Distributed Transaction Management in Microservices Architecture

MUHAMMAD SOHAIL
Shaheed Zulfikar Ali Bhutto Institute of Science & Technology

Abstract- On a general level, it is critical but rather complex to handle distributed transactions in microservices architectures nowadays is one of the core concerns in modern software engineering. This paper looks at the concepts, techniques, and paradigms that underpin the modern best practices in achieving transactional behavior in distributed systems, and also how they apply to the complexities of microservices architecture. It gives a brief comparison of the ACID vs BASE models of transaction processing and examines how those paradigms affect system design. Common used protocols raised here include Two-Phase Commit (2PC) and the Saga Pattern; their benefits and drawbacks are introduced besides with how to apply them. Continuing the evaluation of the CAP theorem pointed out in the article, the author examines the importance of consistency, availability, and partition tolerance for distributed architectures. Filled with real-world case studies, practical tips, and approximations, it provides developers and system architects with the proven approaches to designing maintainable, scalable, and efficient systems that do not corrupt data in large distributed microservices architectures.

Indexed Terms- Microservices Architecture, Distributed Transactions, Saga Pattern, Two-Phase Commit, Eventual Consistency, CAP Theorem, Database Partitioning, Scalability, Resilience, Data Integrity, Service-Oriented Architecture, Distributed System Challenges, Fault Tolerance, Event-Driven Architecture, Transaction Coordination, Choreography vs. Orchestration, Consistency Models, Message Brokers, System.

I. INTRODUCTION

There is no denying that a microservice as a structure for the construction of modern software systems has become the leading approach for the creation of applications with predictable scalability, high availability, and increased adaptability. Due to the decomposition of large systems into simpler components as well as maintaining independently deployable components, microservices have several benefits over more centralised monolithic models. The perceived benefits include more easily achieved scalability, better isolation of failures and the possibility to comprise different parts of the system with different technologies. Nevertheless, the transition from a monolithic architecture to a microservices system brings new issues into the picture – most notable, how to handle transactions which cross several microservices?

By definition, in a monolithic system, transactions occur in a single database and hence management of data consistency, reliability and atomicity based on the ACID principles is not complex. On the other hand, microservices deploy functionality to several isolated services, and every service possesses its database, APIs, and business logic. This architectural decentralization creates a fundamental challenge: how to provide the common interface across multiple services while tackling how to manage simultaneous transaction processing, quality consistency, and fault-tolerant mechanisms at the same time.

Let us consider an e-commerce website they are companies selling products online. During an order, the inventory management service must check whether the product is in stock, the payment gateway must complete the payment securely, and the shipping service must confirm the product's availability for shipping. These operations are connected seamlessly, while taking place in different services. In case any of the actions is unsuccessful, for instance, a payment was declined, or there was no inventory to meet the demand – reversing or making adjustments to the given move is problematic.

These concerns are managed by Distributed transaction management through the use of suitable methods and patterns of work flow control that are applied on services. Standard ACID properties are antic to some extent applicable in centralised environment however they face problems in distributed solution for their impact on availability and system throughput. Instead, flexibility models and new application patterns like the eventually consistent systems, the Two-Phase Commit (2PC) and the Saga Pattern.

Fundamentals of Distributed Transactions

Transaction management for distributed transactions is usually important especially in the present day microservices architecture where operations may cut across different nodes or services. These transactions must be managed co-ordinated to meet the principles of atomicity, consistency, isolation and durability (ACID) that are fundamental traditionally used to maintain data integrity in monolithic systems. However, with microservices comes the challenges of distribution, and so, perfect adherence to the ACID form of transaction management is not feasible. To overcome such limitations distributed systems principles like BASE: Basically Available, Soft state, Eventually consistency (Brewer, 2000) are. This change in fundamental thinking makes it easier to achieve scalability and maintain the functionality during network partitions or failures as opposed to adhering to database traditional strong consistency.

Main Issues in Distributed Transactions

Network Partitioning:

Any programs executed in a distributed fashion can, unfortunately, fail due to the nature of distributed computing – networks fail and nodes or services may become disconnected permanently or temporarily. Such partitions lead to disconnection, and affects transaction co-ordination and synchronization across nodes. For example, a node that drops out of a distributed transaction may cause data inorganism if it is not recovered (Kumar et al., 2018).

