Evaluating the Impact of Inter Process Communication in Microservice Architectures

Benyamin Shafabakhsh[a], Robert Lagerström[b] and Simon Hacks[b]

[a]School of Electrical Engineering and Computer Science,
KTH Royal Institute of Technology,
Stockholm, Sweden

[b]Division of Network and Systems Engineering,
KTH Royal Institute of Technology,
Stockholm, Sweden

Abstract

With the substantial growth of cloud computing over the past decade, microservice architectures have gained significant popularity and have become a prevalent choice for designing cloud-based applications. Microservices based applications are distributed and each service can run on a different machine. Due to its distributed nature, one of the key challenges when designing applications is the mechanism by which services communicate with each other. There are several approaches for implementing inter process communication (IPC) in microservices; each comes with different advantages and trade-offs. While theoretical and informal comparisons exist between them, this paper has taken an experimental approach to compare and contrast the popular forms of IPC communications. Several load test scenarios have been executed to obtain quantitative data related to performance efficiency, and availability of each method. The evaluation of the experiment indicates that, although there is no universal IPC solution that can be applied in all cases, the asynchronous pattern offers various advantages over its synchronous rival.

Keywords

Microservices, Inter Process Communication, IPC, Inter-Service Communication, Distributed Systems, gRPC, RabbitMQ

1. Introduction

Over the past few years, microservices have earned enormous attention and gained popularity from the industry. They helped large organisation such as Amazon and Netflix to serve millions of requests per minutes [1]. Microservice architecture is a style of developing software as a collection of independent services. Each service is running on its own process that is independent from other processes and can be deployed separately from other services [2]. Designing a software based on microservices involves answering questions and overcoming technical challenges that often do not exist in monolithic architecture, like inter process communication (IPC) [3], service discovery [4], decomposition strategy [5], or managing ACID transactions [6].

Despite the growth and importance of microservices in industry, there has not been sufficient research on microservices, partly due to lacking a benchmark system that reflects the characteristics of industrial microservice systems [7]. IPC is one of the important challenges of microservice architectures [8]. In monolithic based systems, components can call each other at the language-level while in microservices each component is running on its own process and possibly on a different machine than other services. The choice of IPC mechanism is an important architectural decision which can impact the software's non-functional requirements [8].

As of today, there are no concrete explanations or any standardized approach that can help to decide the right IPC method when designing microservice based applications. Due to this reason, there is an abundant confusion around the question of when to use which method and what are the trade-offs for choosing that method. Deciding between a synchronous and asynchronous approach is an important decision to take in regards to how services collaborate with each other [9].

There are two questions this paper is working towards answering:

1. From performance efficiency standpoint, what are the implications for utilizing available synchronous and asynchronous methods for implementing IPC in microservice architectures?
2. How does the IPC method choice impact availability of the system?

Woodstock'20: Symposium on the irreproducible science, June 01–05, 2020, Woodstock, NY
EMAIL: bensha@kth.se (B. Shafabakhsh); robertl@kth.se (R. Lagerström); shacks@kth.se (S. Hacks)
ORCID: 0000-0003-3089-3885 (R. Lagerström); 0000-0003-0478-9347 (S. Hacks)
© 2020 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-2767/07-QuASoQ-2020.pdf

The motive behind selecting the performance efficiency, and availability as the two criteria for this research is that the choice of IPC method directly impacts these two non-functional requirements in a microservices based system, while other non-functional requirements such as security [10] and maintainability [11] can span over few other areas and goes beyond IPC. Being able to measure these qualities in the system are critical in order to achieve an efficient management of any software system [12, 13]. Moreover, the chosen quality attributes are among the top priorities for most modern applications [14, 15].

In this work, we describe a systematic approach for selecting IPC method when designing microservices based software. The remainder of this paper focuses on state of the art in identifying different IPC models in section 2. Next, in section 3, we discuss the development of the prototypes built for the purpose of discovering the relationship between each IPC method and its impact on performance efficiency and availability. We then run a test against each prototype to investigate its outcome and discuss previous work conducted in this domain. Finally, we draw a conclusion in section 6.

2. State of the Art

When designing IPC mechanism, there are two type of interaction style to choose from: synchronous and asynchronous, which we will shortly introduce next.

