DATA HANDLING AND MAINTAINING DATA CONSISTENCY IN SCALABLE REPLICATED MICRO-SERVICES

WMNKGTL Weerakoon[1] and Banage TGS Kumara
Faculty of Applied Sciences, Sabaragamuwa University of Sri Lanka, Sri Lanka
[1]*thamira1005@gmail.com*

Abstract- Monolithic Architecture helps to develop server-side enterprise applications. But, it views as a "big ball of mud". That indicates monolithic architecture has many drawbacks. Introduction of cloud based microservice architecture can solve these kind of drawbacks. Microservice architecture helps to scale an application. Most of the applications write less data than reading of that data. Scaling of read model and write model separately is very important. But, scaling applications using microservice architecture is very hard. Further, applications cannot simply use a local ACID (Atomicity, Consistency, Isolation, Durability) transaction. Read part is scaling to more replicas. Thus, maintenance of the data of all replicas in same level is important. Replication of read model and maintenance of data consistency which would provide experimental insight still needs to be developed. To bridge this gap, development of an architecture based on messaging with RabbitMq publish/ subscribe method, event sourcing and Command Query Responsibility Segregation (CQRS) with axon framework is used in this study. Evaluation of this architecture was done by replication of the read model using Docker and Docker-compose. Further, we have analysed data consistency in our experiments.

Keywords- Monolithic Architecture; Microservice architecture; Event sourcing;

I. INTRODUCTION

Monolithic Architecture helps to develop server-side enterprise applications. The major advantage of the monolithic architecture is that most applications typically have a large number of cross-cutting concerns, but in Monolithic applications all components are bundled together as a single application, which most people now have come to view as a "big ball of mud". That says monolithic architecture has many drawbacks (Richardson, 2017). Developers move to microservice architecture to solve these drawbacks.

Substantial development of Service Oriented Architecture leads to the rising of Microservice architecture and this focuses on specific aspects, such as decomposition to small services, scaling of the application, improve agile practices for development, independent deployment and separate testing methodologies, continuous delivery features and usage of infrastructure automation, decentralized data management and decentralized

governance among services. Existing researches investigate the characteristics of Microservice architectures. They defined the main facts that need to design Microservice architectures (Alshuqayran, et al., 2016), (Di Francesco, et al., 2017), (Pahl & Jamshidi, 2016).

Data consistency is the most critical part of the microservice architecture. Therefore, it is important to maintain the data consistency. Creation of data in an application happens less than reading of that data. For an example, let us consider a scenario of reserving an airline ticket. Here, writing operation is the reservation of the Seat. Only one person can do this reservation at a time of selected seat. However, many people access the system to check the availability of that selected seat.

This checking of the availability is the reading operation. Therefore, developer needs to pay particular attention to reading performance. However, writing and reading performances cannot be independently separated. Here, we can use CQRS pattern.

The CQRS principle advises separating writing operations from reading operations effectively. Writing part is identified as 'Command-Model' and reading part is identified as 'Query-Model' (Fowler, 2011).

Further, Command-Model must atomically publish events to the all Query-Model whenever their state changes. Event sourcing persists the state of a business entity. A new event will be added to the list of events with the change of state of business entity. Saving an event is just a single operation and especially atomic. The application has the ability to reconstruct the current state of an entity by replaying the events. In this paper, we propose an approach to handle data and maintain data consistency in scalable microservices using CQRS and event Sourcing. The rest of this paper is organized as follows. In section II, we present the problem and motivation. Section III discusses preliminaries. Section IV Methodology. Section V discusses our experiment and its evaluation. Finally, section VII concludes the paper

II. BACKGROUND AND MOTIVATION

Some business transactions span over multiple services. Thus, we need a mechanism to ensure data consistency across the services. For an example, let us consider the previous scenario of reserving an airline ticket. To reserve a ticket, a customer should have at least the credit amount

equals to air ticket price and service charges of the credit card. If the system approves the transaction without having the required credit amount by mistake, then it will be a critical error, which will violate the reliability of airline service. Since ticket, reserving data and customer's credit amount are in different databases the application cannot simply use a local ACID transaction. CQRS can be used to achieve ACID transaction in distributed microservice environment (Melnik, et al., 2012).