Latency:

Interaction between the services that exist at different geographical locations causes delays within the system, and this affects its general performance. The higher response times slow down the transaction rates and thus affect user experience particularly in cases where near real-time update is required (Patel & Sharma, 2017). Reducing the response time while keeping it efficient has always been an issue.

Fault Tolerance:

It is a challenging job to guarantee data consistency when the node or service in consideration has failed. Retries, compensating transactions, and distributed consensus algorithms are necessary procedures but they are costly and vulnerably prone to errors. For example, a problem with one of the services interdependent on others in a complex transaction means that subsequent services have to ‘roll back’, or run again, to return to a correct state (Anderson et al., 2016).

The following table summarizes these challenges and the corresponding strategies to address them:

Challenge	Description	Mitigation Strategies
Network Partitioning	Loss of connectivity between nodes disrupts transaction consistency.	Adoption of BASE principles, use of quorum-based protocols, and fallback mechanisms to handle partitions.
Latency	Delays in communication across services impact transaction performance.	Optimizing network communication, asynchronous processing, and local caching for critical operations.
Fault Tolerance	Handling service or node failures without losing data consistency.	Implementing compensating transactions, retries, and consensus mechanisms like Paxos or Raft.

BASE in Distributed Transactions

BASE principles are an asynchronous equivalent of the issues with ACID that are not suitable for distributed systems since BASE uses eventual consistency. Like Paxos, BASE systems provide a

guarantee that, after all updates the system is consistent, instead of the guarantee that they are always consistent. This approach enables distributed systems to acquire higher availability as well as fault tolerance, though at the same time ensuring scalability. For instance, BASE principles are used in e-commerce situations where high transaction rates are processed while users have great interaction experience (Stonebraker, 2010).

The CAP Theorem and Its Implications

The CAP theorem, proposed by Eric Brewer in 2000, is a fundamental concept in distributed systems that asserts that it is impossible for a distributed system to simultaneously guarantee all three of the following properties:

Consistency (C): Read operations from the system will always get the most up to date writing from the system. This means that all nodes in for a system have access to the data at the same time.

Availability (A): At the end of the transaction in read or write form, every call will be acknowledged either a success or failure without the consensus of data nodes.

Partition Tolerance (P): Since the system must run properly even when the network is split or nodes are failure, the system will be able to handle requests during such situations.

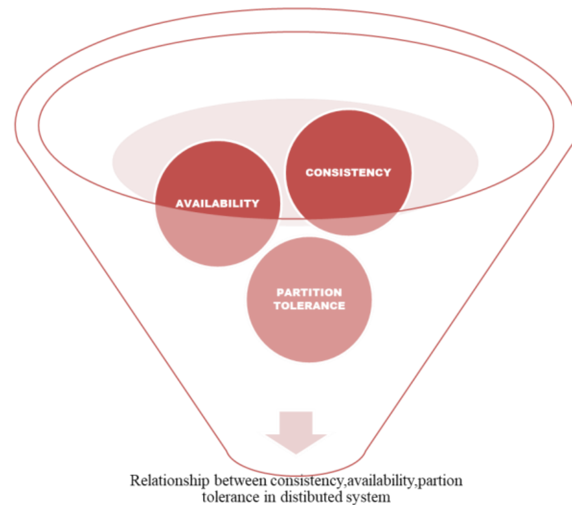
By the CAP theorem, distributed systems can only provide at most two of these three characteristics concurrently. This basically implies that for any specific distributed system at your disposal, you need to make compromises based on the needs you have at that particular instance. Characteristic scenarios occur in further development of microservices architectures where a transaction involves several services and nodes.

CAP in context of microservice

Partition tolerance in microservices is always a constraint because the services are distributed across the node, or data center and the network is not perfect; it fails. Therefore, the microservices architectures generally choose availability and partition tolerance high while consistency is usually weak. In practice, this means that systems may tolerate at some level of inconsistency—updates to the state effected by one service may propagate through the other services and become consistent over time, but not necessarily when the operation is invoked (Pat Helland, 2007).

In some such architectures, there will be use of message brokers or other intermediary systems to handle failures and sequence issues with getting events processed as expected. Such message brokers are used to keep messages that cannot be processed at the moment due to the system partitioning so that the system can come back to life with as high of an availability as possible. This pattern known as the "eventual consistency" is characteristic of the systems where availability and tolerance to the fault are the key values, but strict consistency is not as significant (Pat Helland, 2007).

