INTERNATIONAL JOURNAL ON INFORMATICS VISUALIZATION

journal homepage : www.joiv.org/index.php/joiv

A Study of Database Connection Pool in Microservice Architecture

Nur Ayuni Nor Sobri[a], Mohamad Aqib Haqmi Abas[a], Ihsan Mohd Yassin[b,*], Megat Syahirul Amin Megat Ali[b], Nooritawati Md Tahir[c], Azlee Zabidi[d], Zairi Ismael Rizman[e]

[a] *School of Electrical Engineering, College of Engineering, Universiti Teknologi MARA, 40450 Shah Alam, Selangor, Malaysia*
[b] *Microwave Research Institute, Universiti Teknologi MARA, 40450 Shah Alam, Selangor, Malaysia*
[c] *Research Nexus UiTM, Universiti Teknologi MARA, 40450 Shah Alam, Selangor, Malaysia*
[d] *Faculty of Systems & Software Engineering, College of Computing & Applied Sciences, Universiti Malaysia Pahang, Pahang, Malaysia*
[e] *School of Electrical Engineering, College of Engineering, Universiti Teknologi MARA, 23000 Dungun, Terengganu, Malaysia*

*Corresponding author: *ihsan.yassin@gmail.com*

***Abstract*—The growing number of the Internet presents a higher requirement for backend application systems to be designed to handle thousands of user traffic concurrently. Microservice architecture is also in a rising trend which allows each service to scale horizontally by its throughput, and load helps scale the system efficiently without waste of resources like in the traditional monolithic application system. Among the many strategies to optimize delivery, the database connection pool helps backend systems to access databases efficiently by reusing database connections, thus eliminating the computationally expensive need to open and close connections with new requests. Additionally, database connection pools can help improve applications' connection reliability. This paper aims to determine the most suitable maximum amount of database connections in a microservice setting, where multiple instances of the service are used for scalability and high availability purposes of the system. To tackle the scalability issue and achieve the high availability of our services, we propose running multiple instances of each of our services in production, especially for services that we anticipate would be hit the most during runtime. This allows load balancing of request load between multiple instances and having backup instances to serve HTTP requests when one of the instances is down. The result obtained in this experiment shows that five database connections give the best result in microservice settings as described in our methodology.**

***Keywords*— Microservice; backend application system; database connection pool.**

Manuscript received 10 Dec. 2021; revised 18 Jan. 2022; accepted 21 Apr. 2022. Date of publication 31 Aug. 2022.

I. INTRODUCTION

In the past few years in the cloud services domain, companies have moved from monolithic architecture applications to microservices architecture. The idea of breaking a complex monolithic application that serves the whole functionality in a single application to multiple loosely coupled and single-purpose started with big tech companies like Apple, Google, and Netflix [1]-[3] due to numerous advantages compared to the traditional monolithic architecture.

Scalability and flexibility are some of the most important advantages of microservice architecture [4]-[10]. The traditional approach to handling scalability is to increase the number of instances or the size of the whole monolithic application. Although increasing the number of instances of the application running can help to achieve high availability and fault tolerance, the default way to increase scalability is by increasing the size of the application as it is less complex. However, in monolithic architecture, this is very inefficient because, in most cases, only a few particular domains of services are expected to be used by many users and require high throughput.

In a microservice architecture, each service is loosely-coupled, serving a single purpose and independent from other services [2], [4], [7], [11]-[4]. Hence, this allows for deploying and scaling each service independently and using different policies from the other services [3], [4], [11], [15]-[17].

Most current backend application systems require an interaction between the application and database to store all user's data. Most legacy backend systems use a direct method to invoke a call to the database where the application would first create a database connection in the program, execute the SQL query to the database, and close the database connection [18-22]. However, as the application gets bigger and more complex, making connections to the database is inefficient as it would greatly increase the system overhead to create and close the connection frequently [19].

The new backend application system uses a database connection pool (DCP), where the application would help create and maintain the connections so they can be reused in the future. The general idea is that connections would be in either two states, whether it is being used or idle [23]-[25]. Each time there is a need to make a database request, the application would check if there is any idle connection for it to use. Else it would create a new connection for as long as it has not reached the maximum amount of connection threshold. Postgres databases, by default, have 100 'max_connections' limit, and if this limit is being hit under heavy load, the backend application would return an error to end users [23], [26].

