Tampere University

Johannes Haapakoski

IMPLEMENTING ASYNCHRONOUS SAGAS IN A MICROSERVICE ARCHITECTURE

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Prof. Kari Systä
Prof. David Hastbäcka
September 2023

ABSTRACT

Johannes Haapakoski: Implementing asynchronous sagas in a microservice architecture
Master of Science Thesis
Tampere University
Master's Programme in Information Technology
September 2023

Microservices has become a popular architectural pattern for building applications in recent years. A microservice architecture consists of a set of small loosely coupled autonomous services, where each service typically has its own database.

A microservice architecture offers multiple advantages over a traditional monolithic architecture but also poses a lot of challenges. One such challenge is that a developer doesn't have access to typical database transactions when an operation has to span multiple services and databases.

The saga pattern has been proposed as a solution to this challenge. The idea behind the saga pattern is to split an operation that spans multiple services into partial transactions that can be executed individually in sequence. The partial transactions may need to define a rollback transaction. In the case of a failure during the saga, the rollback transactions are executed in reverse order.

This thesis studies the saga pattern and its implementation in a microservice architecture. The aim of this thesis is to find out how to implement the saga pattern in a microservice architecture such that it functions correctly and reliably. A small example architecture is designed in order to get a practical understanding of the design considerations.

This thesis concludes that the saga pattern can provide high availability and high reliability. However, it also introduces a lot of complexity. One thing to take into account when using the saga pattern is that sagas lack the isolation property provided by ACID transactions. Also, based on the example architecture, implementing the saga pattern seems to require a lot of infrastructure.

Keywords: microservices, saga pattern, distributed transaction

TIIVISTELMÄ

Johannes Haapakoski: Asynkronisten saagojen implementaatio mikropalveluarkkitehtuurissa
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Syyskuu 2023

Mikropalveluarkkitehtuurista on tullut suosittu arkkitehtuuri sovelluksissa viime vuosina. Mikropalveluarkkitehtuuri koostuu joukosta pieniä, löyhästi toiseensa kytkettyjä autonomisia palveluita, joista jokaisella on tyypillisesti oma tietokanta.

Mikropalveluarkkitehtuuri tarjoaa monia etuja verrattuna perinteiseen monoliittiseen arkkitehtuuriin, mutta tuo mukanaan myös paljon haasteita. Yksi tällainen haaste on, että kehittäjällä ei ole käytössä tyypillisiä tietokantatransaktioita silloin, kun jonkin toiminto koskettaa useita palveluita ja täten myös useita tietokantoja.

Saaga-mallia on ehdotettu ratkaisuksi tähän haasteeseen. Saaga-mallin ideana on jakaa useita tietokantoja kattava operaatio osittaisiksi transaktioiksi, jotka voidaan suorittaa yksitellen toinen toisensa jälkeen. Osittaisille transaktioille on määritettävä kompensoivat transaktiot, jotka voidaan suorittaa, mikäli saagaa suoritettaessa kohdataan vikatilanne. Jos jotakin saagan osittaisista transaktioista ei voida suorittaa, suoritetaan kompensoivat transaktiot käänteisessä järjestyksessä.

Tämän diplomityön tarkoituksena on arvioida saaga-mallia mikropalveluarkkitehtuurin kontekstissa. Työssä selvitetään, miten saaga-malli voidaan toteuttaa mikropalveluarkkitehtuurissa siten, että se toimii oikein ja luotettavasti. Työssä suunnitellaan pieni mikropalveluarkkitehtuuri näiden asioiden selvittämiseksi.

Työssä selvitettiin, että saaga-malli voi tarjota korkean luotettavuuden sekä korkean saatavuuden. Toisaalta sen käyttäminen lisää järjestelmään paljon monimutkaisuutta. Saaga-mallia käytettäessä tulee ottaa huomioon, että saagoilla ei ole ACID-transaktioiden eristyneisyys-ominaisuutta. Myöskin saaga-mallin toteuttaminen siten, että se toimii luotettavasti vaikuttaisi, työssä suunnitellun arkkitehtuurin perusteella, vaativan paljon infrastruktuuria.

Avainsanat: mikropalvelut, saga-malli, hajautettu transaktio

PREFACE

This thesis was done out of my own curiosity towards maintaining data consistency in a microservice architecture. I encountered a situation where I had to perform a write operation to two different database systems during one higher level operation. I then began to research different methods to do so such that consistency as a whole is maintained even if one of the write operations fail. I stumbled upon the saga pattern and started to ponder about how it could actually be implemented such that it functions reliably.

I want to thank my thesis supervisor, Professor Kari Systä, for guidance. I also want to thank my family for support during my studies.

Tampere, 10th September 2023

Johannes Haapakoski

CONTENTS

LYHENTEET JA MERKINNÄT

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
CAP	Consistency, Availability, Partition tolerance
JSON	JavaScript Object Notation
REST	Representational state transfer
SMTP	Simple Mail Transfer Protocol
SQL	Structured Query Language

1. INTRODUCTION

Microservices has become a popular architectural pattern for building applications in recent years. A microservice architecture consists of a set of small loosely coupled autonomous services, each typically with their own database. These services then collaborate with each other, usually over a network connection, to produce the desired outcome.

As opposed to a monolithic architecture, where all the application logic exists inside a single process, and all the data inside a single database, the microservices approach makes a system more modular. In a well-built system this provides some desirable characteristics. For example, scaling the application is easier, because instead of scaling all the functionality at once, only the limiting subset of functionality can be scaled in isolation. However, the microservice approach adds a lot of complexity due to it's distributed system nature.

In a monolithic architecture the software developer usually has access to database transactions with atomicity, consistency, isolation and durability (ACID) guarantees. Unfortunately no such guarantees are freely given in a microservice architecture when a transaction has to span over multiple services and databases. This makes it comparatively more challenging to ensure consistency within the system. For example, one service crashing or a network failure during a transaction that spans multiple services could leave the system in an inconsistent state.

The traditional method for managing transactions between different processes in a distributed system is called a distributed transaction. However, this method is not usually suitable for integrating microservices. One problem with distributed transactions is that they require that all the participants are online simultaneously. This reduces availability of the whole system as a single node being down will halt the execution of the whole transaction.[32, Chapter 6][33, Chapter 4] Also, the performance of a distributed transaction is much worse than that of a single database transaction[28, Chapter 9].

The saga pattern is an alternative to distributed transactions for transaction management in a microservice architecture. The advantage of the saga pattern is that it can be implemented by using asynchronous messaging; a service can do it's own part, send a message to a message broker and continue with other work. All the saga participants don't have to be available at the exact moment of execution.[32, Chapter 6][33, Chap-

ter 4] However, the saga pattern is a pattern merely for managing the business logic. It doesn't function correctly if a technical failure happens.[32, Chapter 4]

This thesis studies the saga pattern and its implementation in a microservice architecture. The aim of this thesis is to find out how to implement the saga pattern in a microservice architecture such that it functions correctly and reliably; how to mitigate technical failures and what implementation details should be considered. In chapter 2, a brief introduction to the microservice architecture will be given. Also in chapter 2, service-to-service communication methods will be compared. In chapter 3, the required background knowledge of a message broker is given. In chapter 4, the saga pattern will be looked at. In chapter 5, an example architecture that uses asynchronous messaging, and that is capable of supporting the saga pattern reliably is described. The example architecture implements a simple ticket sale application and uses the saga pattern to achieve data consistency between services. In chapter 6, the example architecture will be evaluated for correctness, reliability, and whether or not it fits the characteristics of a microservice architecture. Finally, in chapter 7, the conclusion of this thesis is given.

2. MICROSERVICES

In today's world of always-on applications and APIs, it is essential for almost all services to have high levels of availability and reliability, as well as the ability to rapidly scale in response to user demand. This is a significant shift from just a few decades ago, when only a small number of mission-critical services needed to meet these requirements. Whether it is a consumer mobile app or a backend payments application, the need for distributed systems that can handle these challenges is now widespread.[8, Ch. 1]

Microservices has become a popular term for describing such distributed software architectures. Microservices are a software architecture approach in which a large application is broken down into smaller, independent components, that can be developed, deployed, and scaled independently. Each microservice is designed to encapsulate some limited, cohesive subset of functionality related to some business domain[32, Ch. 1]. This modular approach to application architecture allows for greater robustness, flexibility and scalability. However, implementing a microservices architecture also requires careful design, as it introduces new challenges related to its distributed system nature.

2.1 Characteristics

It is essential to have a thorough understanding of certain characteristics when examining the use of microservices. These characteristics differentiate microservices from other types of service-oriented architectures. Therefore, it is imperative to delve deeper into these ideas to fully grasp the principles that enable microservices to function effectively.

First of such principles is the idea of independant deployability. When making a change to a microservice, it should be possible to deploy that modified service without having to deploy any of the other services[32, Ch. 1]. Deployment is risky: for example, the engineers at Google have discovered that around 70% of outages are due to changes in a live system[31]. The adoption of a microservice architecture significantly increases the number of components within a system, thereby raising the complexity of deployment. If the services are difficult to deploy, it can lead to a situation where the cost of deployment outweighs the benefits of having a microservice architecture. However, if the services are independantly deployable, the cost of deployment is greatly reduced as smaller overall deployments are possible.[7, Ch. 8] Also, focusing on independant deployability also

leads to better overall system design; in order to achieve independant deployability, the microservices have to be loosely coupled[32, Ch. 1].

Another such principle is that services should be modeled around a business domain rather than around technical concerns[32, Ch. 1][29]. This means that each service should be focused on a specific aspect of the business, such as customer orders, inventory management, or billing. By decomposing microservices into end-to-end slices of business functionality, the architecture can be optimized for efficient changes to business functionality. This is desirable because releasing a change that affects multiple services can be expensive. If the application was decomposed to microservices based on technical considerations, a new business requirement would more likely affect multiple services. By aligning the microservices with specific business goals and objectives, it can be ensured that the architecture is well-suited to support the needs of the business, while also maintaining a high degree of modularity from a technical standpoint.[32, Ch. 1] Domain-driven design can be a helpful technique in achieving this goal[32, Ch. 2][33, Ch. 2].

In the microservices architecture, it is recommended that each service has its own dedicated database. The database can either be a separate instance of the same database technology that other services are using or an entirely different database technology. [32, Ch. 1][33, Ch. 1][29] This is necessary because microservices should embrace the concept of information hiding. A microservice should hide as much information as possible and only expose what is necessary to other services through explicit interfaces. It should be clear what changes can be made within a microservice boundary without affecting outside parties. A shared database is considered an antipattern partly for this reason. With a shared database there would be no way for developers to tell what changes to the database schema can be made without breaking other users of the shared database. Services having clear boundaries results in a system with looser coupling and stronger cohesion.[32, Ch. 1]

The microservice community tends to favor the concept of smart endpoints and dumb pipes[32, Ch. 1][29]. This means that there shouldn't be much logic in the mechanism that is used in the communication between microservices. Microservices should favor RESTish protocols instead of more complex methods. For asynchronous communication microservices should use lightweight message brokers that don't do much more than just routing the messages to the correct destination.[29] The business logic should be contained within the services and it shouldn't leak outside. Putting too much smarts into the communication mechanism can lead to more coupling and less cohesion.[32, Ch. 1]

One consequence of using services as components is the need to design applications that can handle the failure of individual services. This means that the application must be able to continue functioning even if one or more of the services it relies on are unavailable or experiencing problems. Therefore, microservices should be designed for failure.[29]

If the architecture is designed incorrectly, one service failing could lead to a cascading failure of the whole system[32, Ch. 1].

