Java Technologies

Data Management in Microservices

Data Management

- *What's the database architecture in a microservices app?*
- Services must be loosely coupled.
- Some business transactions must enforce invariants that span multiple services or must update data owned by multiple services.
- Some business transactions need to query data that is owned by multiple services.
- Some queries must join data that is owned by multiple services.
- Databases must sometimes be replicated and sharded in order to scale.
- Different services may have different data storage requirements.

MicroProfile and JPA

- The DBMS server will be hosted usually in its own container, running at a dedicated port.
- **At the project level**
  - Configure the persistence unit (persistence.xml)
  - Define the entity classes
  - Perform the CRUD operations
- **At the configuration (micro-server) level**
  - Specify that JPA is a required feature
  - Specifiy the driver files, for example: postgresql.jar
  - Configure the JTA DataSource (that you will be using in persistence.xml)

Example (OpenLiberty+Postgtres)

https://openliberty.io/docs/21.0.0.1/reference/config/dataSource.html

- server.xml

```
<featureManager>
    <feature>microProfile-4.0</feature>
    <feature>jpa-2.2</feature>
</featureManager>
...
<library id="postgresql-library">
    <file name="D:\java\lib\postgres\postgresql.jar"/>
</library>

<jdbcDriver id="postgresql-driver"
   javax.sql.ConnectionPoolDataSource =
            "org.postgresql.ds.PGConnectionPoolDataSource"
   libraryRef="postgresql-library"/>

<dataSource id="demo-datasource"
            jndiName="jdbc/demo"
            jdbcDriverRef="postgresql-driver"
            type="javax.sql.ConnectionPoolDataSource"
            transactional="true">
    <properties serverName="localhost" portNumber="5432"
                databaseName="demo"
                user="postgres" password="password"/>
</dataSource>
```

Example (cont.)

https://openliberty.io/docs/21.0.0.9/reference/feature/jpa-2.2.html

- pom.xml

```
<dependency>
     <groupId>javax.persistence</groupId>
     <artifactId>javax.persistence-api</artifactId>
     <version>2.2</version>
     <scope>provided</scope>
</dependency>
```

EclipseLink is included as the default JPA provider implementation.

- persistence.xml

```
<persistence-unit name="DemoPU" transaction-type="JTA">
    <jta-data-source>jdbc/demo</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
</persistence-unit>
```

Problems and Solutions

https://microservices.io/patterns/data

- **How many databases**?
  - Database per service
  - Shared Database
- How to execute **distributed transactions**?
  - Two Phase Commit
  - Sagas
- How to **query** data owned by multiple services?
  - API Composition
  - Command Query Responsibility Segregation (CQRS)
- How to **update** data owned by multiple services?
  - Domain Events
  - Event Sourcing

Brewer’s CAP Theorem

- CAP: A system can have **only two** of the following: **consistency, availability, partition tolerance**.
- **Consistency:** the same response from different nodes. Every read receives the most recent write or an error.
- **Availability:** every request will receive a response (non-error), without the guarantee that it contains the most recent write.
- **Partition tolerance:** the system continues to operate despite the fact that some nodes cannot communicate, messages being dropped or delayed.
  - No network can guarantee the lack of failure (network partitioning)

Availability and Consistency

- **Sacrificing consistency**
  - Consider two instances of a CRUD service (nodes) that cannont communicate with the system.
  - The system is still available, allowing for reads and writes, but not consistent, because each node contains changes that the other node doesn’t know about.
- **Sacrificing availability**
  - Suppose that each node must see the same data;
  - Since the nodes cannot communicate, guaranteeing consistency is not possible, so the only option is to deny requests.

Consistency Model

- **Distributed Shared Memory**
  - Two nodes A, B of the same type. Client 1 writes/updates X to A. Client 2 reads X from B.
- **Strong** (**Strict**) Consistency
- **Weak** Consistency
- **Eventual** Consistency
  - the system will reach consistency on X at one point in time, that is, all requests will return the last value of X . This is a form of weak consistency because there is a period of time when the system is in an inconsistent state.

Transactions

- A unit of processing that must guarantee a set of properties
- **ACID - Consistency** over Availability
  - Atomicity, Consistency, Isolation, Durability
- **BASE - Availability** over Consistency
  - *Basically Available*: The database appears to work most of the time; there will be a response to every request.
  - *Soft State*: Different node instances don’t have to be mutually consistent all the time. State can change over time, even if there are no reads or writes.
  - *Eventual Consistency*: The system keeps changing the data to make it consistent. Eventually, it becomes consistent.