2.1. Synchronous Communication

Synchronous communication is often regarded as request/response interaction style. One microservice makes a request to another service and waits for the services to process the result and send a response back. In this style, it is common that the requester blocks its operation while waiting for a response from the remote server. Representational state transfer (REST) application programming interfaces (API) [16] and gRPC[1] are the most common framework for implementing Synchronous form of communication in microservices [8].

- **REST API**: REST is an architectural style that is commonly used for designing APIs for modern web services [17]. In a system that uses REST API for its IPC communication, each service typically has its own web-server up and running on a specific port such as 8080 or 443, and each service exposes a set of endpoints to enable the interactions with other microservices and exchange of information between them. The server interacts directly with client through its interface also known as Web API.
- **gRPC**: gRPC is an open source high performance RPC framework designed an developed by Google. Remote procedure call (RPC) is a mechanism used in many distributed applications to facilitate inter process communication. RPC was first implemented by Birrell and Nelson [18] and it has been regarded as a protocol that enables a message exchange between two process with characteristics of low overhead, simplicity and transparency [19]. By default, when a client sends a request to a server it halt the process and waits for the results to be returned. RPC is therefore considered as synchronous form of communication [20]. Figure 1 presents the operational process between client and server in gRPC. In this model, the client implements the same method as its corresponding server through local objects also known as stubs.

2.2. Asynchronous Communication

The asynchronous form of communication can be implemented in microservices when services exchange messages with each other through a message broker. In this form of interaction, the message broker acts as an intermediary between services to coordinate the request and responses [8]. One of the fundamental differences in asynchronous communication as compared to the synchronous mode is that in asynchronous communication the client no longer makes a direct call to the server and expect an immediate answer. Instead, other services subscribe to the same broker to pick-up the available requests and process them further before placing them back to the message queue.

Figure 2 provides an example of the asynchronous pattern. In this sample, when a new order is created, the *customer service* publishes a request to the broker with some metadata such as *customer id*, *customer email address*, etc. Other services such as *loyalty*, *post*, and *email service* subscribe to that broker and take the request from there without having to communicate with Customer service directly.

[1] https://grpc.io

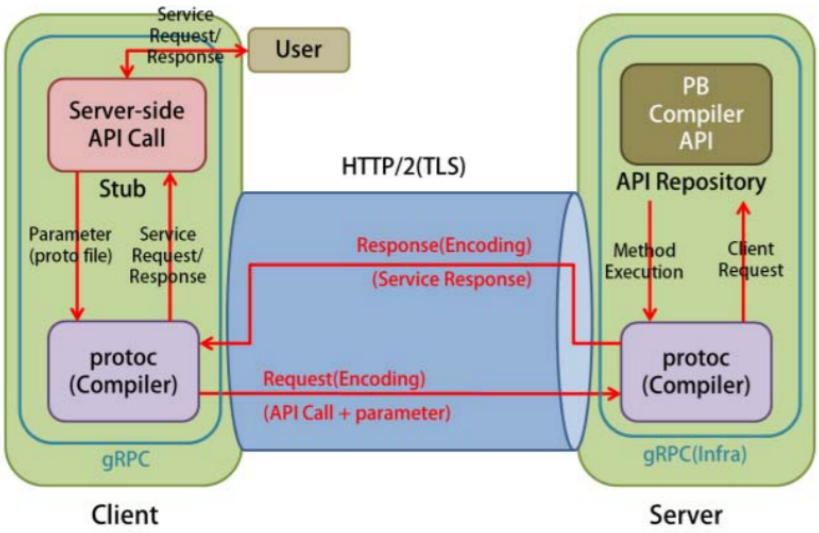

Figure 1: gRPC architecture [21].

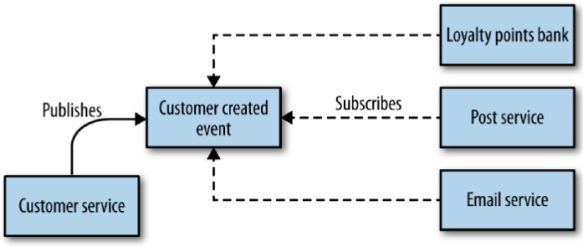

Figure 2: High-level architecture of Asynchronous pattern [9].

