Optimized Strategy for Inter-Service Communication in Microservices

Sidath Weerasinghe, Indika Perera
Department of Computer Science & Engineering, University of Moratuwa, Sri Lanka

***Abstract*—In the last decade, many enterprises have moved their software deployments to the cloud. As a result of this transmission, the cloud providers stepped ahead and introduced various new technologies for their offerings. People cannot gain the expected advantages from cloud-based solutions merely by transferring monolithic architecture-based software to the cloud since the cloud is natively designed for lightweight artifacts. Nowadays, the end user requirements rapidly change. Hence, the software should accommodate those accordingly. On the contrary, with Monolithic architecture, meeting that requirement change based on extensibility, scalability, and modern software quality attributes is quite challenging. The software industry introduced microservice architecture to overcome such challenges. Therefore, most backend systems are designed using this architectural pattern. Microservices are designed as small services, and those services are deployed in the distributed environment. The main drawback of this architecture is introducing additional latency when communicating with the inter-services in the distributed environment. In this research, we have developed a solution to reduce the interservice communication latency and enhance the overall application performance in terms of throughput and response time. The developed solution uses an asynchronous communication pattern using the Redis Stream data structure to enable pub-sub communication between the services. This solution proved that the most straightforward implementation could enhance the overall application performance.**

***Keywords*—Microservices; software architecture; inter-service communication; performance; streams**

I. Introduction

Recently, cloud computing has become considerably popular in the software industry, ultimately making businesses consider migrating their workloads to the cloud environment from their on-premise servers, as managing on-premise server farms is more costly and requires extra effort and maintenance. Based on the particular requirement, the consumers can choose the cloud services such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), software as a Service (SaaS), and Function as a service (FaaS). The main advantages of using the cloud are that consumers can gain whatever service they require according to their budget. The cloud provider fully manages the environment, and consumers no longer have the hassle of worrying about maintenance. Higher availability and easy vertical and horizontal scalability are some of the advantages of using cloud resources.

In the early days, requirements were very bounded, less volatile, and limited. Therefore, maintaining monolithic architecture software was easy. However, in modern society, user requirements are complex and subject to constant change, making it cumbersome to make adequate changes to monolithic systems. In the monolithic architecture, all the data access layers, data store layer, logic layer, and user interface layer are tightly coupled into one single package. As a result, changing the code, adapting to the new technology, and testing the product have become problematic. Therefore, people invented Service Oriented Architecture (SOA) to design loosely coupled services. In modern SOA implementations, all the services are orchestrated by the Enterprise Service Bus (ESB). The main disadvantage of this architecture is that all the service calls are routed through this ESB, which is a single point of failure. The performance also tends to get impacted because of that [1].

Cloud services are designed for the light weighted small-scale artifacts like microservices. Hence, people cannot get the complete advantage of migrating their monolithic system into the cloud. Due to this cause, people are trying to reengineer their existing products into microservices architecture by separating the services and making them individual microservices [2]. There are many strategies to decompose the monolithic service into services, such as decomposing by the domains and subdomains, decomposing by the business capabilities, decomposing by the system responsibilities, and decomposing by the resources [3]. After converting monolithic applications into microservices architecture, people can gain many advantages such as improving code maintainability, adapting to new technologies, efficiently scaling only the required service, improving resilience, gaining more business agility, being easy to understand, etc. [4].

Some systems still have not moved their software to the microservice architecture because of certain performance issues related to response time and the application's throughput. Microservice design is an independent service; such services need to communicate with each other to provide the user requirements. Those services deployed in the distributed environment and for the communication services must send and receive the data packets through the network, adding extra latency compared to the monolithic architecture software. Synchronous and Asynchronous communication styles are the two communication styles that are used for microservices interservice communication [5]. Synchronous type still mainly uses the request/response-based behavior, and the request waits until the response reaches. Most people use HTTP or gRPC communication protocols due to inter-service communication in the microservices. Asynchronous communication styles use message brokers to exchange messages to the relevant microservices. They are mainly using the Pub/Sub mechanism,

which means that requests are not waiting for a response, and there are blocking threads associated with the communication. Researchers invert the broker less asynchronous methods, but there is no guarantee of the exact message delivery [6]. Most programming languages support microservice development, and they also develop the framework in conformance to that. Java language based Spring boot [7] and VertX [8], Node.Js programming language based Molecular [9], and Golang programming language based GoMicro are fine examples of such instances [10].

This research focuses on the main performance issue of the microservice architecture, which is caused by the inter-service communication in the microservice architecture. As a result of the research, the solution was proposed to reduce the communication latency when communicating on microservice. The researcher has brought REST-based behavior to the top of the Redis Streams data structure for distributed communications. The network layer used the TCP-based socket connection and the serialized data packets to reduce the network latency while transferring the data. The rest of the paper discusses the implementation and the evaluation of this research outcome.

