A PERFORMANCE EVALUATION OF DOCKER-BASED MQTT SERVER IMPLEMENTATION ON INTERNET OF THINGS DEVICE

Summary. The paper presents an analysis of the implementation of the MQTT server on the Raspberry Pi device. The standard configuration of the server (without virtualization layer) is compared to the configuration based on a Docker container and the principles of microservices. The test system has been evaluated over the performance of the MQTT server.

Keywords: Internet of Things (IoT), microservices, Docker containers, MQTT

1. Introduction

In the area of issues related to network-based applications and Internet of Things (IoT) systems, there is an increasing interest in using the edge processing-enable components [1]. The main goal of such approach is the reduction of the relatively high end-to-end delays,
which are present in most (if not all) of cloud-based systems. In practice, it means the relocating of remote applications from remote location (usually a cloud infrastructure) to the edges of a network. The growing popularity of IoT systems and especially sensor networks, additionally reinforced this trend. Nowadays, this approach is often connected with so-called fog computing (FC) systems [2]. FC assumes that processing units are placed as close to data source as it is possible. Obviously, besides of many advantages of this approach, several new problems have also arisen. One of them is associated with proper and efficient implementation, configuration and maintenance of an application based on distributed computing units, each with limited hardware resources. Also, the diversity of hardware and software platforms is also a factor which does not make the computing at the edge a straightforward or easy task. One of the promising techniques which offers the solution for above issues, is lightweight virtualization. In this area, a great interest is focused on Container-based Virtualization (CV) [3]. Thanks to this, the last few years brought the significant development in the field of implementation and orchestration of software containers. One of the most promising and the same time wide-accepted example of CV solutions, is Docker container. An availability of new, powerful System on Chip (SoC) devices supporting the Docker containers, opens the opportunity to develop the whole new range of systems and applications dedicated to IoT and FC.

In the paper, we present the results of the performance evaluation of IoT component implemented according to CV principles. We developed the simple test environment to assess the fundamental features of microservices approach. As a sample application, the MQTT broker was used. The rest of paper is organized as follows: Section 2 shortly describes Docker container and the idea of microservices. In Section 3 the test scenarios are presented. Section 4 contains the results and analysis. Finally, Section 5 gives some conclusions.

2. Docker containers and microservices

Through many years, applications have been prepared for particular hardware platform and operating system. However, throughout this time, an effort was put into developing methods for decoupling IT systems from the hardware layer on which they were implemented. The one of the wide-known ideas is the concept of “application in container”. Its roots reach directly to the end of the last century and solutions for Unix-based systems. Thus, it is not quite a new approach in the field of virtualization [3,4] but recently it has achieved more relevance with the introduction of the Docker containers. In a very short time, Docker environment has been adopted by most software and hardware companies. Like its
predecessors, the Docker uses isolation of processes at the operating system level. Thanks to that, it is possible to avoid the overhead related to virtualized hardware and virtual device drivers. Docker provides its own container engine based on Linux kernel as a means of execution of this isolation. However, the factor that we believe plays a key role in Docker's popularity is the availability of tools for container orchestration [3,7]. This set of tools, in combination with the functional API, allows building, managing, and removing a containerized application. According to traditional deployment process, a code is put into an existing hardware node. At the same time, it is expected that all required software components are in place and are properly configured. Docker (or generally, any type of software container), contains all required dependencies which containerized application needs. This feature is also referred as container self-sufficiency and immutability. It plays very important role in the context of implementation of IoT systems [3,5]. From a practical point of view, the fundamental element of the Docker’s environment is so-called image [6]. The image can be sent and started (this way a container is created) on a hardware with installed Docker environment [7].

