Issue 15, 2024

https://doi.org/10.23939/sisn2024.15.449

UDC 004.42

DISTRIBUTED TRANSACTIONS IN MICROSERVICE ARCHITECTURE: INFORMED DECISION-MAKING STRATEGIES

Artem Bashtovyi[1], Andrii Fechan[2]

Lviv Polytechnic National University, Software department, Lviv, Ukraine
[1] E-mail: artem.v.bashtovyi@lpnu.ua, ORCID: 0000-0003-4304-8605
[2] E-mail: andrii.v.fechan@lpnu.ua, ORCID: 0000-0001-9970-5497

The emergence of microservice architecture has revolutionized software development practices by decentralizing components, facilitating scalability, and enabling agility in system design and deployment. There are some benefits of incorporating microservices instead of a single server, however, distributed components introduce extra constraints and complexities in maintaining data consistency as well. As microservices interact independently, coordinating data updates across multiple services becomes challenging, particularly in scenarios where transactional integrity is required. Distributed transactions are one of the solutions for ensuring data consistency across services. Regardless of effectiveness distributed transactions entail different trade-offs and performance implications. Those trade-offs are not always justified. This study highlights the need for a nuanced understanding of distributed transactions in microservices by revisiting challenges in managing distributed transactions within data storage systems. It also represents existing solutions to the different distributed transaction methods. In this paper, through experiments comparing microservices and monolithic systems, the impact of distributed transactions on system performance is evaluated, giving intuition about consequences when a single data source transaction is migrated to the distributed environment. This research also contributes to enhancing understanding and decision-making regarding the utilization of distributed transactions in a microservices architecture. Ultimately, this paper presents an optimized decision-framework for the application of distributed transactions in microservices architecture, aiming to simplify and expedite processes of software architecture for software engineers, solution architects, and developers.

Key words: distributed transactions; microservices; decisions; guidance; consistency; distributed systems.

Introduction

Microservices architecture revolutionizes software development with its decentralized approach, but maintaining data consistency across distributed components poses extra challenges. Distributed transactions offer solutions by ensuring consistency of data across multiple services. However, their application introduces complexities and trade-offs, prompting a nuanced understanding of when and how to use them effectively. Similar to the consensus for multiple database nodes described above, the microservice architecture incorporates approaches and challenges to data consistency between nodes as well. While designed to operate as isolated units of work [16] with individual data storage, microservices are inherently interconnected for data exchange. Challenges arise when one microservices requires updating data on multiple consistently. There are multiple ways to perform updates of several microservices including the

2PC mentioned above, Saga pattern, Event Sourcing, CDC, and others [3, 5, 8, 12, 14, 18, 20, 23, 26]. Those methods are relevant for the respective use cases and have their trade-offs. In this article, we focus on distributed transactions that are applied for data consistency across microservices, analyzing existing solutions. This paper presents a decision framework for usage distributed transactions in microservices architecture, aiming to accelerate and simplify the decision-making process of software engineers, solution architects, and developers. RQ: How do distributed transactions influence decision-making in the context of microservices architecture, and what factors should be considered when determining their use?

Problem statement

In the context of microservices architecture, maintaining data consistency across distributed systems introduces challenges, particularly with the application of distributed transactions. While these transactions offer a solution for ensuring consistency, their usage introduces obstacles and trade-offs, thereby requiring careful decision-making. Despite existing literature exploring various methodologies and challenges, there remains a lack of comprehensive guidance on when and how to effectively utilize distributed transactions in microservices architecture. To fill the gap we have completed these tasks:

1. Conducting a literature review to explore existing challenges and solutions related to distributed transactions in microservices.
2. Investigating the influence of distributed transactions on decision-making processes within microservices architecture by performing empirical experiments, particularly in terms of latency.
3. Developing a decision framework to assist software engineers, solution architects, and developers in determining the appropriate use of distributed transactions in microservices architecture.

Literature review

Before exploring distributed transactions in microservices, let’s revisit the challenges in managing them within data storage systems, systems where transactions were applied initially. It will simplify the understanding of challenges in microservice architecture. Despite powerful hardware, standalone systems have limitations, leading databases to use multiple nodes for improved performance. However, this introduces a fundamental challenge: ensuring data is consistent among nodes. Distributed transactions at the very beginning were used for databases when a consistent update of multiple database nodes was required [28]. This caused an evolution of distributed databases. For instance, Google designed a globally distributed database that is used for replication tasks called Spanner [1]. Spanner uses a distributed storage system and a strongly consistent transactional model. Spanner also uses a combination of pessimistic locking and timestamps to ensure the serializability of transactions, which are components of its distributed transactions. Similar research [25] highlights a paradigm shift in the scalability of distributed transactions, challenging the conventional notion of their limitations. By leveraging advanced network technologies and a redesigned approach to distributed databases, the paper proposes a novel scalable database system, NAM-DB, capable of achieving scalability without the need for complex co-partitioning schemes. Modern databases utilize sophisticated replication mechanisms to address challenges in maintaining data consistency across distributed environments. For example, MySQL’s replication mechanisms aim to improve availability but are vulnerable to network disruptions. MySQL Cluster’s NDB[15] leverages distributed transactions tied to its replication technology, facilitating coordination across multiple nodes. However, interoperability with other technologies may require MySQL’s XA transaction support, based on the two-phase commit (2PC) strategy. 2PC transactions face challenges such as network failures and coordinator failures, leading to performance degradation [13]. Network failures can complicate transaction management, leaving coordinators uncertain about transaction outcomes. Moreover, recovery attempts post-failure may exacerbate issues, leading to amplifying failures and performance degradation due to prolonged participant locks on database rows. Network disruptions in distributed databases can cause delays in transaction processing, resulting in prolonged resource locking and degraded system performance. For example, if a network disruption occurs during a transaction involving multiple database nodes, resources across these