Shared Database

- A single database shared by multiple services.
- Each service freely accesses data owned by other services using local ACID transactions.
- Advantages: simple, familiar
- Disadvantages
  - Development time coupling (schema)
  - Runtime interferences (locking)
  - Lack of control regarding data invariants
  - Lack of flexibility in DBMS choice

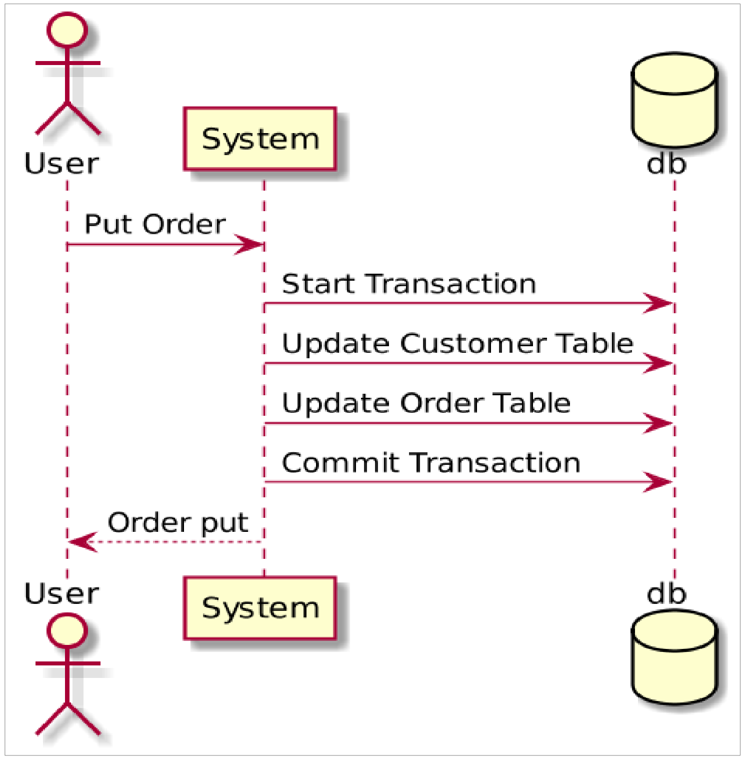
Database per Service

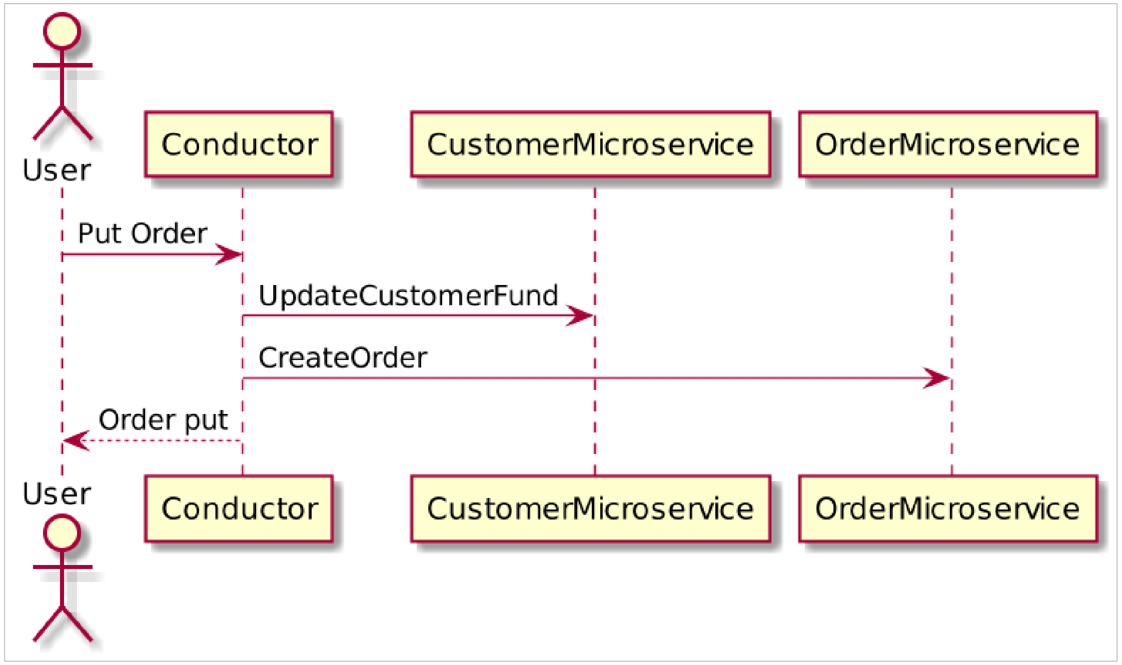
- **Variants**
  - Private-tables-per-service
  - Schema-per-service
  - Database-server-per-service
- **Advantages**
  - services are loosely coupled, independent DBMS
- **Disadvantages**
  - Distributed transactions
  - Complex queries are harder to implement, slower to execute

