ПРОГРЕСИВНІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ,
ВИСОКОПРОДУКТИВНІ КОМП'ЮТЕРНІ СИСТЕМИ

UDC 004:415.2
DOI: 10.20535/SRIT.2308-8893.2026.1.02

OPTIMIZING MICROSERVICES DESIGN PATTERN: MAXIMIZING COMMUNICATION SPEED AND PROLONGING APPLICATION LONGEVITY

Y.E. KOVALOV, Y.V. BOYKO

Abstract. Microservice oriented application design obtained popularity in the past years. Most researchers investigated some aspects in microservice design for implementing application functionality. Little research considered the core functionality of microservices. This research investigates how to construct a microservice communication system by yourself in detail. The results should assist developers and architects to construct their own microservice applications, use less amount of frameworks and therefore prolong overall microservice system life cycle. The standard TCP/IP connection and embedded libraries were used to construct the communication system without using any additional frameworks. As a practical application of this methodology a microservice core system was implemented with a minimum number of microservices to perform performance testing. The measured application layer communication speed turned out to exceed the speed in real application because of database operation limitations. The implemented microservice core system is intended to be used in financial commercial applications as well as in further scientific investigations.

Keywords: microservice, application design, communication speed, domain-driven, monolith application, application life cycle, event, message bus, osi model, delivery guarantee, application layer.

INTRODUCTION

One of the modern programming design patterns is microservice architecture. It was derived from both service-oriented architecture (SOA) and event-driven architecture (EDA) [1]. Many people have heard of it. Some developers used it. The microservice architecture has something in common with challenges in any other application: how quickly it works, how easy to develop and support, how to integrate different parts together etc.

Although some researchers have paid attention to different practical aspects of microservice programming design [2–12], much less research has investigated the core functionality of microservice applications. In this research a practical methodology for creating microservice applications from scratch is represented. First of all, it is a way microservices interact with each other to work as a whole.

Past research mostly used standard communication frameworks and technologies. Although they are good for most practical tasks, the lack of developer control may reduce their longevity. In this research the original event delivering system was created and tested in detail. This approach gives developers full control

© *Y.E. Kovalov, Y.V. Boyko*, 2026

for communications as the most important part of the microservice system thus drastically increasing the life cycle of application.

The research questions we are supposed to answer in this study are as follows:

- What are the main benefits and drawbacks of using applications with microservice design?
- What are the main parts of microservice applications?
- How to take advantage of microservice design and evade drawbacks?
- How to construct a microservices communication system by yourself?

This paper has four parts. First it reviews the literature relevant to microservices design. Then the research methodology is presented and outlines the main parts of microservice application design. In the next step practical results of research are summarized and discussed. The paper concludes with a summary of results and further research.

LITERATURE REVIEW

There have been many architectural paradigms developed over time in computer systems. Some of these approaches, such as the widely known object-oriented programming, have significantly influenced computer languages. Another, such as structured programming, changed how developers write programs by prioritizing clear structure, improving readability, and making development and maintenance easier. Despite their differences, these patterns shared a common goal: to facilitate clearer program structures for human comprehension, thereby streamlining the software development and maintenance.

On the contrary, machine code is straightforward and rudimentary, consisting of a sequence of instructions that direct the processor on how to execute tasks. Unlike humans, machines don't rely on structured systems since they possess the capacity to retain every detail. The creation of computer languages, paradigms, and architectures was solely a human endeavor. It reflects human limitations, yet superior comprehension of grammar, particularly in the context of code syntax.

One of the most contemporary design patterns is the microservice architecture. Growing number of programmers and system architects interact with this technology. Developers engaging with microservices often question why this approach, what drawbacks exist in other technologies, and where to start.

Monolith applications

A so-called monolith application is a program although it consists of different modules, but they are working in a tight-coupled way. One module could not operate without another or at least there’s a central part of the application on which other parts are intertwined. Often it had a single start point and single database to hold information for the entire system. The human brain is an example of such a system in terms the brain has different intertwined tightly coupled parts.

No wonder that most legacy applications and computer systems were architected as monoliths, as this architecture was the simplest way to develop. The monolith system has many advantages that should be considered when creating other systems. First of all, it offers a high speed of interaction between its parts because most interlinks are performed using fast memory operations. It also has

no issues with transaction clarity, as it usually has a single database with built-in transaction processing.

