NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

POSTGRADUATE PROGRAM
COMPUTER SCIENCE

MASTER THESIS

Distributed Transactions using the SAGA pattern

Panagiotis I. Ioannidis

***Supervisor*:** **Mema Roussopoulos,** Professor

ATHENS

March 2023

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΠΛΗΡΟΦΟΡΙΚΗ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κατανεμημένες Συναλλαγές με χρήση του προτύπου SAGA

Παναγιώτης Ι. Ιωαννίδης

***Επιβλέπουσα*:** **Μέμα Ρουσσοπούλου,** Καθηγήτρια

ΑΘΗΝΑ

Μάρτιος 2023

MASTER THESIS

Distributed Transactions using the SAGA pattern

Panagiotis I. Ioannidis
S.N.: CS2200006

SUPERVISOR: **Mema Roussopoulos,** Professor

THESIS COMMITTEE: **Alex Delis,** Professor
Mema Roussopoulos, Professor
Panagiotis Liakos, Postdoctoral Researcher

March 2023

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κατανεμημένες Συναλλαγές με χρήση του προτύπου SAGA

Παναγιώτης Ι. Ιωαννίδης
Α.Μ.: CS2200006

ΕΠΙΒΛΕΠΟΥΣΑ: **Μέμα Ρουσσοπούλου,** Καθηγήτρια

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ	**Αλέξης Δελής,** Καθηγητής
	Μέμα Ρουσσοπούλου, Καθηγήτρια
	Παναγιώτης Λιάκος, Μεταδιδακτορικός Ερευνητής

Μάρτιος 2023

ΠΕΡΙΛΗΨΗ

Οι κατανεμημένες συναλλαγές αποτελούν κρίσιμο συστατικό των σύγχρονων κατανεμημένων συστημάτων, καθώς εξασφαλίζουν τη συνοχή και την ακεραιότητα των δεδομένων σε πολλαπλές βάσεις δεδομένων, διακομιστές ή υπηρεσίες. Είναι ιδιαίτερα σημαντικές στα σύγχρονα κατανεμημένα συστήματα, τα οποία είναι συχνά πολύπλοκα, δυναμικά με πολλά διασυνδεδεμένα στοιχεία, καθώς και σε συστήματα που περιλαμβάνουν ευαίσθητα ή κρίσιμα δεδομένα. Χωρίς κατανεμημένες συναλλαγές, θα ήταν δύσκολο να διασφαλιστεί ότι τα δεδομένα ενημερώνονται με συνέπεια και ορθότητα σε πολλαπλά συστήματα, γεγονός που θα μπορούσε να οδηγήσει σε ασυνέπειες και απώλεια δεδομένων.

Ο στόχος μιας κατανεμημένης συναλλαγής είναι να διασφαλίσει ότι όλες οι λειτουργίες είτε όλες δεσμεύονται και τίθενται σε ισχύ είτε όλες ανακαλούνται και δεν έχουν κανένα αποτέλεσμα, ακόμη και σε περίπτωση αποτυχιών ή σφαλμάτων. Ωστόσο, οι παραδοσιακές προσεγγίσεις για την υλοποίηση κατανεμημένων συναλλαγών, όπως το Two-Phase Commit protocol (2PC), μπορεί να είναι πολύπλοκες και επιρρεπείς σε σφάλματα.

Το πρότυπο SAGA είναι μια πολλά υποσχόμενη εναλλακτική προσέγγιση για την υλοποίηση κατανεμημένων συναλλαγών, καθώς επιτρέπει μεγαλύτερη ευελιξία και ανθεκτικότητα στα κατανεμημένα συστήματα. Σε ένα SAGA, κάθε βήμα αντιμετωπίζεται ως ξεχωριστή συναλλαγή, και εάν κάποιο βήμα αποτύχει, η διαδικασία ανατρέπεται σε μια προηγούμενη γνωστή καλή κατάσταση και ενεργοποιείται μια διαδικασία χειρισμού σφαλμάτων. Αυτό επιτρέπει τη μερική αποτυχία και τη δυνατότητα ανάκαμψης από αυτήν, καθιστώντας το πρότυπο SAGA ιδιαίτερα κατάλληλο για χρήση σε σύγχρονα κατανεμημένα συστήματα.

Σε αυτή τη διπλωματική εργασία, θα διερευνήσουμε τη χρήση του προτύπου Saga για την υλοποίηση κατανεμημένων συναλλαγών σε κατανεμημένα συστήματα. Θα ξεκινήσουμε παρέχοντας μια επισκόπηση των κατανεμημένων συναλλαγών και των προκλήσεων που θέτουν στα σύγχρονα κατανεμημένα συστήματα. Στη συνέχεια θα παρουσιάσουμε το πρότυπο SAGA και θα συζητήσουμε τα οφέλη και τις προκλήσεις του. Αμέσως μετά, θα παρουσιάσουμε μια μελέτη περίπτωσης που υλοποιεί την «orchestrated» προσέγγισή μας - piSaga - για κατανεμημένες συναλλαγές σε Spring Boot microservices και θα πραγματοποιήσουμε ορισμένα πειράματα προκειμένου να μετρήσουμε την απόδοσή της σε σχέση με μια «choreographed» λύση. Τέλος, θα καταλήξουμε συνοψίζοντας τις βασικές ιδέες και τις συνεισφορές της ανάλυσής μας.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Κατανεμημένα Συστήματα

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: SAGA, κατανεμημένες συναλλαγές, μηχανική λογισμικού, αρχιτεκτονική λογισμικού, microservices

ABSTRACT

Distributed transactions are a crucial component of modern distributed systems, as they ensure data consistency and integrity across multiple databases, servers, or services. They are essential in modern distributed systems, which are often complex and dynamic environments with many interconnected components, and in systems that involve sensitive or critical data. Without distributed transactions, ensuring that data is consistently and correctly updated across multiple systems would be challenging, which could lead to inconsistencies and data loss.

The goal of a distributed transaction is to ensure that all operations either commit and take effect or roll back and have no effect, even in the case of failures or errors. However, traditional approaches to implementing distributed transactions, such as the Two-Phase commit (2PC) protocol, can be inflexible and prone to failure in complex and dynamic environments.

The SAGA pattern is a promising alternative approach to implementing distributed transactions, as it allows for more flexibility and resilience in distributed systems. In a SAGA, each step in a long-running business process is treated as a separate transaction, and if any step fails, the process is rolled back to a previously known good state, and an error-handling process is triggered. This allows for partial failures and the ability to recover from them, making the SAGA pattern particularly well-suited for use in modern distributed systems.

In this thesis, we will explore the use of the SAGA pattern for implementing distributed transactions in distributed systems. We will begin by providing an overview of distributed transactions and their challenges in modern distributed systems. We will then introduce the SAGA pattern and discuss its benefits and challenges. Next, we will present a case study that implements our orchestrated approach - piSaga - for distributed transactions in Spring Boot microservices. We will also conduct experiments to measure its performance against a choreographed solution. Finally, we will summarize the key insights and contributions of our analysis.

SUBJECT AREA: Distributed Systems

KEYWORDS: SAGA, distributed transactions, software engineering, software architecture, microservices

To my family for supporting me throughout my entire academic journey.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my family for their constant support and encouragement throughout my academic journey. Their unwavering belief in my abilities has been a constant source of motivation and inspiration.

I would also like to extend my heartfelt thanks to my professors, and especially to Professor Mema Roussopoulos, for their invaluable guidance and mentorship. Their expertise, knowledge, and passion for their subjects have profoundly impacted my academic and professional development. Their generous assistance and support have been instrumental in helping me achieve my goals. Additionally, I would like to thank Professor Alex Delis and Panagiotis Liakos for honoring me with their presence and their participation as my thesis committee.

Finally, I would like to thank all of the individuals who have directly and indirectly contributed to this research and my degree completion. I am deeply grateful for their help and support.

CONTENTS

1. INTRODUCTION 15

2. BACKGROUND 17

2.1 CAP Theorem 17

2.2 ACID 18

2.3 ACD 19
- 2.3.1 Lack of Isolation 19
- 2.3.2 Countermeasures 20

2.4 Microservices 21

2.5 SAGA compared to other distributed transactions protocols 22
- 2.5.1 Two-Phase commit (2PC) 22
- 2.5.2 Two-Phase Commit with Recovery (2PC*) 23
- 2.5.3 Try, Commit, and Cancel (TCC) 23
- 2.5.4 Generalized Recursive Idealized Transactions (GRIT) 24

3. RELATED WORK 26

4. TRANSACTIONAL SAGA 28

4.1 Coordination 28
- 4.1.1 Choreography 28
- 4.1.2 Orchestration 29

4.2 Inter-process Communication 31
- 4.2.1 Synchronous communication 31
- 4.2.2 Asynchronous communication 31

4.3 Consistency 31
- 4.3.1 Atomic consistency 32
- 4.3.2 Eventual consistency 32

4.4 Transaction types 32

4.5 State machine modeling 32

4.6 The variations of the SAGA pattern 33
- 4.6.1 SAO SAGA Pattern 34
- 4.6.2 SAC SAGA Pattern 37
- 4.6.3 SEO SAGA Pattern 39
- 4.6.4 SEC SAGA Pattern 40
- 4.6.5 AAO SAGA Pattern 42
- 4.6.6 AAC SAGA Pattern 43
- 4.6.7 AEO SAGA Pattern 45

4.6.8 AEC SAGA Pattern 47

5. PISAGA 49

5.1 Dependencies 49

5.1.1 Spring Boot 49

5.1.2 Apache Kafka 50

5.2 piSaga library 51

5.3 piSaga case study 52

6. EVALUATION 54

6.1 Implementations 54

6.1.1 ConcurentHashMap cache 55

6.2 Experiment Scenarios 55

6.2.1 Scenario 1 – Successful order creation 55

6.2.2 Scenario 2 – Order creation fails at items reservation 56

6.2.3 Scenario 2 – Order creation fails at payment 57

6.2.4 Scenario 3 – Cancelling order 59

6.3 Experimental Results 60

6.3.1 Scenario 1 – Throughput 61

6.3.2 Scenario 2 – Required time to complete requests 64

6.3.3 Scenario 3 – Processing time for order cancellation 68

7. CONCLUSION 70

8. ABBREVIATIONS - ACRONYMS 71

REFERENCES 72

LIST OF FIGURES

Figure 2-1 Graphical representation of the CAP theorem 17

Figure 2-2 Microservices architecture high-level design 21

Figure 4-1 Choreography coordination 29

Figure 4-2 Orchestration coordination 30

Figure 4-3 SAO utilizes Sync communication, Atomic consistency, Orchestrated coordination 34

Figure 4-4 Communication between microservices in the SAO SAGA pattern 35

Figure 4-5 Compensatory transactions are issued by the orchestrator when an error occurs 35

Figure 4-6 SAC utilizes Sync communication, Atomic consistency, Choreographed coordination 37

Figure 4-7 Each microservice is responsible to issue its own compensatory transactions 37

Figure 4-8 SEO utilizes Sync communication, Eventual consistency, Orchestrated coordination 39

Figure 4-9 Compensatory transactions are issued by the orchestrator when an error occurs 39

Figure 4-10 SEC utilizes Sync communication, Eventual consistency, Choreographed coordination 40

Figure 4-11 Communication between microservices in the SEC SAGA pattern 41

Figure 4-12 AAO utilizes Async communication, Atomic consistency, Orchestrated coordination 42

Figure 4-13 Compensatory transactions are issued by the orchestrator when an error occurs 42

Figure 4-14 AAC utilizes Async communication, Atomic consistency, Choreographed coordination 43

Figure 4-15 Each microservice is responsible to issue its own compensatory transactions 44

Figure 4-16 AEO utilizes Async communication, Eventual consistency, Orchestrated coordination 45

Figure 4-17 The orchestrator communicates asynchronously with the microservices ... 45

Figure 4-18 AEC utilizes Async communication, Eventual consistency, Choreographed coordination 47

Figure 4-19 The microservices communicate asynchronously with each other 47

Figure 5-1 Create order state machine model 52

Figure 5-2 Cancel order state machine model 53
Figure 6-1 Successful order creation workflow 56
Figure 6-2 Compensatory workflow when an error occurs at the Warehouse-Service .. 57
Figure 6-3 Compensatory workflow when an error occurs at the Payment-Service 58
Figure 6-4 Order cancellation workflow at “WAREHOUSE_APPROVED” state 59
Figure 6-5 Order cancellation workflow at “PAYMENT_APPROVED” state 60
Figure 6-6 Graphical representation of throughput for 1 virtual user 62
Figure 6-7 Graphical representation of throughput for 10 virtual users.......................... 63
Figure 6-8 Graphical representation of throughput for 25 virtual users.......................... 64
Figure 6-9 Cache-piSaga needs 33 seconds to complete 10000 requests 65
Figure 6-10 Cache-piSaga needs 28 seconds to complete 10000 requests 66
Figure 6-11 Cache-piSaga needs 26 seconds to complete 1000 requests 67
Figure 6-12 piSaga needs 38 seconds to complete 10000 cancellation requests 68
Figure 6-13 piSaga needs 613 seconds to complete 100000 cancellation requests 69

LIST OF TABLES

Table 4-1 Saga pattern variations....................34
Table 6-1 Successful-request/30sec for 1 virtual user....................61
Table 6-2 Successful-request/30sec for 10 virtual user....................62
Table 6-3 Successful-request/30sec for 25 virtual user....................63
Table 6-4 Required time, in seconds, to complete 10000 requests....................65
Table 6-5 Required time, in seconds, to complete 10000 requests....................66
Table 6-6 Required time, in seconds, to complete 10000 requests....................67
Table 6-7 Processing and total time, in seconds, for 10 virtual uses and 10000 in total 68
Table 6-8 Processing and total time, in seconds, for 50 virtual uses and 100000 in total 69

1. INTRODUCTION

Microservices are a popular architectural style for building software systems that are independently deployable, scalable, flexible, resilient, and modular. They are well-suited for use in modern distributed systems that handle a large and unpredictable workload and require high availability.

In order to support such a system, we also need distributed transactions. Distributed transactions ensure that data is consistently and correctly updated across multiple microservices and databases, which helps maintain the integrity and reliability of the system. Moreover, they allow for partial failures and the ability to recover from them, which makes the system more fault-tolerant and resilient. This is especially important in mission-critical systems that need to be highly available. In addition to these benefits, distributed transactions can also improve the performance of a system by allowing multiple microservices to execute concurrently while reducing the need for costly and time-consuming communication between microservices.

This thesis aims to examine, understand, and evaluate the SAGA pattern. SAGA is a distributed transaction model that is an alternative to several traditional protocols like Two-Phase commit (2PC), Two-Phase Commit with Recovery (2PC*), Try, Commit, and Cancel (TCC), and Generalized Recursive Idealized Transactions (GRIT). Compared to the transaction protocols mentioned above, a SAGA transaction is composed of a series of sub-transactions, each of which can be undone (compensated) if necessary. This allows for more flexibility in handling failures and errors, as the system can recover from a failure by undoing any changes made as part of a sub-transaction. They also allow for more fine-grained control over the scope of a transaction, as sub-transactions can be nested and composed in various ways. Additionally, SAGA transactions can be used where traditional ACID transactions may not be feasible, such as in systems with high levels of data consistency or systems that need to support long-running operations.

However, SAGA transactions are also more complex to implement than traditional ACID transactions, and they may require more resources to maintain the necessary state information for undoing sub-transactions. Therefore, we develop piSaga, a Java library for Spring Boot. This library's primary goal is to successfully define the necessary contracts to represent our SAGA transactions using a state machine model. We aim to make SAGAs easier to understand by employing sensible naming conventions and clarifying the development process.

Lastly, we conducted several experiments to evaluate piSaga's performance and determine whether or not it can overcome the lack of isolation. For this reason, we came up with three distinct scenarios, running them utilizing cache-less and cache implementations. In the first scenario, we measure the number of requests that can be handled within 30 seconds. We achieved this by considering only the successfully completed requests and repeating the experiment for 1, 10, and 25 virtual users. In the second scenario, we examine how long it takes to complete a batch of requests when either the Warehouse-Service or the Payment-Service encounters an error. In the last scenario, we aim to investigate how piSaga performs under heavy load of request using stress testing. We achieve this by calculating the time it takes to cancel an order when 10 and 50 virtual users stream 10000 and 100000 requests, respectively.

Overall, this work makes the following contributions:

- We explain the differences between SAGA and several other protocols.
- We present the SAGA pattern and its variations.
- We develop piSaga - a Java library - to help developers apply the SAGA pattern in Spring Boot applications.
- We evaluate the performance of the piSaga library based on three scenarios and compare it with a more straightforward implementation.

The rest of the thesis is structured as follows: Section 2 discusses related work in the field. Section 3 provides the necessary background knowledge. In Section 4, we elaborate on the characteristics of the SAGA pattern and present its variations. In Section 5, we introduce the piSaga library, and in Section 6, we present the experimental results. We conclude the thesis in section 7.

2. BACKGROUND

To understand distributed transactions, one should have a basic understanding of distributed systems, the CAP theorem, databases, concurrency control, the ACID properties, isolation issues in distributed systems, and measures to address isolation issues such as ACD and compensating transactions. These concepts are fundamental for understanding how distributed transactions work in a world of microservices and the trade-offs that must be made when designing a distributed system that supports distributed transactions.

2.1 CAP Theorem

The CAP theorem [1][2][3], also known as Brewer's theorem after computer scientist Eric Brewer, states that any distributed system can provide only a combination of two out of the following three guarantees:

- Consistency

  The “consistency” of the CAP theorem denotes that all clients simultaneously observe the same data, regardless of the node to which they connect. For instance, to be successful, a write operation to one node must also be replicated or sent to all other nodes in the system simultaneously.

- Availability

  Any client requesting data will receive a response as long as the CAP theorem holds, even if one or more nodes are unavailable. So, all active nodes in a distributed system always respond appropriately to every request.

- Partition tolerance

  In the presence of partition tolerance, the cluster will continue to operate despite any number of communication failures between nodes.

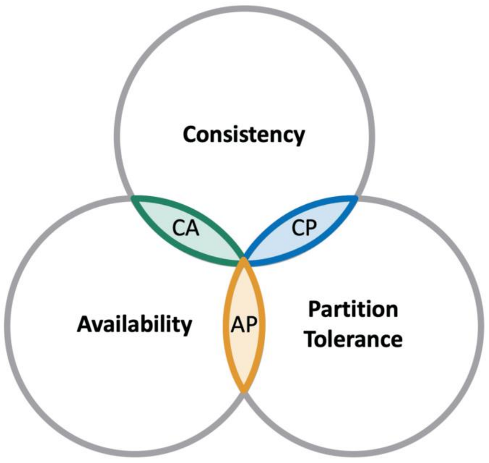

Figure 2-1 Graphical representation of the CAP theorem

So, our system could be classified as the following [3]:

- CP (Consistency – Partition Tolerance)

  A CP system offers consistency as well as partition tolerance. However, this comes at the sacrifice of availability. When two nodes partition, the system must shut down the inconsistent node until the partition is resolved.

- AP (Availability – Partition Tolerance)

  An AP system's availability and partition tolerance come at the cost of data consistency. All nodes remain available throughout a partition; however, those in the incorrect state may retrieve outdated data.

- CA (Consistency - Availability)

  The CA systems are consistent and available between nodes. As a result, it cannot provide fault tolerance if there is a partition between any two system nodes.

2.2 ACID

The acronym ACID describes the essential characteristics of database transactions that lead to a trustworthy method for preserving the integrity of our data. This acronym stands for atomicity, consistency, isolation, and durability. It guarantees that each read, write, or modification of a table is aligned with the following properties [4][5][7]:

- Atomicity

  Ensures that all of the attempted operations inside a transaction either succeed or fail. If any of the adjustments we attempt to make are unsuccessful for whatever reason, the entire process will be canceled, and it will be as if no changes were ever made.