Distributed Transactions

- A distributed transaction includes one or more statements that, individually or as a group, <u>update data on two or more distinct nodes of a distributed database</u>.
- **Participant roles** (forming a session tree):
  - *Client* (references information in a belonging to a different node), *Server* (receives a request for information from another node),
  - *Local Coordinator* (a node forced to reference data on other nodes to complete its part of the transaction), *Global Coordinator* (the node that originates the distributed transaction)
  - *Commit Point Site* (the one node that commits or rolls back the transaction as instructed by the global coordinator)
- **Atomicity** → all participants must reach a uniform consensus on whether a successful commit is possible
- **Solutions**: Two Phase Commit (2PC), Sagas

Example

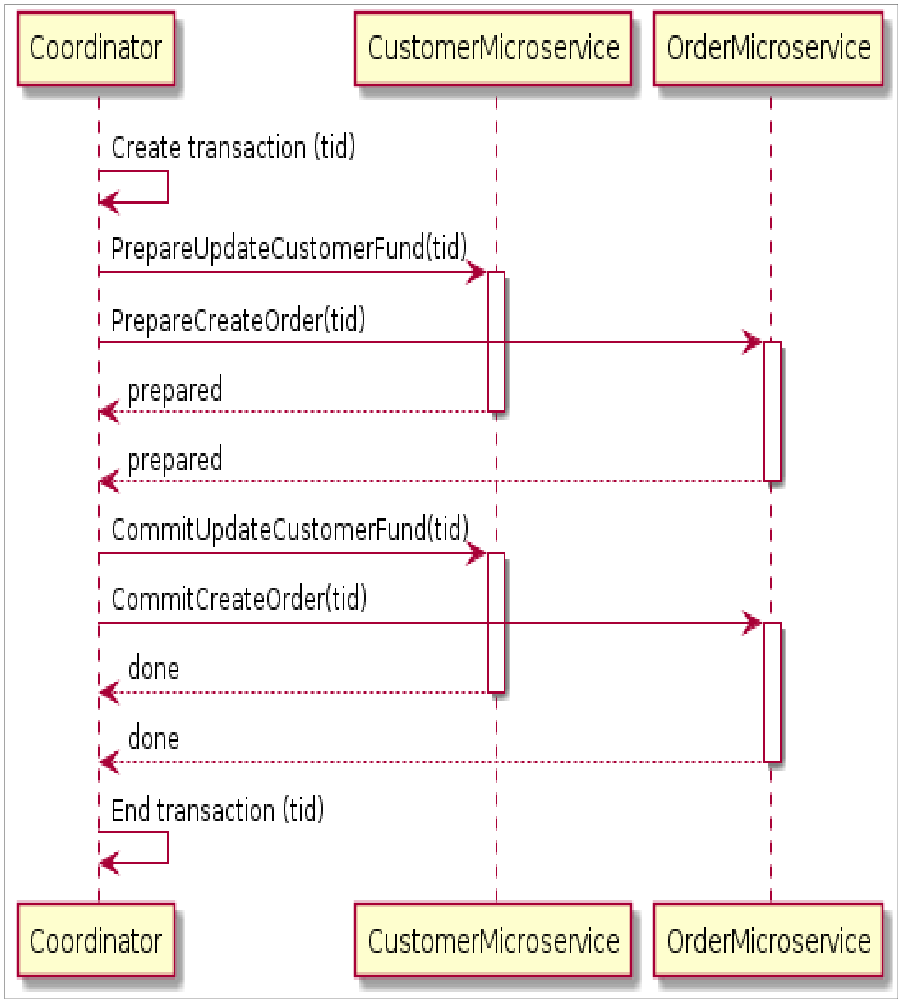

User
System
db
Put Order
Start Transaction
Update Customer Table
Update Order Table
Commit Transaction
Order put
User
System
db