II. Literature Review

A. Microservices

Previous research publications showed that microservices research contributions started around the 2000 decade. After 15 years, Microservice research is drastically getting published in various academic journals and conferences [11]. Before the emergence of Microservice architecture, most engineers used Service Oriented Architecture (SOA) to build enterprise software. But, researchers have proven that they are faced with capacity issues and scaling issues with the SOA applications [12]. AI-Debagy and Martinek conducted a comparative review regarding the monolithic and microservice architecture. The experimental results of that research showed that monolithic applications perform 6% more on the throughput when compared to microservice-based applications [13]. Nevertheless, researchers have failed to elaborate on the underlying reason for the performance issue. The National Polytechnic School researched the challenges and problems faced during the system migration from monolithic to microservices architecture [14]. Finding suitable tools for migration, reorganizing the engineering team to work with the microservices, identifying the correct microservice design, guaranteeing consistency, and learning about the new framework are the challenges/problem they have highlighted in their research article. Lithuanian researchers reviewed the monolithic to microservice architecture, microservices methods and techniques [4]. One of the methods is to identify all subsystems associated with the monolithic architecture and create a dependency graph. Then architects can determine the services that need to be created as a microservice from the monolithic system. Another method is to identify the business logic from the dataflow diagram and decide what microservices can be created based on the independent business logic. As a best practice of the migration process, it is better to identify the minor steps and execute them one by one. With that, they have a guarantee on the path of restoration. The software engineering department of Tashkent University published the mechanism to decompose the monolithic architecture system to microservice-based architecture with less development effort [15]. The process started with analyzing the monolithic system, then extracting micro functions, refactoring the service catalog, and finally, orchestrating the services. However, the researchers have not shown the proposed mechanism's real-world application. Florian Auer and team conducted the assignment to find out the facts that companies consider when migrating their system to microservice architecture [16]. Scalability, maintainability, complexity, reusability, modularity, deploy-ability, reliability and testability quality attributes of the Microservice are considered in their study. They have also figured out that most of the companies do not measure the process, product and the quality attribute in depth before the migration. Only after the migration that the companies realize the implications it has. This is caused because there is no standard framework and tech stack that engineer can use when developing the microservices.

B. Inter-Service Communication

Presently, a lot of programming languages and microservice frameworks have emerged to develop microservices. When developing the Microservice, engineers use a tech stack solely based on their area of expertise. In most scenarios, they do not consider the application’s nature. There are ample ways to perform service-to-service communication, which is the most crucial part of the microservice architecture. But, there is no clear-cut approach to identifying the most suitable and efficient method. Christy Pachikkal researched the microservices communication styles as synchronous communication and asynchronous messaging [5]. According to that research, developers need to intensely go through the system's functional / non-functional requirements and choose the correct communication style. Most developers use the REST protocol because of the ubiquity of the protocol, which makes them architecturally understand how this protocol works [17]. REST protocol mainly uses JSON format. But in certain instances, it uses the XML-based format for message passing. Those message formats take massive amounts of time to message parsing because of the weight of the message. As a solution, REST is supported for the binary JSON (BSON), which also has overhead with the field names within the data structure [18]. Google invented the Remote Procedure Call(RPC) framework-based protocol to get more performance than the REST over HTTP [19]. Abram Perdanaputra conducted research related to the microservice, which is deployed on the Kubernetes environment. In addition to that, all the communication is done by the gRPC protocol. They have decoded the request and the responses for transparent tracing, but that can be achieved in the passive mode. HTTP/2 was introduced in 2015. It is considered a binary protocol, which gives more efficient bandwidth usage and header compression [20]. Researchers have enabled the multiplexing for HTTP/2 protocol so that the clients can send multiple requests via the same TCP connection before the response is received by the client. This implies that if people can use the same TCP connection to send and receive messages, then they can reduce the latency in message passing. Google has invented a new protocol named Quick UDP Internet Connection (QUIC), which uses the User Datagram Protocol

(UDP) instead of the TCP connections, and behaves as a transport protocol for HTTP/2 [21]. Norwegian University researchers argue in other works that QUIC protocol performance degrades when the messages payload size gets larger than the HTTP/1.1 [22]. Gaetano Carlucci et al. conducted research on the QUIC and showed how QUIC and TCP protocols behave when the network is congested. Their experimental results have proven that TCP is able to provide better response time when compared to the QUIC protocol when a network packet loses.