A well-constructed monolith system can perform better than purely designed microservices architecture. So, before making a decision, let's take into account all the strengths and weaknesses of these architectures.

Domain-driven design

Domain-driven design (DDD) was first introduced by Eric Evans in his book “Domain-Driven Design: Tackling Complexity in the Heart of Software” [13]. Over time, this approach became a well-known architecture, with many authors providing their perspectives on this design [14].

The point is that large computer systems are usually developed by multiple teams of developers, not just one developer or a single team. This approach enables the quick development of complex applications because each team works simultaneously and independently.

DDD can be used in both monolithic and microservice applications [3]. It represents an idea on how to divide large systems into parts. Each team focuses on one part of the system independently. Later the teams stitch it together using one of the integration mechanisms.

In this work, some ideas and interpretation of this design were implemented on our way in making the microservice system.

Taking advantage of microservice architecture

A microservice-driven application was a further development of ideas on how to break down a complex system. Once again, it had something to do with human thinking. Our society is an example of such a system. Each individual is responsible for themselves and has their own skills and abilities, acting just like a single microservice in an application. Similarly, society can function as a whole because people communicate with each other and coordinate their efforts to achieve their goals.

One of the main goals of the microservice approach is to prolong the longevity of an application. Although monolith systems are easier to create, they are much harder to support. People tend to migrate to new technologies, computer languages, frameworks etc. Consequently, the number of developers willing to support legacy applications inevitably decreases over time.

As usual, renewing a monolith application means it must be rewritten from scratch. This interrupts the life cycle of old (legacy) applications, even though they may still satisfy all needs. Society wastes time and money by doing this. This issue occurs not only in software development but also throughout our daily lives.

Consider a simple object as a fridge. Some people throw it away just because it broke down or has an outdated design. But what if somebody makes a fridge like a Lego set allowing you to repair, upgrade, and change its parts and design as needed? I think this approach will inspire you to keep your fridge indefinitely. With this mindset, society could use our limited resources much more efficiently. Additionally, this approach is crucial for mitigating climate change.

Interest in the term “microservice” has grown since 2014 [15]. Many authors have made an effort to clarify what it is and how to work with it [2–6]. Table 1 illustrates how they addressed the challenges of microservice design.

Table 1. Microservice design approaches

	A Design Study of Microservice Architecture on White Label Travel Platform [3]	Application of Microservice Architecture in Commodity ERP Financial System [2]	Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture [4]	A Microservice-based Software Architecture for Improving the Availability of Dental Health Records [5]	Introducing Cloud-Assisted Micro-Service-Based Software Development Framework for Healthcare Systems [6]
Messages format					Soap (Simple Object Access Protocol) or JSON
How microservices interact with each other	Asynchronous Message bus	RestAPI inter service communication	RestApi, Kafka		RestApi, AMQP (Asynchronous Message Queuing Protocol)
How microservices interact with frontend	Synchronous Api gateway through RestAPI and Api gateway on some microservices. So some microservices have two-way communication	RestApi communication with Api gateway. Api gateway makes a connection through api with microservices	RestApi	RestApi	Service access tier through RestApi
How it was proposed to divide into microservices	Domain-driven design	Business logic			
Microservice's internal structure	Domain-driven design		Clean architecture		
Working with a database	Several microservices share the same database	One microservice – one database	One microservice – one database	One microservice – one database	One microservice – one database
Microservices connection information and discovery		Services registry configuration center			
What happens if microservice restarts					
What happens on microservice's lost communication		Special guard "Sentinel"			
How to find errors		Special guard "Sentinel"	logback encoder		
Interaction with other systems (including monolith systems)					
Authentication		Through separate microservice named Service Management Center (use spring Security + auth2 0)		Through separate microservice named Dental Service Provider	
Authorization		Several microservices- Authorized authentication service and User micro service		Through separate microservice named Dental Service Provider	
Complex (multiple microservices) transaction implementation			Saga, quota cache, eventual commit sync service		

But how to not find yourself lost in this variety of approaches, in other words how to catch a fish in a microservice's cocktail? This research will find answers to this question.

RESEARCH METHODOLOGY

When one microservice sends a message to others it may not know recipients. It just emits some mark of its own activity with useful information. So, it is more reasonable to name such an activity as an event.