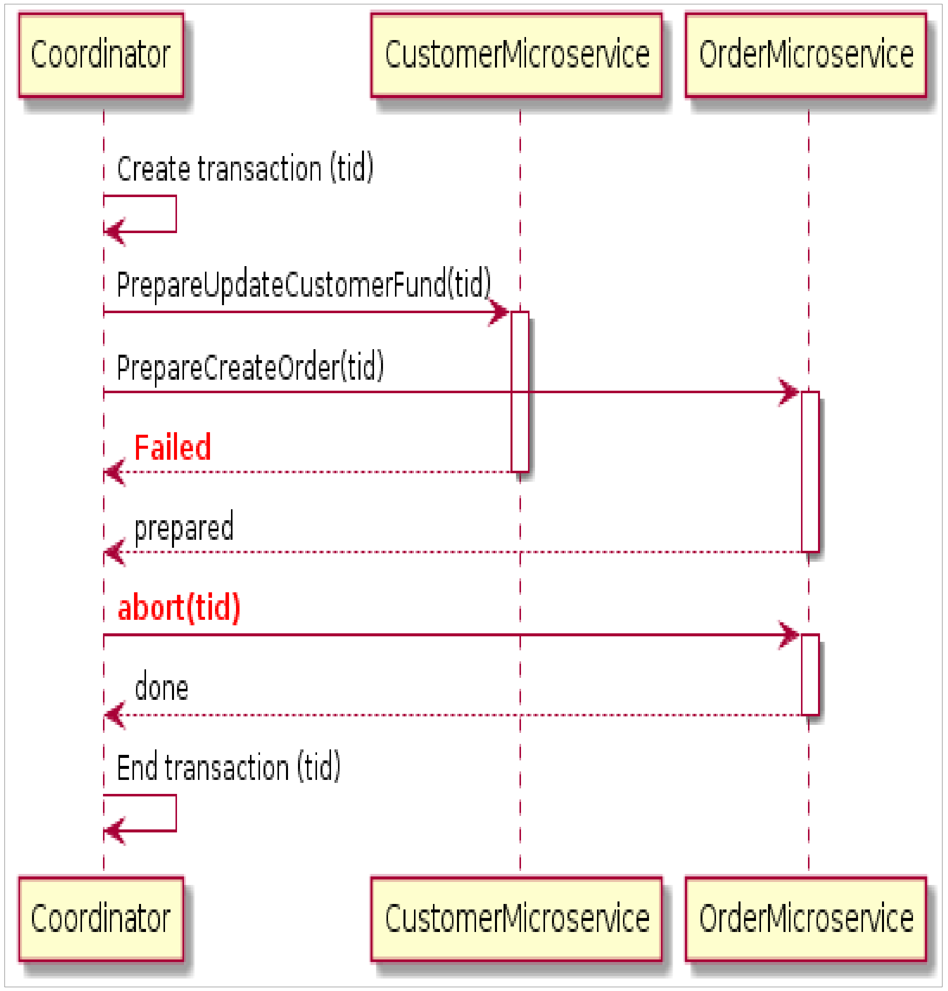

User
Conductor
CustomerMicroservice
OrderMicroservice
Put Order
UpdateCustomerFund
CreateOrder
Order put
User
Conductor
CustomerMicroservice
OrderMicroservice

Two Phase Commit (2PC)

- Widely used in database systems. For some situations, you can use 2PC for microservices
- **The voting (prepare) phase**: The global coordinator, asks participating nodes other than the commit point site to promise to commit or rollback the transaction, even if there is a failure. If any node cannot prepare, the transaction is rolled back.
- **The commit phase:** If all participants respond to the coordinator that they are prepared, then the coordinator asks the commit point site to commit. After it commits, the coordinator asks all other nodes to commit the transaction.
- Disadvantages: synchronous communication, vulnerability to failures, resource locking

2PC Example


Coordinator
CustomerMicroservice
OrderMicroservice
Create transaction (tid)
PrepareUpdateCustomerFund(tid)
PrepareCreateOrder(tid)
prepared
prepared
CommitUpdateCustomerFund(tid)
CommitCreateOrder(tid)
done
done
End transaction (tid)
Coordinator
CustomerMicroservice
OrderMicroservice