Fig. 1. The difference between implementation of lightweight and heavy virtualization
Rys. 1. Różnice pomiędzy implementacją lekkiej i ciężkiej wirtualizacji

Software containers are considered as a lightweight alternative to heavy virtualization (also called as the hypervisor-based virtualization). The origin of this name “lightweight” stems directly from the differences in resource utilization between both approaches. Typically, when a physical machine running a few virtual machines, it would require the same number of operating systems (guest OSs) collaborating with a hypervisor and host OS. It is a very commonly used solution when the full-featured virtual instance is considered. This is not the case when development of distributed IoT application is taking into account. Here, one does not need the virtual host but only a virtual environment, which includes a set of specific
dependencies needed to run a specific application. Thus, all what is needed, is the possibility to share the resources of the host operating system (usually binaries and libraries). The software containers (including Docker containers) offer such solution and, at the same time, ensure proper isolation of resources [8]. The difference between the heavy virtualization and CV approach is illustrated in Fig. 1. Thanks to combination the features of software containers and orchestration tools, it is possible to design scalable applications and implement them in any container-ready hardware platforms [9]. This in turn has led to an increased interest in the concept of microservices.

Generally, microservices describe a method of developing distributed software applications. The main idea of microservices is based on independently deployable, small, modular services, each running a unique process [6,9]. These microservices communicate through a well-defined, dedicated, lightweight mechanism which uses ports and links [10]. While building communication structures between different services, the smart endpoints and dumb pipes approach is favoured [1,10]. The behaviour of the single service is related to a domain logic. Usually, the service acts as a kind of the filter i.e. receives a request, applies logic as appropriate and produces a response [11]. These features directly correspond with requirements of distributed applications with are to be implemented at IoT/FC systems of very different complexity scales.

![Diagram](image_url)

Fig. 2. The difference of scaling process between application based on Docker containers and monolithic application

Rys. 2. Różnica w sposobie skalowania aplikacji opartej na kontenerach Docker i aplikacji monolitycznej

From the implementation points of view, microservices offer the gain in resources utilization while an application scaling is required. Because each service is responsible for different task within the application, it is possible to scale only these services which influence on the application efficiency. In the other words, it does not require to multiple whole monolith application. This allows for application flexibility on the level of its structure and state of distribution. This feature of microservice approach is presented in Fig. 2.
Above facts and features give grounds to say that combination of Docker containers and microservices for developing distributed, IoT application is very promising. This in turn led us to test a performance of container technologies on typical IoT device.

3. Methodology of the tests

The tests of usefulness of the container-based virtualization were conducted on well-known IoT device, the Raspberry Pi 2B (RPi2). Our aim was to realize a comparison based on an example that is very commonly used in IoT systems. For this reason, we have configured the testbed consisting the Message Queuing Telemetry Transport (MQTT) broker and a few MQTT clients.

MQTT is a lightweight publish/subscribe messaging protocol [11]. It is based on the principle of publishing messages and subscribing to topics. The general rule says that messages in MQTT are published on topics and there is no need to preconfigure topics. In other words, the publishing on topic is enough. In typical application, the topics are organized as a hierarchy. The hierarchy resembles the file system structure and it allows flexible arrangement of topics. Clients can receive messages by creating subscriptions. The MQTT defines two types of subscriptions, to an explicit topic or to the fragment of hierarchy (usually, to a single level of hierarchy or to all remaining levels of hierarchy). The MQTT broker creates a simple, common interface to wide-range sensors and IoT devices. The idea of MQTT data exchange schema is presented in Fig. 3.

The MQTT data exchange schema includes three levels of Quality of Service (QoS). The concept of QoS means in this case the level of certainty that a message is received. Messages may be sent at any QoS level. Also, clients may subscribe to topics at any QoS level. According to MQTT protocol, higher levels of QoS are more reliable. For obvious reasons, they also cause higher latency and have higher bandwidth requirements.

Fig. 3. Scheme of data exchange in a system based on the MQTT approach
Rys. 3. Schemat wymiany danych w systemach wykorzystujących MQTT
Below, the brief definitions of all QoS levels is presented:

- **Level 0**: The broker/client delivers the message without confirmation,
- **Level 1**: The broker/client delivers the message with confirmation,
- **Level 2**: The broker/client delivers the message exactly once by using a four steps handshake.