nodes may remain locked until the disruption is resolved. This can lead to decreased transaction throughput and overall system efficiency. Consistency faults in distributed databases emerge when network delays or failures obscure the success or failure of transactional operations, potentially necessitating compensatory actions such as transaction rollback. For instance, in distributed transactions involving multiple services, coordinating compensation logic becomes complex due to the challenge of tracking the state of each transactional operation. Certain actions performed within a transaction, like irreversible actions such as financial transactions, may be difficult to roll back once initiated, adding complexity and increasing the risk of inconsistencies or errors in compensating for failed transactions. Apart from the difficulty of applying rollback, a lock of data is often required to prevent concurrent read or write actions. Mainly because another process can read data that is being modified by distributed transactions. Consistent updates are applied in scenarios apart from replication as well, for instance in the research [4] authors described an application of distributed transactions to data partitioning. This author is delving into an extensive survey exploring various methodologies and factors concerning distributed transactions within distributed databases. The primary objective of exploring such as range partitioning, schema-level partitioning, and graph-level partitioning is to enhance the performance and data availability within distributed systems. However, these days usage of distributed transactions is far beyond the distributed actions within databases. The research [17] defines a comparison of distributed databases with blockchains which rely on the usage of distributed transactions. Another research [21] emphasizes the extensive coverage of distributed transactions, providing clarity on their usage within a microservices architecture. Furthermore, additional papers extend the discourse on distributed transactions, proposing optimization improvements to existing approaches. For instance, one research [27] introduces RedT, a novel distributed transaction processing protocol designed for heterogeneous networks, demonstrating superior throughput and reduced latency. Similarly, Carousel [24] is a distributed database system optimized for low-latency transaction processing across globally distributed partitions, significantly reducing transaction completion time. Additionally, another paper [2] explores the scalability of traditional concurrency control mechanisms such as Two-Phase Locking (2PL) and Two-Phase Commit (2PC) on modern hardware, achieving remarkable throughput rates. The paper “Fast General Distributed Transactions with Opacity” [19] extends the design of a fast remote memory approach to provide strict serializability and fault tolerance while maintaining high throughput and low latency within a modern data center, demonstrating the system's capability to handle millions of transactions per second. Moreover, other authors propose 2PC*, a novel concurrency control protocol for distributed transactions aimed at improving scalability and concurrency across multiple microservices, demonstrating significant improvements in throughput and latency compared to traditional 2PC [19]. Regardless of enhancements and improving 2PC commit considered the authors Daraghmi, Eman Zhang, Cheng-Pu b.; Yuan, Shyan-Ming, work [6] define clear drawbacks of 2PC for microservices due to the coordinator failure consequences and performance degradation. That is why they present an enhancement to the existing saga pattern, as an alternative distributed transaction approach in microservice architecture. They claim that the async nature and usage of local transactions in the saga pattern address issues and prove better performance on a large scale. While recognizing the importance of distributed transactions, the author P. Helland proposes solutions to avoid the usage of distributed transactions in the research [9], and urges their avoidance whenever feasible. Mainly due to the complexity and performance cost they introduce. The author M. Kleppmann in the book [11] states that “cloud services choose not to implement distributed transitions due to the operational problems they introduce”. Another author Janssen, T. describes the drawbacks of distributed transactions and urges to not use them due to potential bottleneck issues [10].

The main goal of paper

In conclusion, some research emphasizes the necessity of distributed transactions for reliability, while others advocate for their avoidance whenever feasible to promote loose coupling across domains. The uncertainty surrounding the application of distributed transactions is evident. The existing papers define extensive approaches for updating multiple services, including distributed transactions and other

options. However, to the best of our knowledge, there is no guidance to decide on the techniques when an update of multiple services is required. Multiple options are available but no defined decision framework exists that would simplify a decision over a cluster of trade-offs we explored above.

Research results and their discussion

One of the factors when choosing whether to implement distributed transactions is understanding their impact on the performance of a system. In this section, we will present the results of the experiments to compare the latency of the distributed system when distributed transactions are applied versus monolithic systems where naturally only a single database operation happens. Multiple microservices that independently process and store data in separate physical locations. The monolithic system on the other hand is a single service that performs the same business data manipulation. At the end of the experiment, the modified data on both systems is equal.