A group of researchers in Indonesia has implemented asynchronous communication for the microservice architecture with the help of RabbitMQ message broker [23]. Seven Microservices were developed and deployed in an environment that could easily scale down. Communication between the services is done in an asynchronous event-driven manner which led to speed up the application because there was no waiting as request/response architecture. They proposed durable topics which can send the events to the subscriber, i.e., microservices when available. With this concept, they guarantee the message delivery to the client. Sanjana et al. researched the highly resilient inter-process communication service for the microservice architecture [24]. In their implementation, they have used the Kafka message broker and have enabled the pub/sub messaging style to do the inter-service communication. Researchers have used the Camel routes, which are capable of message transformations and validation when doing message routing. With that function, they have proven that inter-service communication can be done without changing the existing architecture of the microservice implementations. Therefore, they argue that provided solution is highly resilient and lightweight for inter-service communication. This non-functional requirement is brought up because they have implemented the solution over the existing framework.

III. Methodology

This research focuses on implementing a solution for the inter-service communication method to improve the overall performance in a microservice architecture. When implementing the solution, the researcher has considered asynchronous communication as a communication style according to the facts found in the literature review part. The proposed system uses the Redis with Streams data structure, combined with Pub/Sub communication pattern for message passing in the underlying implementation.

A. Redis

Redis is an open-source solution that people can use as an in-memory data structure store. The advantage of the Redis is that it gives high read/write speed with high concurrency [25]. Redis is, by default, a support for easy scaling with the cluster concept. It also brings high availability and fault tolerance with the concept of virtual hash slots. Redis uses its own communication protocol to engage with clients as RESP (Redis Serialization Protocol), and it is also a binary safe protocol [26]. Since it is a Serialization communication protocol with binary safety, transporting the payloads will take less bandwidth. All the clients are connected to the Redis using the TCP connection. In this proposed method, the stream-oriented connection, which is similar to Unix sockets is used.

B. Redis Streams

Redis streams are introduced from Redis 5.0, which can publish the message to the stream, and consumers who subscribe to that stream can receive them. Redis streams differ from Kafka because the stream is an append-only data structure that helps with real-time messaging. The main advantage over the pub/sub model is that Redis streams persist the messages. Hence, it can guarantee the exact message delivery. Message reliability is the most important part of microservice inter-service communication, and can be achieved from this model.

C. Component Architecture

Fig. 1 depicts the system architecture of the proposed solution. Microservice A, B, and C are independent microservices that are deployed in the distributed environment. In order to produce the functional requirements in the software, each microservice needs to communicate with one another. In the proposed solution, communication is enabled through the Redis Streams as Pub/Sub communication style. Every microservices creates a Unix-based socket connection from the microservice to the Redis server via TCP 6379 port. All the messages are passed through that TCP connection over the RESP protocol. Every time the microservice does not create and close the connection, when the microservice starts, the TCP connection creates and will live until it shuts down. Thus, network creation and closing time can be reduced with this approach. Using this mechanism developed, the Java-based library enables efficient communication between the microservices and brings all attributes of the HTTP protocol to the developed library. Programmers can use this without changing the existing architecture of their system.

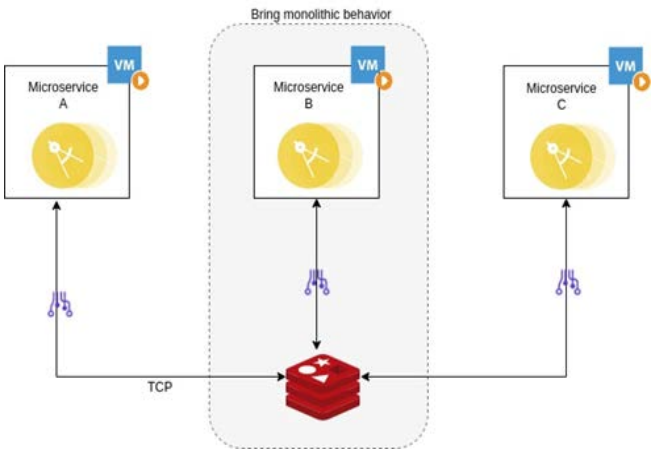

Fig. 1. High-level system architecture

IV. Implementation

This section briefly describes the implementation of the proposed solution. Java programming language and Spring boot microservice framework have been used to implement the proposed solution according to the literature review the researcher has conducted. The researcher has implemented the request/response-based stateless client like HTTP Client, but beneath, it works on the pub/sub communication style. As a

communication medium, the researcher has used the Redis Streams. Spring Boot provides a spring-boot-starter-data-redis library, which can be used to build the solution [27]. That gives high-level and low-level abstractions for integrating with the Redis server.