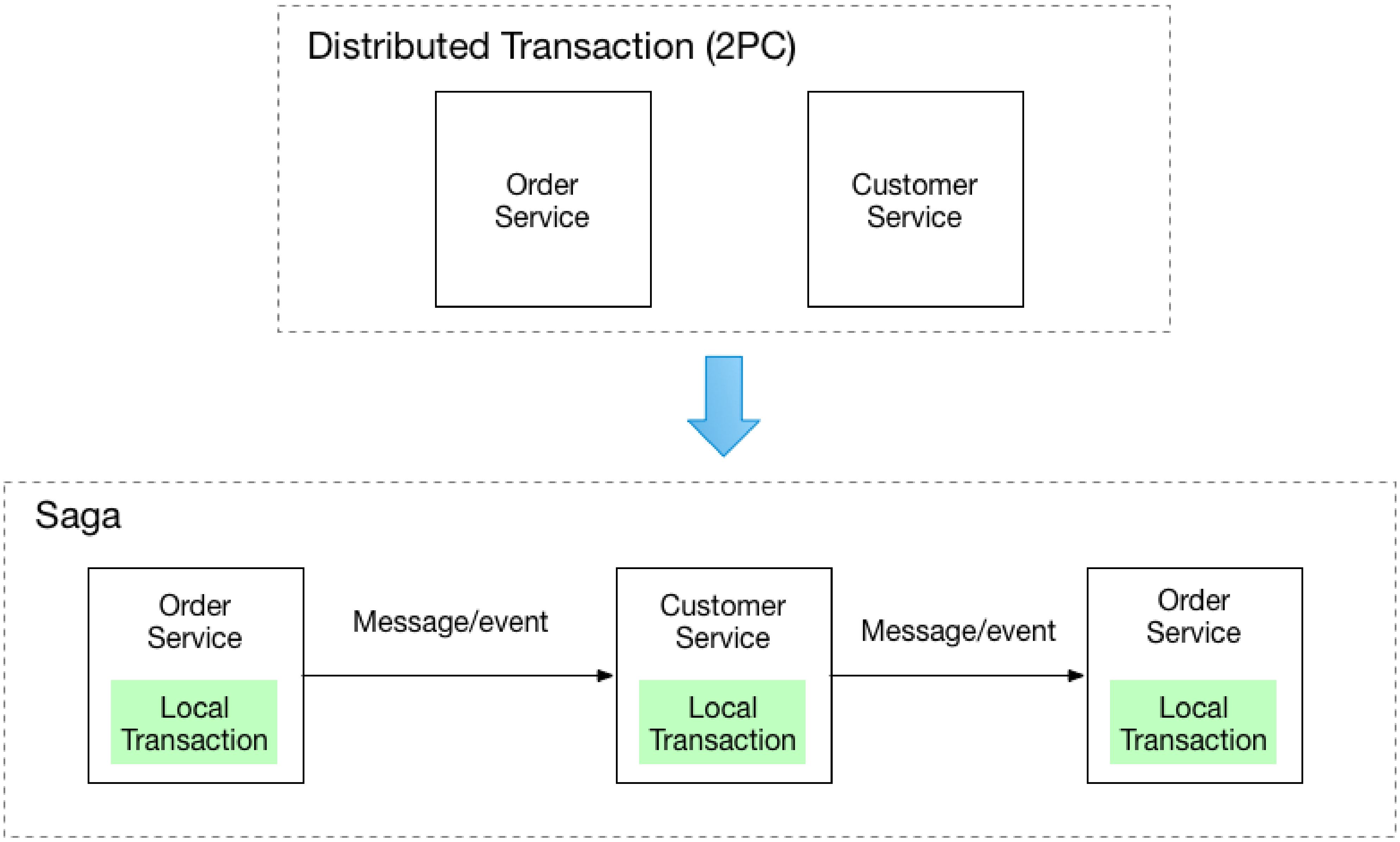

Coordinator
CustomerMicroservice
OrderMicroservice
Create transaction (tid)
PrepareUpdateCustomerFund(tid)
PrepareCreateOrder(tid)
Failed
prepared
abort(tid)
done
End transaction (tid)
Coordinator
CustomerMicroservice
OrderMicroservice

Saga Pattern

A long story of heroic achievement, especially in medieval (scandinavian) prose.

- Some business transactions span over multiple services → we need a mechanism to ensure data consistency for all the services involved.
- 2PC protocol affects availability by using synchronous communication – we don’t want it.
- **A saga is a sequence of local transactions**. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

