dkorez.dev

Architectural Patterns in Microservices

Dejan Korez

December, 2024

Draft version: 0.2

Table of content

1. Scalability & Data Consistency in Microservices

In the world of microservice architecture, scalability and data consistency are among the most critical aspects to address. Scalability is the ability of the system to handle increased workload. There are two ways of scaling systems:

- vertical scaling - adding more resources, meaning increasing memory or CPU
- horizontal scaling - distributing the workload across multiple services

Vertical scaling is useful to some point, but we cannot increase resources indefinitely. Therefore it is often required to distribute the workload and data distribution to make the system scalable. Such systems need to handle massive workloads and diverse functional requirements without compromising data integrity. Interservice communication is crucial here, and direct communication between the services can lead to tight coupling which might negatively affect the scalability. Event-driven architecture and communication via message brokers allows services to operate independently and reduces dependencies among them.

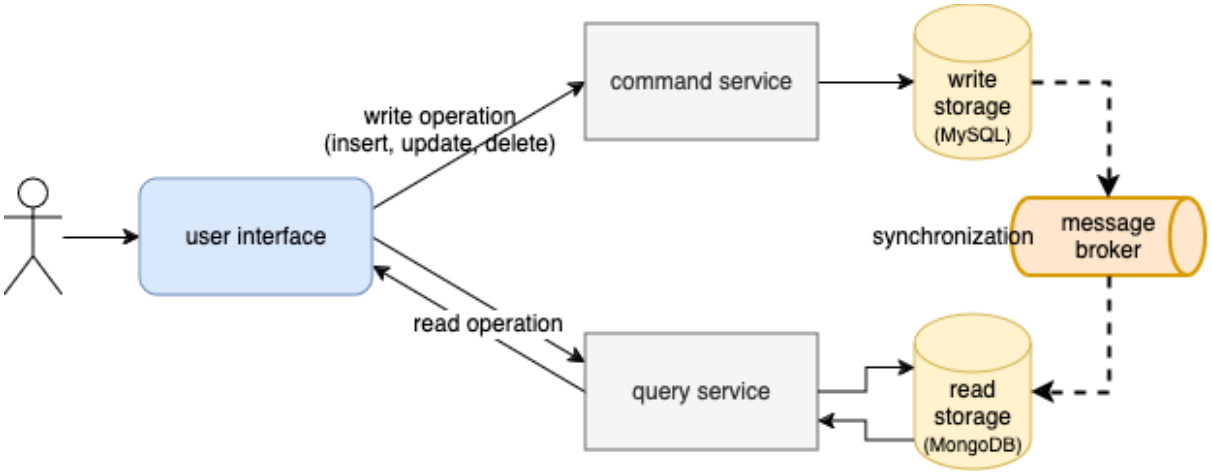
In this chapter we will take a look at the design patterns which use this style to interservice communication and aim to solve the scalability and data consistency challenges in microservices. We will start with the Command Query Responsibility Segregation or CQRS, a pattern that separates read and write operations. We will continue with Event Sourcing which captures all the changes of the application state as an immutable record of events. And finally the Change Data Capture or CDC, a mechanism that captures changes of a database and propagates them in near real-time across the system.

1.1. Command Query Responsibility Segregation - CQRS

In the traditional monolith systems all the operations are performed on a single database. While it is very simple, it has some limitations, particularly scalability and performance related. In case there are multiple operations that are running concurrently, for example if we want to read some data while the write operation has

not finished yet, the read operation will have to wait. Also if we take a look at the database query that needs to join several tables, it can cause deadlocks and the database will not be able to process all the transactions.

The Command Query Responsibility Segregation (CQRS) tries to solve those issues and it is one of the most important patterns in the microservice architecture. It is used to avoid complex queries and ineffective relations between tables by splitting read and write operations. This principle follows the separation of concerns principle. Usually two separated databases are used in this pattern, one for reading and the other for writing data. A good practice here is to have those two databases physically separated and each one of them is customized to its needs, for example a relational database for writing data and NoSQL database for reading. The diagram below shows the high-level overview of the CQRS pattern.



Having two independent databases has another advantage, that is we can independently scale them. For example if we have a system that usually requires more read operations, the database for reading can be scaled independently of the write database. It is just a matter of adding additional replicas for a read database. Moreover, the data model for read operations can be customized. With this approach, we can optimize models to reduce the number of relations and we can easily extend the models with new fields.

When having separated databases for reading and writing, the main concern is how to synchronize the databases. Ideally we want to keep the data synchronized and consistent all the time. The most convenient solution lies in the event-driven

architecture which is based on event publishing. With each write operation an event is generated and published to the message broker. After it is published, a service consumes this event and updates the read database. However we have to keep in mind that the changes are not instant. There is a slight delay between publishing and consuming the event. This is known as eventual consistency.

The CQRS pattern consists of two concepts: commands and events. Commands are used to change the entity state which triggers writing to a database. The events are used to track changes and transfer those changes to the other parts of the system, components responsible for reading data. Operations are split into two parts: the command and the query layer. Commands are used to create, update or delete data, while queries are used to retrieve data.

Let’s take a look at some of the benefits CQRS pattern has:

- It enables efficient implementation of the queries from multiple sources. The read part of the data storage is optimized for efficient reading operations, and the event-driven architecture takes care that the data is synchronized. This enables us to get rid of the consuming reading operations from multiple sources

- Ensures the system is highly scalable. Because read and write data storage are separated, they can be scaled independently

- It brings flexibility regarding data models and queries, and loose coupling. Each service can modify the models so that the effective queries for particular needs can be easily implemented

- It uses the separation of concerns principle. In the CQRS pattern, the commands and queries are separated which eases the maintenance and improves the scalability of each component in the system

- Different technologies can be used in this pattern. It is not necessary for reading data storage to use the same technology as the writing one.

On the other hand, there are some drawbacks. The architecture is more complex, because developers need to implement additional services for queries, which need

to be maintained. One of the biggest drawbacks of the CQRS pattern is the latency between commands and queries. A possible scenario which can happen is when the application updates its state and the changes are not synchronized in time. The query that reads data can return older versions of the data. To mitigate this issue, we can use the principle of versioning. When updating the entity, the version field is increased with each update and returned to the client in the queries.

Use cases

The CQRS pattern is useful in high-traffic systems and in cases where many users are accessing the same data in parallel. It is beneficial when we need to optimize read operation and we are dealing with retrieving data from multiple sources. In this case, the CQRS pattern can be applied to consolidate data read models and limit complex queries. On the other hand, this pattern is not suitable in scenarios where business rules are simple or when we are dealing with basic CRUD operations for data access because of the complexity of this pattern, which may outweigh the benefits of its use in such straightforward cases.

Combining CQRS with other architectural patterns

The CQRS works well in a combination with Event Sourcing pattern. The command level of the CQRS pattern can be used to process changes and generate events that can be used to create an application state. Another benefit of combining those two patterns is that the application log doesn’t need to be queried directly. We can use the projections, which are part of the query level of the CQRS pattern. This enables us to use application log for auditory purposes and is suitable in cases where a high amount of changes are written into the log.

1.2 Event Sourcing

The trouble with the traditional approach to persistence is that it maps classes to database tables. For example, the order aggregate is mapped to the order table and the items on an order are mapped to the order_item table. This leads to a so-called object-relational impedance mismatch, the difference between a tabular relational

schema and the graph structure of the rich domain model with complex relationships. Another limitation is that it only stores the current state of an aggregate. The previous state is lost after the aggregate is updated and we need additional mechanisms to implement regulatory instruments and audit logging. And lastly, the traditional approach doesn't support publishing domain events, which are useful for synchronizing data in microservices. That is where the event sourcing pattern comes in.

Event Sourcing is a solution that enables us to maintain data changes as a sequence of immutable events. Each event is added to the data storage as a new record. In contrast to the basic CRUD operation, the events are used for each operation, and they are added to the data storage, also called an event log or journal. It can be a click of a button, sending a request to another service or a purchase in ecommerce, each of those actions that changes the application's state, are events. We need the events for tracking, auditing, logging, etc… so the idea of the event sourcing is to store the events.

Events are stored as immutable records in the event store, meaning the new events are added as a new record and cannot be changed. However, the effect of an event can be altered by adding another event that will overwrite the original event. For example, if we add an event OrderPlaced with incorrect total amount and we want to change this event, we have to add OrderVoided and another OrderPlaced with the correct total amount. Those characteristics can be used for creating application's snapshots, auditing purposes, tracking or restoring data. Replaying the events from the event log also enables us to rebuild the application's state at any time, and the current state is presented from all of the events. This approach allows us to track the changes in the history and reproduce errors from the previous state.

Let's take a look at some of the fundamental topics in the event sourcing:

- events - represents something that happened in the business domain. They are considered as the source of truth, because the current state of the application is derived from the events. Events are immutable and usually contain unique metadata such as the timestamps, unique identifier (ID) and

the event name. Besides that, data which is populated with the state, is also present in the event model.

- event store - is an event-native database where events are stored. Those kinds of databases are different from traditional databases. They are designed to store history of changes and the state is represented by the log of events. New events are appended to the previous event and are stored in chronological order. Events are immutable and cannot be changed.