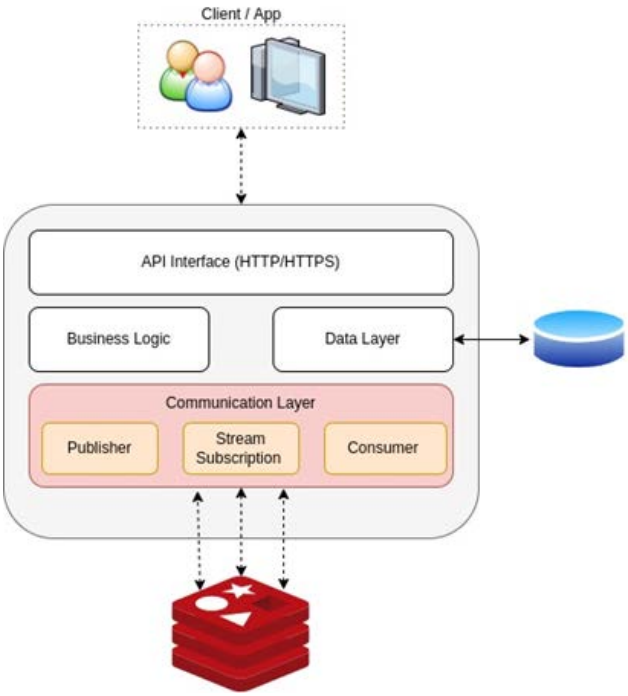

Fig. 2. High-level component architecture

The researcher has segregated one microservice into four main categories (see Fig. 2). The main category is the API interface, the programming component that the microservice communicates with external parties. In this implementation, the interface is neither touched nor changed because the external parties use that and cannot adhere to the implementations by changing their applications. Other categories are the business logic part and the data access layer. Business logic is the main part that contains the functionalities of the microservices. Most of the microservices are segregated from these business functions. Another category is the data layer, the part that communicates with the data sources, such as the database. We have developed the communication layer with three sub-programming components: publisher, stream subscription, and consumer.

1) Publisher: Responsible for sending the message to the correct microservice.

2) Stream subscription: Decides the destinations of the messages.

3) Consumer: Responsible for receiving messages.

A. Request Handling in Microservices

3rd party client sends the HTTP request by invoking the API exposed from the microservice A, as per Fig. 3. The researcher has used Spring Boot REST Controller to expose the API, which gives the developer the to enable restful web services. When the microservice starts, it establishes the Unix socket-based connection to the Redis server using the TCP port with the configured IP and the port. After that, create the subscription using the stream keys, which are also configured in the properties file. Stream keys belong to the other microservices that microservice A needs to send the messages. After receiving the request from the 3rd party client, microservice A starts processing that request and makes the EventStructure object using the request details and processed details. EventStructure is the object sent as a message to microservice B to get more details. That object contains all the HTTP message details such as HTTP method, parameters, headers, body, client details, publisher details, etc. After processing the HTTP request, the microservice decides which microservice needs to be called to provide the client's correct response. Based on that, microservice A chooses the correct stream key and publishes the message. When publishing the EventStructure object, it is serialized and passed as a byte buffer record. Because of this mechanism, network consumption can be vastly reduced when transferring data. Each published event has a unique event ID. Afterward, that ID and EventStructure object will be stored as a callback reference in a HashMap to process the response.

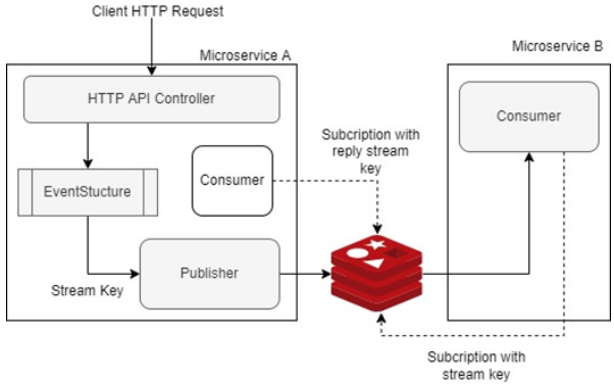

Fig. 3. How request serve

B. Response Handling in Microservices

After processing, the business logic response can be set to the EventStructure object. Publisher and reply stream key data can be retrieved from the EventStructure object and can publish the response using those data. Consumers placed in the microservice A can identify the client data by receiving the stream event and mapping the response data with the HashMap data, which is stored earlier. After processing, the response API controller can send the response back to the client using the HTTP protocol as per Fig. 4.

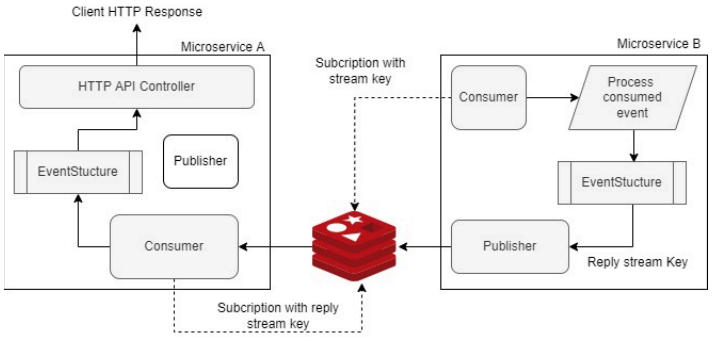

Fig. 4. How response serve

C. Quality Attributes

Quality attributes are the most vital in Microservice architecture. Most people transition to microservice architecture considering scalability, performance, maintainability, traceability, and availability [28]. When implementing the new solution, the quality attributes are preserved and improved.