It is possible to illustrate the consistency, availability, and partition tolerance triangle in this diagram below: The three properties are the vertices of triangle and the system has be at one of the points or on the boundary of the line between two vertices, which means trade-off.



Relationship between consistency,availability,partion tolerance in distibuted system

CP (Consistency and Partition Tolerance): Configurations that can maintain consistent 'projections' in the face of network partition ability but which might be less available during the recovery process.

CA (Consistency and Availability): Stable and highly available consistency and availability systems that could not suit the network partitions.

AP (Availability and Partition Tolerance): Those systems where upper priority has been given to availability over partition tolerance and that so use techniques of soft-state or eventuality along with consistency.

Real World Application and Take Up
In big web applications like e-shopping sites, online video streaming or social service sites, availability and partition tolerance are much more important to retain the site's normal functionality in case of network split or congestion. For instance, Netflix's Cassandra and Ama-azon's DynamoDB make temporary quorum hints—that do not guarantee im-mediate consistency across ser-vices—by allowing services to operate based on eventually-consistent data (Brewer, 2000; Stonebraker, 2010).

Nevertheless, were eventual consistency adopted, this by no means implies that consistency is completely overlooked. Instead, it means that the system can tolerate such inconsistency, for it expects the data to harmonize in the future. This trade-off is good especially when the critical sections can endure a few synchronization points slightly off in terms of time or systems that are expected to work with eventual consistency and methods such as idempotence or conflict resolver (last writer wins) (Kumar et al., 2018).

Approaches to Distributed Transaction Management
Two-Phase Commit (2PC)
The Two-Phase Commit (2PC) protocol ensures atomicity in distributed transactions with two phases:
Prepare Phase: In order to gain confirmation from all services the coordinator proceeds to make the request. If any of the services receive a one or a zero, the whole transaction is nullified.

Commit Phase: All the services receive and if all agree then coordinator sends a "commit" message to complete the transaction. The moment that any service is failed it turns back.

The major downside with 2PC concerning scalability and fault tolerance, is that it guarantees consistency at the price of these disadvantages.

Drawback	Impact
High Latency	Delays due to multiple communication round-trips.
Blocking Resources	Services lock resources, causing potential delays.
Single Point of Failure	Coordinator failure jeopardizes the entire transaction.
Lack of Fault Tolerance	Service failures require aborting the transaction.

II. APPLICABILITY IN MICROSERVICE

However, the latency, blocking and fault tolerance limitations make 2PC ineffective in scalable, highly available microservices architecture. Other approaches such as the Saga Pattern or the models like eventual consistency are more appropriate in large systems (Gray, 1978). Nevertheless, 2PC could still be useful inside small, simple environments without extremely strict requirements to consistency.

Saga Pattern
Saga Pattern can be used to facilitate a solution to distributed transactions through the splitting of each of them into a number of sub-transactions that can be compensated using another sub-transaction in the case of its failure. This pattern is here aimed help the further maintenance of eventual consistency and system robustness as it is guaranteed that the consistency of the totality of an overall system remains guaranteed in case sub-transactions fail.

Sagas can be implemented through two main approaches:
Orchestration: In this model there is a co-ordinator responsible for monitoring the flow of transactions, the performance of the sub transactions and initiation of the compensation transaction in the event of a sub transaction failure. The coordinator also makes sure

that the transaction includes steps for the state, as well as maintaining coherent service states.

Choreography: However, the choreography model relates to each service independently controlling its sub-transaction. Events occur between services and there is no gathering place where the interactions are managed from. However, each service waits for an event and performs the sub-transaction for the service and compels for compensations if necessary.

Although the Saga Pattern gives a system the ability to protect against failure and achieve eventual consistency, it is best used when carefully designed, for example, when addressing issues of cascading failures, the problem of compensating transactions, and idempotence, a condition where executing the same operation twice results in no change (Garcia-Molina & Salem, 1987).

Key Steps:
Service A (Start Action):
The first service triggers an exchange by conducting its function. If it succeeds, it commits. If it does not work out as intended, another corrective action takes place to revert the changes of this service.

Service B (Intermediate Action):
So if Service A commits successfully Service B does its function. In this regard upon success it commits and passes forward the success. In this case it has a compensating mechanism that initiates an action to reverse the operation of the failure.