Some backend applications allow the developer to choose their own configurations for managing the pool. This is often important as each application would have its own different requirements. Generally, a medium-sized monolithic application would usually opt for the default amount of maximum database connections (100 with Postgres database). Having a too high amount of maximum connections can also cause problems as it can overwhelm the database and application system, requiring a larger amount of memory (RAM) for maintaining the connections and high overhead in terms of CPU cycle and RAM for setting up and closing the connection.

Generally, choosing the configuration for the database connection pool, such as the maximum amount of connections and maximum idle time of connection, requires performance testing. It aims to ensure that the most suitable configuration is chosen for the backend application where it would not cause a bottleneck (too low amount of connections) and not waste the system's resources (too high amount of connections). Therefore, this paper aims to find the most suitable maximum amount of database connections in a microservice setting, where multiple instances of the service are used for scalability and high availability purposes of the system.

We have found some relevant journal articles related to our work using database connection pool in their respective research. In Al-Hawari et al. [27], the authors use a database connection pool to develop their Student Information System (SIS). This three-tier web application allows registrars to perform tasks involving system setup, admission, registration, graduation grades processing, and processing and report. The SIS system was developed using Java, and the authors set up a JDBC connection pool to solve the possible issue of scalability of the system.

A study of the database connection pool by Zhang et al. [18] shows a comparison between the traditional connection pool with tomcat, hibernate, and the new proposed connection pool. The result shown from the study shows how the differences in methods used in managing the connection pool directly affect the system's performance.

In both Huang et al. [28], [29], the authors study the security aspect of database connection pool in 3 tiers web systems. In [28], the authors use a formal model of 3 tiers web system, and a few security problems faced in the web system were found in the model. Few methods for solving the security issue were introduced and proposed, such as securing the application, terminal user tracing, and modifying the previous standard on securing the database connection pool. The database connection pool audit system (DCPAS) is proposed by Huang et al. [29] to trace the end user's identity and bind the user's operations to execute the SQL statements to the database. The proposed DCPAS allows for a better security audit, as the admin can trace the detailed SQL statements if an illegal user makes an SQL injection to the system.

II. Material and Method

To tackle the scalability issue and achieve the high availability of our services, we propose running multiple instances of each of our services in production, especially for services that we anticipate would be hit the most during runtime. This allows load balancing of request load between multiple instances and having backup instances to serve HTTP requests when one of the instances is down.

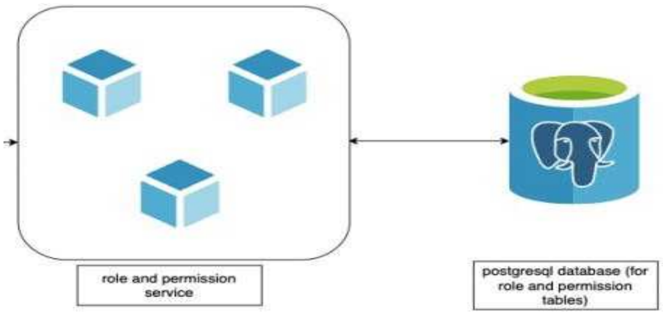
role and permission service
postgresql database (for role and permission tables)

Fig. 1 Multiple instances for role and permission service

Fig. 1 shows the example of a single service with the proposed architecture to run in the production server, where three instances are running for the role and permission service. This is only one small service out of multiple other services we run on the production server. The service in Fig. 1 handles only the roles and permissions information for the system. Any request that requires the roles/permissions logic from the API gateway would be delegated to this service. Each service instance would connect to the Postgres database with the roles table and permissions table. However, the microservice architecture is flexible and does not set any hard requirements for database setup. In the production system, we can set up the database on the same server, set up the database on a different server, or opt for managed database services that most cloud providers offer. However, accessing a database on a different server or managed service would have an increased network latency due to the request calls needing to be made to an external server instead of accessing a database in a different port on the same server.