Technology used

We chose to simulate real-world scenarios accurately using state-of-the-art technologies based on our experience and expertise. Our technology stack included a machine configured with Docker-compose, 16 GB RAM, and an Intel i7(10th gen) @ 1.80 GHz × 8 processor. The monolith (S_1) was built using Spring Framework with PostgreSQL DB. Microservices architecture (S_2) was developed using Spring Framework, incorporating Kafka for messaging, PostgreSQL DB for storage on each microservice, and Zipkin for metrics monitoring. Performance testing was done using the Apache JMeter tool.

Experiment

For the sake of clarity and simplicity, the experiment was associated with a sample of the e-commerce architecture system and classical business operation of customer orders. An end user (customer) can place an order then the system verifies if an item from the order is available in inventory and eventually the customer is charged. This sequence of actions will be named an order flow. Order flow is based on 3 core components of the system: order, inventory, and payment. When an order is placed, the order module saves the order to the database and notifies the inventory to verify the capacity of items in the order. If items are available then inventory reserves items by modifying the database and notifying payment to proceed with payment, otherwise, inventory discards the operation. When payment receives a request from inventory it tries to charge a user by saving data to the database and sending a response back to the order module if payment was successful otherwise, we have to undo the order. The monolithic system performs the order flow within one database transaction thereby guaranteeing easy rollback of the order in case of failure (Fig. 1). Single database transaction denotes that service saved data to PostgreSQL DB in a transaction.

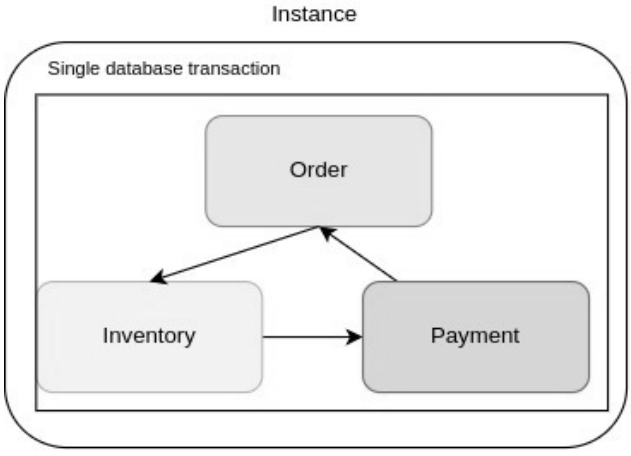

Fig. 1. Monolith architecture (S_1)

In contrast to a monolith, the distributed order flow includes 3 physically separate instances with respective modules (Fig. 2). The arrows on the microservices diagram represent async messages communication via Kafka message broker.

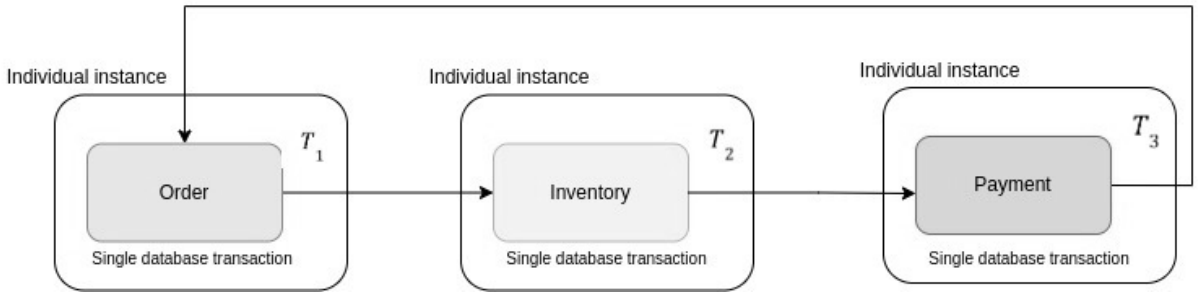

Fig. 2. Microservices architecture(S_2)

Execution of order flow affects 3 instances, each of them executes local transaction to update data, which essentially represents a distributed transaction (T_{dist}).

$$T_{dist} = T_1 \rightarrow T_2 \rightarrow \dots T_n, \quad (1)$$

where T_1 – local single transaction on the first microservice

As discussed above, there are multiple ways to implement distributed transactions in microservice architecture. Based on the analyzed literature above, we decided to pick one of the most effective ways to use Saga pattern. The inherent base of the saga pattern is asynchronous communication mainly using queues and eventually consistent updates, which provides benefits over the other approaches in terms of the goal of an experiment. In our case, we created 3 respective Kafka topics, for individual microservices. The order flow on order microservice is triggered via REST API request. The other services just listen to the messages in Kafka topics and execute local transaction when a message is received. Whenever failure happens during Saga execution, compensating transactions are executed to mitigate the effects of the preceding transaction. Meaning order and inventory services have to execute rollback for the data they updated the moment payment fails. The rollback is essentially a function that returns data to what has been done before Saga transaction. Compensation transaction is executed asynchronously via Kafka message published to the respective topic with necessary data.

