MARKA: A Microservice Architecture-Based Application Performance Comparison Between Docker Swarm and Kubernetes

Tuğba Günaydın
Yıldız Technical University
Computer Engineering
İstanbul, Turkey
E-mail: tugba.gunaydin@std.yildiz.edu.tr

Göker Cebeci
Yıldız Technical University
Computer Engineering
İstanbul, Turkey
E-mail: goker.cebeci@std.yildiz.edu.tr

Özgün Subaşı
Integrated Finance
London, United Kingdom
E-mail: ozgun.subasi@integrated.finance

Abstract—**Container-based distributed programming techniques are used to make applications effective and scalable. Microservice architecture is an approach that has been on the rise among software developers in recent years. This paper presents a case study comparing the performance of two commonly used container orchestrators, Docker Swarm and Kubernetes, over a Web application developed by using the microservices architecture. We compare the performances of Docker Swarm and Kubernetes under load by increasing the number of users. The aim of this study is to give an idea to researchers and practitioners about the performances of Docker Swarm and Kubernetes in applications developed in the proposed microservice architecture. The Web application developed by the authors is a kind of loyalty application, that is to say, it gives a free item in exchange for a certain number of purchased items. With this study, it was concluded that the Docker Swarm is more efficient as the number of users increases compared to Kubernetes.**

Keywords–Microservice Architecture; Performance Evaluation; Docker Swarm; Kubernetes; JMeter.

I. INTRODUCTION

With the microservice architecture, applications are developed that are very flexible and scalable. In the microservice architecture approach, the application is split up into its smallest functions; each function is dedicated for one job only, and it is called as microservice. Microservices are put in packages that are called containers that provide everything necessary for running [1]. Microservices are difficult to operate because they are distributed [2]. Container-based technologies are used to orchestrate microservices. Two technologies stand out in orchestrating microservices: Docker Swarm and Kubernetes [3].

The proposed prototype is a loyalty application. Today, it is very important to gain new customers and retain existing customers for restaurants and cafes. For this reason, loyalty applications are used. A product is offered to the customer free of charge for a certain amount of purchased product. With this study, the concept of loyalty application was realized with the microservice architecture approach.

This paper presents a performance comparison of Docker Swarm and Kubernetes on a microservice architecture-based Web application. As the number of users increased, the time to complete the test scenario of Docker Swarm and Kubernetes was compared. The study can be classified under three main titles: (1) Proposed Software Architecture and Application (software architecture, approaches and application used for implementation), (2) Test Scenario (scenario used to test the system, Docker Swarm and Kubernetes) and (3) Experimental Setup (load tests for orchestration tools). The rest of the paper is structured as follows. In Section II, we present the relevant studies in the literature. In Section III, our developed application and the microservice architecture used are discussed in detail. The test scenario simulating the operation of the application in real life is explained in Section IV. The experimental environment and parameters are shown in Section V. Finally, in Section IV, the results obtained are discussed and we conclude our work.

II. LITERATURE REVIEW

With the increasing importance of scalability in recent years, microservice architectures have become popular [4]. Microservices are used more effectively with container technology. There are different application approaches of container technology and performance evaluation studies of these approaches [5]. We compared performances of Docker Swarm and Kubernetes with an application using microservice architecture model.

Using scaling and microservice architecture approach studies of frequently used orchestrators, the appropriate orchestrator can be selected for a certain application [6]. In this comparison, the effect of more complex applications on the performance of orchestrators is clearly shown [7]. Studies have shown that cloud-based approaches are more performant and flexible than traditional approaches for developing increasingly complex applications [8]. We made a performance comparison by revealing the microservice architecture we use in our application.

There exist many studies focusing on designing and implementing traditional monolithic Web service based Service Oriented Architecture (SOA) systems with a focus on high performance [9]-[10]. However, in this particular study, our main focus is the use of microservices in creating SOA based systems with a focus on load balancing amongst the nodes. To enable the load balancing functionality, we utilize technologies like Docker Swarm and Kubernetes. In this study, the performance was compared by using the loyalty application MARKA in 3 scenarios: without an orchestrator, using Docker Swarm, and using Kubernetes.

