Tampere University

Aleksi Hirvonen

COORDINATION BETWEEN MULTIPLE MICROSERVICES: A SYSTEMATIC MAPPING STUDY

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Davide Taibi
Xiaozhou Li
February 2023

ABSTRACT

Aleksi Hirvonen: Coordination between multiple microservices: A Systematic Mapping Study
Master of Science Thesis
Tampere University
Master's Programme in Information Technology
February 2023

The popularity of microservice architecture has risen recently due to its multiple advantages partly related to the increased independence of services. One of the features that improve independence is decentralized data management, which outlines that each service should manage its own data with preferred data management technologies. However, the usage of decentralized data management brings problems, especially with data consistency when data owned by separate microservices must be modified in coordination. To alleviate this, a shared database between services could be used as it removes the need for coordination altogether, but then again, the usage of a single database could defeat some of the benefits of microservice architecture by increasing tight coupling between services. Therefore, it is important to consider other possibilities to manage the coordination while maintaining the independence of the services.

We conducted a systematic mapping study to find out suitable design patterns to manage the coordination between multiple microservices. Firstly, design patterns that seemed widely discussed and adopted were identified. After this, these patterns were presented using a template that included advantages and disadvantages for each pattern.

The results gathered in the systematic mapping study show that even though traditional systems pursue strict consistency with ACID guarantees, eventual consistency patterns, such as the saga pattern, seem to be more popular in the microservice environment. This is due to drawbacks within distributed transaction protocols including limited concurrency and reduced availability which makes developers choose loosened consistency as a trade-off for higher availability and increased performance. The prevalence of the saga pattern can be seen in the selected works as there are multiple articles proposing methods to manage different parts of the pattern. Also, implementation details were mainly related to the saga pattern in the selected works.

Even though the saga pattern is currently the most prevalent option, there is still interest in highly consistent coordination methods in the research community. Multiple solutions have been proposed, which either propose new consistency protocols with strict consistency guarantees or entirely new solutions to remove the need for coordination completely. However, there are no novel solutions that could manage the requirements of microservice architecture reliably in the industry setting yet. Therefore, further research is still required to refine already proposed solutions or to vision new solutions for this problem.

Keywords: 2PC, TCC, saga, orchestration, choreography, coordination, transaction, microservice, systematic mapping study

TIIVISTELMÄ

Aleksi Hirvonen: Koordinaatio usean mikropalvelun välillä: Systemaattinen kirjallisuuskartoitus
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Helmikuu 2023

Mikropalveluarkkitehtuurin suosio on kasvanut huomattavisti viimeisen kymmenen vuoden aikana sen tarjoamien hyötyjen takia, jotka ovat osittain seurausta palveluiden vähentyneestä riippuvuudesta toisiinsa. Riippumattomuutta lisää esimerkiksi hajautettu tiedonhallinta, jonka mukaan jokaisen palvelun tulisi olla vastuussa omistamansa tiedon hallinasta käyttäen sopivinta tietokantateknologiaa. Vaikka tällä voidaan saavuttaa useita etuja, aiheuttaa se myös uusia ongelmia etenkin tiedon yhtenäisyyden hallinnassa kun usean palvelun hallitsemaa tietoa täytyy muokata yhteistyössä. Tämä ongelma voitaisiin välttää käyttämällä yhteistä tietokantaa palveluiden välillä, mutta se osittain poistaisi mikropalvelun hyödyt tuomalla lisää riippuvuusuhteita mikropalveluiden välille. Tästä syystä on tärkeää tarkastella muita vaihtoehtoja hajautetun tiedon hallintaan siten, että mikropalvelun hyötyjä on mahdollista ylläpitää.

Tässä työssä toteutetaan systemaattinen kirjallisuuskartoitus, jonka tavoitteena on löytää sopivia malleja usean mikropalvelun väliseen koordinointiin. Aluksi työssä tunnistetaan koordinointimallit, joista käydään paljon keskustelua kirjallisuudessa. Tämän jälkeen jokaisesta valitusta mallista keskustellaan käyttäen yhteistä keskustelukaavaa, joka sisältää mallin määrittelyn sekä hyötyjen ja haittojen listaamisen.

Kirjallisuuskartoituksessa saatujen tulosten perusteella huomattiin, että mikropalveluarkkitehtuurissa suositaan malleja jotka tarjoavat lopulta yhtenäistä tulosta (eng. eventual consistent). Tämä eroaa huomattavasti perinteisistä ohjelmistoista, joissa yhtenäisyyden täytyy olla ehdoton ja toteuttaa kaikki ACID periaatteet. Ero johtuu osittain siitä, että mallit joilla voidaan tarjota ehdoton johdonmukaisuus usean palvelun välillä vähentää mahdollisuutta rinnakkaisuudelle ja lisäksi vaikuttaa palveluiden saavutettavuuteen heikentävästi. Tästä syystä mikropalveluarkkitehtuurissa usein luovutaan ehdottomasta yhtenäisyydestä, koska sen seuraksena voidaan saavuttaa korkeampi suorituskyky ja lisääntynyt saavutettavuus. Etenkin saga-mallin havaittiin olevan suosittu yhtenäisyyden hallintaan mikropalveluiden välillä, koska siitä keskusteltiin ja siihen liittyviä parannusehdotuksia ja toteutustapoja ehdotettiin useissa töissä.

Vaikkakin saga-malli on tällä hetkellä yleisesti käytetty tapa mikropalveluiden välisessä koordinoinnissa, valitusta kirjallisuudesta huomattiin myös tarve ehdottoman johdonmukaisuuden toteuttaville malleille. Useita uusia malleja ehdotettiin ratkaisemaan tämän hetkisissä ratkaisuissa olevia ongelmia, mutta myös ratkaisuja joilla voitaisiin poistaa tarve usean palvelun väliseen koordinointiin ehdotettiin. Vaikka ehdotetut mallit ovatkin lupaavia, ne ovat vasta suunnitteluvaiheessa eikä niitä voida käyttää luotettavasti tai helposti teollisuusympäristössä. Tästä syystä lisätutkimuksia tarvitaan näiden uusien mallien jalostamiseen tai kokonaan uusien mallien visiontiin.

Avainsanat: 2PC, TCC, saga, orkestraatio, koreografia, koordinaatio, transaktio, mikropalvelu, systemaattinen kirjallisuuskartoitus

PREFACE

This work was carried out while being employed by Vertex Systems Oy. First, I am thankful for Juhana Rajala, who provided support, helpful advice, and proofreading while acting as my thesis supervisor from Vertex Systems.

I would also like to express my gratitude towards Timo Tulisalmi and Petri Molkkari for the possibility to work full-time on the thesis. Additionally, I am grateful to my co-workers for the great atmosphere, where it was pleasant to work with the thesis.

I also feel thankful to my supervisors from Tampere University, Davide Taibi and Xiaozhou Li for their assistance and feedback during the process. Their guidance on how to conduct a mapping study was invaluable in order to shape the thesis into its final form.

Finally, I want to show my appreciation to Emma, my friends, and my family for their encouraging words and tremendous support during my studies.

Tampere, 6th February 2023

Aleksi Hirvonen

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

2PC	Two-phase commit
ACID	Atomicity, Consistency, Isolation, Durability
BASE	Basically Available, Soft state, Eventually consistent
CAP	Consistency/Availability/Partition tolerance
CQRS	Command and Query Responsibility Segregation
ES	Event Sourcing
GTM	Global transaction manager
LSN	Log sequence number
MAS	Multi-Agent System
MVCC	Multi-version concurrency control
OCC	optimistic concurrency control
P/T	Place and Transition
REST	Representational State Transfer
ROA	Resource Oriented Architecture
SHMT	Single-head-multiple-tails
SMS	Systematic Mapping Study
SOA	Service Oriented Architecture
ST	Single-tail
TCC	Try-Cancel/Confirm
TPS	transactions-per-second

1. INTRODUCTION

Microservice architecture has gotten more popular in the industry over the last ten years and multiple large companies have adopted it such as Netflix, The Guardian, and Amazon [1]. Adoption of the microservice architecture has been high due to the advantages it provides such as higher scalability, independent deployment, and the possibility to choose the most suitable technologies for each service. However, as with each architectural pattern, also microservice architecture involves multiple disadvantages such as increased latency, and possible data consistency issues.

Data consistency problems are related to the usage of multiple databases where microservices can manage their own data with selected database technology. Even though the usage of multiple databases relates to all previously mentioned advantages of the microservice architecture, the possibility for database transaction is lost as a trade-off when data in multiple microservices needs to be modified in coordination. For this, additional patterns are required to manage the coordination. To achieve similar transactional guarantees as with database transactions, distributed transaction protocols such as two-phase commit are required to be used. However, when strict consistency is used for the coordination, the availability of the system decreases as stated by CAP theorem [2]. As developers are willing to relax the consistency to gain better availability for the system, patterns with eventual consistency have risen to challenge more traditional distributed transaction protocols.

The saga pattern is one of the most discussed patterns for coordination in microservices to guarantee eventual consistency. This pattern allows dividing the coordination into smaller microservice level sub-transactions which are executed in a sequence. In case that error occurs, already finished steps are reversed with compensating transactions. As the coordination is managed as a sequence and not inside a single transaction, short-lived failures of participants do not affect the possibility of successfully finishing the task. This is not the same with strict consistency patterns as the availability of each service is required to commit.

A choice between design patterns could be made based on previous experience or by comparing patterns based on commonly available information. Especially when there is no previous experience, the quality of commonly available information becomes crucial

for finding a suitable solution. For this reason, it becomes important to collect a comprehensive description of each design pattern in a way that it can be understood why they are used, how they can be implemented, and what problems could occur in the implementation. In addition, an overview of the coordination methods and related recent works could be beneficial for researchers to understand the current stage of the research field and possible future directions. With this motivation in mind, a Systematic Mapping Study (SMS) is conducted.

The contribution of this work is in four parts:

(i) Design patterns used for coordination between multiple microservices

(ii) Advantages and disadvantages of extracted design patterns

(iii) Implementation of found design patterns and problems that arise in the implementation

(iv) Advantages and disadvantages of extracted implementation patterns

To our knowledge, there has only been a limited amount of work studying design patterns and their implementation [3] [4]. Differences between our work and related literature are further discussed in section 2.3. With this work, we aim to give a comprehensive view of the design and implementation of coordination between services which should help companies to find the most suitable patterns for their needs. Also, the current stage of the research is reviewed to help find trends for future studies.

The structure of this work is as follows. In chapter 2, related concepts of transactions and microservice architecture are discussed. Also, related work is discussed in the chapter. Chapter 3 first defines the used methodology and after that goals, research questions, and how the search was conducted using the chosen methodology. Results of the conducted search are then gathered and organized for further discussion in chapter 4. Results are then discussed in chapter 5, and possible threats to the validity of our work are covered in chapter 6. Finally, this work is concluded in chapter 7.

2. BACKGROUND

In this section, concepts related to our work are defined and discussed. As our work will study coordination between multiple microservices, it is important to understand some basic concepts of transactions as trade-offs based on these concepts are made between different design patterns. After this, the microservice architecture is defined using common characteristics, advantages, and disadvantages of the architecture to better understand properties that might be important when implementing an application using this architectural pattern. After this, related work is reviewed and differences compared to our work are outlined.

2.1 Transaction basics

In software engineering, a transaction is a concept that is used to respond to threats concerning concurrent execution, partial execution, and possible crashes in database systems. As databases implement a transaction paradigm, it gives programmers the possibility to work with data sequentially while the database works concurrently. [5]

2.1.1 ACID properties

In traditional transaction processing systems, data integrity is an important characteristic and should be managed in all cases, even when there are unexpected problems with the hardware or software. Generally, ACID properties have been used to check how well transaction processing systems can manage integrity. These properties can be defined as follows [6]:

- *Atomicity* confirms that either all operations inside the transaction succeed or none succeed which means that there is no possibility for a partial transaction to commit. This means that in case of failure, rollback or abort is performed.
- *Consistency* is a guarantee that initially consistent database should also be consistent after a transaction has been committed. This means that each integrity constraint set must be enforced before and after each transaction. Integrity constraint can be, for example, a rule that each ID must be unique or that balance cannot be negative.

- *Isolation* provides serializability which means that multiple transactions could be executed in a serial order that provides the same output as running the same transactions concurrently. This makes it possible that developers can think that each transaction is run alone in isolation from other transactions.
- *Durability* guarantees that after a transaction has been executed, all updates are in stable storage which handles problems such as power outages or operating system failures.

These properties can be usually guaranteed in traditional databases. However, in distributed systems, where multiple underlying databases are used, this compliance is harder to achieve but is possible using distributed transaction protocols. Even if it is possible to achieve all of these properties in distributed systems, it is not always desirable due to the decreased performance as a trade-off.

2.1.2 Isolation levels

To increase the performance, the isolation property of ACID can be relaxed. There are multiple levels of isolation that define what other users might see while the transaction is in progress. Isolation levels are defined based on anomalies that can be defined as follows [7]:

- *Dirty write:* If there is a possibility of overwriting modifications made by a transaction that has not yet been committed or rolled back, this anomaly is possible. In case either of these transactions wants to rollback, the correct previous value is hard to obtain.
- *Dirty read:* When a transaction T1 modifies a resource and another transaction T2 reads the same resource before T1 rolls back. When T1 rolls back, T2 has a value that should not exist as T1 rolled back its transaction.
- *Non-repeatable read:* If transaction T1 reads a value of a resource and after that, another resource modifies or deletes the same resource, T1 will not get the same value as with the initial read.
- *Phantom read:* When transaction T1 queries a set of resources with specified conditions, it can not be guaranteed that the same set of resources is returned with rereading. This is because there is a possibility that another transaction has created and committed a resource corresponding to the same condition whilst T1 is in progress.
- *Lost update:* There is a possibility that the update made by a transaction is lost in the following case. Transaction T1 reads a resource and after that, transaction T2 reads that same value and commits modifications to the same resource. If transaction T1 then uses that previously read value and commits, updates made by

T2 are lost.

- *Read and write skews:* These anomalies can happen if there is a constraint between two resources X and Y. *Read skew* can happen when transaction T1 reads resource X but before it can read the resource Y, another transaction T2 modifies resources X and Y. Eventually when T1 reads resource Y, constraint between resources X and Y are no longer valid as X is read before the modification and Y after the modification. *Write skew* is possible when transaction T1 reads resources X and Y but before it can commit wanted modifications, transaction T2 modifies the resource Y. Eventually when T1 modifies resource X, constraint between these resources might be broken.

Isolation levels are defined based on anomalies that are not allowed the stronger the isolation gets. The following list includes isolation levels from weakest to strongest.

- *Read uncommitted:* Prevents only *dirty write* anomaly. Other previously mentioned anomalies can occur.
- *Read committed:* Prevents *dirty write* and *dirty write* anomalies. Other previously mentioned anomalies can occur.
- *Repeatable read:* Prevents other anomalies but not *phantom read*.
- *Snapshot:* Phantom read can sometimes occur and write skew can occur. Other anomalies are prevented.
- *Serializable:* Prevents all previously mentioned anomalies which makes it the strongest isolation level.

When isolation is relaxed, higher concurrency can be achieved but as a consequence, there is a possibility for dirty reads or setting an incorrect state. This is why the trade-off when setting isolation level is between performance and correctness of the state. [7]

2.1.3 BASE properties

In all cases, strict consistency provided by ACID properties is not desirable as there are trade-offs to it. According to the CAP theorem, any system can only ensure two desirable attributes from partition tolerance, consistency, and availability. As this theorem is related to the whole system and not only database transactions, the definition of consistency differs from consistency in ACID-properties [2].

- **Partition tolerance:** States that the system should still be receptive even when individual components are unavailable.
- **Availability**: Intended response should be returned for each request made to the system.

- **Consistency:** Each operation made to the system should look like it was made at once. Only the latest value should be returned, otherwise, an error is returned.

When data is partitioned in multiple database servers, partition tolerance is required as a system should be operational even though one of the partitions is unavailable. This leaves a trade-off to be made between consistency and availability. However, it is important to notice that the trade-off must be made only in cases when a partition occurs.

When consistency is desirable, ACID transactions for partitioned databases can be pursued using distributed transaction protocols. However, in the case of a partition when consistency is desirable, the availability of the system is impacted as availability is the product of the components required for the operation. [8]

In cases where high availability is pursued, distributed transactions enforcing ACID principles are not suitable. As an alternative set of principles, BASE semantics has been proposed which stands for *Basically available*, *Soft state*, and *Eventually consistent*. With BASE semantics, outdated data can be momentarily accepted if it is guaranteed to be eventually consistent. Exchange for weaker consistency is made as it allows management of partial failures with less complexity and gives a possibility for better performance and higher availability. [9]

2.2 Microservices

Microservice architecture has become more popular in the last ten years and multiple big companies have adopted it such as Netflix, The Guardian, and Amazon [1]. Related to the definition of this pattern, there has been a discussion in the literature about if microservice architecture is a subset of more traditional Service Oriented Architecture (SOA) or an entirely new architectural pattern [1].

As there is no clear consensus on microservice architecture in the literature, multiple different definitions have emerged for it. In this thesis, an acknowledged set of common characters for microservices from Lewis et al. [1] is used. This definition includes the following nine characters [1]:

- *Componentization via Services.* Service should be independently deployable and runnable where all communication goes through their public interfaces. Running independent services makes it possible to change services without having to redeploy the entire application which is the case for traditional monolithic applications. When a service needs to be updated, only that certain service needs to be redeployed.
- *Organized around business capabilities.* Instead of dividing teams based on technology layers, such as the database layer and the backend layer, in the microservice teams are assembled to manage and implement multiple different layers inside a

singular service. With this, teams have knowledge from multiple different aspects of development, and cross-team implementations are not needed as often.

- *Products not projects.* Microservice application should be seen as a product that is both developed and maintained by one team for the whole lifetime. This differs from the project model, where development is managed by one team, and then reassigned to the maintenance team on completion.
- *Smart Endpoints and Dumb Pipes.* Communication methods between services and clients should be lightweight and not include any logic. This aims to keep services decoupled as all logic related to a service is managed within that service. For example, HTTP requests or messaging protocols such as RabbitMQ can be used as lightweight communication protocols.
- *Decentralized governance.* The technologies used can be selected per microservice. This enables the selection of the most suitable technologies depending on the use case and the team's preferences.
- *Decentralized Data Management.* In each microservice, data is managed in the way preferred by the team. This might include different conceptual models of similar data between different microservices if it is necessary. In addition, each microservice controls the storage of its own data and can choose the technology used according to the needs and expertise of the team.
- *Design for Failure.* Failure of one service should not cascade to other services and failures should affect user experience as little as possible. To detect failures, real-time monitoring is used extensively where architectural and business-related metrics are monitored.
- *Automation.* In microservices, continuous integration and continuous delivery patterns are used, which means that infrastructure automation is used in every stage of the pipeline when setting it up for production.
- *Evolutionary Design.* Microservices should be independently replaceable and upgradeable. As microservices are small sizewise, it is easier to replace or rewrite the service application. Services should be designed in a way that new features can be easily added while still supporting maintainability.

