Web Interface for Distributed Transaction System

Suraj Godage, T Rohith Kumar, Hardik Pandya, Shubham Bhosale, Rushali Patil
[1,2,3,4,5] dept. of computer engineering Army Institute of Technology.

Abstract:
This research delves deep into the saga pattern, which proves to be an effective approach for managing local se- quential transactions across distributed microservices. However, the problem of isolation lacking in the saga pattern can result in incorrect commits on databases due to unfinished transactions. To address this issue and further enhance existing solutions like transaction management protocols (e.g., two-phase commit), this study introduces innovative enhancements, namely quota cache and commit-sync service. These enhancements enable specific operations between database layers, effectively preventing invalid or incomplete commitments on the main databases. An experi- mental test was conducted to evaluate and check the effectiveness and performance of a microservices-based e-commerce system, revealing that this novel approach successfully handled both regular scenarios and exceptions, addressing isolation concerns. In the event of service failures, compensation transactions were executed to undo adjustments made solely within the caching layer. After ensuring all processes were correctly completed, the alterations were committed back into the database. Although promising results were observed, further investigation is required for optimization before widespread adoption as an industry- standard approach.
Keywords: microservices, distributed transaction, two-phase commit, SAGA.
DOI: 10.24297/j.cims.2023.14

1. Introduction

When building a website using a microservice architecture, it is crucial to implement distributed transaction patterns to ensure smooth and efficient transaction handling across the system[1]. However, migrating an application from a monolith to a microservice architecture is an intricate undertaking that demands substantial time, effort, and careful consideration. It is a multifaceted process that is prone to errors, making it imperative to employ appropriate tools that facilitate and guide the decomposition process [2][3]. And hence to build a robust E-commerce website, leveraging Microservices de- veloped with NodeJS in the backend is highly recommended REST APIs will facilitate seamless connection between these Microservices. Proper event handling, including buy- ing, completion, and failure scenarios, requires the utilization of message queue middleware. while effective transaction management across multiple services necessitates the use of Orchestration approach of SAGA pattern [4][5].These message queues, coupled with a

SAGA-based quota-cache database connected through REDIS server technology, significantly enhance system throughput. By employing a commit sync pattern, optimal performance results can be achieved, ensuring efficient event processing and isolation handling within the system[6]. The SAGA pattern proves to be effective for managing distributed transactions in microservices. But the current SAGA pattern faces issue of lack of isolation, which states that reading and writing from uncommitted transaction is not prohibited in it. And hence enhancing the SAGA pattern with queue middleware and a quota cache database server becomes essential to prevent issues like uncommitted reads and writes, ensuring data consistency and reliability throughout the microservices architecture[8][9].

2. Proposed Methodology

The system is composed of five microservices, namely the WarehouseService, OrderService, BillingService, Ship- pingService, and CustomerService. Each microservice has its own dedicated database. Leveraging the power of NodeJS technology, these microservices were developed, inheriting and implementing the relevant use cases seamlessly [10][11]. Communication with the microservices occurs through a REST API, utilizing the simplicity and versatility of the HTTP protocol. This architecture ensures modularity and allows for efficient interaction between the different components, enabling the system to scale and evolve effectively[12][13]. This system offers users a seamless online purchasing ex- perience, enabling them to select desired products, payment methods, and preferred shipping options. Comprised of es- sential components such as WarehouseService, OrderService, BillingService, and ShippingService, among others, the system supports long transactions with the flexibility of both Ware- houseBeforeBilling and BillingBeforeWarehouse approaches, as illustrated in the Figure 1.

The flow commences with Warehouse Services procuring goods, which are subsequently stored as " IN PROGRESS" orders in the Order services. This paper outlines a comprehen- sive process for placing and fulfilling orders, starting with the retrieval of items. If the item retrieval fails, the placed order is tagged as " FAILED." The subsequent step involves payment validation by the Billing Service. Upon successful payment, the transaction is processed, while any payment failure halts the flow at this stage. To facilitate efficient event processing, the system leverages a message broker middleware namely apache kafka which is also an open source software [18]. Kafka enables the publishing and listening of message streams, capable of handling numerous number of messages per second, thereby supporting high throughput applications. Additionally, Kafka' s durability feature ensures message persistence by storing them on disk

[19]. Overall, the described process and technology stack illustrate a robust framework for managing e- commerce orders, emphasizing fault tolerance, scalability, and reliable event streaming capabilities [20].