Copyright (c) IARIA, 2020. **ISBN: 978-1-61208-827-3**

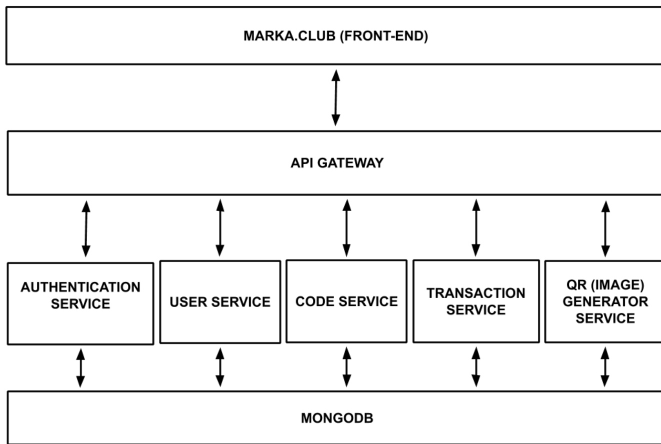

Figure 1. System Architecture Model and Network.

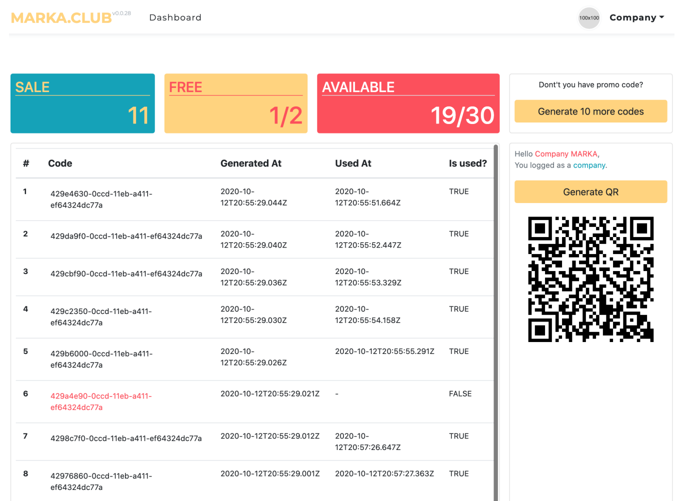

Figure 2. Marka.club Front-End Screen Company Dashboard

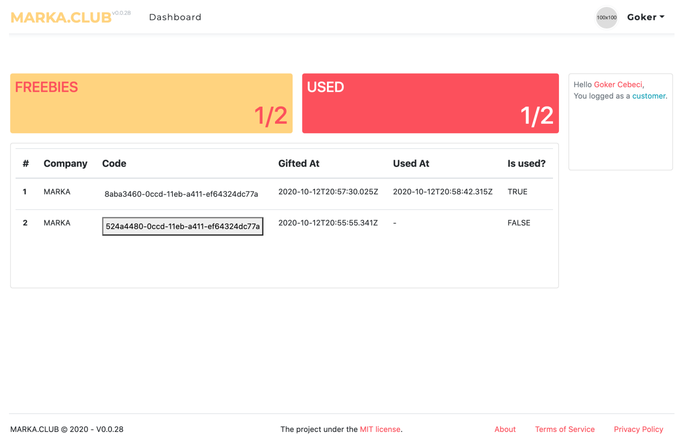

Figure 3. Marka.club Front-End Screen Customer Dashboard

III. Proposed Software Architecture and Application

The Web application MARKA [11] has six microservices: API (Application Programming Interface) Gateway, Authentication Service, User Service, Code Service, Transaction Service, and QR (Quick Response) (Image) Generator Service. It also has a front-end for user control screen and a database for the management of data. The system architecture model can be seen in the Figure 1.

Microservices communicate with each other over the HTTP (Hyper-Text Transfer Protocol). For instance, in order to create a new transaction, the transaction service receives the information of the received code from the code service first, and then it gets the information of the user associated this code from the user service. If the user and the owner of the code are the same, it generates an error.This is because the user who created the code and the user using it cannot be the same. Then, the code service compares the company identification number associated with the code from the database and determines whether the code is a purchased item or a free item. When these jobs are completed, the transaction service creates the transaction. It calls the code service and updates the code as used. If the code is for the purchased item, it also receives the free item quantity information of the company from the user service (information on how many products will be given free of charge in sales). After getting the relevant company and the number of transactions, it calculates how many free items the user has won from the code service. From this, it calculates whether the user earns free items with the final purchased product(s). If the user has won free items, the code service generates as many codes as there are free items.