From 2PC to Saga



Local Transactions

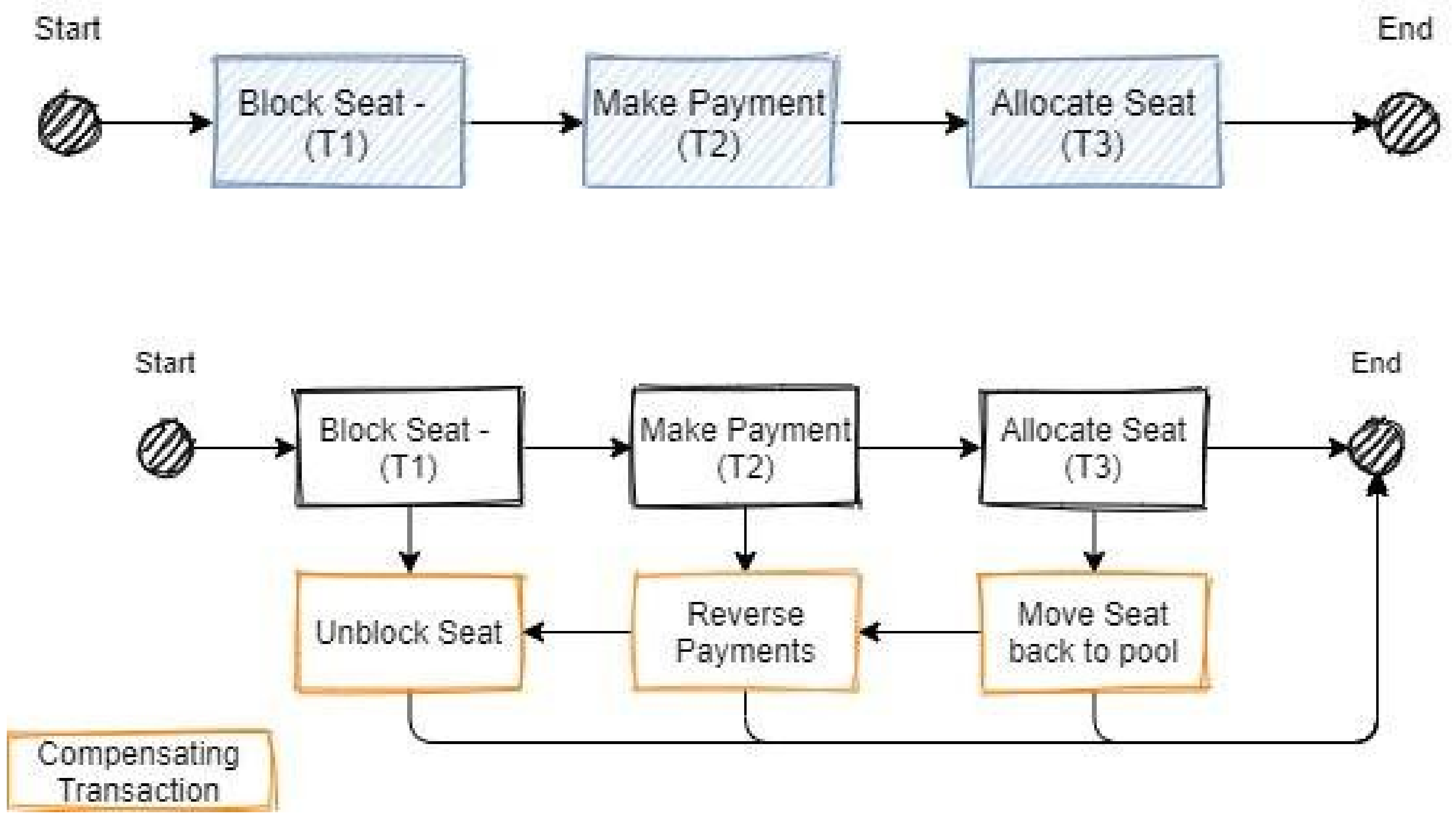
- **Compensatable** transactions: Transactions that can be rolled back using a compensating transaction.

  $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n$ (fails) $\rightarrow C_n \rightarrow \dots \rightarrow C_2 \rightarrow C_1$

- **Pivot** transactions: This transaction determines the success or failure of a saga. If this transaction succeeds so will the rest of the saga, if it fails, it will stop the rest of the saga from executing. It can be:
  - a transaction that is neither compensatable nor retriable
  - or it can be the last compensatable transaction or the first retriable one.
- **Retriable** transactions: These are transactions that come after the pivot transaction and thus, are guaranteed to succeed

