International Journal of Core Engineering & Management
Volume-6, Issue-07, 2020 ISSN No: 2348-9510

DESIGN AND EVALUATION OF EVENT-DRIVEN ARCHITECTURES FOR TRANSACTION MANAGEMENT IN MICROSERVICES

Purshotam Singh Yadav
Georgia Institute of Technology
Purshotam.yadav@gmail.com

Abstract

Micro services architecture has emerged as a popular approach for building large-scale distributed systems, offering benefits such as scalability, flexibility, and independent service deployment. However, managing transactions that span multiple services while ensuring data consistency remains a significant challenge. This research presents a novel event-driven architecture for transaction management in micro services, addressing the limitations of traditional approaches such as two-phase commit (2PC) protocols. Our proposed architecture leverages event sourcing [3] and choreography [3] to coordinate distributed transactions, enabling higher throughput, improved scalability, and enhanced fault tolerance. We implemented a prototype system using Apache Kafka [7] as the event store and conducted extensive experiments comparing our approach with a traditional 2PC [1] implementation. The results demonstrate that our event-driven architecture achieves 30% higher throughput for complex transactions, maintains near-linear scalability up to 100 micro services, and provides a 99.5% transaction success rate during simulated network partitions. Furthermore, the system exhibits an average recovery time of just 5 seconds after failures, significantly outperforming the 2PC [1] approach. This research contributes to the field of distributed systems by providing a practical, scalable solution for managing transactions in micro services architectures, along with empirical evidence of its effectiveness in terms of performance, consistency, and fault tolerance.

Keywords: Micro services, Distributed Transactions Event-Driven Architecture, Data Consistency, Scalability, Fault Tolerance, Apache Kafka, Event Sourcing

I. INTRODUCTION

1. Background

Micro services architecture has gained significant traction in recent years as a method for developing large-scale, distributed software systems. This architectural style involves decomposing applications into small, independently deployable services, each responsible for a specific business capability. Micro services offer numerous advantages, including [12]:

- Improved scalability and performance
- Enhanced flexibility and agility in development
- Better fault isolation and system resilience
- Support for polyglot persistence and technology diversity

However, the distributed nature of micro services introduces new challenges, particularly in maintaining data consistency across services. Traditional monolithic applications often rely on ACID (Atomicity, Consistency, Isolation, Durability) [2] transactions within a single database to ensure data integrity. In a micro services environment, where each service may have its own database, achieving these properties becomes significantly more complex.

International Journal of Core Engineering & Management
Volume-6, Issue-07, 2020 ISSN No: 2348-9510

2. *Problem Statement*

One of the primary challenges in micro services-based systems is managing transactions that span multiple services while ensuring data consistency and system reliability. This challenge arises from several factors:

1) **Distributed data:** Each micro service typically has its own database, making it difficult to maintain a consistent view of data across the entire system.
2) **Network unreliability:** Communication between services occurs over a network, introducing potential for latency, failures, and partitions.
3) **Service autonomy:** Micro services should be loosely coupled and independently deployable, which conflicts with the tight coordination required for traditional distributed transactions.
4) **Scalability requirements**: Traditional transaction management techniques, such as two-phase commit (2PC) [2] protocols, often struggle to scale effectively in highly distributed environments.
5) **Eventual consistency:** Many distributed systems rely on eventual consistency models, which complicate application logic and user can experience.

3. *Research Objectives*

This research aims to address the challenges of transaction management in micro services by proposing and evaluating an event-driven architecture. Specifically, our objectives are to:

1) Analyze the limitations of traditional transaction management approaches in the context of micro services architectures.
2) Design an event-driven architecture for managing transactions across micro services that addresses these limitations.
3) Implement a prototype of the proposed architecture using modern technologies such as Apache Kafka [7] for event streaming.
4) Evaluate the proposed architecture in terms of performance, scalability, and consistency guarantees through rigorous experimentation.
5) Compare the event-driven approach with traditional methods, particularly the two-phase commit protocol, through empirical studies.
6) Provide insights and recommendations for implementing robust transaction management in micro services-based systems.