A. Marka.Club (Front-End)

Front-end (Marka.club) is a software provided for the end users to perform their operations. It enables the user to create an account, log into the system, create codes and use codes. End-user interactions are created here. An example of a company dashboard can be seen in Figure 2 and an example of a customer dashboard can be seen in Figure 3.

Customers buy the product and read the QR code produced by the restaurant. When they purchase the amount of product determined by the restaurant, they have the right to get one free product, so they produce a QR code. The restaurant reads the QR code produced by the customer and gives one free product to the customer.

B. API Gateway

The API gateway is the microservice that ensures that the incoming request is directed to the responsible microservice.

C. Authentication Service

The authentication service is a login and register service. If a user is not yet registered in the system, first, they get registered into the system. There are two roles for registration: customer or company. After users register, they can log in.

D. User Service

The user service is the service that keeps the user information such as e-mail, first and last names, company-customer roles, etc.

E. Code Service

The code service is a microservice that generates the codes that the user will use in another role. For example, if the user's

Copyright (c) IARIA, 2020. ISBN: 978-1-61208-827-3

role is the company, it generates the codes that will be used by the customer when purchasing an item. If the user's role is the customer, he/she generates the codes that will be used by the company to get free items from the company. In other words, when the customer earns a free item, a code is generated for that free item and this code is used by the company. If it is a valid code, the company gives a free item to the customer.

F. Transaction Service

The transaction service keeps the code transaction information, such as which role produced the code, which role was used, and how many codes were produced and used.

G. QR (Image) Generator Service

The QR generator service converts the codes generated to the image file (QR) to be used on mobile devices. This feature will be used when the application is used on mobile devices.

H. Database

A database has been created in which all transactions and information are kept. MongoDB [12] was used as the database. A single database has been created for proof of concept, but each microservice uses its own collections that they are responsible for, and they do not interfere with other areas of responsibility.

IV. TEST SCENARIO

A test scenario was prepared to compare the container orchestration platforms' behavior under load. In this study, Docker Swarm and Kubernetes were used as container orchestration tools. Docker Swarm [13] is developed by Docker Engine. Kubernetes [14] is developed by Google. The responses of the platforms were measured according to the test scenario, depending on the request per unit time. The flow diagram of the test scenario is as shown in Figure 4.

The test scenario was created by simulating the real-time operation of the application: a company signs up for the system, then logs into the system. Anyone who does not exist in the system, be it a company or a customer, must sign up in order to log into the system. By signing into the system, a new user is created each time. The company generates codes for the items to sell. The codes generated by the company are saved into the database because the customers will use them automatically. The customer signs up for the system and then signs into the system. The customer will use all the unused codes stored in the database; we make the customers use all the codes generated to simulate a real-life load scenario for our load testing. The customer generates free codes in return for a certain number of codes used (as initially determined by the company). The unused codes generated by the customer are received and saved into the database. The company signs into the system and uses all of the unused free codes generated by the customer; with this, the customer takes the free items.

V. EXPERIMENTAL SETUP

The test procedure of the application was run in 3 different ways: test without orchestrator, which means without any container (except mongoDB); Docker Swarm test; and Kubernetes test. The application is run with Docker Swarm and Kubernetes on Docker Desktop.