As this set includes common characteristics for microservice architecture, it is expected that all of these characteristics are not visible in each microservice-based architecture [1]. Based on the definition, microservice architecture usually consist of small independent services which communicate through lightweight protocols. As services are independent, the most suitable technologies can be used in the development of each microservice. An example of a microservice architecture in use can be seen in figure 2.1.

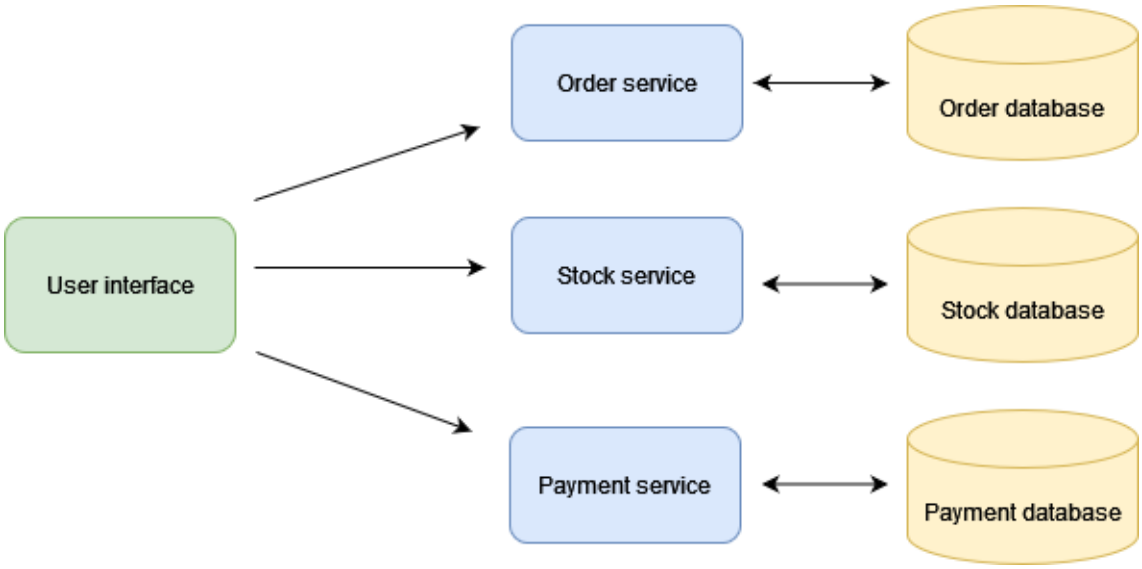

Figure 2.1. *An example of a simple microservice architecture*

2.2.1 Advantages of microservice architecture

As multiple organizations have adopted microservice architecture, there must be major advantages compared to already existing architectural patterns such as monoliths. The following list contains advantages that were mentioned in multiple sources including industry inquiry [10], gray literature reviews[11], and acknowledged gray literature [1] [12].

- *Scalability.* Easier scalability than monoliths.[13][12] In the microservice architecture, only services that require scaling due to higher load can be scaled separately. In the case of monoliths, the whole application has to be scaled even though a higher load affects only a singular part of an application. [12].
- *Organization structure.* As microservice is structured around a single business capability, shared responsibilities among teams are minimized, and team size for a microservice remains manageable [13] [12]. This might help to lower the cost to develop an application [10].
- *Deploy independently.* Compared to monoliths where small changes might require the whole application to be redeployed, microservice can be deployed without affecting the execution of other services [12] [13]. The possibility of independent deployment allows fast release cycles and quick feedback from modifications [13].
- *Diversity in technologies.* As microservices are independent of each other, the most suitable technologies can be chosen for each microservice. [10] [12] [11]

These advantages are all related to dividing an application into smaller services which are all self-contained for easier scaling, independent deployment, and the possibility to choose multiple technologies based on requirements. As microservices manage single-responsibility, organizations can be also structured to minimize cross-team development.

2.2.2 Disadvantages of microservice architecture

As for each architectural pattern, there are a set of disadvantages for the microservice architecture which must be taken into consideration. Disadvantages found in multiple sources including peer-reviewed literature [13] [10] [11], and acknowledged gray literature [12] are listed below.

- *Hard to learn.* Microservice architecture includes multiple different technologies which require an extensive amount of knowledge [12] [13]. This is why it either requires a lot of training to implement a microservice-based application or an already advanced developer [13].
- *Increased latency.* Microservice communication must be done over the network instead of in-process calls which increases latency [12] [10].
- *Data consistency harder to maintain.* As independent databases might be used in a microservice architecture, ensuring data consistency with transactions spanning multiple databases becomes complex. [10] [12] [11].

As seen in the listing, data consistency is one of the problems in microservices. Soldani et al. [11] found out in their work about "the pains and gains" of microservices, that data consistency and distributed transactions were seen as the most significant concerns about storage management in microservices. [11]. Also, Zhou et al. [10] discovered in their industrial inquiry that the database-per-service pattern was only practiced by 10% of the interviewees. According to the authors, problems with distributed transactions while ensuring data consistency caused practitioners to switch back to a single database. [10]

It is possible to avoid the complexity of data consistency by using a shared database among microservices, or by designing services in a way that there is no coordination between multiple microservices [4]. In these cases, coordination is not required, or database transactions can be used instead.

However, using a shared database causes coupling between services [1] which should be avoided for the previously mentioned independent deployment of microservices. A shared database also removes the possibility to use the most suitable database technologies for each microservice which was also listed as one of the advantages. Due to these disadvantages, developers may decide to use a database-per-service pattern even though it introduces possible problems with data consistency.

Also, dividing functionality into microservices is complicated [13] which might lead to setting microservice boundaries incorrectly [10]. There might also be situations where coordination between services is required even though boundaries are set wisely.

2.2.3 Coordination between multiple microservices

In case that shared database cannot be used, and services cannot be designed in a way that transaction objects reside within a single microservice and database, other ways to manage coordination and data consistency between services should be thought of.

As seen with the disadvantages of microservice architecture, data consistency between multiple services is hard to maintain. Strict consistency requires that after an update has been completed, the updated value will be returned for every subsequent access [14]. In microservice coordination, this is harder to obtain as coordination requires modifying resources in multiple databases.

To alleviate this problem, strict consistency between services can be relaxed to eventual consistency [11] as already visited with BASE semantics. Eventual consistency guarantees that if there are no new updates to the database, eventually it will return the last value updated by user [14].

With eventual consistency, each service could be updated as a sequence. In case of a failure in one of the participating services, services could be restored to the initial state with compensating transactions. As an updated service might still return to the initial state in case of failure, strict consistency cannot be promised. Even though eventual consistency alleviates problems with data consistency, it is not still easy to implement [11]. This is why design patterns to implement coordination with eventual consistency are required.

Strict consistency might be still required when working with sensitive data [11], which is why design patterns with strict consistency are also required. For this, distributed transaction protocols that enforce ACID principles could be used.

In this work, design patterns that can be used to manage the coordination between microservices are studied. For each pattern found in the literature, advantages and disadvantages are listed to easily understand trade-offs and possible use cases. Also, implementation methods found in the literature are studied to find the best ways to implement these patterns.

2.3 Related work

In this section, related literature for coordination between multiple services is discussed. Discussion includes a short description of the work done, and how our work will differ from theirs. After a comprehensive search, only two works [3] [4] that compare microservice coordination patterns were found.

Laigner et al. [3] use peer-reviewed articles, open-source repositories, and a survey

for developers to find out used practices for data management in microservices. Found coordination mechanisms are listed, but the advantages and disadvantages are only discussed briefly. Also, the implementation of these patterns is not discussed in this article. Our study will complement this study by discussing patterns more extensively, and by studying the current stage of research for coordination mechanisms.

Ntentos et al. [4] use quantitative method to analyze "grey literate" to find the most used data management practices in the industry. With this information, a reusable architectural design decision model is generated to help with architectural decision-making. As one subsection, coordination between services is discussed and a decision framework for selecting a suitable coordination pattern is proposed. The proposed patterns are discussed briefly and only high-level benefits and disadvantages are given for each. To implement these design patterns, it is necessary to know about the advantages and disadvantages of implementation also. In this sense, our mapping study will complement this study with a more comprehensive view of each design pattern.

3. METHODOLOGY

In this section, the process for a systematic mapping study is presented. A systematic mapping study is conducted based on the process defined by Petersen et al. [15] with an additional snowballing process defined by Wohlin [16].

In the remaining section, the goals for the mapping study and research questions are presented. Based on these, the search strategy is defined which is then used to find relevant articles to be used in the study.

3.1 Goal and research questions

The goal of this work is to map the current state of patterns and techniques for coordination between multiple services. For patterns, high-level design patterns and code-level implementation patterns are considered in this study. Also, an attempt is made to identify directions and opportunities for future research.

Based on set goals, research questions (RQs) for the mapping study are defined as follows.

RQ1 What design patterns could be used in coordination between multiple microservices?

RQ2 What are the advantages and disadvantages of extracted design patterns?

RQ3 How extracted architectural patterns can be implemented and what problems arise with the implementation?

RQ4 What are the advantages and disadvantages of the extracted implementation patterns?

The first research question is used to find out which design patterns have been used in research for coordination between multiple microservices. Design patterns are high-level solutions to frequently occurring problems that are written in a common format for easy interpretation [17]. With RQ1, we want to find out which patterns have been the most used in the context of microservice architecture. To a get better understanding of acquired patterns, the context for possible use cases should be mapped somehow. With RQ2, we aim to collect a list of advantages and disadvantages which can be used to help

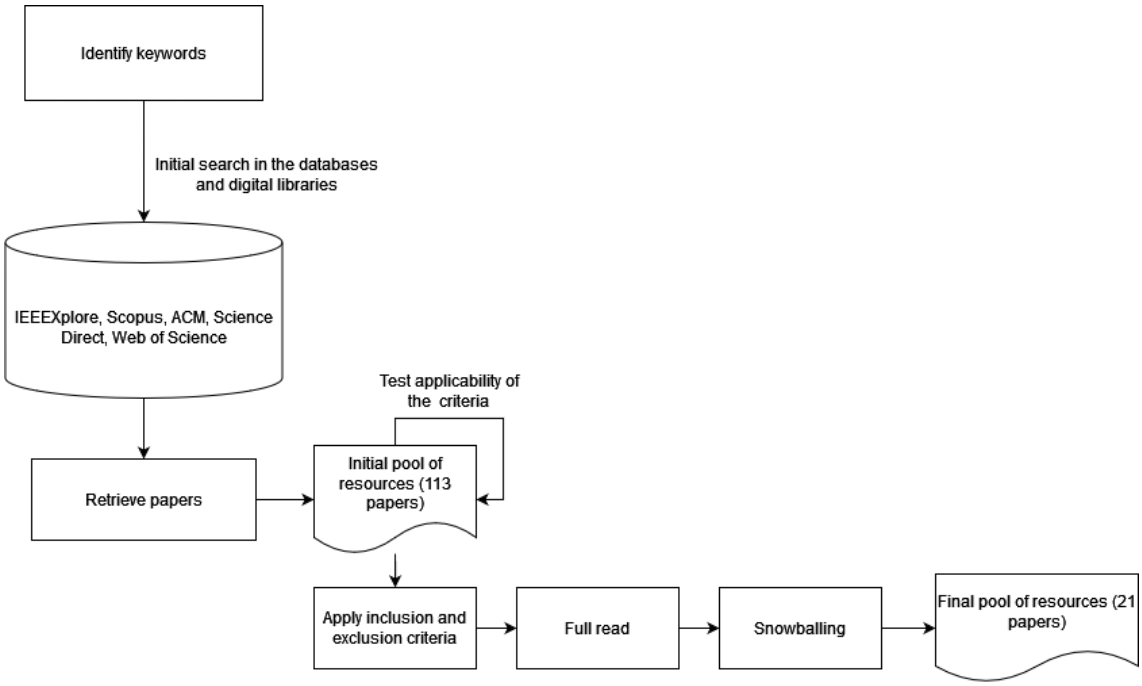

Figure 3.1. *An overview of the selection process*

use case detection for each pattern.

As design patterns are high-level solutions to frequently occurring problems, there is still room to implement these in multiple ways. With RQ3, we aimed to identify these implementation patterns related to findings from RQ1. To get a better understanding of the implementation patterns gathered with RQ3, the advantages and disadvantages of each should be listed. With RQ4, we aim to give a comprehensive listing of advantages and disadvantages for each implementation pattern. As a whole, research questions should give a comprehensive overview of the current state with patterns relating to coordination between multiple services and how to implement them when the context is taken into consideration.

3.2 Search strategy

The search strategy includes a selection of relevant bibliographic sources, the construction of the search string, and a definition of inclusion and exclusion criteria. With help of this strategy, we aim to find the most relevant articles related to our aforementioned goal. The overview of the selection process is represented in figure 3.1.

3.2.1 Bibliographic sources and search string

Relevant bibliographic sources were used to identify relevant resources. As suggested by Kitchenham et al. [18], bibliographic sources were chosen to include publishers' digital

libraries and two general indexing databases. The sources include *IEEEXplore Digital Library, ACM Digital Library, Science Direct, Web of Science, and Scopus* where the first 3 are digital libraries and the last two are general indexing databases.

To find relevant resources, a search string was constructed based on the goals of this work. The search string was divided into two distinct parts: the first part is used to describe the context *"microservice architecture"* and the second part is used to find articles related to the topic of interest *"coordination between multiple services"*. For the first part, multiple variations were used to find all articles referencing *microservices*. For the second part, as there is no singular established way to define coordination between multiple services, only the term *"transaction"* was used in the search string to find all possible variations including the wanted subject. With this term, it should be possible to find articles referring to this topic with different terminology such as *distributed transaction* and *transaction-less coordination*.

Finally, the following search string was used: **(microservice* OR "micro-service*" OR "micro service*") AND transact***. The search string was applied to fields *Abstract* and *Title* in the aforementioned databases and libraries. The symbol * is used in search terms to capture plural forms and verb conjugations.

3.2.2 Inclusion and exclusion criteria

To find relevant articles to this mapping study, a set of inclusion and exclusion criteria was constructed.

Based on our RQs, inclusion criteria for this mapping study were defined as follows:

IC1 Papers discussing advantages and disadvantages with design patterns for coordination between multiple services

IC2 Papers discussing implementation patterns or details for coordination between multiple services

IC3 Papers proposing new methods to implement coordination between multiple services.

For exclusion, the following criteria were set up:

EC1 The paper is not written in English

EC2 The paper is duplicated. In this case, the most recent copy is chosen

EC3 The paper is out of topic. In this case, the terminology might have been used in a different context

EC4 The paper is not peer-reviewed. However, non-peer-reviewed contributions are considered if their citations are considerably higher than the average of citations in

Selection process step	Number of papers
Extract papers from bibliographic sources	113
Filter by title and abstract	86 rejected
Filter by full reading	12 rejected
Backward and forward snowballing	6 added
Papers accepted	**21**

Table 3.1. *The selection process for the literature*

included peer-reviewed papers.

Exclusion criteria consist of generic criteria which can be used to easily filter articles not suitable to be added to the study. The assumption is made that most of the articles related to this topic are published in English which is why papers written in other languages are excluded.

3.2.3 Search and selection process

The search was conducted in January 2022 and a total of 113 unique articles were returned from the aforementioned set of databases and digital libraries when searching by title and abstract.

First, the applicability of constructed criteria was tested before applying inclusion and exclusion criteria to the rest of the papers. The test was conducted by selecting a subset of 10 random papers from retrieved papers.

After applicability testing, refined inclusion and exclusion criteria were applied to the title and abstract of retrieved papers. Each article was read by a singular author and the decision was made him independently. Of 113 initial articles, 27 were included after applying criteria to the title and abstract.

A full read was carried out with 27 papers included by title and abstract. To filter all relevant papers, inclusion and exclusion criteria were used. After this stage, 15 papers were chosen to participate in the mapping study.

In addition to the systematic mapping study process, backward and forward snowballing was performed. This step was carried out with 16 papers selected for the mapping study. For the backward snowballing process all referenced articles were considered and for forward snowballing all articles referencing selected articles were considered. After reviewing articles gathered from snowballing, five additional articles were added to the mapping study. Works by Richardson [SP20] and Newman [SP21] were added from gray literature as there were a notable amount of citations compared to other selected works.

The aforementioned process resulted in 21 works published between 2014 and 2021.

Selected papers include 20 unique works and a book chapter by Fan et al. [SP2] which can be seen as a simplified version of previously presented work [SP1] from the same authors. Results from each step of the process can be seen in the table 3.1.

4. RESULTS

In this section, the results of the systematic mapping study are presented. In section 4.1, found design patterns for coordination between multiple services are presented and discussed. Also, the advantages and disadvantages of previously found design patterns are discussed and collected in a table for easier comparison. To better understand the implementation of these patterns, in section 4.3 the most used implementation patterns found are presented for each design pattern. Also, the advantages and disadvantages observed in selected works are discussed.

For a start, common statistics for the final set of papers are presented. As shown in the previous section, 21 works were retrieved. Figure 4.1 shows the distribution for publication years where it can be concluded that the topic is relatively new for research as the first papers found to relate directly related to microservice implementation are from 2018. A single article from 2014 is also included in selected works, but it discusses coordination from a web service point of view. Even though the sample is small, an upward trend in publications can be seen for this research topic. Figure 4.2 shows the distribution for document types of primary studies where the main type is a conference paper and the second type is an article. This further demonstrates that this topic is new in the research community.

4.1 Design patterns for coordination between multiple services (RQ1 & RQ2)

This section presents established design patterns used to manage coordination between multiple microservices. The potential advantages and disadvantages of these patterns are also visited for easier comparison. In section 4.1.1, the traditional saga pattern, and its two variants are presented and discussed. The next section 4.1.2 includes a discussion about the Two-phase commit (2PC) pattern. Try-Cancel/Confirm (TCC) pattern is discussed in section 4.1.3.