To compare a monolithic system and a distributed system we decided to measure transaction latency. In a monolith, it is the latency of a single function that modifies 3 database tables in a single transaction. In microservices, to calculate transaction latency, we defined a formula for single transaction latency for order flow:

$$t_p = t_{total} - t_{exec}, \quad (2)$$

where t_{total} – total transaction time; t_{exec} – code execution time

Code execution time is the time range that the service requires to process data. It is a volatile value and highly depends on underlying computational hardware. It should not be included in the total latency, because we want to focus on the impact of the application of distributed transactions, Hence, we subtract the code execution time from the total transaction time, thereby calculating *I/O* operations and time of data transfer from one service to another. For microservice code execution time is measured on every individual service and summed. The execution of the experiment was carried out in 3 main iteration scenarios with 100, 500, and 1000 tns/s (order flow transactions per second).

The results shown (Fig. 3) clearly state that distributed transactions negatively affect the latency of the operation. Mainly due to the extra network trips between services. Even on 100 tns/s, the average latency S_1(monolith) is 224 ms, where as S_2 (microservices) is 876 ms, which is almost 4 times higher. On 500 tns/s the average latency S_1 is 255 ms and S_2 the average latency is 3243 ms. On 1000 tns/s the average latency S_1 is 411 ms and S_2 the average latency is 5117 ms. Furthermore, increasing of tns/s rate causes a non-linear latency increase in microservice architecture in comparison to monolith.

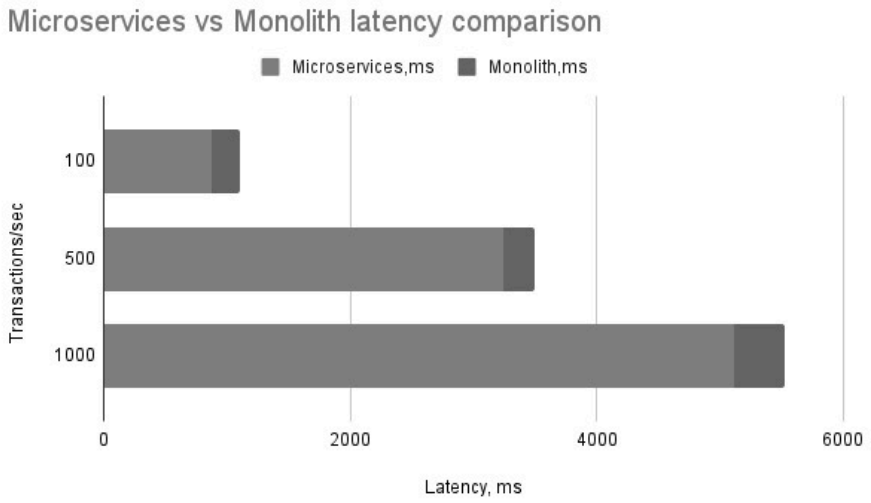

Fig. 3. Transaction latency time in monolith and microservices

In the second experiment, we evaluated the latency of S_2 when errors happen during T_{dist} execution. In our simulation, we made an error on the payment microservice, meaning that we needed to run Saga compensation for all the previous order flow participants order and inventory (Fig. 4). The definition of compensation logic:

$$\text{If } T_{n+1}\text{fails, execute } C_n \rightarrow C_{n-1} \rightarrow \ldots \rightarrow C_1, \tag{3}$$

where C_n – compensation logic for changes made by T_n local transaction

Whenever an error happens, the compensation logic (C_n) is executed on the respective service that runs T_n, which affects the overall latency of the distributed transaction (T_{dist}). In our case, compensation logic causes two more messages to be published via Kafka and two more compensation transactions executed on order and inventory services respectively.