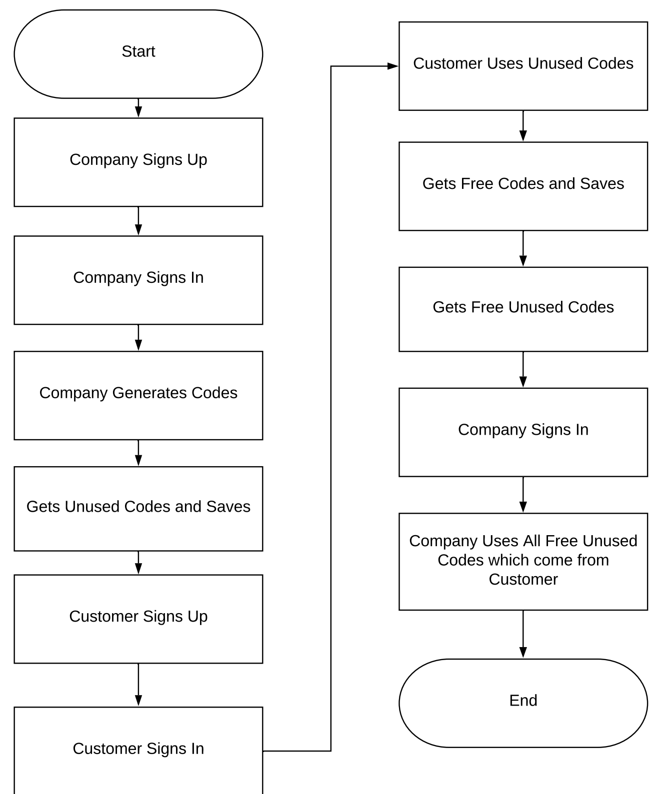

Figure 4. API Test Scenario Flow Chart

JMeter [15] is a tool for load testing. It is used to test the application against real-life situations. There are three basic parameters that should be in a JMeter test plan: Thread Group, Samplers and Listeners [16]. The thread group decides how many threads there will be and for how long each thread will be active. Samplers are for request types such as FTP (File Transfer Protocol) requests, HTTP requests, JDBC (Java Database Connectivity) requests etc. Listeners are used for the visualization of results in the form of a graph, table, etc.

There is a bar graph named "aggregate" in JMeter. The aggregate graph shows the average of the response time for each request in the test. We compared the average response times of requests by the number of users via aggregate graphs for each one of our tests.

Test Without Orchestrator, Docker Swarm and Kubernetes were compared with 10, 20 and 50 users as response times; Docker Swarm and Kubernetes were compared with 100, 200, 400 and 500 users as response times.

The letters on the charts are as follows:

- A: Company Sign Up,
- B: Company Sign In,
- C: Get user info,
- D: Generate codes,
- E: Get codes,
- F: Customer Sign Up,
- G: Customer Sign In,

Copyright (c) IARIA, 2020. ISBN: 978-1-61208-827-3

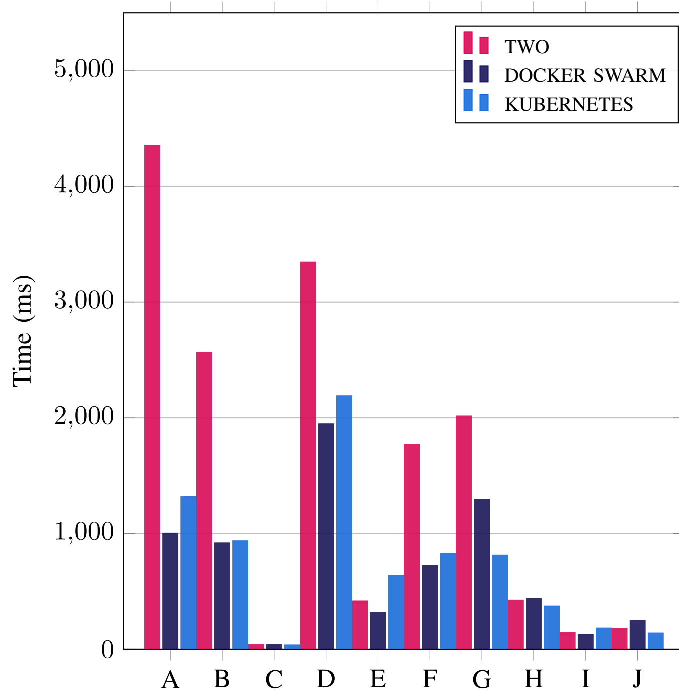

Figure 5. Comparison of Average Response Time with 10 threads

- H: Customer uses codes,
- I: Get gifts,
- J: Use gifts.

The average response times of each request in the test are shown on the aggregate graph with 10 users in Figure 5, 20 users in Figure 6, 50 users in Figure 7, 100 users in Figure 8, 200 users in Figure 9, 400 users in Figure 10 and, finally, 500 users in Figure 11.