2.2 Comparison to a monolithic architecture

The microservice architecture is often presented as an alternative to a monolithic architecture. Therefore, it might be useful to compare the two in order to get a better understanding of the advantages and challenges of a microservice architecture.

The most common example of a monolith is that of a single-process monolith. In this type of system all the application logic exists inside a single process. However, even some service-oriented architectures could be considered monolithic. A monolith mostly refers to the unit of deployment. If all the services in a service-oriented architecture are tightly coupled, and therefore have to be deployed all at once, the architecture can be considered a distributed monolith. A distributed monolith comes with all the challenges of a distributed system while not providing any of the benefits. Therefore, a distributed monolith is considered bad design. However, a basic monolithic architecture is a completely sensible and often preferable approach for smaller organizations due to less overall complexity.[32, Ch. 1]

2.2.1 Advantages

One of the benefits of a microservices architecture is the ability to manage and release updates to your application with a finer granularity. With microservices, a single service can be updated without redeploying the entire application. In contrast, traditional monolithic applications often require the entire application to be redeployed in order to make changes or fix bugs, which can cause delays in the release process. By breaking the application into smaller, independently deployable services, new features and bug fixes can be rolled out more easily.[29][32, Ch. 1]

Another benefit is that of smaller teams and smaller code bases[29][33, Ch. 1]. The problem with large teams is the communication overhead[33, Ch. 1]. If a task requires intercommunication between team members, the communication effort quickly becomes unmanageable. The increase in the required communication effort can be calculated as $n(n-1)/2$, where n is the number of team members. This means that 3 team members require 3 times as much pairwise intercommunication as 2; 4 require 6 times as much as 2, and so on.[26, Ch. 2] The benefit of a smaller code base is that it is more likely easier for the developer to understand[33, Ch. 1].

A microservice architecture provides better scalability[8, Ch. 6][32, Ch. 1][33, Ch. 1]. In a monolithic architecture the whole application has to be scaled all at once while in a microservice architecture only the components that actually need scaling can be scaled

when needed.[32, Ch. 1] Moreover, since each microservice is a self-contained unit, it can be scaled using the approach that works best for its specific needs. This is not possible with a monolithic architecture in which all components are tightly coupled and must be scaled together. In a microservices architecture some services may be stateless and can be scaled horizontally, while others may require sharding or other approaches to scale due to their stateful nature. By decoupling the services, the scaling strategy can be tailored to the needs of each individual service.[8, Ch. 6]

Another advantage is that multiple different technologies can be used[33, Ch. 1] [32, Ch. 1][29]. In a monolithic architecture, the team can be locked in to an increasingly obsolete technology stack. Migrating to a new technology stack in a monolithic architecture is really difficult because the whole code base has to be rewritten all at once.[33, Ch. 1] In a microservice architecture, each service can be implemented using the most suitable technologies[33, Ch. 1][29]. Also, newer technologies can be tested out with less risk since the new technology can be tried in a single microservice for viability[32, Ch. 1]. Moreover, migrating from an obsolete technology stack is not such a daunting task since the migration can be carried out in smaller chunks[33, Ch. 1].

Finally, microservices can potentially provide better reliability[33, Ch. 1][32, Ch. 1]. In a monolithic architecture, a fault in the system can bring down the whole system. In a microservice architecture, a fault in one service can be isolated while other services continue to function normally. This of course requires that the system is designed in such a way that a single service failing wont lead to a cascading failure of the whole system.[32, Ch. 12]

2.2.2 Challenges

One of the main challenges of a microservice architecture is getting the service boundaries right. Due to service boundaries being more rigid than the boundaries of a module within a monolithic application, the cost of fixing poor design decisions is likely to be higher: data might need to be migrated between two databases, refactoring might need to be done across multiple different code bases, and there may exist unidentified implicit dependencies between services which could lead to errors[7, Ch. 1].

In a monolithic architecture, updates to multiple components can be deployed atomically. In a microservice architecture, deploying features that span multiple services requires careful coordination. The deployments have to be correctly ordered according to the dependencies between services.[33, Ch. 1]

Another challenge is that debugging and monitoring a microservice architecture is much more difficult. Due to functionality being divided across multiple different processes, it's not possible to load the whole application into a debugger to find out what went wrong.

Moreover, reproducing a failure state is also really difficult due to the same reason.[8, Ch. 6] In a microservice architecture, each service is a possible point of failure. Determining where the problem lies requires additional work.[32, Ch. 1]

As the architecture grows in the number of services, it might not be feasible for a developer to run the whole stack on their machine. Due to this fact, the developer experience can suffer.[32, Ch. 1]

Writing end-to-end tests in a microservice architecture can be challenging. To run an end-to-end test in a microservice architecture, multiple services need to be deployed and properly configured. Also, the tests can more often randomly fail due to environmental issues like network failures.[32, Ch. 1]

In the 90's Peter Deutsch and James Gosling defined the eight fallacies of distributed computing. These fallacies are: the network is reliable, latency is zero, bandwith is infinite, the network is secure, topology doesn't change, there is one administrator, transport cost is zero, and that the network is homogeneous.[34] These fallacies need to be avoided when designing a microservices[7, Ch. 6].

For example, making a request from one service to another over a network is a very different thing as opposed to making an in-process method call. The request has to be serialized, transmitted, and deserialized. This adds significant overhead to each request. While it might be reasonable to make a large number of in-process method calls, making a large number of requests over a network is not.[32, Ch. 4] Also, the request can sometimes fail, for example, due to the network being unreliable, or the receiving service being offline[34]. Moreover, more thought has to be given to security; endpoints have to be secured so that only authorized parties can call them, and data in transit has to be protected from bad actors[32, Ch. 1].

Finally, since each microservice should have its own database, maintaining the consistency of data is more challenging. When working with a single database, one can use a database transaction to manage a state change. However, when working with state changes across multiple different databases, a different approach is required.[32, Ch. 1]

2.3 Service-to-service communication

An important aspect of designing a microservice architecture is deciding which service-to-service communication methods to use. Service-to-service communication is a form of interprocess communication. Interprocess communication methods can be categorized either as synchronous or asynchronous. Asynchronous communication means that the sender continues immediately after having submitted the message for transmission. Synchronous communication means that the sender blocks until it receives a response.[38, p. 173]

When thinking about communication in a microservice architecture, the interaction styles can be roughly divided into three categories: request-response, event-driven and common data. A request-response style of communication can be implemented in either a synchronous or an asynchronous fashion. Communication through common data and event-driven communication are inherently asynchronous.[32, Ch. 4]

2.3.1 Request-response

A request-response style interaction is one in which one service makes a request to another and expects back a response. Common use cases for this style of interaction are retrieving data from another service and executing operations in a certain order. This style of interaction can be implemented in either a synchronous or an asynchronous fashion.[32, Ch. 4]

Common synchronous request-response schemes are REST-over-HTTP and gRPC[33, Ch. 3]. When making a synchronous request, typically a network connection is opened with the downstream service. This connection is kept open until the downstream service responds.[32, Ch. 4]

An asynchronous request-response style of communication usually makes use of a message broker. When making an asynchronous request, the request is first sent to a queue that sits on the message broker. The service that receives the request consumes the queue whenever it is able to. Once the receiving service has consumed the request, it sends a response to another queue. The calling service then consumes this queue to receive the response.[32, Ch. 4][33, Ch. 3]

2.3.2 Event-driven

In an event-driven interaction, rather than a microservice asking another service to do something on its behalf, it simply emits an event that something happened. Other services can then act on this information accordingly.[32, Ch. 4][17]

An event is simply a statement that something occurred. That something most probably being something related to the domain of the service that emitted the event.[32, Ch. 4][33, Ch. 3]

The emitting service doesn't need to know about the services that are consuming its events. Its only responsibility is to broadcast the event to some feed. This makes this style of interaction inherently asynchronous in nature. This style of interaction, similarly to asynchronous request-response, can be implemented by making use of a message broker.[32, Ch. 4]

For example, consider a microservice that is tasked with handling payments. The pay-

ment service could store a payment entity which could include the following fields:

1. Id, id of the payment
2. UserId, id of the user
3. OrderId, id of the associated order
4. Status, an enumerated type with one of the following values: pending, cancelled, or completed

When a user makes a payment, the payment service could broadcast a payment completed event which could include the entity in the message. Another service, for example a service tasked with handling the delivery of orders, could then act accordingly.

2.3.3 Common data

One possible way to integrate microservices is to have them to communicate through some common shared data. This common data could for example be a file on a file system.[32, Ch. 4]

Examples of this pattern include data warehouse and data lake. These types of solutions are designed to help with processing a large amount of data. An important thing to note is that in these types of solutions the flow of data is unidirectional; one service publishes data into a shared data store which is then consumed by downstream consumers. A shared database, which can be updated by multiple services, could be problematic as outlined earlier.[32, Ch. 4]

2.4 Comparison of service-to-service communication methods

The synchronous request-response model of communication is what is usually the most familiar to developers. Therefore, it might be a good default method of communication to select.

However, a synchronous communication scheme causes temporal coupling between services[33, Ch.3][32, Ch. 4]. This is especially problematic if long call chains involving multiple services are formed. One service responding slowly somewhere along the chain will cause all the other services upstream to it to also respond slowly. Also, one service being offline will cause all the calls involving it to fail. Moreover, this also causes significant resource contention; each service has to have a network connection open waiting for a response.[32, Ch. 4]

An asynchronous communication scheme fixes the temporal coupling problem. When using asynchronous communication, the service receiving the request doesn't have to be reachable at the exact moment of the request being made.[32, Ch. 4][33, Ch. 3]

The disadvantage of asynchronous communication compared to synchronous communication is that it is less familiar to developers. Also, an asynchronous communication scheme is considered to be more complex.[32] There is also the question of which type of asynchronous communication to use.

Asynchronous communication through common data can be useful in cases where integration with a legacy system, which can't support more modern technologies, is required. It can also be a useful approach when a large volume of data needs to be transferred in one go. For other cases the pick is between asynchronous request-response and event-driven communication.[32, Ch. 4]

A request-response scheme makes sense in situations where a response is required before further processing can be carried out. It can also make sense in situations where in a failure case some compensating action has to be carried out.[32, Ch. 4]

An event-driven approach can be used when information needs to be broadcast to multiple receivers. Compared to a request-response style of interaction, a higher level of loose coupling can be achieved because the service that emits an event doesn't need to worry about receiving a response. Therefore, it doesn't explicitly need to know about the consumers of the event.[17][32, Ch. 4] The trade-off is that there is still an implicit coupling between the event's emitter and its consumers; if the event were to change in the future, there is no way, at least by looking at the code, to find out where such a change will have an effect[17].