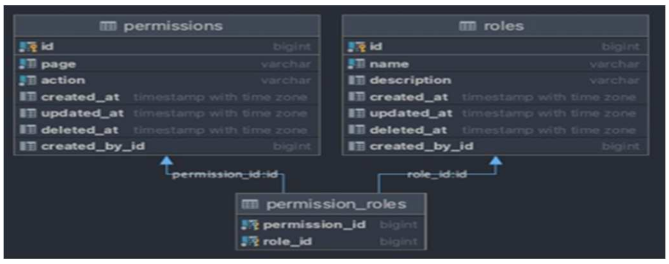
permissions
id bigint
page varchar
action varchar
created_at timestamp with time zone
updated_at timestamp with time zone
deleted_at timestamp with time zone
created_by_id bigint
roles
id bigint
name varchar
description varchar
created_at timestamp with time zone
updated_at timestamp with time zone
deleted_at timestamp with time zone
created_by_id bigint
permission_id:id
role_id:id
permission_roles
permission_id bigint
role_id bigint

Fig. 2 Roles and Permission table

Fig. 2 shows the roles and permissions table with its intermediary many-to-many table. Our system uses a role-based access control (RBAC) authorization model. In RBAC, users have access to an object/page/module based on their respective assigned roles in the system [30]. Roles are commonly assigned based on job function, and permissions are defined based on job authority and job responsibility.

To find the most suitable maximum amount of database connections, we run the performance testing on this service with two different scenarios; first, with a single instance, as shown in Fig. 3; and second, with three instances and a load balancer, as shown in Fig. 4.

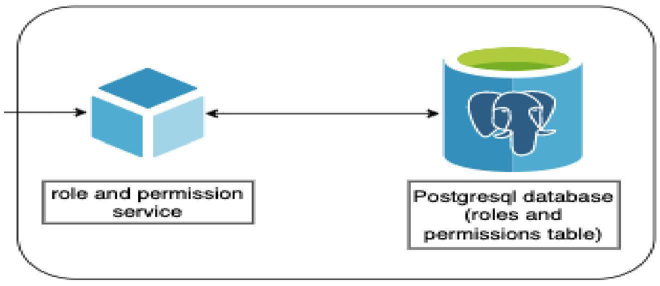
role and permission service
Postgresql database (roles and permissions table)

Fig. 3 Single instance for role and permission service

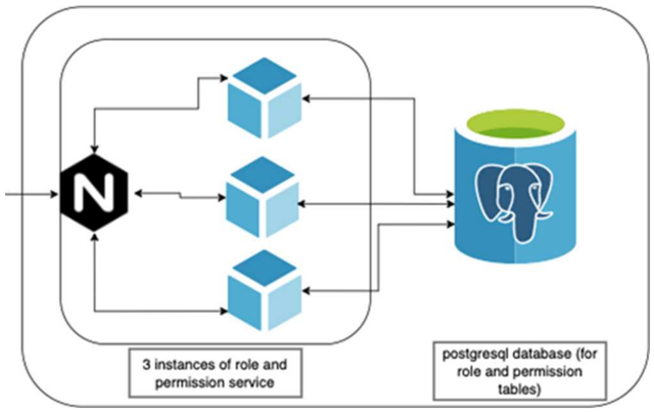
3 instances of role and permission service
postgresql database (for role and permission tables)

Fig. 4 3 instances for role and permission service with Nginx as load balancer

The load testing was done with Arm64v8 CPU architecture. The limitation of the platform applies to this project. We also limit the go runtime (for each instance) to use a single CPU (with GOMAXPROCS = 1) and 128MB of memory (ulimit). However, we found that neither limiting the CPU nor RAM affects our experiment as none of the tests would even hit the limit. However, the situation would be different in production servers when we deploy the services with a more limited amount of CPU cycles and RAM configuration for our machine. The benchmark performance testing would be done using the Vegeta load testing tool written in Go. In this test, we are using the default setting of the Postgres database as it would be in production without tuning any configuration. We also did not change any optimization being done by Postgres for similar SQL request calls either by its shared buffer cache or operating system cache method. The only manipulated variable for this experiment is the maximum number of connections and instances (for the two different scenarios); everything else would be similar throughout the test.

We would test on four different amounts of connections for the database connection pool, which are 1, 5, and 10. The load tester would make 500, 1000, and 2000 requests per second (rps) to the service. The load test would be done for 5 seconds for each test. Only one API endpoint would be tested for this experiment, which is the "/roles" endpoint that would give all the roles in the database table, including its permissions relation. The reason that role/permission services are chosen for this experiment is due to this service being one of the most used in the system. Multiple endpoints in the system require authorization checks on whether a specific user has the necessary role and permission to access the endpoint.