According to microservice architecture database per each Microservice is very important. Thus, if we replicate Microservices, every Microservice needs its own database. Assume we have an application with replicas and this application requires new data, which should be added through HTTP request. Once we add the new data to one replica, it needs to synchronize with remaining replicas Fig. 1 illustrates a scenario where synchronization of data is still in progress. In this instance, if someone tried to retrieve data from the third replica, then data retrieved will not be up to date as third replica is still in synchronization process. In that case, if doesn't have eventual consistency. Thus, this research is done to propose a new architecture to solve this problem.

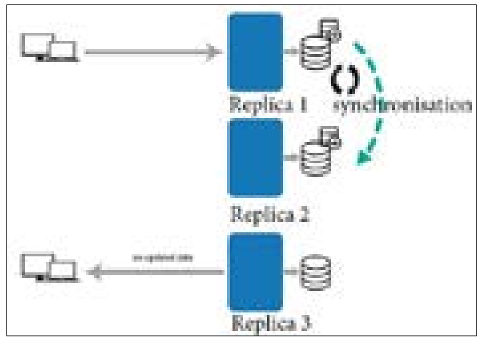

Figure 1. Command Query Responsible Segregation pattern

III. PRELIMINARIES

A. Command Query Responsible Segregation

CQRS is a pattern that use a different model to update information and read information. For some complex operation handling situations, this separation can be valuable. This helps to allocate resources independently to fulfil on-demand provisioning of computing resources. Figure 3 displays the CQRS pattern (Data read part and write part are separately denoted).

Fig 3. shows that application is divided to two parts. One for data reading and one for data writing to database.

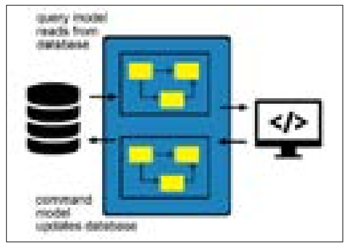
query model reads from database
</>
command model updates database

Figure 2. Database level Synchronisation

According to CQRS pattern data read part is called as query model and data write part is called as command model.

B. Command model (Write model)

Commands are responsible for introducing all changes to the system. They lead to change the state of the system. It indicates requests to domain. While request indicates command, it may be accepted or rejected. An accepted event leads to change in the database. Reject command indicate an exception and rollback the system to stable state. Command should not return any value. If there is a distributed system, command emit zero or more events being emitted to incorporate new facts into the system.

Through replicating the instances of read models and by balancing the load, scalable processing could be achieved. Each node could manage its own databases instance with the help of a complete model. Command need to update all the replicas at the same time to manage data consistency.

According to seat reservation system, following two commands help to state change.

- SeatReseveCreateCommand
- SeatReservationUpdateCommand

C. Query model (Read model)

Query is a READ operation. It responsible for reading the state of the system, analyse aggregates, get appropriate data to query. As the response, data is transformed to most useful format. It can be JSON/XML or HTML. According to CQRS theory, query model cannot make any change to the database. In order to achieve the scalability, replicate the read model and introduce database to each replica. It leads to manage request load. Requests can be distributed as a round robin method to all replicas. For some efficient read manner, we can use different kind of databases to this read model. For example, Redis is more efficient than MySQL in big data analysis because more speed is required to analyse more data. Some query models can be implemented with different databases according to requirements.