- Consistency

  The database is always left in a valid, consistent condition after any modifications are made to it.

- Isolation

  Permits concurrent operation of numerous transactions without interference. This is accomplished by guaranteeing that any state modifications performed during a transaction are not visible to subsequent transactions.

- Durability

  Guarantees that its associated data will remain safe after a transaction has been processed, even if the underlying system crashes.

2.3 ACD

Through its isolation attribute, ACID transactions guarantee that the results of running several transactions simultaneously are equivalent to doing them in a particular order [8]. The database gives the impression that each ACID transaction has exclusive data access. Isolation makes it much easier to write business logic that runs concurrently.

On the other hand, the ACD properties provided by distributed transactions ensure that data is always consistent and that changes are permanent, making it an essential tool for maintaining data integrity in a distributed environment. However, multiple transactions could interfere with one another. Thus, the lack of isolation can cause a number of problems that can compromise the integrity and consistency of data.

An example of ACD properties in action would be a distributed transaction that updates multiple databases across different systems. The transaction ensures that all updates are treated as a single, indivisible unit of work and that the data remains consistent across all systems. If any part of the transaction fails, the entire transaction is rolled back, and the data remains in a consistent state.

2.3.1 Lack of Isolation

The lack of isolation can generate anomalies [8] since it is a critical attribute in distributed transactions. Isolation guarantees that numerous transactions are conducted separately, without interfering with one another, because it helps preserve data consistency and avoid conflicts and deadlocks. Without isolation, distributed transactions can result in a variety of issues that jeopardize data consistency and integrity.

Inconsistent data is one of the primary challenges produced by the lack of isolation. Numerous transactions can access the same data without isolation, resulting in inconsistencies. For instance, if two transactions update the same record simultaneously, the record's final value may be unexpected, leading to data discrepancies. This can lead to system errors and the production of inaccurate results.

Deadlocks are another issue created by the lack of isolation. Without isolation, several transactions may end up in a deadlock state, where one transaction is awaiting the completion of the other before continuing. This may cause the machine to become inactive and unresponsive.

Concurrent access is an additional problem induced by the lack of isolation. Multiple transactions may access the same data concurrently in the absence of isolation, resulting in Concurrent Access concerns such as race conditions, lost updates, and dirty reads.

So, the isolation anomalies can be categorized as follows:

- Lost updates

  A lost update anomaly happens when one SAGA completely overwrites an update made by another.

- Dirty reads

  A dirty read happens when one SAGA accesses data that another SAGA is currently updating.

- Fuzzy reads

  Two separate phases of a SAGA read the same data, but each one provides different outcomes because another SAGA has made modifications.

2.3.2 Countermeasures

A set of countermeasures exists for addressing anomalies brought on by the lack of isolation, intending to prevent anomalies or lessen their impact [8][9].

- Semantic lock

  The compensable transaction sets a flag in every record it creates or edits. The flag shows that the record has not yet been saved so that it could change. The flag can either be a lock that stops other transactions from accessing the record or a warning that other transactions should be careful with that record. It is resolved either with a retriable transaction, which indicates that it is successfully finishing or through a compensatory transaction, which indicates that it is rolling back.

- Commutative updates

  Commutativity refers to the ability of a set of operations to be performed in any given order. This countermeasure is effective since it prevents lost updates.

- Pessimistic view

  The pessimistic view countermeasure reorders the steps of a SAGA to minimize business risk due to a dirty read.

- Reread value

  The countermeasure for rereading values prevents lost updates. Using this countermeasure, a transaction participant rereads a record before updating it, confirms that it is unchanged, and then updates it. If the record has changed, the transaction terminates and possibly starts over.

- Version file

  The version file countermeasure keeps track of the actions done on a record in order to reorganize them.

- By value

  It is a method for selecting concurrent techniques based on the level of business risk. The attributes of each request are evaluated by an application that implements this countermeasure in order to determine whether or not to employ sagas or distributed transactions.

2.4 Microservices

Microservices is a trendy architectural style that has grown significantly in the past few years [7]. "Microservices Architecture" has emerged to represent a specific approach to structuring software programs as collections of independently deployable services. While this architectural style has no specific definition, it has similar traits around business capability, automated deployment, intelligence in the endpoints, and decentralized management of languages and data [11].

In a nutshell, the microservice architectural style is a method for building a single application as a collection of independent services that share only a common set of data, run in their own processes, and communicate with one another using lightweight mechanisms, like RESTful APIs [11]. These services are based on business capabilities and may be independently deployed using a fully automated deployment strategy. These services have minimal centralized administration, which may be implemented in different programming languages and utilize various data storage systems.

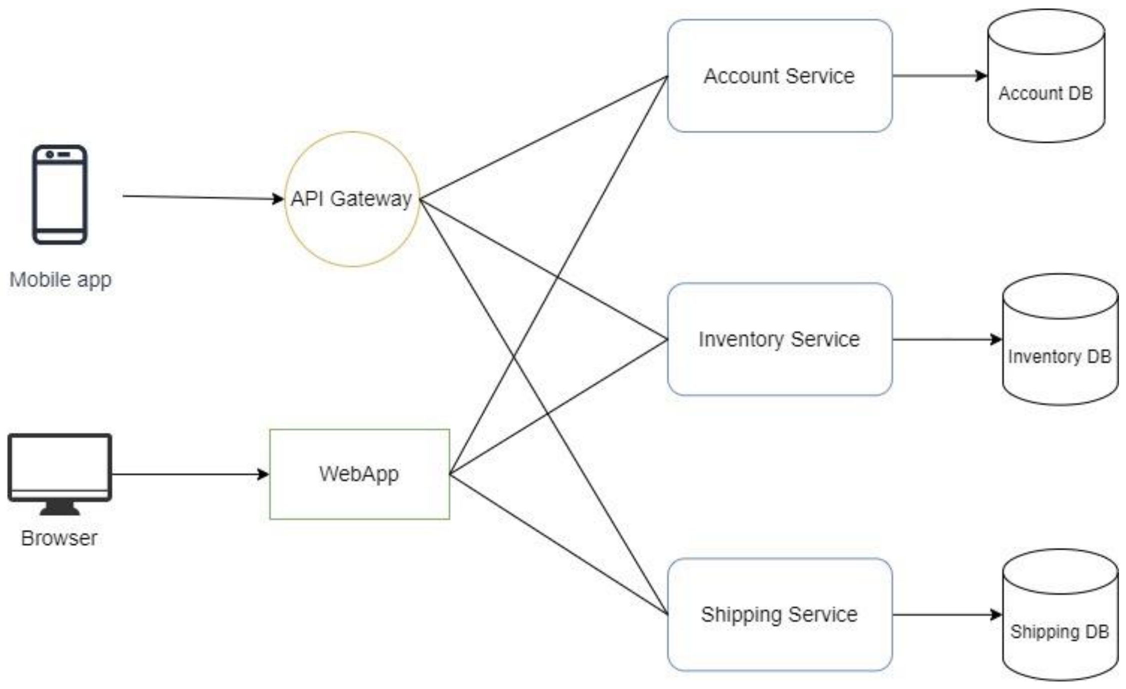

Figure 2-2 Microservices architecture high-level design

Initially, microservices were hosted in physical computers, but now they have shifted to virtual machines and containers [10][12]. Separating the services this much makes it easy to solve a common problem in architectures that host many kinds of applications on the same infrastructure. For example, using an application server to manage multiple running applications lets us reuse network bandwidth, memory, and disk space, among other things. However, if all the apps that can use the shared infrastructure continue to expand, there will be a shortage of some kind.

Separating each service into its own procedure eliminates any issues caused by sharing. Before the gradual growth of publicly accessible open-source operating systems and the development of automated machine provisioning, it was impracticable for each domain to have its infrastructure. However, now more than ever, thanks to cloud computing and containerization, we can benefit from extreme decoupling at the domain and operational levels.

On the other hand, microservices can suffer from poor performance because of their decentralized architecture. Network calls are significantly slower than method calls, and security verification at each endpoint adds additional processing time.

2.5 SAGA compared to other distributed transactions protocols

Distributed transactions are a way to ensure the atomicity of a series of operations that are carried out across multiple nodes in a distributed system. There have been numerous approaches to implementing distributed transactions in the field of distributed systems. Below, we list some of the most well-known and compare them with the SAGA pattern.

2.5.1 Two-Phase commit (2PC)

Two-Phase commit (2PC) [14][19] is a protocol used to ensure the atomicity of a distributed transaction, which involves multiple participants that may be distributed across different systems or locations. In 2PC, a coordinator coordinates the commit or rollback of a transaction across all participants. The coordinator sends a prepare request to each participant, deciding whether it is ready to commit. If all participants are ready, the coordinator sends a commit request to all participants, causing them to commit the transaction. If any participant is not ready, the coordinator sends a rollback request to all participants, causing them to roll back the transaction.

2PC and SAGA are designed to serve the same purpose, and they are both distributed transaction models used to ensure the consistency and reliability of transactions across multiple resources. However, they have a few key differences.

One of the main differences between 2PC and SAGA is how they handle communication between participants in a transaction. 2PC is a synchronous model, while SAGA is an asynchronous model. In 2PC, all participants in the transaction must communicate with the coordinator before the transaction can be committed, and all participants must be available and responsive during the commit phase. In contrast, SAGA allows participants to commit their changes independently and reconcile conflicts later. This means that SAGA is more flexible in handling errors and failures as it allows participants to fail or be unavailable during the transaction.

Another difference between 2PC and SAGA is how they handle a transaction's outcome. In 2PC, the transaction is either committed or rolled back as a whole. The transaction is rolled back if a participant fails during the commit phase. In contrast, in SAGA, a transaction can be partially committed. This allows for more flexibility in handling errors and failures as it allows for compensating actions to be taken for a participant that failed during a transaction.

2.5.2 Two-Phase Commit with Recovery (2PC*)

Two-Phase Commit with Recovery (2PC*) [20] is an extension of the classic 2PC protocol that adds support for recovery from failures. It is designed to address the problem of indeterminate outcomes in 2PC, which can occur when the coordinator fails before sending a commit or rollback request to the participants. In 2PC*, each participant maintains a log of the prepare and commit/rollback requests it has received from the coordinator. If a participant does not receive a commit or rollback request from the coordinator, it can use its log to determine the appropriate action based on the prepare requests it has received. 2PC* provides more robust atomicity guarantees than 2PC because it allows recovery from coordinator failures. However, it requires additional storage and processing resources to maintain the log of requests, which may impact performance. It is also more complex to implement than 2PC.

Distributed transaction models like 2PC* and SAGA are used to guarantee the integrity of transactions over a network of computers and other devices. When comparing 2PC* to SAGA, it is essential to keep in mind that 2PC* is an extension of the classic 2PC concept, whereas SAGA is a whole new approach. With the addition of recovery methods, 2PC* can deal with errors during the commit phase. Because of this, errors may be handled synchronously in 2PC*. However, SAGA provides a more adaptable method for addressing errors and failures by permitting compensatory transactions. Thus, SAGA is built with asynchronous fault tolerance in mind.

Another difference is that in 2PC*, the entire transaction is committed or rolled back. However, with SAGA, it is possible to commit a transaction partly.

2.5.3 Try, Commit, and Cancel (TCC)

Try, Commit, and Cancel (TCC) [21] is a protocol used to implement distributed transactions in a way that is more efficient than traditional 2PC protocols. It is also known as “Optimistic Two-Phase Commit” or “Single-Phase Commit with Compensations”. In the TCC protocol, a coordinator sends each participant a “Try” request, indicating that it would like to commit a transaction. If all participants are able to commit, the coordinator sends a commit request to all participants, causing them to commit the transaction. If any participant cannot commit, the coordinator cancels the transaction by sending a cancel request to all participants. The TCC protocol has several benefits compared to 2PC. It is faster because it does not require a separate prepare phase, and it is more resilient to failures because it does not require the coordinator to send a commit or rollback request to each participant. However, it is less widely supported than 2PC and may not provide the same level of guarantee of atomicity in all cases.

TCC and SAGA differ significantly in how they handle the various stages of a transaction. Unlike SAGA, which consists of several phases, TCC only has three. The TCC model divides the transaction into three steps: Try, Commit, and Cancel. On the other hand, SAGA divides the transaction into many stages, each of which may include several sub-steps depending on the nature of the transaction and the requirement for compensating actions.

How successful and unsuccessful transactions are handled is another area in which TCC and SAGA diverge. With TCC, we can only commit or cancel the transaction as a whole. In contrast, SAGA allows only a portion of a transaction to be committed simultaneously. As a result, there is more flexibility for dealing with errors and failures, as compensatory steps can be performed for a failed participant in a transaction.

2.5.4 Generalized Recursive Idealized Transactions (GRIT)

Generalized Recursive Idealized Transactions (GRIT) [22] is a protocol for implementing distributed transactions that are based on the idea of "idealized transactions", which are transactions that are free from conflicts and can be executed in any order. GRIT allows for the efficient execution of transactions that can be decomposed into smaller transactions that can be executed concurrently. In GRIT, a coordinator coordinates the execution of a transaction by dividing it into smaller transactions and sending each of these transactions to a different participant to be executed. The coordinator then waits for a response from each participant indicating whether the transaction was successful or not. If all participants respond successfully, the coordinator sends a commit request to all participants, causing them to commit the transaction. If any participant responds with a failure, the coordinator sends a rollback request to all participants, causing them to roll back the transaction. GRIT is a highly efficient protocol that allows for the concurrent execution of transactions, which can significantly improve the performance of distributed systems. Therefore, it requires that transactions be decomposed into smaller transactions that can be executed concurrently, which may not always be possible.

One of the commonalities between GRIT and SAGA is that they both manage distributed transactions via nested transactions. However, GRIT and SAGA do not have many similarities. In contrast to GRIT's emphasis on short-lived transactions that can be made of numerous sub-transactions, SAGA prioritizes long-running transactions composed of multiple sub-transactions. Thus, SAGA is better suited for complex, long-running business processes, while GRIT is better suited for simple, transient transactions.

Another distinction between GRIT and SAGA is their consistency policies. While GRIT relies on a consensus mechanism, SAGA employs a state machine replication strategy to guarantee data integrity. This implies that while GRIT uses a consensus mechanism to verify that all nodes in the system agree on the state of a transaction, SAGA relies on replicating the state of a transaction across various nodes in the system.

Furthermore, unlike GRIT, SAGA permits compensatory transactions, which may be used to reverse the effects of prior transactions. If a long-running transaction fails in SAGA, the database may be restored to an earlier point, which is impossible in GRIT.

3. RELATED WORK

Our community has been trying to find viable solutions for the distributed transactions problem for a long time. Thus, many implementations aim to improve this space. This section presents piSaga and a few well-known libraries related to the SAGA pattern and orchestration in general.

Implementing the MicroProfile Long Running Actions (LRA) in an application provides several benefits, including simplified management of long-running tasks and better control over transactional behavior [28][33]. This feature ensures that long-running operations can be handled effectively and that the system remains responsive during the operation. The microservice architecture allows the application to scale horizontally by breaking down tasks into smaller chunks. Moreover, it reduces the chance of data inconsistencies and failures in long-running operations. However, there are some potential drawbacks to consider when implementing the MicroProfile LRA, including the increased complexity of implementing a microservice architecture, which requires a good understanding of the architecture and design of the application. Additionally, the distributed nature of microservices may require additional infrastructure and operational considerations, such as service discovery, load balancing, and failover.

The Saga pattern implementation in Apache Camel is a practical, open-source approach for managing distributed transactions in a reliable and scalable manner [30][31]. The advantages of using Apache Camel include improved reliability, flexibility, scalability, and ease of implementation. However, this approach also has some potential disadvantages, including increased complexity in the code, limited rollback capabilities, potential performance impact, and difficulty debugging. While the Apache Camel Saga pattern implementation can ensure data consistency and handle large numbers of transactions and messages, its impact on system performance and complexity should also be considered.

The primary goal of implementing the Eventuate Tram SAGA framework is to provide a distributed, scalable, and reliable solution for managing long-running transactions across microservices [29][8]. The framework offers a declarative programming approach that simplifies handling distributed transactions, decreasing the chance of data inconsistencies and allowing developers to add new business processes or alter existing ones swiftly. The framework's benefits include better fault tolerance, scalability, and maintainability of microservice-based systems. Additionally, the framework's use of the event sourcing pattern enables the tracing of events and provides a reliable audit trail of changes to the system. Yet, there are possible drawbacks to consider while utilizing the Eventuate Tram SAGA framework, such as the requirement for a robust infrastructure to manage events and transactions, the complexity of integrating existing services with the framework, and the steep learning curve for developers.

The Axon Framework SAGA implementation gives several benefits, including enhanced transaction management stability, scalability, and maintainability [28][32]. The framework makes it easier to manage distributed transactions by giving developers a declarative programming approach that does not need them to write complicated code. This method lessens the possibility of data discrepancies and allows developers to implement new business processes or alter current ones quickly and simply. The Axon Framework also offers event sourcing and command handling capabilities to guarantee the system's scalability and resilience. Nevertheless, there are some negatives to consider while using the Axon Framework SAGA, including the necessity for developers to learn a new programming paradigm and framework, which can raise the learning curve. Additionally, because the framework is based on CQRS and event sourcing, extra operational and technical concerns, such as database architecture, versioning, and event storage, may be required.

Despite the fact that piSaga is a simple implementation of the SAGA pattern, it can accomplish its objectives successfully. piSaga enables developers to design their implementations based on the state-machine paradigm. This state machine is composed of a set of states and events that move the machine from one state to another. Moreover, piSaga is a handy solution for handling distributed transactions in a reliable and scalable manner. Using piSaga can result in increased reliability, flexibility, and scalability. In addition, there is a slight learning curve because piSaga provides only the necessary contracts to develop a SAGA. However, piSaga is not a divine solution for distributed transactions; there are a few potential drawbacks to take into account. First, it is not a battle-tested solution. Second, the IPC of piSaga depends on Apache Kafka, at least for now. Next, it has limited integration with some enterprise systems since it focuses on Spring Boot.

Last but not least, piSaga is an open-source library whose purpose is to contribute to the open-source community and be a small but meaningful contribution to computer science and distributed systems.

4. TRANSACTIONAL SAGA

The notion of a SAGA in architecture predates microservices, as it was first concerned with restricting the scope of database locks in the early distributed systems [14][15][17]. The SAGA pattern for microservices is characterized as a succession of local transactions in which each update triggers the next update in the sequence by publishing an event. If any of these updates fail, the SAGA will issue a series of compensatory updates by erasing the previous alterations made throughout the workflow. Thus, the SAGA pattern should leverage coordination, consistency, and inter-process communication (IPC) to perform optimally.

Coordination refers to the process of managing access to shared resources by multiple processes, ensuring that they are used in a consistent and synchronized manner. Consistency, on the other hand, refers to the property of the system that ensures that all processes have a consistent view of the shared state. Inter-process communication is the mechanism through which processes coordinate and share information with each other. To understand the SAGA pattern, it is essential to have a solid understanding of these concepts and how they relate to distributed systems.

This section of the thesis will delve deeper into these concepts and explore how the SAGA pattern incorporates them to achieve scalability and fault tolerance in distributed systems. Additionally, we will elaborate on the eight variations of the SAGA pattern.

4.1 Coordination

The implementation of a SAGA comprises logic that organizes the SAGA's stages. When a system command initiates a SAGA, the coordination logic must identify the first SAGA participant and instruct it to conduct a local transaction. As soon as this transaction concludes, the SAGA's sequencing coordination identifies and activates the subsequent SAGA participant. This procedure continues till each phase of the SAGA has been carried out. If a local transaction fails, the SAGA must perform compensatory operations in reverse order. There are two types of coordination, choreography and orchestration.