Example

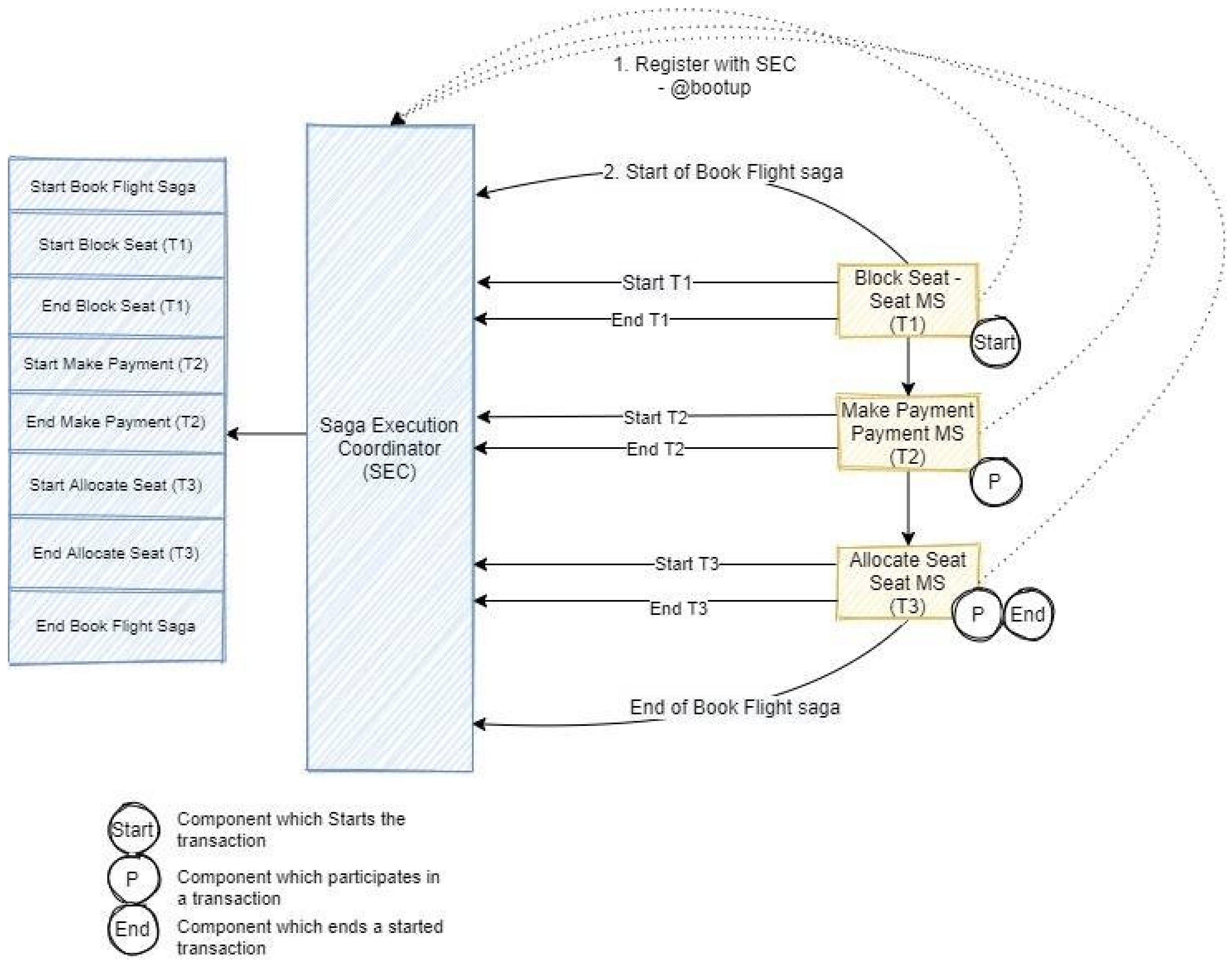
https://developer.ibm.com/articles/use-saga-to-solve-distributed-transaction-management-problems-in-a-microservices-architecture/