3. MESSAGE BROKER

A message broker is a software component whose responsibility is to abstract the routing of messages. When using a message broker, the application sending a message doesn't need to concern itself with the location of the receiver or receivers, it only needs to know the location of the message broker. This greatly improves the maintainability of a message-based system.[25, Ch. 1] Another benefit is that a message broker holds on to a message until it can be received[33, Ch. 3]. Message brokers are a popular choice for implementing asynchronous communication in a microservice architecture because they offer a wide range of such useful capabilities[32, Ch. 5]. ActiveMQ, RabbitMQ and Apache Kafka are examples of popular open source message brokers[33].

In this chapter we will introduce concepts related to message brokers. We will also look at what kind of guarantees a message broker can provide in terms of message delivery, message ordering, and fault tolerance.

3.1 Message channels

A message channel is a virtual pipe that connects the sender and receiver of the message. Message channels are logical addresses inside a message broker. When a publisher send a message it sends it to a particular message channel. A consumer receives the messages from the message channels it listens to. A message broker doesn't come preconfigured with the channels. The developers need to configure the channels they need for communication. The message channels are usually fixed at deployment time instead of being created dynamically when the application is running. There are two different types of message channels a message broker can typically support: point-to-point channels and publish-subscribe channels.[25, Ch. 3]

A point-to-point channel guarantees that only a single consumer receives a message. If a point-to-point channel has a single consumer, that consumer receives all the messages. If on the other hand multiple consumers are present, the channel makes sure that any given message is consumed by exactly one consumer. A thing to note is that the channel can have multiple messages being consumed concurrently.[25, Ch. 4] Point-to-point channels are often referred to as queues.

A publish-subscribe channel can deliver a single message to multiple consumers. This means that a publish-subscribe channel has a single input channel and multiple output channels, one output channel for each subscriber.[25, Ch. 4] Publish-subscribe channels are often referred to as topics.

3.2 Message delivery

One of the more attractive features of message brokers is that they can be configured to guarantee message delivery. This is useful in the context of microservices because the service sending a message doesn't need to worry about the state of downstream services; even if a service was offline temporarily, the broker will make sure it eventually receives the message[32, Ch. 5].

The message broker can achieve guaranteed delivery by writing the messages it receives on disk rather than just keeping them in memory. This way, even if the broker were to crash, no messages are lost. There is a trade-off here in that persisting the message on disk takes time, and slows down the messaging system.[25, Ch. 4] Whether or not the message broker persists messages on disk is usually a configurable setting.

A thing to note is that a hundred percent guarantee is hard to achieve. There is almost always a case where messages can be lost. For example, a hardware failure might occur which may lead to messages being lost even if they are written on disk. The more redundancy there is in the system, the less the likelihood of such a situation happening.[25, Ch. 4] Most message brokers are cluster-based systems in order to ensure that a single machine failing doesn't cause message loss[32, Ch. 5].

Another aspect of message delivery is how many times does the broker guarantee to deliver the same message. Ideally the delivery guarantee would be an exactly-once guarantee. This means that each message would be delivered exactly once. However, it's debatable if an exactly-once delivery guarantee is possible[32, Ch. 5]. At least it's often considered too costly[33, Ch. 3]. Therefore, it's usually safe to assume that a broker can only provide an at-least-once delivery guarantee. An at-least-once guarantee means that the broker will attempt to deliver the message once, but in a failure situation the message might get delivered more than once.[33, Ch. 3]

If a message broker claims to provide an exactly-once delivery guarantee, it is a good idea to carefully look through the documentation to find out what it means in practise[32, Ch. 5]. For example, Kafka is one message broker that has an exactly-once delivery guarantee. In practise however, it only applies when transferring and processing data between Kafka topics. If the consumer were to communicate with an external system, Kafka cannot guarantee that that will happen exactly once.[14]

3.3 Order of messages

Another useful guarantee that a message broker can provide is that of message ordering being preserved. Not all brokers can provide this guarantee however, and there might be some additional limitations.[32, Ch. 5]

In the case of a single consumer instance on one channel, it's easy to make sure that messages are received in order. The consumer instance simply processes the messages sequentially. However, in this case the throughput is limited by the processing speed of a single instance. In order to avoid this bottleneck, it might be necessary to have multiple instances of the consumer.[25, Ch. 10] However, preserving message ordering with multiple consumer instances present is trickier.

An approach to preserving message ordering with multiple consumer instances, used by modern message brokers, is to use partitioned channels[33, Ch. 3]. For example, in Kafka, a topic is divided into partitions, and each partition has exactly one consumer. This way each instance processes the messages from the partition sequentially. The decision as to which consumer instance receives which message is made based on a message key. Messages with the same message key are routed to the same partition.[24] A thing to note is that message ordering in Kafka is only guaranteed inside a single partition[32, Ch. 5].

3.4 Reliability

A message broker can be configured in multiple different ways to support multiple different use-cases. In some use-case, some messages being lost now and then might not matter, and instead, low latency is the most important aspect. In this case, messages held in memory on a single message broker instance might be the optimal approach. However, in the case that losing messages is not acceptable, a single point of failure is not ideal.

Most message broker systems can be operated as a cluster[11, 1, 12]. In this case, a message broker consists of a cluster of message broker instances instead of a single instance. Clustering can provide higher availability and better fault tolerance.

A cluster can be configured to replicate messages across multiple broker instances[2, 15, 11]. This way the system can continue working without losing messages even in the case of some nodes completely failing. For example, RabbitMQ provides a type of durable, replicated, first-in-first-out queue called a quorum queue. A quorum queue promises not to lose any received messages as long as a majority of nodes are not made permanently unavailable.[2] In Kafka, replication is configurable for all topics by default[15]. Each message broker system is different, and therefore, it is important to study the message broker's documentation in order to correctly configure for reliability.

Proper cluster management is also vitally important, but is out of scope for this thesis. Many cloud vendors offer hosting of message brokers like RabbitMQ, and some offer even virtual message queues like Amazon SQS. These services may be worth considering.

3.5 Dead letters

There is a possibility that a message simply can not be delivered or consumed. For example, a bug could cause the consumer to always throw an exception when attempting to consume the message. This can block the message channel until the message is manually removed or the bug is fixed. Therefore, a maximum retry limit should be configured.[32, Ch. 5]

Then there is the question of what should be done with messages that pass the maximum retry limit. Easiest fix would be to just discard them, but that is not an option for all applications. Another option is to route them to another message channel. This message channel, often referred to as a dead letter channel, can route the messages to a consumer that implements some handling logic for such situations.[25, Ch. 4] The invalid messages can, for example, be shown in an administrator's user interface for manual inspection of what went wrong[32, Ch. 5].

4. SAGA PATTERN

The idea behind the saga pattern was originally introduced in a paper “Sagas”, written by Kenneth Salem and Hector Garcia-Molina in 1987[20]. The original idea behind the saga pattern is to divide long lived transactions, transactions lasting hours or days, into smaller partial transactions that can be executed independently. A partial transactions might be required to define a compensating transaction that undoes, from a semantic point of view, the actions performed by the partial transaction. In this way, the long lived transaction can be executed in smaller segments thus reducing the high abortion rate caused by the long lived transaction having to lock all the database objects involved in the long lived transaction for its entire duration. The difference between a saga and just some grouping of database transactions is that all the transactions that are part of a saga are either completed or somehow amended as a single unit.[20]

While only describing the details for a centralized database system, the original paper states that this pattern can be applied in distributed database environments as well[20]. In recent years in literature and in various online guides, the saga pattern has been proposed to be used to manage data consistency in microservice architectures[32, Ch. 6][33, Ch. 4][30][3].

As mentioned in chapter 2, it is good practise for each microservice to have their own database. However, this can lead to consistency problems when an operation that causes state changes in multiple services is required to happen in an atomic way. A database management system usually guarantees the atomicity of transactions, but that guarantee only applies to a single service’s database. The idea behind the saga pattern in a microservice architecture context is to model business processes spanning multiple services as a set of transactions that each service can perform independently. The saga pattern implementation will make sure that all the partial transactions are either completed or, in a failure case, semantically reverted.[32, Ch. 6].

The traditional method for handling such situations where an operation has to interact with two or more data repositories in a transactional manner is to use a distributed transaction. A distributed transaction is a transaction that involves multiple data repositories and coordination over a network. A common algorithm for distributed transaction management is the two-phase commit (2PC) algorithm. While a distributed transaction can provide the

usual ACID guarantees of a transaction, it has some drawbacks.[32, 33]

The 2PC algorithm as the name implies has two phases[43, Ch. 19]:

1. The voting phase
2. The decision phase

The 2PC algorithm is depicted in Figure 4.1. In addition to the participant data stores, the 2PC algorithm requires an additional component, a coordinator, that keeps track of the state of the distributed transaction. During the voting phase, the coordinator queries the participant nodes for whether or not they can commit their local transactions. During this phase the participants write the transactions on disk and check whether or not any constraints are violated. After the voting phase has finished, the coordinator has received an answer from each participant on whether or not they are able to commit, and it can then move on to the decision phase. If all participants are able to commit, the coordinator sends a message to all the participants to commit. If any one of the participants is unable to commit, the coordinator sends a rollback message to each participant.[43, Ch. 19]

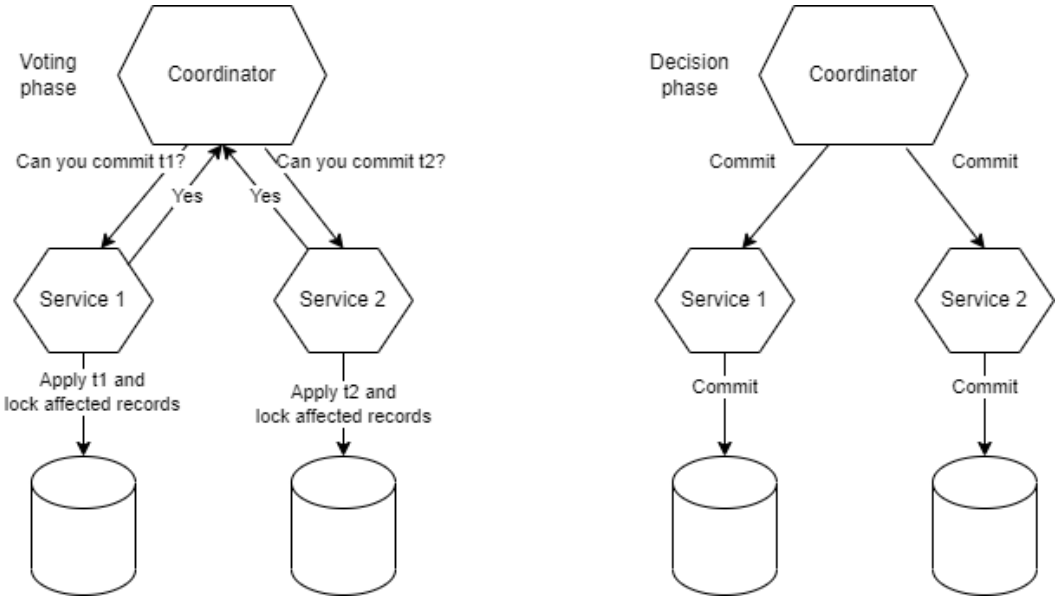

Figure 4.1. *Two-phase commit algorithm*

The participants obviously need to apply locks to the affected records during the voting phase in order to guarantee that they will not be altered before the commit of the second phase. This means that each participant needs to have the records locked for as long as the distributed transaction is active. Managing locks in a distributed system is tricky. There are also multiple failure modes to consider. For example, what if a participant votes to commit during the voting phase but no longer responds when asked to commit. There is also a window of inconsistency between when each service can actually perform their commit since there is no way to guarantee that the commits will happen at exactly the same time. This means that the isolation property of transactions is violated for a short

time period.[32, Ch. 6]