4.1.1 Choreography

Using choreography is one method for implementing a SAGA [10]. There is no mediator in charge of informing every service in a SAGA on how to continue the workflow. The SAGA participants subscribe to each other's events and respond accordingly.

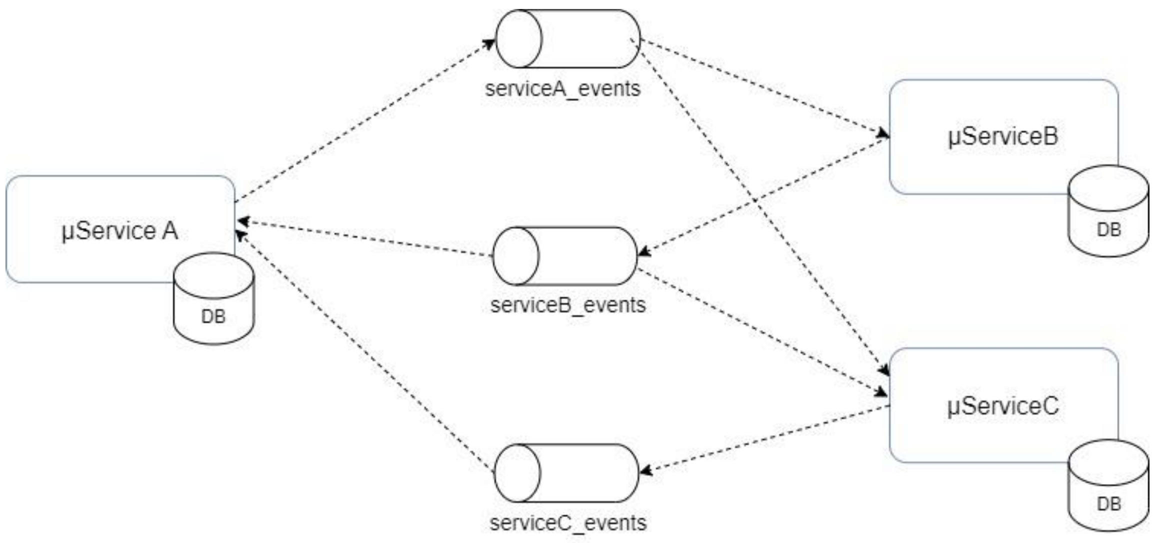

Figure 4-1 Choreography coordination

When building choreography-based SAGAs, there are several interservice communication-related factors to consider. The initial concern is ensuring that a SAGA participant publishes an event and updates its database as part of a database transaction. Each step of a choreography-based SAGA updates the database and publishes an event. The database update and event publication must occur simultaneously. Therefore, for reliable communication, SAGA participants must employ transactional messaging. The second thing to think about is ensuring that each participant in the SAGA can map each event it gets to the data it has on its own end. A SAGA participant must publish events with a correlation id assigned to them. This correlation id will help the rest of the SAGA participants to access the correct data.

Choreography has both benefits and drawbacks. Its benefits are “simplicity” and “loose coupling”. The services broadcast events when creating, updating, or deleting business objects. Moreover, the SAGA participants subscribe to events without direct knowledge of one another.

On the other hand, it is not easy to understand where choreography has been used. It spreads the execution of the SAGA among the several services. In addition, there might be circular dependencies between the services because the SAGA participants would subscribe to each other's events. Also, due to the fact that each service should subscribe to every other event, we could result in a tightly coupled system.

4.1.2 Orchestration

SAGAs may also be implemented using orchestration [10]. When utilizing orchestration, we must establish a class whose primary function is to orchestrate the SAGA participants on what actions to perform. The SAGA orchestrator connects with the participants utilizing a command/asynchronous-response interaction approach. In order to execute a SAGA step, it sends a command message to a participant specifying the operation to be performed. After completing the operation, the participant in the SAGA sends a response message to the orchestrator. The orchestrator then analyses the message and decides which SAGA step to execute next.

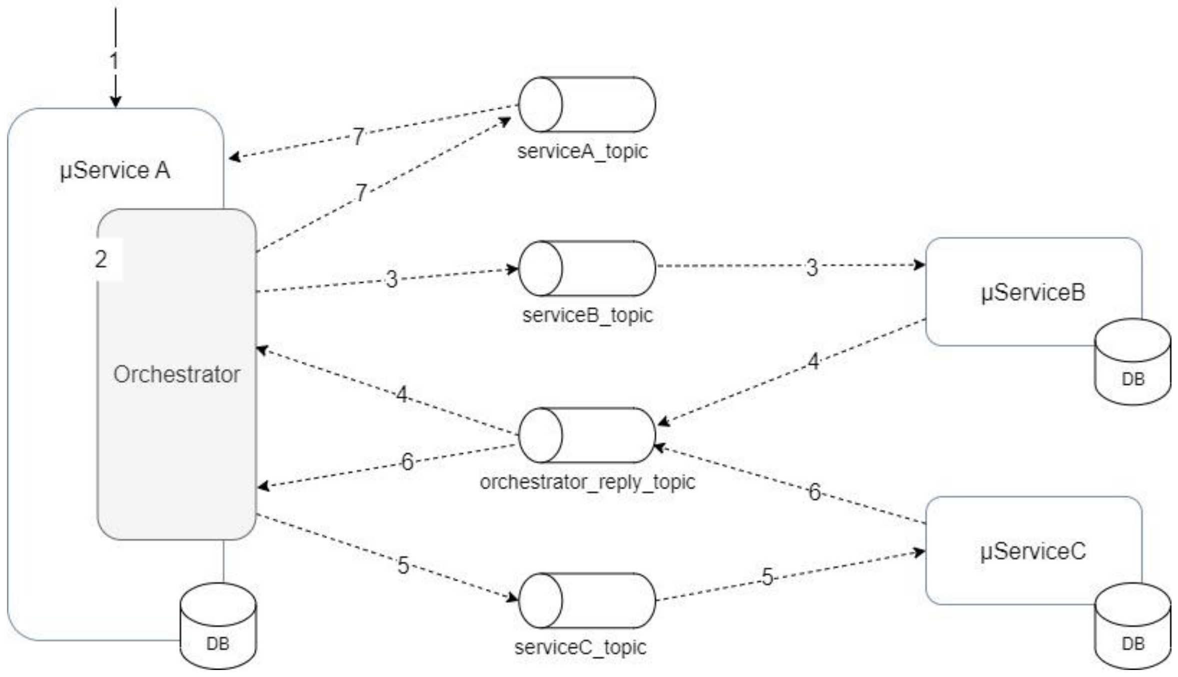

Figure 4-2 Orchestration coordination

A service updates a database and publishes a message at each stage of an orchestration-based saga. Hence, it must utilize transactional messaging to update the database atomically and publish messages.

Orchestration is not the silver bullet. It has both strengths and weaknesses. First, the orchestration has the advantage of not introducing circular dependencies. The orchestrator invokes the participants in the workflow, but the participants do not invoke the orchestrator. Therefore, there are no circular dependencies because the orchestrator depends on the participants but not vice versa. Second, it favors loose coupling since each service has its API that the orchestrator calls; the orchestrator need not be aware of the events published by the SAGA's participants. Last but not least, it enhances the separation of concerns and simplifies business logic. The SAGA coordination logic is centralized within the SAGA orchestrator. The domain objects are less complex and are unaware of the sagas in which they engage.

While orchestration has many benefits, there is also a potential downside in centralizing too much business logic in the orchestrator. The resulting architecture has the intelligent orchestrator instructing the naive services on how to carry out their tasks. The critical point to avoid this situation is to develop orchestrators that are simply responsible for managing the request workflow and do not contain any other business logic.

4.2 Inter-process Communication

Microservices are required to connect through an inter-process communication (IPC) mechanism [16]. So, when defining how the services will connect, we need to consider a number of challenges, including how services interact with one another, how to declare the API for each service, how APIs might grow, and how to deal with partial failure. Microservices can employ two types of IPC mechanisms: synchronous and asynchronous communication.

4.2.1 Synchronous communication

Using synchronous communication [7][16], a service sends a request of some form to a downstream process (usually to another microservice) and blocks until the call completes and probably until a response is received.

A synchronous blocking call often awaits a response from a downstream operation. This might be because the call's outcome is required for subsequent action or because the system wants to ensure the call succeeded and retried if necessary.

Consequently, synchronous calls make a system more vulnerable to cascading problems caused by downstream failures than the usage of asynchronous calls.

4.2.2 Asynchronous communication

When using asynchronous communication [7][16], the microservice that is issuing a call will not become blocked while the call is being sent out across the network. It can proceed with any additional processing without waiting for a response. There is a wide variety of nonblocking asynchronous communication protocols, but the most well-known is communication through message queues and event streams.

Nonblocking asynchronous communication allows for the temporary decoupling of the service making the initial call and the service(s) receiving the call. It is not necessary for the services that receive the call to be reachable at the exact same moment that the call is being made. This communication style is also beneficial if the functionality is triggered by a call that will take a long time to process.

4.3 Consistency

When we talk about consistency, we refer to the transactional integrity level that must be maintained throughout all communication calls. There are two types of consistency the atomic and the eventual.

4.3.1 Atomic consistency

All the steps of a transaction must be completed successfully in order for the transaction to be considered atomic [13]. All previously performed steps must be reversed if even a single step fails. For this reason, an operation is said to be atomic if it either takes place in its entirety or does not take place at all.

One of the most significant issues with atomicity in microservices is that a single transaction may involve several independent tasks being completed by separate microservices. Thus, a method is needed to undo previously completed transactions if a local transaction fails.

4.3.2 Eventual consistency

Eventual consistency [6][25] is a model used in distributed computing to ensure high availability. This model provides an informal promise that if no additional modifications are made to a particular data item, all accesses to that item will ultimately return the most recently modified value.

If there are no system failures, the maximum size of the inconsistency window may be calculated by considering variables such as communication delays and the number of copies participating in the replication scheme. When a system has reached eventual consistency, it is commonly referred to as having converged or having achieved replica convergence.

4.4 Transaction types

A functional SAGA structure paradigm is proposed in [9]. In this paradigm, a SAGA is composed of the following three types of transactions:

- Compensable transactions

  Transactions that may be reversed via a compensating transaction.

- Pivot transaction

  A successful commit to the pivot transaction will cause the SAGA to continue till its completion. A pivot transaction is a transaction that is neither compensable nor retriable.

- Retriable transactions

  Transactions that are guaranteed to succeed after the pivot transaction.

4.5 State machine modeling

A state machine is a powerful and intuitive modeling tool for understanding the inner workings of a SAGA orchestrator. At its core, a state machine comprises a collection of states and events that allow the machine to transition from one state to another. These transitions can be accompanied by an action, which in the context of a SAGA typically involves the invocation of a SAGA participant.

One key aspect of a state machine in the context of a SAGA is how it handles local transactions executed by SAGA participants. When a local transaction is completed, it can trigger a transition between states. The current state of the machine and the outcome of the local transaction determines the specific transition and corresponding action.

Using a state machine paradigm to model a SAGA makes it significantly easier to design, construct, and test the orchestrator. The clear and visual representation of states and transitions allows developers to reason easily about the different paths a SAGA can take and the actions that are invoked at each step. Using a state machine also allows more flexibility to handle failures, debugging, and troubleshooting processes and become more efficient.

4.6 The variations of the SAGA pattern

Distributed transactions are a key component of many distributed systems since they coordinate operations across numerous nodes in a manner that guarantees the atomicity, consistency, and isolation of these processes. The SAGA pattern is a popular implementation strategy for distributed transactions in distributed systems. Since its introduction in 1987 by Hector Garcia-Molina and Kenneth Salem [14][15][17], it has been a cornerstone in distributed system design.

The SAGA pattern is based on the concept of long-running transactions that span numerous systems or locations, and that can be deconstructed into smaller, simultaneously executable transactions. It employs a process orchestration methodology in which a coordinator organizes the execution of a transaction by sending requests to various participants to conduct smaller transactions. The coordinator checks the status of these transactions and takes corrective action in the event of errors or failures.

One of the distinguishing characteristics of the SAGA pattern is the use of compensating transactions, which undo the consequences of prior operations and restore the system to a consistent state. This enables the SAGA pattern to tolerate faults and errors gracefully and to maintain the atomicity and consistency of a transaction, notwithstanding these obstacles.

There are several variations of the SAGA pattern. We can distinguish them based on their communication, consistency, and coordination approach. Table 4-1 lists the eight most well-known variations.

Table 4-1 Saga pattern variations

Name	Communication	Consistency	Coordination
SAO	Synchronous	Atomic	Orchestrated
SAC	Synchronous	Atomic	Choreographed
SEO	Synchronous	Eventual	Orchestrated
SEC	Synchronous	Eventual	Choreographed
AAO	Asynchronous	Atomic	Orchestrated
AAC	Asynchronous	Atomic	Choreographed
AEO	Asynchronous	Eventual	Orchestrated
AEC	Asynchronous	Eventual	Choreographed

4.6.1 SAO SAGA Pattern

The SAO SAGA pattern (SAO) is considered the "traditional" SAGA pattern. Because of how it works, it is also called an "Orchestrated SAGA". Its dimension correlations are illustrated in Figure 4-3.

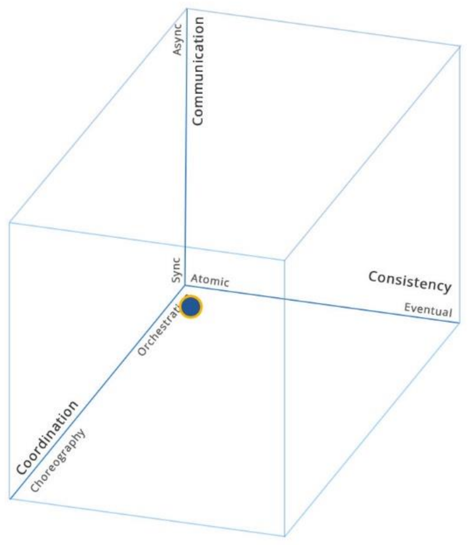

Figure 4-3 SAO utilizes Sync communication, Atomic consistency, Orchestrated coordination

Here, an orchestrator manages a workflow that involves updates for three services that are supposed to happen transactionally - either all three calls are successful or none are, Figure 4-4. If any of the calls fail, the prior state is restored. This coordination challenge may be solved in a number of different ways in distributed architectures. Such transactions are incompatible with many databases and are notorious for their frequent failure types.

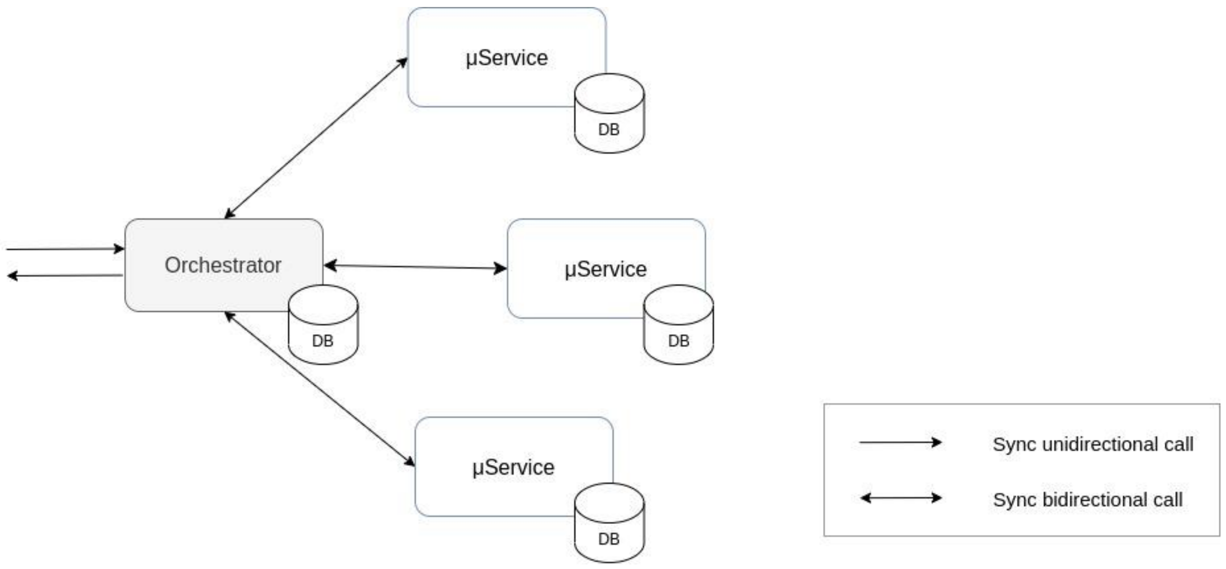

Figure 4-4 Communication between microservices in the SAO SAGA pattern

In a distributed transaction, a compensating transaction undoes a data write action carried out by another service, e.g., undoing an update, reinserting a previously deleted row, or deleting a previously inserted row. While compensating updates seek to undo changes to restore distributed data sources to their initial state before a distributed transaction begins, they are full of complex issues, obstacles, and trade-offs.

Unfortunately, the challenges are brought on by the error conditions. As shown in Figure 4-5, the orchestrator service is responsible for monitoring the success of calls and initiating compensating calls to other services in the event that one or more of the requests are unsuccessful.

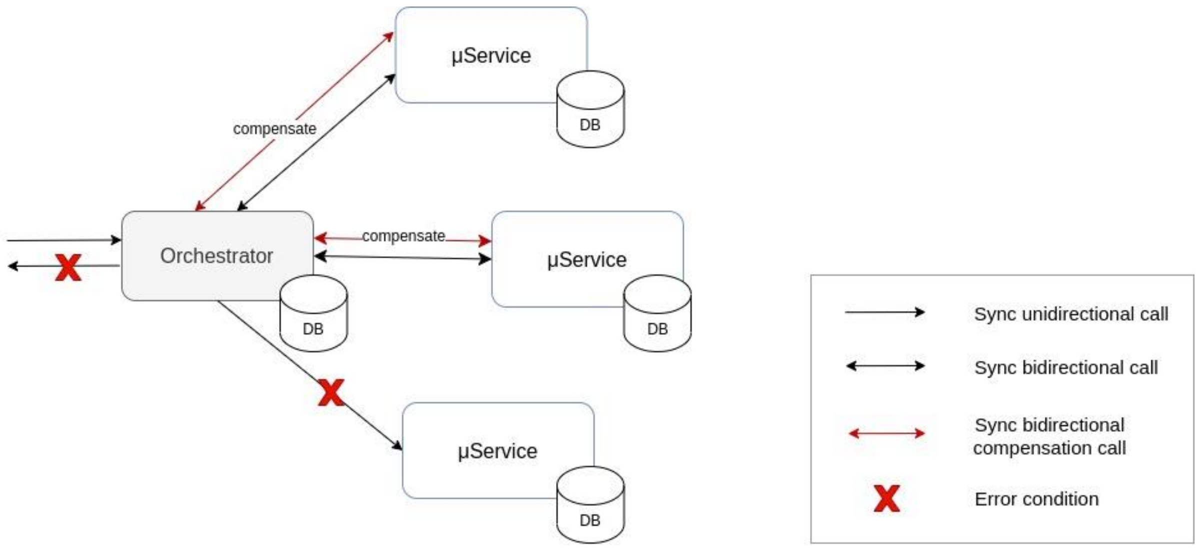

Figure 4-5 Compensatory transactions are issued by the orchestrator when an error occurs

The orchestrator service both handles the requests and manages the transaction workflow. As we can see, the calls on the first two services were successful, but the third call failed. To achieve atomic consistency, as specified by the SAO, the orchestrator must employ compensatory transactions to request that the other two services roll back the previously performed operation, restoring the system to its state before the transaction.