Service C (Intermediate Action):
Service C works like Service B it executes its role in the transaction. If it works then it stakes, if not, then it reverses its operations if there was a failure.

Service D (End Action):
Service D is the finishing service in the transaction. It either records the final outcome or, if there was a problem, initiates a compensating action that cancels its changes while also reversing all subsequent ones.

The moment any service strikes a failure point, a failure event is initiated and the corresponding compensating actions are performed in a backward cascade beginning from Service D and ending at Service A.

Saga Pattern Characteristics:
Orchestration vs. Choreography:
Orchestration: One service delivers all the methods of a transaction (e.g., Service A contains all the methods for performing the transaction).
Choreography: Only the instruction of leinProcess Developer and leinDrives will vary, while each service bases its actions on event and is aware when to start or undo its part.

This diagram demonstrates how the Saga Pattern can handle multiple, distributed transactions that wouldn't behave predictably; it makes sure that everything is consistent eventually, even if there are failures across several services.

Event-Driven Transactions
In event-driven architectures, services communicate asynchronously using message brokers like RabbitMQ or Kafka. Events trigger downstream processes, decoupling services and enhancing fault tolerance. However, this approach necessitates robust mechanisms for ensuring message delivery and handling duplicates (Kleppmann, 2017).

Tools and Technologies used in Supporting Transaction Management
The management of transactions in current large-scale systems involves numerous solutions that can cope with emerging challenges based on consistency, fault tolerance, and scalability in microservices environments.

1. Distributed Databases
Distributed database is basic in addressing data distribution aspect with multiple services in microservices architectures with focus on data coherence and access.

NoSQL Databases: Cassandra and MongoDB are examples of databases with eventual consistency, that means they are highly available and horizontally scalable. These databases employ the BASE model as

consistency is traded for availability and partition tolerance These databases are suitable for distributed systems that require scalability Coulouris et al., 2012, Pritchett 2008).

Cassandra: Originally implemented with no single point of failure and thus being distributed, Cassandra has configurable consistency levels offering a compromise between strong and eventuality consistency (Lakshman & Malik, 2010).

MongoDB: MongoDB supports multi-document transactions, which means atomicity in collections, something important for the microservices that cross services and databases (Chodorow, 2013).

Comparison of Distributed Databases

Feature	Cassandra	MongoDB
Consistency Model	Eventual Consistency	Strong consistency with tunable levels
Scaling	Horizontal, distributed scaling	Horizontal scaling
Transaction Support	Limited (ACID support in 2.0)	Multi-document ACID transactions

2. Message Brokers
Asynchronous communication refers to services that need to be included in distributed transactions in event-driven system and this is handled through message brokers.

Kafka: Apache Kafka is a distributed streaming platform that guarantees the ability of high throughput and fault tolerance. It also defines the modeled event-driven architecture by enabling services to publish and consume messages in an asynchronous fashion because it is important to implement the eventual consistency and Sagas (Kreps et al., 2011).

RabbitMQ: Another type of messaging is request-response, which is supported by RabbitMQ; besides, guarantee is given to delivery a message no matter the failures. It is useful for compensating transactions in such patterns as the Saga Pattern, as every service can respond to the situation and carry out the needed transactions (Sacks, 2013).

3. Orchestration Tools
Transaction management and application of the proper orchestration workflow guarantee reliable and consistent services with the possibility to recover from several failures.

Kubernetes: Kubernetes announced and schedule containerized microservices deploying, scaling and failure handling. IT has a particularly important function in achieving high availability and handling transactions in distributed systems (Hightower et al., 2017).

AWS Step Functions: AWS Step Functions coordinate AWS services for running complex tasks and works well for distributed transactions, which can have failures like retry and rollback. What this does is to enhance the organization of transactions across cloud-based designs through the visualization of a work flow.

4. TM Framework
Bears Software's Atomikos and Narayana are some of the frameworks used in managing distributed transactions dealing with different systems at varying degrees of compliance with the ACID and Saga transactions.

Atomikos: In the context of distributed transactions, Atomikos provides implementation of the Two-phase commit protocol (2PC) and the Saga Pattern in order to keep the consistency of operations across services and databases (Baker et al., 2006).

Narayana: Narayana is an open source transactional middleware based on Java Technology, dealing both with normal ACID Transaction as well as with newer ones like Saga; thus, combining a broad-scope applicability with robustness and flexibility, Narayana is an ideal solution for complex, large-scale Java Transaction applications (The Narayana Team, 2019).