1) Scalability: Proposed solution can be horizontally (scaling out) and vertically (scaling up) scalable. If this solution is deployed in the cloud-based VM, the VM specification can be increased at any given time and ultimately support vertical scaling. Thus, JVM has more resources to execute computations, and there is no barrier to the implantation. By performing horizontal scaling, people can add more microservice instances based on business needs. When adding the new application, it creates the subscription using the stream key. Redis server is responsible for delivering the messages solely to one subscriber, which means that the request is received only by one microservice. The Redis service covers service discovery and load balancing. Therefore, the developer does not need to ponder on it when scaling the applications. Hence, message duplication is not happening with this implementation, guaranteeing the exact message's delivery.

2) Maintainability: There is no impact on the overall maintainability of this implementation. Developers can use this implementation for internal communication instead of HTTP Clients. There's no requirement for maintenance in the internal load balancer for inter-service communication.

3) Traceability: This is the most important quality attribute in relation to technical support, as the troubleshooting support engineers need to know what has happened to the request and the response. The developed implementation supports end-to-end request/response tracing via the Redis server. If there's a need to trace the request and the response, the Redis GUI client can be installed after connecting to the Redis server of that client.

4) Availability: With this implementation, exact message delivery is guaranteed from the Redis Streams. Hence, availability can be achieved through this.

V. Results and Discussion

By critically reviewing the microservice architecture, the researcher analyzed that the impact of inter-service communication on performance is very high as a result of all the microservices deployed in the distrusted environment, making it mandatory to call each other over the network to produce the results. In this research, the researcher has proposed and implemented a solution that can be used to improve inter-service communication, ultimately bringing in overall application performance in terms of response time and throughput. HTTP inter-service communication and implemented solution have been deployed in a cloud VM-based environment to test and evaluate the application response time and throughput.

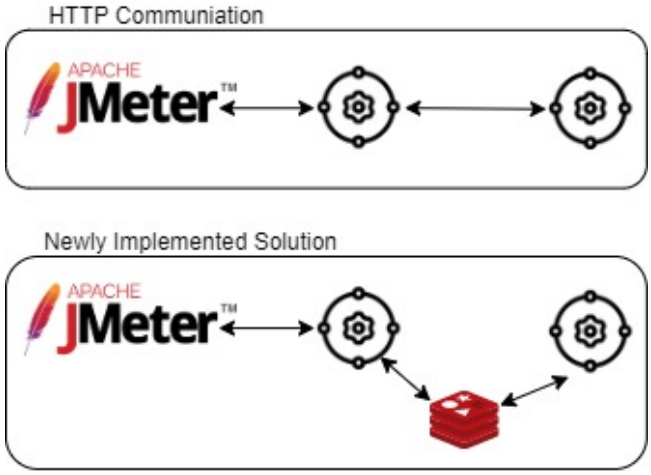

Fig. 5. Testing architecture

The Fig. 5 depicts the high-level system architecture of the load-testing environment. To generate the load, the researcher has used Apache JMeter, which is the most famous and vastly used tool in the industry, as well as in academic research [29]. Well-known cloud provider AWS (Amazon Web Services) has been used to deploy the systems [30]. Most enterprise software companies and enterprise-grade software are developed in the AWS cloud. Numerous scientific types of research are also conducted recently from the AWS cloud. Hence, the AWS platform is chosen in this instance as well to evaluate the system [31]. AWS EC2 is a virtual machine infrastructure as a service that contains the Intel Xeon processors with burstable for high frequency and a balanced memory/network and IO resources.

The T2 instance type has been chosen as it is a low-cost general-purpose instance category that provides better CPU performance for microservices and low-latency interactive applications [32]. T2.Medium EC2 instance type has been used to deploy the two microservices and JMeter, which has two virtual CPUs and 4GB memory on each. T2. A small instance type is used to deploy the Redis server, which contains one virtual CPU and 2GB of memory. All the VMs are provisioned in one virtual private network (VPN) and the same subnet under one security group. This network architecture can minimize network latency by calling the application through the same subnet and improves security by using the same security group.

In Fig. 6, straight arrows are the path that conducts the test for the common standard HTTP communication. The dotted arrows depict the newly implemented solution load test path.

The test is executed in two different methodologies,

1) Scenario A: Controlled the application's overall throughput and the request/response size and then measured the inter-service communication turnaround time.