Another problem with 2PC is that all the participants have to be online at the same time in order to carry out the commit. This compromises availability as the availability of the distributed transaction is the product of the availability of all its participants. For example, if the distributed transaction involves 2 participants which are available 99% of the time, the distributed transaction is available only 98% of the time.[33, Ch. 4]

The CAP theorem[18] states that only two of the following three properties can be satisfied for a networked system providing shared data[38, p. 463]:

1. Consistency
2. Availability
3. Partition tolerance

Consistency in this context means that a request on the distributed shared data must appear as if it completed in a single instant. This can be rephrased as the need for requests made to a distributed shared memory to behave as if they were being executed on a single node, handling operations sequentially one after the other. A thing to note is that this property is different from the consistency property in ACID transactions.[21]

Availability means that a request received by an online node must result in a response. A system can respond with stale data and still be considered available.[21]

Partition tolerance means that the system will keep functioning even in the event of a network partition. When a network partition happens, nodes in different components of the partition cannot communicate with each other. This means that messages between nodes can be lost.[21]

When using distributed transactions the system can be considered to achieve consistency and partition tolerance; the system will not respond if the requested object is in the middle of a distributed transaction. A saga makes the opposite trade-off, preferring availability over consistency.[33, Ch. 4]

An execution of a saga is depicted in Figure 4.2. A successful execution flow is depicted with black arrows. The local transactions t_1, t_2, t_3 are applied one after the other until either the saga completes or a failure is encountered. In the case of a failure, the flow depicted with red arrows is entered from the point of failure. Compensating transactions are applied until all the changes have been reverted. If an error is encountered during t_2, only c_1 is applied. If an error is encountered during t_3, c_2 is applied first and c_1 after.

A saga can have two recovery modes in case of a failure: a backward recovery mode and a forward recovery mode. In a backward recovery, the compensating transactions are applied in the reverse order so that the changes that had been made up to the point of

failure are semantically undone as depicted in Figure 4.2 In a forward recovery, the saga picks back up from the point of failure and reattempts to carry out the transaction. [20]

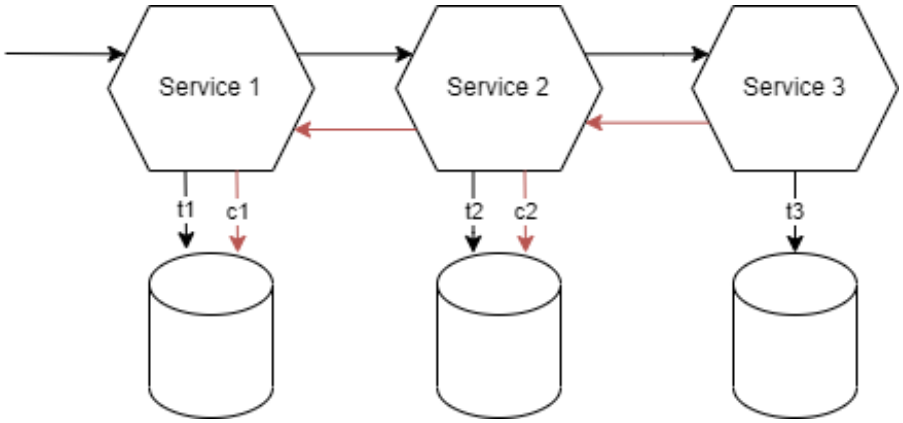

Figure 4.2. *Saga execution*

The execution of the saga is not consistent in the CAP theorem sense; an incomplete state can be observed while the saga is executing. However, higher availability compared to a distributed transaction is achieved as not all participants have to be present at the same time.

4.1 Implementation styles

There are 2 different implementation styles for the saga pattern: orchestration-based sagas, and choreography-based sagas. An orchestration-based saga has a central coordinator, while a choreography-based saga works just simply by each microservice reacting to events published by other microservices. It is also possible to mix and match different implementation styles.[32, Ch. 6]

In an orchestration-based saga implementation, a dedicated component is tasked with invoking the microservices involved in the saga in the correct order. If a service reports a failure, the saga coordinator is also tasked with invoking the microservices to apply compensating transacations. An orchestration-based saga can make use of a request-response style of communication.[32, Ch. 6]

The coordinator doesn't necessarily need to run as it's own service. It can be included as a part of one of the services. [33, Ch. 4]

The state of the saga should be persisted to a database after each step. This way, even if the coordinator was to crash, the state could still be recalled from the database. [33, Ch. 4]

The saga coordinator can be modeled as a state machine. A state machine consists of a set of states and some transitions between those states that are triggered by events. A state transition can trigger an action, which in the context of sagas is an invocation of

a microservice that is participating in the saga. Which transition to apply is determined by the current state of the saga and the result of the invocation. The example saga that is introduced in Chapter 5 can be modeled as the simple state machine depicted in Figure 4.3. Modeling the saga as a state machine is useful because it makes the design, implementation and testing of sagas easier.[33, Ch. 4]

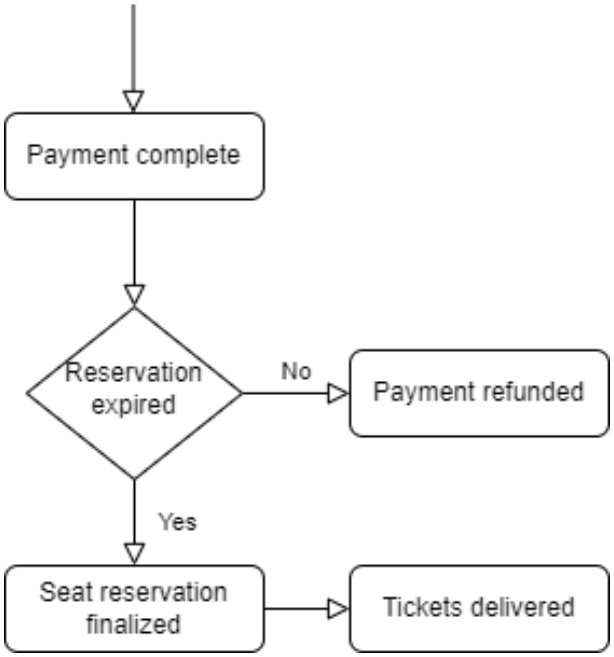

Figure 4.3. *Saga state machine example*

In a choreography-based saga, no central coordinator is required. The microservices simply publish domain events and expect other services to act accordingly. A choreography-based saga usually makes use of a message broker to ensure reliable delivery of the events. It's also possible to have multiple services react to the same event to achieve parallel processing.[32, Ch. 6]

As compared to an orchestration-based saga, in a choreography-based saga no one component needs to know about all the saga participants. Each service simply needs to know how to react to specific events. This makes for a more loosely coupled architecture. On the other hand, the benefit of an orchestration-based saga is that the steps of the saga are explicitly defined in the saga coordinator which makes it easier to figure out what the saga actually does. For a choreography-based saga, it can become difficult to figure out what the steps of the saga are because one would need to look at each service in isolation and see which events they are listening for.[32, Ch. 6]

A way to mitigate the problems in a choreography-based saga is to use a correlation id. For each saga execution a unique identifier is generated. If using topics for messaging, one service could listen for all the events related to a saga and keep track of its state. Another reason correlation ids are useful is that they can be used in conjunction with a

log aggregation tool. Each service can use the correlation id in its local logs, and then its possible to search for a specific saga execution from the aggregated logs.[32, Ch. 6]

4.2 Comparison to database transactions

Atomicity, consistency, isolation and durability (ACID) are properties that a transaction processing system typically promises for a transaction. Atomicity means that either all the steps of the transaction are completed or no steps are completed. This means that a partial execution of a transaction shouldn't be possible. If a transaction processing system encounters an error during a transaction it will undo the partial steps that it had already applied. Consistency means that the system should be in a consistent state before and after a transaction. Isolation means that a system running a set of transactions should produce an outcome that is equal to running the transactions one after the other. The durability property means that once a transaction has completed the results are saved in a persistent storage.[5][Ch. 1]

A saga performs local database transactions at each step and therefore the individual steps of a saga are ACID compliant; as long as the underlying database systems are ACID compliant. Therefore, the local databases will make sure that transactions that are committed are durable and consistent. As stated earlier, a saga will make sure that either all the steps are completed, or that they are all semantically rolled back. Therefore, sagas can be thought to be atomic as well. However, sagas lack the isolation property because once a local database transaction commits, other sagas can see the committed results immediately. This can lead to data anomalies.[33, Ch. 4]

The five following anomalies can happen due to the lack of isolation[19]:

- Dirty reads
- Lost updates
- Non-repeatable reads
- The phantom anomaly
- The dirty write anomaly

A dirty read anomaly can happen when a transaction updates a record and another transaction reads the updated value. After this, the first transaction is rolled back. In this case the second transaction has read a non-existing version of the record.[19]

The lost update anomaly happens when a transaction reads a value for update. After this, a second transaction updates the record. Later, the first transaction updates the value. In this scenario the second transaction's update is lost. The dirty write anomaly can be considered a special case of the lost update anomaly.[19]

A non-repeatable read anomaly is a situation in which a transaction reads a record, but before the transaction has finished the record is updated and committed by another transaction. In other words, the read record cannot be trusted to be correct.[19]

The phantom anomaly happens when a transaction reads records by using a search condition. Later, during the execution of the transaction, records have been updated in such a way by other transactions that the search would return different results.[19]

4.3 Lack of isolation countermeasures

The lack of isolation should be taken into consideration when designing sagas. There are 5 countermeasures to either prevent isolation anomalies completely, or to reduce their business risk for sagas. The countermeasures are[33, Ch. 4]:

- Semantic lock
- Commutative updates
- Pessimistic view
- Reread value
- By value