3. Implementation

To identify the quality attributes of each IPC method, we have designed and developed a set of microservices for an e-commerce scenario. In this scenario, the goal is to simulate fetching all the information required to display a product page of an e-commerce website. A client requests a product page to be displayed on his/her device and behind the scenes the following microservices work together to serve that request:

- **Product Information Service**: This microservice is responsible for fetching the primary metadata associated with the requested product. Information such as product name, price, description, color, and image are stored in this microservice database.
- **Product Review Service**: This microservice is responsible for fetching the customer reviews associated with the requested product from its database.
- **Product Recommendation Service**: This microservice is responsible for fetching the product recommendations based on the requested productId from its database.
- **Product Shipping Service**: This microservice is responsible for fetching available shipment options and the delivery estimates based on the given product from its database.
- **Customer Shopping Cart Service**: This microservices is responsible for fetching the existing items in the customer's shopping cart in order to display them to the customer.

All the microservices have been developed using NodeJS[2]. A non-relational database system, MongoDB[3], has been used as the database solution for all the microservices except for the service responsible for providing shipment information. Due to the nature of data required by shipping service, the shipping service uses MYSQL[4]. Docker[5] has been utilized to containerize all the microservices. In order to run the test system, the services have been deployed to Microsoft Azure Kubernetes Cluster Service[6]. Table 1 shows the

[2]https://nodejs.org/en/
[3]https://www.mongodb.com/
[4]https://www.mysql.com/
[5]https://www.docker.com/
[6]https://azure.microsoft.com/sv-se/services/kubernetes-service/

Table 1
Kubernetes Cluster specification of the test system.

Instance Type	Azure DS2-v2
vCPU	2
Memory	7 GiB
Storage	8 GiB, SSD, 6400 IOPS
Kubernetes Version	1.14.8
Node Count	3

hardware specification of the testing system used for this research.

In the synchronous mode both REST API, and gRPC have an identical architecture; in both methods, there is a direct communication between API Gateway[7] and each microservice. Each microservice acts as server, and the API Gateway acts as a client of those server. The key difference between REST API and gRPC is the underlying communication protocol as well as the format of the messages they exchange. gRPC has adopted protocol buffer[8] as its proprietary message format, while the REST API uses JSON [22] format to exchange data.

The asynchronous architecture uses RabbitMQ as message broker. In this pattern, the communication between API gateway and other services does not take place directly, rather it goes through a mediator also known as message queue. In both synchronous and asynchronous methods, the API Gateway is the entry point to the system, which receives a request with specific *product id* from client's device such mobile app or web browser over HTTPS protocol. The gateway then communicates back and forth with each microservice depending on the IPC method the system uses.

4. Results and Evaluation

4.1. Performance Efficiency

Three test cases have been designed and executed using Apache JMeter[9]. All test cases aim to measure the throughput of each IPC method. *Throughput* is an essential attribute for calculating performance efficiency. In all three test experiments, the test duration was 180 seconds, while the number of concurrent virtual users that continuously send requests to the system and wait for response has been varied. The motive behind having test duration as a constant variable and number of virtual users as the controlled variable is to understand how each IPC method reacts differently when the concurrent requests and traffic to the system increase or decrease.

Throughput is calculated by the total number of requests and responses the method managed to make within the specified duration of 180 seconds; the higher the number, the higher the throughput and the better it is.

The results are presented in figure 3. The data indicates that gRPC has outperformed REST API, and RabbitMQ in the first case with 50 users by being able to process 43 requests higher than REST API, and 147 requests more than RabbitMQ; this signifies that synchronous form of communication can offer higher throughput than the asynchronous method in the situation when the load to the system is relatively low. Meanwhile, the result of the first case also reveals that synchronous form of communication can process requests slightly faster than asynchronous form and, therefore, has lower latency when the number of concurrent threads[10] in the system is low.

The second case has double the number of virtual users as compared to the first one. Increasing the number of virtual users causes the number of concurrent threads in the system to grow and results in longer processing time. The same data imply that gRPC has the highest throughput by processing a higher number of requests compared to RabbitMQ and REST API; however, the gap between gRPC and RabbitMQ is now more narrowed than in the first case. In this test, gRPC managed to score the best average response time than REST API and RabbitMQ by 200 milliseconds. The processing time between REST API and RabbitMQ are equal to each other; however, RabbitMQ managed to process extra 25 requests than its synchronous rival.