The message queue apache kafka middleware plays a crucial part in managing both failure and completion events within a distributed event based architecture. It enables seamless mes- sage sending and also message receiving between and among all the microservices while maintaining the order of requests for accurate final commitment. Various forms of message queue middleware were utilized, with Apache Kafka serving as the central component. This middleware effectively handles potential exceptions that may arise from failures during the process, particularly when service validation is necessary.

The introduction of an in-memory quota cache proves to be a valuable addition to the microservices architecture. It opti- mizes data access by providing faster retrieval of frequently used data, thereby reducing the reliance on frequent database access. However, in the event of any issues during Warehouse- Service actions that could potentially affect the main database and impact client orders, compensation transactions are swiftly deployed to undo these changes and cancel the order.

The implementation proposes an in-memory quota cache can significantly enhance the performance and reliability of microservices-based systems. It can be customized to cater to the specific requirements of different applications. In the improved version, the CRUD (Create, Read, Update, Delete) operations have been shifted from the primary database layer to an intermediate cache tier. This relocation further optimizes system performance and reinforces reliability within the mi- croservices architecture.

In computational contexts, caches serve as advanced data storage components that significantly reduce access time to primary memory systems. Among the different cache tech- niques, quota caching stands out as an effective in-memory databasing approach. It ensures the accuracy of read and write operations through CRUD actions, preventing any potential

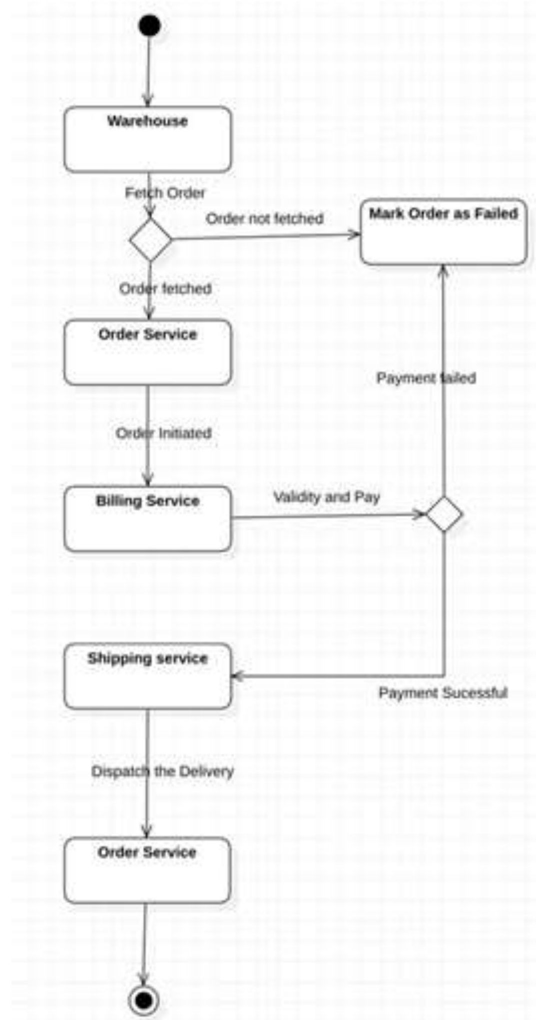

Fig. 1. State Diagram

incorrect commit disasters that could affect the main database. By implementing a cache system, overall performance is maximized by minimizing latency across multiple databases or services.

This paper introduces the implementation of an in-memory quota cache, specifically designed to enhance the performance of microservices. Compared to the baseline implementation using the saga pattern, the proposed in-memory quota cache demonstrates improved performance and efficiency. By lever- aging the benefits of cache systems, microservices can achieve faster access to data, reduced latency, and optimized resource utilization. Overall, the utilization of an in-memory quota cache in microservices architecture presents a promising solu- tion for

maximizing performance and ensuring data accuracy. By freeing the main database from incorrect commit disasters and leveraging the power of cache technology, this approach offers significant improvements in the overall performance of microservices- based systems.

3. Result

A. Test 1 : Evaluating the SAGA in use case 1

To comprehensively assess the Baseline Standard System' s capabilities, we conducted Test 1, focusing on Use case 1. As per the SAGA pattern' s guidelines, the orchestrator module assumes responsibility for capturing the crucial " buy-event" [14][15]. Subsequently, the respective microservices come into play, seamlessly managing the event processing workflow. In strict adherence to the pattern, the WarehouseService takes the lead in the initial stage by meticulously retrieving the precise amount of goods required. This critical operation unfolds within the database, as denoted by the distinctive " logger name" field [16][17]. Continuing with remarkable precision, the OrderService takes charge, skillfully initiating the order process. Building upon this momentum, the BillingService enters the stage, expertly executing payment validation. Should the validation encounter any hiccups, the workflow promptly concludes, and the order assumes the " FAILED" status.