The objective of this research is to create microservice application cores in a step-by-step manner. It includes communication, event format, events routing, database, event delivery, event processing and API gateway.

Communication

Communication is the imperative part of the microservice application since it relates to all microservices. Most applications under investigation used standard communication technology and frameworks (see Table1) such as RestApi for synchronous communication. For asynchronous communication top known methods were Kafka [16] and RabbitMQ [17].

As already mentioned above, monolith applications had an advantage of huge communication speed between the parts of the system. So, it is very important to provide a quick connection between microservices, otherwise it becomes a bottleneck of our application. Thus, it must be something lightweight and quick. RestApi may be a good choice, but it isn't lightweight and isn't quick enough as was concluded during interconnection speed investigation.

Of course, it is possible to use one of the open source message systems. But both methods that were mentioned above didn't guarantee that messages or events would not be duplicated. Additional efforts must be implemented to avoid duplication. Moreover, what if something goes wrong with them? Microservice systems may need some additional functions and don't need others. Maybe future developed microservice would require exact computer language and this open source system would not support it. Moreover, open source communication systems may suffer from a security vulnerability that would affect the whole system. And because it is open source all around would know about that. And finally, the lifecycle of microservice systems and open source systems can be different. You would have a problem if the open source system team stops supporting it.

If you want to do something good — do it yourself, especially when it is comparably easy to perform. It's reasonable to rely on technologies and methods that are supposed to survive longer than our microservice system's lifetime. Let's try figuring it out using the Open System Interconnection model (OSI) [18]. This model was developed in 1984 by the International Organization for Standardization (ISO). Let's scrutinize this model and try to make the best of it.

So, there are seven layers in this model (Fig. 1). The lowest level is Physical, the highest is Application. The lower layer the quicker communication but the harder to write a managed program.

The lower three layers are media layers. They deal with hardware and low-level software such as drivers, on-board programs and operating system's services. They are way too low for our purposes. But there was something important

we must keep in mind about physical connections – there were duplex connections and usually there were input and output low-level data caches. It means that a communication system must be constructed with embedded support of simultaneous input and output data flow to maximize performance.

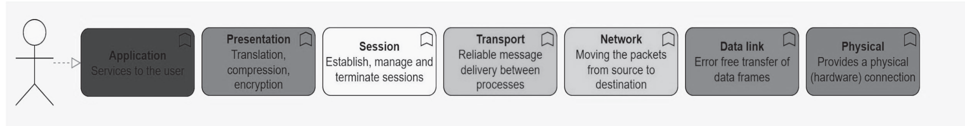

Fig. 1. OSI model

The upper four layers are host layers. And the lowest in this group (Transport) is good enough for our needs. But that's just a model. How about practical implementation?

TCP/IP model is a more practical implementation of communication suite compared to OSI.

It used four layers instead of seven layers in OSI model (Table 2) [19]. Now it is a communication standard supported almost by all devices, systems, operating systems, computer languages etc. It survived for a long time, and is supposed to survive even longer.

Table 2. OSI vs. TCP/IP model

<table>
<tr><th>OSI</th><th>TCP/IP</th></tr>
<tr><td>Application</td><td rowspan="3">Application</td></tr>
<tr><td>Presentation</td></tr>
<tr><td>Session</td></tr>
<tr><td>Transport</td><td>Transport</td></tr>
<tr><td>Network</td><td>Internet</td></tr>
<tr><td>Data link</td><td rowspan="2">Link</td></tr>
<tr><td>Physical</td></tr>
</table>

There were two main protocols on a Transport layer – transport control protocol (TCP) and user datagram protocol (UDP). The UDP protocol is stateless while TCP establishes and holds connections. UDP is quicker but the main disadvantage, meaning that it was less applicable for our purposes, was that it didn't track the sequence of data. The messages should be accepted by our receiver microservice exactly in succession they were transmitted. Thus, TCP protocol is best for our application core.

Event format

It is reasonable to use the message format that most of the computer languages and operating systems are using now and is expected to use in the future. One of the well-known formats that fit this demand is text format.

Well-known standard for encoding object information and data interchange in a text form is JavaScript Object Notation (JSON) [20]. It is commonly used in web application programs. Many computer languages have libraries to parse it.