The number of virtual users in the third case has increased four times as compared to the first case. The outcome of the third testing experiment implies considerable difference between synchronous versus asynchronous form of communication both in throughput and latency when the number of parallel requests increases. In this test, asynchronous form of communication using RabbitMQ has outperformed the other two methods by being able to process a total of 4480 requests within the given period while gRPC managed to process 132 requests lower than RabbitMQ, and REST API processed 146 less requests than its asynchronous rival. What makes the asynchronous pattern to operate better in the third test case is that, in asynchronous form the performance decline take place more gradually while in the synchronous

[7] https://microservices.io/patterns/apigateway.html
[8] https://developers.google.com/protocol-buffers
[9] https://jmeter.apache.org/

[10] Each virtual user occupies one thread in the system.

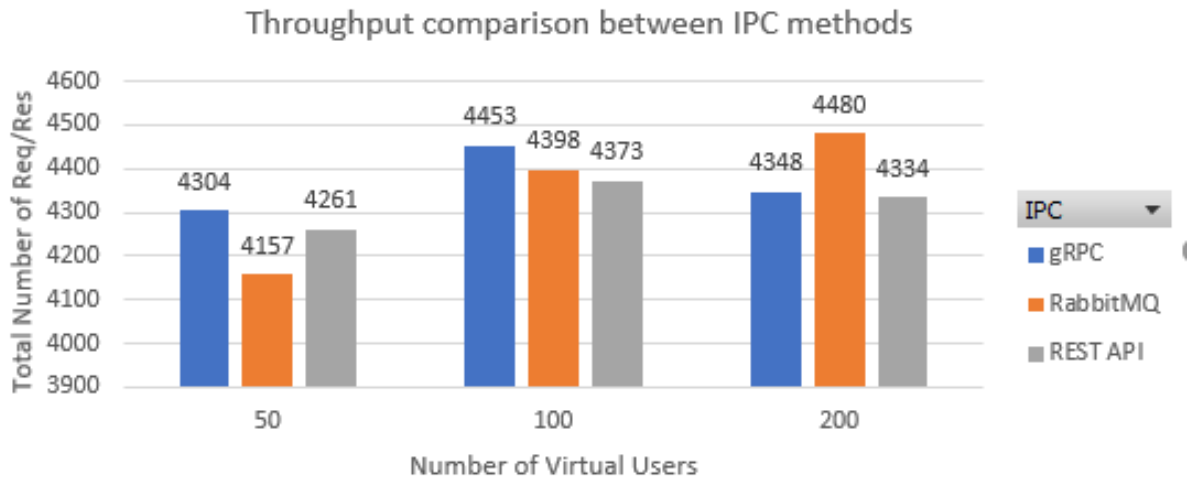

Figure 3: Throughput comparison.

pattern the performance begins to drop radically as soon as the load to the system intensifies.

4.2. Availability

There are variety of parameters that can affect availability of a system –even hardware components can play a role in determining the availability rate of a system. For this measurement, all the parameters outside IPC has been ignored. The availability of each IPC method has been calculated by using the following equation [23]:

$$Availability = \frac{MTTF}{MTTF + MTTR},$$

with MTTF standing for "Mean Time to Failure," and MTTR for "Mean Time to Recovery." MTTF represents the duration that the system is expected to last in operation before failure occurs. In contrast, MTTR represents the duration the system requires to return to operation after a failure has occurred. The higher the MTTR, the longer it takes for the system to recover from a failure, which consequently reduces the availability of the system.

Based on this formula, three other tests were executed using Apache Jmeter against all the three different IPC methods to discover which one offers higher availability. Unlike the previous test cases that had a fixed duration, these test cases had no specific duration. They ran as long as the services became unavailable due to the high number of requests coming to the system. Further, in this test, the average response time, and the number of requests/responses were not been tracked since they do not contribute to determining the availability of the method. The first case ran with 200 virtual users, the second with 300 virtual users, and third with 400 virtual users. Without having a high number of parallel users measuring availability becomes more challenging as the system remains operational for a significantly longer duration.

Figure 4 provides a summary of the conducted tests. During the first test, it took about seven minutes for the services to become unavailable using RabbitMQ, while gRPC went down after about five minutes, and the REST API took approximately four and a half minutes. These numbers were then dropped in each method in the subsequent tests as the number of parallel requests were doubled. After the services became unavailable, the Kubernetes cluster has been manually restarted. From that moment, both gRPC and REST API took about 20 seconds only to become available again, while RabbitMQ took ten extra seconds. The main reason behind RabbitMQ taking longer than synchronous form to return back to operation is the fact that it has an extra component known as a message broker that requires to be refreshed and establish a new connection with each service. From this experiment, it is possible to infer that an asynchronous approach offers higher availability than its synchronous opponents.