Fortunately, in this particular Use case, the validation pro- ceeds without a hitch, ensuring a flawless sequence of events. Moving into the next phase, the BillingService diligently collects the requisite payment, instilling trust and confidence in the transaction. Once this financial milestone is firmly established, the ShippingService springs into action, promptly dispatching the eagerly anticipated delivery in accordance with the customer' s preferences. Lastly, the OrderService meticulously finalizes the order, leaving no detail unattended. From updating the order status to recording the shipment ID, quantity, and overall order status, every aspect is diligently managed to ensure a comprehensive and satisfactory customer experience. This meticulous evaluation of the Baseline Stan- dard System in Use case 1 highlights its remarkable effective- ness in orchestrating a flawless and error-free order processing flow. By seamlessly coordinating the various microservices, this standardized approach showcases its inherent reliability and aptitude for delivering impeccable results.

B. Test 1: Testing the Optimized System with Use case 1

In the enhanced SAGA pattern, Steps 1 and 2 exhibit a striking resemblance to the original SAGA implementa- tion, with the WarehouseService and BillingService embracing the suggested

approach. However, significant enhancements have been introduced. One notable improvement lies in the utilization of Redis, an in-memory cache, for data access instead of relying solely on the database. This innovative approach eradicates the necessity for transactions on the primary database during these processes, leading to enhanced efficiency and performance. By leveraging Redis as the data store, the system achieves optimal speed and responsiveness, ultimately resulting in a more streamlined and robust execution of Steps 1 and 2 in the SAGA pattern. To provide a visual representation, Figure 2 illustrates the initial request of items the customer wishes to purchase. This request is received in the Kafka message broker, accompanied by crucial details such as product ID, quantity, customer ID, price, and rating, as depicted in Figure 3.

Fig. 2.

Fig. 3.

Step 1 entails the WarehouseService checking the availabil- ity of the desired items in stock, as illustrated in Figure 4. If the items are available, the warehouse-check status is updated as ” true” ; otherwise, it is marked as ” false.” The updated warehouse-check status is then transmitted as a message via the Kafka message broker, as showcased in Figure 5.

Fig. 4.

Moving on to Step 2, the message received in Figure 5 is forwarded to the OrderService, where a unique order ID is assigned to the requested order, as demonstrated in Figure 6. Subsequently, the message, now containing the unique order ID, is published in the Kafka message broker, as depicted in Figure 7.

Step 3 commences with the initiation of the BillingService. In this phase, payment validation transitions from the database to the in-memory cache. The payment status is updated as " true" if the payment is successfully processed, and " false" otherwise, as illustrated in Figure 8. A message is then

Fig. 5.

Fig. 6.

Fig. 7.

published in the message broker, carrying the payment status to the Shipping microservice, as shown in Figure 9.

Fig. 8.

Finally, Step 4 involves the ShippingService, which verifies the payment status and warehouse status. Only if both condi- tions are true, the order is confirmed, shipped, and marked as a success. The order status is updated accordingly, with the

Fig. 9.

shipping status marked as " dispatched," as demonstrated in Figure 10. After the task is finished, the orchestrator module promptly dispatches the completion event to the designated message broker. Upon receipt of the " completion-event" mes- sage, both the

WarehouseService and BillingService swiftly initiate the specified transactions, directly interacting with the database. It is worth highlighting that both the Warehous- eService and BillingService have seamlessly transitioned their update processes to the in-memory cache. Consequently, upon receiving the completion event message, they efficiently carry out the required transactions on the database without delay, ensuring optimal performance and streamlined operations. This improved implementation of the system, as demonstrated in Test 1, showcases the improved efficiency and performance achieved by leveraging the improved Saga pattern. By in- tegrating advanced technologies and optimizing data access, the system achieves seamless coordination and execution of transactions, ensuring a robust and reliable order processing workflow.

Fig. 10.