It is not very difficult to write such a library on your own for legacy systems. It has basic types for Number, String, Boolean, Array, Object. From our point of view this format is good enough to be used as an event format, but you can choose any other text format you prefer.

Events routing

When events are emitting it is not the microservice's responsibility to know recipients. Otherwise, they would be tightly coupled, which is considered as a bad smell for microservices architecture [21]. So, the most reasonable approach is to implement subscriptions and event types (let's name it EventId). Microservice's subscriptions should somehow be remembered. Events of specified EventId must be sent to its recipients.

As it was concluded the microservice would be implemented for this functionality. Let's name it EventBroker. It works like a delivery system (Fig. 2).

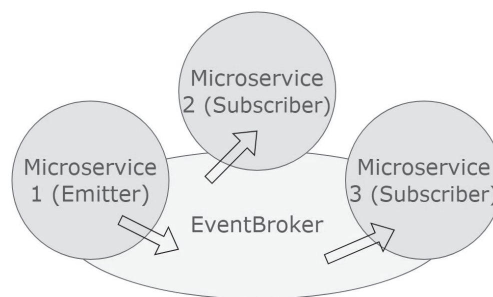

Fig. 2. Events routing

At the startup EventBroker subscribes itself to receive subscription events (startup record in a subscription table). Every microservice first emits a subscription event either on the startup or at any time. On receiving, EventBroker fixes this information in the database. From now on all events of that EventId will be forwarding to its subscribers. Fig. 2 portrayed stances in which microservice 1 is an event emitter and microservices 2 and 3 are subscribers for that EventId.

Sadly, EventBroker in this architecture is a single point of error, which is considered a bad smell in miscroservice's design. If EventBroker stops working the whole system will freeze. But let's keep in mind that there is already a single point of error in any microservice system – network. Everything goes wrong if the network stops working. And EventBroker is an extension of the network with responsibility to deliver events. If the network layout is super resilient and has several reserved lines it is possible to create several EvenBroker microservices for each line to avoid a single point of error. This would not change our methodology.

The EventBroker is a central coordinator of all events and simultaneously logs keeper. If something goes wrong in our microservice system – having logs is vital for debugging and moreover it is considered a bad smell if the microservice system doesn't have a central log keeper. Our goal is to create a microservice system, not just a bunch of microservices.

EventBroker establishes tcp/ip connection with every microservice we have. In order to maximize performance, it must actively use parallel execution tech-

nologies of your favorite computer language. Thus, there are two simultaneously running tasks for each microservice (input and output events processing).

Database

Every system needs data storage. The microservice system has a lot of microservices that have to be loose coupled. Shared databases in this approach are a way out of line. So, it is imperative for all microservices to have their own database.

It is up to developers to choose the database and framework to deal with. It can be Sql, NoSql, object-oriented and so on. Of course, there is no reason for one development team to use different database types in one project.

Event delivery guarantee – not less than once, not more than once

Communication is commonly used by many microservices, so it is reasonable to build a communication library for each computer language used. There are several reasons that may affect microservice system event delivery. First of all, it is a connection issue and microservice application problem.

Connection issues are caused by unstable network connection. If this happens one or several tcp/ip connections between EvenBroker and microservice may interrupt. The communication library must track this somehow and initialize connection renew. The simplest way to implement this functionality is periodic connection testing with small data amounts.

Microservice applications may be affected by following: be under the maintenance, in the restart process, hangs up etc. As a result, events suffer from not being delivered at all or being delivered several times. How to mitigate these artifact behaviors?

There is no problem at all sending events again and again until the subscriber accepts it. But to avoid duplication every event should be unique and easily identified. And keeping in mind that any microservice (including EventBroker) can be interrupted for reasons mentioned above, the only way to provide this is keeping the event state in our storage – database. Using database transaction technique, it is possible to keep atomicity of database changes even during restart.

So, every microservice must have a pool of outgoing and incoming events in the database with a unique id for every outgoing. When a microservice receives an event it first checks against the database if the event with that id already was registered. It registers events in the database if not. It replies to the sender with "Event with this id was received" either after registration or when it was already found in the database (Fig. 3).