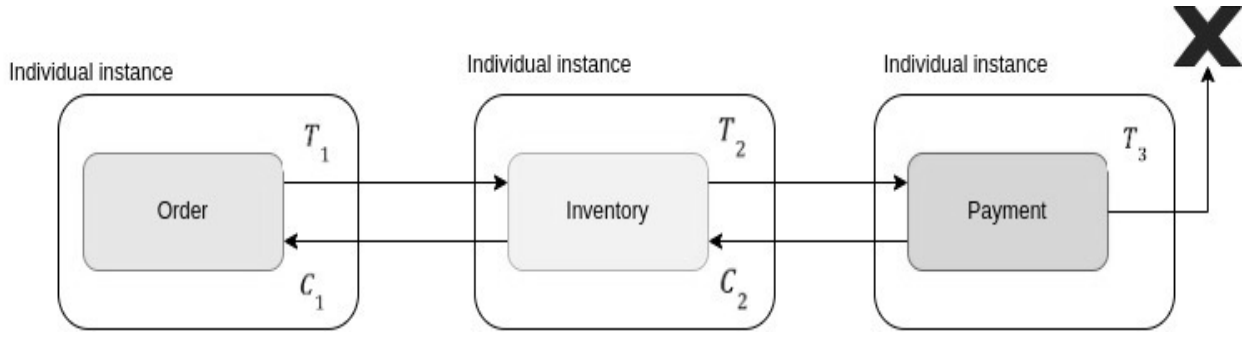

Fig. 4. Error during the transaction on the last microservice

We have conducted experiments for 3 different error rate percentages 10 %, 20 %, 50 % with the same tns/s rates as in the first experiment. The error rate simulation was implemented in a simple Round-Robin fashion, meaning we produce errors 1, 2, and 5 times per 10 transactions respectively. Results clearly define that the errors that happen during the execution affect the overall latency (Fig. 6). On average, 100 tns/s case causes a latency of 941 ms, 912 ms, and 1001 ms for 10 %, 20 %, and 50 % error rates respectively. Which is only slightly bigger than success flow latency. On 500 tns/s all of the percentage rates show more than twice 100 tns/s latency time. Over and above, 1000 tns/s cause about 40 % increased latency time for all error rates compared to 500 tns/s. This is expected since more compensation transactions are generated.

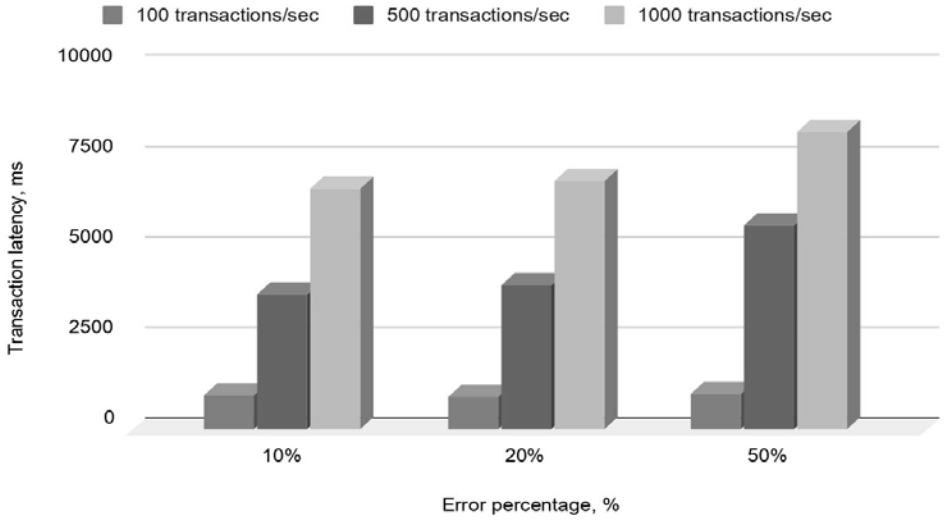

Fig. 5. Error during the transaction on the last microservice

The experiment above shows that throughput declines significantly when a regular business operation is segregated into a couple of physical services. Moreover, the experiment included extra infrastructure components that must be managed. Having said that, decisions about distributed transactions must be thoroughly made, since the moment a distributed transaction is applied new challenges arise.

As a conclusion of a literature review and experiment, we established a decision framework to simplify the distributed transaction incorporation process (Fig. 6). It's important to assess the trade-offs and implications of each decision within the specific context of the application and its requirements. The decision framework is not an ultimate guide, nevertheless, it helps guide engineering teams to quickly understand possible options.

The first step identifies the necessity for updating multiple separate services. Distributed transactions are not applicable for an update of data on a single node.

The second step involves considering a potential reorganization of the existing microservice architecture. Distributed transactions are typically necessary for straightforward and small-scale data updates. However, if multiple nodes require updating for numerous business operations, reassessing the data boundaries of the services becomes imperative. By restructuring how data is distributed and managed across multiple services, it becomes possible to eliminate the need for distributed transactions. This restructuring effort may be complex, yet it often yields significant performance and maintenance benefits for the system. Ideally, we want to move data from all of the dependent instances to a single service, so all of the necessary changes happen on a single database instance.

The third step defines the consistency of multiple node updates. If such consistency is threatened as optional or not required then we do not need to use distributed transactions. The services can simply try to update other services, if this fails then they can either retry or skip the update at all.

The fourth step block defines consistency type. Strong consistency in microservices, when distributed transactions are applied, ensures that all participating services reach a consensus on the outcome of the transaction. This means that once a distributed transaction is committed, all data updates performed by the transaction are immediately visible to all services involved, providing a unified and coherent view of the system's data across all nodes. Strong consistency ensures that the effects of the distributed transaction are applied uniformly and reliably across the entire system, maintaining data integrity and accuracy. In general strong consistency of distributed nodes is extremely costly since this often requires blocking application resources. Eventual consistency in microservices implies that while updates may not be immediately propagated to all services, they will eventually converge to a consistent state over time. In this model, there is a temporary period where inconsistencies may exist across different services due to the asynchronous

nature of distributed transactions. The saga pattern, mentioned above, is a great candidate for a scenario of eventually consistent transactions [7].