III. Result And Discussion

The results of the performance tests are as follows:

TABLE I
MAXIMUM LATENCY (IN MS) FOR DIFFERENT NUMBER OF HTTP REQUESTS PER SECOND MADE TO DIFFERENT NUMBER OF DATABASE CONNECTIONS WITH SINGLE INSTANCE SERVICE

Number of Connections	**Requests Per Second**		
	500	**1000**	**2000**
1	438.966ms	9387ms	19433ms
5	64.758ms	156.088ms	1658ms
10	24.278ms	119.675ms	499.172ms

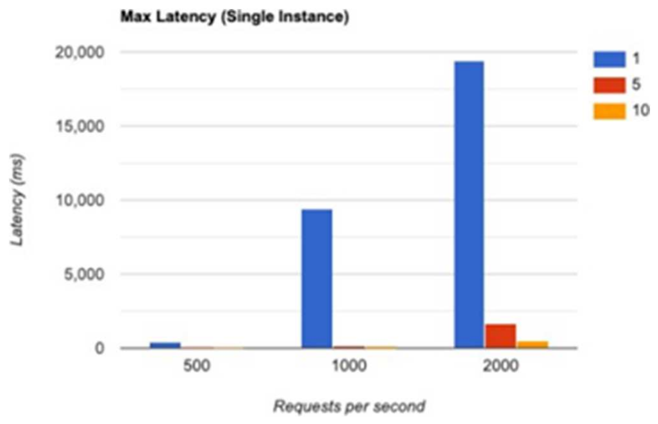

Fig. 5 Bar chart for maximum latency (in ms) for different number of HTTP requests per second made to different number of database connections with single instance service

Table 1 and Fig. 5 show the result of maximum latency for a different number of requests made to a different number of connections in a single instance service (as shown in Fig. 3). As seen, the max latency for 500 requests per second (rps) made for 1, 5 and 10 connections declines as the number of connections increase. For a single connection, the latency is at 438.966ms, then drops to 64.758ms when having five connections and 24.278ms for ten connections. For 1000 and 2000 HTTP rps, we can see that having a single database connection becomes a bottleneck to the service as it requires 9387ms and 19433ms, respectively. Note that this is without tuning any shared buffer cache or operating system level cache for the Postgres database default setting, which shows the latency struggle of having a single connection to the database. Meanwhile, for five connections, the service starts to bottleneck when having 2000 rps where it records 1658ms latency. For 1000 rps the service can still tolerate the throughput with 156.088ms latency. For 20 connections, the latency increases as the number of requests increase to 1000 and 2000 with 119.675ms and 499.172ms, respectively, but it is still bearable compared to having 5 and 10 connections.

TABLE II

99TH PERCENTILE LATENCY (IN MS) FOR DIFFERENT NUMBER OF HTTP REQUESTS PER SECOND MADE TO DIFFERENT NUMBER OF DATABASE CONNECTIONS WITH A SINGLE INSTANCE SERVICE

Number of	Requests Per Second		
Connections	500	1000	2000
1	203.918ms	7442ms	17749ms
5	19.166ms	55.95ms	426.78ms
10	4.741ms	35.356ms	205.427ms

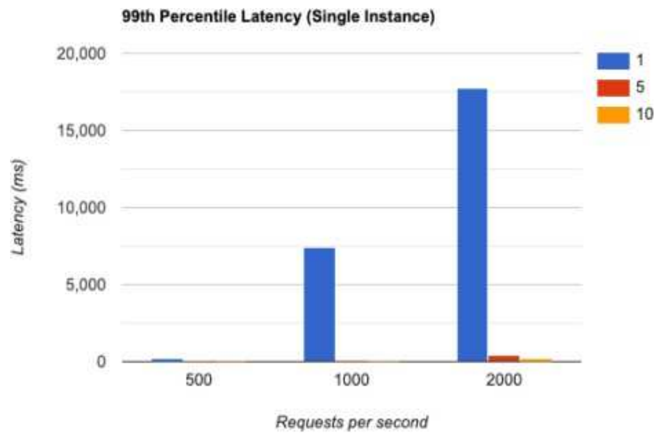

Fig. 6 Bar chart for maximum latency (in ms) for different number of HTTP requests per second made to different number of database connections with single instance service