In table 4.1, patterns discussed in the selected works and the frequency of mentions are gathered. General saga pattern was mentioned and discussed in eight works [SP3], [SP5], [SP9], [SP11], [SP15], [SP17], [SP20], [SP21], six works discussed saga choreography and orchestration discussed in [SP4], [SP10], [SP14], [SP18], [SP20], [SP21]

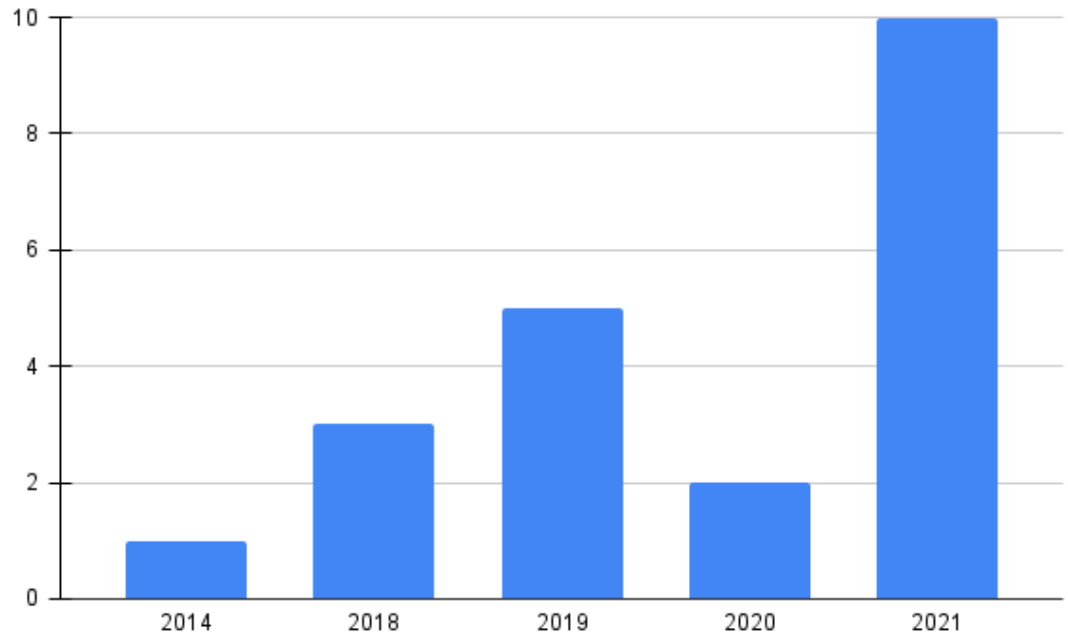

Figure 4.1. *Distribution of publication years*

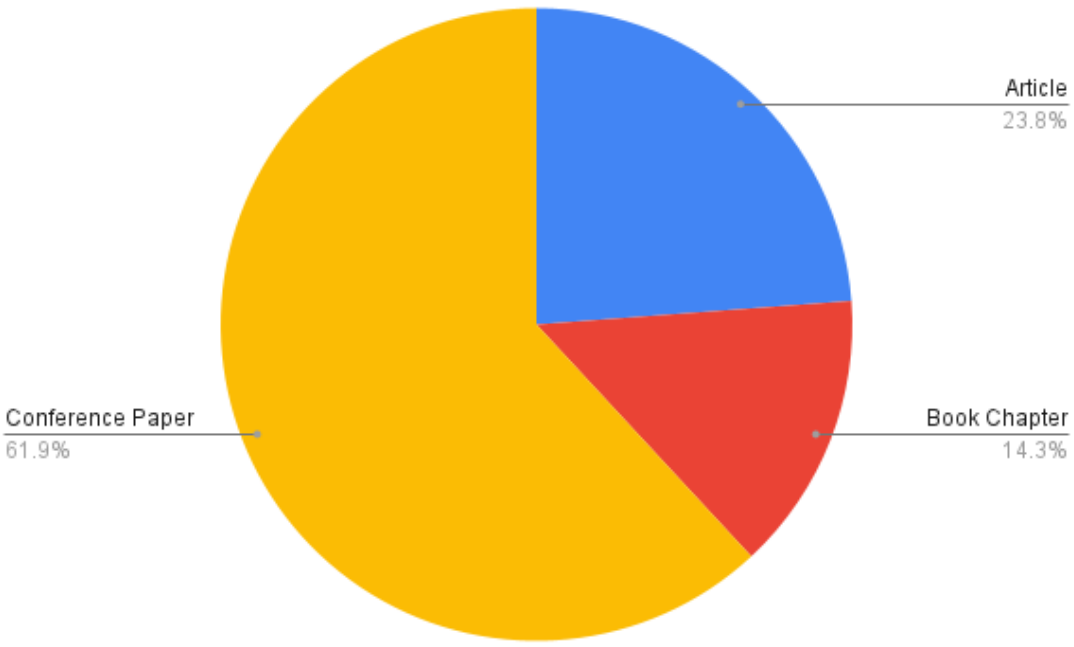

Figure 4.2. *Types of selected works*

Design pattern	# of studies
Two-phase commit (2PC)	13
Saga pattern	12
Try-confirm/cancel (TCC)	6
Other	5
Novel saga-based patterns	4
Novel approaches	4
Novel 2PC-based patterns	2

Table 4.1. *Patterns mentioned in selected works*

which of both were mentioned in all six articles.

Pattern 2PC was mentioned and discussed in 13 studies [SP1] [SP5] [SP6] [SP7] [SP8] [SP9] [SP11] [SP14] [SP15] [SP17] [SP18], [SP20] [SP21] and TCC in six studies [SP3], [SP6], [SP11], [SP12], [SP13], [SP19].

Five works discussed novel approaches for coordination based on the saga pattern. Novel 2PC-based patterns were discussed in two works by the same authors. Four works discussed novel approaches not based on any visited patterns. Other possibilities such as the usage of distributed databases and the usage of stream processors were discussed in 5 studies.

It is observed that the most prominent patterns discussed in the literature are the saga pattern and 2PC but this itself doesn't mean that these patterns are suitable to be used with microservices. It was noticed while reviewing the literature, that 2PC is usually given as an example of how transactions were implemented in distributed systems before microservice architecture. However, as availability is an important characteristic of microservices, traditional 2PC is not usually encouraged to be used. 2PC is further discussed in section 4.1.2.

Concept, Goal, Properties, Evolution, Reported Usage, Advantages, and Disadvantages template introduced by Taibi et al. [19] is used to discuss the design patterns.

4.1.1 Saga pattern

In this section, the saga pattern and its two variants, orchestration and choreography, are discussed. General advantages and disadvantages are applicable to both choreography and orchestration. Orchestration and choreography define the inner structure of how a saga manages coordination between services.

Saga pattern

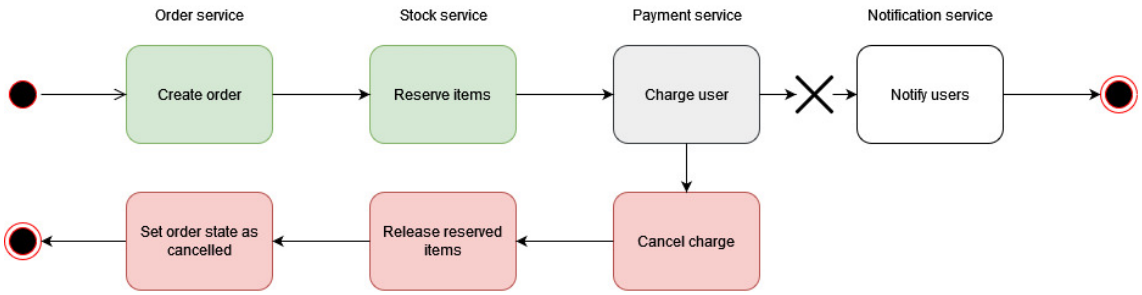

Figure 4.3. *A general flow of a saga pattern (Modified from source [20])*

Concept: Microservice architecture embraces the decentralized data management principle with a database-per-service pattern to improve loose coupling of the services and to enable the usage of different database technologies in the services. However, this removes the possibility to use database transactions for transactions involving multiple microservices. As a consequence, an additional pattern for the coordination of operations involving multiple microservices is required. In the selected works, the saga pattern appeared as a highly used pattern to implement coordination between services.

The saga pattern was introduced in 1987 by Garcia-Molina et al. [21] as a way to manage long-lived transactions, and later was rediscovered for managing coordination between microservices. With the saga pattern, a change that affects multiple services is separated into smaller microservice level sub-transactions which are run in sequence to reach the desired outcome. Each of these sub-transactions within a single microservice can be implemented with ACID guarantees. The objective of the saga pattern is either to successfully finish all sub-transactions participating or to cancel all operations if any of them fails. In case of a failure, already finished sub-transactions are compensated using reverse operation. The reverse operation should return the state of each participating service to the state before the saga was initiated or to a similar state. An example of a transition to a similar state is given in the example below.

An example of the general structure of the saga pattern including transactions and compensating transactions are shown in figure 4.3. In the example, there is a failure in the payment service which starts the compensation process of the saga. As failure happens in the payment service, subsequent notification service is not called. After compensating transactions have finished, the saga ends. In the case of this example, the order service returns to a similar state as it was before the saga execution by setting the order status to *cancel*. Stock service returns to the same state by removing the reservations.

The saga consists of a reversible, pivot, and repeatable sub-transactions. All reversible transactions happen before the pivot transaction, and all repeatable transactions happen after the pivot transaction. If failure happens at the latest in the pivot transaction, the saga is canceled. If the pivot transaction succeeds, the saga can be successfully performed as all following operations can be repeated until they succeed. [SP10]

Goal: The goal of the saga pattern is to provide a way to manage the coordination of services when operations span multiple services. As each performed step must be compensated, the state will be consistent eventually.

Properties: In the saga pattern, compensating transactions are related to recovering from business failures and not technical failures such as participating service which is not responding. [SP21]

For durability, the state of the saga could be persisted to logs in stable storage [SP17] [SP15]. These logs could be used in case of failure to continue the saga flow.

Evolution and reported usage: The saga pattern was first introduced in 1987 for long-lived transactions by Garcia-Molina et al. [21], and later reconsidered as a pattern for microservice architecture. Discussion of the general saga pattern for microservices was found in eight works [SP3], [SP5], [SP9], [SP11], [SP15], [SP17], [SP20], [SP21] where general advantages for the saga pattern was discussed.

General **advantages** for the saga pattern are:

- *Higher availability.* This is achieved as a trade-off for losing strict consistency as stated by CAP theorem [SP10]. To manage this, ACID principles for the saga workflows must be relaxed which means introducing BASE semantics as an alternative to follow [SP15]. Also, as an isolation property is broken due to dividing coordination into microservice level sub-transactions which can be visible during a saga execution, availability is improved [SP10].
- *Improved performance.* In the experiments conducted by Xue et al. [SP11], it was noted that when coordination is moved out of the local transactions, performance is improved. De Heus et al. [SP5] mention improved performance due to separating coordination into smaller sub-transactions. Also, throughput is improved over 2PC as resource locks are required only for local transactions [SP17] [SP11].
- *Scalability.* As stated by Stefanko et al. [SP15], relaxed ACID semantics also improve the scalability.

Multiple **disadvantages** was also mentioned for the general saga pattern:

- *Missing isolation property.* Intermediate states might be visible to other transactions during the saga execution [SP15] [SP20]. As isolation is missing, a dirty read is possible when transaction *T2* reads a resource modified by a transaction *T1*, and later transaction *T1* needs to be compensated. In this case, *T2* will have an inconsistent state for the resource. [SP11] [SP20]
- *Risk of inconsistent state.* Might be possible in cases where other participating services have committed while at least one aborts due to service failure [SP11]. Stefanko et al. [SP15] also mention possible participant failures as a problem that

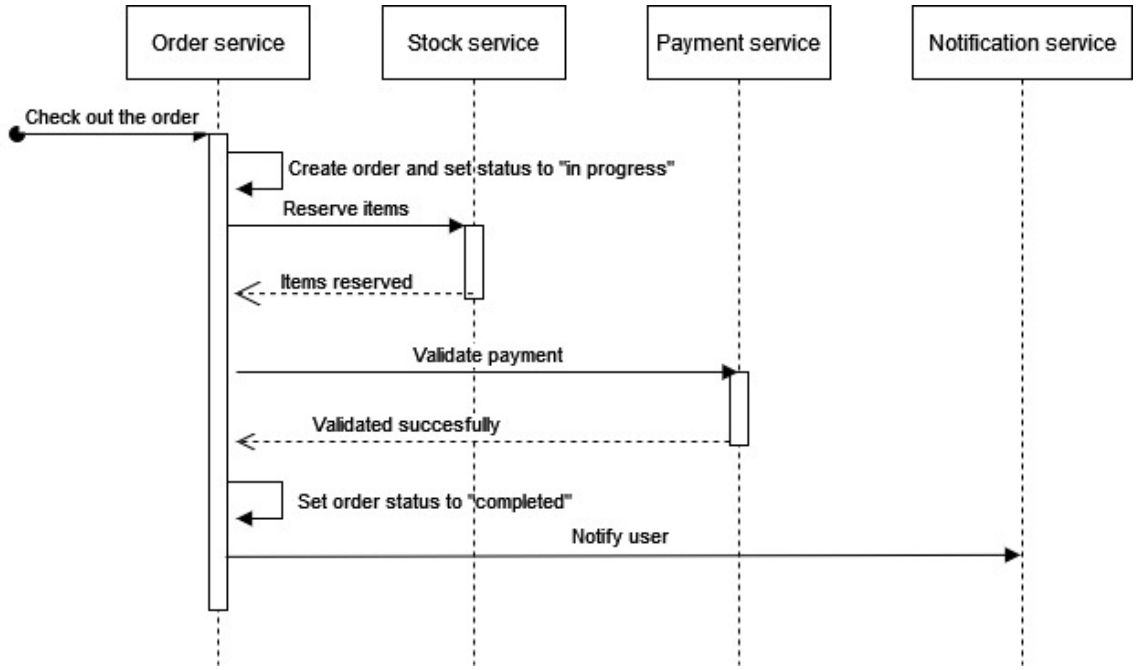

Figure 4.4. *An example of a saga orchestrator*

needs to be solved.

- *Only eventually consistent.* For higher availability, only strict consistency is replaced with eventual consistency. Eventual consistency promises that the state will eventually become consistent after the saga has been completed. [SP15] Eventual consistency is mentioned by [SP10] [SP20] is only provided. [SP9]

The saga pattern is usually divided based on coordination styles which are orchestration and choreography. First, the saga orchestrator is discussed.

Saga orchestrator

Concept: In the saga orchestrator pattern, coordination is managed by the central orchestrator. This means that participating microservice receives an initiating request from the orchestrator and after completing the required tasks, sends a response to the orchestrator about a successful or failed operation. An orchestrator can either be one of the participants, which could mean that the first service to participate also manages orchestration. Orchestration could be also managed in a separate service, where the only task is to manage all sagas of the system.

An example of a saga orchestrator, where the participant acts also as an orchestrator is shown in figure 4.4. In the example, the order service manages the flow of the saga and calls other microservices in sequential order.

Properties: Even though the communication method between the orchestrator and participants is not outlined, according to Newman request/reply seems to be the preferred solution [SP21]. Also, Richardson [SP20] states that a command/reply interaction pattern

should be used when implementing saga with orchestration.

Reported usage: Six works mentioned and discussed saga orchestration [SP4], [SP10], [SP14], [SP18], [SP20] [SP21]. In these works, the following advantages and disadvantages were observed.

Advantages for saga orchestration are:

- *Low coupling between services.* Services are loosely coupled as only the orchestrator calls them and services never call others [SP10]. This is also mentioned as an advantage in three other works. [SP18] [SP14] [SP20] However, there is still high coupling between participating services and the orchestrator [SP21].
- *More maintainable.* As saga-related logic is in one place, maintaining becomes easier [SP10]. In an experiment conducted by Rubrabhatla [SP4] it was remarked that when the saga becomes bigger, management is easier with the orchestrated approach as logic resides in one place. Also, it might not be necessary to modify service behavior when integrating it to a saga flow [SP18].
- *Easier to understand.* As logic resides in one place, it is easier to understand the flow of the saga without having to look at multiple places [SP21].

and **disadvantages**:

- *More messages required for coordination.* Two works [SP18] [SP4] mentioned that the orchestrated approach requires more messages for coordination than a similar solution with choreography as all communication must be transported through the central orchestrator. As seen in an experiment conducted by Rubradhatla [SP4], additional messages decreased the performance of the orchestrated solution in comparison to the choreographed one. [SP4]
- *Single point of failure.* According to Röwekamp et al. [SP10], a single point of failure is a trade-off for loosely coupled architecture and better maintainability. Also, Limon et al. [SP14] mention a single point of failure as a problem that also leads to the centralization of traffic.
- *Risk of centralization of logic.* There is a risk that too much logic will reside in the coordinator as it might be easy to relocate logic that should be managed by services into the orchestrator. However, this causes services to become *dump* components directed by *smart* orchestrator. [SP21] [SP20] According to Newman [SP21], this could be avoided by using participating service also as a coordinator which distributes saga coordination logic to multiple different places. Also, if the coordinator only manages the flow of the sagas with call sequencing, this problem should not occur [SP20].

Saga choreography

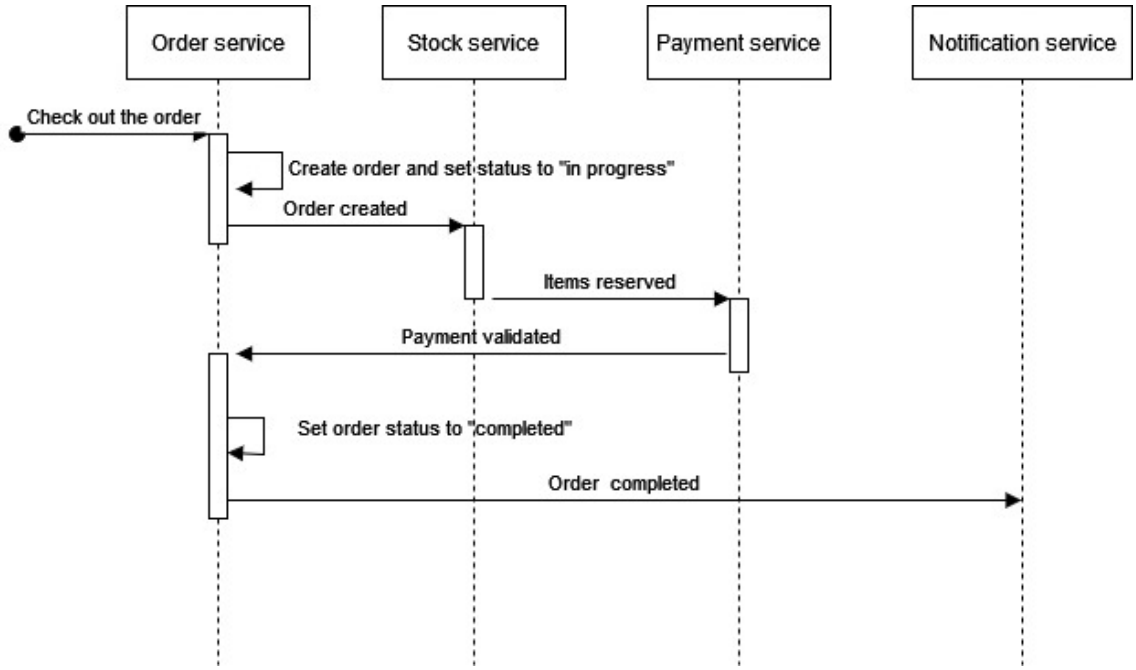

Figure 4.5. *An example of a saga choreography*

Concept: In saga choreography, coordination is managed as collaboration with all participating services. This means that each service participates in a saga by catching an event sent by a previous service to initiate required actions, and then by sending an event for the successful or failed operation. An event for a successful operation is captured by a subsequent service and an event for failed operation is captured by a previous service to perform compensating actions.