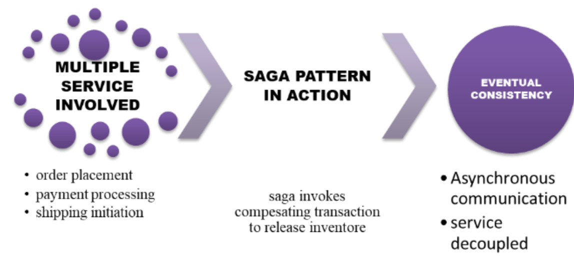
Case Studies and Real-World Applications
Many enterprises across different sectors have adopted distributed transaction management in the microservices architecture Of these, e-commerce

solutions and banking systems are some of the most common. These case studies illustrate how Saga Pattern, Two-Phase Commit (2PC), and other distributed transactions protocols are applied to maintain data integrity, availability, and recoverability in distributed systems.

1. E-Commerce Platforms
One of the primary examples of distributed transaction management is associated with e-commerce platforms, which is inherently a process of integrating various services within one system in order to accomplish a single customer order. Several critical options, including order putting, payment authorization, stock reservation, and shipment origination, imply synchronized functioning of the distributed services that are located in different databases as a rule.

Saga Pattern in Action: A business-to-consumer order-processing in most e-commerce platforms consists of several service operations. When the consumer makes an order, then the inventory service blocks stock for it, payment service handles the transaction, and the shipment service gets ready to dispatch order. The Saga Pattern guarantees that if any step in this process fails for example, payment failure then compensating transactions are called. For example, when payment cannot be made after having reserved the inventory, the Saga releases the latter through a compensating action.

Eventual Consistency: Typically, e-commerce platforms face the issue of work-stress between keeping everything to some extent homogenized while maintaining high quality. In particular, eventual consistency makes it possible to improve throughput and provide service availability at the e-commerce systems level by using asynchronous communication, for example, message brokers (RabbitMQ Kafka, etc. type).



2. Financial Systems
Financial systems are usually very consistent and their transaction integration is very rigid because handling of financial information is very sensitive. After a while and as with any software, distributed ledgers and financial applications require intricate transactions to be managed between the services and various applications in terms of balances, records, and regulation.

Hybrid Approach (2PC and Eventual Consistency): In the financial systems, therefore, balanced hybrid approaches are presented to meet the demands of consistency in relation to performance and fault tolerance. While basic operations are local transactions, certain key operations such as transfer of funds from one account to another are strongly consistent and are usually handled using protocols such as the Two-Phase Commit (2PC) to ensure that either the whole transaction takes place or none at all in an attempt to avoid situations where some of the services in the cluster respond to the transaction while others do not hence create huge cash deficits.

Eventual Consistency for Non-Critical Operations: For class III tasks, like updating account balances and processing interest, that are non-time-sensitive financial systems might decide to process transactions at some time in the future and might use the Model where different services are asynchronous. This minimizes the delays involved and results in the system being able to handle constant throughput for the data to be in a congregate state. In cases where, as with fraud detection or reporting, eventuality of Consistency is more valuable than immediate consistency, Eventual consistency is the optimal choice (Chockler et al., 2006).

Distributed Ledgers and Blockchain: The management of distributed transactions worldwide especially in the financial sector can be explained by the usage of Blockchain and Distributed Ledger Technology (DLT). Blockchain systems offer some characteristics which are naturally inherent to their architecture such as data immutability and strong consistency because every transaction in the system is connected to a previous transaction through encryption. This makes it very ideal for use in keeping records of every transaction that occurs between different participants without the need for a controlling center. Banks, payment processors and other financial institutions are starting to look at blockchain to help increase the level of security, efficiency and clarity of transactions (Narayanan et al., 2016).

3. Healthcare Systems

Microservices and distributed architectures are becoming more common in healthcare systems where the data belonging to patients are being managed, appointments are being scheduled, billing is being done and insurance claims are being processed. In such situations, transactions extend through various systems like hospital information system, insurer systems, and electronic health record systems.

Saga Pattern for Patient Care Coordination: Health care application can be an appointment making, a test result processing, or a claim sending application. The Saga Pattern can assist in the management of such services to guarantee that if one service fails, the other services aimed at compensating that failure are activated; for instance, if the insurance claim submission is unsuccessful, then the appointment is canceled, or the scheduling needs to be readjusted. This pattern is more beneficial when planning service interactions where many parties take part, including healthcare providers, insurance firms, and customers (Sanders et al., 2017).