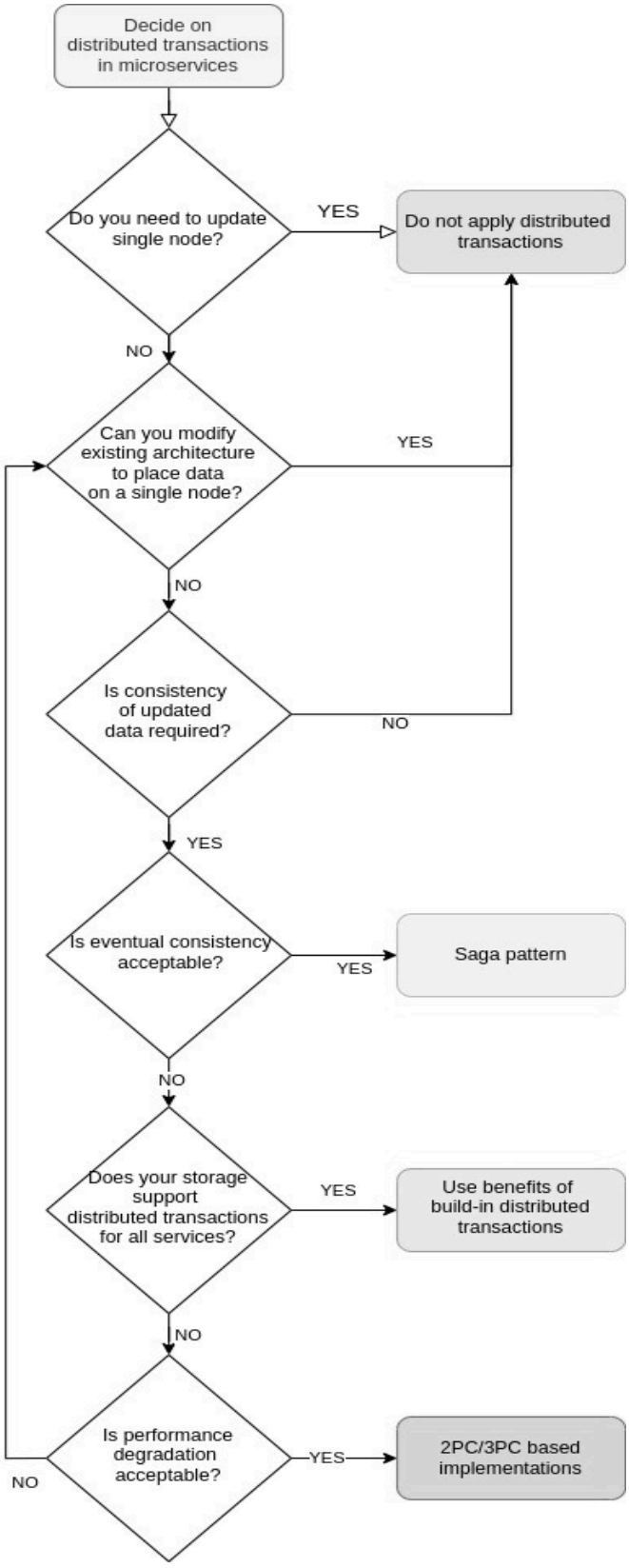

Fig. 6. Decision flow-chart for usage of distributed transactions

The fifth step defines whether the data sources used by all microservices that participate in updates support distributed transactions. Some storages support distributed transactions between their nodes. For instance, Mongo DB [22] supports such transactions over different shards. Build-in functionality still uses classic distributed transaction implementations, however, it is optimized to the storage technology and managed by a database storage engine, which allows to use of ready-to-go functionality without custom adoption and selection of distributed transaction approaches.

Lastly, in the literature review section, we defined that transactions that required consistency have performance trade-offs. If performance degradation is deemed acceptable within the system's requirements, then implementing 2PC may be a viable option to achieve strong consistency. However, if maintaining performance levels is crucial, then a review of the architecture may be necessary to explore alternative approaches that balance consistency requirements with performance considerations.

Conclusions

Existing research offers insights into specific scenarios of distributed transactions usage however, it leaves unanswered questions about the optimal approach and when to apply the transactions in microservices architecture. The uncertainty about when to use them highlights the importance of careful consideration and analysis. Our experiment demonstrated that microservice system on average performs 5 times slower in comparison to the monolithic architecture when distributed transactions are applied, which in fact demonstrates performance drawback. Additionally, the second experiment proved the relation between the amount of errors and latency in distributed transactions when applied to microservices. We established that the higher the amount of transactions per second that fail the higher the latency of such transactions. More specifically, with 20 % of failed transactions 100 transactions per second perform more than 8 times faster on average than 1000 transactions per second. This evidence adds clarity to our understanding of the trade-offs when designing distributed transactions in microservices, providing valuable insights for decision-makers. Having said that, distributed transactions cause performance degradation on a large scale when applied to microservice architecture, which is why they must be used carefully and ideally they must be avoided. Ultimately, our research has allowed to create the algorithm for for productive decision-making process regarding the utilization of distributed transactions in microservices architecture. The decision framework aims to equip software engineers and developers with the knowledge needed for distributed solutions development.