An example of a saga choreography is shown in figure 4.5. In the example, it can be seen that each service receives an event to initiate its action, and after finishing the action, a message about a successful or failed operation is sent. This event is then caught by subsequent service to initiate the next step.

Properties: Even though interaction style is not enforced, an event-driven approach is preferred with a choreographed saga. A message broker could be used to send and deliver these events reliably. [SP21]

As the state of the saga is distributed amongst participants, *correlation ID* is required to be passed with events to connect the state. For example, this identification can be used in the case of compensating transactions to find a correct resource [SP21] [SP20]. In the previous example, the order ID could be sent as a correlation ID to identify the state associated with the saga.

Reported usage: Six works mentioned and discussed saga choreography [SP4], [SP10], [SP14], [SP18], [SP20], [SP21]. In these works, the following advantages and disadvantages were observed.

Mentioned saga choreography **advantages** are:

- *Fewer messages required for coordination.* In an experiment conducted by Rudrabhatla [SP4], the response time of the choreographed solution was lower than that of the orchestrated solution. According to the author, this was due to fewer messages required to coordinate the choreographed solution [SP4].
- *Low coupling between services.* There is no direct knowledge about other services as each service only needs to respond to certain events and send an event after any modifications made to state [SP21] [SP20] [SP14]. However, it can also be argued that choreography is tightly coupled as visited in the disadvantages list.

and following **disadvantages** were mentioned:

- *More complex.* Two works [SP21] [SP20] mentioned that it might be hard to understand the flow of the saga when the coordination logic is distributed amongst participating services. Similar results were seen in an experiment conducted by Rubrabhatla [SP4] where it was noted that the choreographed approach was more complex to implement when required events increased.
- *Risk of tight coupling.* There is a risk of tight coupling when it is required to modify an event sent by participating service. In these cases, the sender and receiver need to be updated in cooperation to work. [SP20] Due to the tight coupling, it might become complicated to change a saga that uses a choreographed approach [SP10]. Tight coupling between services was also mentioned by one other work [SP18].

As seen in the above chapter, there are fundamental differences between orchestrated and choreographed implementation of the saga pattern. Even though both of the coordination methods can be used when implementing saga, preferred solutions can be seen in the works for different situations. Newman [SP21] mentions that it might be easier to use orchestrated approach if a team manages each service inside the saga flow. However, if services participating in a saga are managed by multiple teams, it might be easier to use the choreographed solution [SP21]. Rubrabhatla [SP4] noticed with experiments that for sagas which consist of a small number of services, choreography can be useful. This is especially the case when there are strict performance requirements. However, when the saga flow is more complex, the orchestrated approach might become preferable [SP4] [SP20]. According to Röwekamp et al. [SP10], the orchestrated approach seems to be the preferred solution in the industry as downtimes related to the central point of failure are uncommon in comparison to the problems caused by using choreographed solution [SP10].

One or more of the aforementioned issues of the saga pattern could be solved using novel saga-based patterns demonstrated in the upcoming section 4.2.1. Certain implementa-

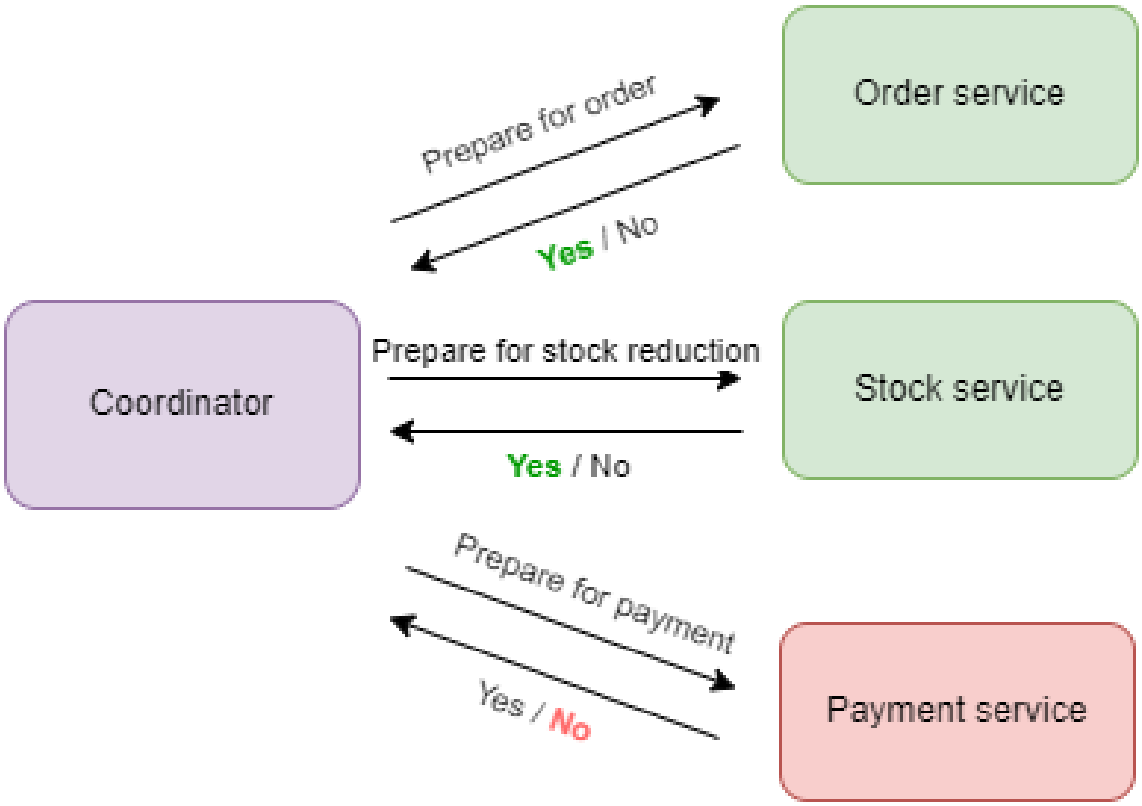

Figure 4.6. *The first phase of a two-phase commit (Modified from source [20])*

tion patterns could be also used to resolve one or more of the aforementioned issues which are further discussed in section 4.3.

4.1.2 Two-phase commit

Concept: In the first phase, the coordinator sends *PREPARE* message to all participating services to prepare for a transaction. Service returns either a *YES* or a *NO* depending on if it can participate in the transaction. In the second phase, the coordinator sends *COMMIT* message if all participating services have returned *YES* vote. If even one service cannot participate in the transaction, the coordinator sends *ABORT* message to each service. [22]

The first phase of the two-phase commit is shown in figure 4.6, and the second phase is shown in figure 4.7. In this example, the payment service cannot participate in the transaction and returns no vote. This means that the coordinator sends *ABORT* message to participants that returned *YES* vote as resources reserved for the transaction should be released. It is not necessary to send *ABORT* message to participants that returned *NO* vote [22]. If all participating services return a *YES* vote, the coordinator sends a *COMMIT* message to each service, and services return acknowledgment after a successful operation. Acknowledgment is sent for the coordinator to inform that participants are aware of the end result and have acted properly [22]. After the coordinator has received all *DONE*

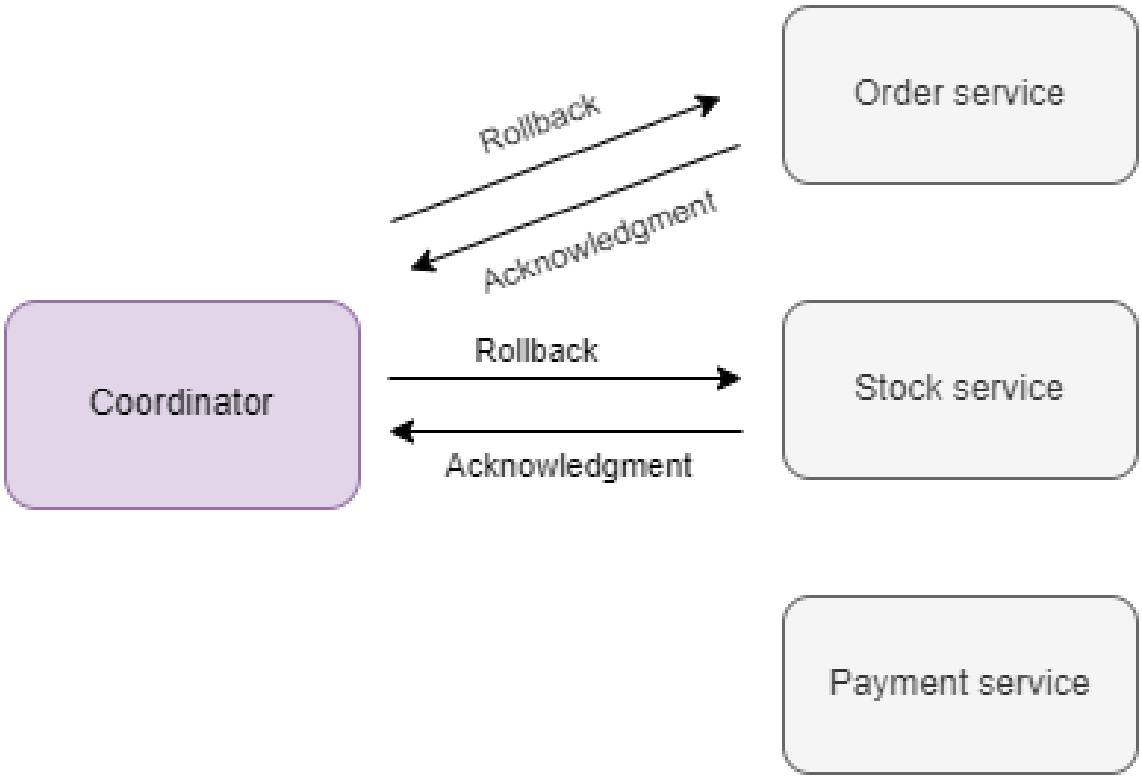

Figure 4.7. *The second phase of a two-phase commit in case of rollback (Modified from source [20])*

messages, it can forget the transaction.

Goal: The goal of this pattern is to manage distributed transactions with the central coordinator consistently and atomically. No commits are made to databases before the coordinator is sure that each participating service can commit.

Properties: In the first phase, each service makes the necessary preparations so it can guarantee commitment. This means that each service might lock necessary resources and wait until the coordinator sends either *COMMIT* or *ABORT* message to release them.

Logs are written to persistent storage during the protocol in the coordinator and all participating services to ensure failure handling [22]. The coordinator must write three log entries which include *start-transaction* log before sending the *PREPARE* messages, *commit* log before sending *COMMIT* messages, and *Done* log after all *DONE* messages are received. After *commit* log is written, a transaction can be considered to be committed. Logs *start-transaction* and *commit* should be force-written which means that they must be synchronously written to persistent storage before sending messages to participants. [23]

Each participant must force-write two logs called *prepare* before sending *YES* vote and *committed* before sending *DONE* message. After *prepare* log is written, a participant can not unilaterally abort the transaction and must wait for the decision from the coordinator.

[23]

In the prepare-phase after a participant has persisted modifications to be made by the transaction, a participant can be considered to be *prepared*. Each participating service must be in the prepared state before any of them should commit. If this property is not enforced, 2PC could lead to non-atomic results when one participant commits while there is no guarantee that *non-prepared* participant can restore required modifications in case of service failure. [23]

After a participant has sent *YES* vote, it must wait for a decision from the coordinator. As participants cannot rollback without a permit from the coordinator, a failure of the coordinator could cause prolonged locking of resources. Blocking cannot be avoided by transaction protocols that guarantee atomicity. [23]

Reported usage: Thirteen works mentioned or used 2PC pattern [SP1], [SP5], [SP6], [SP7], [SP8], [SP9], [SP11], [SP14], [SP15], [SP17], [SP18], [SP20], [SP21] and following advantages of 2PC were observed:

- *Strict consistency.* In certain sectors such as e-finance or e-commerce, strict consistency is desired which requires the usage of protocols such as 2PC [SP1].
- *Full ACID guarantees*. According to de Heus et al. [SP5], ACID properties are guaranteed as a trade-off for performance decrease. Also, three other works [SP15] [SP9] [SP6] mention ACID guarantees when discussing 2PC.
- *Strictly serializable.* As a trade-off for decreased performance, 2PC provides highest isolation level for transactions [SP5]. As stated by Fan et al. [SP1], strict serializability is preferred by developers in traditional distributed databases which can be acquired with 2PC.

As can be seen, there are not many mentions of the advantages of traditional 2PC in the literature related to microservices. Most of the aforementioned advantages were related generally to this pattern and not especially related to implementing 2PC in a microservice architecture.

Below is a list of **disadvantages** which might make 2PC not suitable for microservice architecture as quality attributes of microservices might be at risk:

- *Modern technologies might not support XA-standard*. A preferred standard for distributed transaction management is X/Open XA, which must be supported by each participating database and message broker to have the possibility to use a two-phase commit protocol [SP20]. However, modern technologies such as MongoDB or RabbitMQ do not support this standard [SP20]. As stated by Knoche et al. [SP6], the missing support of distributed transactions with modern technologies might cause developers to switch to *eventual consistency* protocols. Even when suit-

able technologies are used, application-level code might be required to implement functionalities such as *prepare* and *abort*, which requires a profound knowledge of database concepts [SP9].

- *Reduced availability.* All participating services must be available at the same time, which reduces the whole availability of the process as the overall availability is the product of the availability of participating services. [SP20]
- *Decreased performance.* As locks are held for the whole duration of the 2PC process, performance is limited for high-concurrency and high-scale systems such as microservices where the possibility for transaction conflicts is increased [SP1]. This happens as transactions must lock resources for the whole duration of the protocol and block other transactions from using the same resources [SP1]. According to de Heus et al. [SP5], the protocol trades-off performance for strict atomicity with its blocking process. In an experiment conducted by Fan et al. [SP1], a dramatic decrease in throughput can be seen when contention for the same resources increases as the transaction must block resources for the whole duration of the process. Similarly, when contention for the same resources increases, the latency of 2PC increases dramatically due to the blocking nature of the protocol as seen in the experiment conducted by Fan et al. [SP1]. In the experiment conducted by Xue et al. [SP11] total request times were measured for transactions managed by 2PC, TCC, and Saga pattern. It was seen that composition using 2PC had twice as long total request time compared to other management methods, which according to Xue et al. [SP11] was due to additional waiting caused by locked resources. Two other works [SP8] [SP21] also mention increased latency as a problem caused by locking resources for the whole duration of the transaction. Also, committing rates dropped significantly in an experiment conducted by Fan et al. [SP1] when contention increased. This was also due to blocking caused by locks held by other ongoing transactions [SP1].

Even though there are problems with traditional 2PC which might make it non-suitable for a microservice architecture in most cases, there is still a need for highly consistent transaction patterns with ACID properties [SP1]. Improvements based on 2PC patterns are proposed and demonstrated in section 4.2.2.

4.1.3 Try-Cancel/Confirm

Concept: Instead of implementing rollback and commit, Try-Cancel/Confirm implements cancel and confirm. The pattern has a similar structure as other two-phase commit protocols where in the first phase participants are asked to try to participate in the transaction. If each service can participate, confirm-message is sent to each to confirm the transaction.

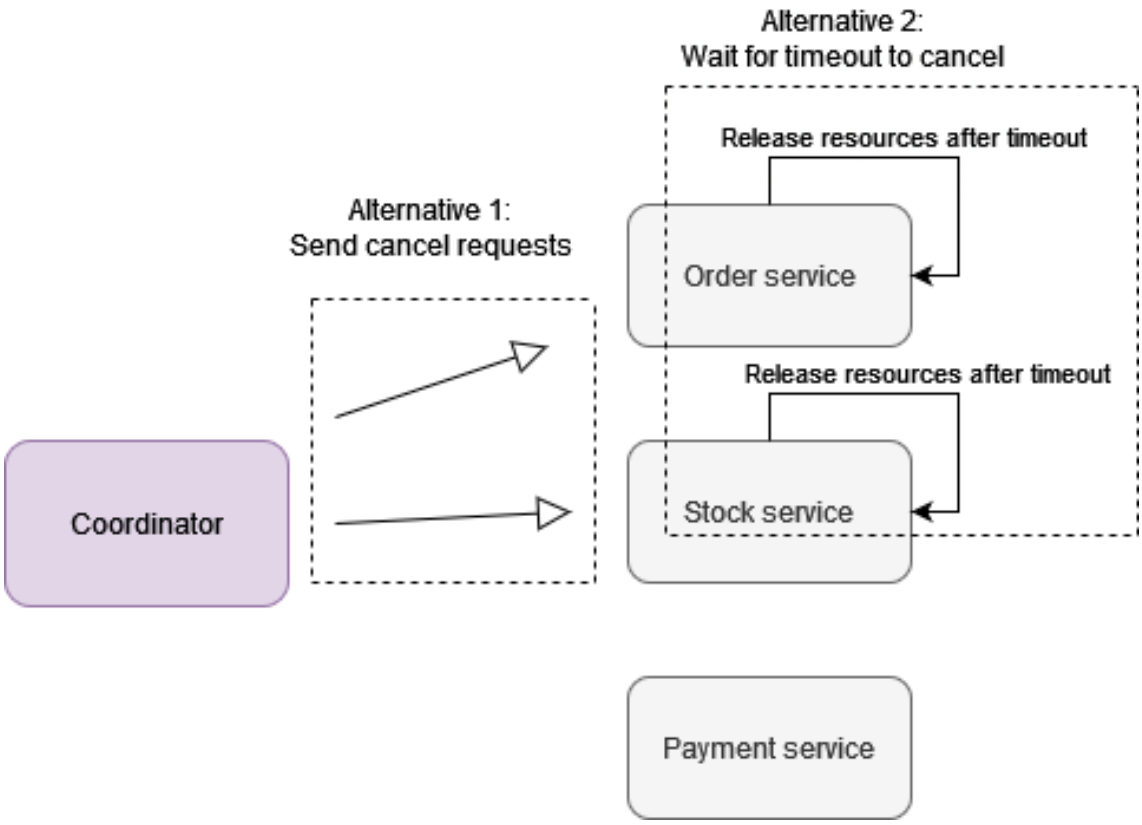

Figure 4.8. *Try-confirm/cancel patterns alternative ways to cancel*

The pattern differs from 2PC in the way that participating services manage their resources. Instead of locking resources with database locks, intermediate states are used instead for reservation. While outlining the protocol in the work by Pardon et al. [SP19], the assumption is made that each participating service must be able to reserve its resources for coordination. As each service sets intermediate states using short-lived ACID transactions at service-level [24], isolation property is removed from the transaction which increases concurrency related to 2PC [SP19]. When resources are reserved with intermediate states in the try-phase, a timeout is also set for them. If the reservation times out, services will cancel the reservation without a need for input from the coordinator. As participating services can perform cancellation with a timeout, a possible anomaly where at least one service cancels while others confirm might occur [SP19]. As participating transactions are not managed inside one global ACID transaction, each cancel requires the usage of a reversible transaction which should restore the state back to the state before the transaction started [SP13].