- event streams - stores the events that refer to a particular domain and present the source of truth for the domain objects and contain the full history of the changes.

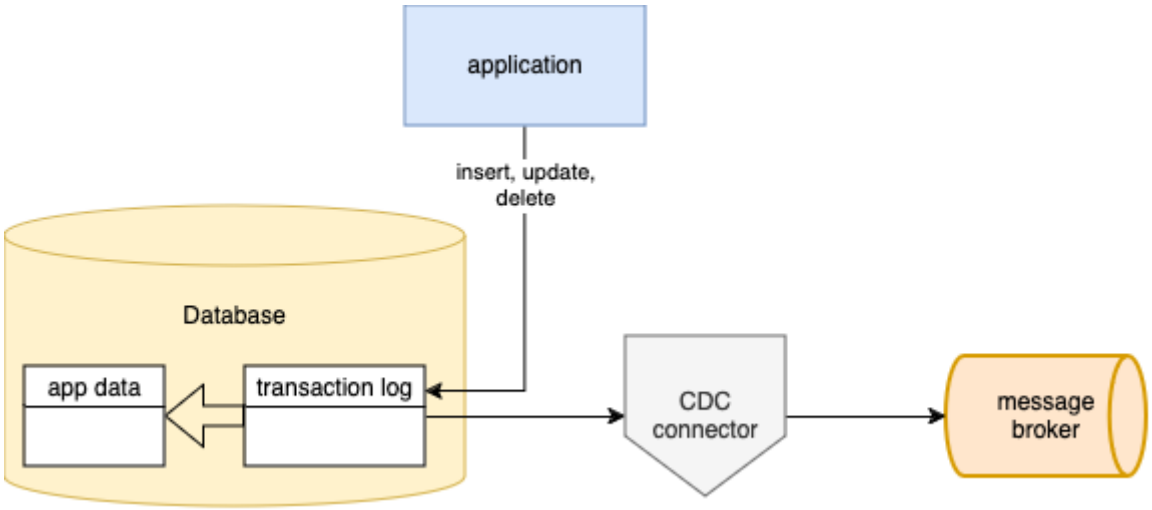
- projections - provide a view on the event based data-model. Projections should not be confused with the state. The state is derived from the events; projections are an interpretation of the raw data. They are methods of populating our read models, for example to create invoices and reports in financial applications.

Event sourcing is useful in the financial industry, such as banks, trading platforms and insurances. This industry requires mechanisms that guarantee high consistency of the data and enable them regulatory and revision activities. It is also suitable in the logistics, transport and retail industry for solutions that require up-to-data record and for tracking various user activities.

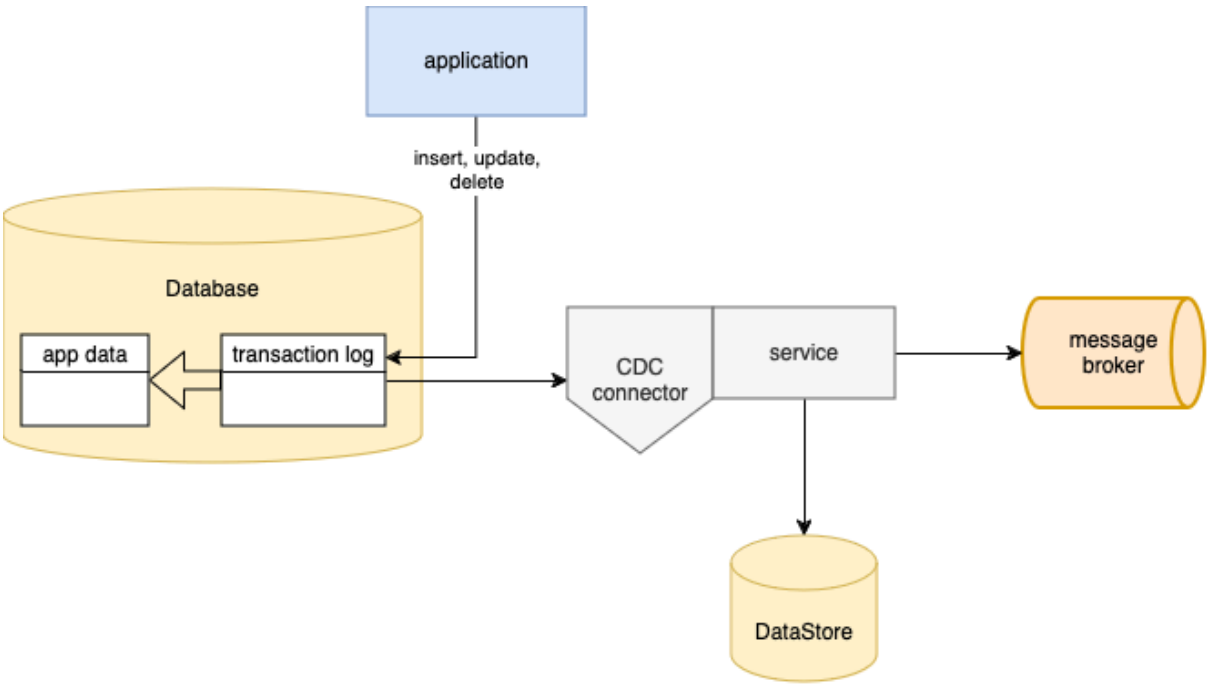
1.3 Change Data Capture - CDC

Some databases support an option to automatically publish the changes on a database. This solution does not need additional tables to keep track of the changes. It is called Change Data Capture (CDC). This pattern uses database transaction log or a similar mechanism to track change events. This concept externalizes the transaction log of a database and forwards those events to other consumers. It allows the application state to be externalized and synchronized with external stores.

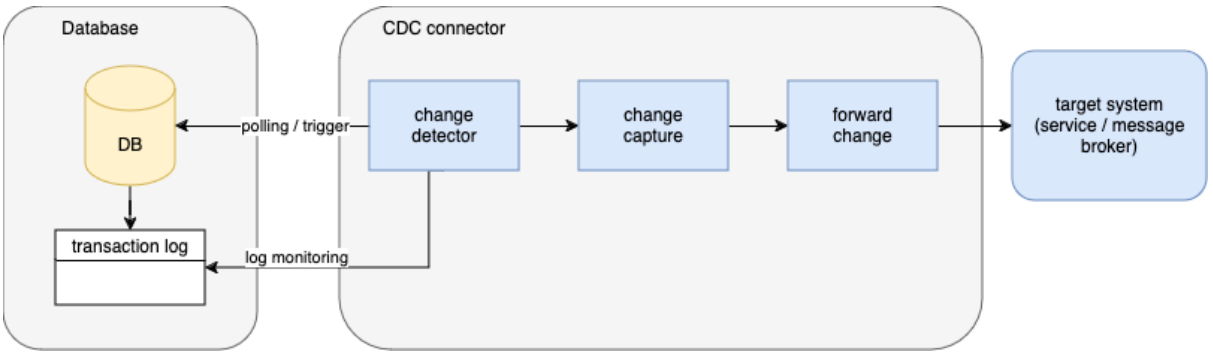
The implementation usually consists of the following characteristics: an external process that reads the transaction log of a database and forwards those events in the form of a message. As we can see in a diagram below, there are multiple options on how this process can be implemented. The CDC connector can be a separate component that scans the database for changes and pushes those events to the message broker, from where other systems can process them.



In the other case, a CDC connector is embedded into a client service which processes the events. The service can persist events directly to a data store, sends them to the message broker, or a combination of both options.



CDC captures the changes on a database in real time. It operates in three phases: detecting changes, capturing changes and forwarding changes. To detect changes there are several options available and some modern databases support mechanisms to automatically detect the changes. A diagram below demonstrates the overview of the CDC change detection.



But the CDC can also be implemented on any kind of database, however we need to implement some additional methods to detect the changes if the chosen database does not support this:

- monitoring the transaction log of a database. Since every database logs its transactions, we can implement log scanners that can identify any changes

- periodically polling data from tables. This usually requires us to query the database and compare timestamp and version fields

- use database triggers. To use this approach, we have to define triggers on a database. This mechanism affects the overall system performance as it requires transformations of the changes into the events, duplications for the records and additional maintenance work.

In the cloud native architecture, triggers are often used in a combination with the event streaming. When a database change triggers a log, those changes are sent to the CDC in streams. In the process of event streaming, additional operations such as aggregations and transformations can be applied, and the processing of data occurs in real-time. There are also tools which can capture the changes on a database and push those changes to other components or systems. Debezium is one of them.

There are two possible options to migrate captured data to the data warehouse. First one is a direct migration, and another is migration with transformation. In the first case, data is sent to other systems as it is, while in the second case, additional transformations are applied. Data can be sent directly to the service that will store data in a data warehouse, or event processing tools can be used. The most common option is usage of message brokers such as Apache Kafka, Apache Pulsar or RabbitMQ.

The CDC offers many benefits, such as maintaining the source of truth in the transaction log, effective and real-time data retrieval, real-time analysis and consistent data synchronization.

2. Managing Transactions and Consistency

Unlike in monolithic systems, where a single database often ensures atomicity, microservices operate with multiple databases and independent services. This approach brings new challenges like how to manage transactions and ensure data consistency, integrity and reliability.