The SAO has a distinct advantage over competing solutions due to its monolithic-like transactional coordination. On the other hand, the drawbacks are numerous and diverse. First, the combination of orchestration with transactionality may affect operational architectural features like performance, scalability, and flexibility. The orchestrator must ensure that all participants in the transaction have succeeded or failed, causing time bottlenecks. Second, the different patterns used to create distributed transactionality, like compensating transactions, are susceptible to a broad range of failures and introduce intrinsic complexity through undo operations.

The following features distinguish the SAO SAGA pattern. This design demonstrates exceptionally high degrees of coupling across all available dimensions, including synchronous communication, atomic consistency, and orchestrated coordination. This is not unexpected, as it mirrors the communication behavior of strongly connected monolithic systems, but it poses various problems in distributed designs.

Added to the necessity for atomicity, error conditions and other heavy coordination add complexity to this design. Using synchronous calls reduces some of the complexity since developers do not have to worry about race situations and deadlocks during calls.

Orchestration impedes responsiveness, particularly when it must coordinate transactional atomicity. This style employs synchronous calls, which hinders performance and responsiveness further. This pattern will fail if any services are unavailable or an unrecoverable error occurs. Similar to responsiveness, the bottleneck, and coordination needed to implement this pattern make it difficult to address the system's scalability.

4.6.2 SAC SAGA Pattern

The main difference between the SAC SAGA pattern (SAC) and the SAO is that the SAC uses choreography instead of orchestration, Figure 4-6.

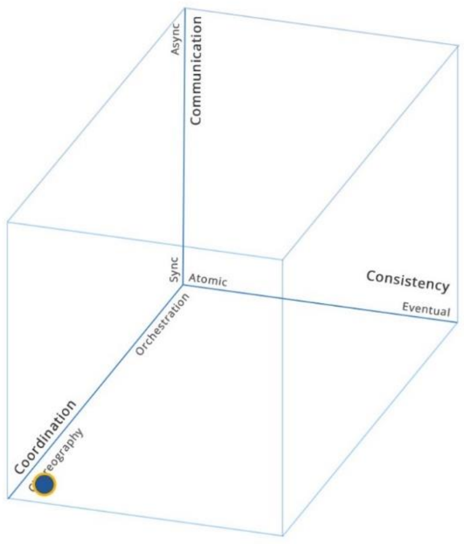

Figure 4-6 SAC utilizes Sync communication, Atomic consistency, Choreographed coordination

Hence, the structural communication illustrated in Figure 4-7 is undergoing the equivalent modification.

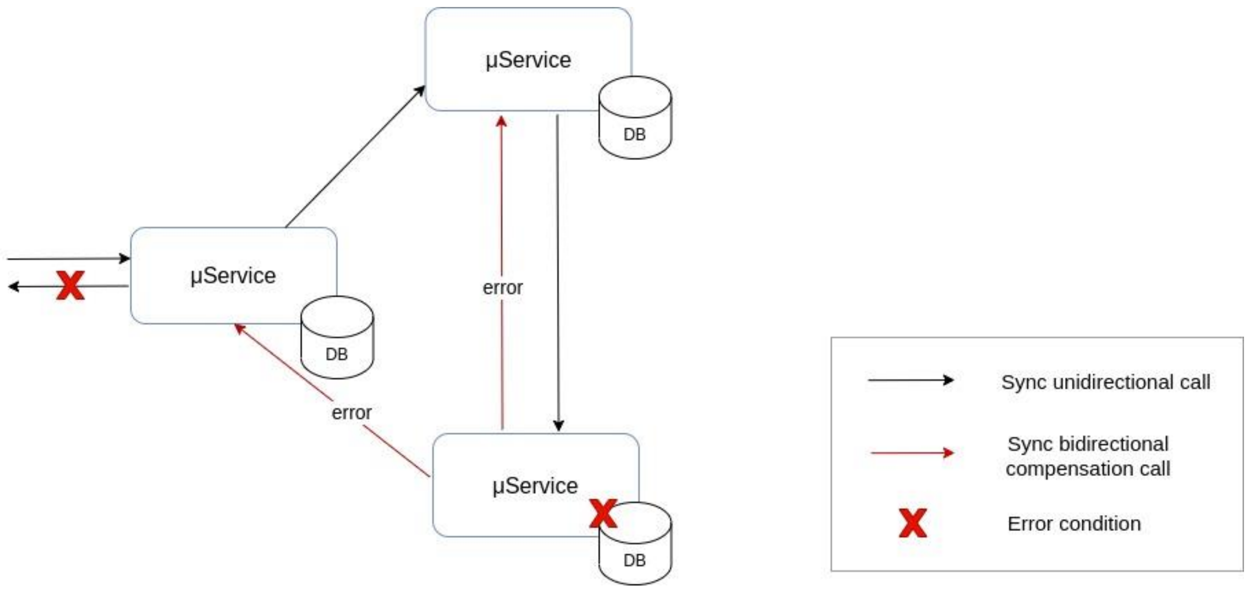

Figure 4-7 Each microservice is responsible to issue its own compensatory transactions

The SAC combines atomicity with choreography with no explicit orchestrator. However, atomicity requires some cooperation. In Figure 4-7, the service that was first called becomes the coordination point, also known as the front controller. It then forwards the request to the subsequent service in the process, which continues until the workflow completes. However, if an error occurs, each service must have business logic to support compensating requests.

Due to the importance of coordinating atomicity in transactions, this logic must be located someplace in the architecture. As a result, domain services need to have more logic about the workflow environment in which they operate, such as error handling and routing. The front controller in this design becomes as complicated as most mediators when used to more advanced processes, which lessens the pattern's usefulness and popularity. Therefore, this design pattern is frequently employed for low-complexity operations that demand increased scalability at the expense of some speed.

Even with synchronous communication, using choreography reduces bottlenecks under non-error situations. The final service in the process can return the result, allowing for more throughput and fewer bottlenecks. Due to a lack of coordination, the performance can be increased compared to the SAO. However, resolving error conditions will be significantly slower in the absence of an orchestrator since each service must unwind the call chain, which increases the coupling between services.

A positive aspect of non-orchestrated designs is the absence of a coupling singularity or a single point to which the workflow couples. In general, coupling reduction promotes scalability. As the scalability increases without orchestration, the domain services become more complicated to handle the workflow issues in addition to their primary function.

The SAC SAGA pattern has a relatively unique mix of characteristics. Typically, choreography is correlated with asynchronicity. In certain instances, synchronous calls eliminate race situations by ensuring that each domain service completes its part of the workflow prior to invoking the next.

4.6.3 SEO SAGA Pattern

The SEO SAGA pattern (SEO) utilizes synchronous communication, eventual consistency, and orchestration, as shown in Figure 4-8.

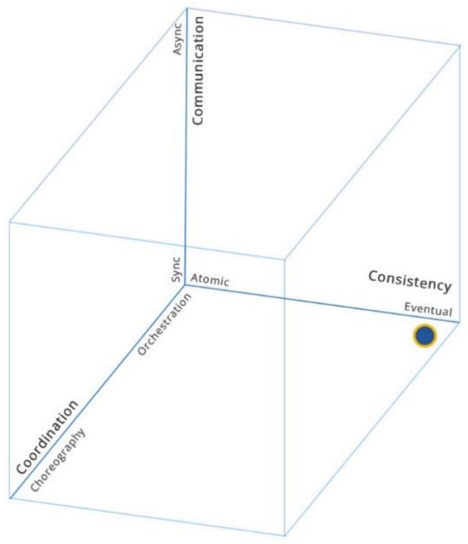

Figure 4-8 SEO utilizes Sync communication, Eventual consistency, Orchestrated coordination

This structure of communication eases the strict atomic constraint, giving system designers a plethora of new opportunities. For instance, if a service is momentarily unavailable, eventual consistency permits data caching until the service is restored. Figure 4-9 depicts the design of the SEO SAGA pattern.

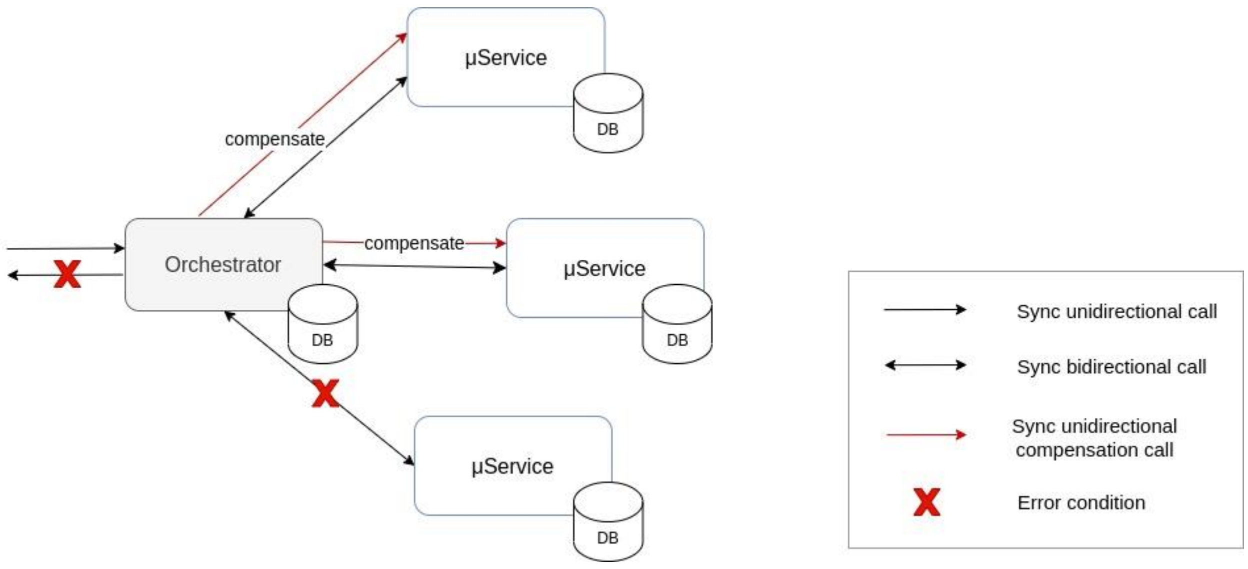

Figure 4-9 Compensatory transactions are issued by the orchestrator when an error occurs

In this paradigm, request, response, and handling of errors are all coordinated by an orchestrator. However, the orchestrator is not responsible for transaction management, which remains the responsibility of each domain service. Because of this, the orchestrator can handle compensation calls, even if they do not happen within a currently running transaction.

Having an orchestrator makes it simpler to manage processes. Synchronous communication is simpler than asynchronous, and eventual consistency eliminates the most challenging coordination obstacle, especially when dealing with errors. The absence of holistic transactions is the most enticing feature of SEO. Each domain service handles its transactional behavior, relying on eventual workflow consistency.

This pattern demonstrates a balanced set of trade-offs compared to many other patterns. The orchestrator must continue to handle complicated workflows but is not constrained to do so inside a transaction. SEO is not overly complex; it provides the most convenient possibilities with the fewest constraints. Even though the calls are synchronous, the mediator only needs to keep a less time-sensitive state regarding current transactions, which allows for better load balancing and hence greater responsiveness. In general, reducing transactional coupling will allow each service to expand more independently, leading to greater scalability.

4.6.4 SEC SAGA Pattern

The SEC SAGA pattern (SEC) is characterized by choreographed workflow, synchronous communication, and eventual consistency, Figure 4-10. This approach eliminates a central mediator by placing the workflow duties completely on the participating domain services, Figure 4-11.

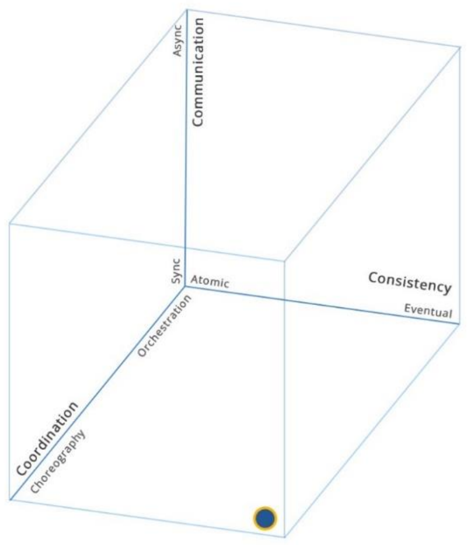

Figure 4-10 SEC utilizes Sync communication, Eventual consistency, Choreographed coordination

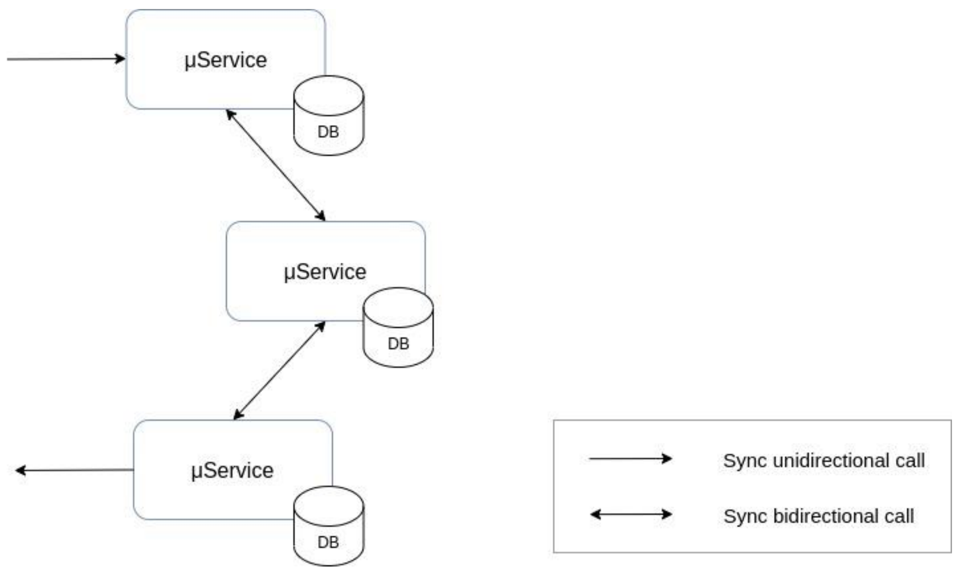

Figure 4-11 Communication between microservices in the SEC SAGA pattern

In this process, one service receives a request, processes it, and then passes it on to the next service. This architecture can implement the Chain of Responsibility design pattern [18] and the hexagonal architecture style [8], as well as any workflow with a sequence of one-way stages. Given that each service "owns" its own transactionality under this pattern, the workflow for the error conditions must be incorporated into the domain architecture. Due to the absence of built-in coordination via a mediator, there is a proportionate complexity link between process complexity and choreographed solutions—the more complicated the workflow, the more complex the choreography.

Workflows are simpler to represent using the SEC since no transactions are involved. Most workflow states and metadata must be stored in each domain service without an orchestrator. Hence, this pattern is best suited for simple processes. This pattern works exceptionally well for systems that benefit from high throughput. However, domain services must handle errors and coordination since no orchestrator exists.

In the SEC SAGA pattern, the coupling is moderate. This is because the reduced coupling caused by the absence of an orchestrator is compensated by the continued coupling of synchronous communication. The elimination of transactionality reduces the complexity of this pattern. This is well-suited to high-throughput, one-way communication systems, and its coupling level is well-suited to this architectural style.

4.6.5 AAO SAGA Pattern

The AAO SAGA pattern (AAO) utilizes atomic consistency, asynchronous communication, and orchestrated coordination, Figure 4-12.

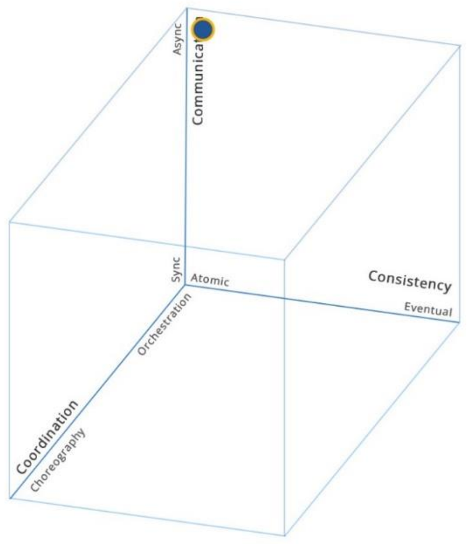

Figure 4-12 AAO utilizes Async communication, Atomic consistency, Orchestrated coordination

This pattern is similar to the SAO in all areas except communication. This design employs asynchronous communication rather than synchronous communication.

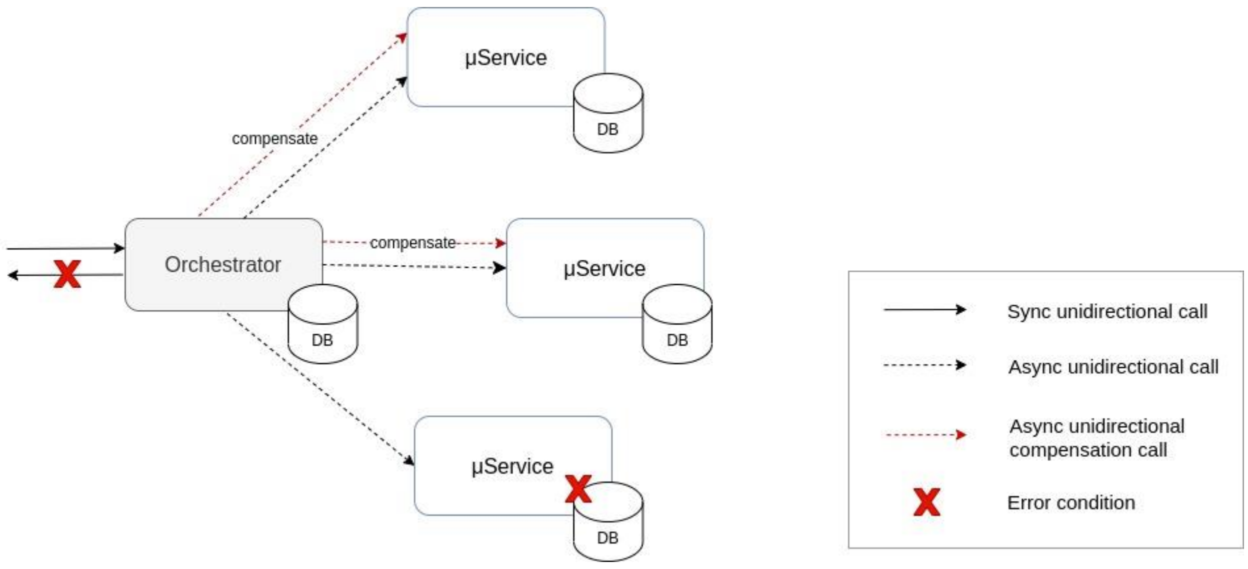

Figure 4-13 Compensatory transactions are issued by the orchestrator when an error occurs

However, asynchronicity is not a straightforward modification; it adds several levels of complexity to architecture, particularly in the area of coordination, necessitating an orchestrator with significantly more complexity. The orchestrator is responsible for monitoring the status of all ongoing transactions, Figure 4-13. Adding asynchronicity to orchestrated workflows introduces an asynchronous transactional state, which can lead to deadlocks, race situations, and other difficulties inherent in parallel systems since it eliminates serial assumptions about ordering.

In addition to design difficulty, there is also the debugging and operational complexity of dealing with large-scale asynchronous workflows. As a result of this pattern's effort at transactional coordination across calls, responsiveness will be negatively degraded overall and catastrophic if any of the services are unavailable.

4.6.6 AAC SAGA Pattern

The AAC SAGA pattern (AAC) is distinguished by its use of asynchronous communication, atomic consistency, and choreographed coordination, Figure 4-14.