Consequently, if microservices use a synchronous based communication both client and server must be responsive at all time, otherwise the request will fail after a specific duration depending on the configura-

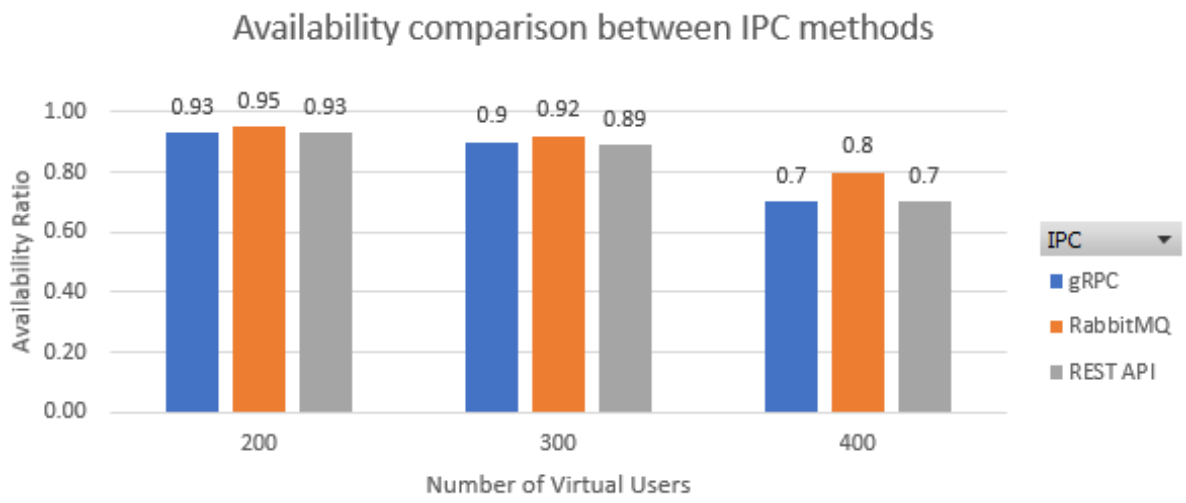

Figure 4: Availability comparison.

tion. In contrast, a temporary outage of the server in an asynchronous setting causes minimal to no impact to the consumer, since the consumer is loosely coupled with the server. The requests can stay in the message queue and be processed at the later timing when the server is back to operation. The asynchronous pattern offers capabilities that can help the system to improve its availability and resiliency from outage. It allows continuous operation even if there is a failure in one of the system's components without compromising the availability of the entire system.

4.3. Discussion and Threats to Validity

In addition to the two non-functional requirements that have been evaluated throughout this work, it is important to take into account the functional requirements for which microservices are being developed for. It is essential to distinguish whether the scenario requires an immediate response back from services or not. To elaborate further on this, in the proof of concept scenario that was built during this work, displaying a product page for an e-commerce was simulated. In this scenario, the client sends a request to load the product page and expects an immediate result back. The result of the request can either be the product page or an error that indicating the request was failed. The key point in this scenario is that the client expects an immediate result. In such scenarios the synchronous form of communication can be more suitable as these scenarios cannot take advantage of the features that an asynchronous form can offer.

Furthermore, our research incorporates some threats to validity. First, we performed our experiments just with single technologies as representatives for certain principles (synchronous vs. asynchronous). Therefore, our results can just indicate certain advantages of these principles. Second, we simulated no complete system but just a small part of a bigger system, e.g., there is no communication between the microservices during our requests. However, this ensures that we are not testing other effects, but only the interaction between the gateway and the microservices. Third, we were using technologies that are highly configurable, thus a completly different configuration could lead to other results. However, as we just changed configurations where necessary, we assume that others can reproduce our results, especially as they are in accordance with our theoretical expectations.

5. Related Work

Sufficient work has been done to benchmark the performance of microservices, and compare and contrast it with other architectures such as service oriented architecture (SOA) [24], or with the monolithic architecture [25, 26, 27, 28, 29].