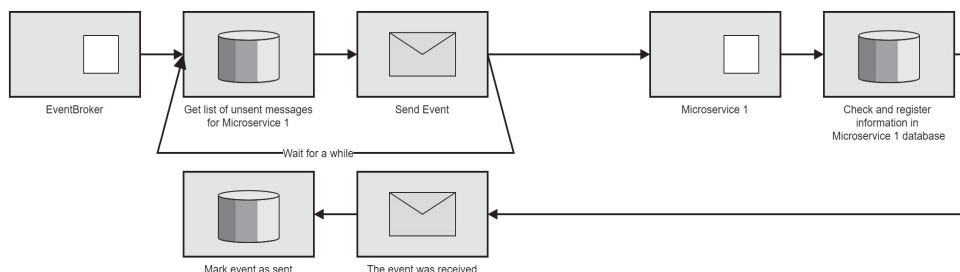

Fig. 3. Event delivery

Event processing

The event delivery approach proposed above is not only about the resilience of the microservice system but also gives the opportunity to perform event processing atomically.

Microservice is like a black box. It is not necessary for you to know how exactly it is working unless you send and receive events and understand its structure. But what if something happens in the middle between events being received and sent in other words when events are under processing?

In the proposed approach we can put the whole sequence into one transaction (Fig. 4).

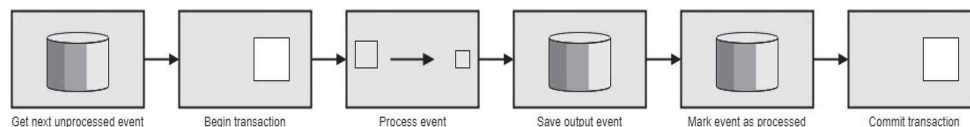

Fig. 4. Event Processing

Thus, either during processing the answer will be saved or the database state will be returned to the initial state just after the input event was received.

API gateway

As was mentioned earlier, RestApi today is the most widely used protocol for web applications. In our microservice architecture it can be used to connect other applications, frontend applications and so on. All we need is just another microservice (name it APIGateway), connected to EventBroker but with specific behavior.

APIGateway performs interlink between internal tcp/ip and Rest Api protocols adding caching capabilities. It is also a good place for user authentication since other parts of a system could be hidden from the outer world.

PRACTICAL RESULTS OF RESEARCH AND DISCUSSION

The microservice system was implemented based on the research methodology presented above. It was coded with c# for .net multiplatform program language. Entity framework, System.Net.Sockets library and MySql database were used as helper components.

In addition to EventBroker and APIGateway, a test microservice named "Ping" was implemented. With help of these microservices some vital microservice core parameters were measured.

Application layer speed

First of all, the application layer speed was measured in isolation without database operations. Using APIGateway we generated an event for Ping microservice. On event arrival Ping microservice generates 50000 messages of specified size addressed to EventBroker.

The software measured both times consumed to publish and receive messages. Two computers were used to measure performance with 1 Gigabit per

second network connection between them. First computer, with Ping microservice installed, is equipped with CPU "Intel Core I5-6400 2.7 Ghz" and 8 gigabytes of memory on Windows 10 system. Second, with ApiGateway, EventBroker microservices and MySql has CPU "Intel Core I5-6600K 3.5 Ghz" and 16 gigabytes of memory on Windows 10 system. One thread was used on both publish and receive sides.

It turned out the bottleneck in application layer performance was CPU on a publish computer with Ping microservice launched (one CPU kernel was fully engaged). Thus, consumed and published times were identical with few deviations, for that reason our graph shows only consumed time transformed to speed without consideration to publish time (Fig. 5).