Data Privacy and Compliance: The nature of health data means that distributed transaction management in healthcare must also conform to strict compliance regulation therefore; US hipaa or EU GDPR among others. Frameworks such as FHIR (Fast Healthcare Interoperability Resources) and HL7 standards offer structures and guidelines for Transactional Privacy ad Security since transactions that occur over different systems are brokered (Dixon et al., 2019).

Challenges and Limitations

Microservices architectures when combined with distributed transaction management results in several inherent problems that need to be solved to make the solution reliable, fast, and scalable.

1. Performance Overheads

Coordination protocols like Two-Phase Commit (2PC) bring a lot of latency from multiple round trips over services. This overhead poses a problem to overall system performance, particularly in environment with heavy traffic such as e-commerce application, or financial transactions where system latency is a major concern (Newman, 2015). As mentioned, delays happen when multiple services are included becoming the key problem and restraining factor affecting efficiency as it grows.

2. Fault Handling Complexity

Managing of failures during distributed transactions is complicated. In systems such as 2PC, the absence of a means of continuing once an issue occurs during the execution of a transaction stagnates the process and requires complex rollback techniques. When a failure happens, compensating transactions such as the ones seen in the Saga Pattern reverse the sequence of actions, but their design must be well thought out to prevent new failures (Gray, 1978). Maintaining consistency in such environment, especially if failures are across services and/or databases, is a huge challenge (Garcia-Molina & Salem, 1987) Controlling for update anomalies is very difficult in such a system since failure can occur in multiple services or databases simultaneously.

3. Debugging and Monitoring

Isolation of problems in distributed systems is complex because a transaction involves multiple services. Some tools, such as distributed tracing (OpenTelemetry), distributed logs (like ELK stack), are Indispensable but hard to set up and maintain (Newman, 2015). These aids allow tracking the flow of transactions across the services, however using these tools implies additional planning needed in designing interfaces, as well as may add additional load in highly loaded applications.

4. Scalability: The main concern for the current RPC technique is that most resources are.
contention: This may be true but the major drawback for the current RPC technique is resource contention. But as the distributed systems expand the problem of achieving scalability without invoking resources is cumbersome. As in the case of 2PC, it causes resource locking that is detrimental to scalability (Pritchett, 2008). The Saga Pattern is free from this issue since it incorporates an asynchronous form of communication so that services may execute concurrently. However, it mandates concurrent consistency which may become a challenge to manage in system design (Gupta et al., 2015).

5. Consistency vs. Availability Trade-offs: It is a fundamental choice in caching strategies between being consistent or available, but makes the most sense when read in their full title as Cache Inconsistency vs Cache Invalidity.
Distributed systems are known for the CAP theorem (Brewer, 2000) stating that distributed systems could either be consistent or available, but can almost never be both. When it comes to the availability during the network partitions, the systems that prioritized the consistency like 2PC might be a problem. On the other hand, those supporting availability (for example, the so-called eventual consistency) can provide a temporary mechanism that can temporarily maintain consistency inconsistent when services work (Helland, 2007).

Methodology
To analyze distributed transaction behavior, this paper compiles pre-2019 literature synthesizing 2PC, the Saga Pattern, and complementary techniques. The information sources used in the study include published papers, industry examples, and technical blogs. As it will be seen in the following sources, each offers a unique view of managing microservices transactions.

1. Literature Review
The review of the literature of the articles published before 2019 forms the primary data source that informs the foundation of this article. This is true because the four research articles centre their discussion on theoretical and practical issues of distributed transactions in relation to distributed systems, including 2PC and Saga patterns. These articles offer basic knowledge of distributed transactions and their management problems.

Summary of Transaction Patterns

Transaction Pattern	Characteristics	Strengths	Weaknesses
Two-Phase Commit (2PC)	Synchronous, atomic	Strong consistency guarantee	High latency, single point of failure
Saga Pattern	Asynchronous, compensating	Fault tolerance, scalability	Complex design, eventual consistency

2. Data Sources
The information used in this research comprises of articles, and technical blogs, case studies. These sources contain all the theoretical framework, examples of 2PC and Saga, their advantages and implementation of distributed transaction problems that are needed for the work.