The trying phase is similar as seen in figure 4.6 where the coordinator sends a try-request to participating services, and services return a yes-vote if they can participate. As a difference, each participating service sets an expiration time for reservation which can be used to release the resources as seen in figure 4.8 if no confirm-message is sent by the coordinator. As an alternative to a timeout, the coordinator can also send a cancel-

message to participating services.

Goal: The goal of this pattern is to offer atomic transactions[1] for multiple services with a simple rollback.

Properties: Participating service might reserve resources for the duration of the transaction but the coordinator can cancel reservations before timeout if it is known that all participants cannot participate. However, as database locks are not used, it is possible to view resources before the transaction has finished. Resources are released at the latest when reservations time out.

Reported usage: The usage of TCC was discussed in six studies [SP3] [SP6] [SP11] [SP13] [SP19] [SP12]. Experiments including TCC were carried out by Xue et al. [SP11]. Ramirez et al. [SP3] implemented an empirical study on microservice development, where TCC was proposed as one of the patterns to maintain data consistency across microservices. Frosini et al. [SP12] compared their proposed approach with multiple other methods including TCC.

The following **advantages** were observed in four works:

- *Easy implementation of rollback.* As each participating service sets timeout which will cancel reserved resources automatically, no central rollback from the coordinator is required [SP19]. Timeouts are also enforced for TCC in an experiment conducted by Xue et al [SP11]. Xie et al. [SP13] also mention easy rollbacks as an advantage but also mention that the usage of reversible transactions might require a lot of developing effort.
- *Can process other tasks while waiting for a command from the coordinator.* In the experiment concluded by Xue et al. [SP11], implementation with TCC was twice as fast to finish the experiment compared to 2PC-based implementation. According to the authors, this was due to the possibility to respond to requests concurrently [SP11].
- *Atomicity guaranteed*. As the coordinator waits for a decision from each participant, all reservations are either committed or compensated [SP19]. This is also mentioned in the work by Frosini et al. [SP12]. A specific case where atomicity might not be guaranteed is discussed in the disadvantages listing.

However, following **disadvantages** for the pattern were observed:

- *Resource reservation time might be long.* As there should be enough time for all services to try and confirm, resources might be reserved for a long time [SP13]. According to Pardon et al. [SP19], the pattern uses intermediate states instead of database locks which makes it possible to view reserved resources while a transac-

[1] Exception for atomicity reviewed in the disadvantages listing

tion is ongoing. In some cases, this makes it possible to modify the same resources concurrently[2] [SP11]. However, with some resources concurrency is not possible which might cause poor performance and long waiting times [SP13]. Pardon et al. [SP19] also mention that resources must be compatible with the pattern to work properly [3].

- *Atomicity can not be preserved in a specific case.* There is a possibility that after starts sending confirm-messages to participants, one of the participants might timeout before it can commit. In this case, atomicity can not be guaranteed as one service cancels a reservation while others commit. In these cases, human intervention might be required to fix the state. [SP19]
- *Missing isolation property*. As intermediate states are used instead of database locking, resources modified by the ongoing transaction are visible to others [SP19]. Missing isolation is also mentioned by one other work when comparing their proposal with existing solutions such as TCC [SP12].

4.1.4 Other solutions

This section includes a discussion about other solutions found in the selected works. First, stream processing systems and how those could be used to implement coordination between services in the future are discussed. Then, distributed database systems are shortly discussed as an alternative to the previously mentioned patterns.

Stream processing systems

Katsifodimos et al. [SP9] argue that dataflow engines could be used as a base for data management and stream processing in event-driven microservices.

Even though stream processing systems have been known as analytic engines, modern solutions also support consistency for the distributed state which is required for microservices. Consistency is guaranteed with exactly-once-state processing guarantees. Even though consistency can be guaranteed, requirements for microservice architecture such as transactions, and query-able state cannot yet be achieved by stream processors. According to the authors, these deficiencies must still be solved for stream processing systems for them to be reliable data management solutions for microservices.[SP9]

Distributed databases

Distributed online transaction processing systems such as VoltDB and H-Store and other distributed databases such as Spanner are also discussed in multiple selected works. As an advantage, scalability, consistency, and the possibility to query global state are

[2] E.g. a stock service could freeze items with reservation. This makes it possible for other services to also freeze the same item if there is enough stock which enables concurrency

[3] Pardon et al. [SP19] uses airline seat reservation system as an example of reservable resources.

mentioned [SP9]. Even though distributed OLTP systems partition their data, at least H-Store can partition data in a way that most of the transactions are performed in a single partition [SP1].

As a disadvantage for distributed databases, it is required to persist the state of all microservices in a single distributed database to achieve consistent transactions [SP9] [SP2] [SP8]. Also, these systems require the code to run transactions to be persisted as a stored procedure. This might bring problems as transaction logic is usually inseparable from microservices [SP9].

4.2 Novel design patterns for transactions spanning multiple services (RQ1 & RQ2)

In this section, novel solutions found in the selected works are discussed. First, novel solutions include patterns based on the saga pattern which are discussed in section 4.2.1. After this, section 4.2.2 includes a discussion about 2PC-based solutions. Lastly, solutions that are not based on previously visited patterns are discussed in section 4.2.3.

4.2.1 Novel saga-based patterns

Novel patterns based on saga pattern were reported in four studies [SP10], [SP11], [SP14], [SP18]. Proposed novel patterns aim to improve the shortcomings of the traditional saga pattern. Improvements over traditional the saga pattern, and advantages and disadvantages of novel saga-based patterns are collected in table 4.2.

Petri Net Saga

Röwekamp et al. [SP10] remarked that concurrency within saga is hard to implement with current solutions as complexity increases rapidly when parallel steps are added. To alleviate the complexity, they have proposed a solution where the saga is modeled using Petri net formalisms, with the possibility to use a graphical representation of the saga. According to the authors, this solution should add better concurrency handling and transparent execution context.

In their proposal, two different types of Petri Net formalisms called Place and Transition (P/T) nets and Reference nets are used. P/T nets use places and transitions to move from one state to another via predefined steps. In the context of sagas spanning multiple microservices, the place is mapped to a service participating in the saga, and transitions are mapped to messages which are used to initiate functionality in the next participating service. P/T net is then transformed into reference net, which is the second Petri net formalism used, to include necessary properties such as failure handling and possible rollbacks. As there are concepts specific to the saga pattern, such as pivot-transaction,

general solutions for transformation to reference net could not be used.

With the P/T net, a saga can be modeled sufficiently and with lowered complexity while still having high readability. With the implementation created in the first phase, error handling and compensating actions can be implicitly deduced while transforming into reference net format. As the Reference net is a higher-level formalism, it is possible to model failure cases within it. Transformation to reference nets is also required as generated saga is then run in RENEW-simulator[4] which only supports this format. RENEW-simulator is written with Java which restricts the proposed solution to Java programming language. Also, the saga needs to be written with Petri net formalism using *Petri Net Modeling Language*[5] (PNML) which is not generally used technique amongst developers.

An example of a bookstore that uses presented Petri Net Sagas is given. This solution is built with *Spring*[6] as the backend framework, and *Eventuate Tram Sagas* [7] to handle saga execution. With this example, they concluded that concurrency handling can be improved with the possibility of parallelizing normal step-by-step processes inside a singular saga which means that there is a possibility to run tasks inside multiple participating microservices at the same time. However, there cannot be data passing between concurrently run sub-transactions which reduces possibilities for parallel processes as usually data is passed to subsequent the participant in the saga.

Single-head-multiple-tails & Single-tail

Xue et al. [SP11] remarked that there are unsolved problems with the saga pattern including the possibility to view incomplete data, and insufficient management for communication and node failures. To fix these problems, they proposed new composition mechanisms for saga choreography based on data dependencies between services to reach a consensus in run-time.

Based on structures found in data dependency analysis, namely sequence, fork and join, two composition patterns called Single-tail (ST) composition and Single-head-multiple-tails (SHMT) composition are defined. In these compositions, the head is a service that only calls other services, and the tail is a service that only receives calls from other services. Defined patterns can be found in any microservice compositions by evaluating data dependencies between services when each service is responsible for its own data.

In ST composition, other services but tail handle their tasks and sends information to succeeding services. After the tail service has received a request, it handles its task and sends a confirmation to other services if the task has succeeded. ST composition can reach consensus under conditions that confirmation can be fetched from tail service and

[4] Renew-simulator http://www.renew.de/
[5] Petri net modeling language https://www.pnml.org/
[6] Spring-framework https://spring.io/
[7] Eventuate Tram Sagas https://github.com/eventuate-tram/eventuate-tram-sagas

proper timeouts are set for services. Timeouts are used for compensation in case any step of composition fails or a confirmation message is not sent by the tail service.

In SHMT, each service except tail handles its task and sends information to succeeding services. The tail service also handles its task but then reports the state to the confirmation service. One of the participants works as a confirmation service to collect information from tail services about a successful or failed operation and after receiving the state from each tail service sends a confirmation to the whole composition. Similarly to ST composition, consensus can be reached if timeouts are set for each participating service and confirmation can be fetched from the confirmation service.

Both of these patterns use resource reservation to make incomplete data isolated from other processes. Also, required information about the request is persisted before each task. It is proven in the article that these composition mechanisms can reach a consensus between services in run-time if previously mentioned conditions are met. Timeouts are used in services to restore the state with compensation in case of service or communication failures and when services cannot query the state from the tail service or confirmation service. For these patterns to work, it is required that the state of the saga can be somehow fetched from confirmation or tail service to release resources in participating services. Other errors, such as errors related to business logic boundaries, are presumed to be handled correctly by services.

Two experiments using public datasets are conducted to compare composition patterns presented in this paper with TCC and 2PC patterns. The first experiment uses SHMT composition and the second experiment uses ST composition to compare patterns using total request time and resource reservation time for different sets of compensated and finished transactions. With experiment results, it can be seen that these novel composition patterns can effectively reach consensus with a performance comparable to or faster than centrally coordinated TCC.

SagaMAS

Limon et al. [SP14] argued that missing higher-level concept of transaction in the saga pattern causes problems that need to be solved with additional implementation patterns such as Event Sourcing (ES) and Command and Query Responsibility Segregation (CQRS). These implementation patterns themselves bring more complexity to the system and possibly a new set of problems to handle such as centralized points of traffic in case of ES.

To lessen complexity by integrating higher-level transaction concepts into sagas, they have visioned a novel solution called SagaMAS which includes Multi-Agent System (MAS) layer integrated as part of a microservice system to handle saga transactions. In the proposed model, saga-specific logic is relocated to the agent layer which enables independence between microservices. In the agent layer, each microservice has its own agent,

which manages the state of the microservice as part of a larger transaction. This means that the agent calls the necessary functions from the microservice as part of the transaction and, if necessary, compensates for calls that have already been made in case of errors. In addition, the agent persists the state of the transaction in case of agent layer errors, so that the transaction can be returned to, tried again, or compensated. In the proposed model, saga starts with the user calling microservice which then calls its own agent to initiate the transaction. This is the only place where saga-related logic is required in the microservice layer as after this MAS layer takes care of the rest of the transaction flow and necessary calls to microservices. For messaging between agents and microservices, REST-based microservices are chosen for lower coupling.

For evaluation of the visioned model, the first phase of the Prometheus methodology called *System specification phase* is done, and the last two phases are left for future work. The system specification phase includes the identification of goals and basic functionalities of the visioned system. Even though architectural and detailed design phases of Prometheus are out of the scope of this paper, some implementation patterns are chosen for SagaMAS including JaCaMo framework[8], which consists of Jason, CArtAgO and MOISE to handle the implementation of MAS layer.

Clustered orchestrator

Malyuga et al. [SP18] argue that instead of creating an own coordinator for each new saga, a central coordinator could be used which handles the coordination of each saga in the system to lower the complexity of creating a new coordinator service for each individual saga. However, a failure of the central coordinator could cause major problems with saga flows as only it can handle the orchestration. To help with failure tolerance, they have proposed the inclusion of an active-passive pattern to saga orchestration with one central coordinator.

Novel pattern includes cluster orchestrator based on *active-passive availability* pattern which consists of active and passive nodes for better fault-tolerance. Active node handles saga flows and replicates data to passive nodes. In case of an active node failure, a passive node can take its place and continue the coordination without losing any data. In addition to cluster orchestrator, the paper proposes the usage of RESTful microservices which makes it possible to handle transactions without adding additional logic to microservices. As an active node might fail before the response is received via HTTP request, each participating endpoint must be implemented as idempotent for the system to work correctly. This enables passive nodes to continue unfinished Saga without causing any inconsistencies in participating services even in cases where services have done modifications and could not send responses about successful operations. According to the authors, the effort to implement idempotent services must be thought through as in

[8] JaCaMo-framework https://jacamo.sourceforge.net/

Design pattern	Advantages	Disadvantages	Improvements over the traditional pattern
Petri net sagas [SP10]	Performance gain with parallelization Simplified error handling and rollback implementation	Limited data passing between parallel processes Saga must be written with Petri net model Currently limited to Java	Possibility to parallelize the steps in the saga
SagaMAS [SP14]	Low coupling between microservices No central point of failure	Requires knowledge of MAS related technologies	No central point of failure in saga management Higher level abstraction for saga flows
Clustered orchestrator [SP18]	Higher fault tolerance No modifications to services	Idempotent endpoints might be complex to implement Increased total saga time	Improved fault-tolerance for a central orchestrator
Single-tail & Single-head-multiple-tails [SP11]	Can reach consensus in case of network or node failures Incomplete data is not viewable	Services must wait for a timeout in compensation if other than tail service fails If the state cannot be fetched, resources stay reserved	Improved management in case of network or node failure

Table 4.2. *Novel solutions based on the saga pattern*

some cases it might be easier to migrate to messaging services instead of modifying endpoints. As passive nodes must know about the status of requests made to services to handle possible failures correctly, total Saga time might increase. To minimize the time required for data replication, optimization methods for required information about sagas could be used. This requires replicating only the necessary data to either continue the coordination. As an optimization method for memory usage in replicas, in-memory saving of replicated data is presented which saves time as database operation is no longer needed.

The proposed system is modeled to calculate the total time of the saga when orchestrator replication is included. Provided model is evaluated using source values gathered from other literature including payload sizes, throughputs, data coefficients, and saga sizes as according to authors values would otherwise depend too much on implementation details. Based on the evaluation, it could be concluded that the time used for replication before each local transaction is insignificant in comparison to the whole saga duration, and optimizations in this department might not be that necessary. As stated by the authors, this could be a result of chosen source value for local transaction time which is quite high. Evaluations only include a comparison with the unoptimized saga using clustered orchestrator and a traditional saga without using clustered orchestrator is not used as the comparison value.

4.2.2 Novel 2PC-based solutions

Novel solutions based on 2PC were reported in two studies which include one article and one book chapter. Both of these are based on the same study conducted by Fan et al. [SP1] [SP2], where the book chapter [SP2] is a shortened version of the model proposed in the article [SP1].

Two-phase commit*

In the article, Fan et al. [SP1] argued that as 2PC is a locking protocol, its performance lacks in high-concurrency microservices, and the probability for deadlocks rises when the amount of competing threads increases. To fix these shortcomings of traditional 2PC, a novel transaction control protocol 2PC* is proposed which consists of a novel secondary asynchronous optimistic-lock (SAOL) and novel concurrency control protocol.

SAOL allows multiple transactions to modify the same resource with sequential version numbers controlling the order of execution. SAOL is divided into *firstLock*, which manages resource locking between multiple different transactions, and *secondLock*, which manages possible compensations for failed transactions. Resource reservation is divided into three steps including *begin*, *pre-commit*, and *second-commit* phases. Begin phase is used to get the correct version number and latest value of the wanted resource including handling when another transaction is currently reserving the value. In pre-commit, wanted resources are locked with *firstLock* if there are no other transactions currently modifying them. Otherwise, initiated transactions are rolled back and the process is completed without success. In the second-commit, locks are released and transactions committed. To ensure the correctness of the proposed locking protocol, states of the protocol are written with formal specification language TLA+[9] and run with TLC tool to find possible errors. As zero errors were returned by the tool, the authors argued that the novel locking protocol will not lead to deadlocks and is strictly serializable.

To manage concurrent transactions, a novel concurrency protocol is proposed, which is divided into *begin* and *commit* phases similar to in 2PC. To differentiate from the traditional pattern, a graph structure to manage transactions and their possible conflicting transactions are used. In begin phase, each microservice participating in distributed transactions is called to initialize graph-node which contains conflicting transactions in that service. A list of conflicting transactions is collected by going through all other unfinished transactions and checking if they modify the same resources. This conflict list is then sent to the coordinator which aggregates it with the existing graph of other transaction nodes and their conflict-lists and removes possible duplicates. In the commit phase, the transaction must wait for all of its preceding transactions to finish before all conflicting transactions can be calculated using a graph algorithm called *Tarjan*. After each conflicting transaction

[9] TLA+ https://lamport.azurewebsites.net/tla/tla.html

is resolved, the transaction can be committed and the coordinator can be notified about the result.