D. Events

Events are treated as notifications. If something happens, event is responsible to notify that to other interested parties. Command emit 'event' if there is any state change. If all the events are logged in a database, that events can be replayed and check all the state of change. According to seat reservation system two events emitted by commands are;

- SeatReservationCreateEvent
- SeatReservationUpdateEvent

E. Event Sourcing

Event Sourcing is a major part that helps to improve reliability and consistency when CQRS is considered. Event sourcing is a simple theory that logs the state changings. According to seat reservation system events are produced, that represent every change made in the system. Every change is logged according to event sourcing theory. If events are replayed from beginning to current stage that stored in the event store, it helps to understand, how system has changed and will get to the current state of the system. It is similar to technology used in version control systems such as GitHub, BitBucket. In version control systems commit log is used to understand how the code changed by contributors. If there is any conflict, user can replay and can get a stable state. A business object is persisted by storing a sequence of state changing events. Whenever an object's state changes, a new event is appended to the sequence of events. Since that is considered as a one operation, it is inherently atomic. An entity's current state is reconstructed by replaying its events.

F. Eventual consistency

When an application makes a change to a data item on one replica, that change has to be conveyed to the other replicas. Since the change conveyed does not take place at once, there is a time period in which some replicas have the most recent change while other replicas does not. At this time period replica will be mutually inconsistent. However, at the end the changes will be conveyed to all the replicas, and hence the term "eventual consistency". Following Fig 3. shows how eventual consistency is achieved in current study.

According to Fig 3. command handler distributes all new reservation request to the replicas in same time. Steaming and queue ensure all the request are distributing to the replicas.

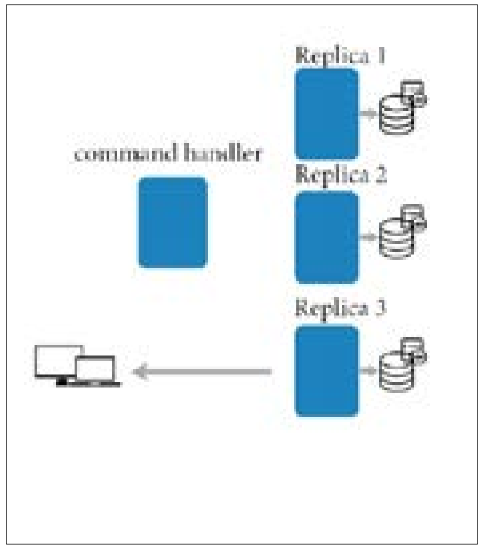

Figure 3. Data distribute to all query model replicas by command model

IV. METHODOLOGY

The purpose of this research is to describe effective and efficient way of "Data Handling and Maintaining Data Consistency in Scalable Replicate Microservices". For that, we applied the Microservice architectures for the proposed system. Then the system is tested for different architectures and most efficient architecture is selected.

In this research, seat reservation system that is applicable to places such as airport or cinema was first selected. This scenario represents a situation where more people access the system at the same time to check the availability of the seat. Some of the requests come to server to only check the availability but the possibility of reserving a seat is less. From that request, few of them reserve the seat. The people who visit the system to check the availability introduce more traffic to the system. An application was developed using CQRS pattern to meet with the requirement of above scenario.

To develop this main application CQRS pattern and Event Sourcing Methods were used. Following technologies were used in order to implement the architecture; Spring-boot 2, MySQL, Axon framework, RabbitMq, Docker and Docker-compose. Fig 4. shows the basic structure of the proposed architecture. Here, seat booking was selected as the core service which acts as the Command-Model. Seat Reserving acts as replicated Query-Model.

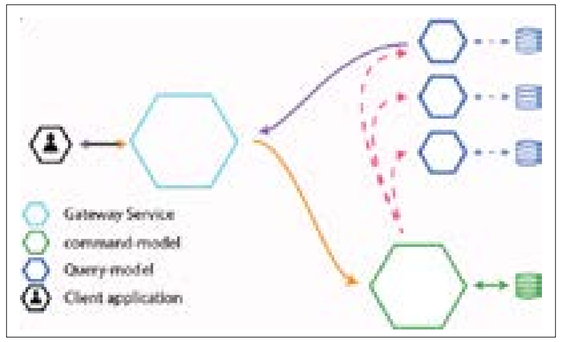

Figure 4. Basic Architecture

Microservice gateway pattern was the main architecture used to implement this system. Gateway service helped to select one instance of seat reservation service, that provides available seats. If a seat is booked from one service, all services need to be updated at the same time by publishing to multiple subscriber pattern. To accomplish this, two 'distribute messages pattern' were used. Those patterns were simple HTTP request and Message Streaming. The results of these two patterns were then compared. After comparison of results, problems related to that scenario were identified. The best pattern was selected and the efficiency of the pattern was maximized.