A semantic lock is an application level lock for a record that the saga creates. When a compensatable saga transaction creates a record, it marks it with a flag that signals to other transactions that the record is involved in a saga execution that isn't yet finished. If the saga completes successfully, it clears the flag. The flag is also cleared in a backward recovery. A noteworthy problem with locking is that it can lead to a deadlock situation. This should be taken into consideration if a semantic lock countermeasure is used.[33, Ch. 4]

Sometimes it is possible to make updates commutative. Commutativity means that the same result will be produced no matter in which order operations are executed. This removes the possibility of a lost update anomaly.[33, Ch. 4]

A pessimistic view countermeasure can be applied to reduce the business risk associated with dirty reads. Updates that decrease the options of a user should be compensatable, while updates that increase the options should be retriable. This way, if a dirty value is read, it's one that allows for less options for the user.[33, Ch. 4]

In a reread countermeasure, a record is verified to be unchanged by other sagas between when it was first read, and when an update is attempted. This works to prevent the lost update anomaly. If another saga has modified the record, a rollback is performed.[33, Ch. 4]

A concurrency mechanism can be selected dynamically based on business risk. An ap-

plication using the by value countermeasure could dynamically choose between, for example, sagas and distributed transactions. High-risk requests could be executed by using the more heavy distributed transactions, and low-risk requests could be executed using sagas.[33, Ch. 4]

While the lack of isolation should be taken into account, it isn't necessarily always a problem. For database management systems the isolation level is usually lower than fully serializable by default.[33, Ch. 4]

5. IMPLEMENTATION OF EXAMPLE ARCHITECTURE

In this chapter an example backend microservice architecture that uses the saga pattern will be described. The objective of the architecture is to implement a ticket sale system. The goal is to find out how the saga pattern can be implemented correctly and how well it fits the microservice paradigm. This chapter will look at the design and implementation. In the following chapter, Chapter 6, how well the implemented system conforms to the microservice principles will be evalauted.

The example microservices are implemented using the Spring framework[42], Spring Boot[40], and Java. PostgreSQL[22] is used as a database for each microservice. React[37] is used to implement a simple user interface. The user interface is not relevant to the functioning of the saga pattern and therefore is not described in this thesis in more detail. Apache Kafka[12] is used as a message broker for communication between the services. Kafka Connect[13] and Debezium[9] will be used to relay messages from the service's databases into Kafka. Stripe[39] will be used for payment integration. More details will be provided later in the architecture overview section.

5.1 Ticket sale system requirements

The goal is to implement a simple ticket sale system with limited functionality using the microservice architecture style and the saga pattern for integration between the microservices. The ticket sale system should fulfill the following requirements:

1. User should be able to reserve a number of randomly selected seats
2. User should be able to make payment for tickets for the seat reservation
3. The seat reservation should be released after 10 minutes if the user doesn't complete the payment
4. The tickets should be delivered via email to the user
5. If the tickets can't be delivered after the user has made payment because the seat reservation has expired, the user's money should be refunded automatically

The system should be designed in such a way that it adheres to the microservice architecture characteristics described in Chapter 2. Namely each microservice should have a dedicated database, one microservice crashing shouldn't affect other services, business

logic should be contained inside the services and should not leak into the communication mechanism, the services should be independently deployable, and the services should be designed around business concerns.

5.2 Architecture overview

In this section, the architecture of the system will be looked at. The C4 model[6] is utilized to describe the system architecture.

Firstly, an introduction to the C4 model is provided. Then two of the higher levels of C4 diagrams are presented and the used technologies are described in more detail. After the components are described, a description of how services communicate with each other during a saga is provided. Finally, reasoning behind using the specific technologies is given.

5.2.1 The C4 model

The C4 model is a way for software development teams to describe software architecture. The model consists of a set of hierarchical abstractions and a set of hierarchical diagrams. In the C4 model a software system consists of one or more containers, the containers consist of one or more components, and the components are implemented by one or more code elements. Each of these levels of abstraction can be described using notation independent diagrams. In addition, the model includes a person abstraction which represents a specific type of user of the software.[6]

A software system is the highest level of abstraction. It describes something that delivers value to users. The users could be humans or other software systems. The software system that is being modeled could depend on other software systems.[6]

A container is the second level of abstraction. In the C4 model, a container does not refer to a docker container but instead it represent something that needs to be running in order for the overall system to function. A container could be an application or a data store. For example, a container could be a server-side web application, a client-side web application, a database, a serverless function, or even a shell script.[6]

A component is a set of interrelated functionality that is wrapped in a clearly defined interface. A container consists of components. For example, in the Spring Framework a component could be a spring bean. A component in the C4 model does not refer to an individually deployable unit as it might in some other context. Instead, all components inside a container usually execute inside one process.[6]

Finally, further detail can be given about each component by providing a code level description. This can be achieved using UML class diagrams, for example. It's not neces-

sary to draw these diagrams by hand as a code level diagram can usually be automatically generated by tooling.[6]

It's not always necessary to include diagrams for all 4 levels of abstraction. Only the ones that add value can be included. Only a system context diagram and a container diagram are often sufficient for most teams.[6]

5.2.2 C4 Diagrams and technology descriptions

The system context diagram for the ticket sale system is shown in Figure 5.1. A user is able to purchase tickets using the system and is able pay for the tickets using Stripe. The ticket sale system uses a Gmail SMTP server to deliver the purchased tickets to the user and receives a notification from the Stripe integration when a user completes a payment.

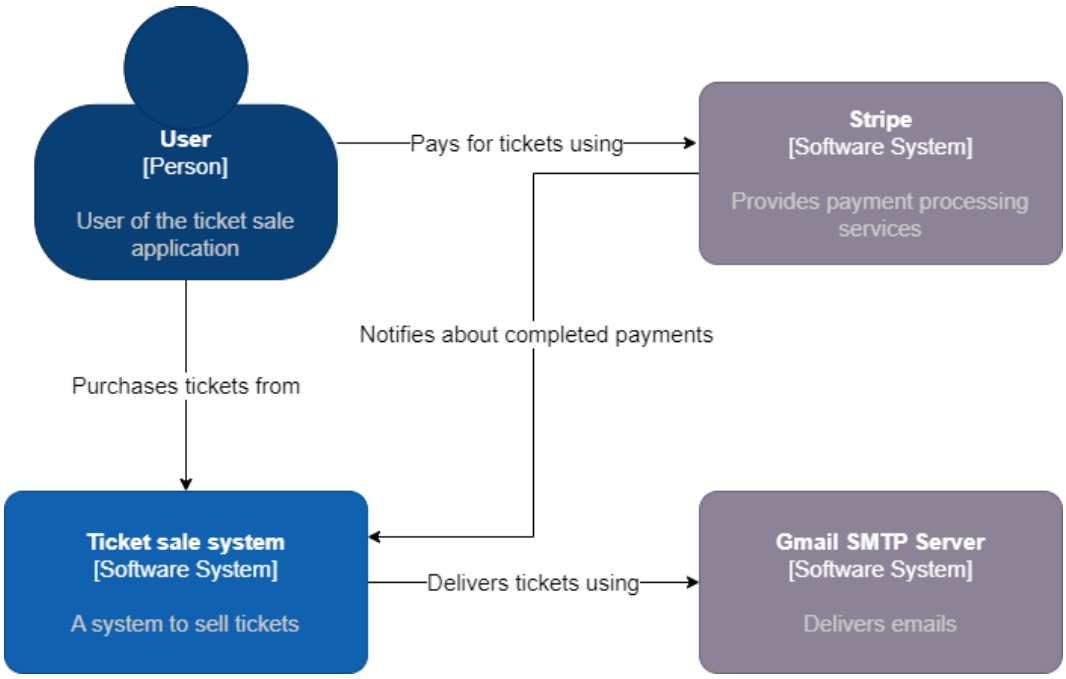

Figure 5.1. *System context diagram*

Simple Mail Transfer Protocol (SMTP) is the protocol used to deliver email[27]. An SMTP server provided by Gmail is used.

Stripe is an payment services provider[39]. A payment services provider is a company or service that enables businesses and individuals to accept and process electronic payments for goods and services over the internet. Payment service providers act as intermediaries between merchants and customers, securely handling the transfer of funds and facilitating electronic transactions.[4]

The container diagram is presented in Figure 5.2. The architecture consists of two microservices each with its own PostgreSQL database. The user can reserve seats through the seating service and order tickets for the reserved seats through the order service.

The order service communicates with the external systems. The services communicate with each other through Apache Kafka. Kafka Connect is used to facilitate communication between the services during a saga. The communication between services will be described in depth in the section following this one. The services are implemented using the Spring Framework with Spring Boot and Java.