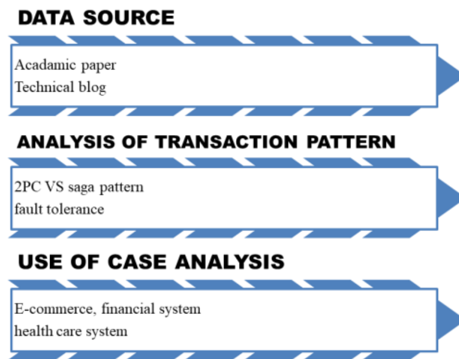
3. Analysis of the transactional history
One of the important subjects of this study is the evaluation and characterization of the comparison of Two-Phase Commit (2PC) protocol and Saga Pattern. These transaction patterns are assessed for scalability, fault tolerance, consistency and availability with regard to the requirements of the distributed microservices architectural style.
Cases in Distributed Transactions

Use Case	Transaction Pattern	Key Challenge	Solution
E-commerce	Saga Pattern	Payment failure handling	Compensating transactions (Gupta

Platform			et al., 2015)
Financial Systems	2PC, Saga Pattern	Maintaining consistency	Hybrid approach (Pritchett, 2008)
Health care Systems	Saga Pattern	Coordination of multiple services	Eventual consistency (Sanders et al., 2017)

4. Use Case Analysis
This section looks into real life use of distributed transaction management patterns in e-commerce, financial and health sectors. Two examples of the 2PC and Saga patterns are presented, and case studies are used to make comparisons of how both patterns promote the consistency of data and the fault-tolerant abilities of large distributed systems.

III. LITERATURE REVIEW



This methodology diagram provides a clear, step-by-step visualization of how the research was conducted, from literature review to analysis of use cases, guiding the reader through the process of synthesizing and evaluating distributed transaction management patterns.

Discussion
Microservices' distributed transaction management examined consensus reached in consensus, fault tolerance, and scalability. Two of them are the Two-Phase Commit (2PC) and the Saga Pattern, which have different advantages and drawbacks based on the application of the distributed system.

1. Understanding Two-Phase Commit (2PC) and Its Limitiations
This makes 2PC ensure that all the participants either commit the transaction or roll back the transaction. However, it has some cons such as high latency, resource blocking and has single point of failure which are completely not suitable for the large scalable fault tolerant microservices systems. The fact that the two phases of 2PC happen simultaneously introduces a level of latency and failure in the system has known to create system contention thereby reducing scalability and availability.

2. The Saga Pattern: A Scalable Alternative
The Saga Pattern is a better solution to 2PC because of using atomic sub-transactions, each of which has a compensating action in the event of failure. by not in the meantime require global locking and a centralized co-ordination to support this type of consistency level followed by extendibility and tolerance to failure. Nevertheless, its implementation calls for proper design of the compensating actions to be undertaken and may pose some difficulties in replicability across the time horizon.

3. Some of the properties include; Event-Driven Architectures and Fault Tolerance.
A second way that EDA contributes to DTMS is by de-coupling of services and asynchronous communication. This enhances high availability, reliability, and utilization through making services run independently. With help of message brokers and popular ones are Kafka or RabbitMQ, the systems can be designed as fail-safe, where the events can be taken as a buffer and the systems will be able to be eventually consistent. However, these modes of event delivery and the message consistency require monitoring to prevent data integrity issues.

4. Selecting the Right Tools and Patterns
The choice between 2PC, Saga, and event-driven architectures depends on the application's requirements:

2PC: Most applicable in systems that must be very homogeneous but not in large, highly reliable systems.
Saga Pattern: Designed for large applications that require availability and basic tolerance to faults, with eventual consistency.
Event-Driven Architecture: Most suitable for large scale, distributed environments where service loose coupling and high availability are primary requirements.
When selecting a pattern, then performance, scalability and fault tolerance must be met so that microservices are effective in managing distributed transactions.

CONCLUSION

Any intervention involving distributed transactions in microservices architectures must consider conflicting priorities for consistency, scalability and robustness. The numerous and multiple steps required in managing transactions across distributed services imply that the best patterns and tools must be selected based on the current needs of the network. Saga and event-driven transactions have appeared as specific solutions, which overcome problems associated with using such protocols as Two-Phase Commit (2PC). These patterns offer more flexibilities; make it possible for services to run autonomously, at the same time still maintaining data integrity over the various service phases.

