of this paper was to experimentally validate the effectiveness of applying microservices using containers for IoT systems which has become popular.

The main benefits of such an architecture and subsequent implementation include the ease of adding new sensors or services, ease of deployment due to dependencies being packed with the applications in the containers and usage of lightweight communication protocols (Message Queueing Telemetry Transport in our case). The trade-off is the increased overhead on the system as containers are more resource heavy than individual applications combined.

In this paper, typical IoT applications have been simulated as a suite of small and independent (micro)services which were deployed as isolated containers for feasibility and effectiveness investigation. The simulation of IoT applications was realized using multiple containers and Docker Swarm for container management. The simulation demonstrated a successful rapid application prototyping experiment that preserves the benefits stated above, which reveals that the concept could be further turned into real

testbed implementation using a large number of real sensors and common IoT-related controllers, e.g., Arduino or Raspberry Pi.

References

1. Gartner Report. https://www.informationweek.com/mobile/mobile-devices/gartner-21-billion-iot-devices-to-invade-by-2020/d/d-id/1323081. Accessed 16 July 2019
2. Butzin, B., Golatowski, F., Timmermann, D.: Microservices approach for the Internet of Things. In: Proceedings of the IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1–6 (2016)
3. Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html. Accessed 16 July 2020
4. Alshuqayran, N., Ali, N., Evans, R.: A systematic mapping study in microservice architecture. In: Proc. of the 9th IEEE International Conference on Service-Oriented Computing and Applications (SOCA), pp. 44–51 (2016)
5. Circuit Breaker pattern. https://msdn.microsoft.com/de-de/library/dn589784.aspx. Accessed 16 July 2020
6. Cloud Computing Patterns Load Balancer. https://patterns.arcitura.com/cloud-computing-patterns/mechanisms/load_balancer. Accessed 16 July 2020
7. Petrenko, A., and Bulakh, B.: Intelligent service discovery and orchestration. In: Proceedings of IEEE 1st International Conference on System Analysis & Intelligent Computing (SAIC), pp. 1–5 (2018)
8. Campeanu, G.: A mapping study on microservice architectures of internet of things and cloud computing solutions. In: Proceedings of the 7th Mediterranean Conference on Embedded Computing (MECO), pp. 1–4 (2018)
9. Datta, S.K., Bonnet, C.: Next-generation, data centric and End-to-End IoT architecture based on microservices. In: Proceedings of IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia), pp. 206–212 (2018)
10. Khazaei, H., Bannazadeh, H., Leon-Garcia, A.: SAVI-IoT: a self-managing containerized IoT platform. In: Proceedings of the IEEE 5th International Conference on Future Internet of Things and Cloud, pp. 227–234 (2017)
11. Khanda, K., et al.: Microservice-based IoT for smart buildings. In: Proceedings of the 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA), pp. 302–308 (2017)
12. Soliman, M., Abiodun, T., Hamouda, T., Zhou, J., Lung, C.-H.: Smart home: integrating Internet of Things with web services and cloud computing. In: Proceedings of IEEE 5th International Conference on Cloud Computing Technology and Science, pp. 317–330 (2013)
13. Nearform. https://github.com/nearform/micro-services-tutorial-iot. Accessed 16 July 2020
14. Container. https://www.docker.com/resources/what-container. Accessed 16 July 2020
15. Docker. https://www.docker.com/why-docker. Accessed 16 July 2020
16. Docker Swarm. https://docs.docker.com/engine/swarm/swarm-tutorial/. Accessed 16 July 2020
17. REST. https://restfulapi.net/. Accessed 16 July 2020
18. Senaca. https://senecajs.org/getting-started/. Accessed 16 July 2020