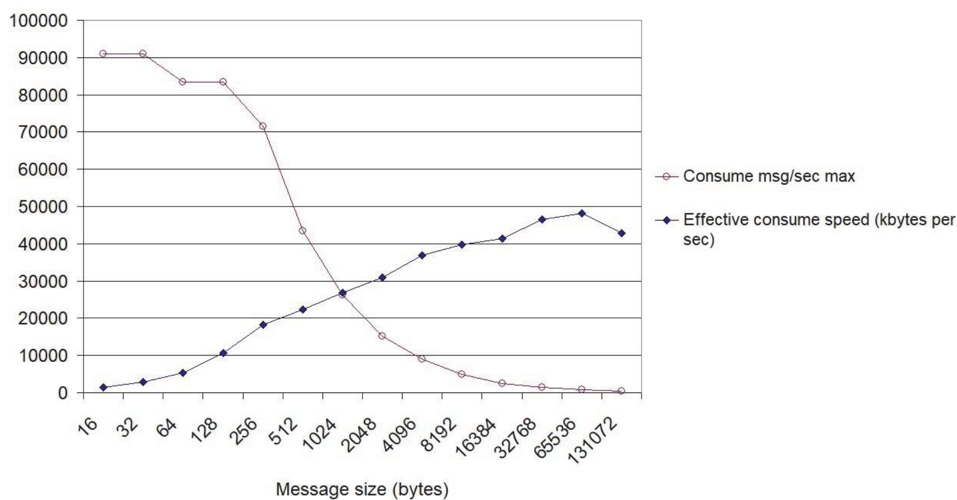

Fig. 5. Application layer speed

In addition, the effective consume speed shows that only a part of network connection bandwidth was engaged, as for 1 gigabit per second network theoretical bandwidth limit is about 122000 kilobytes per second (Fig. 5). But as already mentioned above this is because of a CPU bottleneck on a publishing computer.

For the most practical cases this speed is more than enough and exceeds performance of other universal message brokers already mentioned in paragraph "Communication".

Events processing speed

During the next phase let's add database operations to our experimental environment. The computer hardware configuration is the same as mentioned in a previous step. The results are on Fig. 6. Event size here is 398 bytes.

The events flood publisher here is microservice Ping. It generates 100 events and sends them to event receiver EventBroker microservice. How quickly EventBroker can save to the database was measured. EventBroker can engage multiple threads (c# Task library) to process events. With the number of tasks 4 the maximum saving rate was reached in this configuration. This happens when 100 % CPU usage was registered on EventBroker microservice (number of tasks 4 and 5).

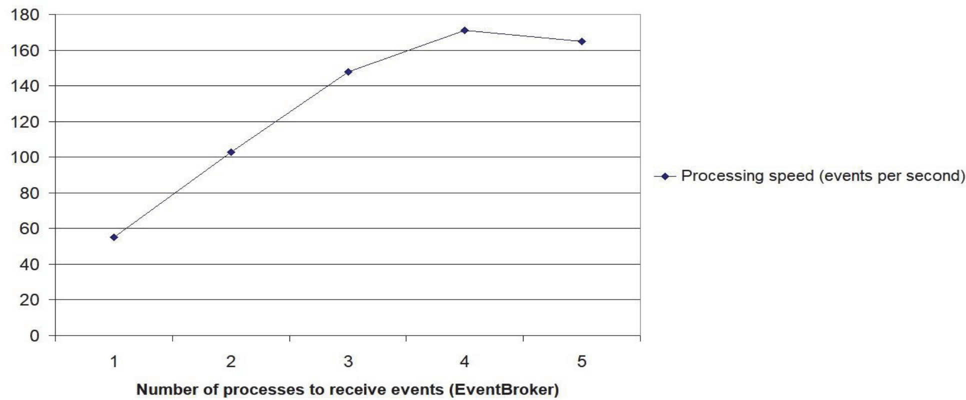

Fig. 6. Database was engaged in speed measurement

In the next phase the different event sizes were investigated with the same hardware configuration. Number of tasks engaged at EventBroker was fixed to 4 (Fig. 7).

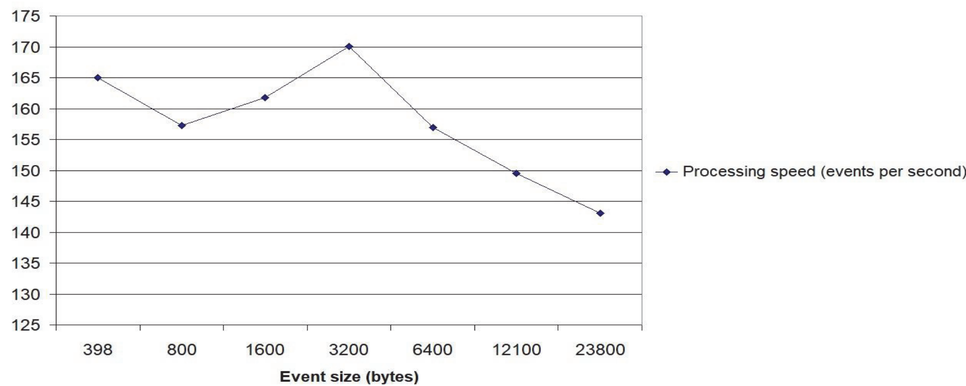

Fig. 7. Event size dependence in database engaged configuration

If we trace over the graph it is clear that processing speed is almost the same with few deviations until event size 6400 bytes is reached and then it slowly drops down. Thus, for this system architecture it is preferable to generate less amounts of bigger events to increase performance.