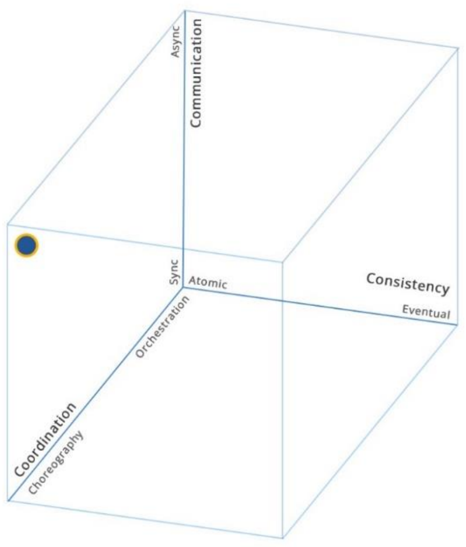

Figure 4-14 AAC utilizes Async communication, Atomic consistency, Choreographed coordination

It combines the strictest consistency-based atomic consistency with the two loosest coupling methods, asynchronous and choreography. Figure 4-15 shows the structural communication for this design.

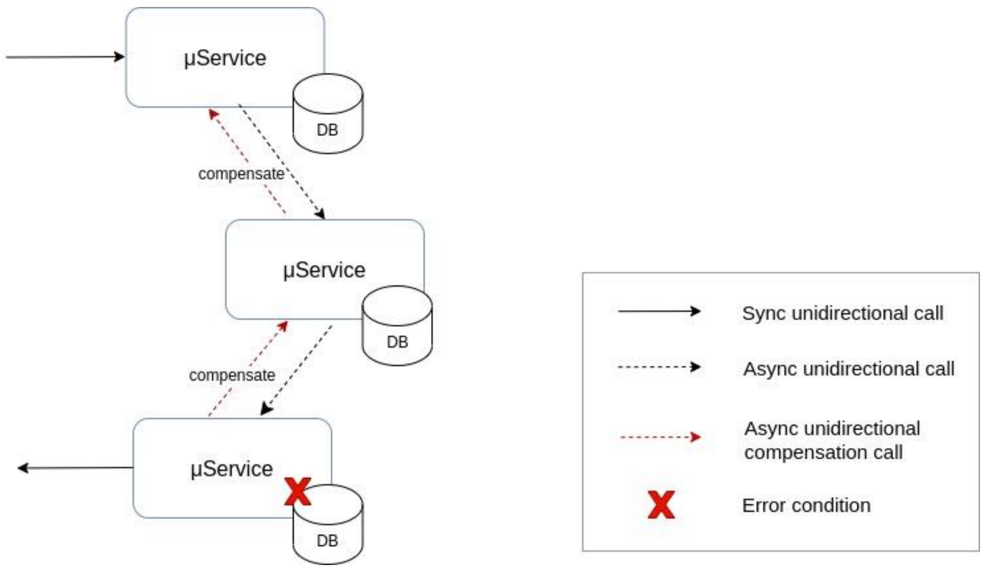

Figure 4-15 Each microservice is responsible to issue its own compensatory transactions

No intermediary is used in this pattern's asynchronous service-to-service communication model; hence transactions between different services might be inconsistent. Consequently, each domain service must keep track of undo information for many pending transactions, which may occur out of sequence due to asynchronicity, and coordinate under error conditions. Imagine that transaction T1 begins, and while it is waiting, transaction T2 begins. The choreographed services must reverse the sequence of execution, reversing each piece of the transaction along the way because one of the calls for the T1 transaction has failed.

This design tries to implement a transactional dependency, which is the worst sort of single coupling. However, it alleviates the other two by eliminating the need for a mediator and coupling, boosting synchronous communication. Additionally, its complexity is horrifying, since it demands the most rigorous criterion (transactionality) and the most challenging combination of other criteria to meet that requirement (asynchronicity and choreography). On the other hand, this design scales better than others with a mediator, and asynchronicity offers the capacity to do additional tasks concurrently.

4.6.7 AEO SAGA Pattern

The AEO SAGA pattern (AEO) has two significant changes compared to the SAO, making it easier to implement. It is based on asynchronous communication and eventual consistency. Figure 4-16 depicts the dimensional diagram of the AEO.

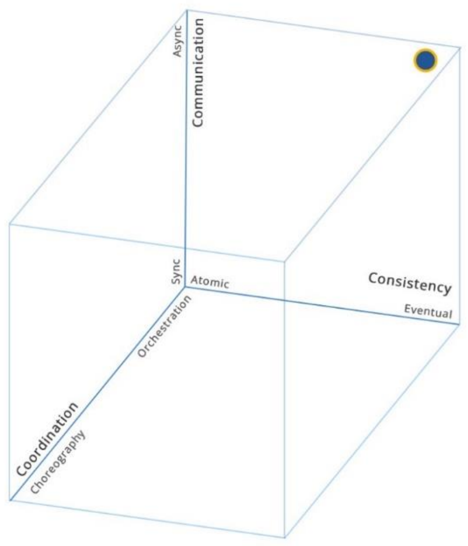

Figure 4-16 AEO utilizes Async communication, Eventual consistency, Orchestrated coordination

This pattern employs an orchestrator, which makes it appropriate for complicated procedures. However, it employs asynchronous communication, which enables improved responsiveness and parallel processing. The domain services are responsible for pattern consistency, as they may need to synchronize shared data in the background or via the orchestrator.

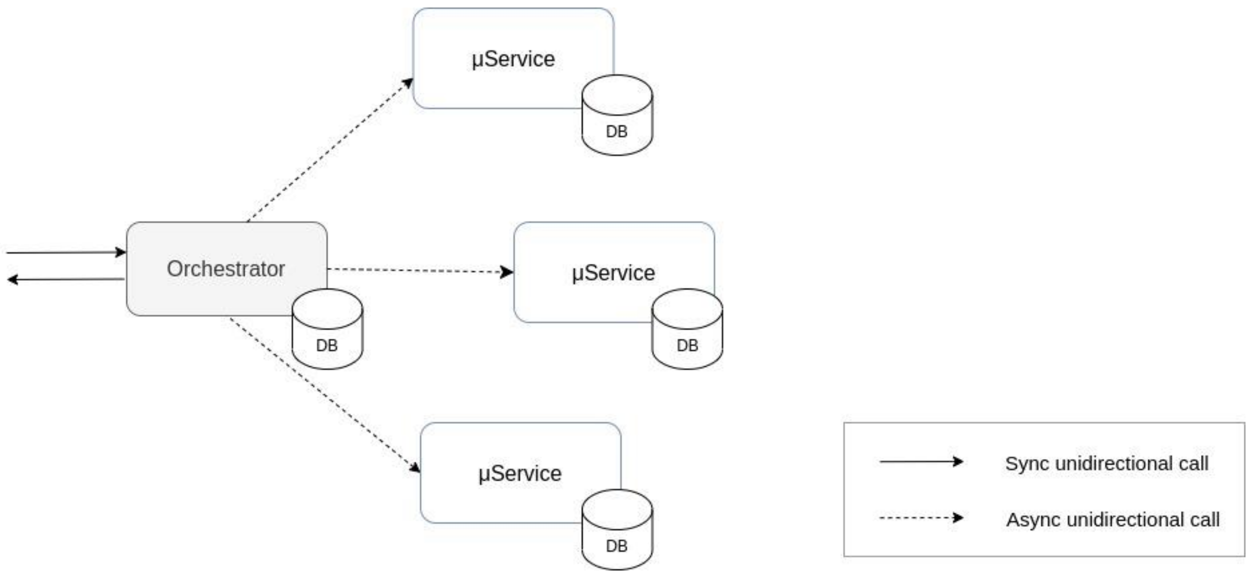

Figure 4-17 The orchestrator communicates asynchronously with the microservices

For instance, if an erroneous workflow execution happens, it might be resolved in a number of ways, including by the orchestrator sending asynchronous messages to all affected domain services to retry the modification or to synchronize the data. The lack of transactionality increases the load on the orchestrator to fix workflow errors and other problems. Even though asynchronous communication provides superior responsiveness, resolving timing and synchronization concerns is more challenging. Race situations, deadlocks, queue reliability, and other distributed architecture headaches might be raised.

This design has a low coupling level, isolating the coupling-intensifying power of transactions to the scope of their respective domain services. In addition, it uses asynchronous communication, which further decouples services from wait conditions and enables more parallel processing. The complexity of the AEO is likewise minimal, reflecting the previously mentioned reduction in coupling. This pattern is straightforward, and the orchestration enables simplified workflow and error-handling solutions.

This design scaled well thanks to asynchronous communication and reduced transaction boundaries, providing effective separation between services. For example, some endpoint services might require higher levels of scale and elasticity in a microservices architecture. In contrast, the backend services might not require scale but relatively higher levels of security. This case can be resolved by separating the two types of services into separate layers. Additionally, domain-level transaction isolation allows the architecture to scale independently of the domain's notions.

Due to the absence of coordinated transactions, this architecture's responsiveness is excellent. Moreover, because each of these services retains its own transactional context, this architecture is well-suited to highly varying service performance footprints amongst services.

4.6.8 AEC SAGA Pattern

The AEC SAGA pattern (AEC) employs asynchronous communication, eventual consistency, and choreographed coordination, giving the least linked example of these patterns, Figure 4-18.

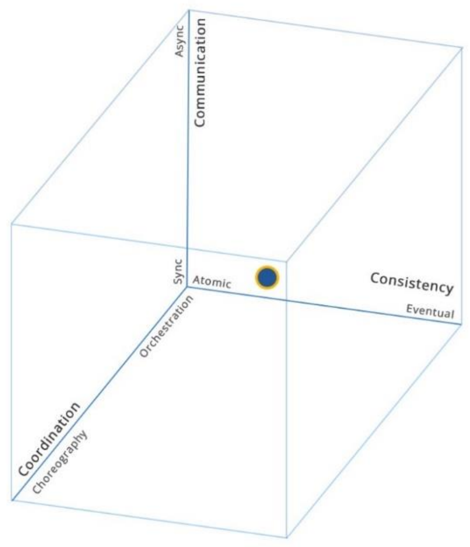

Figure 4-18 AEC utilizes Async communication, Eventual consistency, Choreographed coordination

Figure 4-19 shows how the AEC uses message queues to send messages to other domain services in a way that does not require orchestration. This means that each domain service must provide additional context about the task flows in which they participate, such as error handling and other coordination mechanisms, because each service maintains its own transactional integrity, and there is no orchestrator.

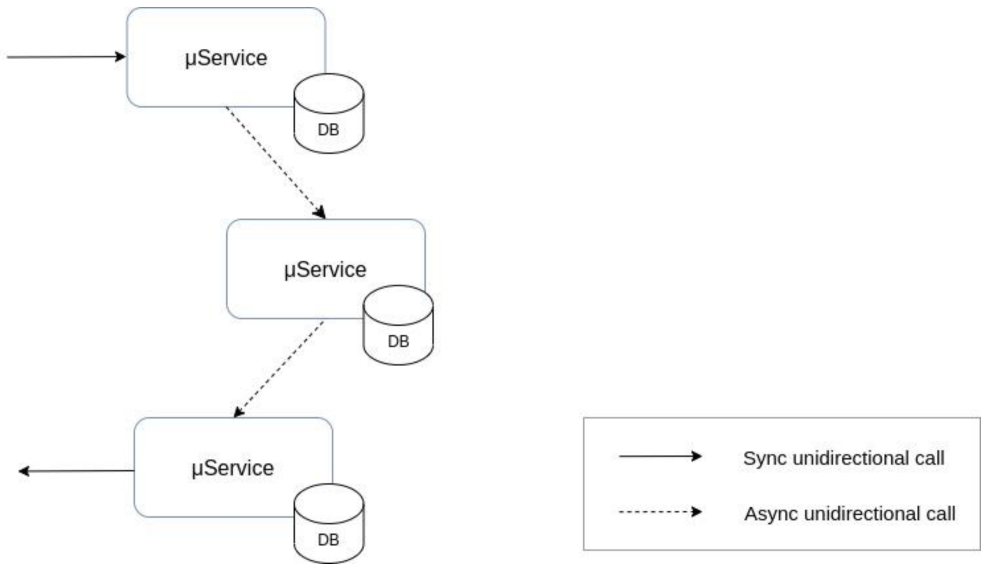

Figure 4-19 The microservices communicate asynchronously with each other

While the lack of orchestration increases the complexity of services, it also enables greater throughput, scalability, and flexibility in the underlying operational architecture. This design has no bottlenecks or coupling choke points, allowing for exceptional responsiveness and scalability.

This pattern is optimal for basic, predominantly linear workflows requiring high processing throughput. It has the greatest potential for high performance and scalability, making it a good option when these characteristics are essential. On the contrary, this pattern is not optimal for handling complicated procedures, especially those revolving around fixing inconsistencies in data. Also, the degree of decoupling makes coordination difficult, and it is prohibitive for complicated or important procedures.

5. PISAGA

The need for a Java library that implements the SAGA pattern arises from the increasing popularity of distributed systems and the need for a robust and scalable solution to manage coordination and consistency in these systems. The SAGA pattern, which is based on the idea of using long-lived transactions and compensating actions, is an effective way to achieve scalability and fault tolerance in distributed systems. However, the current state of the art in the field of SAGA implementation is either complex or expensive to use since it is part of commercial products.

We were inspired to develop a Java library that implements the SAGA pattern because of our experience working with distributed systems and the difficulty we encountered finding a simple and practical solution that implements the SAGA pattern. The benefits of such a library to the community are numerous. First, it would allow developers to quickly implement the SAGA pattern in their systems, increasing their systems' scalability and fault tolerance. Additionally, it would encourage other developers to contribute to the library and enhance its functionality, which would benefit the entire community.

In this section, we present the piSaga[1] library. We provide information about its dependencies, inner functionality, and the case study we created to evaluate it.

5.1 Dependencies

The piSaga library has been developed to implement the SAGA patterns on microservices based on Spring Boot, and they communicate through Apache Kafka. Together, they are a powerful combination for creating microservices. Spring Boot provides the infrastructure and functionality needed to create microservices. In contrast, Apache Kafka provides messaging and event-streaming capabilities to handle the real-time data flow between microservices. The combination of these two technologies allows for the creation of highly scalable, fault-tolerant microservices that can handle a large number of events in real-time.

5.1.1 Spring Boot

Spring Boot [23] is a subproject of the Spring Framework, which is a Java-based platform that provides comprehensive infrastructure support for developing Java applications. It was designed to simplify the process of creating stand-alone, production-grade Spring-based applications by providing tools and libraries that allow developers to quickly create and deploy Spring applications with minimal configuration.

Spring Boot takes an opinionated view of building Spring applications, meaning it has a pre-defined way of doing things in the best way. This means that it makes assumptions

[1] https://github.com/ioannidis/pisaga

about the configuration and dependencies of a Spring application and automatically configures and sets them up for the developer. This can save developers a lot of time and effort, as they do not have to manually configure and set up the various components of a Spring application.

Spring Boot also provides many valuable features and tools, such as:

- Embedded servers allow developers to run their applications as standalone Java applications efficiently.
- The automatic configuration of Spring and its dependencies are based on added jar dependencies.
- Support for a wide range of external libraries and frameworks, such as Hibernate, Jackson, and Lombok.
- A Command Line Interface (CLI) that allows developers to quickly create and run Spring Boot applications from the command line

Overall, Spring Boot is a powerful and easy-to-use platform for developing and deploying Spring-based applications. It can significantly reduce the time and effort required to set up and configure a Spring application and provides several useful features and tools to help developers create high-quality, production-ready applications.

5.1.2 Apache Kafka

Apache Kafka [24] is a distributed, fault-tolerant publish-subscribe messaging system. It was originally developed by LinkedIn and later donated to the Apache Software Foundation. Kafka is designed to be horizontally scalable, fast, and able to handle large volumes of data with low latency.

One of the key features of Kafka is its ability to process and transmit streaming data, which makes it a popular choice for building real-time data pipelines and streaming applications. Kafka is often used to build real-time data pipelines that ingest data from multiple sources and process and transmit the data to multiple consumers.

In addition to its streaming capabilities, Kafka also provides a number of other useful features, such as:

- Durable message storage: Kafka stores all published messages for a configurable amount of time, allowing consumers to read messages at their own pace.
- High throughput: Kafka is designed to handle high volumes of data with low latency, making it suitable for use cases requiring high data ingestion and processing levels.
- Scalability: Kafka is horizontally scalable, which can be easily deployed across a cluster of machines to support high data ingestion and processing levels.

Overall, Apache Kafka is a robust, scalable, and fault-tolerant publish-subscribe messaging system well-suited for building real-time data pipelines and streaming applications.

5.2 piSaga library

As we said earlier, the state machine is a useful modeling choice when attempting to understand how a SAGA orchestrator works. A state machine is made up of a set of states and a set of events that move the machine from one state to another. Each transition might have an action, which in the context of a SAGA, is the invocation of a SAGA participant. Completing a local transaction executed by a SAGA participant triggers the transitions between states. The existing state and the precise outcome of the local transaction determine the state transition and subsequent action. Consequently, employing a state machine paradigm makes developing, constructing, and testing sagas easier.

Therefore, the piSaga defines the contracts needed to model our SAGA transactions as a state machine. We aim to simplify the sagas using intuitive naming conventions, enlightening the development process. The following list contains details about piSaga's implementation.

- interface DomainEvent
  DomainEvent interface defines a contract for the behavior of an object that represents the events of the domain model.
- class DomainModelAndEvents
  Sometimes, when we update the domain model, we need both the update domain model and the events that correspond to the update. Thus, here is where the DomainModelAndEvents come into the picture. The "model" attribute contains the updated domain model, and the "events" attribute is a list that contains all the related events.
- abstract class SagaCommand
  The SagaCommand abstract class is used to model the commands that the orchestrator sends to the SAGA participants. A SAGA command contains three attributes. The id, type, and the actual command data.
- @interface SagaCommandListener
  The SagaCommandListener annotation is used to annotate the class, which is responsible for listening to the commands published to the specified topic.
- @interface SagaCommandHandler
  The SagaCommandHandler annotation is used to annotate the methods that will handle and process the incoming SAGA commands. Each SAGA command handler method is responsible for handling the specified SAGA command type that is defined by the method argument.
- abstract class SagaEvent
  The SagaEvent abstract class is used to model the events that the SAGA participants send to the orchestrator. A SAGA event contains three attributes. The id, type, and the actual event data.
- @interface SagaEventListener
  The SagaEventListener annotation is used to annotate the class, which is responsible for listening for the events that are published to the specified topic.

- @interface SagaEventHandler
  The SagaEventHandler annotation is used to annotate the methods that will handle and process the incoming SAGA events. Each SAGA event handler method is responsible for handling the specified SAGA event type that is defined by the method argument.
- class UnsupportedStateTransitionException
  The UnsupportedStateTransitionException exception is used to notify the developers that the attempted state transition is not supported.

5.3 piSaga case study

In order to demonstrate and evaluate the capabilities of the piSaga library, we have created a case study based on online orders. The goal of this system is to observe the responsiveness and measure the performance of our library. Multiple experiments were performed for validation and evaluation purposes. The results indicate that the proposed method can address the absence of isolation in the SAGA pattern.