2) Scenario B: Controlled only the request/response size and measured the throughput, overall application response time, and inter-service communication turnaround time.

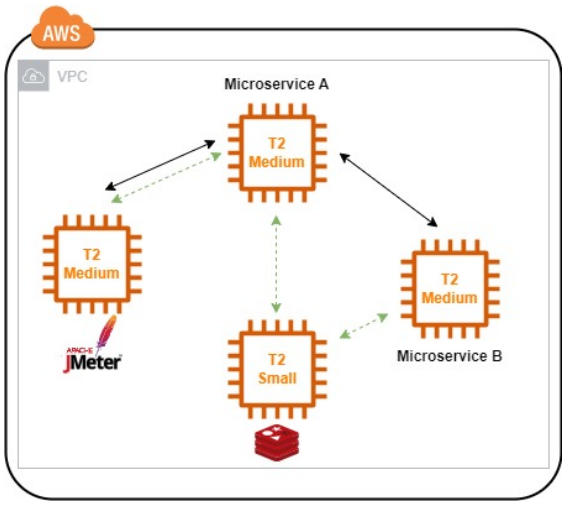

Fig. 6. Deployment architecture

A. Scenario A

Based on the studies conducted, the researcher has chosen different testing scopes to evaluate the system with existing systems. The researcher started with different throughput values and payload sizes. For each constant throughput value, the payload size has been changed as below, and traffic has been generated from the JMeter.

- Call the HTTP GET method from the microservice, and the backend microservice returns the 200 OK HTTP response code with the empty body.

Call the HTTP POST method with a 1KB size JSON payload, and the backend microservice returns the 200 OK HTTP response with a 1KB JSON format response.

Using the above test scenarios (Table I), the researcher has evaluated the HTTP communication method, inter-service communication turnaround time, and the newly implemented solution for inter-service communication turnaround time. Each test scenario was run for a time period of 1 hour and repeated three times, generating an average value of the inter-service communication.

Fig. 7 reflects the difference between the turnaround time on the proposed solution and the HTTP protocol implementations. When the throughput gets high, both the proposed solution and the HTTP protocol solution's turnaround time increase; when considering the payload size, it can be comprehended that both perform the same behavior. But in the all-test scenarios, the proposed solution's turnaround time is getting much lower than HTTP protocol implementation. In the HTTP protocol, a socket connection needs to be created for each connection to close the connection once the response is received. Furthermore, the network packets are neither in a binary method nor fully serialized. Therefore, when transferring the data packet through the network consumes considerable time. In the new implementation, Microservices has established a TCP socket-based connection with the Redis server. Hence, when Microservices starts, it acts as part of the particular Microservice. When sending data to other Microservices, it needs to be serialized and passed as a byte buffer record to reduce the usage of network resources. Due to that, the implemented solution turnaround time is less than the standard HTTP communication method.

B. Scenario B

In this scenario, only the payload sizes have been controlled and evaluated for the application's overall response time and the Microservice's inter-service communication turnaround time. The researcher has conducted this test scenario in the same cloud environment, and for the payload size, only 1KB sized JSON payload, 5KB sized JSON payload, and URL were chosen. To capture the overall application response time, JMeter listeners are being added. The inter-service communication turnaround time has been calculated by processing the logs. Each test scenario was run for 1h and continued three times to get the average value of the inter-service communication turnaround time, overall application response time, and application throughput.

TABLE I. TEST SCENARIOS

Number	Test Case Scenario
1	Controlled the throughput to 10TPS and send HTTP GET request
2	Controlled the throughput to 10TPS and send HTTP POST request
3	Controlled the throughput to 100TPS and send HTTP GET request
4	Controlled the throughput to 100TPS and send HTTP POST request