We will start with the explanation on how the transactions works and how the ACID principle ensures data integrity on traditional monolithic systems. In distributed systems such as microservices, we will explore the limitations of relying on ACID principles with the distributed transactions that span across multiple services. Microservices can leverage mechanisms like two-phase commit but it’s not recommended as we will explain later.

A more suitable approach comes from the event-driven architecture. We will explore patterns like saga, which coordinates transactions across services, and the outbox pattern, which guarantees successful message delivery. Lastly we will take a look at receiving and processing messages, possible issues that duplicated messages can bring, and how to handle them. With this topic, we will present an inbox pattern, which is one of the mechanisms that handles effective message processing in distributed systems.

2.1 Transaction management

In software architecture, a transaction represents a sequence of operations treated as a single unit. It is independent of other transactions. This ensures that all operations either succeed or fail, ensuring the data integrity is maintained. While the transaction is changing data, the data might be inconsistent, and other transactions will have to wait while the changing transaction finishes. When a transaction commits, the changes are finalized and permanently stored in a database. Committed transactions cannot be undone. This is because the data is visible to other transactions after being committed, and other transactions rely on this data.

Transaction guarantees

To ensure data integrity transactions must follow the ACID principle. It is an acronym which stands for:

- Atomicity - ensures that all the operations in the transaction are treated as a single unit, meaning either all of the operations will succeed if committed or the transaction will be rolled back
- Consistency - ensures that the transaction will leave the data in a consistent state during the execution, no matter if it is committed or rolled back
- Isolation - ensures that the changes of the ongoing transaction will not affect the data accessed by another transaction
- Durability - ensures that when a transaction is committed, all the changes are permanently stored in a database

A transaction that executes its operations on a single service is called local transaction. In local transactions ACID properties can be applied in a straightforward manner. However, in microservice architecture, the services often have their own database, and transactions span across multiple services. Those are called distributed transactions, and ACID principles cannot be applied. It would require locking databases in all the services and all the services to be available during the transaction, which would significantly decrease the performance.

An alternative approach is called BASE. It focuses on high availability and allows eventual consistency. The idea of this approach is to cut the whole distributed transaction into smaller pieces, each of them still following the ACID principle. To synchronize all the smaller, ACID transactions across the whole system, messaging is used. This means that the changes between the services will be propagated by exchanging messages through the message broker. This is an asynchronous operation and there is a delay between the messages being sent and consumed by another service, meaning that there is also a delay between the first and last ACID transaction. And that explains what eventual consistency is; the data can be temporarily inconsistent due to delays and should eventually synchronize and become consistent. The acronym BASE means:

- Basically Available, meaning the system is available all the time, even in case of failures,
- Soft state, meaning the state of the system can change even thought there are no pending updates
- Eventually consistent, meaning the data will eventually synchronize across the whole system

BASE also meets the constraints of the CAP theorem, which states that in distributed systems all three of consistency, availability and partition tolerance cannot be guaranteed at the same time. CAP stands for:

- consistency - all the microservice nodes have the same view in the data
- availability - every service can read and write data to any other service at any time
- partition tolerance - the system as a whole works despite the physical partitions between the services

Since the microservices should be physically partitioned by default, in the microservice architecture, CAP theorem states that only either consistency or availability can be guaranteed.

Transaction models

The coordination of individual transactions under the scope of an enclosing transaction is described with the transaction models. The simplest are so-called flat transactions, where the entire transaction is rolled back if one of the steps fails. In this model, the steps of the transaction are executed sequentially. In nested transactions, atomic transactions are embedded in other transactions. This model allows transactions to be nested at multiple levels, where each sub-transaction is encapsulated by the parent transaction. It also controls whether the sub-transactions will be reverted. Sagas are similar to the nested transaction. The difference is that in sagas, each transaction has additional compensating transaction which reverts changes if any of the steps fails. Last model describes chained transactions. In this

model, each transaction relies on the previous one. When each transaction is executed, it commits its changes and those changes are available to other transactions. On the downside, if one of the steps fails, only the failed one can be reverted, as previous steps are committed.

Concurrency in database transactions

Concurrency is a characteristic of a database system where simultaneous transactions operate on shared data without affecting data integrity. Without proper concurrency control, following anomalies can occur:

- lost updates - occurs when two concurrent transactions update the same data, and the one transaction's changes are overwritten by the other
- dirty reads - happens when a transaction reads uncommitted changes made another transaction
- non-repeatable reads - occur when a transaction reads the same data multiple times and gets different results, due to another transaction modifying the data during the execution of the first transaction
- phantom reads - occur when a transaction executes the same query twice but gets different results because another transaction inserted or deleted rows between the queries

To prevent those anomalies, database systems have implemented mechanisms to mitigate the risk of anomalies and define transactional integrity. Those mechanisms are called isolation levels and define the degree to which the operations of one transaction are isolated from the operations of another. Higher isolation levels offer more consistency but the performance and scalability may be affected. Common levels are:

- read uncommitted - allows transactions to read data modified by other uncommitted transactions. This level prevents lost updates, but permits dirty reads, non-repeatable and phantom reads, and is achieved by using write locks

- read committed - ensures that a transaction reads only committed data. This level also prevents dirty reads, but non-repeatable and phantom reads might still occur

- repeatable read - read transactions block write transactions but not other read transactions, while write transactions block all transactions. This isolation level prevents lost updates, dirty reads and non-repeatable reads, while phantom reads might still occur

- serializable - is the strictest level of isolation, where all transactions are fully isolated by mechanisms like full table locking or multi-version control. This level prevents all anomalies, but it can significantly impact scalability

Choosing the appropriate isolation level balances integrity and performance based on application requirements. For example, higher isolation levels like serializable enforce stricter consistency but downgrade performance. Lower isolation levels provide higher throughput but expose risk of anomalies. This makes explicit locking critical for managing concurrency, as it helps enforce isolation levels. Database locking is a mechanism that restricts access to a resource in a database. There are two categories of locking mechanisms:

- optimistic locking pairs well with lower isolation levels where conflicts are rare. It allows concurrent access to data with conflicts being detected during commit and enables faster processing of transactions. Versioning, a mechanism which increments version on updates, is commonly used to detect conflicts.

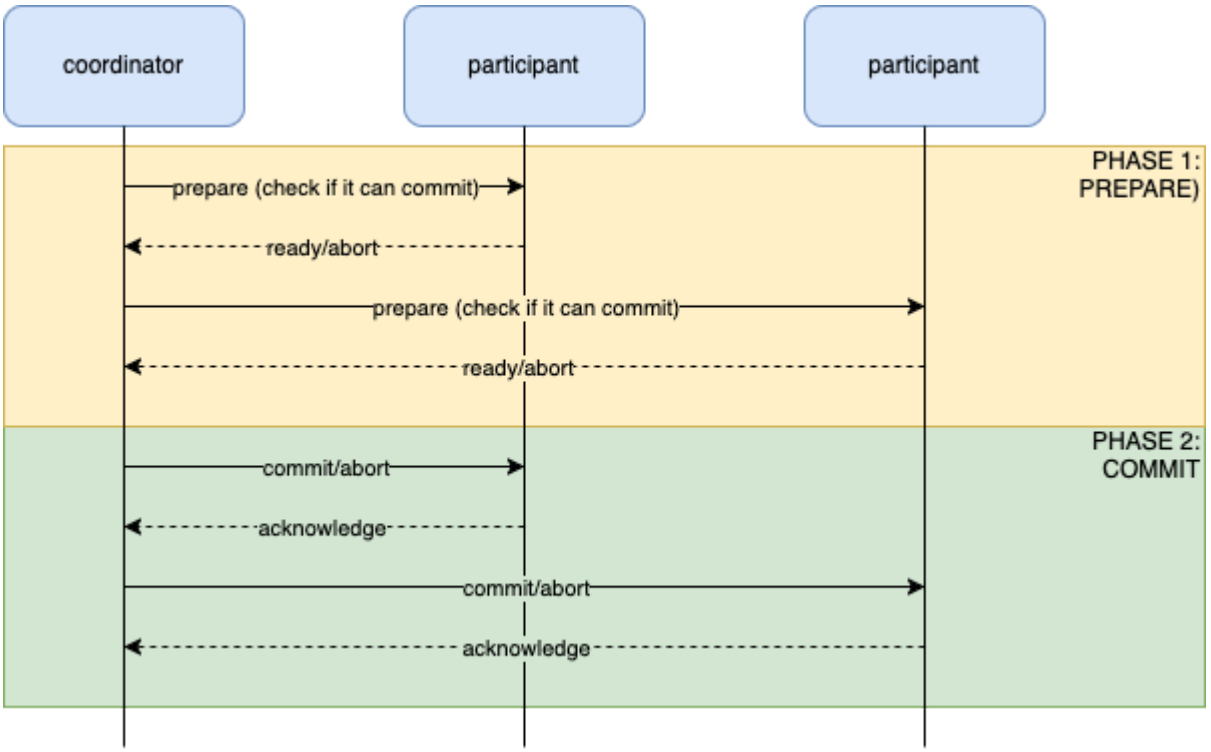
- pessimistic locking aligns more closely with higher isolation levels as it prevents concurrent access to data. It applies locks, which prevents other transactions to access data, and ensures operations are safely executed within the locking transaction