We have implemented two functionalities, the creation of a new order and the cancellation of an order. Figure 5-1 depicts the Create Order SAGA state machine:

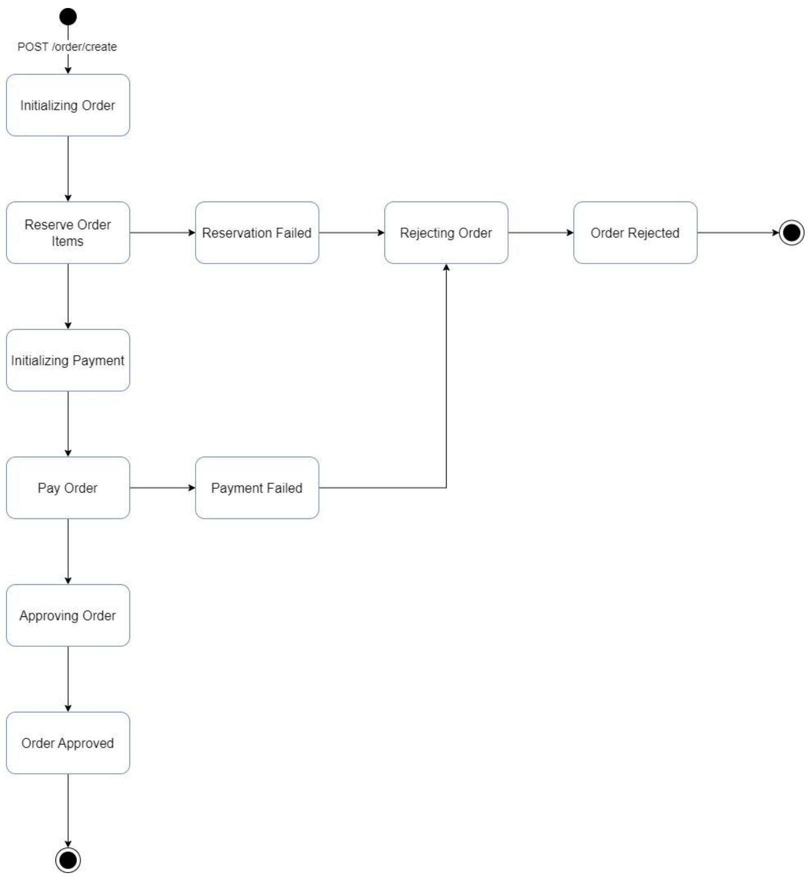

Figure 5-1 Create order state machine model

- Initializing Order - The initial state.
- Reserve Items – Waiting for the Warehouse Service to reserve the requested product.
- Pay Order - Waiting for Payment Service to complete the payment.
- Order Approved - A final state indicating that the SAGA was completed successfully.
- Order Rejected - A final state indicating that the Order was rejected by one of the participants.

Additionally, the state machine defines multiple state transitions. For instance, the state machine transitions from the "Initializing Order" state to "Reserve Items" or "Rejected Order". When it receives a successful reply to the "Reserve Items" command, it switches to the "Payment" state. Alternatively, the state machine moves to the "Rejected Order" state if the Payment-Service cannot complete the payment.

The initial operation of the state machine is to emit the "Reserve Items" command to Warehouse-Service. Warehouse-Service's reaction initiates the subsequent state shift. If the reservation of the item was successful, the payment is initialized, and the state is changed to "Pay Order". However, if the item reservation fails, the "Order" is rejected, and the SAGA enters the "Rejecting Order" state. The state machine endures multiple additional state changes, driven by the responses from SAGA participants until it achieves either "Order Approved" or "Order Rejected" as its final state.

Figure 5-2 depicts the Cancel Order SAGA state machine:

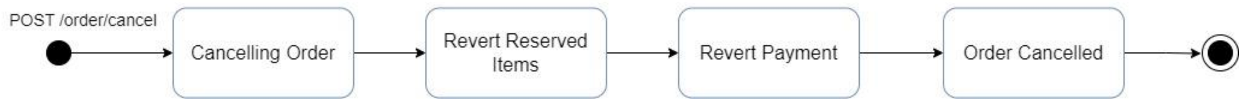

Figure 5-2 Cancel order state machine model

- Canceling order - The initial state.
- Order canceled - A final state indicating that the SAGA canceled successfully.

6. EVALUATION

We have conducted several experiments to evaluate piSaga's performance and ability to overcome the lack of isolation. The experiments' infrastructure consists of the Order-Service, Warehouse-Service, and Payment-Service. Additionally, we deployed Apache Kafka and PostgreSQL on Docker.

For the evaluation, we used Apache JMeter [26]. JMeter is an open-source performance testing tool used to analyze and measure the performance of various services and applications. It is particularly useful for measuring the performance of web applications. Hence, JMeter simulates a heavy server load and analyzes overall performance under different load types.

To evaluate the behavior of piSaga under different situations, we created three scenarios with various configurations.

6.1 Implementations

We decided to execute the scenarios on two popular variations of the SAGA pattern combined with the default cache that Spring Boot provides.

We implemented the SAC variation by creating Spring Boot microservices that communicate with each other using the OpenFeign integration. OpenFeign [27] is a Java library that allows you to create declarative HTTP clients. It simplifies the process of interacting with RESTful web services by providing an easy-to-use, type-safe, and expressive interface for making HTTP requests. The SAC SAGA pattern combines the principles of atomicity and choreography without the need for a designated orchestrator. However, to achieve atomicity, some degree of cooperation is necessary. The initially accessed service is the Order-Service, which acts as the front controller responsible for initiating the workflow. It then forwards the request to the subsequent service in the process, continuing until the workflow is completed successfully. All services implement the corresponding business logic to support compensating requests. By utilizing synchronous communication and choreography, bottlenecks can be minimized in non-error scenarios. The benefits of this pattern include improved throughput and reduced bottlenecks, as the final service in the process can return the result. However, the lack of a mediator can negatively impact performance, as communication between services is necessary.

Next, we developed the AOE variation by developing microservices that communicate using the piSaga library and Apache Kafka. The AEO pattern is based on asynchronous communication and eventual consistency. This strategy employs an orchestrator responsible for managing the whole transaction process, making it suited for complicated operations. In our experiments, the orchestrator is a component of the Order-Service. Each microservice that participates in the workflow is accountable for maintaining a consistent state, as they may need to synchronize shared data in the background or via the orchestrator. This solution is straightforward, and the orchestration feature enables simplicity, flexibility, and error-handling solutions.

It is important to note that even though the SAGA orchestrator resides in the Order-Service, it still sends a command message to the Order-Service in the very last step. Theoretically, the “Create Order SAGA” might approve the Order by directly modifying it. To maintain consistency, however, the SAGA treats Order-Service as another participant.

6.1.1 ConcurentHashMap cache

Spring Boot supports default caching by integrating the Spring Framework's caching abstraction. This abstraction allows developers to use different caching providers in their applications without changing the code.

The default caching implementation in Spring Boot uses a simple in-memory cache based on the ConcurrentMapCacheManager, which is a thread-safe implementation of the CacheManager interface that stores cache entries in a ConcurrentHashMap. This means that each cache is essentially a map, with the cache name as the key and a ConcurrentMap as the value. The advantages of using a ConcurrentHashMap as a cache in Spring Boot are that it is easy to configure, swift and efficient, and allows multiple threads to access the cache simultaneously without requiring explicit locks.

In our experiments, we will use the ConcurentHashMap cache in order to cache the SAGAs. So there is a difference between the implementations that utilize cache and the others that will not. When we do not use the cache, every time that we perform an operation on a SAGA, we have to retrieve it from the database. On the other hand, when we use a cache, we need to access the database only the first time we retrieve a SAGA. Then, any subsequent operation will retrieve the corresponding SAGA from the cache. By doing this, we improve the performance of our application since database access is expensive, especially on distributed systems.

6.2 Experiment Scenarios

The system’s performance will be evaluated using the following three different scenarios:

1. Successful completion of transaction - no exceptions.
2. An exception occurred during the item’s reservation in Warehouse-Service or the payment in Payment-Service.
3. Order cancellation by the user.

6.2.1 Scenario 1 – Successful order creation

When the orchestrator has captured a "Create Order Event", the event workflow will be taken care of by the microservices that are relevant to that event. First, the Order-Service initializes the order. Second, the Warehouse-Service initiates the retrieval of the requested quantity of products. Next, the validation of the payment will subsequently be handled by the Payment-Service.

The Order-Service completes the order and updates the necessary information, including the order status in the last step. The workflow will be ended, and the order will be marked as "APPROVED" since the transaction will be completed successfully, Figure 6-1.

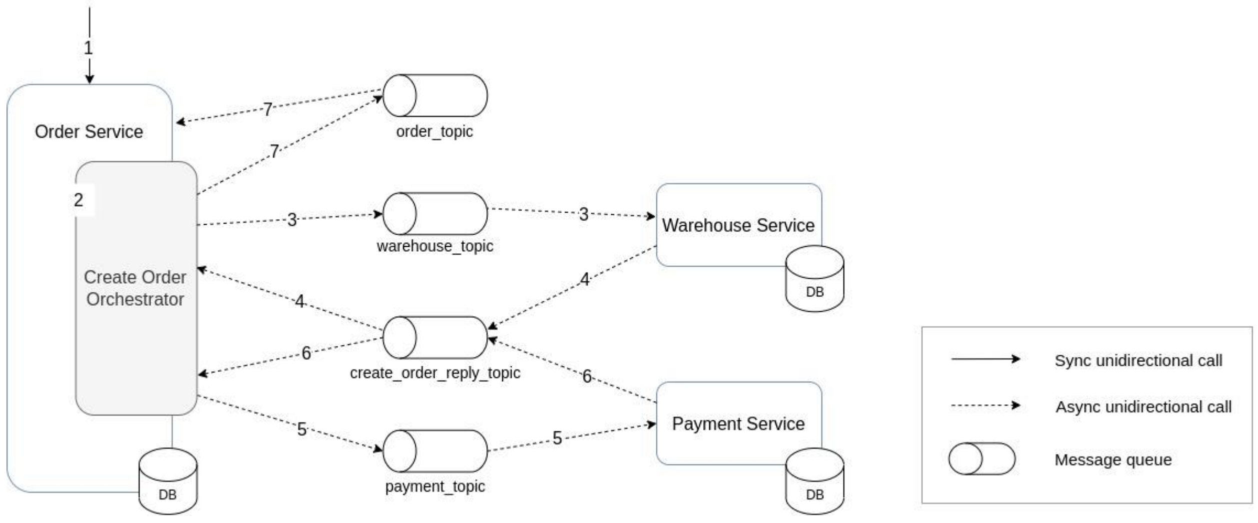

Figure 6-1 Successful order creation workflow

The steps for this scenario are the following:

1. `Order-Service` receives a request for a new order.
2. `Order-Service` creates an `Order` and a `Create Order Orchestrator`.
3. The `Create Order Orchestrator` sends a `ReserveOrderItemsCommand` command to `Warehouse-Service`, via the `warehouse_topic`.
4. `Warehouse-Service` replies with an `OrderItemsReservedEvent` event to `Create Order Orchestrator`, via the `create_order_reply_topic`.
5. The `Create Order Orchestrator` sends a `PayOrderCommand` to `Payment-Service`, via the `payment_topic`.
6. `Payment-Service` replies with a `PaymentApprovedEvent` event via the `create_order_reply_topic`.
7. The `Create Order Orchestrator` sends an `ApproveOrderCommand` command to `Order-Service`, via the `order_topic`.

6.2.2 Scenario 2 – Order creation fails at items reservation

Following the orchestrator's successful capture of a "create-order-event," the event process will be handled by the appropriate microservices. The Order-Service starts the ordering process. Second, when a customer has placed an order, Warehouse-Service begins pulling the necessary inventory. Here, an error will occur because the requested amount is unavailable. Hence, the order must be rejected.

The Warehouse-Service will fire an event, informing the orchestrator that the requested amount could not be allocated. Then the orchestrator will initiate the procedure for rejecting the order, Figure 6-2.

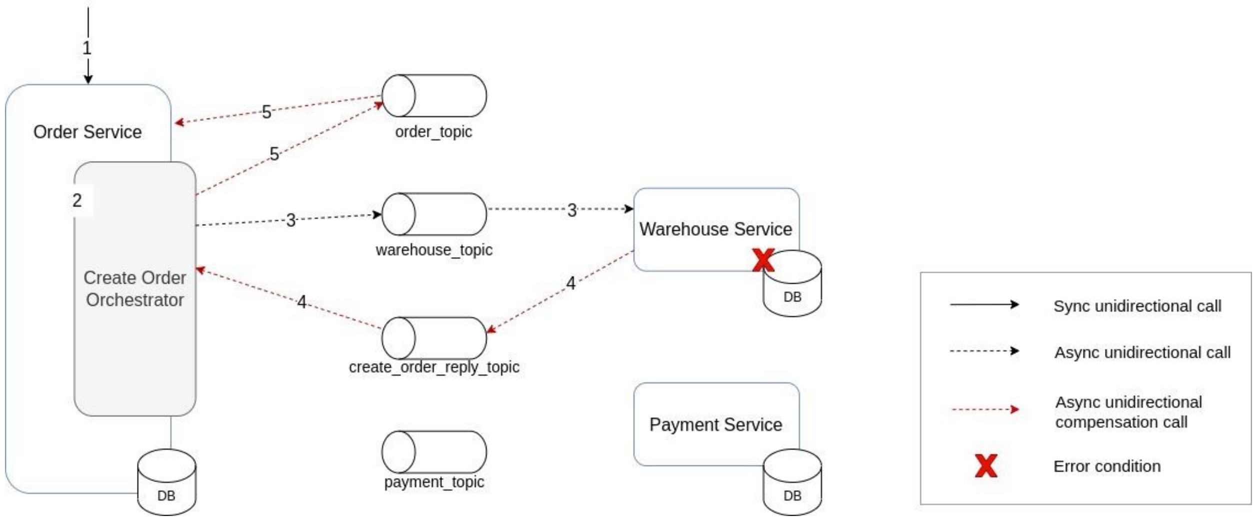

Figure 6-2 Compensatory workflow when an error occurs at the Warehouse-Service

For this scenario, the flow is as follows:

1. `Order-Service` receives a request for a new order.
2. `Order-Service` creates an `Order` and a `Create Order Orchestrator`.
3. The `Create Order Orchestrator` sends a `ReserveOrderItemsCommand` command to `Warehouse-Service`, via the `warehouse_topic`.
4. The `Warehouse-Service` replies with an `OrderItemsOutOfStockEvent` event via the `create_order_reply_topic`.
5. The `Create Order Orchestrator` sends a `RejectOrderCommand` command to `Order-Service`, via the `order_topic`.

6.2.3 Scenario 2 – Order creation fails at payment

Following the orchestrator's successful capture of a "Create Order Event", the event process will be handled by the relevant microservices. The Order-Service initiates the ordering procedure. Second, when a consumer has put in an order, the Warehouse-Service pulls the required inventory. Next, the payment will be handled by the Payment-Service. At this point, we will experience an error because the user will not have the needed amount of money to pay the order. Hence, the order must be rejected.

Nevertheless, we also have to revert the actions made in Warehouse-Service for the corresponding order.

The Payment-Service will emit an event informing the orchestrator that the user does not have the credit to pay for the order. Then the orchestrator will initiate the procedure for rejecting the order, sending a “Reject Order” command to Warehouse-Service and Order-Service, Figure 6-3.

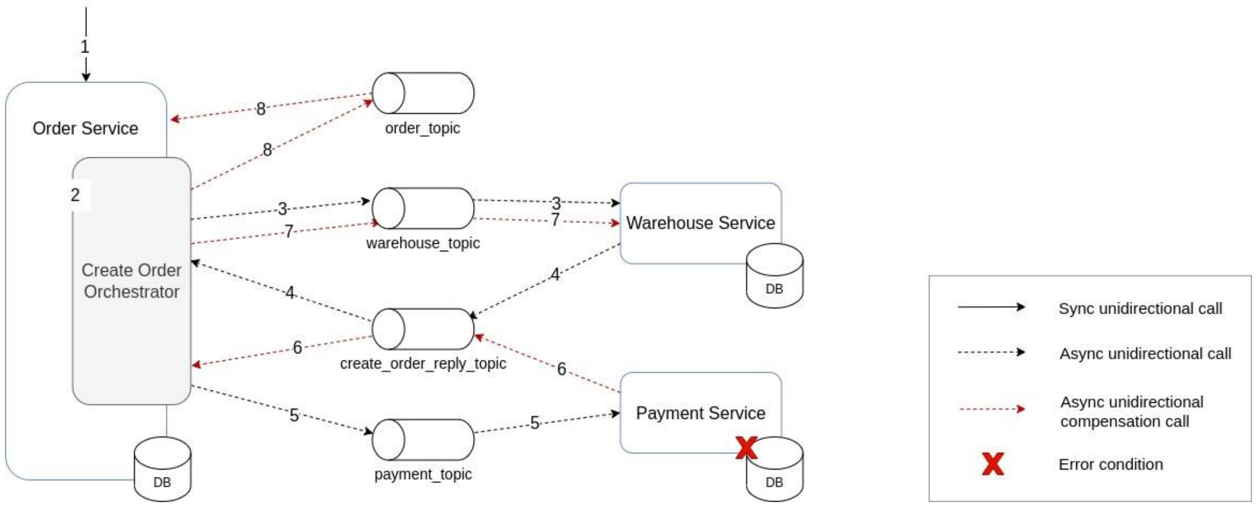

Figure 6-3 Compensatory workflow when an error occurs at the Payment-Service

This flow contains a few more steps in addition to the previous one:

1. `Order-Service` receives a request for a new order.
2. `Order-Service` creates an `Order` and a `Create Order Orchestrator`.
3. The `Create Order Orchestrator` sends a `ReserveOrderItemsCommand` command to `Warehouse-Service`, via the `warehouse_topic`.
4. `Warehouse-Service` replies with an `OrderItemsReservedEvent` event to `Create Order Orchestrator`, via the `create_order_reply_topic`.
5. The `Create Order Orchestrator` sends a `PayOrderCommand` to `Payment-Service`, via the `payment_topic`.
6. `Payment-Service` replies with a `PaymentRejectedEvent` event, via the `create_order_reply_topic`.
7. The `Create Order Orchestrator` sends a `RejectOrderCommand` command to `Warehouse-Service`, via the `warehouse_topic`.
8. The `Create Order Orchestrator` sends a `RejectOrderCommand` command to `Order-Service`, via the `order_topic`.

6.2.4 Scenario 3 – Cancelling order

The difference between this scenario and scenarios 2, and 3 is that the cancellation of the order is performed by the user. This action is not part of the automated workflow that the SAGA provides. The order can be canceled when the state is “WAREHOUSE_APPROVED” or “PAYMENT_APPROVED”.

The compensating transactions that should be emitted depend on the current state of the order. If the current state is “WAREHOUSE_APPROVED” then the orchestrator sends only a “Cancel Order” command to Order-Service, Figure 6-4.