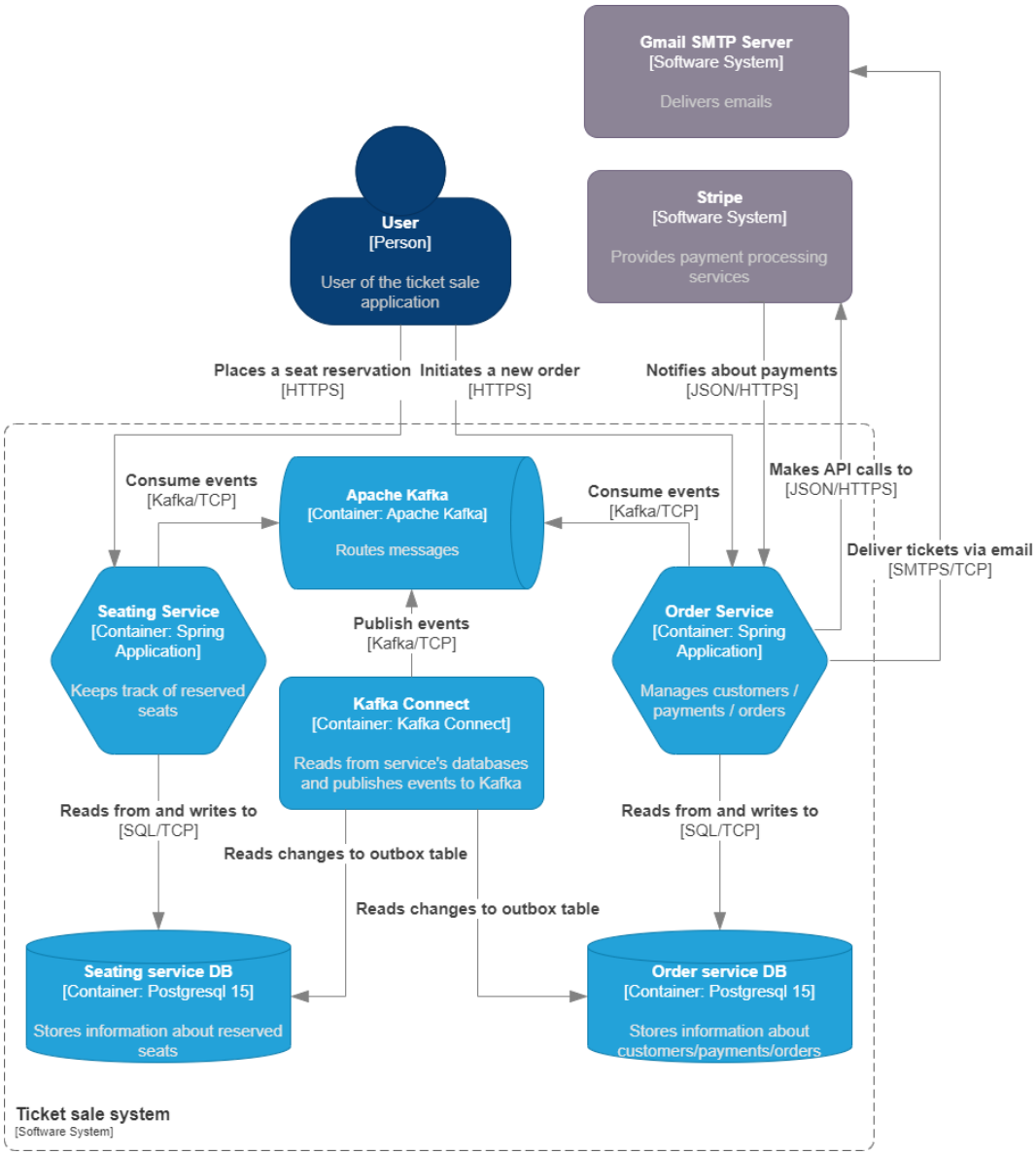

Figure 5.2. *Container diagram*

The Spring Framework is a framework for building Java applications[42]. The Spring Framework provides the Spring Data JPA package which provides an object-relational mapping[41]. The Spring Data JPA package will be used for database operations in the microservices. The Spring Framework also provides a client for Kafka[35]. The Spring Kafka client is used for communication with Kafka. Finally, Spring Boot provides auto-configuration for the Spring Framework which makes it easier to develop Spring based applications[40].

PostgreSQL is a relational database management system. PostgreSQL is ACID-compliant and uses the SQL language.[22]

Apache Kafka is an event streaming platform. Although Kafka coins itself as an event streaming platform, it also provides all the functionality of a typical message broker like RabbitMQ.[12] An in-depth description of a message broker was given in Chapter 3.

Kafka Connect is a tool that provides a framework for integrating Kafka with other data systems. A Kafka Connector is a component that is hosted on Kafka Connect. Two types of connectors can be hosted on Kafka Connect: sink connectors and source connectors. A sink connector reads data from Kafka and delivers it to some other system, for example, a SQL database. A source connector read data from some other system and delivers it to a Kafka topic.[13]

In the system, a Debezium source connector is used to read data from the service's databases into Kafka topics. Debezium is a set of source connectors for Kafka Connect[9].

5.2.3 Communication between services during a saga

In the example architecture, a Debezium source connector for Kafka Connect is used to move messages from the microservice's PostgreSQL databases into Kafka topics. Each service's database has a special table, called an outbox table, to which messages to other services are written to. The Debezium connector hosted on Kafka Connect can efficiently capture the changes made to the outbox table and route the written messages to the correct Kafka topics.

Schema for the outbox table is shown in Figure 5.3. A random UUID, ***message_id***, is used as the primary key. The ***type*** field specifies the type of event: create, update or delete. The field ***aggregate_type*** specifies the type of the aggregate that the event is about. The field ***aggregate_id*** is the identifier of the aggregate. The ***payload*** field contains the created, updated or deleted aggregate in its entirety in a JSON format.

Outbox	
PK	message_id UUID NOT NULL
	aggregate_type VARCHAR(255) NOT NULL
	aggregate_id VARCHAR(255) NOT NULL
	type VARCHAR(255) NOT NULL
	payload JSONB NOT NULL

Figure 5.3. *Outbox table schema*

For example, if the order service wishes to update an order aggregate and publish a message informing other service's about the updated order, it updates the order aggregate and inserts a new record to the outbox table with the following values:

- ***message_id***: random UUID
- ***aggregate_type***: Order
- ***aggregate_id***: identifier of the order
- ***type***: UPDATE
- ***payload***: JSON represantation of the updated order

During a single step of a saga, a service has to execute its partial transaction, and after, instruct the next participant to execute theirs. Using the outbox approach, both the local database changes and the message can be persisted in a single transaction.

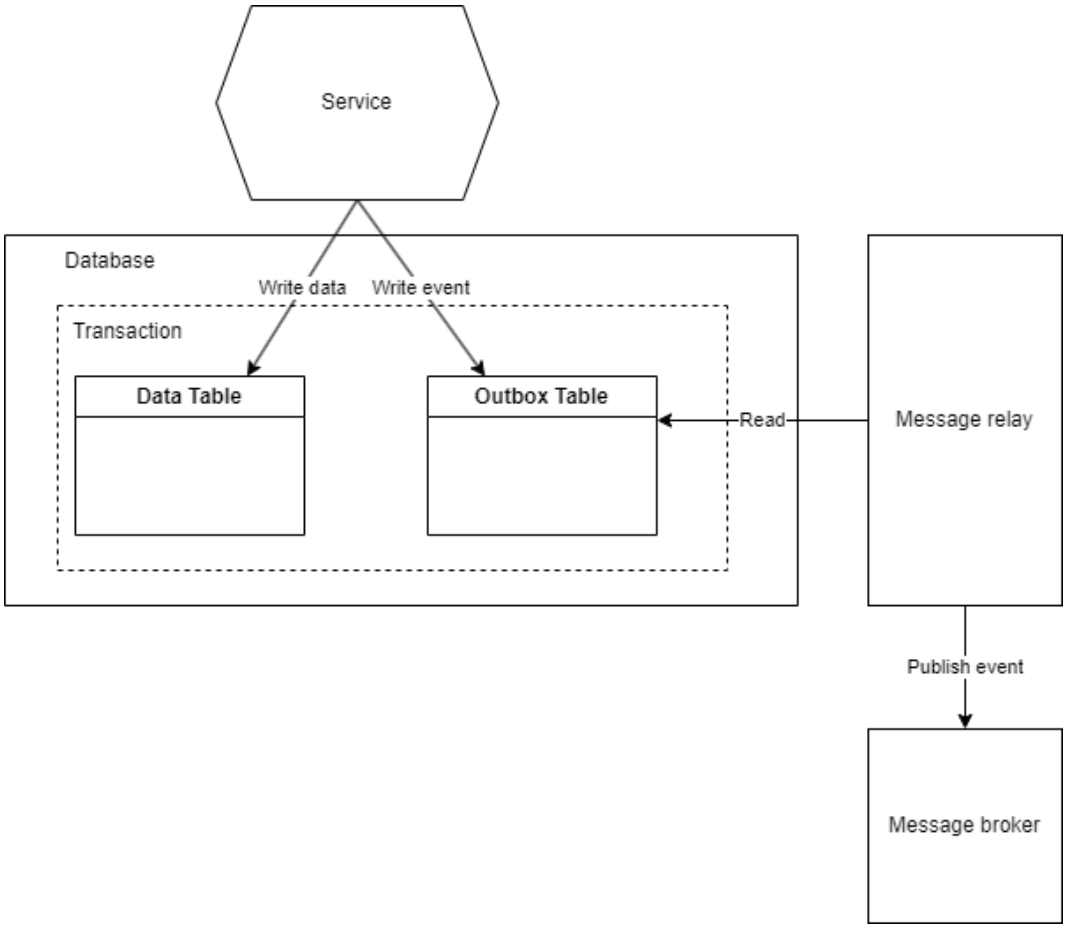

Figure 5.4. *Transactional outbox pattern*

A naive approach would be to simply write the changes to the database and then attempt to publish the event to the message broker. However, this way of doing things would require a dual write, writing to two different systems, Kafka and PostgreSQL in this case. This could lead to data inconsistencies; if making the local database changes would

succeed but publishing of the event would fail for any reason, perhaps due to a network error, the system would be left in an inconsistent state.

This method is called the transactional outbox pattern[33, Ch. 3]. As described, it provides a way to atomically update a local database and publish an event to a message broker. The pattern is shown in figure 5.4. A service can write its database changes and the corresponding event in a single transaction to the database. Another process, called the message relay, can read the events from the outbox table and relay them to the message broker.

5.3 Technology choices

In this section, reasons for using the specific technologies will be described. Some technology choices are mostly arbitrary while others have a specific reason for choosing them.

The choice of using asynchronous messaging instead of a synchronous communication scheme between the microservices is perhaps the most defining one of the whole architecture. Using a synchronous messaging scheme might not be a good idea. As outlined in Chapter 2, it would cause the microservices to be temporally coupled. Due to this, some retry logic would have to be implemented for the case that the invoked microservice wasn't available at the time of the invocation. Also, as a saga usually contains multiple calls between services, the long call chains could quickly lead to major resource congestion.

The reason for choosing Kafka as the message broker of choice is largely the existence of Kafka Connect. Kafka Connect provides a platform for connectors to other datasystems. This makes it possible with Debezium, a connector for Kafka Connect, to easily move the messages from the outbox table to Kafka. No other message broker provides such features out of the box. While it would be possible to roll out one's own way of moving the messages from the outbox table to the message broker, Debezium provides a purpose-built solution for said problem. Kafka Connect also provides scalability and fault tolerance, characteristics that would also need to be implemented for a custom built solution.

The choice of Stripe as the payment integration was largely arbitrary. Any payment integration would have been suitable. However, Stripe provides extensive documentation which helped with the implementation. The choice of having a payment integration in the first place was to discover any limitations pertaining to external API's in the context of the saga pattern.

The choice of PostgreSQL was mainly due to the author's familiarity with said database, and as such was largely arbitrary as well. One important aspect of PostgreSQL however, is that it works with Debezium. Any other relational database that works with Debezium could have been chosen instead.

The Spring framework and React were also chosen due to the author's familiarity with them. Stripe also had good documentation for React. However, from the point of view of the saga pattern, any backend and frontend framework could have been used.

5.4 Example saga description

In this section, the example saga will be looked at more closely. The steps of the saga will be explained and some important implementation details will be discussed.

The example saga starts once a notification about a completed payment from the payment processor is received. The example saga makes sure that either the user receives their tickets or is refunded if their seat reservation had expired. The example saga is visualized as a state machine in Figure 4.3.

5.4.1 Steps of the saga

Before the saga execution takes place, the user has registered a seat reservation through the user interface. This creates a new seat reservation in the seating service. The seat reservation is valid for 10 minutes. An identifier for the seat reservation is returned to the frontend application.

Next, the frontend application invokes the order service with the identifier of the seat reservation and their email for the ticket delivery. This creates a new order entity in the order service. The order service also invokes Stripe to create a new intent to pay for the amount of the tickets. The user is shown a form to fill in their credit card details to complete the payment.

The actual execution of the saga takes place once the user has completed the payment. The order service has a webhook endpoint to listen for notifications about payments from Stripe. Once the user has completed the payment, a notification is sent from stripe to the order service that a payment pertaining to a specific order has been completed.