Two-phase commit

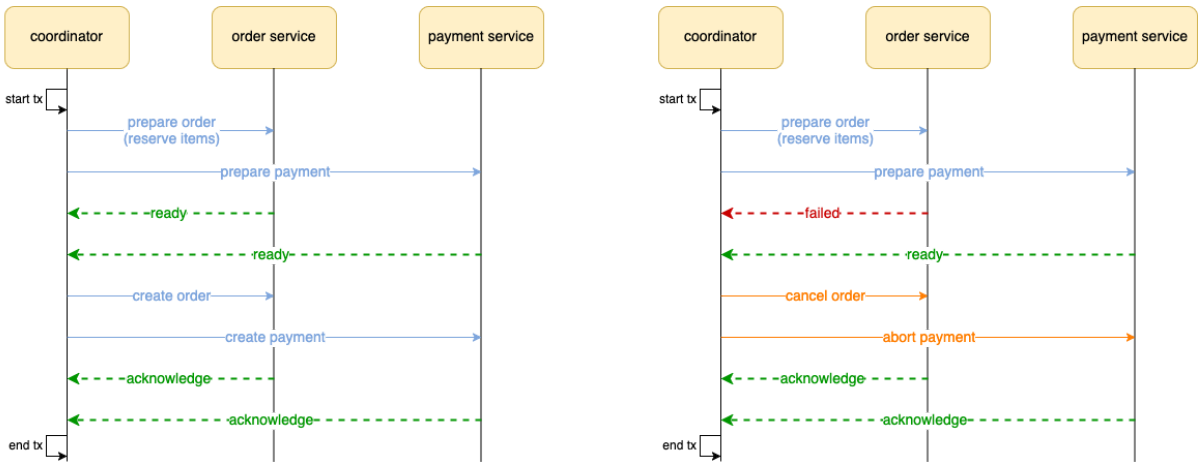
One way of applying ACID principles in the microservice architecture is two-phase commit. As the name implies, it ensures updates occur in two phases:

- prepare phase - the transaction manager or coordinator checks if all the participants can execute the commit
- commit phase - the commit is carried out

In the first phase, all the participants check if they can guarantee the execution of the update and respond to the coordinator. The commit phase only happens if all the participants respond with a successful response. If any of the participants cannot proceed, the transaction is aborted.



Let's take a look at the scenario of an e-commerce shop. A diagram below demonstrates how a two-phase commit works. A coordinator checks if an order can be carried out by asking the order service if the order can be placed and the payment service if a payment can be done. We have a successful case on the left side, both services responded with ready and in the second phase, two commit commands are set to actually execute the order and the payment. On the right side we have a case in which the order service responded with a failure. This can be due to database issues, business rule violations or service issues. In this case, the order is canceled by sending the abort/rollback command to both of the services.



Although this mechanism guarantees high data consistency, it is not recommended for microservice based systems. Two-phase commit works in a synchronous mode, meaning the resource being modified will be locked during the duration of the ongoing transaction and will not be available for other transactions. Furthermore, some modern database technologies and message brokers don't support distributed transactions. Additionally, it has the following limitations:

- transaction manager can represent a single point of failure
- if one of the participants fails to respond, the entire transaction will be blocked
- a commit can still fail; if one participant responded that it can commit the transaction, 2PC assumes it can definitely commit the transaction, which is not the case all the time
- due to inter-service communication and dependency on the transaction manager the protocol is complex, error-prone and slow by design, which can cause scalability issues

Recommended approach to handle distributed transactions in the microservice architecture implies using asynchronous, event-based communication with local transactions. In this type of architecture, a service responsible for updating a database, will publish an event to the event bus. From there, consumer services can consume those events and update their databases in the scope of a local transaction.

Transaction boundaries

When designing distributed systems, such as microservices, we can't look at the data storage as a one single relational database with ACID properties. Another issue with the distributed systems is unreliable network and asynchronous communication. Additionally, implementing two-phase commits across multiple services to solve distributed data ignores the autonomy of the services and causes scalability issues. Transaction boundaries refer to the smallest unit of atomicity required to maintain business rules. When designing a system, we want to make transactional boundaries as small as possible, ideally a single transaction on a single object.

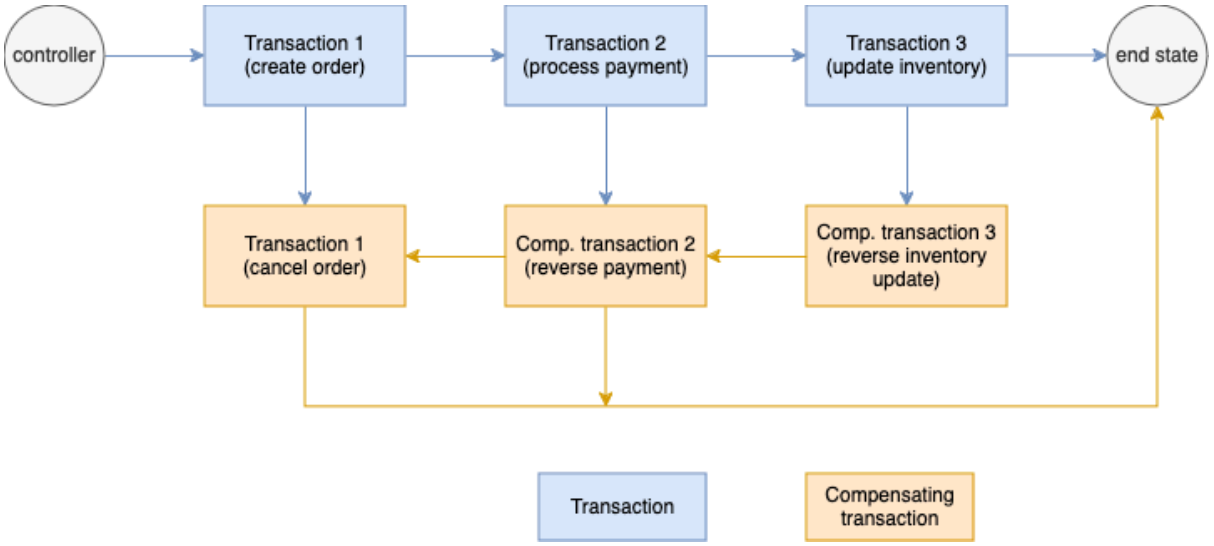
2.2 SAGA pattern

One of the solutions to guarantee data consistency across the microservices are distributed transactions. However many modern database technologies such as NoSQL databases and message brokers don't support distributed transactions. Another downfall of the distributed transactions is the fact that they use synchronous communication. Therefore mechanisms that base on a principle of loose coupling and asynchronous communication should be used.

One of those is the Saga pattern. It works on a principle of applying the sequence of local transactions, where each one updates data within a single service based on ACID principle. A coordinator triggers the first step of the saga, meaning the first local transaction executes. After one local transaction is completed, it triggers the execution of the next local transaction. The benefit of asynchronous messaging is that it ensures that all the steps of saga are executed, even if one of the participants is temporarily unavailable.

A in ACID stands for the atomicy and this means that all the steps of a transaction are executed or the transaction is rolled back in case of failures. To ensure the atomicity of the transactions in Saga, we need methods to rollback the changes made by local transactions. Those methods are called compensating transactions. It is important for the compensating transactions to be idempotent and retryable. Idempotency means that each operation can be executed multiple times and the

result will always remain the same. And the mechanism of automatically retrying the transactions in case of failures ensures that there is no need for any manual interventions.



The diagram above shows the overview of the Saga flow. As you can see, each transaction is triggered once the previous is finished. And in case of failures, the appropriate compensating transaction is executed. The compensating transaction has to call other compensating transactions in reverse order to reset changes to the initial state.

We need a mechanism that will take care of the coordination of the Saga flow. A Saga coordinator is responsible for this case. It contains logic that selects the first participants of the Saga and invokes the execution of its local transaction. Then it has to invoke the next participants until all the steps are completed. If any of the steps fails, a coordinator has to execute the compensating transactions in reverse order.