Popularity of the MQTT protocol caused that for out tests, we have chosen it as an exemplar implementation of IoT application. The MQTT broker has been developed based on the Eclipse Mosquito server (version 1.4.10). This server is the open source (EPL/EDL licensed) implementation of the MQTT (MQTT protocol versions 3.1 and 3.1.1). The same version of this server was used in the Docker container. The Dockerfile had been written based on the official DockerHub implementation. The Docker environment was in the latest, official version (17.04.0-ce, build 4845c56) for Raspbian (kernel 4.4.50-v7+).

In addition to the two server implementations, the test environment contained a package responsible for simulating MQTT client collaboration with the MQTT broker. We used the mqtt-malaria solution [12]. The mqtt-malaria is a set of tools developed to help with testing the scalability and load behavior of MQTT protocol.

### 4. Evaluation of server implementations

We used mqtt-malaria as the publishers (multiple, separated sources of MQTT messages). All test scenarios were repeated for both implementations of the MQTT server. The number of publishers (MQTT sources) was from 1 up to 90. The sources had been publishing three different number of messages (denoted as M). We tested values of M, 100, 200 and 300, respectively. Moreover, three sizes of messages were applied for all sources. These were approximately (size of messages to send was gaussian at (x, x/20) [12]: 100, 1000, and 10000 bytes, respectively. The description of the scenarios is as follows:

- **Scenario 1** - QoS Level 0, small and medium loads (Fig. 4),
- **Scenario 2** - QoS Level 1, small and medium loads (Fig. 5),
- **Scenario 3** - QoS Level 2, small and medium loads (Fig. 6),
- **Scenario 4** - all QoS Levels, heavy load (Fig. 7).

The scenarios were repeated 20 times. The presented results are the average values in two meanings. Firstly, in each single test, the results are average for all used publishers. Secondly, the final results, presented in the following figures are the averages for all 20 repeats.
The test scenario 1 did not saturate the interface of the MQTT broker. The sources were publishing messages as fast as they could. The results presented in figure 4 show that for relatively small number of clients (in our case, below 50), there is no degradation in MQTT broker operation. However, with the increase of the message size, the stability of messages processing becomes worse.

Fig. 4. Results for test scenario 1
Rys. 4. Wyniki dla pierwszego scenariusza testów

Fig. 5. Results for test scenario 2
Rys. 5. Wyniki dla drugiego scenariusza testów
Particularly the behavior of the MQTT broker on the Docker container has demonstrated the significant performance differences between the individual measurements. Comparing the two server implementations, the highest difference of number of messages processed per second (within a single test) occurred for 60 publishers and it was equal to 21.3% on the favour of native implementation (Raspberry Pi). Same trend was the case for all 20 tests. The distribution of average number of messages processed per second was smaller but still significant. The biggest difference was recorded for 80 publishers and it was equal to 16.4%. However, comparing result obtained during the scenario 1 (Fig. 4) and scenario 2 (Fig. 5), the described above degradation is absent for the QoS Level 1.

The third scenario utilized the QoS Level 2 thus a four steps handshake was applied to the message exchange procedure. Obviously, it caused the decrease in the number of messages processed per second as it is shown in Fig. 6. It can be also observed the performance degradation of implementation based on the Docker container. This degradation is especially strong for higher values of message sizes. Moreover, it occurred for relatively low number of publishers (in our case it was 20). It is worth noting, however, that this phenomenon is seen for higher numbers of messages published by single source. According to presented results (Fig. 6), this performance degradation was recorded in case of more than 100 messages per source.

The fourth scenario was conducted in order to evaluate the performance of both implementations of MQTT broker under heavy loads. The messages size has been increased to the value of 10000 bytes. In general, this scenario confirmed the previously observed
A performance evaluation of Docker-based MQTT server…

(in previous tests) behavior of the Docker container in comparison to the direct implementation of the MQTT broker. Again, the greatest instability of message processing was observed for QoS Level 0, as it is presented in Fig. 7. The distribution of obtained numbers of messages per second was comparable to the results for Scenario 1.