CONCLUSIONS AND FUTURE WORK

The microservice system with direct network connection application level was constructed. In this system special microservice EventBroker was implemented to be in charge of delivering events to other microservices. It also makes sure every microservice receives not less than one event, not more than one event. The event processing speed was measured in isolation without database operations and in real situations with database transactions.

The performance of application layer speed without database operations exceeds the universal message broker's speed even with one CPU engaged and reaches 90000 events per second. It is also possible to perform multithreading events processing for even better results.

The performance in real applications depends on how quickly the database operations are performed. For the experimental part a MySql database was involved. It shows a maximum productivity of about 170 events per second in the test environment. Multithreading environment boosts the performance. Thus, in a real environment with powerful servers the high system productivity is expected.

Self written microservices communication library gives developers control over application lifetime as the most important part of this design pattern.

The goal of the further research may include testing different database types in terms of productivity increasing. Also, the important parts of the microservice application should be investigated such as authentication, authorization.

REFERENCES

1. O.O. Petrenko, "A comparison of architecture types of services," *System Research and Information Technologies*, no. 4, pp. 48–62, 2015.
2. Wan Yan, Fu Shuai, "Application of Microservice Architecture in Commodity ERP Financial System," *International Journal of Computer Theory and Engineering*, vol. 14, no. 4, November 2022, pp. 168–173. doi: 10.7763/IJCTE.2022.V14.1324
3. I Gede Rahmat Wijaya, Ahmad Nurul Fajar, "A Design Study of Microservice Architecture on White Label Travel Platform," *Journal of System and Management Sciences*, vol.13, no. 4, pp. 249–264, 2023. doi: 10.33168/JSMS.2023.0415
4. Eman Daraghmi, Cheng-Pu Zhang, Shyan-Ming Yuan, "Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture," *Applied Sciences (Switzerland)*, vol. 12, issue 12, June-2 2022, Article number 6242, pp. 1–24. doi: 10.3390/app12126242
5. Juan Arcila-Diaz, Carlos Valdivia, "A Microservice-based Software Architecture for Improving the Availability of Dental Health Records," *International Journal of Computing*, vol. 21, issue 4, pp. 475–481, 2022. doi: 10.47839/ijc.21.4.2783
6. John Zaki, S.M. Riazul Islam, Norah Saleh Alghamdi, M. Abdullah-Al-Wadud, Kyung-Sup Kwak, "Introducing Cloud-Assisted Micro-Service-Based Software Development Framework for Healthcare Systems," *IEEE Access*, vol. 10, March 22, 2022, pp. 33332–33348. doi: 10.1109/ACCESS.2022.3161455
7. Zhongyi Lu, Declan T. Delaney, David Lillis, "A Survey on Microservices Trust Models for Open Systems," *IEEE Access*, vol. 11, March 23, 2023, pp. 28840–28855. doi: 10.1109/ACCESS.2023.3260147
8. Randa Ahmad Al-Wadi, Adi A. Maaita, "Authentication and Role-Based Microservice Architecture: A Generic Performance-Centric Design," *Journal of Advances in Information Technology*, vol. 14, no. 4, pp. 758–768, 2023. doi: 10.12720/jait.14.4.758-768
9. Ahmet Vedat Tokmak, Akhan Akbulut, Cagatay Catal, "Boosting the visibility of services in microservice architecture," *Cluster Computing*, vol. 27, pp. 3099–3111, September 18, 2023. doi: 10.1007/s10586-023-04132-5
10. Iury Araujo, Nuno Antunes, Marco Vieira, "Evaluation of Machine Learning for Intrusion Detection in Microservice Applications," *LADC '23: Proceedings of the 12th Latin-American Symposium on Dependable and Secure Computing*, pp. 126–135, October 17, 2023. doi: 10.1145/3615366.3615375
11. Wesley K.G. Assunção, Jacob Krüger, Sébastien Mosser, Sofiane Selaoui, "How do microservices evolve? An empirical analysis of changes in open-source microservice repositories," *The Journal of Systems & Software*, vol. 204, October 2023, 111788, pp. 1–14. doi: 10.1016/j.jss.2023.111788
12. Francisco Ponce, Jacopo Soldani, Hernán Astudillo, Antonio Brogi, "Smells and Refactorings for Microservices Security: A Multivocal Literature Review," *Journal*