There are two ways of the implementation for the Saga coordinator logic:

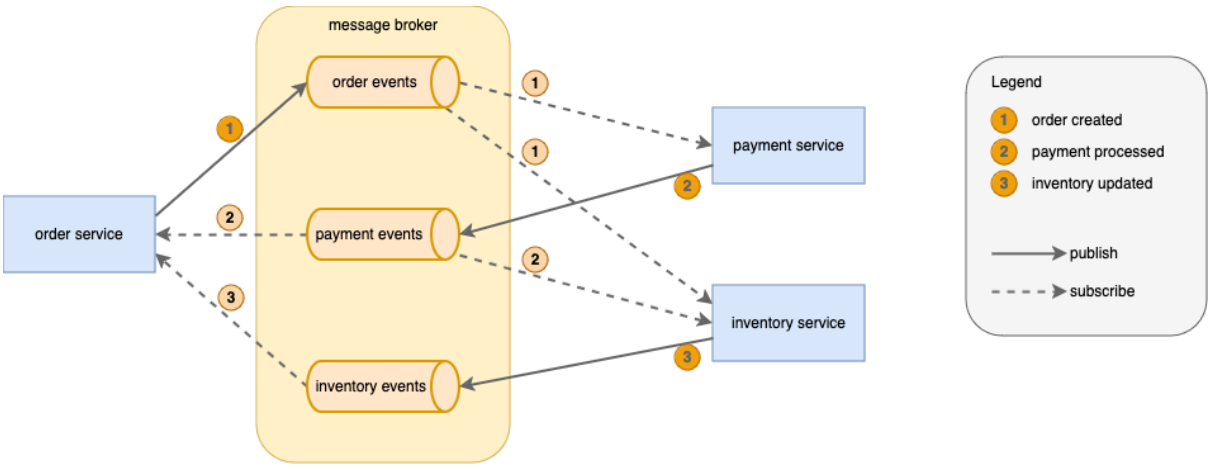
- choreography - coordination logic and sequencing is distributed among the participants, meaning the participant invokes the next step after its local transaction is completed. This king of communication is primarily based on the events

- orchestration - coordination logic is centralized in a dedicated component, which sends commands to other participants telling them which operation to execute

Choreography

In this way of implementation there is no central component to tell Saga participants what to do. Therefore the participants must take care that the next steps in the Saga flow are executed. Usually the participants are subscribed to each other's events and the published events are an indicator which tells other participants what to do. Each participant executes its local transaction and publishes an event. This event triggers the next participant, which repeats the same thing. The flow is considered completed when all the participants have successfully finished their local transactions. In case of failures, participants have to publish the appropriate events that trigger compensation transactions to revert their local changes.

We can demonstrate the choreography based saga flow on an e-commerce example. When a user creates an order, an event order_created is sent to the message broker. From an order channel, a payment service consumes this event and processes the payment, and at the same time, an inventory service reserves quantity for ordered items. When a payment is processed, a new event payment_processed is published to the payment channel of the message broker. Inventory service can update quantity based on this event, and the order service can update the status. When the inventory is updated, an inventory_updated event is published and the order service can finish the order when consuming this event. The diagram below demonstrates the choreography based saga flow.

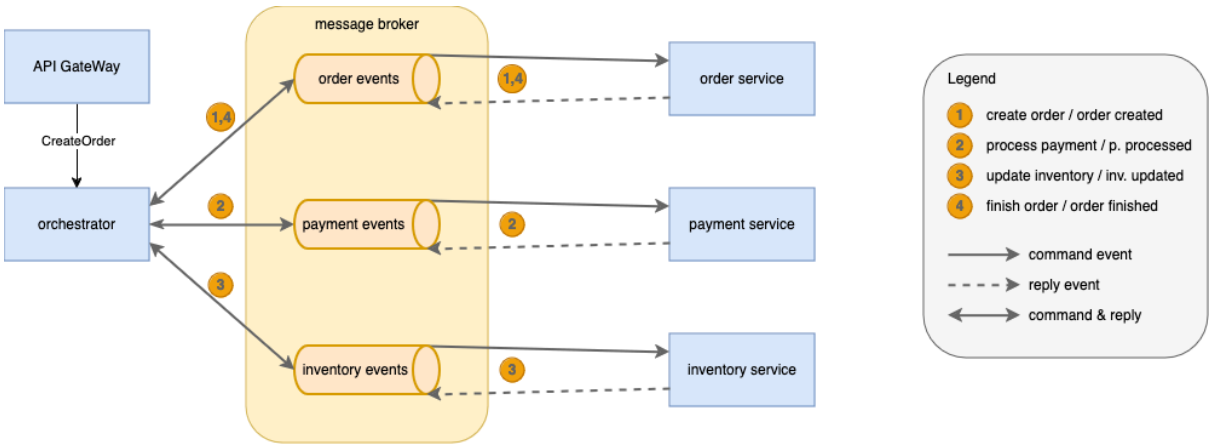


This approach is simple but there are some considerations, we have to be aware. It leads to a circular dependency and a possibility of tight coupling, because the participants have to subscribe to all the events. Therefore it's suitable for the simple Sagas.

Orchestration

In the case of the orchestration, a special component for coordination is used. This component is called an orchestrator and is responsible for managing the whole Saga flow. The communication is based on the asynchronous command-reply pattern. An orchestrator sends the command to the participant and awaits for the response. Based on the response, it invokes the next steps of the Saga, or a compensation transaction in case of failure. Each command triggers a local transaction on the participant's database. When sending commands, transactional messaging should be used, to ensure reliable communication between the orchestrator and the participants.

If we compare the orchestration based diagram with the choreography based, we can see some differences. There is one additional component, the orchestrator. And the whole flow goes through this component. In this example, the event based communication is used, and the orchestrator is a standalone component. That is not necessarily the case, it can be implemented in one of the services as well. We will keep the flow simple and we will skip the command to reserve quantity as it is in the choreography based flow.



When an order is created, a request is routed to the orchestrator, which creates a command message create_order. Order service consumes this message and creates a new order in its database, and replies to the order channel in the message broker with the event order_created. This event is consumed by the orchestrator, which sends the next command, process_payment. The payment service processes the payment and replies to the same channel, from which the orchestrator consumes the event payment_processed. At this point an additional command reserve quantity could be sent as in the choreography based flow, but for simplicity, we skipped this step. In the next step, the orchestrator sends a command to the inventory channel, and the inventory service updates the quantity. When the orchestrator receives the reply, the last command will be sent, that is the command to order channel to finish the order.

Orchestration based sagas bring some benefits such as simpler dependencies which prevents a circular dependency and loose coupling where each participant is independent of others. On the other hand, we have to be careful not to implement too much business logic within the orchestrator. This leads to the design where a smart orchestrator calls dumb services and it should be avoided. The orchestrator should be responsible only for sequencing and should not contain any business logic. Orchestration is the preferred way of implementing sagas.

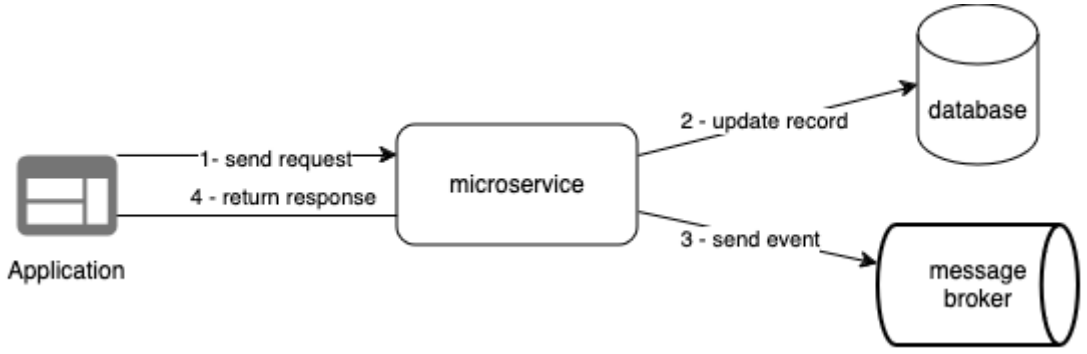
2.3 Outbox pattern

The outbox pattern operates on a principle that acts as a temporary storage for events. The idea is not to send events directly to other systems after some operation was performed on the API, but to store them in a local temporary table, called the outbox table. Later on, those events are read from the outbox table and sent to the message broker. The key point here is that all the business logic and events are stored in the scope of a single transaction. This ensures that either all the steps inside a transaction will be executed, or none of them.

And that is the main purpose of this pattern; to ensure atomicity and consistency. When the event is sent after the service performs the operation on a database, two things can happen:

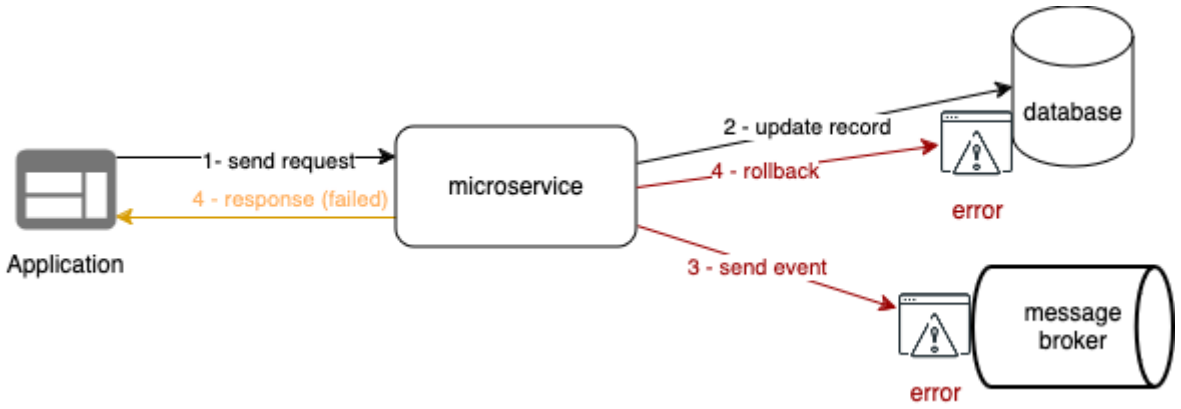
- if the database operation is successful but the event is not sent, the downstream service will not be aware of the change;
- if the database operation fails, but the event is sent, data might be corrupted.