Table 2 and Fig. 6 show the result of 99th percentile latency for different requests made to different numbers of connections in a single instance service. In some benchmark situations, this number is often used as a realistic measure of latency where 99 percent of end users would receive this latency, while maximum latency can show if there has been a sudden hiccup to a system (that might happen for a single request). As seen, the latency shows the same pattern as in the maximum latency result, where the latency decreases as the number of connections increases, and a single connection shows a bottleneck in performance in both 1000 and 2000 rps tests. For 500 rps, a single connection gives 203.918ms latency, followed by 5 connections with 19.166ms and 10 with 4.741ms. For 1000 rps, a single connection still shows a bottleneck result with a high number of 7442ms, followed by 55.95ms for five connections and 33.356ms for ten connections. For 2000 rps, we can see five connections start to show the bottleneck in performance as well with 426.78ms but it is far lower than 17749ms recorded by a single connection. 10 connections show a good performance of 205.427ms.

TABLE III

MAXIMUM LATENCY (IN MS) FOR DIFFERENT NUMBER OF HTTP REQUESTS PER SECOND MADE TO DIFFERENT NUMBER OF DATABASE CONNECTIONS WITH THREE INSTANCES SERVICE

Number of	Requests Per Second		
Connections	500	1000	2000
1	33.455ms	52.652ms	164.504ms
5	36.346ms	29.725ms	52.948ms
10	35.298ms	59.407ms	226.646ms

Table 3 and Fig. 7 show the result of maximum latency for a different number of requests made to different numbers of connections in a three-instance service (as shown in Fig. 4 in Section III).

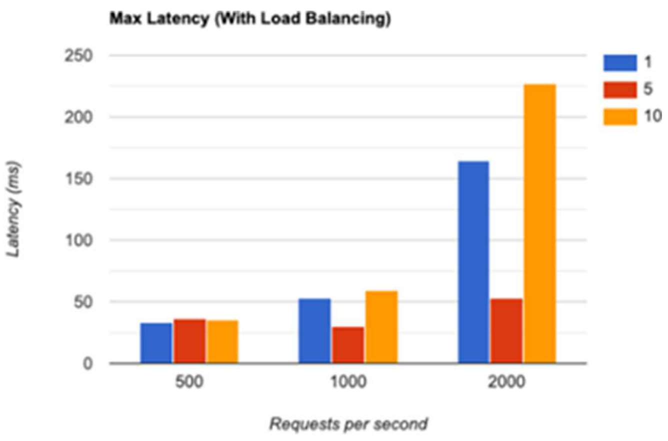

Fig. 7 Bar chart for maximum latency (in ms) for different number of HTTP requests per second made to different number of database connections with three instances service

We can see that even having a single connection, the service does not suffer the same performance impact as when having only a single instance of service. This shows that having multiple instances helps to balance the throughput load. For 500 rps, a single connection gives the best latency with 33.455ms, followed by ten connections with 35.298ms, and lastly, five with 36.346ms. For 1000 and 2000 rps, five connections show far better performance latency compared to single and ten connections. In 1000 rps result, five connections only recorded 29.725ms, better than their performance in 500 rps, followed by a single connection with 52.652ms and ten connections with 59.407ms. For 2000 rps, five connections record a low 52.948ms, followed by a single connection with 164.504ms and ten connections with 226.646ms.

TABLE IV

99TH PERCENTILE LATENCY (IN MS) FOR DIFFERENT NUMBER OF HTTP REQUESTS PER SECOND MADE TO DIFFERENT NUMBER OF DATABASE CONNECTIONS WITH THREE INSTANCES SERVICE

Number of	Requests Per Second		
Connections	500	1000	2000
1	7.496ms	14.672ms	54.697ms
5	6.227ms	13.279ms	18.513ms
10	7.111ms	18.173ms	55.855ms