The user completing the payment is considered the first transaction of the saga. The compensating transaction for the user completing the payment is to refund the payment. At this step, the webhook handler updates the order entity that was created when the user went to make the payment to indicate that it has been paid for and invokes the next transaction.

The next transaction in the sequence is to mark the seat reservation as purchased. This happens in the seating service. This step can fail as the seat reservation may have been expired during the time that the user takes to complete the payment. If so, the compensating transaction for the first transaction is invoked. Otherwise, the next transaction is invoked.

The third and final transaction is to deliver the tickets to the user. This happens in the order service. Delivery of the tickets is done via email.

5.4.2 Implementation details

In Chapter 4, two different implementation methods for the saga pattern were introduced; choreography and orchestration. In this example application, the choreography method is used. However, orchestration could have as well been used using the specific components and patterns described in this chapter. The reason for choosing choreography instead of orchestration for this example application is that it contains only a single saga which does not contain that many steps. For more complex sagas orchestration could be considered.

As outlined earlier, publishing messages is done using the transactional outbox pattern which makes it possible to atomically complete the step of the saga and instruct the other service to carry out the next step. The next service in the sequence then has to define a consumer endpoint that consumes the message from the topic and completes the saga step. In its consumer endpoint, the next service can then again use the outbox pattern to update some local state and atomically invoke the next saga participant.

In the saga, the logic for the steps is in the consumer endpoints. As an example, the consumer endpoint for step 2 of the saga is depicted in Function 5.1. It contains multiple noteworthy details.

On line 1 the listen method is annotated with the **@KafkaListener** annotation. Annotations are a feature of the Java programming language that are used to provide additional information about the program. The annotations can be used to, for example, generate boilerplate code.[36] The **@KafkaListener** annotation is an annotation that is defined in the Kafka client for Java. Simply put, it configures the listen method to run once a new message arrives in the **outbox.orderservice.OrderEntity** topic on Kafka.

The **@RetryableTopic** annotation on line 2 is another annotation from the Kafka client library. It configures retry topics for the topics that are assigned to the kafka listener method. In case an exception is thrown during the execution of the consumer method, the message is republished to the first retry topic and the listen method is retried. By default, 2 retry topics are created and in total 3 attempts are made to process the message. If all 3 attempts fail the message is published to a dead letter topic. An additional dead letter handler, not depicted here, is defined that contains the logic for handling a message that cannot be processed.

Without the **@RetryableTopic** configuration, an unprocessable message would block the handling of messages indefinitely. This is because messages are handled in order in Kafka. A possible source of such situation could be, for example, an error in the deserial-

ization logic. A noteworthy side-effect of retrying is that the message ordering is lost. In this saga the order of messages is not important but in other sagas it could be. If the order of messages was important, additional logic would need to be implemented to account for it.

The **@Transactional** annotation on line 6 configures the database operations that are executed inside the method to run inside a transaction. This is important in order to avoid the partial execution of the database operations.

In addition to the message payload, the consumer endpoint takes the message id as an argument. Each message has a unique identifier. On line 12, whether or not the message has already been processed is checked. If the message has already been processed, the function returns early. On the last line of the function, the message id is persisted in the service's local database to mark the message as processed.

Persisting the processed message's id and checking whether or not a message has already been processed makes the endpoint idempotent. The endpoint has to idempotent because of the at least once delivery guarantee of Kafka. The debezium connector might also in some situations broadcast a message twice. Having the endpoint be idempotent ensures that each message is processed only once.

The function that implements the actual logic of the saga step is invoked on line 16. The invoked function is depicted in Function 5.2.

On line 1, the function is also marked as **@Transactional**. The **Propagation.MANDATORY** argument means that database operations that are executed inside the function share the same transactional context as the one that is initialized in the consumer endpoint.

The function simply updates the state of the seat reservation to **finalized** if the seat reservation is still valid and publishes an update message to the outbox. If the seat reservation is expired, it also publishes a message to the outbox. The difference is that the status of the entity on the update message is **expired** if the seat reservation was expired, and **finalized** if the seat reservation was successfully finalized. Publishing the **expired** message corresponds to a failure in the saga, and publishing the **finalized** message corresponds to a successful execution of this step.

The order service has a similar consumer endpoint to listen for incoming changes to the seat reservation. Based on the status of the seat reservation, it can execute the correct action. If it receives an **expired** message, it calls Stripe to refund the user's payment, and if it receives a **finalized** message, it will deliver the tickets to the user.

A noteworthy detail is that the Stripe API for refunds is also idempotent. Due to the fact that the consumer endpoint can be entered multiple times, if the API was not idempotent, the user could be refunded more than once. This is something to consider if a saga step

Function 5.1. *Consumer endpoint*

```
@RetryableTopic
@KafkaListener(
  id = "SeatingService",
  topics = "outbox.orderservice.OrderEntity"
)
@Transactional
public void listen(
  @Payload String payload,
  @Header("id") String messageId
) {
  UUID messageUUID = UUID.fromString(messageId);
  if (processedMessageRepository.existsById(messageUUID)) {
    return;
  }
  OrderEntity orderEntity = deserializePayload(payload);
  seatReservationService
    .finalizeSeatReservation(
      orderEntity.getSeatReservationId()
  );
  processedMessageRepository.save(new ProcessedMessage(messageUUID));
}
```

has to call external APIs.

Function 5.2. *Saga step*

```
@Transactional(propagation = Propagation.MANDATORY)
public void finalizeSeatReservation(UUID seatReservationId) {
    SeatReservationEntity seatReservation =
      seatReservationRepository.findById(seatReservationId)
        .get();
    SeatReservationEntity.Status seatReservationStatus =
      seatReservation.getStatus();
    if (seatReservationStatus
      .equals(SeatReservationEntity.Status.RESERVED)) {
        seatReservation.setStatus(
          SeatReservationEntity.Status.FINALIZED
        );
        seatReservationRepository.save(seatReservation);
        outboxRepository.writeToOutbox(
                SeatReservationEntity.class,
                seatReservation.getId(),
                OutboxMessage.Type.UPDATE,
                seatReservation
        );
    } else if (
      seatReservationStatus.equals(
        SeatReservationEntity.Status.EXPIRED
      )
    ) {
        outboxRepository.writeToOutbox(
                SeatReservationEntity.class,
                seatReservation.getId(),
                OutboxMessage.Type.UPDATE,
                seatReservation
        );
    } else {
        throw new IllegalStateException(
          "SeatReservationEntity had an unexpected status"
        );
    }
}
```

6. EVALUATION OF EXAMPLE ARCHITECTURE

The aim of this thesis was to find out how to implement the saga pattern in a microservice architecture. The main focus was to find out a reliable and correctly working solution.

In this chapter, the proposed example architecture described in the previous section is evaluated. Firstly, whether or not the architecture adheres to the microservice architecture characteristics described in Chapter 2 is evaluated. Secondly, the reliability and correctness of the parts relating to the saga pattern are evaluated. Some evident drawbacks of the proposed architecture are also discussed.

6.1 Adherence to microservice architecture characteristics

The characteristics of a microservice architecture were described in Chapter 2. In this section whether or not the example architecture adheres to those characteristics.

First of the characteristics was independent deployability. As long as no changes are made to the schema of the messages sent to Kafka, each service can easily be independently deployed. If however the schema of the messages changes, changes to the consumer should be deployed first so that it can handle both old and new messages. After changes to the consumer are deployed the updated publisher can be deployed. Even though this adds a bit of complexity, the microservices can still be considered to be independently deployable.

Another characteristic was that services should be tolerant to failures of other services. Because the services communicate through Kafka, if one service is offline, it should have no bearing on another service as there is no direct communication between the two.

The characteristic of smart endpoints and dumb pipes is obviously satisfied. Kafka doesn't do anything else but routes the messages to their correct receivers. All the logic for handling the messages is in the consumer endpoints.

Another obviously satisfied characteristic is that each service has a dedicated database. Both microservices have their own database.

Finally, each service is modeled around a business domain instead of technical concerns. One service is dedicated to managing seats and another to managing payments and

orders. Whether or not this is a sensible dichotomy, however, is outside the scope of this thesis.

6.2 Correctness

Perhaps the most interesting characteristic to examine is whether or not the architecture supports the saga pattern correctly. What is meant by correctness in the context of the saga pattern is that either all the steps are completed as a whole, or if there is an identified failure, all the rollback steps are completed as a whole.

As identified in the previous chapter, there are a few characteristics that a step of the saga, either rollback or forward step, need to fulfill in order for the saga to work correctly. These are:

1. Idempotency. Each saga step has to be idempotent. If calling external systems during a saga step, their APIs also have to be idempotent.
2. All database operations are executed within a single transaction.
3. The outbox pattern is used to invoke the next participant.

Besides that, a point not discussed in the previous chapter but in Chapter 4, is that the lack of the isolation characteristic should be taken into consideration. If some database operation can modify an entity in a database that is in the middle of being modified by a saga, additional logic is required.

6.3 Reliability

Reliability in software architecture refers to the extent to which a system operates as expected within defined conditions and over a designated duration. This attribute encompasses several subcategories, including[10]:

1. Maturity: Assessing whether the software meets reliability requirements during regular operation, ensuring it functions consistently and predictably.
2. Availability: Verifying that the software remains operational and accessible, allowing users to interact with it when needed.
3. Fault tolerance: Evaluating the software's ability to function correctly despite hardware or software faults or failures, avoiding catastrophic disruptions.
4. Recoverability: Determining whether the software can recover from failures by restoring affected data and reinstating the system to its desired state.

In the following subsections the example architecture will be evaluated through the lens of these subcategories, all except maturity as maturity is hard to evaluate on a new system. An emphasis is obviously based on the functioning of the saga pattern.

6.3.1 Availability

The availability of the example saga as a whole depends on the availability of the microservices and their respective databases, Stripe, the SMTP server, Kafka and Kafka Connect. For the purpose of this thesis, Stripe and the SMTP server are not as interesting as they are specific only to the example saga.

An obvious way to increase availability is to introduce redundancy. Kafka provides the option of running as a cluster where messages can be replicated across multiple instances to provide redundancy[15]. Kafka Connect also provides clustering for means of achieving high availability[16]. In a similar manner, Postgresql also provides a way to replicate messages across multiple database servers in case one server fails[23].

The microservices are stateless, meaning they hold no state of their own. All the state information is stored either in the microservice's respective databases or Kafka. Statelessness provides the option of introducing load balancing that could route traffic to another instance if one became unavailable.

All in all, there are no obvious obstacles that would prevent the high availability of the system. All components provide, or could provide, high availability if needed.

6.3.2 Fault tolerance

Clustering and replication of Kafka and Postresql also provides fault tolerance. Replication of data provides protection against hardware failures.

An identified cause of possible catastrophic disruptions is an unprocessable message getting into Kafka. To guard against this, the dead lettering mechanism was added. It makes it possible to continue processing even in the case of an unprocessable message. Without it a manual intervention would be required which could halt processing for a long time.