To evaluate the proposed solution, middleware prototype[10] based on Spring-frameworks aspect-oriented programming is implemented and evaluated using an experimental case with three microservices. Microservices are implemented using Apache Dubbo framework [11] with technologies such as Zookeeper, Nginx, Eureka, and Redis. With this setup, experiments including tests for consistency, throughput, latency, committing rate, and compensation are carried out. It can be concluded that consistency with distributed transactions consisting of multiple microservices can be reached even in cases of run-time exceptions. In throughput and latency experiments, a novel solution is compared with the traditional 2PC pattern with varying conflict amounts between transactions. Conflicts are caused by running an increasing amount of concurrent threads from 10 threads up to 500 threads.

It can be seen that throughput with a lower amount of concurrent threads doesn't differ significantly between traditional pattern and 2PC*. When a higher amount of concurrent threads are used for a higher possibility of conflicts, the gap between traditional pattern and 2PC* starts to widen significantly with throughput performance. In the case of 500 concurrent threads, experimented throughput for 2PC is only 12.7 transactions-per-second (TPS) versus 2PC* patterns 304.6 TPS. Authors argue that the difference in throughput is caused by using a novel locking protocol which reduces the blocking of transactions.

Similarly in latency, an experiment with a lower amount of concurrent threads shows no advantage for 2PC* compared to the traditional pattern. In experiments with moderate or high amounts of concurrent threads, a clear advantage for 2PC* over the traditional pattern can be observed. In the case of 500 concurrent threads, the reported latency of 2PC is 1352.7 milliseconds versus 2PC* patterns 623.2 milliseconds. Authors argue that this is due to the possibility to avoid deadlocks in 2PC* using a graph structure and due to less blocking of 2PC*.

In experiments relating to committing, it can be observed that with high conflict test cases traditional patterns committing rate drops to almost zero. In the case of 2PC*, committing rate stays consistently close to 100% which means that it can reliably commit transactions even with high conflict amounts. The authors argue that 2PC* can more reliably commit transactions with a high amount of concurrent threads as it tries to avoid aborting or retrying transactions. In the experiment related to compensations, it can be observed that 2PC* can compensate unsuccessful transactions reliably related to traditional patterns.

Two-phase commit+

[10] 2PC* middleware prototype https://github.com/Leofan93/2pc-star
[11] Dubbo-framework https://dubbo.apache.org/en/index.html

In the book chapter, Fan et al. [SP2] propose a pattern called 2PC+ based on 2PC. The novel pattern includes a novel secondary asynchronous optimistic-lock algorithm (SAOLA) which is the same as SAOL proposed in the article about 2PC* [SP1]. However, the concurrency control protocol is not proposed in this book chapter. For evaluation, the same experimental case is used with a slightly modified system setup, where 2PC* tests use *Intel i7*-processor and 2PC+ tests use *Intel i5*-processor. Only throughput and latency experiments are run to compare 2PC+ and traditional 2PC. Similar results can be observed with 2PC* concluding that 2PC+ has a clear advantage in both latency and throughput experiments compared to traditional 2PC. Remarkable is that 2PC+ can respond to request in the latency experiment with 500 concurrent threads in 473.6 milliseconds compared to 623.2 milliseconds in 2PC* experiment. Similar differences in response time, in favor of 2PC+, can be seen in experiments with 200 and 300 threads. In throughput experiments, 2PC* and 2PC+ have similar results with slight variation.

4.2.3 Other novel patterns

Patterns that cannot be classified in previous groups were reported in four studies [SP7] [SP8] [SP12] [SP16]. In this section, these patterns are presented, and the advantages and disadvantages for each presented pattern are collected in the table 4.3.

Fed-agent

Nikolic et al. [SP7] argue that multiple different representations of data using different database paradigms could be used to make a system high-performance. This could include having a specialized database for text search and for analysis of statistics. However, the usage of multiple databases to store different representations of a singular object causes problems with consistency as database transactions can only provide ACID properties at a local level. To handle this problem and provide global ACID properties and consistency for multiple representations of a singular data object, a novel solution called *fed-agent* is proposed.

Fed-agent works as a layer on top of microservices to handle read and write operations of singular objects located in multiple databases. The layer consists of multiple agent-nodes working together in Raft consensus group [25] to choose a leader amongst participating nodes. Only leader-node can respond to write requests, whereas each node including followers can respond to read requests. If the leader fails, a new leader is chosen from the followers. To handle concurrency amongst multiple representations of a data object, Multi-version concurrency control (MVCC) is used. In MVCC, objects in databases are not overwritten but versioned as a new instance with an incremented version number. The latest committed object is returned when an object is read even though a newer uncommitted object exists. With MVCC, serializable isolation can be provided in a multi-database transaction for a singular object. No modifications are required to be made to

participating microservices as coordination can be managed by implementing configuration files for the agent-layer. The internal structure of fed-agent or configuration files is not specified in this article.

To evaluate the proposed solution, a setup including an agent cluster and three microservices is used. Microservices are written using *Go*-language, and a data object is saved to databases *PostgreSQL*, *ElasticSearch*, and *Tile38*. Microservices themselves don't include code related to distributed transaction management as configuration files are used for transaction coordination. The setup is evaluated using Yahoo Cloud Serving Benchmark where overhead caused by fed-agent in comparison to direct microservice calls is calculated. Overheads are tested for multiple situations including payload sizes, number of agent nodes, number of concurrent threads, and amount of contention in the system. For write requests, overhead in most test cases is between 7-10 milliseconds. This is true with an exception for payload sizes higher than 200kB and with the amount of threads higher than 50 as in these cases overhead rises up to 20 milliseconds. For read requests, overhead is approximately 1 millisecond. The authors argue that overhead caused by the addition of fed-agent is trivial as response times of microservices used in production are significantly higher.

The proposed model does not currently work with the usual microservice architecture where data is divided into multiple databases. However, in future work, a share-nothing model is proposed to be added to the existing model.

GRIT

Zhang et al. [SP8] argue that current solutions for distributed transactions in microservices are not suitable when high consistency is required. This is because current solutions for highly consistent transactions, such as 2PC and deterministic databases, require locking during the process which increases the possibility of conflicts and aborts.

To solve this problem, the authors propose a novel solution called *GRIT* which promises consistent transactions across multiple microservices. GRIT takes advantage of deterministic database engines and patterns such as Paxos-based logging. Also, optimistic concurrency control (OCC) is used for transaction logic in microservices.

The solution consists of three phases called *optimistic execution phase*, *logical commit phase*, and *physical materialization phase*. In the optimistic execution phase, the transaction reads and modifies necessary resource objects and writes these steps to the read/write set. Log sequence number (LSN) is also captured in this phase to be used in the upcoming conflict resolution.

In the logical commit phase, conflict resolution is done on the database-level and global-level. At first, at the database-level, the local read/write set about the transaction is sent to the database transaction manager to check for possible conflicts. Conflicts are checked

from recently committed and cached write-sets. If conflicts are not found and the transaction only includes a singular database, write-sets can be added to the transaction log to finish the logical commit. If the transaction includes multiple databases, a local commit decision is sent to the Global transaction manager (GTM) which then orders participants to either commit or abort the transaction. As the commit decision is made using cached write-sets, no database locking is required during GTM decision. The logical commit procedure includes adding transaction write-set and corresponding LSN to the transaction log but not yet committing to the database.

The physical materialization phase then handles database commits by playing the transaction logs sequentially at the database-level. The underlying database engine must be deterministic to handle the log-playing. A deterministic database guarantees that the execution order is the same as the write-sets appended to the log. As database execution can be separated from the commit decision, scalability and performance can be acquired according to the authors. Also, snapshot isolation can be provided with LSN.

The solution is demonstrated in the paper but no evaluation is made. Demonstration describes a system including two microservices that then handle distributed transactions using GRIT. Source files for this demonstration are not given.

Microservice-based database

Laigner et al. [SP16] argue that distributed data management in microservices causes multiple problems such as additional coordination at the application-level, insufficient consistency, fractured reads, and inconsistencies caused by a lack of commit protocols. To handle these problems, the authors vision a novel database architecture to manage requirements set by microservice architecture. This requires the addition of microservice abstraction to the database-level to handle move complex communication of microservices.

The solution is visioned where a single distributed database system called *microservice-based database* is used to remove problems caused by *database-per-service* pattern. The authors argue that a single database is not a problem with the decentralized data management principle of microservices if developers can define constraints and the database can enforce these constraints between microservices. To handle this problem, virtual microservices are envisioned as an abstraction for novel microservice-based database systems. Virtual microservices are abstract copies used inside the database system to handle necessary constraints related to microservices including communication, and private state management. The authors argue that when microservice is not a black box to the database, it can handle microservice-related issues. In addition, the database could handle consistent atomic distributed transactions between microservices.

The vision includes an architectural description for microservices and databases. For

microservices, the framework is used to handle database calls. An example of this framework is given in Java and Spring. Using this framework, the database can build an abstract representation of microservices and their dependencies on each other in the start-up. In the database, architecture is layered into transaction processing, query processing, and data storage. Transaction processing manages transactions and events across microservices using the aforementioned virtual copies. Query processing can manage aggregate private states from multiple microservices without affecting the strong isolation. Data storage is divided into two other layers for the possibility to change storage providers and the possibility to scale instances based on workload.

As the paper only includes vision, no evaluation is made. However, the authors argue that this solution could be used to handle problems caused by the database-per-service pattern and decentralized data management of microservices.

ReLock

Frosini et al. [SP12] propose a transaction model for web services including microservices, which are Resource Oriented Architecture (ROA) and Representational State Transfer (REST) compliant. According to the authors, the proposed solution can manage transactions with full ACID guarantees.

ReLock consists of three services called Transaction Proxy, Transaction Service, and Lock Service. Services are built as intermediary components between client and target services to manage transactional needs. Each of the services must be stateless and comply with REST principles and ROA.

Transaction proxy is used to manage the discovery of *Transaction services*, and coordination of the services. Coordination includes communication with *Lock service*, Transaction service, and REST service in the correct order to manage the integrity of resources properly. This includes locking resources with a shared lock when it is read by the client and with an exclusive lock when a client wants to commit modifications to it. The Proxy requires transaction identifiers in HTTP-headers to coordinate the flow. The identifier is created in the transaction service during transaction initialization.

Transaction service is used to manage transactions' whole lifecycle and transaction resources. When the transaction is initialized, timeout is set to rollback transaction automatically if there is no commit or rollback request from a client. The transaction can be committed or rolled back by a client using appropriate HTTP-requests. For each resource modified by a transaction, a copy of the resource is created in the transaction service to reflect the state of the resource before any modifications, and after any operations made to that resource. The copy is used to compensate for any modifications made.

Lock service is used to manage access to resources with shared and exclusive locks. The resource can have multiple shared locks simultaneously, but only one exclusive lock

at a time. A client can not acquire an exclusive lock for resources if there are shared locks owned by other clients.

The solution is compared to related work using a set of properties. ReLock can reach atomicity and isolation properties as only one transaction can modify resource or resource collection at the same time. Resource collection is modified by adding a new resource to it or deleting an existing resource. In this case, an exclusive lock is required for resource and resource's collection. Also, compliance with HTTP and REST principles was compared amongst related work and ReLock. ReLock complies with both of them appropriately.

ReLock is not suitable for long-running transactions as pessimistic locking is used, and it locks resources for the whole duration of the transaction. For distributed transactions, the client must find a common transaction service to handle the transaction. Performance evaluation of the solution is not implemented as part of this paper but is mentioned as future work.

Design pattern	Advantages	Disadvantages
Fed-agent [SP7]	ACID and consistent transaction for single object over multiple databases	Does not support share-nothing model
	No code modifications	Response time overhead especially for write-operations
GRIT [SP8]	Consistent transactions	Requires usage of deterministic database engines
	No locks for database objects	
	Scalability and performance increase by separated logical/physical commit	
Microservice-based database [SP16]	Supports database transactions with microservices	Database must support logical constraints handling
	Logic related to distributed queries and transactions are relocated to database-layer	As the single database is used in this pattern, Polystore is required if data needs to be handled in multiple formats.
ReLock [SP12]	Services do not require modifications	Pessimistic lock required with modify/delete
	Easy scalability and availability for services	Requires common transaction service for transaction involving multiple services
	Full ACID compliance	

Table 4.3. *Advantages and disadvantages of other novel patterns*

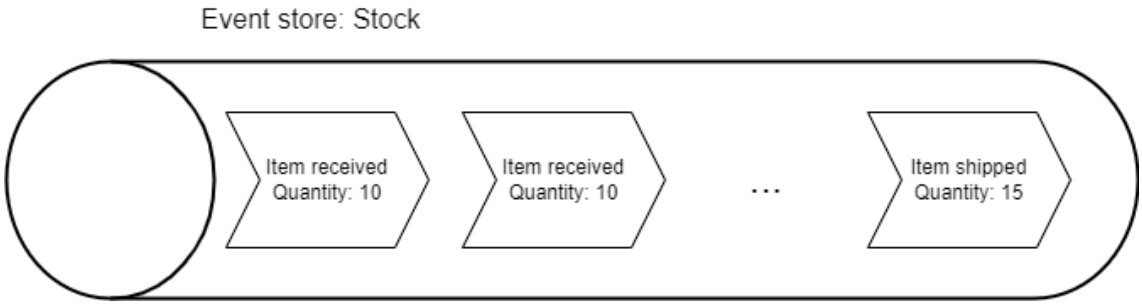

Figure 4.9. *An example of an event store consisting of events*

4.3 Implementation of the saga pattern (RQ3 & RQ4)

As seen in the section 4.1.1, there are multiple advantages but also disadvantages for the saga pattern. In section 4.2.1, novel solutions based on the saga pattern for some of these disadvantages were studied. In this section, deficiencies within the saga pattern are examined one by one, and possible solutions to fix these are discussed. Deficiencies include missing isolation property, and requirement for atomicity between local operation and invoking the next participant.

4.3.1 Local operation and sending message must be atomic

Problem: Each participating service must complete a local operation and send a message to the coordinator or to the subsequent participant. For this to be reliable, local operation and sending messages must be an atomic operation where either both succeed or neither succeed. [SP20] [SP14]

There are multiple possible solutions to overcome the requirement of atomicity between local operation and invoking the next participant. Next, the most prominent solutions are visited and discussed.

Event sourcing

Concept: State of the resources is managed as a sequence of events. When the state is modified, a new event is added to the event queue. As added events can also be used to invoke the subsequent participant, atomicity can be guaranteed. In figure 4.9, an example of an event store for event sourcing is demonstrated. The example presents a stock store that includes stock quantity for items.

To retrieve the current state, it is required to run these events in order as there is no single object to portray the resources [SP14]. In the example from figure 4.9, it is necessary to replay each event from the start if the current quantity of items is required. As it is hard to obtain the current state of the resources with event sourcing, additional pattern command-query responsibility segregation (CQRS) could be used to alleviate this [SP14] [SP20].

With CQRS, modifications and queries to the state are segregated from each other. In the case of event sourcing, the command-side could be used to add new events in the event sourcing pattern, and the query-side of CQRS could maintain the current state of the event store for easier queries [SP14]. A simplified example of the query state and how it could be composed is shown in figure 4.10. The state could be composed by taking periodic snapshots [SP14] from the event store's state or by subscribing to events and updating the table [SP20] as shown in figure 4.10.

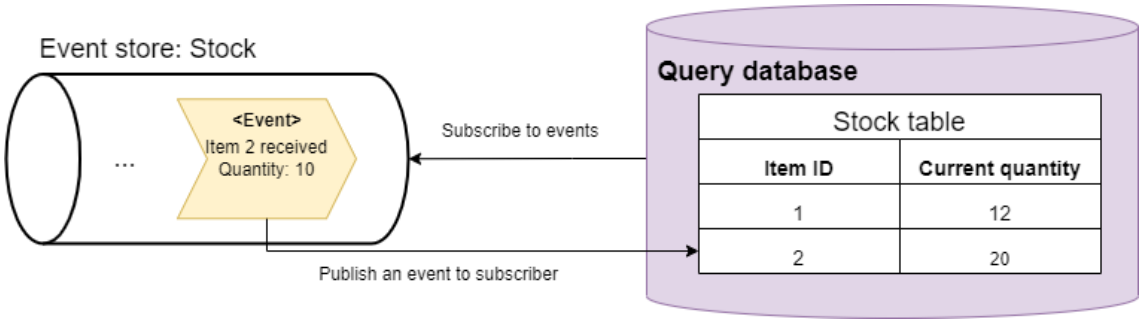

Figure 4.10. *A snapshot of current quantities of items*

Reported usage: The usage of event sourcing was discussed in three works [SP20] [SP14] [SP15]. As seen in the work [SP15], at least one implementation library is based on the event sourcing pattern. Also, CQRS is used as a pattern in libraries to implement the saga pattern [SP15].

Advantages include a possibility to update the state and send a message atomically [SP20]. Also, as each state modification is persisted as an event, it is easy to view the history of the state [SP20].

However, as a *disadvantage*, the state becomes harder to obtain which might require the usage of additional patterns [SP14]. Also, as it is required to modify the state to invoke the next participant, it might become problematic in case of errors when the state is not modified [SP20].

Transactional outbox

Concept: For invoking the next participant and local operation to be atomic, a message could be first added to an additional table in the database as a part of the local transaction. As a message and local operation are persisted in a single transaction, atomicity is guaranteed. [SP20] An example of *Transactional outbox* is shown in figure 4.11. After a message is atomically saved to the database, it is still required to send this message to the coordinator or to the subsequent participant. For this additional patterns such as *Log tailing* or *Polling publisher* patterns could be used. [SP20]

Polling publisher pattern periodically queries the additional table to check if there are new messages that should be sent. *Log tailing* uses built-in transaction logs of databases to

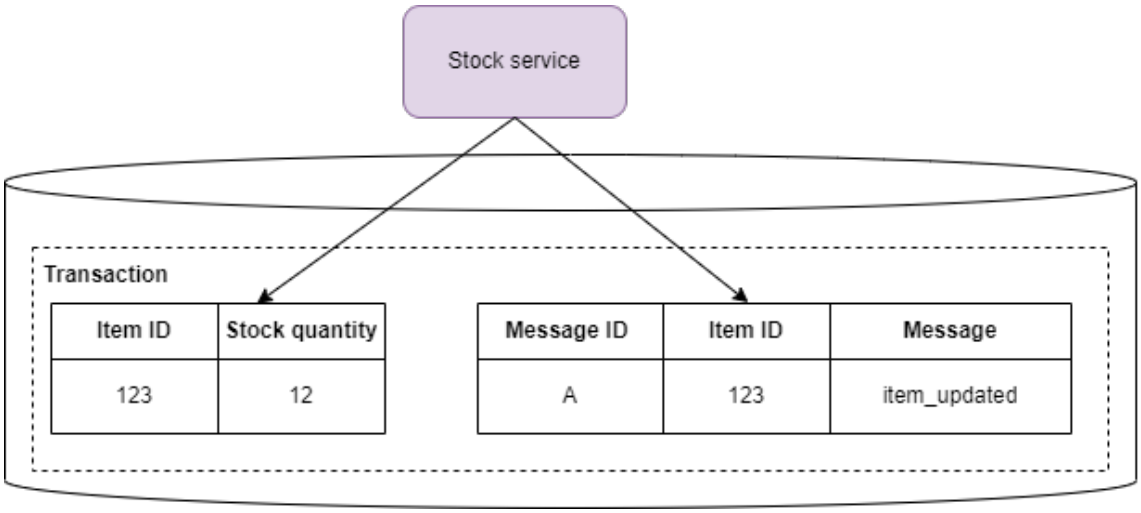

Figure 4.11. *An example of a transaction outbox (Modified from source: Richardson [SP20])*

find all modifications made and to publish a message to the coordinator or to the subsequent participant. [SP20]

Reported usage: The usage of *Transactional outbox* was discussed in one work [SP20]. As seen in the work by Stefanko [SP15], transactional messaging was used in one implementation framework called Eventuate Tram. However, Eventuate Tram is implemented by Chris Richardson[12] which is also the author of the selected paper [SP20] discussing this pattern.