A. *Test Without Orchestrator (TWO)*

For TWO, all services are used locally, without using any container or orchestration tool. The local system information is as follows: the Operating System is Windows 10 Home, the RAM is 8 GB, the Processor is Intel(R) Core(TM) i5-8250U CPU 1.60 GHz 1.80 GHz, and the System Type is 64-bit OS, x64-based processor. The JMeter settings for the number of threads (users) are 10, 20 and 50, the rump-up period (in seconds) is 0 and the loop count is 1.

Below are the times taken for each the test scenario to complete, based on the number of users:

- 10 users: 1 minute and 9 seconds,
- 20 users: 1 minute and 43 seconds,
- 50 users: 3 minutes and 40 seconds.

When we performed the test with 100 threads, some threads started timing out, so the test was not completed.

B. *Docker Swarm Test*

For Docker Swarm test, the application is using the Docker Desktop and Docker Swarm as orchestrator. Docker Swarm worked with 3 replicas. The local system information is the same as in the case of TWO. The JMeter settings for the

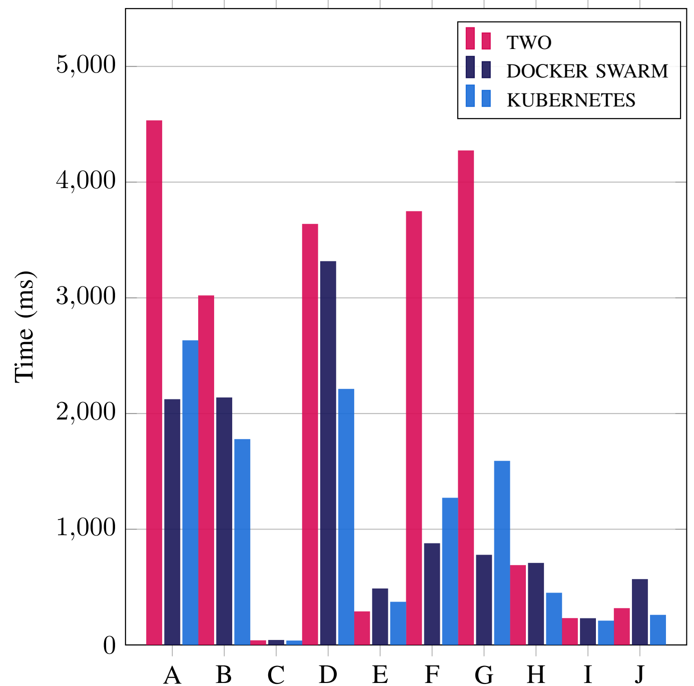

Figure 6. Comparison of Average Response Time with 20 threads

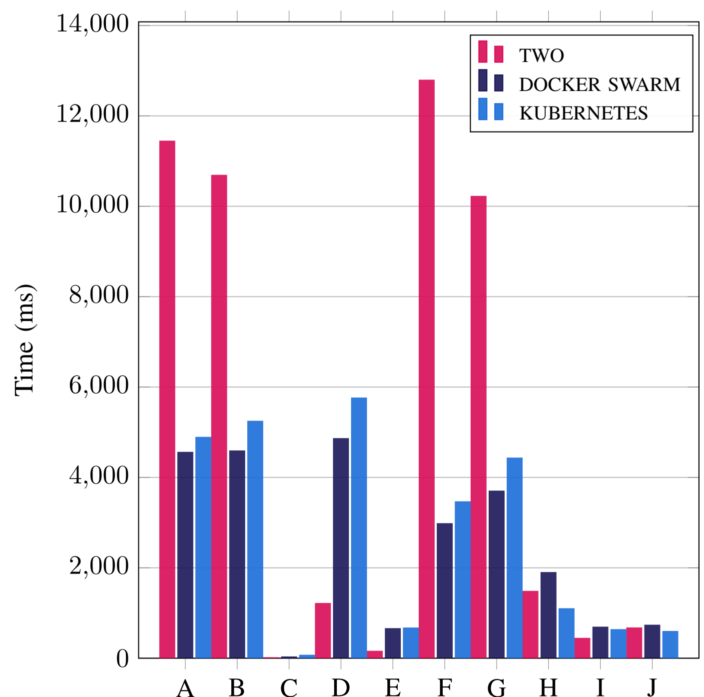

Figure 7. Comparison of Average Response Time with 50 threads

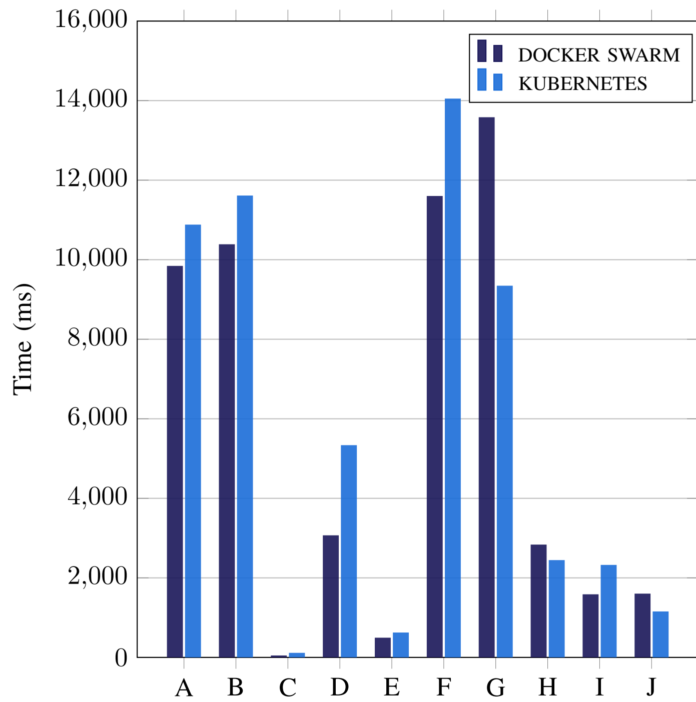

Figure 8. Comparison of Average Response Time with 100 threads

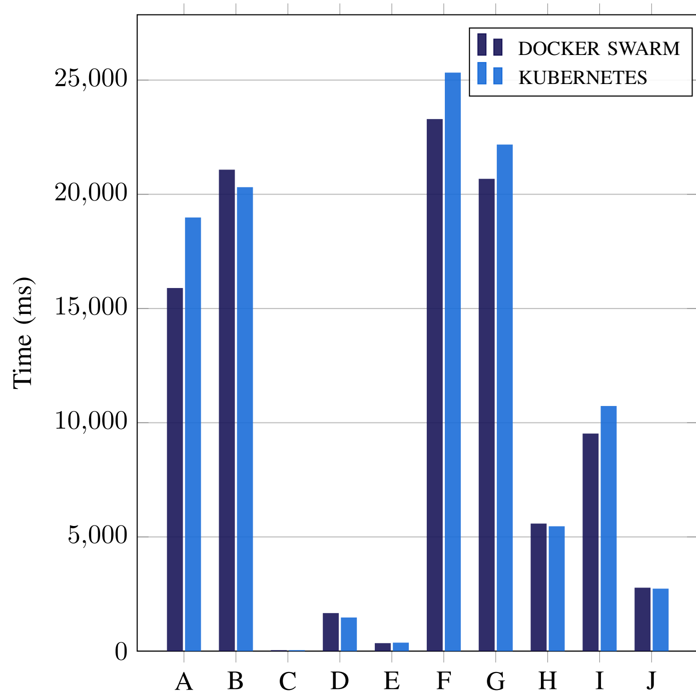

Figure 9. Comparison of Average Response Time with 200 threads

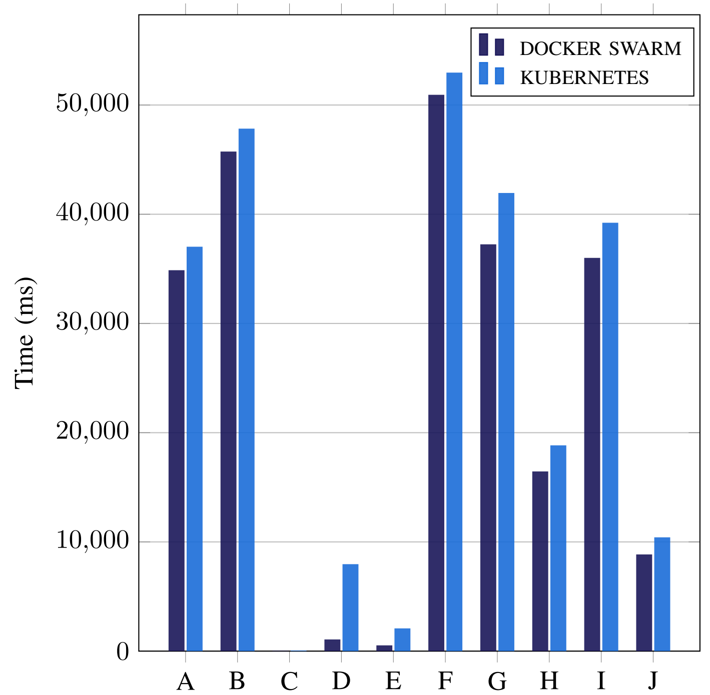

Figure 10. Comparison of Average Response Time with 400 threads

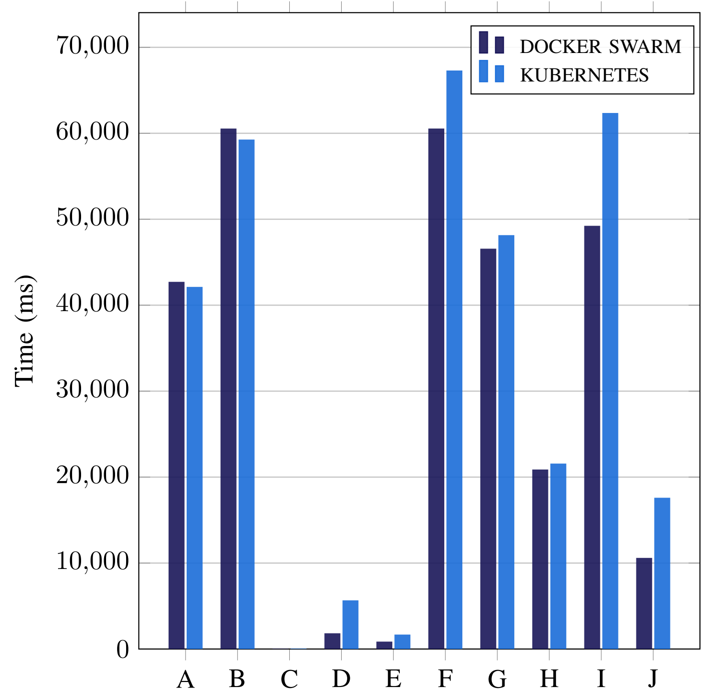

Figure 11. Comparison of Average Response Time with 500 threads

Copyright (c) IARIA, 2020. ISBN: 978-1-61208-827-3

number of threads (users) are 10, 20, 50, 100, 200 and 400, the rump-up period (in seconds) is 0 and, finally, the loop count is 1.

Below are the times taken for each the test scenario to complete, based on the number of users:

- 10 users: 52 seconds,
- 20 users: 1 minute 42 seconds,
- 50 users: 4 minutes 7 seconds,
- 100 users: 6 minutes 53 seconds,
- 200 users: 12 minutes 18 seconds,
- 400 users: 34 minutes 51 seconds,
- 500 users: 44 minutes 16 seconds.

When we performed the test with 1000 threads, some threads started timing out, so the test was not completed.

C. Kubernetes Test

For Kubernetes Test, the application stand-up with using Docker Desktop and Kubernetes as orchestrator. Kubernetes worked with 3 replicas. Local system information is same with TWO. The JMeter settings for the number of threads (users) are 10, 20, 50, 100, 200 and 400, the rump-up period (in seconds) is 0 and, finally, the loop count is 1.

Below are the times taken for each the test scenario to complete, based on the number of users:

- 10 users: 51 seconds,
- 20 users: 1 minute 6 seconds,
- 50 users: 2 minutes 36 seconds,
- 100 users: 5 minutes 40 seconds,
- 200 users: 12 minutes 20 seconds,
- 400 users: 40 minutes 25 seconds,
- 500 users: 48 minutes 33 seconds.