C. Test 2: Testing the Saga with Use case 2

This Test aims to showcase the event processing capabilities of an e-commerce microservices-based system, specifically focusing on a payment exception Use case. Both versions of the system will be tested using the same workflow steps outlined by the standard saga pattern. In the first two phases, the system retrieves the required products and sets up the order, following the standard saga pattern. However, a significant deviation occurs in Step 3, where an excessively high sum is supplied, surpassing the consumer’ s affordability. As a result, the payment validation fails, triggering an exception in the process. To effectively address this exception, the system incorporates rollbacks in Step 4, which initiate the process of reverting the fetched products to their original state. By executing these rollbacks, the OrderService ensures that the order is completed, albeit marked as unsuccessful. This meticulous approach guarantees data consistency and integrity by maintaining the integrity of the fetched products and upholding the system’ s overall reliability.

Fig. 11.

D. Test 2: Testing the Optimized System with Use case 2

In the optimized system, the initial phases mirror those of the baseline standard version. The order request is received in message form through the Kafka message broker, which forwards it to the WarehouseService. The WarehouseService checks its database for product availability, updates the ware- house status accordingly, and publishes a message in the message broker, which is subsequently routed to the OrderSer- vice. As demonstrated in the previous Test, the OrderService assigns a unique order ID to the requested order. Step 3 commences with the initiation of the BillingService, where payment validation transitions from the database to the in-memory cache.

As per the defined test Use case, the payment status is updated as " false," necessitating the execution of compen- sating transactions. Figure 11 visualizes this update and the subsequent publishing of a message in the Kafka message broker. In Step 4, as illustrated in Figure 12, the retrieval of items from the WarehouseService is highlighted, utilizing a GET request. The retrieved data is then stored in the Redis cache server, effectively caching it for subsequent transactions. Notably, these subsequent transactions operate solely on the data stored in the Redis cache database server, bypassing the database entirely. By employing the improved saga pattern, the Optimized System ensures robustness and flexibility in handling exceptional Use cases, such as payment exceptions. Through efficient data caching and localized transactions, the system guarantees data consistency while providing an optimized and resilient order processing workflow.

Upon closer examination of the timestamps in the last two steps, a remarkable similarity becomes evident, suggesting that the system effectively transmitted the failure event and concluded the order simultaneously. This synchronization is depicted visually in Figure 13, illustrating how the Warehous- eService adeptly utilizes the in-memory cache to compensate for the fetched products, all while ensuring the integrity of the original database remains unaffected. It is worth noting that the logs provide supporting evidence that the enhanced

Fig. 12.

recommended solution excels at handling exceptions and elim- inates the necessity for rollbacks to the database in the event of errors.

4. Conclusion

This paper introduces a novel solution that harnesses the power of temporary commit sync services and caches, ef- fectively transferring transactions from the database layers to the memory layers. By adopting this approach, the solu- tion ensures secure and reliable execution of CRUD (create- read-update-delete) operations, significantly mitigating the risk of incorrect commits on the primary databases. Embracing this method within the saga pattern guarantees a robust and consistent operation, reducing the likelihood of conflicts and errors. Node.js is known for its single-threaded nature, which simplifies the management of multiple threads. Unlike the Spring Boot world, where Java web applications typically run on multiple threads, Node.js relieves you from the com- plexities associated with thread management and hence this research delas with building the existing enhanced saga using NodeJS as bakcend framework instead of previous research that used spring boot [21]. we have meticulously conducted extensive research and testing to showcase the solution’s efficacy, emphasizing its potential benefits for the realm of distributed systems. This research serves as a crucial resource for ensuring the safe and dependable functioning of systems that employ the saga pattern, providing valuable guidance to practitioners in the field. To maintain eventual consistency among all microservices, any modifications that solely impact the cache levels undergo compensation through a dedicated compensation transaction in the event of a failure. This compensatory action is seamlessly facilitated by employing delayed database commit, efficiently

managed through the message broker. This meticulous attention to detail ensures the attainment of optimal system performance while preserving data integrity and consistency.