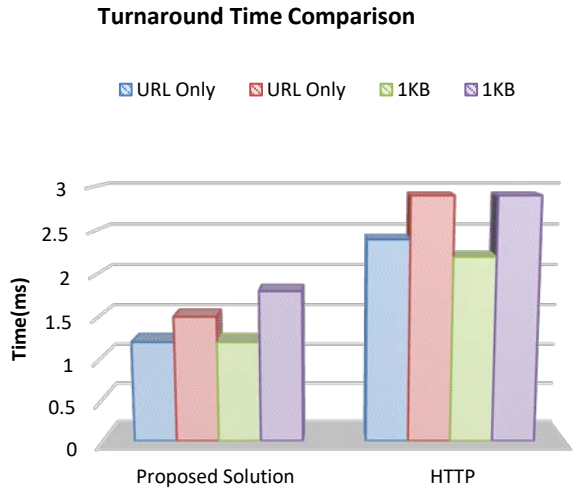

Fig. 7. Turnaround time comparison chart

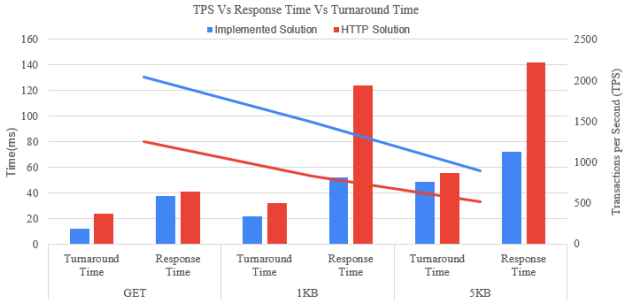

Fig. 8. TPS / response time / turnaround time comparison chart

As in Fig. 8 it can be observed that with the increase of the payload size, throughput is getting deceased. It is a typical network behavior that, when packets get heavy, will decrease the overall network performance. Hence, the response time also increases when the payload size increases. By comparing the implemented solution and the generic HTTP interservice communication method, it can be seen that in all the cases, response time gets better in implemented solution compared to the HTTP communication method.

Critically evaluating the above diagram, it can be concluded that;

$$\text{Inter} - \text{service communication turnaround time} \propto \text{Application response time}$$

Inter-service communication turnaround directly impacting to the whole application response time. This means that if some systems took more time to communicate between services, then overall response time will become high on that system due to inter-service communication.

$$\text{Payload size} \propto 1/\text{Throughput}$$

Request and response payload size impacting to the system throughput because transferring large network packets will take some considerable time between services. Hence response time will be getting increased. With the results of that, overall system throughput will be getting decreased.

VI. Conclusion and Future Work

In Microservice architecture, all the services are deployed as independent services in a distributed environment. However, unlike in monolithic software, the data must be derived through the network call to share the data between services. As a result of that, additional latency will be added to the overall application response time. Most software companies are faced with issues related to performance in terms of response time and throughput when migrating their monolithic architecture to microservice-based architecture. However, there is a capacity-wise and cost-wise advantage by scaling required services when necessary.

This research focuses on finding a solution to reduce the inter-service communication time between services. The initial studies found that most of the existing protocols take time for connection establishment and connection closure when sending and receiving the response. Besides, sending massive payloads will cause additional latencies. We have implemented the solution by addressing the above-mentioned problems and reducing latency when communicating between the services. We have used the Redis Stream data structure and built the request/response-based message-passing solution for inter-service communication. A TCP-based socket connection is created when the microservice starts. When sending the payload, it will be serialized and sent as a protocol buffer. Redis server is responsible for the exact message delivery based on the subscription and the stream key. The test scenarios are conducted by deploying the implemented solution in the AWS cloud-based VMs, and the system is evaluated against the Spring Boot standard implementation. Test results depict that the implemented solution performs well in terms of application response time and throughput. This research will continue to find a cloud-native solution to gain more performance and maintainability.

References

[1] S. Weerasinghe and I. Perera, “An exploratory evaluation of replacing ESB with microservices in service oriented architecture,” presented at the International Research Conference on Smart Computing and Systems Engineering, Sep. 2021.

[2] A. Makris, K. Tserpes, and T. Varvarigou, “Transition from monolithic to microservice-based applications. Challenges from the developer perspective,” Open Res. Eur., vol. 2, p. 24, Feb. 2022, doi: 10.12688/openreseurope.14505.1.

[3] Chris Richardson, Microservices patterns. Manning Publications, 2018.

[4] J. Kazanavicius and D. Mazeika, “Migrating Legacy Software to Microservices Architecture,” in 2019 Open Conference of Electrical, Electronic and Information Sciences (eStream), Vilnius, Lithuania, Apr. 2019, pp. 1–5. doi: 10.1109/eStream.2019.8732170.

[5] Christy Sibi Pachikkal, “Interservice Communication in Microservices,” Int. J. Adv. Res. Sci. Commun. Technol.

[6] S. Raje, “Performance Comparison of Message Queue Methods”, doi: 10.34917/16076287.

[7] “Spring Boot.” https://spring.io/projects/spring-boot.

[8] “Eclipse Vert.x.” https://vertx.io/.

[9] “Moleculer - Progressive microservices framework for Node.js,” Moleculer - Progressive microservices framework for Node.js. https://moleculer.services/index.html.

[10] A. Aslam, “Go Micro.” [Online]. Available: https://github.com/asim/go-micro.

[11] Sidath Weerasinghe and Indika Perera, “Taxonomical Classification and Systematic Review on Microservices,” Int. J. Eng. Trends Technol. - IJETT, Accessed: Jul. 30, 2022. [Online]. Available: https://ijettjournal.org/archive/ijett-v70i3p225.

[12] L. D. S. B. Weerasinghe and I. Perera, “An exploratory evaluation of replacing ESB with microservices in service-oriented architecture,” in 2021 International Research Conference on Smart Computing and Systems Engineering (SCSE), Sep. 2021, vol. 4, pp. 137–144. doi: 10.1109/SCSE53661.2021.9568289.