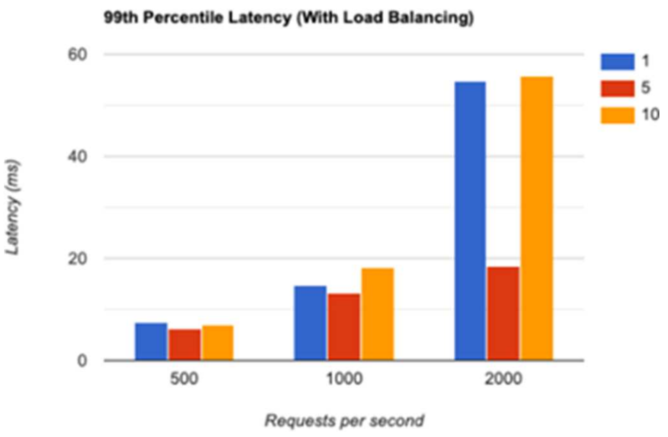

Fig. 8 Bar chart for maximum latency (in ms) for different number of HTTP requests per second made to different number of database connections with three instances service.

Table 4 and Fig. 8 show the result of 99th percentile latency for different requests made to different numbers of connections in a three-instance service. Five connections show the best-recorded performance for all 500, 1000, and 2000 rps. In 500 rps, five connections record the lowest latency with 6.227ms, followed by 7.111ms by ten connections, and lastly, a single connection with 7.496ms. For 1000 rps, five connections give 13.279ms latency, followed by 14.672ms for a single connection and 18.173ms for ten connections. Lastly, for 2000 rps, five connections only give 18.513ms compared to a single connection with 54.697ms and ten connections with 55.855ms.

Based on all results shown in this section, we can see that a low number of database connections would start to become a bottleneck when being hit with a larger load, especially with a single instance service. However, the performance improves when multiple instances are involved as load balancing the requests throughput helps to distribute the load instead of only a single instance to serve the requests. Having a larger amount of connections is not guaranteed to have a better performance in terms of latency. As we can see from the result in the experiment ran with multiple instances, the diminishing return effect for this could be caused by multiple factors such as the algorithm used to assign connection pool to request and how the performance from the database side when handling numerous concurrent connections.

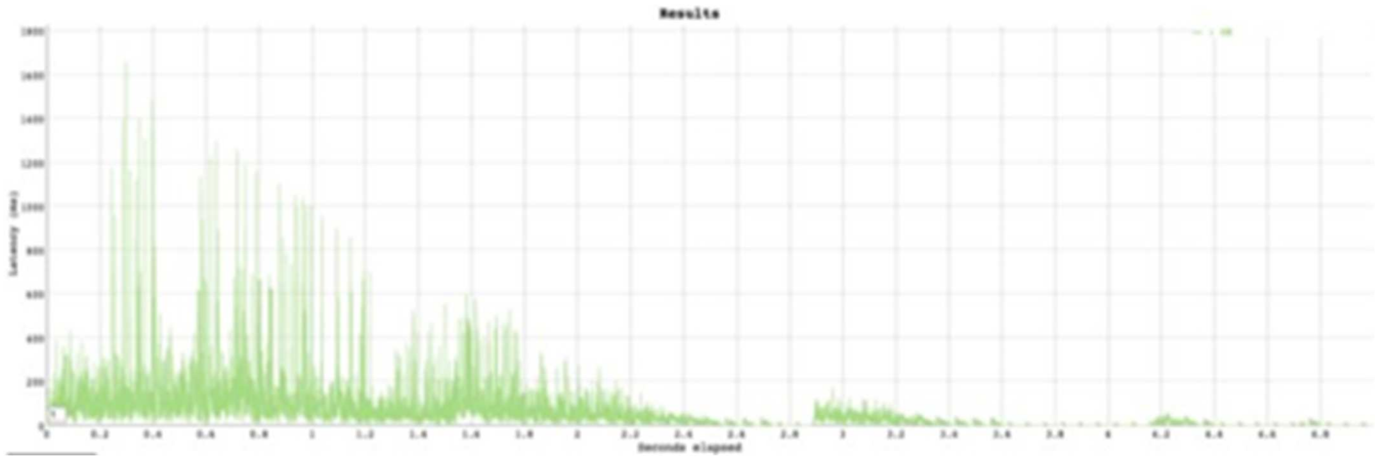

Fig. 9 Graph result for 2000 requests per second with five database connections in a single instance