In a typical scenario, an event flow looks like on the diagram below. After the service receives the request, some database operation is performed and the event is sent to notify other systems. After both operations succeed, the successful response will be returned.



But what happens if sending the event fails? We can rollback the operation on a database and return a failed response. But what if rollback also fails? The service will return a failed response but let’s see what happened to the data. The first operation on a database will modify the data, the event will not be delivered and the

second operation on a database will not revert the changes made in the first operation. This will lead to an inconsistent state as the data will be corrupted.

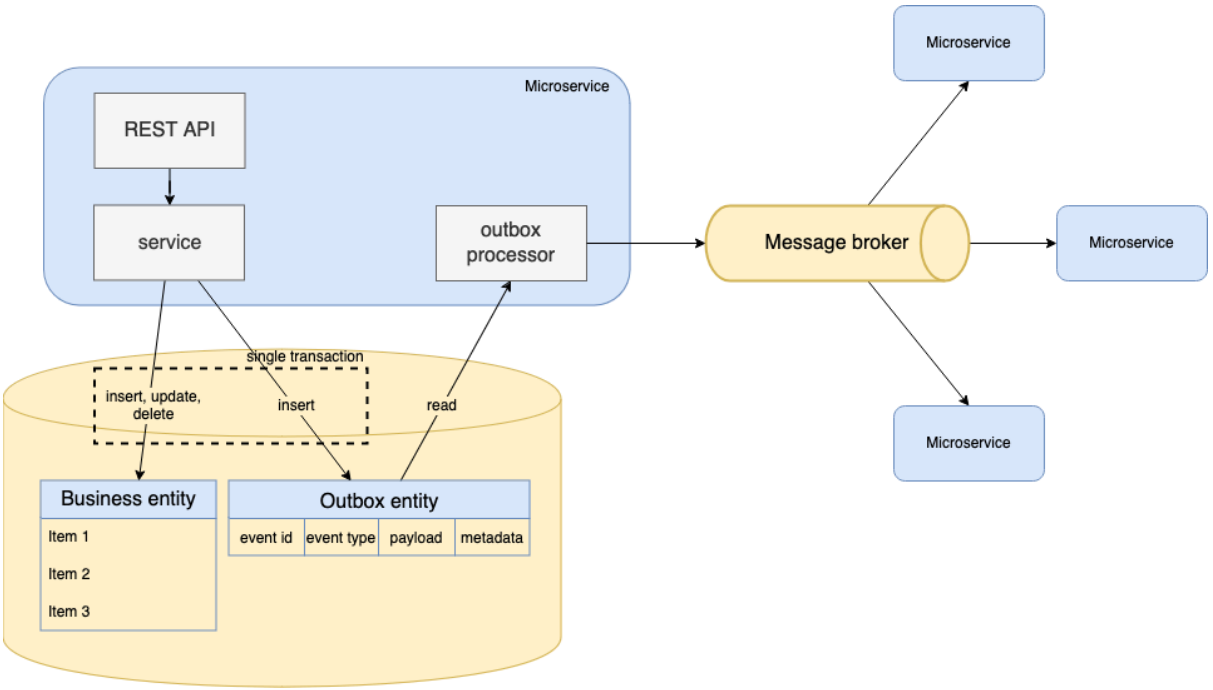


Therefore those two operations should run atomically, so either all steps in the same transaction are executed or none of them is. The outbox pattern stores the events in the outbox table before sending them to the message broker. So what happens if updating the database fails or inserting the event into the outbox table fails? Since both operations are running in a single transaction, the whole operation will fail in this case. And if both succeed, the event will later be retrieved from the outbox table and sent to the message broker. If that fails, it can be retried. This example also describes the principle of a guaranteed message delivery.

From a technical point of view, the outbox pattern has a special table, called the outbox table. It serves as a temporary storage for events. Service that implements the outbox pattern uses this table along standard create, update and delete operations. When those kinds of operations are executed, the event is stored into the outbox table in the same database transaction. Events that are stored inside the outbox table are sent to the message broker, which then distributes the events to other services. For reading the events from the outbox table and sending them to the message broker, the outbox pattern uses a special component, an outbox processor. This can be a simple job that periodically checks for unpublished events and sends them to the message broker.

Following diagram demonstrates the outbox pattern's behavior. There is a database that is used for storing business related data, and an outbox table. As we see in the diagram, when a service performs an operation like inserting, updating or deleting

business related data, this event is also inserted into the outbox table at the same time. Then there is an outbox processor which reads the events from the outbox table and publishes them to the message broker.



If we look at the architectural structure of an outbox pattern, it should contain following components:

- application service logic that supports operations:
    - business logic that performs some operations, updates the application state and generates events
    - outbox persistence mechanism that stores the events into outbox table
    - outbox entity, that stores fields like event id, event type, event payload and some metadata such as timestamps
- outbox processor, which is a background process that periodically scans the outbox table for new events and publishes them to the message broker. This component also needs to take care of the error handling and retries for failed event publications
- message broker that delivers events to the subscribed services

- downstream systems that are consuming and processing events from the message broker

As mentioned, the outbox processor contains logic that scans the outbox table for new events and then publishes them. Once the event is published, we need to make sure to appropriately process this record in the outbox table to avoid being resend again. We can either delete this record, or mark it as sent with a boolean field for example. In case of later, we have to pay attention that the outbox pattern is not meant for the auditory purposes but for the reliability. Therefore send events should be deleted eventually to avoid the outbox table to grow indefinitely.

Another thing that is in the scope of the outbox processor's task, is to implement error handling and retry mechanisms in case of failed events. Retrial mechanism is important, because it will try to resend the events that failed previously. This guarantees at least once delivery. On the downside, this mechanism can lead to duplicate messages being sent. This can happen if the error occurs after the event is being sent and we fail to update the event. Next time the outbox processor scans for the unpublished events, this event will still be marked as unsent and will be delivered to the message broker again. One solution to mitigate this issue is to implement idempotent consumers. They should expect the same messages can be sent and they should handle this properly.

The Outbox pattern works well with the CDC (Change Data Capture) pattern. Some databases support mechanisms to capture changes on a database level and publish events based on the changes. This solution eliminates the need for an additional table or the scheduler that needs to periodically scan for the unpublished events.

Use cases

Outbox pattern is useful when we want to guarantee atomicity of operations across multiple services. It's useful when we are dealing with services where data consistency is highly important, such as in the financial industry. The outbox pattern ensures the events are published only in case the operation in the database succeeded and guarantees the events are published in order they were generated. One of the drawbacks of the outbox pattern is that it can cause multiple messages

being sent. We can bypass this issue by implementing an idempotent message handler.

2.4 Reliable message processing

In modern distributed systems, reliable and consistent event processing is crucial for maintaining data integrity and ensuring system stability. One of the common challenges is handling duplicate messages. Some patterns like an Outbox pattern and Sagas ensure reliable message delivery, but the message broker can deliver the same message multiple times. Many messaging systems have built-in mechanisms to eliminate duplicate messages, but due to network failures or failing to send acknowledgement and retry mechanisms, a consumer service can still receive duplicate messages.

Before moving forward, let’s take a look at the delivery guarantees from message brokers. In general, there are three types of guarantees:

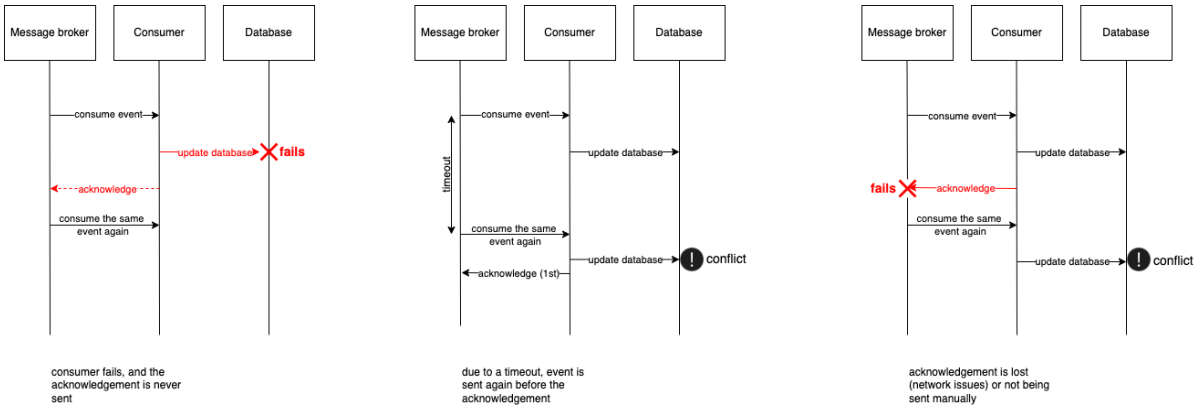
- at most once - consumer receives a message once or possibly not at all
- at least once - consumer receives a message once or possibly multiple times
- exactly once - producer sends exactly once and the message is delivered to consumers exactly once, excluding failures and retries, and is considered the most complex delivery guarantee

The most common guarantee among message brokers is at least once delivery. This type does not consider the message processed until the consumer acknowledges the delivery. There are many cases where acknowledgement is not sent to the message broker:

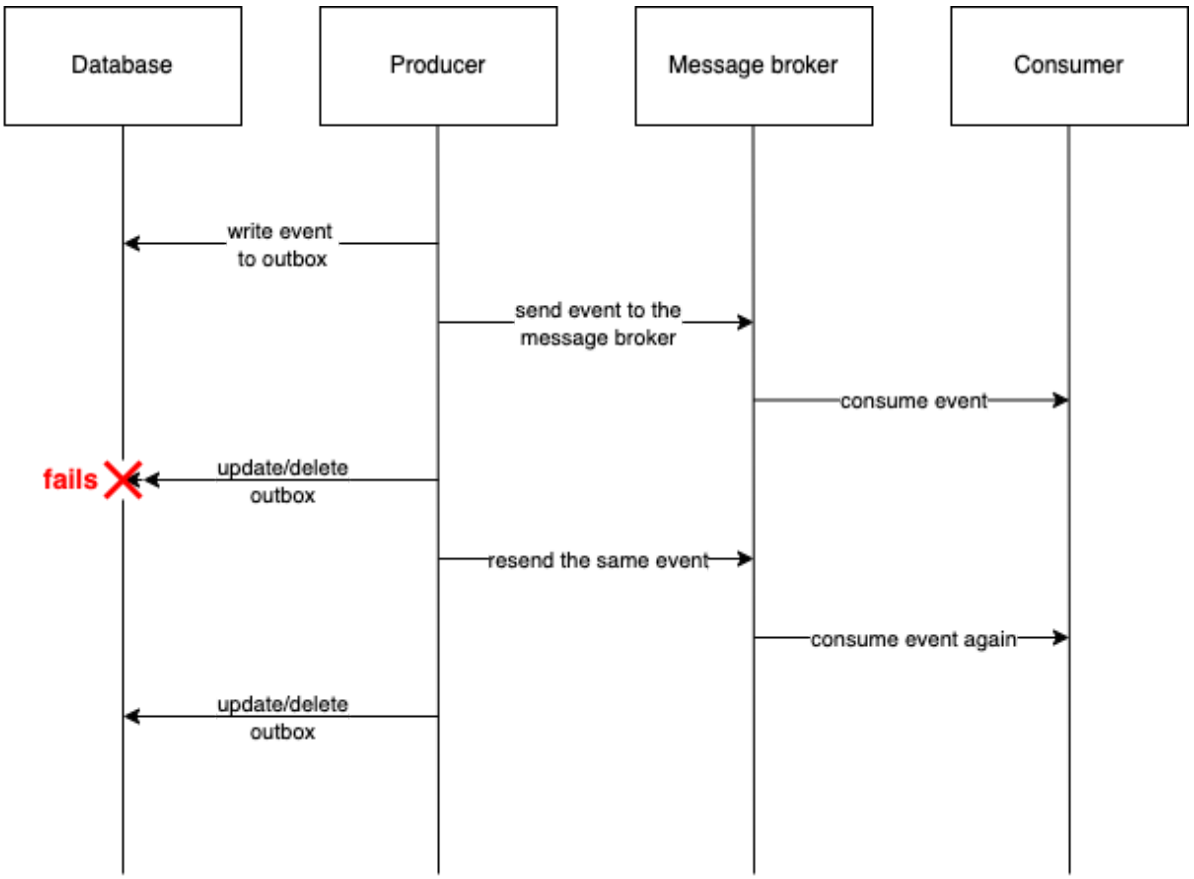
- consumer fails and the acknowledgement of delivery is never sent to the broker. The broker will resend the message
- generally there are timeouts that an acknowledgement needs to be sent. If the acknowledgement is not sent in that time frame, the broker will consider the unacknowledgement and send the message again

- acknowledge is not sent due to a network failure
- some libraries or implementations requires manual acknowledgement in the code, and if for any reason it never occurs, the message will be send again

The scenarios are demonstrated on the diagram below.



Another reason for receiving duplicate messages is because the producer is sending the same message more than once. It can be due to a bug, but there is also a possibility that the Outbox pattern will send the same message more than once due to its design. It ensures at least once delivery and if it sends a message to the broker but fails to update its database, it will send the same message again. The following diagram demonstrates this case.



Processing duplicate messages can lead to data corruption, redundant operations and violations of business rules, and therefore it is important for consumers to implement mechanisms that prevent those issues. A service that can safely consume the same message multiple times is called an idempotent consumer. Idempotency is a term that in mathematics describes a function that produces the same result if it is applied to itself (Hohpe and Woolf, 2023). In the concept of interservice communication, this concept translates into a message that has the same effect whether it is received once or multiple times.

A service is considered an idempotent consumer if it can safely process the same message multiple times without causing unintended effects (Richardson, 2019). One way to implement an idempotent consumer is through the Eventuate Tram framework, which ensures idempotency by recording unique identifiers of processed messages. It maintains a PROCESSED_MESSAGES table as part of the local ACID transactions used by the business logic, enabling reliable detection and discarding of duplicates (Richardson, 2019). Alternatively, idempotency can be achieved through explicit de-duplication, which involves maintaining a buffer of unique message

identifiers to identify and ignore duplicates, or by designing message semantics to inherently support idempotency (Hohpe and Woolf, 2023). The latter approach avoids repeated side effects by structuring message content to specify a desired state rather than an action, such as setting a balance to 110 instead of adding 10 to the balance.

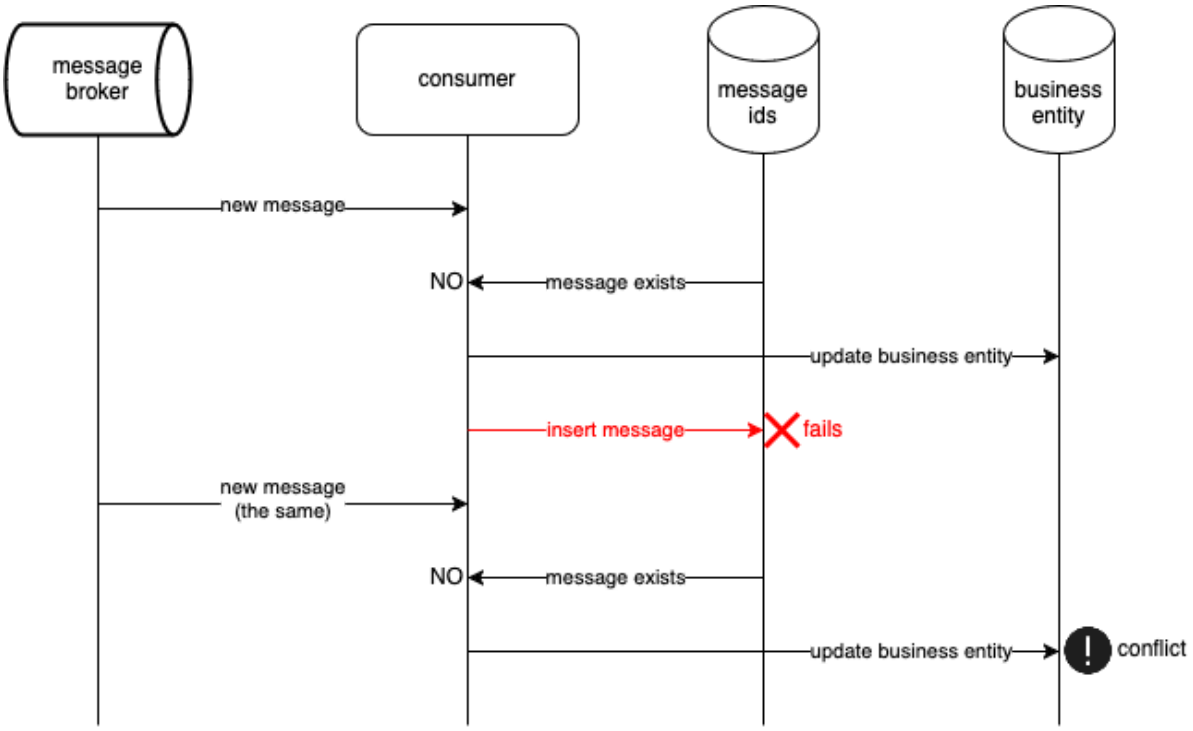
Based on those theoretical findings, we need to structure the event messages so that they cannot violate business rules when being processed multiple times, and/or keep a record of processed events. As we explained, application logic is idempotent if calling it multiple times with the same input values, it has no effect. An example of an idempotent message is cancelling an order; if we cancel an already cancelled order, there is no difference. We have to be a bit more careful with creating an order. If we provide order id, we can use this field as a unique constraint and the same order will not be created twice. Those messages are considered safe, as processing them multiple times won't cause harm. Unfortunately, all the messages are often not idempotent, and in this case we must implement a mechanism for tracking messages on the consumers (Richardson, 2019).

Tracking consumed messages is one of exactly once delivery strategies, and it requires messages to have some kind of a unique identifier, let’s call it message id, to check if the message was consumed. If message id is stored, it can be compared with new messages and any duplicates can be discarded. With relational databases, we can use a separate table to track messages. When a message id is saved, the business logic should be updated within the same transaction. If a message id exists, the whole transaction, including updating the business entity, will fail due to a unique database constraint on the message id. A pseudocode demonstrating this approach is following:

```
begin transaction;
update business entity from a message payload;
save message id from the message;
end transaction;
```

The limitation of this approach is that it only works on relational databases. Many NoSQL databases don’t support transactions or have limited support for transactions

when updating two tables, and we cannot rely on this approach. A possible solution is to track message ids in any available application table, and changing the business logic to perform a manual check if the message was processed (Richardson, 2019).