In future work, we will focus on fine-grained analysis of strong and eventual consistency in microservices architecture. Specifically, we will compare existing solutions and their effect on the performance of microservice architecture. This will help software engineers and developers to make decisions about specific technologies when the adoption of distributed transactions is inevitable.

References

1. Bacon, D., Kogan, E., Lloyd, A., Melnik, S., Rao, R., Shue, D., Taylor, C., Holst, M., Woodford, D., Bales, N., Bruno, N., Cooper, B., Dickinson, A., Fikes, A., Fraser, C., Gubarev, A., Joshi, M. (2017). Spanner: Becoming a SQL system. Proceedings of the 2017 ACM International Conference on Management of Data, 331–343. https://doi.org/10.1145/3035918.3056103

2. Barthels, C., Müller, I., Taranov, K., Alonso, G., & Hoefler, T. (2019). Strong consistency is not hard to get: Two-phase locking and two-phase commit on thousands of cores. Proceedings of the VLDB Endowment, 12(13), 2325–2338. https://doi.org/10.14778/3358701.3358702

3. Bashtovyi, A., & Fechan, A. (2023). Change data capture for migration to event-driven microservices case study. https://doi.org/10.1109/CSIT61576.2023.10324262

4. Bharati, R. D., & Attar, V. Z. (2018). A comprehensive survey on distributed transactions based data partitioning. 2018 International Conference on Current Trends towards Converging Technologies (ICCT), 1–5. https://doi.org/10.1109/ICCUBEA.2018.8697589

5. Binildas, C. (2019). Transactions and microservices. In Practical microservices architectural patterns, 483–541. Apress. https://doi.org/10.1007/978-1-4842-4501-9_14

6. Daraghmi, E., Yuan, S. M., & Zhang, C. P. (2022). Enhancing saga pattern for distributed transactions within a microservices architecture. *Applied Sciences*, 12(12), 6242. https://doi.org/10.3390/app12126242

7. Dürr, K., Lichtenthaeler, R., & Wirtz, G. (2022). Saga pattern technologies: A criteria-based evaluation. In Proceedings of the 17th International Conference on Software Technologies, 141–148. SciTePress. https://doi.org/10.5220/0010999400003200

8. Fan, P., Liu, J., Yin, W., & Wang, H. (2020). 2PC*: A distributed transaction concurrency control protocol of multi-microservice based on cloud computing platform. *Journal of Cloud Computing*, 9(1), Article 11. https://doi.org/10.1186/s13677-020-00183-w

9. Helland P.(2017). Life beyond distributed transactions. Commun. *ACM*, 60, 2, 46–54. https://doi.org/10.1145/3009826

10. Janssen, T. (n.d.). Distributed transactions – Don't use them for microservices. Thorben Janssen. Retrieved April 1, 2024, from https://thorben-janssen.com/distributed-transactions-microservices/

11. Kleppmann, M. (2017). Designing Data-Intensive Applications (1st ed.). O'Reilly Media, Inc.

12. Limón, X., Guerra-Hernández, A., Sanchez G., A., & Pérez-Arriaga, J. C. (2018). SagaMAS: A software framework for distributed transactions in the microservice architecture. In Proceedings of the 6th International Conference in Software Engineering Research and Innovation, 50–58. IEEE. https://doi.org/10.1109/CONISOFT.2018.8645853

13. Lin, Q., Chang, P., Chen, G., Ooi, B. C., Tan, K.-L., & Wang, Z. (2016). Towards a non-2PC transaction management in distributed database systems. In Proceedings of the 2016 International Conference on Management of Data, 1659–1674. Association for Computing Machinery. https://doi.org/10.1145/2882903.2882923

14. Munonye, K. & Martinek, P. (2020). Enhancing Performance of Distributed Transactions in Microservices via Buffered Serialization. *Journal of Web Engineering*. https://doi.org/10.13052/jwe1540-9589.1956.

15. MySQL NDB Cluster CGE. (n.d.) MySQL. Retrieved April 1, 2024, from https://www.mysql.com/products/cluster/

16. Richardson, C. (n.d.). Pattern: Database per service. Microservice Architecture Retrieved April 1, 2024, from https://microservices.io/patterns/data/database-per-service.html

17. Ruan, P., Dinh, T. T. A., Loghin, D., Zhang, M., Chen, G., Lin, Q., & Ooi, B. C. (2021). Blockchains vs. distributed databases: Dichotomy and fusion. In Proceedings of the 2021 International Conference on Management of Data, 1504–1517. Association for Computing Machinery. https://doi.org/10.1145/3448016.3452789

18. Sekhar, R. R. (2020). Microservices, Saga Pattern and Event Sourcing: A Survey. *International Research Journal of Engineering and Technology*, 7(5), 633–636. https://www.irjet.net/archives/V7/i5/IRJET-V7I5124.pdf

19. Shamis, A., Renzelmann, M., Novakovic, S., Chatzopoulos, G., Dragojević, A., Narayanan, D., & Castro, M. (2019). Fast general distributed transactions with opacity. In Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19), 433–448. Association for Computing Machinery. https://doi.org/10.1145/3299869.3300069

20. Shi, L., Wang, K., Li, M., Tong, J., Qiao, H., & Jiang, Y. (2023). Research on distributed relational database based on MySQL. Other Conferences.