of Systems and Software, vol. 192, October 2022, 111393, pp. 1–18. doi: 10.1016/j.jss.2022.111393

13. Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley, 2004, 501 p.
14. Vaughn Vernon, *Domain-Driven Design Distilled*. Boston: Addison-Wesley, 2016, 136 p.
15. Victor Velepucha, Pamela Flores, "A Survey on Microservices Architecture Principles, Patterns and Migration Challenges," *IEEE Access*, vol. 11, 15 August 2023, pp. 88339–88358. doi: 10.1109/ACCESS.2023.3305687
16. *Kafka documentation*. Available: https://kafka.apache.org/documentation/
17. *RabbitMQ documentation*. Available: https://www.rabbitmq.com/docs/documentation
18. *OSI Model*. Available: https://www.javatpoint.com/osi-model
19. *What is TCP/IP?* Available: https://www.techtarget.com/searchnetworking/definition/TCP-IP
20. *Introducing JSON*. Available: https://www.json.org/json-en.html
21. Tomas Cerny, Amr S. Abdelfattah, Abdullah Al Maruf, Andrea Janes, Davide Taibi, "Catalog and detection techniques of microservice anti-patterns and bad smells: A tertiary study," *The Journal of Systems & Software*, 206 (5):111829, December 2023, pp. 2–43. doi: 10.1016/j.jss.2023.111829

Received 17.09.2024

INFORMATION ON THE ARTICLE

Yurii E. Kovalov, ORCID: 0009-0002-1649-751X, Taras Shevchenko National University of Kyiv, Ukraine, e-mail: yuk123@meta.ua

Yuriy V. Boyko, ORCID: 0000-0003-1417-7424, Taras Shevchenko National University of Kyiv, Ukraine, e-mail: yuriyboyko@knu.ua

ОПТИМІЗАЦІЯ ШАБЛОНУ СТВОРЕННЯ МІКРОСЕРВІСІВ: МАКСИМІЗАЦІЯ ШВИДКОСТІ ЗВ'ЯЗКУ ТА ПОДОВЖЕННЯ ЧАСУ ЖИТТЯ СИСТЕМИ / Ю.Е. Ковальов, Ю.В. Бойко

Анотація. Останнім часом набуло популярності створення застосунків із використанням технології мікросервісів. Більшість дослідників аналізують можливості цієї технології для реалізації функціонування застосунку. Небагато досліджень присвячено функціям ядра функціонування мікросервісної системи. Мета дослідження — детальний розбір можливості самостійної побудови системи зв'язку мікросервісної системи. Результатами дослідження можуть скористатися розробники й архітектори програмного забезпечення для побудови своїх мікросервісних систем таким чином, щоб у них задіювалася менша кількість програмних шаблонів, таким чином збільшуючи життєвий цикл системи. Комунікаційну систему побудовано на базі стандартного TCP/IP з'єднання та вбудованих бібліотек для роботи із ним без використання додаткових програмних шаблонів. Як приклад практичного використання цієї методології розроблено ядро мікросервісної системи із мінімальною кількістю мікросервісів, необхідних для перевірки швидкості роботи. Як виявилося, виміряна швидкість зв'язку рівня застосунку перевищує швидкість у реальній ситуації через обмеження швидкості роботи із базою даних. Заплановано використати реалізоване ядро мікросервісної системи для розроблення комерційних фінансових застосунків та у ході проведення подальших досліджень.

Ключові слова: мікросервіс, архітектура програми, швидкість з'єднання, домен-орієнтований, монолітний застосунок, життєвий цикл програми, подія, шина повідомлень, модель osi, гарантування доставки, рівень застосунку.