As an *advantage*, required modifications to resources and saving a message can be done atomically. As an additional pattern is required for publishing the outbox messages, the advantages and disadvantages of these patterns should also be visited. An *advantage* of *polling publishers* is the simplicity as a simple query to the database is required to be done. However, a *disadvantage* is that it might be an expensive operation when done frequently. An *advantage* of *log tailing* is that it outperforms the *polling publisher* pattern as built-in APIs of the databases are used. However, as a disadvantage, it might be more complex to implement than a simple query to the database. [SP20]

4.3.2 Missing isolation property

Problem: As the saga pattern divides the coordination of multiple microservices into a sequence of sub-transactions, it is possible to see modifications made by currently running sagas that could still be aborted or compensated. This causes problems such as dirty reads [SP11] [SP15] [SP17] [SP20], lost updates [SP20] [SP5], and non-repeateable reads [SP20].

[12] Eventuate Tools https://eventuate.io/about.html

There are multiple ways to alleviate the problems with missing isolation property. These methods are listed below. Most of these countermeasures were demonstrated by Richardson [SP20] and one demonstrated by Xue et al. [SP11].

- *Resource reservation* guarantees that other operations cannot access locked resources [SP11].
- *Semantic locking* uses flags to indicate the state of the resource. For example, this could be a *pending* state while saga is unfinished to express that resource could still be rolled back in case of failure. [SP20]
- *Commutative updates* can be in any order without affecting the end result. This alleviates the problems with lost updates as the same end result will be achieved with any order of modifications made to resources. [SP20]
- *Pessimistic view* reduces the risk of dirty reads by reordering the operations inside a saga. For example, the operation with the most business risk could be rearranged to be the last operation in a saga to minimize the possibility of failure and compensation. [SP20]
- *Reread value* remedies lost updates by re-reading the resource before updating it to check that it remains unchanged. In case that resource has been modified, the saga should be aborted or restarted. [SP20]

4.3.3 Implementation frameworks

As there are multiple things to consider when implementing the saga pattern, an implementation might become laborious. To simplify the process of using the saga pattern, implementation frameworks could be used which are discussed in two works by Stefanko [SP15] and Durr [SP17].

Stefanko et al. [SP15] discussed and compared four Java-based frameworks to implement the saga pattern called Axon[13], Eventuate Event Sourcing[14], Eventuate Tram[15] and MicroProfile Long Running Actions (LRA)[16]. Comparison is made by implementing the same sample application with each framework and comparing attributes such as maintainability, scalability, and performance. In performance testing, two scenarios were run for each implementation where the first test included 1000 requests, and the second test included 10 000 requests.

In the tests conducted by Stefanko et al. [SP15], it was noticed that Eventuate Event Sourcing could not perform efficiently with the testing scenarios, and there were restric-

[13] Axon https://docs.axoniq.io/reference-guide/v/3.1/
[14] Eventuate Event Sourcing https://eventuate.io/gettingstarted-es.html
[15] Eventuate Tram https://eventuate.io/abouteventuatetram.html
[16] LRA https://github.com/eclipse/microprofile-lra

tions to use patterns such as CQRS and Event Sourcing. Also, another framework from Eventuate called Eventuate Tram is discussed. With these justifications, Eventuate Event Sourcing is left out of the further discussion.

Durr et al. [SP17] evaluated and compared two libraries to implement the saga pattern called Eventuate Tram and Netflix Conductor[17]. The evaluation was conducted by implementing a sample application with the help of these frameworks. In the evaluation, Durr et al. [SP17] used characteristics for general saga execution, and microservice architecture characteristics to compare the chosen patterns.

Eventuate Tram

Eventuate Tram is a framework specifically designed to implement the saga pattern. As seen in the evaluation made by Durr et al. [SP17], Eventuate Tram realizes most of the general saga characteristics such as the possibility to specify compensating transactions and automate them, and the possibility to use either choreography or orchestration as a coordination pattern. However, parallel execution of the transactions is not possible with the Eventuate Tram [SP17].

As Eventuate Tram is a Java framework, it is only possible to use Java to write the orchestrator [SP17]. This might limit the possibility of orchestrating the saga from the participant because every participant who also acts as an orchestrator would have to be implemented using Java. However, it is possible to use any programming language for services that are only participating in sagas [SP17].

Stefanko et al. [SP15] noticed in their performance testing that Eventuate Tram could work efficiently as a saga implementation framework. However, at the time of the writing, the framework had a problem with improper timeouts with Kafka in the second performance scenario with 10 000 requests. When the second scenario was successfully finished, the sample application with Eventuate Tram managed to get the lowest time among the selected frameworks.

Axon

The Axon is a framework to help with multiple core functionalities of the microservice architecture such as communication and coordination. According to Stefanko et al. [SP15], a problem with saga implementation using the Axon was that it could not manage the internal lifecycle of the saga which left the implementation to developers. Also, this framework is based on the CQRS pattern which might limit the usage if the pattern is not used in all participating services. As this article by Stefanko et al. [SP15] where this framework is evaluated is written in 2019, the situation might be different at present.

In the first scenario for performance testing, the sample application with Axon finished

[17] Netflix Conductor https://conductor.netflix.com/

successfully. The second scenario with 10 000 requests and 100 threads caused problems as the database lock could not be opened. This caused the testing to only successfully finish under 6000 requests out of 10 000 requests. However, Stefanko et al. [SP15] reported this problem to the maintainers.

MicroProfile Long Running Actions (LRA)

LRA is a specification for an application programming interface (API) to implement consistent long-running transactions without a need for any locking. This specification is based on other specifications such as Java API for RESTful Web (JAX-RS), and the Context and Dependency Injection (CDI). As a communication method, HTTP and REST are proposed in the specification. [SP15]

Stefanko et al. [SP15] studied Narayana's implementation of this specification. Even though the specification proposes the usage of CDI and REST for implementation, Narayana doesn't set these as restrictions when developing. However, at the time of the writing, REST was the only implemented method for communication. Saga execution and structuring can not be managed by this framework as it only has the coordination of the flow included. [SP15]

In the performance testing, both scenarios were successfully executed with the sample application using Narayana LRA. However, the total time in both performance testing scenarios was doubled compared to the sample application with Eventuate Tram. [SP15]

Netflix Conductor

Netflix Conductor is a framework to implement general distributed workflows and is not specifically designed to implement the saga pattern. The Conductor works as a central coordinator which accepts workflows in a JSON-format. [SP17]

As the Conductor is required as the central coordinator, choreography coordination can not be used with this framework. When compensation is required, the framework only allows for one failure workflow which then runs each compensation step even though each transaction step has not run yet. This differs from Eventuate where only already finished transactions are compensated. However, the Conductor allows parallel execution of transactions which is not allowed with Eventuate Tram. [SP17]

Participating services can be written with any language, and clients for Java and Python are included in the framework to help with the implementation. For failure cases, the Conductor enforces timeouts for the saga execution which might be required in some situations. However, when the saga should never expire, enforced timeout becomes problematic. [SP17] According to Durr et al. [SP17], Netflix Conductor can be used to implement saga workflows efficiently.

4.4 Implementation of other patterns (RQ3 & RQ4)

In this section, implementation details for TCC are discussed briefly. The 2PC pattern is not discussed as there was no literature related to implementation patterns for 2PC in the selected works. The only prototype implementation of 2PC was proposed in the article by de Heues et al [SP5]. However, this implementation was related to the Function-as-a-Service environment which is why it is not further discussed in this chapter. Even though Fan et al. [SP1] and Xue et al. [SP11] used 2PC as a reference in their testing, no implementation details were shared.

The implementation details of the previously introduced novel patterns are demonstrated in the section 4.2, which is why those patterns are not further discussed here.

Implementation of the TCC pattern

Implementation of the TCC pattern is discussed in one work by Pardon et al. [SP19] where the design is based on RESTful architectural style. Communication between services and coordinator is discussed, and examples are given on returned values. As implementation is only discussed at a high-level, there are no further details about the possible implementation patterns.

The selected works do not include other discussions about implementation details for the TCC pattern even though Xue et al. [SP11] used it as a reference pattern in their experiments.

5. DISCUSSION

As seen in table 5.1, the selected papers only included implementations as research prototypes which mostly were used to evaluate the proposed solution or to compare existing frameworks to implement coordination based on design patterns. Research prototypes for the saga pattern and 2PC were implemented for serverless systems in the work by de Heues et al. [SP5]. However, this work is not included in the table as the implementation is not strictly related to microservices.

Also, novel design patterns were discussed without implementation in multiple works which are classified as vision papers in table 5.1. Vision papers included additions to existing patterns or novel solutions to manage coordination between multiple services but did not include a prototype in the evaluation.

In this section, answers to each research question are first summarized. After this design patterns and implementation patterns are more closely discussed. Also, similarities and differences in comparison to related work are visited.

Answers gathered from the selected works can be summarized as follows:

RQ1 *What design patterns could be used in coordination between multiple microservices?*

The most prominent pattern seemed to be the saga pattern and after that the TCC. Both of these patterns use relaxed ACID properties to increase the possibility of concurrency, resulting in improved performance. Especially the prevalence of the saga pattern can be seen clearly as the selected works include multiple proposals to improve different sections of the protocol.

Even though these patterns are the most discussed, there seems to be a need for highly consistent protocols used within microservices there are multiple proposals in the research field for new solutions. However, these proposals are still in the early stages of research and are not ready to be used in the field. Also, 2PC was discussed as a solution for highly consistent coordination which is not recommended for microservice architecture due to its blocking nature and decreased performance.

RQ2 *What are the advantages and disadvantages of extracted design patterns?*

The main advantages of the saga pattern are improved availability and perfor-

mance. However, due to relaxed ACID principles, isolation is lost which can cause problems with the state being visible to external processes or risk of the state becoming inconsistent. There are two coordination methods for the saga pattern: orchestration and choreography.

Orchestration includes central coordination which improves maintainability, readability, and low coupling between services. As a trade-off, it brings a single point of failure to the system and a risk for the centralization of logic. Also, as each message must be transported through the orchestrator, more messages are required to manage the flow which could decrease the performance. In the choreography participating services manage the coordination in collaboration. This method requires fewer messages to manage the coordination but might become more complex as the coordination logic is divided into the participating services. Also, as services must receive messages sent by other services, there is a risk of tight coupling where services must be updated in unison if the sent message must be modified for some reason.

The usage of TCC brings advantages such as easier implementation for the rollback, guaranteed atomicity in most cases, and the possibility to add concurrency. However, resources must be compatible with the protocol for the possibility of concurrency. If resources cannot be used concurrently, resource reservation time might become long which decreases the overall performance. Also, atomicity cannot be guaranteed in a specific error case which could cause problems within implementation if not taken care of properly.

Other novel solutions were also proposed including GRIT, 2PC*, Fed-agent, ReLock, and Microservice-based database. However, these are still in the early phases of research, and cannot be seen as applicable solutions to implement the coordination yet.

RQ3 *How extracted architectural patterns can be implemented and what problems arise with the implementation?*

Implementation details were mainly discussed for the saga pattern where a couple of problems arise.

The first problem found in the selected works was the missing isolation which can cause dirty reads, lost updates, and non-repeatable reads. However, multiple solutions to manage this were found including resource reservation, semantic locking, commutative updates, pessimistic views, or rereading the values.

The second implementation problem found was the requirement for atomicity while sending the message to the subsequent participant and executing the local operation. Solutions for this included the usage of ES and CQRS patterns in cooperation,

or the usage of *Transactional outbox*. Two methods found in the selected works to implement *Transactional outbox* were *Polling publisher* and *Log tailing*.

Implementation frameworks for the saga pattern were compared in two of the selected works where it was seen that the Eventuate Tram is at least a suitable solution when implementing the saga pattern.

Implementation of the TCC protocol only included the usage of a RESTful architectural pattern where communication between the coordinator and participating services is managed with REST.

RQ4 *What are the advantages and disadvantages of the extracted implementation patterns?*

There were no advantages and disadvantages found for the implementation patterns related to missing isolation property. However, for some of those patterns, the advantages and disadvantages can be deduced as those were already used in other related design patterns such as in the TCC and 2PC. Resource reservation could cause blocking, which decreases performance, in case database locks are used for required resources. If intermediate states are used, correct use cases must be found to enable the possibility of increased concurrency. Otherwise, performance could suffer if not used proficiently. Also, re-reading values could cause increased amounts of abort or restarts as before updating, resources are re-read to alleviate the possibility for lost updates. Commutative updates and a pessimistic view require finding suitable use cases which might be hard without a deep understanding of these methods.

Advantages and disadvantages were shortly discussed for the implementation patterns related to the required atomicity between sending a message and executing a local operation. With Event Sourcing, atomicity is built-in as only a single event is required to add modifications to the state and to send a message. However, this causes problems when it is required to send a message but not to make any modifications. Also, it becomes hard to obtain the current state as the state is managed as a series of events. To help with the state, CQRS can be used to supply a periodic snapshot from the event store.

As an advantage, *Transactional outbox* was seen as a suitable solution to provide atomicity. Other advantages and disadvantages related to *Transactional outbox* found in the selected works were related to implementing it either with *Polling publisher* or *Log tailing*. *Polling publisher* was seen as a simple solution as well-known query languages are used to poll the current state periodically. However, this periodic polling might cause performance issues. Log tailing uses built-in APIs of databases to fetch the transaction log. According to one selected work, this could

	Vision paper	Research prototype
Saga-based solutions	[SP14] [SP18]	[SP4] [SP10] [SP11] [SP15] [SP17]
2PC-based solutions	-	[SP1], [SP2]
TCC-based solutions	[SP19]	[SP13]
Other solutions	[SP8] [SP12] [SP16]	[SP7]

Table 5.1. *Reported design patterns in the selected works*

be harder to implement but performance could be increased in comparison to the usage of the *Polling publisher*.

5.1 Design patterns

The most prominent patterns discussed in the selected works for the coordination between multiple services were Two-phase commit, the saga pattern, and Try-Cancel/Confirm.

In the selected works discussion about 2PC was mainly related to it not being recommended as a way to implement the coordination as the availability and the performance are weakened compared to other solutions with loosened consistency requirements. This result corresponds to related work by Ntentos et al. [4] where 2PC was not seen as a suitable pattern due to performance impact even though strict consistency is provided. Similarly, there were no signs of 2PC usage in open-source libraries or peer-reviewed articles in the review by Laigner et al. [3]. Even though the result that 2PC might not be a suitable pattern in microservice is equivalent to the related work, our work gives a more comprehensive view of the advantages and disadvantages related to this pattern. This differs from the related work where patterns were only discussed briefly.

Even though 2PC doesn't seem to be a highly used pattern within microservice architecture, a need for strict consistency patterns still exists. As seen in the related work, two works by Fan et al. [SP1] [SP2] proposed a solution based on 2PC to improve its usage in high contention situations such as microservices. Also, multiple other works proposed novel solutions for strict consistency. As an outcome, a clear demand for suitable strict consistency patterns to be used within microservice architectures can be seen.

Another solution for consistent coordination found in the selected literature was the Try-Cancel/Confirm protocol, which was not noted in the related literature. The pattern itself seemed like a trade-off solution between 2PC and the saga pattern where the commit is made in two phases but isolation is relaxed for higher concurrency support. As resource reservation is used instead of database locking, suitable business cases for this pattern must be found where concurrency can be increased even when resources must be reserved for the ongoing transaction. Also, there is a special case where atomicity can not be guaranteed which could cause problems if not dealt with properly. In our opinion

TCC requires more research including working prototypes to prove that it is suitable in a microservice architecture.

Overall, the saga pattern seems to be the most discussed and preferred solution to implement the coordination between services when there are no requirements for strict consistency. This can be seen in the selected works as there are most research prototypes implemented using the saga pattern with microservice architecture. Also, a discussion about frameworks to implement coordination was only found for the saga pattern which indicates its prevalence as a design pattern for microservices. In the literature related to the saga pattern, both choreography and orchestration were both discussed equally but it seemed that orchestration was the preferred solution when the coordination task got more complex.

In the related work by Ntentos et al. [4], the saga pattern with its two coordination styles was seen as a suitable method, and differences between choreography and orchestration were shortly discussed. In the work by Laigner et al. [3], it was noticed that open-source libraries tend to use event-based workflows and choreographed coordination. In the industry setting, orchestration-based solutions seem to be more prevalent. Therefore, a correlation between our results and this work might be present, as it can be assumed that coordination in an industry setting might get complex. Similar results in our work and in related work improve the reliability of the results gathered in our work.

As a difference to related works, our work puts more effort into understanding the advantages and the disadvantages of the saga pattern and both of its coordination variants. Also, the implementation of the saga pattern is thoroughly discussed, and possible problems in it are noted. As seen in the selected literature, problems within the implementation of the saga pattern were discussed in multiple works, and solutions were proposed. There still seem to be problems, mainly regarding failure management, that needs to be solved for the saga pattern to be more suitable for the microservice architecture. There are also problems that already have working solutions, such as missing isolation and the requirement for atomicity between an operation and sending a message, and might not need attention from the scientific community.