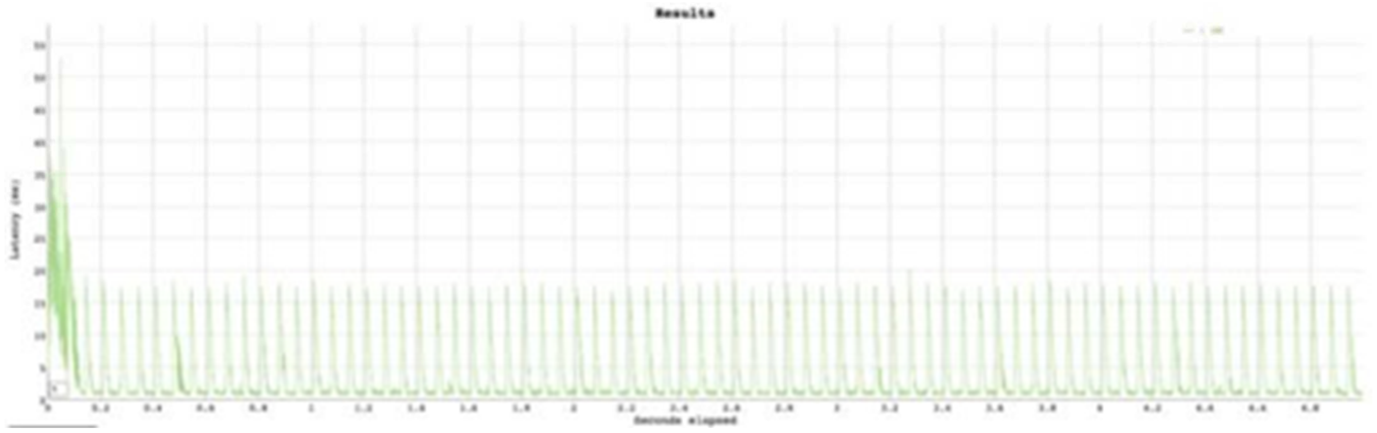

Fig. 10 Graph result for 2000 requests per second with five database connections with three instances

In Fig. 9 and Fig. 10, another noticeable difference we see between serving load with a single instance and multiple instances is we notice there is a constant spike for every few milliseconds recorded, which could be because of how the load balancer works when distributing the load between instances. However, even with the spike in latency, the overall result of distributing load with multiple services is far better than serving all the requests with only a single instance.

IV. Conclusion

We have presented the load testing done to our service to obtain a suitable number of database connections for our database connection pool (DCP). We tested for a single instance of our role/permission service as it is one of the most used services in our system, mostly due to authorization middleware checks for our users to access endpoints. From the result of our experiment and our proposed architecture for a production environment, we choose the five connections configuration as it gives the best performance for multiple instances service setup, as shown in Table 3 and Table 4 result.

Acknowledgment

The authors thank the Ministry of Higher Education, Malaysia, for financial support through the Long-term Research Grant Scheme (LRGS) 600-RMC/LRGS 5/3 (001/2020).

REFERENCES

[1] A. R. Sampaio *et al.*, "Supporting Microservice Evolution," presented at the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017.

[2] Y. Gan *et al.*, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," presented at the Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019.

[3] V. S. S. K. Peddoju, "Container-based Microservice Architecture for Cloud Applications," presented at the International Conference on Computing, Communication and Automation.

[4] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina, "Microservices: How To Make Your Application Scale," in *Perspectives of System Informatics*, (Lecture Notes in Computer Science, 2018, ch. Chapter 8, pp. 95-104.

[5] W. Hasselbring and G. Steinacker, "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce," presented at the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), 2017.

[6] I. A. S. D. P. E. Subyantoro, "Designing microservice architectures for scalability and reliability in ecommerce," *Journal of Physics: Conference Series,* 2020.

[7] M. Villamizar *et al.*, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conference (10CCC)*, 2015: IEEE, pp. 583-590.

[8] M. Viggiato, R. Terra, H. Rocha, M. T. Valente, and E. Figueiredo, "Microservices in practice: A survey study," *arXiv preprint arXiv:1808.04836,* 2018.

[9] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, 2018: IEEE, pp. 000149-000154.