When we performed the test with 1000 threads, some threads started timing out, so the test was not completed.

VI. CONCLUSION

The performances of Docker Swarm and Kubernetes under load were compared via an application. A conclusion has been reached regarding the performances of Docker Swarm and Kubernetes in the architecture described in this study.

The test we did without using an orchestrator (TWO) could not handle the load in more than 50 threads. Thus, it is clearly seen that it is not efficient as the load on the application increases. The application could not respond to high loads.

Although Docker Swarm takes longer time in tests with fewer users, when the number of users increased, it was completed in a shorter time than Kubernetes. The ability of Docker Swarm and Kubernetes to be scalable in load tests has not been tested in this study.

Considering the architecture of the application and the number of microservices, we can say that its complexity is low. For this reason, as the number of users increases, we see that the Docker Swarm test yields better results than Kubernetes and also completes in a shorter time.

In this study, 3 replicas were used in Kubernetes and Docker Swarm. As the number of incoming requests increases, the automated replica creation capabilities test will be discussed in a future study.

ACKNOWLEDGMENT

We would like to thank Associate Professor Mehmet Sıddık AKTAŞ (affiliation: Yıldız Technical University, Computer Engineering, İstanbul, TURKEY) for his valuable contributions and inspiring guidance.

REFERENCES

[1] A. Modak, S. Chaudhary, P. Paygude, and S. Ldate, “Techniques to secure data on cloud: Docker swarm or kubernetes?” in 2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT). IEEE, 2018, pp. 7–12.

[2] R. Heinrich et al., “Performance engineering for microservices: research challenges and directions,” in Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, 2017, pp. 223–226.

[3] N. Marathe, A. Gandhi, and J. M. Shah, “Docker swarm and kubernetes in cloud computing environment,” in 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI). IEEE, 2019, pp. 179–184.

[4] N. Alshuqayran, N. Ali, and R. Evans, “A systematic mapping study in microservice architecture,” in 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), 2016, pp. 44–51.

[5] M. Amaral et al., “Performance evaluation of microservices architectures using containers,” in 2015 IEEE 14th International Symposium on Network Computing and Applications, 2015, pp. 27–34.

[6] L. Mercl and J. Pavlik, “The comparison of container orchestrators,” in Third International Congress on Information and Communication Technology, X.-S. Yang, S. Sherratt, N. Dey, and A. Joshi, Eds. Singapore: Springer Singapore, 2019, pp. 677–685.

[7] Y. Pan, I. Chen, F. Brasileiro, G. Jayaputera, and R. Sinnott, “A performance comparison of cloud-based container orchestration tools,” in 2019 IEEE International Conference on Big Knowledge (ICBK), Nov 2019, pp. 191–198.

[8] W. Li and A. Kanso, “Comparing containers versus virtual machines for achieving high availability,” in 2015 IEEE International Conference on Cloud Engineering, 2015, pp. 353–358.

[9] G. C. Fox et al., “Real time streaming data grid applications,” in Distributed Cooperative Laboratories: Networking, Instrumentation, and Measurements. Springer, 2006, pp. 253–267.

[10] M. Aktas et al., “iservo: Implementing the international solid earth research virtual observatory by integrating computational grid and geographical information web services,” in Computational Earthquake Physics: Simulations, Analysis and Infrastructure, Part II. Springer, 2006, pp. 2281–2296.

[11] Marka.club. [accessed Oct. 2020]. [Online]. Available: https://github.com/kodkafa/marka.club

[12] Mongodb. [accessed Oct. 2020]. [Online]. Available: https://www.mongodb.com/

[13] Docker swarm. [accessed Oct. 2020]. [Online]. Available: https://docs.docker.com/engine/swarm/

[14] Kubernetes. [accessed Oct. 2020]. [Online]. Available: https://kubernetes.io/

[15] Jmeter. [accessed Oct. 2020]. [Online]. Available: https://jmeter.apache.org/

[16] D. Nevedrov, “Using jmeter to performance test web services,” Published on dev2dev, 2006, pp. 1–11.

Copyright (c) IARIA, 2020. ISBN: 978-1-61208-827-3