[13] 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), Budapest, Hungary, Nov. 2018, pp. 000149–000154. doi: 10.1109/CINTI.2018.8928192.

[14] V. Velepucha and P. Flores, “Monoliths to microservices - Migration Problems and Challenges: A SMS,” in 2021 Second International Conference on Information Systems and Software Technologies (ICI2ST), Quito, Ecuador, Mar. 2021, pp. 135–142. doi: 10.1109/ICI2ST51859.2021.00027.

[15] D. Kuryazov, D. Jabborov, and B. Khujamuratov, “Towards Decomposing Monolithic Applications into Microservices,” in 2020 IEEE 14th International Conference on Application of Information and Communication Technologies (AICT), Tashkent, Uzbekistan, Oct. 2020, pp. 1–4. doi: 10.1109/AICT50176.2020.9368571.

[16] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, “From monolithic systems to Microservices: An assessment framework,” Inf. Softw. Technol., vol. 137, p. 106600, Sep. 2021, doi: 10.1016/j.infsof.2021.106600.

[17] O. Zimmermann, “Microservices tenets: Agile approach to service development and deployment,” Comput. Sci. - Res. Dev., vol. 32, no. 3–4, pp. 301–310, Jul. 2017, doi: 10.1007/s00450-016-0337-0.

[18] M. Ya. Afanasev, Y. V. Fedosov, A. A. Krylova, and S. A. Shorokhov, “Performance evaluation of the message queue protocols to transfer binary JSON in a distributed CNC system,” in 2017 IEEE 15th International Conference on Industrial Informatics (INDIN), Jul. 2017, pp. 357–362. doi: 10.1109/INDIN.2017.8104798.

[19] L. N. T. Thanh, “SIP-MBA: A Secure IoT Platform with Brokerless and Micro-service Architecture,” Int. J. Adv. Comput. Sci. Appl., vol. 12, no. 7, p. 8, 2021.

[20] R. Corbel, E. Stephan, and N. Omnes, "HTTP/1.1 pipelining vs HTTP2 in-the-clear: Performance comparison," in 2016 13th International Conference on New Technologies for Distributed Systems (NOTERE), Jul. 2016, pp. 1–6. doi: 10.1109/NOTERE.2016.7745823.

[21] H. Bakri, C. Allison, A. Miller, and I. Oliver, "HTTP/2 and QUIC for Virtual Worlds and the 3D Web?," Procedia Comput. Sci., vol. 56, pp. 242–251, Jan. 2015, doi: 10.1016/j.procs.2015.07.204.

[22] M. S. Nyfløtt, "Optimizing Inter-Service Communication Between Microservices," p. 103.

[23] S. A. Asri, I. N. G. A. Astawa, I. G. A. M. Sunaya, I. M. R. Adi Nugroho, and W. Setiawan, "Implementation of Asynchronous Microservices Architecture on Smart Village Application," Int. J. Adv. Sci. Eng. Inf. Technol., vol. 12, no. 3, p. 1236, Jun. 2022, doi: 10.18517/ijaseit.12.3.13897.

[24] S. G. B. and G. R. S. N. S., "High Resilient Messaging Service for Microservice Architecture," Int. J. Appl. Eng. Res., vol. 16, no. 5, p. 357, May 2021, doi: 10.37622/IJAER/16.5.2021.357-361.

[25] X. Chen, F. Wang, J. Xu, D. Zhu, P. Tan, and J. Ma, "A distributed cache system based on Redis for high-speed railway catenary monitoring system," in 2020 Chinese Automation Congress (CAC), Shanghai, China, Nov. 2020, pp. 2048–2053. doi: 10.1109/CAC51589.2020.9326531.

[26] "RESP protocol spec," Redis. https://redis.io/docs/reference/protocol-spec/.

[27] "Spring Data Redis." https://spring.io/projects/spring-data-redis.

[28] T. Schirgi, "Architectural Quality Attributes for the Microservices of CaRE," p. 46.

[29] R. B. Khan, "Comparative Study of Performance Testing Tools: Apache JMeter and HP LoadRunner," p. 57.

[30] "AWS Lambda – Serverless Compute - Amazon Web Services," Amazon Web Services, Inc. https://aws.amazon.com/lambda/.

[31] "Security and Safety in Amazon EC2 Service – A Research on EC2 Service AMIs," Int. J. Innov. Technol. Explor. Eng., vol. 8, no. 6S4, pp. 736–738, Jul. 2019, doi: 10.35940/ijitee.F1149.0486S419.

[32] "Amazon EC2 T2 Instances – Amazon Web Services (AWS)." https://aws.amazon.com/ec2/instance-types/t2/.