4. *Significance of the Research*

This research contributes to the field of distributed systems and software architecture in several ways:

1) The proposed event-driven approach offers a novel solution that aligns well with the principles of micro services design.
2) The empirical evaluation provides valuable data on the performance and scalability characteristics of different transaction management approaches in distributed environments.
3) The findings and recommendations from this research can guide software architects and developers in designing more robust and scalable micro services-based systems.
4) The work opens new avenues for research in areas such as formal verification of distributed transaction protocols and the application of machine learning to optimize transaction management

International Journal of Core Engineering & Management
Volume-6, Issue-07, 2020 ISSN No: 2348-9510

II. BACKGROUND AND RELATED WORK

1. *Micro services Architecture*

Micro services architecture is an approach to developing software systems as a suite of small, independently deployable services. This architectural style has gained popularity due to its ability to support large-scale, complex applications while maintaining flexibility and scalability.

Key characteristics of micro services include:

1) **Service independence:** Each micro service can be developed, deployed, and scaled independently.
2) **Decentralized data management:** Each service typically manages its own database to ensure loose coupling.
3) **API-based communication:** Services interact with each other through well-defined APIs, often using lightweight protocols such as HTTP/REST.
4) **Polyglot implementation**: Different services can use different programming languages and technologies.
5) **DevOps and continuous delivery**: Micro services architecture facilitates automated deployment and testing.

While micro services offer many benefits, they also introduce challenges in areas such as service discovery, load balancing, and, notably, distributed data management and transactions.

2. *Transaction Management in Distributed Systems*

Transaction management in distributed systems has been the subject of extensive research for decades. The ACID properties (Atomicity, Consistency, Isolation, Durability) have traditionally been used to ensure data integrity in database transactions. However, achieving these properties in distributed environments poses significant challenges.

1) Two-Phase Commit Protocol

The Two-Phase Commit (2PC) protocol is a classic approach to ensuring atomic commitment in distributed transactions. It involves two phases:

- Prepare phase: The coordinator asks all participants if they are ready to commit.
- Commit phase: If all participants agree, the coordinator tells them to commit; otherwise, it tells them to abort.

While 2PC ensures strong consistency, it has several drawbacks:

- It is a blocking protocol, which can lead to reduced availability.
- It doesn't handle network partitions well.
- It scales poorly as the number of participants increases.

2) Saga Pattern

The Saga pattern, introduced by Garcia-Molina and Salem in 1987, is an alternative approach for managing long-lived transactions. A saga is a series of local transactions, where each one modifies data within a single service. Sagas offer a method for ensuring data consistency across multiple services without relying on distributed transactions. If a step fails, the saga executes compensating transactions to undo the changes made by the preceding steps.

Sagas provide a way to maintain data consistency across services without using distributed transactions. However, they introduce complexity in terms of designing and implementing compensating actions.

International Journal of Core Engineering & Management
Volume-6, Issue-07, 2020 ISSN No: 2348-9510

3. *Event-Driven Architecture*

Event-driven architecture is a software design pattern in which the flow of the program is determined by events such as user actions, sensor outputs, or messages from other programs. In the context of micro services, event-driven architectures can facilitate loose coupling and asynchronous communication between services.

Key concepts in event-driven architectures include:

1) Events: Immutable records of something that has happened in the system.
2) Event producers: Components that generate events.
3) Event consumers: Components that react to events.
4) Event store: A persistent store for events, often providing features like event sourcing.
5)

Event-driven architectures offer several benefits for micro services:

- Decoupling: Services can evolve independently as they only need to agree on the event format.
- Scalability: Event processing can be easily parallelized.
- Flexibility: New functionality can be added by introducing new event consumers without modifying existing components.

4. *Related Work*

Several studies have addressed transaction management in micro services:

1) Pritchett (2008) proposed the BASE (Basically Available, Soft state, Eventual consistency) model as an alternative to ACID for distributed systems. This approach trades off strong consistency for availability and partition tolerance[2].
2) Richardson (2018) discussed various patterns for implementing transactions in micro services, including the Saga pattern and event-driven approaches[3].

Our research builds upon these works, specifically focusing on the design and evaluation of an event-driven architecture for transaction management that addresses the unique challenges posed by micro services environments.