Messages queuing using Kafka as an asynchronous messaging system and other distributed databases including Cassandra, and MongoDB are instrumental in these approaches in supporting transaction management strategies. These tools enable eventual consistency, scalability and fault tolerance that are important for the sustained operation of the large scale distributed systems.

As with the current solutions they do offer sound frameworks for coordinating distributed transactions except for some shades of merkmal. Subsequent studies should explore how other advanced technologies such as Artificial Intelligence (AI) can be integrated to improve fault forecasting, automated remediation in distributed systems' reinforcement. The AI integrated systems could also proactively anticipate possible transactions' failure scenarios, correct the work processes and the transaction protocols, as well as optimize the general system availability.

REFERENCES

[1] Limón, X., Guerra-Hernández, A., Sánchez-García, A. J., & Arriaga, J. C. P. (2018, October). SagaMAS: a software framework for distributed transactions in the microservice architecture. In *2018 6th International Conference in Software Engineering* Research and Innovation (CONISOFT) (pp. 50-58). IEEE.

[2] Lungu, S., & Nyirenda, M. (2024). Current Trends in the Management of Distributed Transactions in Micro-Services Architectures: A Systematic Literature Review. Open Journal of Applied Sciences, 14(9), 2519-2543.

[3] Daraghmi, E., Zhang, C. P., & Yuan, S. M. (2022). Enhancing saga pattern for distributed transactions within a microservices architecture. Applied Sciences, 12(12), 6242.

[4] Salah, T., Zemerly, M. J., Yeun, C. Y., Al-Qutayri, M., & Al-Hammadi, Y. (2016, December). The evolution of distributed systems towards microservices architecture. In 2016 11th International Conference for Internet Technology and Secured Transactions (ICITST) (pp. 318-325). IEEE.

[5] Bashtovyi, A., & Fechan, A. (2024). DISTRIBUTED TRANSACTIONS IN MICROSERVICE ARCHITECTURE: INFORMED DECISION-MAKING STRATEGIES.

[6] Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F., & Edmonds, A. (2015, April). An architecture for self-managing microservices. In *Proceedings of the 1st international workshop on automated incident management in cloud* (pp. 19-24).

[7] Yadav, P. S. DESIGN AND EVALUATION OF EVENT-DRIVEN ARCHITECTURES FOR TRANSACTION MANAGEMENT IN MICROSERVICES.

[8] Shabani, I., Mëziu, E., Berisha, B., & Biba, T. (2021). Design of modern distributed systems based on microservices

architecture. *International Journal of Advanced Computer Science and Applications*, *12*(2).

[9] Navarro, A. (2022). Fundamentals of Transaction Management in Enterprise Application Architectures. *IEEE Access*, *10*, 124305-124332.

[10] Nylund, W. (2023). Comparing Transaction Management Methods in Microservice Architecture.

[11] Zhang, G., Ren, K., Ahn, J. S., & Ben-Romdhane, S. (2019, April). GRIT: consistent distributed transactions across polyglot microservices with multiple databases. In 2019 IEEE 35th International Conference on Data Engineering (ICDE) (pp. 2024-2027). IEEE.

[12] González-Aparicio, M. T., Younas, M., Tuya, J., & Casado, R. (2023). A transaction platform for microservices-based big data systems. Simulation Modelling Practice and Theory, 123, 102709.

[13] Godage, S., Kumar, T. R., Pandya, H., Bhosale, S., & Patil, R. (2023). Web Interface for Distributed Transaction System. Computer Integrated Manufacturing Systems, 29(6), 214-227.

[14] Christudas, B., & Christudas, B. (2019). Distributed Transactions. Practical Microservices Architectural Patterns: Event-Based Java Microservices with Spring Boot and Spring Cloud, 385-481.

[15] Fan, P., Liu, J., Yin, W., Wang, H., Chen, X., & Sun, H. (2020). 2PC*: a distributed transaction concurrency control protocol of multi-microservice based on cloud computing platform. Journal of Cloud Computing, 9, 1-22.

[16] Newman, S. (2021). Building microservices. " O'Reilly Media, Inc.".

[17] Christudas, B., & Christudas, B. (2019). Transactions and Microservices. Practical Microservices Architectural Patterns: Event-Based Java Microservices with Spring Boot and Spring Cloud, 483-541.