6.3.3 Recoverability

Under the assumption that data is not lost from the data stores, the system should be able to recover the state of the saga and continue normal operation. A microservice performs the following operations during a saga step:

1. a message is pulled from Kafka by the service
2. If the message has already been processed, an acknowledgment to move to the next message is sent to Kafka and the saga step returns early
3. The service performs database operations within a transaction or calls an idempo-

tent API

4. An acknowledgment is sent to Kafka

If there is a failure at any point, the service starts the operations again from the beginning. Because the messages are pulled from Kafka by a service and Kafka keeps track of which message is next to be processed, a service can easily recover from a crash.

6.4 Drawbacks

The most evident drawback of the architecture is that there are a lot of components to maintain and monitor. A large organization would be required to run such an architecture. This will inevitably drive up the cost.

Another evident drawback is complexity. In this simple example of two services and one actual saga, the complexity is maintainable. If however the architecture consisted of many more services and implemented multiple sagas, it could quickly turn unmaintainable. The lack of isolation of the saga pattern could easily cause hard to detect bugs.

Due to the large number of components involved in a saga, testing is quite laborious. The setup of integration tests requires intricate deployments.

7. CONCLUSION

The aim of this thesis was to study how the saga pattern might be implemented in a microservice architecture in order to achieve a reliable method of integrating microservices with each other. The study was mainly an empirical one, where a software architecture that fits the microservice architecture characteristics was created and evaluated.

Arbitrary functional requirements were made up for the example system in order to get a more practical understanding of what kind of hurdles might be encountered in adopting the saga pattern. However, the findings of this thesis should be generalisable to meet other functional requirements as well.

A large defining factor of this thesis was the choice of asynchronous messaging over a synchronous communication style for the example architecture. Additional research would be required in order to determine if, and how the saga pattern could be implemented reliably using a synchronous communication style.

Although Kafka was chosen as the message broker for the example architecture, an architecture making use of another message broker technology could make use of similar patterns and methods as described in this thesis. The only condition is that the message broker should support the at least once delivery of messages. Similarly, PostgreSQL could be exchanged for another relational database technology.

Another choice was to implement a choreography based saga instead of an orchestration based saga. However, the findings of this thesis should be helpful in implementing an orchestration based saga as well.

The main findings of this thesis pertain to how an individual step of the saga should be conducted. There were multiple things to consider in order to have a saga work reliably.

One hurdle to overcome is that a saga step should atomically be able to make its local changes and to invoke the next participant by publishing a message to the message broker. The remedy to this problem, discussed in more detail in Chapter 5, was to use the outbox pattern. Instead of directly attempting to publish the message, the message is written to a microservice's local database during the same transaction that the local database changes are made. The message can then be picked up by another running process, Kafka Connect with the Debezium connector, that actually publishes the mes-

sage.

Due to the fact that a message broker can only provide an at least once delivery guarantee for a message, a saga step should be able to ignore duplicate messages in order to avoid processing the same message multiple times. The solution to this problem was to mark a message as processed in the microservice's local database during the same transaction that the local changes are made. A saga step handler can then ignore the duplicate messages it receives.

An observation was made about calling external APIs during a saga step's execution. The API needs to provide an idempotency guarantee.

Finally, in order to provide better fault tolerance, a dead lettering mechanism was added. Due to the fact that messages are processed in order, an erroneous message could completely halt processing. With the dead lettering mechanism, possible erroneous messages are directed to a dead letter topic from which they can be examined further.

One aspect that was not looked into in more detail in the example architecture is the lack of the isolation property of sagas. This was, however, somewhat discussed in Chapter 4.

Overall, the saga pattern can provide a reliable way to integrate microservices. To achieve reliability, however, a lot of properly maintained infrastructure is required. Moreover, the lack of the isolation property can introduce a lot of complexity.

REFERENCES

[1] VMware Inc. or its affiliates. *Clustering RabbitMQ*. URL: https://www.rabbitmq.com/clustering.html (visited on 01/29/2023).

[2] VMware Inc. or its affiliates. *Quorum Queues RabbitMQ*. URL: https://www.rabbitmq.com/quorum-queues.html (visited on 01/29/2023).

[3] Baeldung. *Saga pattern in Microservices*. URL: https://www.baeldung.com/cs/saga-pattern-microservices (visited on 02/09/2023).

[4] De Nederlandsche Bank. *What is a payment service provider?* URL: https://www.dnb.nl/en/sector-information/supervision-sectors/payment-institutions/licensing-requirement-for-payment-service-providers-overview/what-is-a-payment-service-provider/ (visited on 04/17/2023).

[5] Philip A. Bernstein. *Principles of transaction processing*. eng. 2nd ed. The Morgan Kaufmann series in data management systems. Burlington, MA: Morgan Kaufmann Publishers, 2009. ISBN: 1-282-16931-9.

[6] Simon Brownk. *The C4 model for visualizing software architecture*. URL: https://c4model.com/ (visited on 04/14/2023).

[7] Morgan Bruce. *Microservices in action*. eng. 1st edition. Shelter Island, New York: Manning, 2019. ISBN: 1-63835-606-8.

[8] Brendan Burns. *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*. eng. Sebastopol: O'Reilly Media, Incorporated, 2018. ISBN: 9781491983645.

[9] Debezium Community. *Debezium Features Debezium*. URL: https://debezium.io/documentation/reference/stable/features.html (visited on 09/07/2023).

[10] Neal Ford and Mark Richards. *Fundamentals of Software Architecture*. eng. O'Reilly Media, Inc, 2020. ISBN: 9781492043447.

[11] Apache Software Foundation. *Clustering ActiveMQ*. URL: https://activemq.apache.org/clustering (visited on 01/29/2023).

[12] Apache Software Foundation. *Introduction Apache Kafka*. URL: https://kafka.apache.org/documentation/#introduction (visited on 01/29/2023).

[13] Apache Software Foundation. *Kafka Connect Apache Kafka*. URL: https://kafka.apache.org/documentation/#connect (visited on 07/27/2023).

[14] Apache Software Foundation. *Message Delivery Semantics Apache Kafka*. URL: https://kafka.apache.org/documentation/#semantics (visited on 01/21/2023).

[15] Apache Software Foundation. *Replication Apache Kafka*. URL: https://kafka.apache.org/documentation/#replication (visited on 01/29/2023).

[16] Apache Software Foundation. *Running Kafka Connect Apache Kafka*. URL: https://kafka.apache.org/documentation/#connect_running (visited on 07/27/2023).

[17] Martin Fowler. *Event collaboration*. June 19, 2006. URL: https://martinfowler.com/eaaDev/EventCollaboration.html (visited on 01/10/2023).

[18] A. Fox and E.A. Brewer. “Harvest, yield, and scalable tolerant systems”. eng. In: *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*. IEEE, 1999, pp. 174–178. ISBN: 9780769502373.

[19] Lars Frank and Torben U. Zahle. “Semantic acid properties in multidatabases using remote procedure calls and update propagations”. eng. In: *Software, practice & experience* 28.1 (1998), pp. 77–98. ISSN: 0038-0644.

[20] Hector Garcia-Molina and Kenneth Salem. “Sagas”. eng. In: *SIGMOD record* 16.3 (1987), pp. 249–259. ISSN: 0163-5808.

[21] Seth Gilbert and Nancy Lynch. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”. eng. In: *SIGACT news* 33.2 (2002), pp. 51–59. ISSN: 0163-5700.

[22] The PostgreSQL Global Development Group. *PostgreSQL*. URL: https://www.postgresql.org/about/ (visited on 09/08/2023).

[23] The PostgreSQL Global Development Group. *Replication Postgresql*. URL: https://www.postgresql.org/docs/current/runtime-config-replication.html (visited on 07/27/2023).

[24] Gregor Hohpe and Bobby Woolf. *Competing Consumers*. URL: https://www.enterpriseintegrationpatterns.com/patterns/messaging/CompetingConsumers.html (visited on 01/21/2023).

[25] Gregor. Hohpe. *Enterprise integration patterns designing, building, and deploying messaging solutions*. eng. 1st edition. Boston: Addison-Wesley, 2003. ISBN: 0-13-306509-X.

[26] Frederick P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering*. eng. Addison-Wesley Professional, 1995. ISBN: 0201835959.

[27] Dr. John C. Klensin. *Simple Mail Transfer Protocol*. RFC 5321. Oct. 2008. DOI: 10.17487/RFC5321. URL: https://www.rfc-editor.org/info/rfc5321.

[28] Martin Kleppmann. *Designing data-intensive applications : the big ideas behind reliable, scalable, and maintainable systems*. eng. First edition. Beijing: O’Reilly, 2017. ISBN: 978-1-449-37332-0.

[29] James Lewis and Martin Fowler. *Microservices*. Mar. 25, 2014. URL: https://martinfowler.com/articles/microservices.html (visited on 12/19/2022).

[30] Microsoft. *Saga pattern*. URL: https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga (visited on 02/09/2023).

[31] Niall Richard Murphy et al. *Site Reliability Engineering: How Google Runs Production Systems*. eng. Sebastopol: O’Reilly Media, Incorporated, 2016. ISBN: 149192912X.

[32] Sam Newman. *Building Microservices, 2nd Edition*. eng. O'Reilly Media, Inc, 2021. ISBN: 9781492034018.

[33] Chris Richardson. *Microservices Patterns*. eng. Manning Publications, 2018. ISBN: 9781617294549.

[34] Arnon Rotem-Gal-Oz. "Fallacies of Distributed Computing Explained". In: *Doctor Dobbs Journal* (Jan. 2008).

[35] Gary Russell et al. *Spring for Apache Kafka*. URL: https://docs.spring.io/spring-kafka/reference/html/ (visited on 07/16/2023).

[36] JetBrains s.r.o. *Annotations*. URL: https://www.jetbrains.com/help/idea/annotating-source-code.html#external-annotations (visited on 07/16/2023).

[37] Meta Open Source. *React*. URL: https://react.dev/ (visited on 09/08/2023).

[38] Maarten van Steen and Andrew S. Tanenbaum. *Distributed systems*. 3rd edition. 2017. URL: distributed-systems.net.

[39] Inc. Stripe. *Stripe*. URL: https://stripe.com/ (visited on 09/08/2023).

[40] Inc. or its affiliates VMware. *Spring Boot*. URL: https://spring.io/projects/spring-boot (visited on 09/08/2023).

[41] Inc. or its affiliates VMware. *Spring Data JPA*. URL: https://spring.io/projects/spring-data-jpa (visited on 09/08/2023).

[42] Inc. or its affiliates VMware. *Spring Framework*. URL: https://spring.io/projects/spring-framework (visited on 09/08/2023).

[43] Gerhard. Weikum. *Transactional information systems theory, algorithms, and the practice of concurrency control and recovery*. eng. 1st edition. The Morgan Kaufmann Series in Data Management Systems. San Francisco: Morgan Kaufmann, 2002. ISBN: 1-281-02451-1.