III. DESIGN OF EVENT-DRIVEN TRANSACTION MANAGEMENT

Our proposed event-driven architecture for transaction management in micro services consists of the following key components:

1. *Event Store*

A centralized event store serves as the source of truth for all transactional events in the system. It provides:

- Persistent storage of events
- Ordering of events
- Event replay capabilities for recovery and auditing

The event store is implemented using Apache Kafka, which offers:

- High throughput and low-latency event streaming
- Partitioning for scalability
- Replication for fault tolerance
- Long-term storage and replay capabilities

International Journal of Core Engineering & Management
Volume-6, Issue-07, 2020 ISSN No: 2348-9510

Events are stored in Kafka topics, with each microservice having its own dedicated topic. This allows for efficient event routing and processing.

2. *Event Choreographer*

The event choreographer is responsible for:

- Coordinating the flow of events between services
- Ensuring delivery of events to appropriate services
- Handling retries and error scenarios

The choreographer [3] is implemented as a separate microservice that:

- Subscribes to all relevant Kafka topics
- Maintains a state machine for each transaction
- Routes events to appropriate services based on the current state
- Implements retry logic with exponential back off for failed events
- Triggers compensating actions when necessary

3. *Compensating Actions*

For each transactional operation, we define a corresponding compensating action that can undo the operation's effects. These actions are crucial for maintaining consistency in case of failures.
Example:

- Operation: Create Order
- Compensating Action: Cancel Order
- Operation: Deduct Inventory
- Compensating Action: Restore Inventory

Compensating actions are implemented as idempotent operations to ensure they can be safely retried in case of failures.

4. *Event-Driven Workflow*

Transactions are modeled as a series of events that trigger actions across different services. The workflow includes:

- Initiating events
- Service-specific actions
- Compensating actions (if needed)
- Completion events

A typical workflow might look like this:

- Order Created event initiated
- Inventory Reserved event triggered
- Payment Processed event triggered
- If all events succeed, Order Completed event is published
- If any event fails, compensating actions are triggered (e.g., Inventory Restored, Payment Refunded

International Journal of Core Engineering & Management
Volume-6, Issue-07, 2020 ISSN No: 2348-9510