A. HTTP Request

Implementation of application was done using "Spring boot" web framework and "Rest Template" to test efficiency of simple HTTP messaging. As the first step of implementation, Spring RestController class was created to catch a message. The application then distributes the message to Query-Model. Query part catch the message and save it in the database. A loop is used to send thousands of requests to query module. The time of massage sent (t1) and saved (t2) should be recorded in the database. Then the latency (L) is calculated using following

(1) (Latency is the time taken for massage sending).

L = t2 - t1(1)

Graph between L and total time taken for send 1000 message is plotted.

B. Message Streaming

The procedure done using HTTP request can be reproduced using data streaming method. This carried out using spring data streaming with RabbitMq message broker. Results showed that data streaming method is the most suitable method for efficient and 'eventual consistent' data transaction. Thus, Message Streaming method is selected to implement CQRS and event sourcing.

V. RESULTS AND DISCUSSION

When implementing the Seat reservation system based on the API gateway pattern, we have to consider about the communication between microservices. When Scaling of Query model and command model is done, those models need to communicate to share the data using HTTP or message stream. Selecting the best communication method is the main part of this research. Fig 5. shows the Latency of two methods used to send 1000 messages using loop. Latency of Data streaming method is illustrated in yellow (Series2) colour line. Orange (Series1) colour line illustrates the latency of HTTP messaging. According to the results, HTTP shows comparatively low latency, so that method is efficient than Data streaming method.

Looping time in HTTP method is higher than the streaming method. According to the results looping time in stream is 707ms and looping time in HTTP is 8491ms. Hence, we consider about looping time stream method is better than HTTP method.

Delay time in HTTP method is lower than the streaming method. According to the results, delay time of streaming is 3266ms and maximum delay time of HTTP is 33ms. When we consider about delay, HTTP is better than streaming.

The difference between HTTP and streaming method is HTTP method waits for the response of recently sent message to send the next message and this is how HTTP method ensure the message is sent but, this is not the case in Streaming method. In Streaming method, message broker takes the responsibility of sending all the messages without failure. Thus, Delay in streaming method is negligible.

While communication is done using HTTP method, if there is a problem in server application, then requested messages can be lost. The lost messages have to be managed manually and failed messages have to be re-sent. But, in streaming process, if the server is down the messages are stored in queue in the message broker. So, we don't have to worry about the losing the messages. In real world scenario manual management of failed messages are not possible, because it will become problematic in scenarios such as first come first out scenarios.

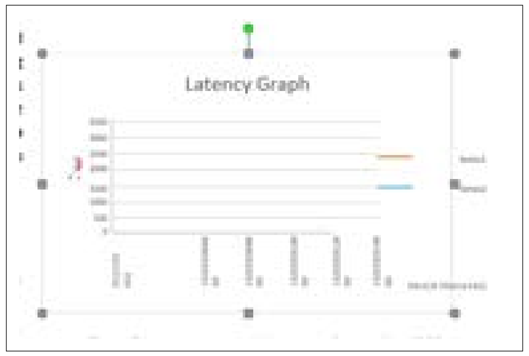

Figure 5. Latency and total time taken for sending 1000 messages

According to Seat reservation system requirement, many people can request for seat reservation at once. In that case request cannot be prioritized manually. Because of the competition to reserve seats, some requests can be lost. We can't handle failed requests without queue. But, streaming method sends all the requests to the queue and messages are stored in the queue until the communication is complete. Thus, no requests will be lost. Once a person requests to reserve a seat that person need to have the minimum credit amount to fulfil the requirement. When the person's credit amount is not sufficient, the request will fail. In this case the chance for reserving the seat should go to the next person who made the request after the first person. This is difficult to achieve by HTTP method. But, data streaming method can fulfil this requirement, because it saves requests in the queue.