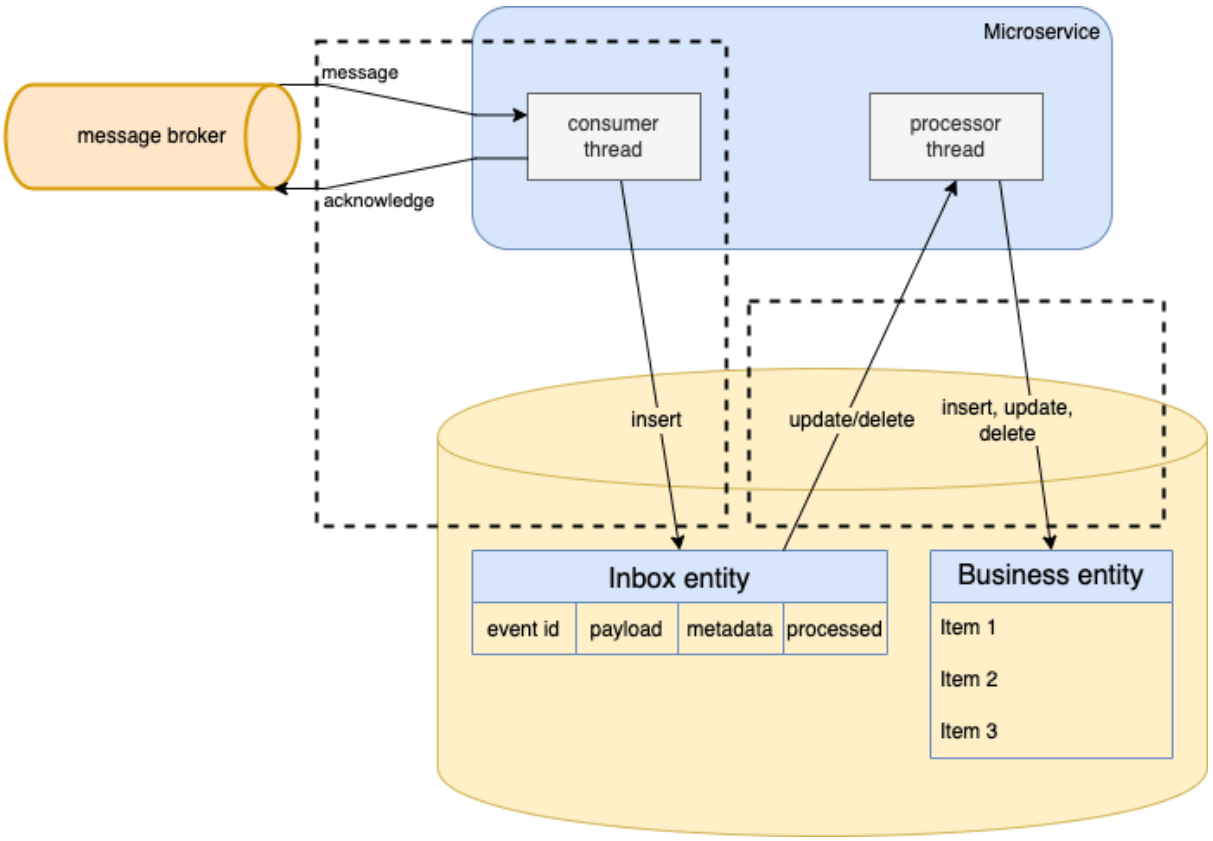


A diagram above demonstrates what can go wrong if we are not using the same transaction when tracking message ids and updating business logic. As shown, we updated the business entity and failed to insert a message id. When the same message is consumed again, the check will consider the message not to be processed and the service will update the business entity again, which can cause conflicts or inconsistent state. We can separate tracking and processing logic into two separate processes as we will demonstrate in the next architectural principle, the inbox pattern.

Inbox pattern

To keep a record of processed events, we need some kind of storage, it can be a simple in-memory buffer or a database table. When the consumer receives an event, it needs to check in the storage if the event was processed, which is usually done by comparing the event's unique identifier. If the event was not processed, it should be stored and processed, otherwise, if the event was processed, it should be discarded.

Inbox pattern enhances this process of message storage on the consumer side. It has a dedicated table to store incoming messages. When a new message is consumed, it is not processed immediately, but stored into this inbox table, and the acknowledgement is sent to the message broker. Then another process, a scheduler for example, scans the table for new, unprocessed messages. New messages are processed in a separate thread than consumed, and once processed, the record in the inbox table is either updated or removed. It looks similar to the outbox patterns, the difference is that it works in the opposite direction.



The inbox table should at least contain fields for the unique identifier, the payload and a flag to mark events as processed. Unique identifier, or message id, makes sure that the incoming messages are de-duplicated. If the application crashes after saving a duplicate message id, the acknowledgement will not be returned and the message will get resend. To prevent this issue, we should perform a check if a message id already exists in the database and discard duplicates at this point.

Inbox pattern can be helpful in case the order of messages is important. Some message brokers guarantee the order of messages, but not all of them. If the

messages are stored in a database and have an increasing identifier, the order can be restored by the process which processes the messages. Keeping the order of the messages in the inbox table can be useful when some messages are missing for some reason. In such cases, we can launch a redelivery strategy by manually asking the producer to send all the messages from the missing offset again.

This pattern brings benefits like ensuring exactly once delivery by providing a de-duplication mechanism, it helps maintain the order of the messages and can be used for auditing purposes from the log of received messages.

On the downside, the index pattern adds additional complexity, more load to the database and increases latency. Some performance optimizations can mitigate those obstacles, like introducing parallelism with multiple schedulers concurrently scanning the inbox table and processing messages or starting processing asynchronously immediately after inserting it into the inbox table. In case of concurrency we have to make sure that the locks are used to prevent other threads from processing the same rows. Lastly, to prevent the inbox table from growing too much, processed messages can be deleted regularly.

3. References

Books

1. Christudas, Binildas. *Practical Microservices Architectural Patterns: Event-Based Java Microservices with Spring Boot and Spring Cloud*. Apress, 2019.
2. Fernando, Chanaka. *Solution Architecture Patterns for Enterprise: A Guide to Building Enterprise Software Systems*. Apress, 2023.
3. Goniwada, Shivakumar R. *Cloud Native Architecture and Design: A Handbook for Modern Day Architecture and Design with Enterprise-Grade Examples*. Apress, 2022
4. Hohpe, Gregor, and Woolf, Bobby. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.
5. Richardson, Chris. *Microservices Patterns*. Manning Publications, 2019.
6. Siriwardena, Prabath, and Kasun Indrasiri. *Microservices for the Enterprise: Designing, Developing, and Deploying*. Apress, 2018.
7. Tudose, Cătălin. *Java Persistence with Spring Data and Hibernate*. Manning Publications, 2023.

Online articles

8. Atlasik, Krzysztof. “Microservices 101: Transactional Outbox and Inbox.” Software Mill, 2022, https://softwaremill.com/microservices-101
9. Chandrakant, Kumar. “Introduction to Transactions.” Baeldung, 2024, https://www.baeldung.com/cs/transactions-intro
10. Comartin, Derek. “Handling Duplicate Messages (Idempotent Consumers).” CodeOpinion, 2020, https://codeopinion.com/handling-duplicate-messages-idempotent-consumers
11. Kritiotis, Panayiotis. “Outbox Pattern - Why, How and Implementation Challenges.” 2021, https://pkritiotis.io/outbox-pattern-implementation-challenges/
12. Ludwikowski, Andrzej. “Message delivery and deduplication strategies” Software Mill, 2022, https://softwaremill.com/message-delivery-and-deduplication-strategies
13. Murphy, Eric. “Distributed Data for Microservices - Event Sourcing vs. Change Data Capture” 2020, https://debezium.io/blog/2020/02/10/event-sourcing-vs-cdc/

14. Musib, Somnath. “Saga Pattern in Microservices.” 2023, https://www.baeldung.com/cs/saga-pattern-microservices
15. Nanayakkara, Crishantha. “Microservices Patterns: The Saga Pattern.” 2023, https://medium.com/cloud-native-daily/microservices-patterns-part-04-saga-pattern-a7f85d8d4aa3
16. Ozkaya, Mehmet. “Outbox Pattern for Microservices Architectures.” 2021, https://medium.com/design-microservices-architecture-with-patterns/outbox-pattern-for-microservices-architectures-1b8648dfaa27
17. Posta, Christian. “The Hardest Part About Microservices: Your Data.” 2016, https://blog.christianposta.com/microservices/the-hardest-part-about-microservices-data/
18. “A Beginner’s Guide to Event Sourcing.” 2021, https://www.kurrent.io/event-sourcing
19. “Consistency in Microservices: Transactional Outbox Pattern.” Stackademic, 2023, https://blog.stackademic.com/consistency-in-microservices-transactional-outbox-pattern-bcd9d3b08676