Ueda et al. [30] conducted research at IBM that aimed to design an infrastructure that is optimized for running microservice architectures. The team built two versions of the sample application. One based on monolithic and the other based on microservices. The team discovered a significant performance overhead and higher hardware resource consumption in the mi-

croservices version of the application as compared to monolithic one. The paper has marked poor design of process communication in microservice architectures as one of the significant performance degradations, and, therefore, unleashed the potential for further research and improvements in this topic. The paper has also pointed out that network virtualization techniques, which are often used in a microservice architectures, is another non-negligible reason behind the performance gap of monolithic versus microservice architecture. The paper, however, has not prescribed any specific solution or suggestion as to how to overcome these challenges but rather pointed out the potential future work for it.

Fernandes et al. [31] compared REST API performance versus advanced message queuing protocol (AMQP) [32], which is one of the protocols used in message-based communication that falls under asynchronous category. The study has been done by measuring the averaged exchanged messages for a period of time using the REST API and AMQP. The authors performed the experiments by setting up two independent software instances that constantly received messages for a 30 minutes period with an average 226 request per second. Each instance processed the received input message and stored them into a persistent database. After executing the experiments, the authors concluded that for scenarios where there is a need to receive and process-intensive amount of data, AMQP performs far better than REST API as it has a better mechanism for data loss prevention, better message organization, and utilize lower hardware resources.

In contrast to Fernandes et al., check we in our work the behavior of the systems with different loads. We recognize that synchronous approaches perform good with low loads while asynchronous approaches scale better at higher loads.

Meanwhile, Dragoni et al. [28] have conducted a migration for a real-world mission-critical case study in the banking industry by transforming a monolithic software into a microservice architecture. They observed how availability and reliability of the system changed as a result of the new architecture. The solution consists of decomposing several large components to which some of them requires to communicate with third-party services. The services in the new architecture use message-based asynchronous communication as its IPC model to exchange data with each other. The authors believe that aiming to have a simple and decouple integration between services and following principle to handler failure will eventually lead to higher reliability in microservice architecture.

Further, the authors argue that microservice architectures lead to a higher availability as the new system is broken down into several components and decoupled from each other, which makes it possible to load-balance individual services as needed. This was particularly not possible in the legacy monolithic based system. At the same time, the new architecture offers higher reliability and can better cope with failures. This is due to the fact that in the new system the communication relies on a message-broker that can be configured to ensure all messages get delivered eventually.

6. Conclusion

When developing a microservices based system, the choice of IPC method is an important decision to make. In this paper, we compared synchronous and asynchronous IPC methods with regards to performance efficiency and availability. The outcome of our evaluation indicates that on average asynchronous approach provides better performance efficiency and higher availability. We also discussed a scenario where synchronous methods are more suitable to be utilized. Therefore, both synchronous and asynchronous type of communication has to be adopted according to the functional and non-functional requirements of the specific components.

References

[1] J. Thönes, Microservices, IEEE software 32 (2015) 116–116.

[2] D. Namiot, M. Sneps-Sneppe, On micro-services architecture, International Journal of Open Information Technologies 2 (2014) 24–27.

[3] L. L. Peterson, N. C. Buchholz, R. D. Schlichting, Preserving and using context information in interprocess communication, ACM Trans. Comput. Syst. 7 (1989) 217–246. doi:10.1145/65000.65001.

[4] S. Haselböck, R. Weinreich, G. Buchgeher, Decision guidance models for microservices: service discovery and fault tolerance, in: Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems, 2017, pp. 1–10.

[5] J. Fritzsch, J. Bogner, A. Zimmermann, S. Wagner, From monolith to microservices: A classification of refactoring approaches, in: J.-M. Bruel, M. Mazzara, B. Meyer (Eds.), Software Engineering Aspects of Continuous Development

and New Paradigms of Software Production and Deployment, Springer International Publishing, Cham, 2019, pp. 128–141.

[6] C. K. Rudrabhatla, Comparison of event choreography and orchestration techniques in microservice architecture, Int J Adv Comput Sci Appl 9 (2018) 18–22.

[7] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, W. Zhao, Poster: Benchmarking microservice systems for software engineering research, in: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), IEEE, 2018, pp. 323–324.

[8] C. Richardson, Microservices patterns: with examples in Java, Manning Publications, 2019.

[9] S. Newman, Building microservices : designing fine-grained systems, first edition.. ed., 2015.