Because of the scenario described above, the current developing architecture was carried out using the message streaming method with queue to avoid any complications. Furthermore, architecture need to enhance for update multiple Query-Models using multiple-queue at same time without update as database level synchronization describe in fig. 1.

Further, according first graph in fig. 6, behaviour of five replicas are the same. But, numerically each curves are different from each other. Second graph in fig. 6 illustrates part of the first graph. It shows that there is some delay between the replicas. That average delay is 3521 milliseconds.

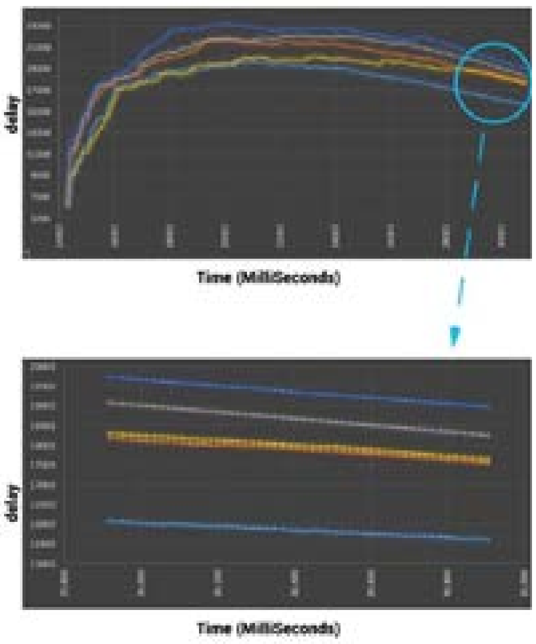

Figure 6 . Behaviour of five replicas

VI. CONCLUSION

This paper contributes to the research on CQRS and event sourcing for Data Handling and Maintaining Data Consistency in Scalable Replicated Microservices. According to the result of this study, we can fulfil below functions. Management of data consistency, replication of unlimited query models and management of eventual consistency. Data inconsistency could happen while communication between command model and query replicas. To prevent that we have to use message streaming process with queues. Predefined queues make limitations for replication of query models and unused queues waste the resources. Generated queues help to reduce wastage of resources. Further, if there is any state change, command model publishes event to all the replicas in same time using streaming, which is better than database level synchronization of replicas.

As the future work, we plan to reduce the average delay to a minimum level using reactive manifesto. Future more, if there is a new instance introduce to the application as a replica of query-model also update as same as other replicas using replaying past states.

ACKNOWLEDGEMENT

For the success of this study it would not have been possible without the kind support, encouragement and help of many individuals and organizations. I would like to extend my sincere thanks to all the parties who joined with me to make this research success. I am highly indebted to Dean Prof. Udaya Rathnayaka, Faculty of Applied Sciences, Sabaragamuwa University of Sri Lanka, and all other staff members of the department of Computing and Information Systems in the Faculty of Applied Sciences, Sabaragamuwa University of Sri Lanka, for their help and wishes for the successful completion of this research.

REFERENCES

Alshuqayran, N., Ali , N. & Evans, R., 2016. A Systematic Mapping Study in Microservice Architecture. Macau, China, IEEE.

Di Francesco, P., Malavolta, I. & Lago, P., 2017. Research on architecting microservices. Gothenburg, Sweden, IEEE. Fowler, M., 2011. CQRS. [Online] Available at: https://martinfowler.com

Melnik, G. et al., 2012. Exploring CQRS and Event Sourcing A journey into high scalability, availability, and maintainability with Windows Azure. s.l.:Microsoft Developer Guidance.

Pahl, C. & Jamshidi, P., 2016. Microservices: A Systematic Mapping Study. Rome, Italy, s.n., pp. 137-146 . Richardson, C., 2017. Monolithic Architecture. [Online] Available at: http:// microservices.io/index.html

The delay could affect the architecture to become vulnerable. Therefore, as a future enhancement it is needed to focus on steps to decrease this kind of delays.