[10] H. Dinh-Tuan, M. Mora-Martinez, F. Beierle, and S. R. Garzon, "Development frameworks for microservice-based applications: Evaluation and comparison," in *Proceedings of the 2020 European Symposium on Software Engineering*, 2020, pp. 12-20.

[11] C.-Y. Fan and S.-P. Ma, "Migrating monolithic mobile application to microservice architecture: An experiment report," in *2017 ieee international conference on ai & mobile services (aims)*, 2017: IEEE, pp. 109-112.

[12] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, 2019, pp. 25-32.

[13] D. Taibi, V. Lenarduzzi, C. Pahl, and A. Janes, "Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages," in *Proceedings of the XP2017 Scientific Workshops*, 2017, pp. 1-5.

[14] D. S. Linthicum, "Practical use of microservices in moving workloads to the cloud," *IEEE Cloud Computing,* vol. 3, no. 5, pp. 6-9, 2016.

[15] H. Zhao, S. Deng, Z. Liu, J. Yin, and S. Dustdar, "Distributed redundancy scheduling for microservice-based applications at the edge," *IEEE Transactions on Services Computing,* 2020.

[16] D. I. Savchenko, G. I. Radchenko, and O. Taipale, "Microservices validation: Mjolnirr platform case study," in *2015 38th International convention on information and communication technology, electronics and microelectronics (MIPRO)*, 2015: IEEE, pp. 235-240.

[17] C. Richardson, *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[18] T. F. Zhang, Y. J. Zhang, and J. Yao, "A Study of Database Connection Pool," *Applied Mechanics and Materials,* vol. 556-562, pp. 5267-5270, 2014, doi: 10.4028/www.scientific.net/AMM.556-562.5267.

[19] F. Liu, "A Method of Design and Optimization of Database Connection Pool," presented at the 2012 4th International Conference on Intelligent Human-Machine Systems and Cybernetics, 2012.

[20] R. Luo and X. Tang, "Design and Realization for JDBC-based Database Connection-pool," *Computer Engineering,* vol. 9, p. 036, 2004.

[21] X. Zhongke, "Database connection pool technology and its application," *Journal of Changsha University of science and Technology,* vol. 2, pp. 67-71, 2015.

[22] Q. Liang, Z. Shen, J. Luo, H. Fan, D. Ming, and J. Li, "Study of database connection pool in LBS platform," *Comput. Eng,* vol. 18, pp. 39-41, 2006.

[23] A. Edwards, "Let's Go Further," in *Let's Go Further*, vol. 1, 2021, ch. Configuring the Database Connection Pool, pp. 116-122.

[24] C. Hou, Z. Yang, and W. Liu, "Application and improvement of database connection pool based on J2EE architecture," *Computer Technology and Development,* vol. 16, no. 10, pp. 8-10, 2006.

[25] G.-l. Feng and L.-h. Yang, "A New Method in Improving Database Connection Pool Model," *World Academy of Science, Engineering and Technology,* vol. 29, pp. 246-249, 2007.

[26] A. Trzop. "Estimate database connections pool size for Rails application." https://docs.knapsackpro.com/2021/estimate-database-connections-pool-size-for-rails-application (accessed.

[27] F. Al-Hawari, A. Alufeishat, M. Alshawabkeh, H. Barham, and M. Habahbeh, "The software engineering of a three-tier web-based student information system (MyGJU)," *Computer Applications in Engineering Education,* vol. 25, no. 2, pp. 242-263, 2017, doi: 10.1002/cae.21794.

[28] B. H. Huang, T. J. Wang, Y. Ma, and F. Jiang, "Security Problem Modeling of Database Connection Pool," *Applied Mechanics and Materials,* vol. 543-547, pp. 3276-3279, 2014, doi: 10.4028/www.scientific.net/AMM.543-547.3276.

[29] B. H. Huang, Y. Ma, and F. Jiang, "Research on the Security Audit of Database Connection Pool," *Applied Mechanics and Materials,* vol. 543-547, pp. 3286-3289, 2014, doi: 10.4028/www.scientific.net/AMM.543-547.3286.

[30] N. Meghanathan, "Review of Access Control Models for Cloud Computing," presented at the Computer Science & Information Technology (CS & IT), 2013.