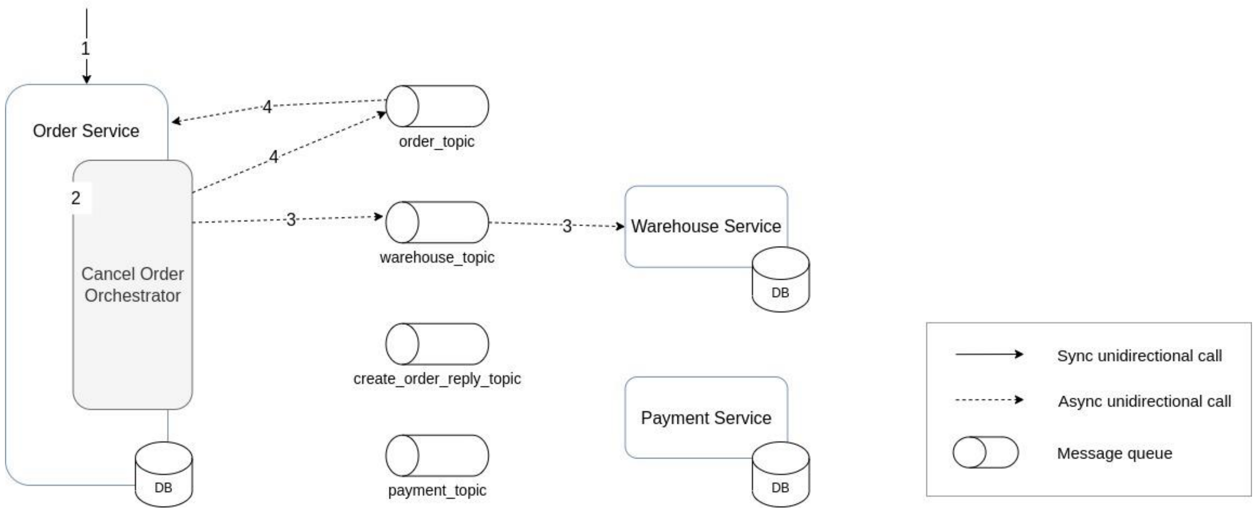

Figure 6-4 Order cancellation workflow at “WAREHOUSE_APPROVED” state

The flow for this scenario is the following:

1. `Order-Service` receives a request to cancel an order.
2. `Order-Service` creates a `Cancel Order Orchestrator`.
3. The `Create Order Orchestrator` sends a `CancelOrderCommand` command to `Warehouse-Service`, via the `warehouse_topic`.
4. The `Create Order Orchestrator` sends a `CancelOrderCommand` command to `Order-Service`, via the `order_topic.`

If the current state is “PAYMENT_APPROVED” the orchestrator sends the “Cancel Order“ command both to Warehouse-Service and Order-Service, Figure 6-5.

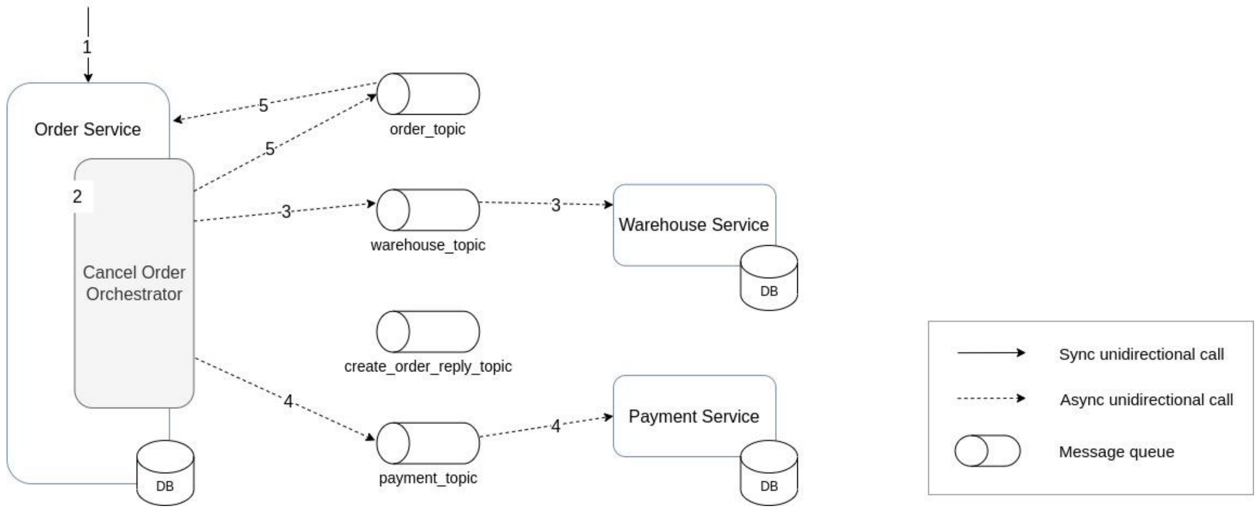

Figure 6-5 Order cancellation workflow at "PAYMENT_APPROVED" state

The flow for this scenario is the following:

1. `Order-Service` receives a request to cancel an order.
2. `Order-Service` creates a `Cancel Order Orchestrator`.
3. The `Create Order Orchestrator` sends a `CancelOrderCommand` command to `Warehouse-Service`, via the `warehouse_topic`.
4. The `Create Order Orchestrator` sends a `CancelOrderCommand` command to `Payment-Service`, via the `payment_topic.`
5. The `Create Order Orchestrator` sends a `CancelOrderCommand` command to `Order-Service`, via the `order_topic.`

6.3 Experimental Results

We ran the experiment scenarios we extensively described in the previous section using various parameters. As we mentioned earlier, the scenarios will be applied to two systems composed by:

1. Choreographed microservices using RESTful communication based on the SAC pattern.
2. Orchestrated microservices using the piSaga event-driven communication based on the AOE pattern.

It is worth mentioning that during the execution of all of the experiments with the piSaga library, we did not face any errors related to the lack of isolation. This means that the piSaga successfully handles the isolation at the application level of each microservice.

6.3.1 Scenario 1 – Throughput

As throughput, we define the number of requests processed in 30 seconds. We consider only the requests in the “APPROVED” state as completed. We evaluated the performance of various configurations in terms of throughput by running the scenario with 1, 10, and 25 virtual users. We achieved this by configuring the virtual users in jMeter to stream requests for 30 seconds. Then we counted how many SAGA flows had been successfully completed. The experiment was repeated five times to ensure the consistency and accuracy of the results.

6.3.1.1 1 virtual user

The results for having a single virtual user are presented in Table 6-1. The baseline configuration is “OpenFeign without cache”, with an average throughput of 8222.2 requests per 30 seconds.

When comparing the baseline configuration with the “piSaga without cache” configuration, we can see that the latter performs slightly better, with an average throughput of 8331.6 requests per 30 seconds. This represents an improvement of 1.33%. The “OpenFeign with cache” configuration also improves over the baseline, with an average throughput of 8407 requests per 30 seconds, representing an improvement of 2.5%. However, the best performance is achieved by the “piSaga with cache” configuration, which reaches an average throughput of 8516.6 requests per 30 seconds. This represents an improvement of 3.58% over the baseline configuration.

It is clear that the configuration of “piSaga with cache” provides the highest throughput among all the configurations we tested. The difference in throughput between the best option and the other configurations is 3.58%, 2.22%, and 1.30%, respectively.

Table 6-1 Successful-request/30sec for 1 virtual user

	No Cache		***Cache***	
#	**OpenFeign**	**piSAGA**	**OpenFeign**	**piSAGA**
1	8199	8372	8343	8545
2	8199	8285	8430	8574
3	8170	8343	8430	8401
4	8314	8343	8401	8545
5	8229	8315	8431	8518
avg	8222.2	8331.6	8407	8516.6

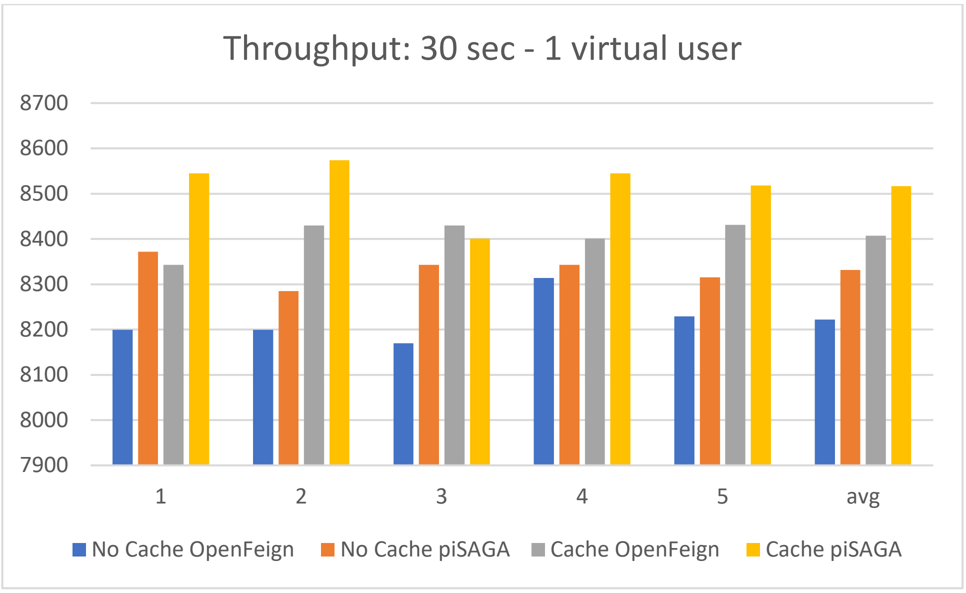

Figure 6-6 Graphical representation of throughput for 1 virtual user

6.3.1.2 10 virtual users

The results for having 10 virtual users are presented in Table 6-2. The first configuration we evaluated used the “OpenFeign without cache”, resulting in an average throughput of 9480.5 requests per 30 seconds.

This configuration serves as the baseline for comparison with the other configurations. Additionally, using “piSaga without cache” resulted in an average throughput of 9588 requests per 30 seconds, showing an improvement of 1.13% compared to the baseline. Using “OpenFeign with cache” resulted in an average throughput of 9671.5 requests per 30 seconds, showing an improvement of 2.01% compared to the baseline. Last, it was found that using “piSaga with cache” resulted in the highest average throughput at 9837.5 requests per 30 seconds. This configuration showed a significant improvement of 3.77% compared to the baseline

Based on the results, it is clear that the combination of piSaga with a cache resulted in the highest throughput of 9837.5 requests per 30 seconds. It is important to note that the percentage differences between the best option and the others are 3.77%, 2.60%, and 1.72%, respectively.

Table 6-2 Successful-request/30sec for 10 virtual user

	No Cache		*Cache*	
#	**OpenFeign**	**piSAGA**	**OpenFeign**	**piSAGA**
1	9518	9719	9778	9787
2	9387	9486	9652	9817
3	9463	9562	9695	9795
4	9644	9677	9745	9912
5	9428	9627	9594	9826
avg	9480.5	9588	9671.5	9837.5

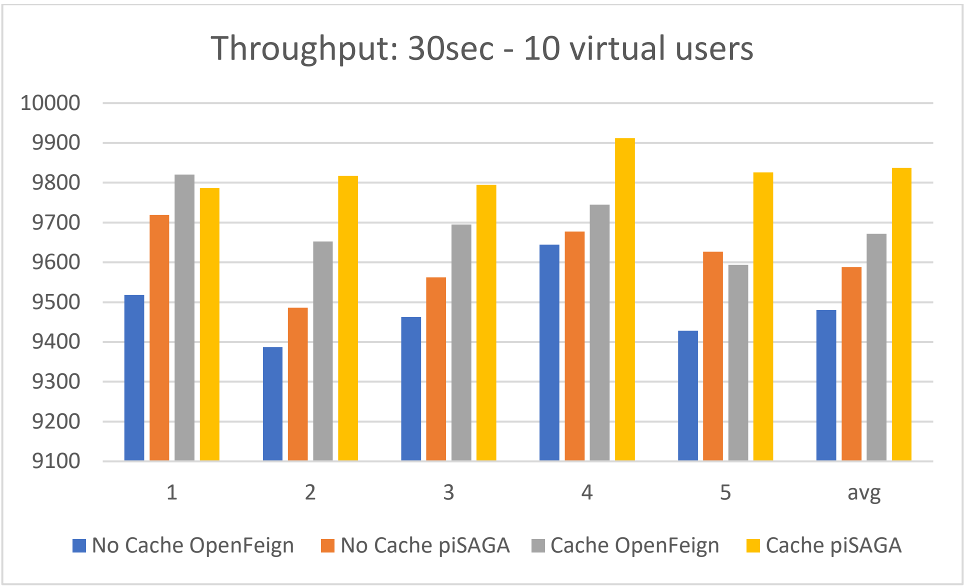

Figure 6-7 Graphical representation of throughput for 10 virtual users

6.3.1.3 25 virtual users

The results for having 25 virtual users are presented in Table 6-3. The implementation with “OpenFeign without cache” resulted in an average throughput of 9993.4 requests per 30 seconds and served as the baseline for comparison.

It can be observed that the implementation with “piSaga without cache”, which resulted in an average throughput of 10151.2 requests per 30 seconds, has a 1.58% higher throughput compared to the baseline. Next, implementing “OpenFeign with cache” resulted in an average throughput of 10251.2 requests per 30 seconds and a 2.58% higher throughput than the baseline. Lastly, the implementation with “piSaga with cache” resulted in an average throughput of 10403.6 requests per 30 seconds and a 4.10% higher throughput than the baseline.

It can be concluded that the configuration with cache and piSaga is the best option, providing the highest throughput. The difference in throughput between the best option and the other configurations is relatively small, with the most significant difference being 4.06% between the best option and the others are 4.10%, 2.49%, and 1.49%, respectively.

Table 6-3 Successful-request/30sec for 25 virtual user

	No Cache		*Cache*	
#	**OpenFeign**	**piSAGA**	**OpenFeign**	**piSAGA**
1	9915	10150	10034	10256
2	9892	10284	10372	10601
3	10099	10174	10131	10240
4	10052	10054	10405	10495
5	10009	10094	10314	10426
avg	9993.4	10151.2	10251.2	10403.6

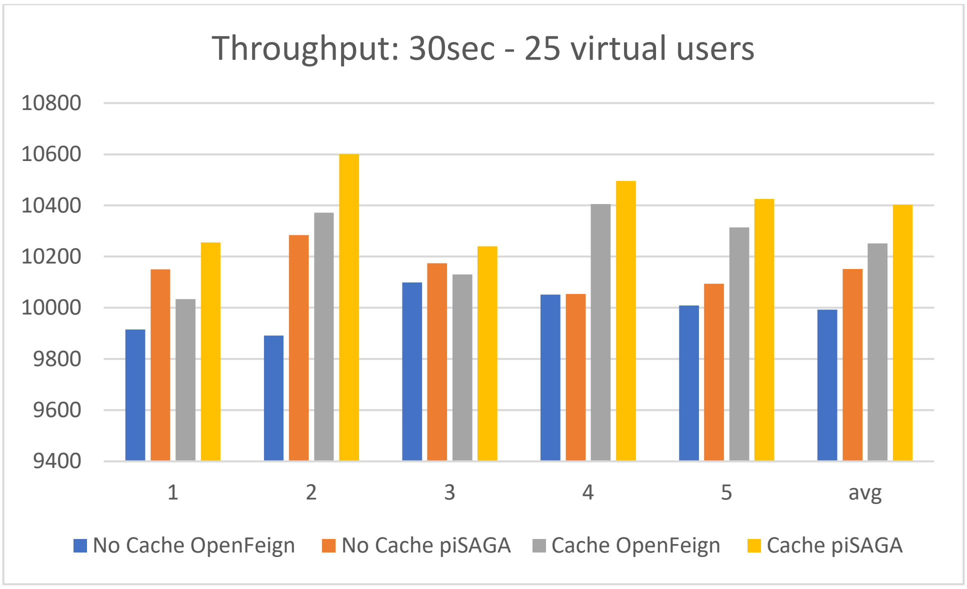

Figure 6-8 Graphical representation of throughput for 25 virtual users

6.3.1.4 Experiment outcome

Based on the experimental results, it can be concluded that the combination of piSaga with a cache consistently results in the highest throughput across all configurations and virtual user scenarios. The improvement in throughput ranges from 1.13% to 4.10% compared to the baseline configuration of “OpenFeign without cache”. Additionally, the difference in throughput between the best option and the other configurations is relatively small. The most significant difference is 4.06% between the best option and the implementation with “OpenFeign without cache”.

It is also worth noting that using cache alone, whether with OpenFeign or piSaga, results in a slight improvement in throughput compared to the baseline configuration. This suggests that using a cache is an essential factor in optimizing performance.

In summary, the best option for improving performance and increasing throughput is to use the configuration of piSaga with a cache. This configuration consistently results in the highest throughput across all scenarios and significantly improves performance compared to the baseline configuration.

6.3.2 Scenario 2 – Required time to complete requests

In this scenario, we investigate how long it takes to complete 10000 requests when there is an error either in the Warehouse-Service or the Payment-Service. Hence, using jMeter, we configured the 1 virtual user to stream 10000 requests.

6.3.2.1 Experiment 1

In this experiment, 20% of the incoming requests will fail in the Warehouse-Service; hence compensation transactions will be made to revert the corresponding order to a valid state. We measure how long it takes to either approve or reject an order.

Table 6-4 Required time, in seconds, to complete 10000 requests

Error Rate		*No Cache*		*Cache*	
WS	**PS**	**OpenFeign**	**piSAGA**	**OpenFeign**	**piSAGA**
20%	**0%**	39	36	35	33

The results in Table 6-4 show that “piSaga with cache” performed the best, taking only 33 seconds to complete 1000 requests. This is a significant improvement over the other configurations, with “OpenFeign with cache” coming in second place at 35 seconds, “piSaga without cache” taking 36 seconds, and “OpenFeign without cache” taking the longest at 39 seconds.

It is clear from these results that the use of a cache can significantly improve the performance of the system, with the “piSaga with cache” configuration reducing the required time by 9.38% compared to “piSaga without cache”, and by 18.75% compared to “OpenFeign without cache”. Additionally, using “piSaga with cache” outperforms “OpenFeign with cache”, reducing the required time by 6.25%.

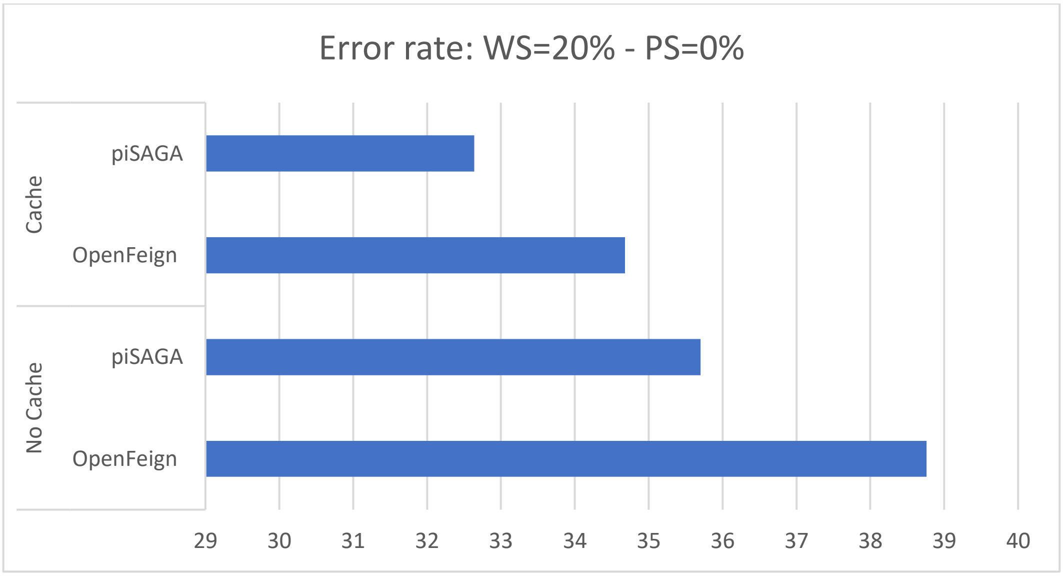

Figure 6-9 Cache-piSaga needs 33 seconds to complete 10000 requests

6.3.2.2 Experiment 2

In this experiment, 40% of the incoming requests will fail in the Warehouse-Service. Next, 20% - out of the 60% of the requests that the Warehouse-Service approved - will fail in the Payment-Service. Therefore, a compensation transaction will be made in order to revert the corresponding order to a valid state. We measure how long it takes to either approve or reject an order.

Table 6-5 Required time, in seconds, to complete 10000 requests

Error Rate		*No Cache*		*Cache*	
WS	**PS**	**OpenFeign**	**piSAGA**	**OpenFeign**	**piSAGA**
40%	**20%**	34	30	31	28

The results in Table 6-5 show that the best performance was achieved by the “piSaga with cache” approach, which took only 28 seconds to complete the task. This is a significant improvement when compared to the other approaches. The “piSaga withour cache” approach was the second-best option, taking 30 seconds, followed by the “OpenFeign with cache” and “OpenFeign without cache” with 31 and 34 seconds, respectively.