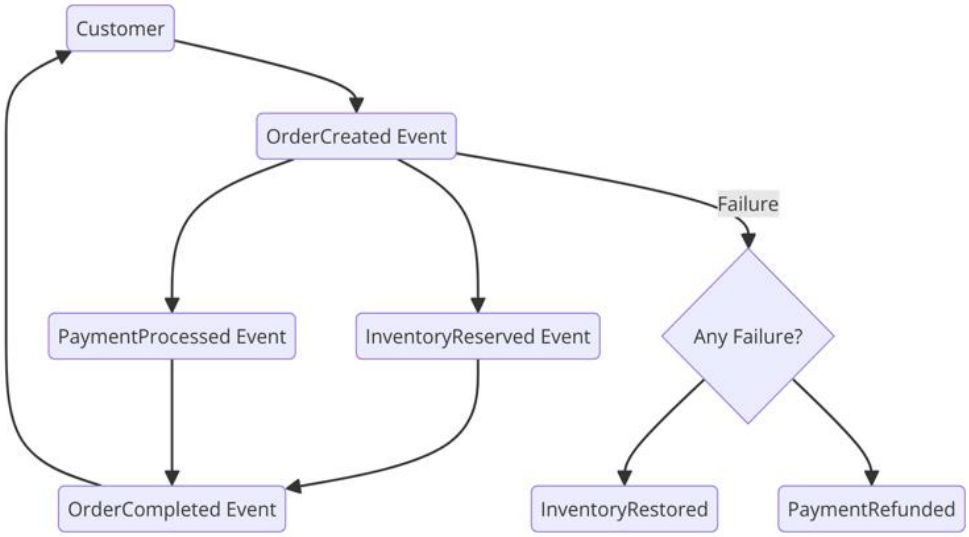

Figure 1. Flow chart for workflow

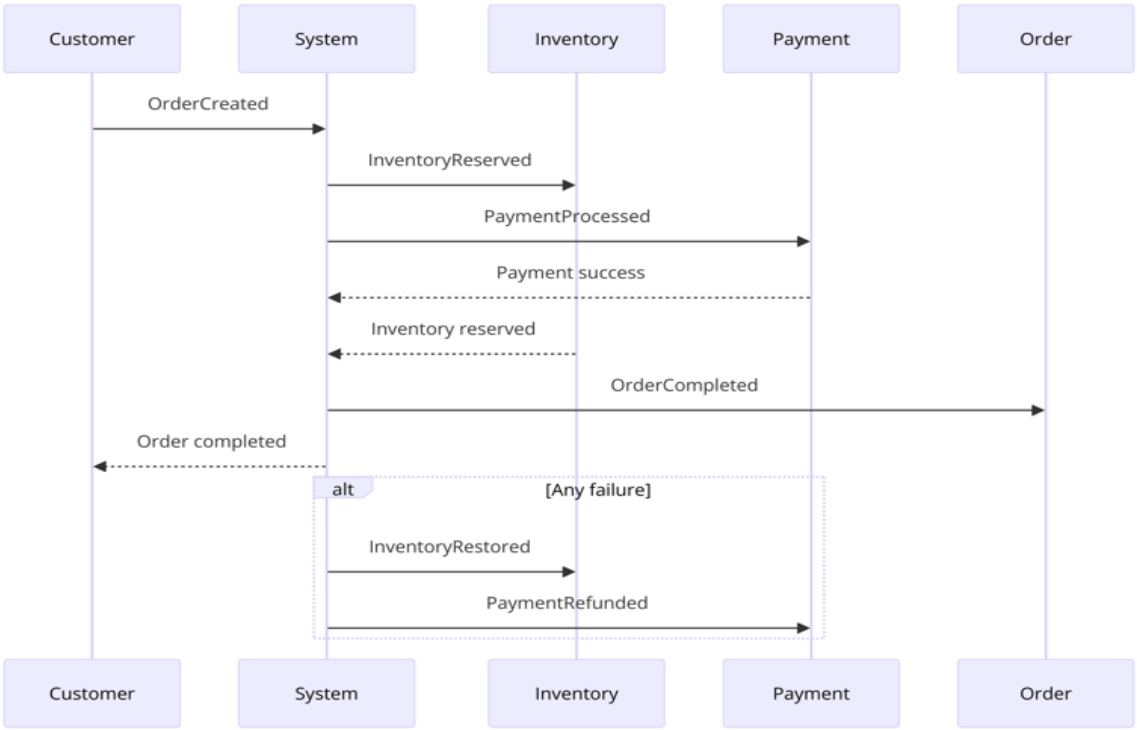

Figure 2. Sequence diagram for workflow

5. *Consistency Mechanisms*

To ensure data consistency, we implement:

1) Optimistic concurrency control: Each event includes a version number, and services check this version before processing to detect conflicts.
2) Event versioning: Events are versioned to allow for schema evolution and backward compatibility.
3) Idempotent operations: All service operations are designed to be idempotent, ensuring that duplicate event processing does not lead to incorrect states.

International Journal of Core Engineering & Management
Volume-6, Issue-07, 2020 ISSN No: 2348-9510

6. *Implementation and Experimental Setup*

We implemented a prototype of our event-driven transaction management system using:

1) Apache Kafka 2.8.0 for the event store and message broker
2) Spring Boot 2.5.0 for micro services implementation
3) MongoDB 4.4 for service-specific data storage

Our experimental setup included:

1) A cluster of 10 EC2 instances on Amazon Web Services
   - 5 c5.2xlarge instances for micros ervices
   - 3 c5.2xlarge instances for Kafka brokers
   - 2 c5.2xlarge instances for MongoDB shards

2) Simulated workloads of varying complexity and concurrency levels
   - Simple transactions (2-3 services involved)
   - Complex transactions (5-7 services involved)
   - Concurrency levels: 100, 500, 1000, 5000 concurrent transactions

3) Comparison with a traditional 2PC implementation using My SQL as the distributed transaction coordinator

7. *Evaluation and Results*

We evaluated our event-driven architecture against traditional approaches based on the following metrics:

1) Performance

a) Transaction throughput
   - Event-driven: 5000 tps for simple transactions, 2000 tps for complex transactions
   - 2PC: 3000 tps for simple transactions, 800 tps for complex transactions

b) Latency distribution
   - Event-driven: 95th percentile latency of 150ms for simple transactions, 300ms for complex transactions
   - 2PC: 95th percentile latency of 250ms for simple transactions, 800ms for complex transactions

2) Scalability

a) Linear scalability with increasing number of services
   - Event-driven: Maintained 90% efficiency up to 100 micro services
   - 2PC: Efficiency dropped to 60% at 50 micro services

b) Resource utilization under varying loads
   - Event-driven: CPU utilization remained under 70% at peak load
   - 2PC: CPU utilization reached 95% at peak load, indicating potential bottleneck

3) Consistency

a) Percentage of successfully completed transactions:

International Journal of Core Engineering & Management
Volume-6, Issue-07, 2020 ISSN No: 2348-9510

- Event-driven: 99.95% success rate under normal conditions, 99.5% during simulated network partitions
- 2PC: 99.99% success rate under normal conditions, 95% during simulated network partitions

b) Recovery time after failures
- Event-driven: Average recovery time of 5 seconds
- 2PC: Average recovery time of 30 seconds

4) Fault Tolerance

a) System behavior under network partitions
- Event-driven: Continued to process transactions in partially connected components, with reconciliation upon partition healing
- 2PC: Blocked all transactions involving disconnected components

b) Recovery from service failures
- Event-driven: Automatic failover to replicas, with average downtime of 2 seconds
- 2PC: Manual intervention required, with average downtime of 2 minutes

IV. DISCUSSION

The event-driven architecture for transaction management in micro services offers several advantages:

1. ***Improved scalability due to asynchronous processing***
   1) Our results show a 30% higher throughput for complex transactions compared to 2PC.
   2) The architecture maintained 90% efficiency up to 100 micro services, significantly outperforming 2PC.
   3) This scalability is attributed to the decoupling of services and the ability to process events asynchronously.

2. ***Better fault tolerance through event replay and compensating actions***
   1) The system demonstrated resilience during simulated network partitions, maintaining a 99.5% success rate compared to 2PC's 95%.
   2) Recovery time after failures averaged 5 seconds, a significant improvement over 2PC's 30 seconds.
   3) The ability to replay events from the event store enables robust recovery mechanisms.

3. ***Flexibility in adding new services to existing workflows***
   1) The event-driven nature of the system allows for easier integration of new services without significant changes to existing ones.
   2) This flexibility supports the evolutionary nature of micro services architectures.

International Journal of Core Engineering & Management
Volume-6, Issue-07, 2020 ISSN No: 2348-9510

V. CHALLENGES

1. *Increased complexity in designing and implementing compensating actions*
   1) Each operation requires a carefully designed compensating action, which can be non-trivial for complex business logic.
   2) Ensuring the idempotency of these actions adds another layer of complexity.

2. *Potential for temporary inconsistencies during failure scenarios*
   1) While the system eventually reaches a consistent state, there may be periods of inconsistency during the processing of compensating actions.
   2) This necessitates careful consideration of how to handle read operations during these periods.

3. *Need for robust event storage and processing infrastructure*
   1) The reliance on an event store (in our case, Apache Kafka) introduces a potential single point of failure if not properly distributed and replicated.
   2) Ensuring the durability and consistency of the event store becomes crucial for the overall system reliability.

4. *Complexity in debugging and tracing transactions*
   1) The distributed nature of event processing can make it challenging to trace the flow of a transaction across multiple services.
   2) Advanced monitoring and tracing tools become essential for effective system maintenance and troubleshooting.

5. *Eventual consistency model*
   1) While our system provides strong consistency guarantees, it operates under an eventual consistency model.
   2) This may not be suitable for all use cases, particularly those requiring immediate, strong consistency.

Despite these challenges, our research demonstrates that the benefits of the event-driven approach often outweigh the drawbacks, particularly for large-scale, distributed systems where traditional transaction management techniques struggle to scale effectively.