References

1. Yamina Romani, Okba Tibermacine, Chouki Tibermacine, Towards Migrating Legacy Software Systems to Microservice-based Architectures: a Data-Centric Process for Microservice Identification, DOI: 10.1109/ICSA-C54293.2022.00010, ISSN : 2768-4288 Carrasco, B. V. Bladel and S. Demeyer, " Migrating towards mi- croservices: Migration and architecture smells" , Proceedings of the 2nd International Workshop on Refactoring, pp. 1-6, 2018.
2. M. Gysel, L. Ko¨lbener, W. Giersche and O. Zimmermann, " Service cutter: A systematic approach to service decomposition" in Service- Oriented and Cloud Computing, Springer, pp. 185-200, 2016.
3. Umakant Dinkar Butkar, et.al "Accident Detection and Alert System (Current Location) Using Global Positioning System" JOURNAL OF ALGEBRAIC STATISTICSVol. 13 No. 3 (2022) e-ISSN: 1309-3452. Retrieved from https://publishoa.com/index.php/journal/article/view/591
4. Krishna Mohan Koyya, B Muthukumar, A Survey of Saga Frame- works for Distributed Transactions in Event-driven Microservices, 2022 Third International Conference on Smart Technologies in Com- puting, Electrical and Electronics (ICSTCEE), DOI :10.1109/IC-STCEE56972.2022.10099533, Electronic ISBN: 978-1-6654-5664-7
5. Mr. Umakant Dinkar Butkar, Manisha J Waghmare. (2023). Novel Energy Storage Material and Topologies of Computerized Controller. Computer Integrated Manufacturing Systems, 29(2), 83–95. Retrieved from http://cims-journal.com/index.php/CN/article/view/787.
6. Pan Fan, Jing Liu, Wei Yin, Hui Wang, Xiaohong Chen and Haiying Sun, " 2PC*: a distributed transaction concurrency control protocol of multi- microservice based on cloud computing platform" , Journal of Cloud Computing: Advances Systems and Applications, 2020.
7. Hector Garcia-Molina and Kenneth Salem, " Sagas" , ACM Sigmod Record, vol. 16, no. 3, pp. 249-259, 1987.
8. Evgeny Volynsky, Merlin Mehmed, Stephan Krusche, Architect: A Framework for the Migration to Microservices, 2022 International Conference on Computing, Electronics &

Communications Engineering (iCCECE), INSPEC Accession Number :22027055, DOI: 10.1109/iC-
9. CECE55162.2022.9875096
10. Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe and Ferhat Khendek, " Deploying microservice based applications with ku- bernetes: Experiments and lessons learned" , 2018 IEEE 11th Interna- tional Conference on Cloud Computing (CLOUD), pp. 970-973, 2018.
11. Chaitanya K Rudrabhatla, " Comparison of event choreography and orchestration techniques in microservice architecture" , International Journal of Advanced Computer Science and Applications, vol. 9, no. 8, pp. 18-22, 2018.
12. Ernst Oberortner, Uwe Zdun and Schahram Dustdar, " Domain-specific languages for service-oriented architectures: An explorative study" , European Conference on a Service-Based Internet, pp. 159-170, 2008.
13. Sahin Aydin, Cem Berke C¸ebi, Comparison of Choreography vs Or- chestration Based Saga Patterns in Microservices, 2022 International Conference on Electrical, Computer and Energy Technologies (ICE- CET), DOI: 10.1109/ICECET55527.2022.9872665, INSPEC Accession Number 22028506
14. Umakant Dinkar Butkar, Dr. Nisarg Gandhewar. (2022). ALGORITHM DESIGN FOR ACCIDENT DETECTION USING THE INTERNET OF THINGS AND GPS MODULE. Journal of East China University of Science and Technology, 65(3), 821–831. Retrieved from http://hdlgdxxb.info/index.php/JE_CUST/article/view/313
15. W. Andreas et al., " Model-as-You-Go for Choreographies: Rewinding and Repeating Scientific Choreographies" , IEEE Transactions on Ser- vices Computing, vol. 13, no. 5, pp. 901-914, 2020.
16. G. Anushri, P. Panagiotopoulos and F. Bowen, " An Orchestration Approach to Smart City Data Ecosystems" , Technological Forecasting and Social Change, vol. 153, 2020.
17. Anis Boubaker, Hafedh Mili, Yasmine Charif, Abderrahmane Leshob, Methodology and Tool for Business Process Compensation Design, 2013 17th IEEE International Enterprise Distributed Object Comput- ing Conference Workshops, Electronic ISBN: 978-1-4799-3048-7, DI: 10.1109/EDOCW.2013.23
18. A. Boubaker, H. Mili, A. Leshob, and Y. Charif. A Value-Oriented Approach to Business Process Compensation Design. In 2nd IEEE Int. Conference on Information Technology and e-Services, 2012.

19. C Ghidini, C.D Francescomarino, M Rospocher, P Tonella, and L Serafini. Semantics-Based Aspect-Oriented Management of Exceptional Flows in Business Processes. Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on, 42(1):25-37, 2012.
20. Daraghmi, E.; Zhang, C.-P.; Yuan, S.-M. Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture. Appl. Sci. 2022, 12, 6242. https://doi.org/ 10.3390/app12126242