5.2 Future directions

As observed in this work, coordination between multiple services might be hard to implement due to multiple aspects that need to be taken into consideration for the coordination to be reliable. For this reason, coordination should be avoided as long as possible by designing service boundaries in a way that cross-service calls are minimized. This could be done, for example, by combining two services into one if there are a lot of dependencies between them. It is also possible to use a shared database between multiple services, which however adds coupling between services and therefore might not be a

suitable workaround with microservice architecture. As it is not always possible to avoid coordination, coordination methods should be further improved for them to be easier to implement with high reliability. In this section, possible future directions for the research concerning the coordination methods are demonstrated.

Despite the weaknesses within the saga pattern, it can be observed within the selected works that it is a commonly used way to implement coordination. The prevalence of the saga pattern can also be seen as multiple selected works offer novel solutions to manage its weaker aspects. With our work, the current stage of the research is extensively presented which makes it easier for the researchers to find future possibilities for the research in the context of improving already proposed solutions or to find problems that do not yet have any research initiated. As possible future works, implementation-related patterns could be further compared to understand their difference and impact on microservice-related properties such as availability and performance. Also, more research could be conducted to further improve failure management or to compare already existing solutions within the saga pattern especially when an orchestrator with a central coordinator is used.

Try-Cancel/Confirm was discussed in multiple selected works, but there were no implementation details shared other than a high-level description by Pardon et al. [SP19]. As a possible future work, use cases for TCC in the microservice architecture could be visited more closely as the pattern requires used resources to be compatible with the protocol.

It can be observed, from the results of this work, that current solutions are not adequate in all coordination situations due to their shortcomings. A need for high-consistency patterns or solutions to avoid coordination altogether can be also observed as there are multiple studies considering these possibilities in the selected works. For example, novel 2PC-based patterns that could offer high consistency with sufficient performance are covered in the selected works. Also, the use of stream processing systems is visioned even though it has not been traditionally seen as a possibility for data management in microservices. Additionally, a new type of database for microservices is visioned in the selected work by Laigner et al. [SP16] to avoid the need for coordination between services altogether.

However, for the new patterns to challenge established solutions in the future, new research is still required. As a potential future work, more research is required for the proposed highly consistent patterns as all of them were in the early stages of research. Future works could also include improving the stream processing systems, also as stated by Katsifodimos et al. [SP9], for them to be a viable solution when managing coordination between multiple services.

6. THREATS TO VALIDITY

In this section, possible threats to validity, and how these threats are avoided are discussed. Discussion is based on three validity categories namely internal, external, and construct validity.

6.1 Construct validity

Construct validity considers how the study is designed, and if there are possible threats within it.

The selection of a search string affects end result of the work significantly and if done incorrectly can be seen as a major threat. The search string includes the words *microservice* and *transaction* which both are generally used words when talking about this subject. However, there is a possible threat with design patterns that might not be considered as a transaction by some authors, and do not have an established terminology. One of these patterns is the saga pattern which was described as a *transactionless coordination*, in others as *coordination mechanism between services*, and in some as *distributed transaction pattern*. Even though there is no established terminology for the saga pattern, it is usually discussed in the context of transaction management which partly mitigates the threat.

The selection of inclusion and exclusion criteria for papers is important as those are used to include the most important papers and to exclude low-grade works and out-of-the-topic results. Threats concerning exclusion are minimized using general criteria such as removing duplication, non-English articles, and out-of-topic results. Also, non-peer-reviewed articles were only considered if citations were considerably higher than with other selected works to minimize the threats.

6.2 Internal validity

Internal validity relates to how the study was conducted and if there are any threats to the data analysis for this work.

To ensure that all the papers related to our topic were retrieved, the search was conducted in multiple databases including digital libraries and general indexing databases. Gray lit-

erature was also considered if it had a considerable amount of citations compared to other selected works. Also, to strengthen the results, backward and forward snowballing was conducted. However, there is still a threat that some related papers were not retrievable due to licensing issues or inadequate indexing.

As data extraction is manually implemented by the author, the possibility of bias can be seen as a threat. However, this is mitigated by following SLR guidelines. Also, the data extraction was first carried out by the author and then confirmed by examiners to reduce the threat of bias.

As only descriptive statistics were used when processing results, threats to validity in the analysis are minimal.

6.3 External validity

External validity defines how the results can be used outside the scope of this work and if the results are generalizable. As no conclusions are made from the results of the systematic mapping study, there are no threats to external validity.

7. CONCLUSION

In this work, we conducted a systematic mapping study to find out ways to manage the coordination between multiple microservices. As a result, the most prevalent design patterns were presented and discussed. Also, the implementation details of these patterns found in the selected literature were reviewed and discussed.

As seen from the results, distributed transaction protocols, such as Two-phase commit, which offer strict consistency were not the preferred solution in microservice architecture due to limited concurrency and decreased availability. To increase performance, ACID principles must be relaxed which was done by two other patterns found. The most mentioned patterns with relaxed ACID principles were the saga pattern (57%) and the Try-Cancel/Confirm (24%). Even though 2PC had the highest occurrence amongst selected works (62%), the discourse was mainly centered around its incompatibility with the characteristics of microservice architecture.

To confirm the prevalence of the saga pattern, four works proposed novel solutions for the internal problems within the saga pattern, and two works discussed and compared implementation frameworks for the saga pattern. Solutions related to internal issues were related to fault tolerance of central orchestration, fault tolerance in case of a node failure, and an attempt to increase performance with parallelization. Also, implementation details were mainly discussed for the saga pattern where it was seen that there are a couple of problems that have existing solutions. The first problem is the missing isolation which can be managed using solutions such as resource reservation or semantic locking. The second problem is the requirement for atomicity between sending a message to the subsequent participant and executing a local operation. This could be managed using *log tailing* pattern or *Event sourcing* together with CQRS. In the comparison of the implementation frameworks, it was noticed that there are multiple frameworks that can be used to implement the saga flows sufficiently. Especially a framework called *Eventuate Tram* performed well in the experiments conducted by both authors as it contained most of the required saga characteristics but also had high performance compared to other solutions.

In an addition to patterns with relaxed consistency, six novel solutions were proposed in the selected works that provide highly consistent transactions. Based on this, it was observed that there is still a need for high consistency solutions in certain use cases even

though current solutions, such as 2PC, are not preferred in the microservice architecture.

As seen from the results, the prevalence of the saga pattern and its two variants was explicit. However, there still seems to be a need for highly consistent solutions at least in the research community. Further research is still required to refine these strict consistency solutions forward. Additionally, further research might be still necessary to deal with the internal problems within the saga pattern. This could be done by either refining already proposed implementation solutions or by finding new ways to manage the problematic parts.

REFERENCES

[1] Lewis, J. and Fowler, M. *Characteristics of a Microservice Architecture*. URL: https://martinfowler.com/articles/microservices.html. (accessed 9.12.2021).

[2] Gilbert, S. and Lynch, N. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: https://doi-org.libproxy.tuni.fi/10.1145/564585.564601.

[3] Laigner, R., Zhou, Y., Salles, M. A. V., Liu, Y. and Kalinowski, M. Data Management in Microservices: State of the Practice, Challenges, and Research Directions. *Proc. VLDB Endow.* 14.13 (Sept. 2021), pp. 3348–3361. ISSN: 2150-8097. DOI: 10.14778/3484224.3484232. URL: https://doi-org.libproxy.tuni.fi/10.14778/3484224.3484232.

[4] Ntentos, E., Zdun, U., Plakidas, K., Schall, D., Li, F. and Meixner, S. Supporting Architectural Decision Making on Data Management in Microservice Architectures. *Software Architecture*. Ed. by T. Bures, L. Duchien and P. Inverardi. Cham: Springer International Publishing, 2019, pp. 20–36. ISBN: 978-3-030-29983-5.

[5] Vossen, G. Transaction. *Encyclopedia of Database Systems*. Ed. by L. LIU and M. T. ÖZSU. Boston, MA: Springer US, 2009, pp. 3150–3151. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_436. URL: https://doi.org/10.1007/978-0-387-39940-9_436.

[6] Bernstein, P. A. and Newcomer, E. Chapter 1 - Introduction. *Principles of Transaction Processing (Second Edition)*. Ed. by P. A. Bernstein and E. Newcomer. Second Edition. The Morgan Kaufmann Series in Data Management Systems. San Francisco: Morgan Kaufmann, 2009, pp. 1–29. ISBN: 978-1-55860-623-4. DOI: https://doi.org/10.1016/B978-1-55860-623-4.00001-9. URL: https://www.sciencedirect.com/science/article/pii/B9781558606234000019.

[7] A Critique of ANSI SQL Isolation Levels. eng. *SIGMOD 95: International Conference on Management of Data* 24.2 (1995), pp. 1–10. ISSN: 0163-5808.

[8] Pritchett, D. BASE: An Acid Alternative: In Partitioned Databases, Trading Some Consistency for Availability Can Lead to Dramatic Improvements in Scalability. *Queue* 6.3 (May 2008), pp. 48–55. ISSN: 1542-7730. DOI: 10.1145/1394127.1394128. URL: https://doi-org.libproxy.tuni.fi/10.1145/1394127.1394128.

[9] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A. and Gauthier, P. Cluster-Based Scalable Network Services. eng. *Operating Systems Review (ACM)*. Vol. 31. 5. New York, NY: Association for Computing Machinery, 1997, pp. 78–91.

[10] Zhou, X., Li, S., Cao, L., Zhang, H., Jia, Z., Zhong, C., Shan, Z. and Babar, M. A. Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry. English. *Journal of Systems and Software* 195 (2023). URL: www.scopus.com.

[11] Soldani, J., Tamburri, D. A. and Van Den Heuvel, W.-J. The pains and gains of microservices: A Systematic grey literature review. eng. *The Journal of systems and software* 146 (2018), pp. 215–232. ISSN: 0164-1212.

[12] Newman, S. *Building microservices*. eng. O'Reilly, 2015. ISBN: 1491950358.

[13] Taibi, D., Lenarduzzi, V., Pahl, C. and Janes, A. Microservices in Agile Software Development: A Workshop-Based Study into Issues, Advantages, and Disadvantages. *Proceedings of the XP2017 Scientific Workshops*. XP '17. Cologne, Germany: Association for Computing Machinery, 2017. ISBN: 9781450352642. DOI: 10.1145/3120459.3120483. URL: https://doi-org.libproxy.tuni.fi/10.1145/3120459.3120483.

[14] Vogels, W. Eventually Consistent. *Commun. ACM* 52.1 (Jan. 2009), pp. 40–44. ISSN: 0001-0782. DOI: 10.1145/1435417.1435432. URL: https://doi.org/10.1145/1435417.1435432.

[15] Petersen, K., Vakkalanka, S. and Kuzniarz, L. Guidelines for conducting systematic mapping studies in software engineering: An update. eng. *Information and software technology* 64 (2015), pp. 1–18. ISSN: 0950-5849.

[16] Wohlin, C. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE '14. London, England, United Kingdom: Association for Computing Machinery, 2014. ISBN: 9781450324762. DOI: 10.1145/2601248.2601268. URL: https://doi-org.libproxy.tuni.fi/10.1145/2601248.2601268.

[17] Khwaja, S. and Alshayeb, M. Survey On Software Design-Pattern Specification Languages. *ACM Comput. Surv.* 49.1 (June 2016). ISSN: 0360-0300. DOI: 10.1145/2926966. URL: https://doi-org.libproxy.tuni.fi/10.1145/2926966.

[18] Kitchenham, B. and Brereton, P. A systematic review of systematic review process research in software engineering. *Information and Software Technology* 55.12 (2013), pp. 2049–2075. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2013.07.010. URL: https://www.sciencedirect.com/science/article/pii/S0950584913001560.

[19] Taibi, D., Lenarduzzi, V. and Pahl, C. Architectural patterns for microservices: A systematic mapping study. English. *CLOSER 2018 - Proceedings of the 8th International Conference on Cloud Computing and Services Science*. Vol. 2018-January. Cited By :106. 2018, pp. 221–232. URL: www.scopus.com.

[20] Pal Singh, N. and Deshpande, A. *Solving distributed transaction management problems in microservices architecture using Saga*. URL: https://developer.ibm.com/articles/use-saga-to-solve-distributed-transaction-management-problems-in-a-microservices-architecture/. (accessed 4.4.2022).

[21] Garcia-Molina, H. and Salem, K. Sagas. 16.3 (Dec. 1987), pp. 249–259. ISSN: 0163-5808. DOI: 10.1145/38714.38742. URL: https://doi-org.libproxy.tuni.fi/10.1145/38714.38742.

[22] Mohan, C., Lindsay, B. and Obermarck, R. Transaction Management in the R* Distributed Database Management System. *ACM Trans. Database Syst.* 11.4 (Dec. 1986), pp. 378–396. ISSN: 0362-5915. DOI: 10.1145/7239.7266. URL: https://doi-org.libproxy.tuni.fi/10.1145/7239.7266.

[23] Bernstein, P. A. and Newcomer, E. Chapter 8 - Two-Phase Commit. eng. *Principles of Transaction Processing*. Second Edition. Elsevier Inc, 2009, pp. 223–244. ISBN: 1558606238.

[24] Pardon, G. and Pautasso, C. Towards Distributed Atomic Transactions over RESTful Services. eng. *REST: From Research to Practice*. New York, NY: Springer New York, 2011, pp. 507–524. ISBN: 9781441983022.

[25] Ongaro, D. and Ousterhout, J. In Search of an Understandable Consensus Algorithm. *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 305–319. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

APPENDIX A: THE SELECTED PAPERS

[SP1] Fan, Pan, et al. "2PC*: a distributed transaction concurrency control protocol of multi-microservice based on cloud computing platform." Journal of Cloud Computing 9.1 (2020): 1-22.

[SP2] Fan, Pan, et al. "2PC+: A High Performance Protocol for Distributed Transactions of Micro-service Architecture." Intelligent Mobile Service Computing. Springer, Cham, 2021. 93-105.

[SP3] Ramirez, Francisco, et al. “An Empirical Study on Microservice Software Development.” 2021 IEEE/ACM Joint 9th International Workshop on Software Engineering for Systems-of-Systems and 15th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (SESoS/WDES), IEEE, 2021, pp. 16–23, https://doi.org/10.1109/SESoS-WDES52566.2021.00008.

[SP4] Rudrabhatla, Chaitanya K. “Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture.” International Journal of Advanced Computer Science & Applications, vol. 9, no. 8, Science and Information (SAI) Organization Limited, 2018, pp. 18–22, https://doi.org/10.14569/ijacsa.2018.090804.

[SP5] De Heus, Martijn, et al. “Distributed Transactions on Serverless Stateful Functions.” DEBS 2021 - Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems, 2021, pp. 31–42, https://doi.org/10.1145/3465480.3466920.

[SP6] Knoche, Holger, and Wilhelm Hasselbring. "Drivers and barriers for microservice adoption–a survey among professionals in Germany." Enterprise Modelling and Information Systems Architectures (EMISAJ)–International Journal of Conceptual Modeling: Vol. 14, Nr. 1 (2019).

[SP7] Nikolic, Lazar, and Vladimir Dimitrieski. “Fed-Agent - a Transparent ACID-Enabled Transactional Layer for Multidatabase Microservice Architectures.” 2021 16th Conference on Computer Science and Intelligence Systems (FedCSIS), Polish Information Processing Society, 2021, pp. 489–92, https://doi.org/10.15439/2021F46.

[SP8] Zhang, Guogen, et al. “GRIT: Consistent Distributed Transactions Across Polyglot Microservices with Multiple Databases.” Proceedings - International Conference on Data Engineering, vol. 2019-, 2019, pp. 2024–27, https://doi.org/10.1109/ICDE.2019.00230.

[SP9] Katsifodimos, Asterios, and Marios Fragkoulis. “Operational Stream Processing:

Towards Scalable and Consistent Event-Driven Applications.” Advances in Database Technology - EDBT, vol. 2019-, 2019, pp. 682–85, https://doi.org/10.5441/002/edbt.2019.86.

[SP10] Röwekamp, Jan Henrik, et al. “Petri Net Sagas.” CEUR Workshop Proceedings, vol. 2907, 2021, pp. 65–84.

[SP11] Xue, Gang, et al. “Reaching Consensus in Decentralized Coordination of Distributed Microservices.” Computer Networks (Amsterdam, Netherlands: 1999), vol. 187, Elsevier B.V, 2021, p. 107786–, https://doi.org/10.1016/j.comnet.2020.107786.

[SP12] Frosini, Luca, et al. “ReLock: a Resilient Two-Phase Locking RESTful Transaction Model.” Service Oriented Computing and Applications, vol. 15, no. 1, Springer London, 2021, pp. 75–92, https://doi.org/10.1007/s11761-020-00311-z.

[SP13] Xie, Yang, et al. “Research on the Architecture and Key Technologies of Integrated Platform Based on Micro Service.” 2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), IEEE, 2018, pp. 887–93, https://doi.org/10.1109/IAEAC.2018.8577921.

[SP14] Limon, Xavier, et al. “SagaMAS: A Software Framework for Distributed Transactions in the Microservice Architecture.” 2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT), IEEE, 2018, pp. 50–58, https://doi.org/10.1109/CONISOFT.2018.8645853.

[SP15] Stefanko, Martin, et al. “The Saga Pattern in a Reactive Microservices Environment.” ICSOFT 2019 - Proceedings of the 14th International Conference on Software Technologies, 2019, pp. 483–90, https://doi.org/10.5220/0007918704830490.

[SP16] Laigner, Rodrigo, et al. “A Distributed Database System for Event-Based Microservices.” DEBS 2021 - Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems, 2021, pp. 25–30, https://doi.org/10.1145/3465480.3466919

[SP17] Dürr, Karolin, et al. “An Evaluation of Saga Pattern Implementation Technologies.” CEUR Workshop Proceedings, vol. 2839, 2021, pp. 74–82.

[SP18] Malyuga, Konstantin, et al. “Fault Tolerant Central Saga Orchestrator in RESTful Architecture.” 2020 26th Conference of Open Innovations Association (FRUCT), vol. 2020-, no. 1, FRUCT, 2020, pp. 278–83, https://doi.org/10.23919/FRUCT48808.2020.9087389.

[SP19] Pardon, Guy, and Cesare Pautasso. “Atomic Distributed Transactions: a RESTful Design.” Proceedings of the 23rd International Conference on World Wide Web, ACM, 2014, pp. 943–48, https://doi.org/10.1145/2567948.2579221.

[SP20] Richardson, Chris. “Microservices Patterns” https://www.manning.com/books/microservices-patterns , Manning Publications. Accessed: 14.4.2022

[SP21] Newman, Sam. "Building Microservices" https://learning.oreilly.com/library/view/building-microservices-2nd/9781492034018/ , 2nd Edition. O’Reilly Media. Accessed: 14.4.2022