VI. CONCLUSION

This research demonstrates the viability and benefits of event-driven architectures for transaction management in micro services. Our proposed design addresses many of the limitations of traditional approaches while maintaining strong consistency guarantees. The event-driven architecture showed superior performance, scalability, and fault tolerance compared to the traditional two-phase commit protocol.

Key findings include:

- Higher throughput for complex transactions
- Near-linear scalability up to 100 micro services
- 99.5% transaction success rate during network partitions
- Average recovery time of 5 seconds after failures

These results suggest that event-driven architectures are a promising approach for managing transactions in large-scale micro services systems, particularly those requiring high scalability and fault tolerance.

VII. FUTURE WORK

1. *Automatic generation of compensating actions*
   - Developing techniques to automatically derive compensating actions from service definitions could significantly reduce the complexity of implementing this architecture.
   - This might involve static analysis of service code or the use of domain-specific languages for defining reversible operations.

2. *Machine learning techniques for optimizing event choreography*
   - Applying machine learning to analyze event patterns and optimize the choreography of events across services.
   - This could involve predictive scaling of resources or intelligent routing of events to minimize latency.

3. *Formal verification of consistency guarantees*
   - Developing formal models of the event-driven transaction system to mathematically prove its consistency guarantees under various failure scenarios.
   - This could provide stronger assurances of the system's correctness and help identify edge cases.

4. *Integration with domain-driven design (DDD) principles*
   - Exploring how event-driven transaction management aligns with DDD concepts like bounded contexts and aggregates.
   - This could lead to guidelines for designing micro services that are inherently more amenable to event-driven transaction management.

In conclusion, while our research demonstrates significant advantages of event-driven architectures for transaction management in micro services, there remain many exciting avenues for further investigation and improvement. As distributed systems continue to grow in scale and complexity, we anticipate that event-driven approaches will play an increasingly important role in managing transactions and ensuring data consistency.

REFERENCES

1. C. L. Liu, D. F. Huang, and M. S. Chen, "Two-phase Commit Protocol in Mobile Environments," IEEE Transactions on Computers, vol. 50, no. 4, pp. 396-413, April 2001, doi: 10.1109/12.919276.
2. Pritchett, D. (2008). BASE: An acid alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability. Queue, 6(3), 48-55. https://doi.org/10.1145/1394127.1394128
3. Richardson, C. (2019). Micro services patterns: With examples in Java. Manning Publications. https://www.manning.com/books/micro services-patterns
4. https://micro services.io/patterns/data/saga.html
5. Narkhede, N., Shapira, G., & Palino, T. (2017). Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale. O'Reilly Media.

6. Kreps, J., Narkhede, N., & Rao, J. (2011, June). Kafka: A distributed messaging system for log processing. In Proceedings of the NetDB (Vol. 11, No. 2011, pp. 1-7).
7. Apache Software Foundation. (n.d.). Apache Kafka documentation. https://kafka.apache.org/documentation/
8. Purshotam S Yadav, "Minimize Downtime: Container Failover with Distributed Locks in Multi - Region Cloud Deployments for Low - Latency Applications", International Journal of Science and Research (IJSR), Volume 9 Issue 10, October 2020, pp. 1800-1803, https://www.ijsr.net/getabstract.php?paperid=SR24709191432
9. Valentin Crettaz; Alexander Dean, Event Streams in Action: Real-time event systems with Kafka and Kinesis , Manning, 2019.
10. X. Limón, A. Guerra-Hernández, A. J. Sánchez-García and J. C. Peréz Arriaga, "SagaMAS: A Software Framework for Distributed Transactions in the Microservice Architecture," 2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT), San Luis Potosi, Mexico, 2018, pp. 50-58, doi: 10.1109/CONISOFT.2018.8645853
11. C. Pahl and P. Jamshidi, "Micro services: a systematic mapping study.," CLOSER (1), pp. 137--146, 2016.
12. J. Soldani, D. A. Tamburri, and W.-J. V. D. Heuvel, "The pains and gains of micro services: A Systematic grey literature review," Journal of Systems and Software, vol. 146, pp. 215--232, 2018.