[10] P. Johnson, D. Gorton, R. Lagerström, M. Ekstedt, Time between vulnerability disclosures: A measure of software product vulnerability, Computers & Security 62 (2016) 278–295.

[11] R. Lagerström, P. Johnson, M. Ekstedt, Architecture analysis of enterprise systems modifiability: a metamodel for software change cost estimation, Software quality journal 18 (2010) 437–468.

[12] P. Närman, P. Johnson, R. Lagerström, U. Franke, M. Ekstedt, Data collection prioritization for system quality analysis, Electronic Notes in Theoretical Computer Science 233 (2009) 29–42.

[13] M. Ekstedt, U. Franke, P. Johnson, R. Lagerström, T. Sommestad, J. Ullberg, M. Buschle, A tool for enterprise architecture analysis of maintainability, in: 2009 13th European Conference on Software Maintenance and Reengineering, IEEE, 2009, pp. 327–328.

[14] U. Franke, M. Ekstedt, R. Lagerström, J. Saat, R. Winter, Trends in enterprise architecture practice–a survey, in: International Workshop on Trends in Enterprise Architecture Research, Springer, 2010, pp. 16–29.

[15] P. Johnson, R. Lagerström, P. Närman, M. Simonsson, Extended influence diagrams for system quality analysis, Journal of Software 2 (2007) 30–42.

[16] R. T. Fielding, R. N. Taylor, Architectural styles and the design of network-based software architectures, volume 7, University of California, Irvine Irvine, 2000.

[17] M. Masse, REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces, " O'Reilly Media, Inc.", 2011.

[18] A. D. Birrell, B. J. Nelson, Implementing remote procedure calls, ACM Transactions on Computer Systems (TOCS) 2 (1984) 39–59.

[19] J.-K. Lee, A group management system analysis of grpc protocol for distributed network management systems, in: SMC'98 Conference Proceedings. 1998 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No. 98CH36218), volume 3, IEEE, 1998, pp. 2507–2512.

[20] R. A. Olsson, A. W. Keen, Remote Procedure Call, Springer US, Boston, MA, 2004, pp. 91–105. doi:10.1007/1-4020-8086-7_8.

[21] S. G. Du, J. W. Lee, K. Kim, Proposal of grpc as a new northbound api for application layer communication efficiency in sdn, in: Proceedings of the 12th International Conference on Ubiquitous Information Management and Communication, 2018, pp. 1–6.

[22] C. Severance, Discovering javascript object notation, Computer 45 (2012) 6–8.

[23] P. Johnson, R. Lagerström, M. Ekstedt, M. Österlind, It management with enterprise architecture, KTH, Stockholm (2014).

[24] T. Erl, Service-oriented architecture: concepts, technology, and design, Pearson Education India, 1900.

[25] T. Cerny, M. J. Donahoo, J. Pechanec, Disambiguation and comparison of soa, microservices and self-contained systems, in: Proceedings of the International Conference on Research in Adaptive and Convergent Systems, RACS '17, Association for Computing Machinery, New York, NY, USA, 2017, p. 228–235. doi:10.1145/3129676.3129682.

[26] D. Taibi, V. Lenarduzzi, C. Pahl, Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation, IEEE Cloud Computing 4 (2017) 22–32.

[27] R. Chen, S. Li, Z. Li, From monolith to microservices: A dataflow-driven approach, in: 2017 24th Asia-Pacific Software Engineering Conference (APSEC), 2017, pp. 466–475.

[28] N. Dragoni, S. Dustdar, S. T. Larsen, M. Mazzara, Microservices: Migration of a mission critical system, arXiv preprint arXiv:1704.04173 (2017).

[29] Z. Kozhirbayev, R. O. Sinnott, A performance comparison of container-based technologies for the cloud, Future Generation Computer Systems 68 (2017) 175 – 182. doi:10.1016/j.future.2016.08.025.

[30] T. Ueda, T. Nakaike, M. Ohara, Workload characterization for microservices, in: 2016 IEEE international symposium on workload characterization (IISWC), IEEE, 2016, pp. 1–10.

[31] J. L. Fernandes, I. C. Lopes, J. J. Rodrigues, S. Ullah, Performance evaluation of restful web services and amqp protocol, in: 2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN), IEEE, 2013, pp. 810–815.

[32] S. Vinoski, Advanced message queuing protocol, IEEE Internet Computing 10 (2006) 87–89.