Fig. 7. Results for test scenario 4
Rys. 7. Wyniki dla czwartego scenariusza testów

The new element in scenario 4 is that heavy loads also degraded the performance in case of QoS Level 1 and relatively high number of publishers (above 50).

5. Conclusion

The evaluation of two implementations of the MQTT broker running on top of the Raspberry Pi 2, has shown that impact of the container-based virtualization layer is not negligible. For relatively small number of messages (or publishers) which in our test was about 1000 messages per seconds, the performance of containerized server follows the performance of the native implementation of the server. The increase of number of messages per second affects the deterioration of stability of messages processing. Our tests proved that in this case, special attention should be paid to the QoS Level 0 transmissions. The biggest
impact on the performance of the MQTT server was associated with messages size. Unfortunately, heavy loads made worse the performance of both tested implementation. By analyzing all the obtained results, it can be stated that generally, the difference between tested implementations of the MQTT broker is not significant enough to exclude the interest in the idea of microservices in the case of development of distributed applications. Also, the presented results allow to state that Docker containers can be viable platform for effective implementation of many components of IoT systems.

BIBLIOGRAPHY

Omówienie

Artykuł przedstawia porównanie dwóch wdrożeń brokera protokołu Message Queuing Telemetry Transport (MQTT) na popularnej platformie komputerów jednoukładowych Raspberry Pi 2B. Celem tego porównania jest ocena wpływu mechanizmów lekkiej wirtualizacji na funkcjonowanie elementów typowych systemów Internetu Rzeczy (IoT). W trakcie testów przedstawicielem lekkiej wirtualizacji był kontener Docker. Ilustrację podstawowych różnic pomiędzy zasadą realizacji ciężkiej wirtualizacji (opartej na hyperwizorze) oraz na zasadzie konteneryzacji przedstawiono na rysunku 1.

Dzięki wykorzystaniu kontenerów Docker możliwe jest opracowywanie i wdrażanie szerokiej gamy aplikacji rozproszonych według idei mikrousług. Idea ta pozwala nie tylko na szybsze tworzenie złożonych systemów informatycznych, ale również daje możliwość elastycznego skalowania systemu w zależności od specyfiki jego zastosowania. Koncepcja skalowania na bazie kontenerów jest przedstawiona na rysunku 2.

Zgodnie z założeniem przedstawionym na wstępie, badanym elementem systemu IoT był broker MQTT. Protokół MQTT jest obecnie bardzo popularną metodą realizacji wymiany danych pomiędzy źródłami danych, np. czujnikami. Podstawowy schemat wymiany danych według tego protokołu przedstawiony jest na rysunku 3. W trakcie testów wykorzystywane były wszystkie trzy poziomy Quality of Service (QoS), które omówiono w rozdziale 3.

Zasady przeprowadzenia testów oraz wykorzystane narzędzia informatyczne zostały przedstawione w rozdziale 3. Wszystkie testy podzielone zostały na 4 scenariusze. Uzyskane wyniki są przedstawione na rysunkach 4-7, kolejno dla poszczególnych scenariuszy testów. Na ich podstawie można stwierdzić, że wpływ lekkiej wirtualizacji jest zauważalny, lecz z punktu widzenia wydajności brokera MQTT, szczególnie dla relatywnie małych i średnich obciążeń, pozwala traktować kontenery Docker oraz idee mikrousług jako bardzo obiecującą metodę projektowania i wdrażania rozproszonych systemów IoT.

Addresses

Sławomir PRZYŁUCKI: Lublin University of Technology, Institute of Informatics, ul. Nadbystrzycka 36B, 20-618 Lublin, Poland, s.przylucki@pollub.pl
Dariusz CZERWIŃSKI: Lublin University of Technology, Institute of Informatics, ul. Nadbystrzycka 36B, 20-618 Lublin, Poland, d.czerwinski@pollub.pl
Artur SIERSZEŃ: Łódź University of Technology, Institute of Applied Computer Science, ul. Stefanowskiego 18/22, 90-924 Łódź, Poland, artur.sierszen@p.lodz.pl