21. Štefanko, M., Chaloupka, O., & Rossi, B. (2019). The saga pattern in a reactive microservices environment. In Proceedings of the 14th International Conference on Software Technologies, 483–490. SciTePress. https://doi.org/10.5220/0007918704830490

22. Transactions. (n. d.). MongoDB. Retrieved April 1, 2024, from https://www.mongodb.com/docs/manual/core/transactions/

23. Wu, Y., & Liang, Z. (2018). Implementation of distributed XA transactions in MyCat based on table broadcasting mechanism. In 2018 IEEE/ACM 5th International Conference on Big Data Computing Applications and Technologies (BDCAT), 215–216. IEEE. https://doi.org/10.1109/BDCAT.2018.00038

24. Yan, X., Yang, L., Zhang, H., Lin, X., Wong, B., Salem, K., & Brecht, T. (2018). Carousel: Low-latency transaction processing for globally-distributed data. *In Proceedings of the 2018 International Conference on Management of Data*, 231–243. Association for Computing Machinery. https://doi.org/10.1145/3183713.3196912

25. Zamanian, E., Kraska, T., Binnig, C., & Harris, T. (2016). The End of a Myth: Distributed Transactions Can Scale. *Proceedings of the VLDB Endowment*, 10(6), 483–541. https://doi.org/10.14778/3055330.3055335

26. Zhang, G., Ren, K., Ahn, J. S., & Romdhane, S. B. (2019). GRIT: Consistent Distributed Transactions Across Polyglot Microservices with Multiple Databases. https://doi.org/10.1109/ICDE.2019.00230

27. Zhang, Q., Li, J., Zhao, H., Xu, Q., Lu, W., Xiao, J., Han, F., Yang, C., & Du, X. (2023). Efficient distributed transaction processing in heterogeneous networks. *Proceedings of the VLDB Endowment*, 16(5), 1372–1385. https://doi.org/10.14778/3583140.3583153

28. Zhu, T., Guo, J., Zhou, H., Zhou, X., & Zhou, A.-Y. (2018). Consistency and availability in distributed database systems. *Ruan Jian Xue Bao/Journal of Software*, 29(1), 131–149. https://doi.org/10.13328/j.cnki.jos.005433

РОЗПОДІЛЕНІ ТРАНЗАКЦІЇ В МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ: СТРАТЕГІЇ ПРИЙНЯТТЯ ОБҐРУНТОВАНИХ РІШЕНЬ

Артем Баштовий[1], Андрій Фечан[2]

Національний університет “Львівська політехніка”,
кафедра програмного забезпечення, Львів, Україна
[1] E-mail: artem.v.bashtovyi@lpnu.ua, ORCID: 0000-0003-4304-8605
[2] E-mail: andrii.v.fechan@lpnu.ua, ORCID: 0000-0001-9970-5497

Виникнення мікросервісної архітектури істотно модернізувало практики розроблення програмного забезпечення завдяки децентралізації компонентів, що полегшило масштабованість та сприяло гнучкості у проєктуванні та впровадженні систем. Використання мікросервісів замість одного сервера має певні переваги, проте розподілені компоненти також спричиняють додаткові обмеження та складнощі у підтримці узгодженості даних. Оскільки міскросервіси взаємодіють незалежно один від одного, координація оновлень даних через кілька сервісів ускладнюється, особливо в сценаріях, де потрібна транзакційна цілісність даних. Розподілені транзакції – одне із рішень для забезпечення узгодженості даних між сервісами. Незважаючи на ефективність, розподілені транзакції передбачають різні компроміси та вплив на загальну продуктивність системи. Це дослідження підкреслює потребу у виваженому розумінні розподілених транзакцій у мікросервісах, повертаючись до викликів у керуванні розподіленими транзакціями в системах зберігання даних. Досліджено також відомі рішення для різних методів розподілених транзакцій. В цій роботі оцінено вплив розподілених транзакцій на продуктивність, зроблено висновки про наслідки перенесення транзації з однієї бази даних у розподілене середовище на підставі експериментів, у яких порівнювали мікросервісні та монолітні системи. Це дослідження також сприяє покращенню розуміння та прийняттю рішень щодо використання розподілених транзакцій у міскросервісній архітектурі. В підсумку запропоновано оптимізований метод прийняття рішень для застосування розподілених транзакцій у мікросервісній архітектурі, спрямований на спрощення та прискорення процесів проєктування програмного забезпечення для програмістів, архітекторів рішень та розробників.

Ключові слова: розподілені транзакції; мікросервісна архітектура; рішення; посібник; узгодженість; розподілені системи.