Comparing the best option to the other approaches, we can see that it had a 6.90% improvement over the “piSaga without cache”, an 10.34% improvement over the “OpenFeign with cache”, and a 20.69% improvement over the “OpenFeign without cacahe”.

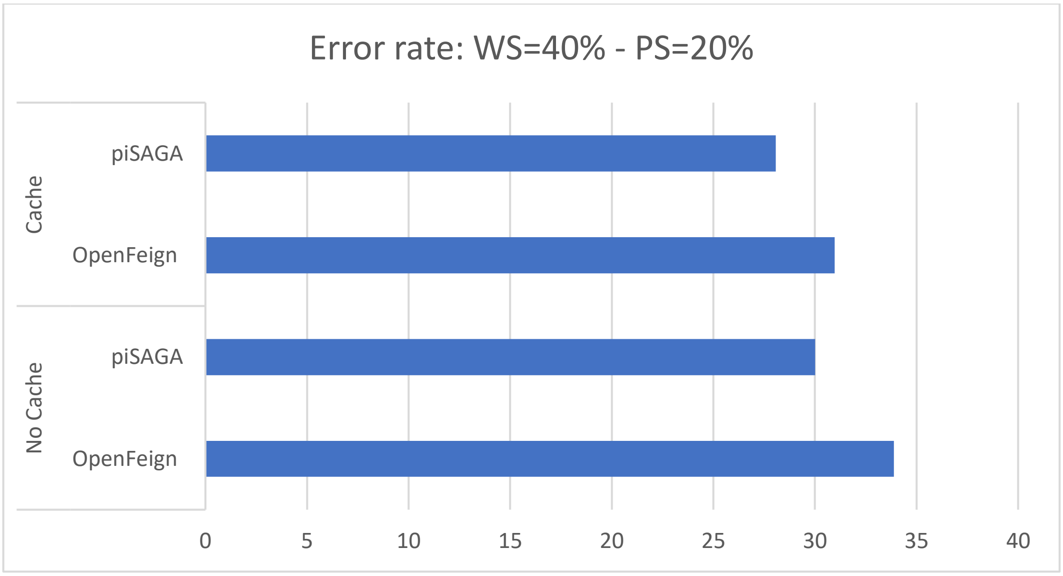

Figure 6-10 Cache-piSaga needs 28 seconds to complete 10000 requests

6.3.2.3 Experiment 3

In this experiment, 60% of the incoming requests will fail in the Warehouse-Service. Next, 40% - out of the 40% of the requests that the Warehouse-Service approved - will fail in the Payment-Service. Therefore, a compensation transaction will be made in order to revert the corresponding order to a valid state. We measure how long it takes to either approve or reject an order.

Table 6-6 Required time, in seconds, to complete 10000 requests

Error Rate		*No Cache*		*Cache*	
WS	**PS**	**OpenFeign**	**piSAGA**	**OpenFeign**	**piSAGA**
60%	**40%**	32	30	28	26

Table 6-6 shows that "piSaga with cache" performs better than the other approaches, needing only 26 seconds. This is a 7.96% improvement over the second-best option, which is "OpenFeign with cache", with 28 seconds. The no-cache options, piSaga, and OpenFeign, have average completion times of 30 and 32 seconds, respectively. These results suggest that incorporating a cache system, specifically using piSaga, can significantly improve the efficiency and speed of the system in high error rate scenarios. Additionally, the percentage difference between the best option and the no-cache options in this scenario is 15.38% and 26.92% for piSaga and OpenFeign, respectively.

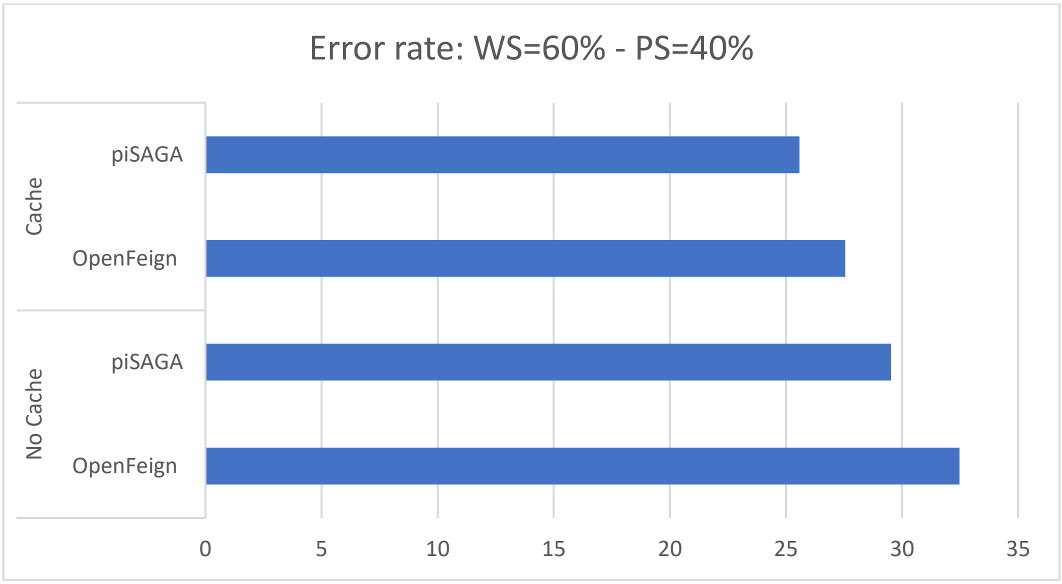

Figure 6-11 Cache-piSaga needs 26 seconds to complete 1000 requests

6.3.2.4 Experiment outcome

In summary, the experimental results show that incorporating a cache system can significantly improve the performance of the system. The best configuration was found to be "piSaga with cache", which had the fastest completion time in all runs. This configuration outperformed the other approaches, reducing the required time by at least 6.90% compared to the no-cache options and by 6.25% compared to OpenFeign. The results also suggest that piSaga is more effective than OpenFeign in high error rate scenarios.

6.3.3 Scenario 3 – Processing time for order cancellation

Our goal for this performance test is to investigate how piSaga performs under heavy load of request using stress testing [34]. To accomplish this, we run two experiments on that scenario. In the first one, we configured 10 virtual users on jMeter to stream 10000 requests, and in the second 50 virtual users to stream 100000 requests. We compared only the implementations with ConcurrentHashMap cache, both using OpenFeign and piSaga, since they are the ones that perform the best.

We measured the “Processing time” and the “Total time”. As “Processing time”, we define the time between the last sent request and the last processed order, whereas the “Total time” is the time between the first sent request and the final processed order.

6.3.3.1 Experiment 1

Table 6-7 presents the results of the performance execution when 10 virtual users initiated 1000 requests, resulting in a total of 10000 requests. The results show that piSAGA performs better than OpenFeign, with a processing time of 34 seconds compared to 38 seconds for OpenFeign. Additionally, the total time for piSAGA is 38 seconds, while it is 42 seconds for OpenFeign. This results in a percentage difference of 11.76% in processing time and 10.53% in total time in favor of piSAGA.

Table 6-7 Processing and total time, in seconds, for 10 virtual uses and 10000 in total

	Processing time	*Total Time*
OpenFeign	38	42
piSAGA	34	38

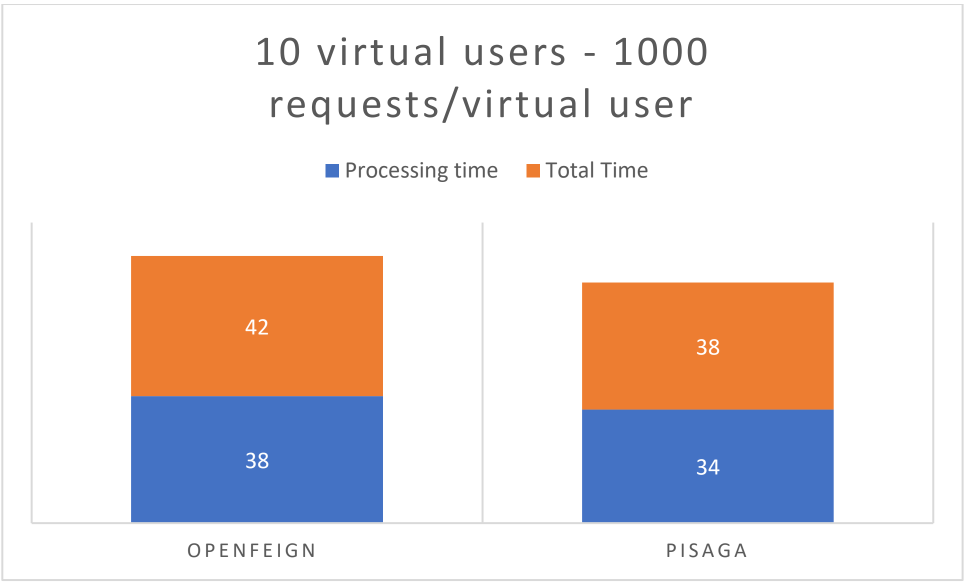

Figure 6-12 piSaga needs 38 seconds to complete 10000 cancellation requests

6.3.3.2 Experiment 2

In Figure 6-8 we present the results of the performance execution when 50 virtual users initiated 2000 requests. So, the total number of requests is 100000. This experiment shows that processing time and total time are shorter when using piSAGA than OpenFeign. Specifically, the processing time for piSAGA is 599 seconds, while for OpenFeign, it is 608 seconds, representing a 1.5% difference. Similarly, the total time for piSAGA is 613 seconds, while for OpenFeign, it is 625 seconds, representing a 1.96% difference.

Table 6-8 Processing and total time, in seconds, for 50 virtual uses and 100000 in total

	Processing time	***Total Time***
OpenFeign	608	625
piSAGA	599	613

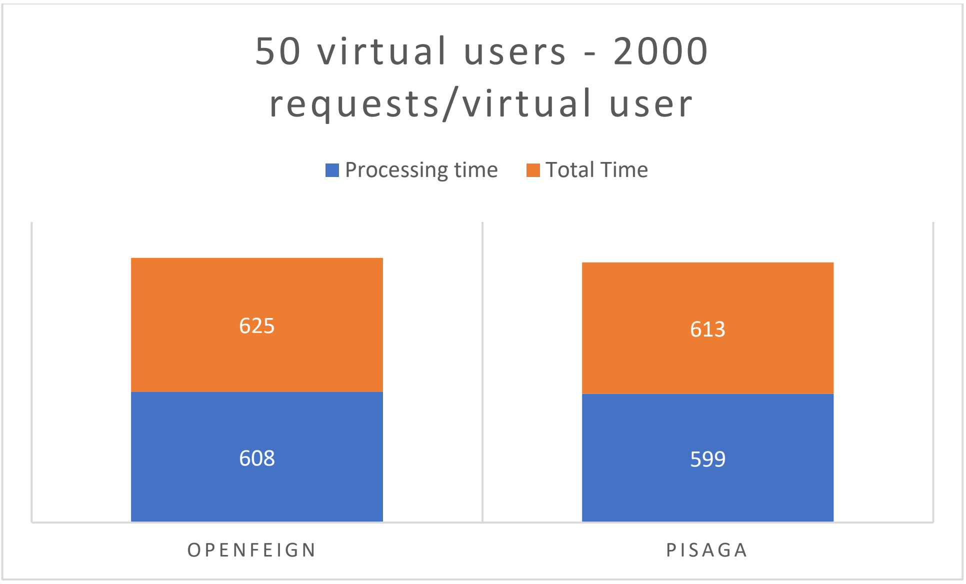

Figure 6-13 piSaga needs 613 seconds to complete 100000 cancellation requests

6.3.3.3 Experiment outcome

In conclusion, based on the results of both experiments on this scenario, it can be seen that piSAGA performs better than OpenFeign in terms of processing time and total time. This suggests that piSAGA may be a better option for systems that require low latency and fast processing times. It is important to note that these results are obtained from a specific set of conditions. The performance may vary depending on factors such as network conditions and the specific implementation of the tested system. Additionally, we can see that piSaga outperforms OpenFeign by at least 1.5% and 1.96% in processing and total time, respectively, in both experiments. These results demonstrate that using piSaga with a cache is an efficient and effective approach for improving system performance.

7. CONCLUSION

In conclusion, the SAGA pattern is a powerful approach to implementing distributed transactions in distributed systems. It provides robust assurances of atomicity, consistency, and managing isolation on the application level while permitting the execution of long-running transactions that may span several systems or locations.

Another significant feature of the SAGA pattern is its ability to handle failures and errors gracefully. In the event of a failure or error, the SAGA pattern permits the execution of compensating transactions that can undo the results of prior operations and restore the system to a consistent state. This is particularly crucial in distributed systems, where failures and errors are more frequent due to the complexity and diversity of the involved systems.

Despite these advantages, it is essential to carefully analyze the SAGA pattern's trade-offs. Specifically, the high amount of coordination and communication needed by the SAGA pattern may have a negative impact on performance, especially in systems with high concurrency. In addition, implementing the SAGA pattern may be more complex than other approaches to distributed transactions, which may raise the risk of errors and the amount of time necessary to design and maintain the system.

Regarding the piSaga library, we can see that the “piSaga with cache” approach performs better than all the other approaches. Based on the extended experiments, simulating three scenarios with several configurations, we can see that it provides better throughput and needs less time to successfully serve the same number of requests (reaching a final state). Ultimately, cache can significantly improve the performance of the system piSaga is a good option for developing practical microservices that support distributed transactions.

Overall, the SAGA pattern is a valuable tool for developers who wish to create scalable and reliable distributed systems since it offers a variety of advantages that may significantly boost the performance and reliability of these systems. However, it is essential to carefully consider the trade-offs associated with it and select the best suitable technique for a specific use case.

8. ABBREVIATIONS - ACRONYMS

2PC	Two-Phase Commit
2PC*	Two-Phase Commit with Recovery
ACD	Atomicity Consistency Durability
ACID	Atomicity Consistency Isolation Durability
CAP	Consistency Availability Partition-Tolerance
CA	Consistency Availability
CP	Consistency Partition-Tolerance
AP	Availability Partition-Tolerance
IPC	Inter-process Communication
RESTful	Representational state transfer
TCC	Try, Commit and Cancel
GRIT	Generalized Recursive Idealized Transactions
μService	Microservice

REFERENCES

[1] Seth Gilbert and Nancy Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services", ACM SIGACT News, Volume 33 Issue 2 (2002), pg. 51–59.

[2] “Cap theorem,” Wikipedia, 29-Jul-2022. [Online]. Available: https://en.wikipedia.org/wiki/CAP_theorem

[3] IBM Cloud Education, “What is the CAP Theorem?” IBM. [Online]. Available: https://www.ibm.com/cloud/learn/cap-theorem

[4] Kleppmann, M., 2017. Designing Data-Intensive Applications. O'Reilly Media, Inc.

[5] ACID - Wikipedia. Available at: https://en.wikipedia.org/wiki/ACID

[6] Werner Vogels. 2009. Eventually consistent. Commun. ACM 52, 1 (January 2009), 40–44. https://doi.org/10.1145/1435417.1435432

[7] S. Newman, Building microservices: Designing fine-grained systems. Sebastopol, CA: O'Reilly, 2021.

[8] C. Richardson, Microservice patterns meap. Shelter Island, NY: Manning, 2017.

[9] Frank, Lars, and Torben U. Zahle. "Semantic ACID properties in multidatabases using remote procedure calls and update propagations." Software: Practice and Experience 28.1 (1998): 77-98.

[10] M. Richards and N. Ford, Fundamentals of Software Architecture: An Engineering Approach. O'Reilly, 2020.

[11] “Microservices,” martinfowler.com. [Online]. Available: https://martinfowler.com/articles/microservices.html.

[12] B. Christudas, Practical microservices architectural patterns. Apress, 2019.

[13] “2 ways to implement atomic database transactions in a microservice-based solution,” Apriorit, 28-Jul-2022. [Online]. Available: https://www.apriorit.com/dev-blog/777-web-atomic-transactions-in-microservices

[14] K. Xiang, “Patterns for distributed transactions within a microservices architecture,” Red Hat Developer, 23-Sep-2022. [Online]. Available: https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture

[15] “Solving distributed transaction management problem in microservices architecture using Saga,” IBM developer. [Online]. Available: https://developer.ibm.com/articles/use-saga-to-solve-distributed-transaction-management-problems-in-a-microservices-architecture

[16] C. R. of Eventuate, “Building microservices: Inter-process Communication,” NGINX, 24-Oct-2019. [Online]. Available: https://www.nginx.com/blog/building-microservices-inter-process-communication

[17] Garcia-Molina, Hector, and Kenneth Salem. "Sagas." ACM Sigmod Record 16.3 (1987): 249-259.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design patterns: Elements of reusable object-oriented software. 1995.

[19] “Two-phase Commit Protocol - Wikipedia.” Two-phase Commit Protocol - Wikipedia, https://en.wikipedia.org/wiki/Two-phase_commit_protocol

[20] Fan, P., Liu, J., Yin, W. et al. 2PC*: a distributed transaction concurrency control protocol of multi-microservice based on cloud computing platform. J Cloud Comp 9, 40 (2020). https://doi.org/10.1186/s13677-020-00183-w

[21] Community, Alibaba Cloud. “An In-Depth Analysis of Distributed Transaction Solutions.” Alibaba Cloud Community, www.alibabacloud.com/blog/an-in-depth-analysis-of-distributed-transaction-solutions_597232

[22] G. Zhang, K. Ren, J. -S. Ahn and S. Ben-Romdhane, "GRIT: Consistent Distributed Transactions Across Polyglot Microservices with Multiple Databases," 2019 IEEE 35th International Conference on Data Engineering (ICDE), 2019, pp. 2024-2027, doi: 10.1109/ICDE.2019.00230.

[23] “Spring Boot.” Spring Boot, www.spring.io/projects/spring-boot

[24] “Apache Kafka.” Apache Kafka, www.kafka.apache.org

[25] Vogels - https://www.allthingsdistributed.com/, Dr Werner. “Eventually Consistent - Revisited.” Eventually Consistent - Revisited | All Things Distributed, 23 Dec. 2008, www.allthingsdistributed.com/2008/12/eventually_consistent.html

[26] “Apache JMeter.” Apache JMeter, www.jmeter.apache.org

[27] “Spring Cloud OpenFeign.” Spring Cloud OpenFeign, www.docs.spring.io/spring-cloud-openfeign/docs/current/reference/html

[28] Štefanko, Martin, et al. “The Saga Pattern in a Reactive Microservices Environment.” Proceedings of the 14th International Conference on Software Technologies, SCITEPRESS - Science and Technology Publications, 2019. Crossref, https://doi.org/10.5220/0007918704830490.

[29] Dürr, Karolin & Lichtenthaeler, Robin & Wirtz, Guido. (2021). An Evaluation of Saga Pattern Implementation Technologies.

[30] Ibsen, Claus, and Jonathan Anstey. Camel in Action. 2010.

[31] “Saga.” Apache Camel, www.camel.apache.org

[32] “Sagas - Axon Reference Guide.” Sagas - Axon Reference Guide, www.docs.axoniq.io/reference-guide/axon-framework/sagas

[33] “Eclipse MicroProfile LRA.” Eclipse MicroProfile LRA, 19 May 2020, www.download.eclipse.org/microprofile/microprofile-lra-1.0-M1/microprofile-lra-spec.html

[34] Cloud, Alibaba. “Introduction to Stress Testing.” Medium, 29 Dec. 2017, www.alibaba-cloud.medium.com/introduction-to-stress-testing-a80d115b3715