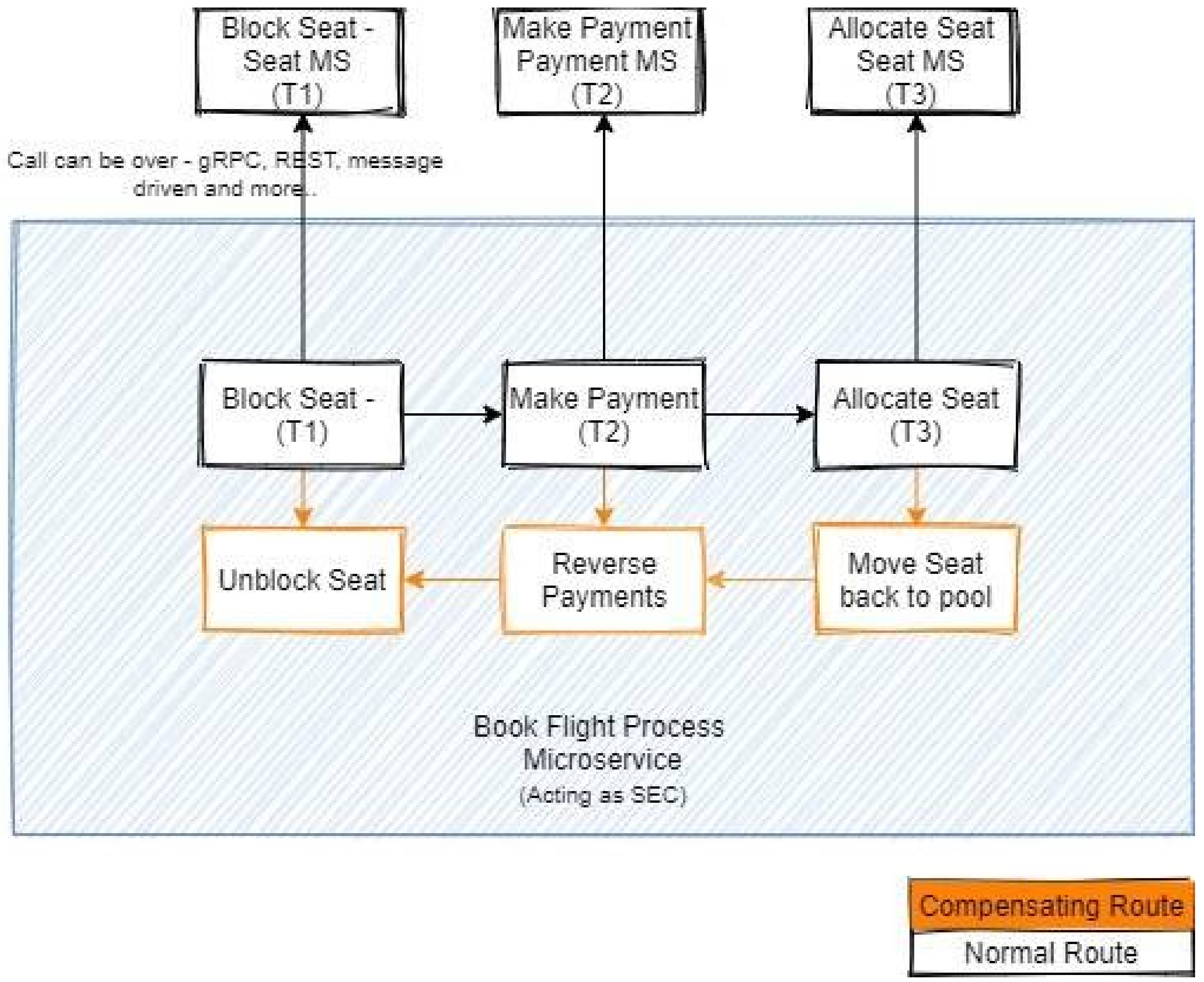
Saga Execution Coordinator

- It maintains a **Saga log** that contains the sequence of events of a particular flow. If a failure occurs within any of the components, the SEC queries the logs and helps identify which components are impacted and in which sequence the compensating transactions must be executed. Essentially, the SEC helps maintain an eventually consistent state of the overall process.
- If the SEC component itself fails, it can read the SEC logs when coming back up to identify which of the components are successfully rolled back, identify which ones were pending, and start calling them in reverse chronological order.

Choreography-based saga


1. Register with SEC - @bootup
2. Start of Book Flight saga
Start Book Flight Saga
Start Block Seat (T1)
End Block Seat (T1)
Start Make Payment (T2)
End Make Payment (T2)
Start Allocate Seat (T3)
End Allocate Seat (T3)
End Book Flight Saga
Saga Execution Coordinator (SEC)
Start T1
End T1
Block Seat - Seat MS (T1)
Start
Start T2
End T2
Make Payment Payment MS (T2)
P
Start T3
End T3
Allocate Seat Seat MS (T3)
P
End
End of Book Flight saga
Start
Component which Starts the transaction
P
Component which participates in a transaction
End
Component which ends a started transaction

Orchestration-based saga


Block Seat - Seat MS (T1)
Make Payment Payment MS (T2)
Allocate Seat Seat MS (T3)
Call can be over - gRPC, REST, message driven and more..
Block Seat - (T1)
Make Payment (T2)
Allocate Seat (T3)
Unblock Seat
Reverse Payments
Move Seat back to pool
Book Flight Process Microservice
(Acting as SEC)
Compensating Route
Normal Route

Lack of Isolation - Problems

- **Lost updates**: The saga overwrites changes made by another saga without reading them.
- **Dirty reads**: A saga reads the incomplete changes made by another saga.
- **Fuzzy/nonrepeatable reads**: Different steps of the same saga get different values for the same variable because another saga has made updates.

Lack of Isolation - Solutions

- **Semantic lock**: at the application level. A flag is placed on a record that could have its value changed. This either prevents other transactions from accessing it or acts as a warning that its value might change. Other transactions when encountering such flag could either fail or retry the operation. In the case of retry, a retry logic must be included.
- **Commutative updates**: Update operations that can be executed in any order. If the operations can be executed in any order then there is no possibility of different sagas to cause each other side effects through these operations.
- **Pessimistic view**: Reordering of the saga steps in order to minimize the business risk of a dirty read.
- **Reread value**: to verify it is not modified before overwriting it.
- **Version file**: Keep track of the updates in a file so that they can be reordered. It’s a way to make noncommutative operations act like commutative ones.
- **By value**: Dynamically select the concurrency mechanism based on the business risk of each request. The business risk is calculating by evaluating the properties of each request. This can help determine if it is more appropriate for a certain use case to use sagas or distributed transactions.

Saga Implementations

- **MicroProfile Long Running Actions (LRA)**
  - https://github.com/eclipse/microprofile-lra
  - https://openliberty.io/blog/2021/01/27/microprofile-long-running-actions-beta.html
- **Axon** http://www.axonframework.org/
- **Eventuate Tram** http://eventuate.io/
- **Seata** https://seata.io/

Event Sourcing

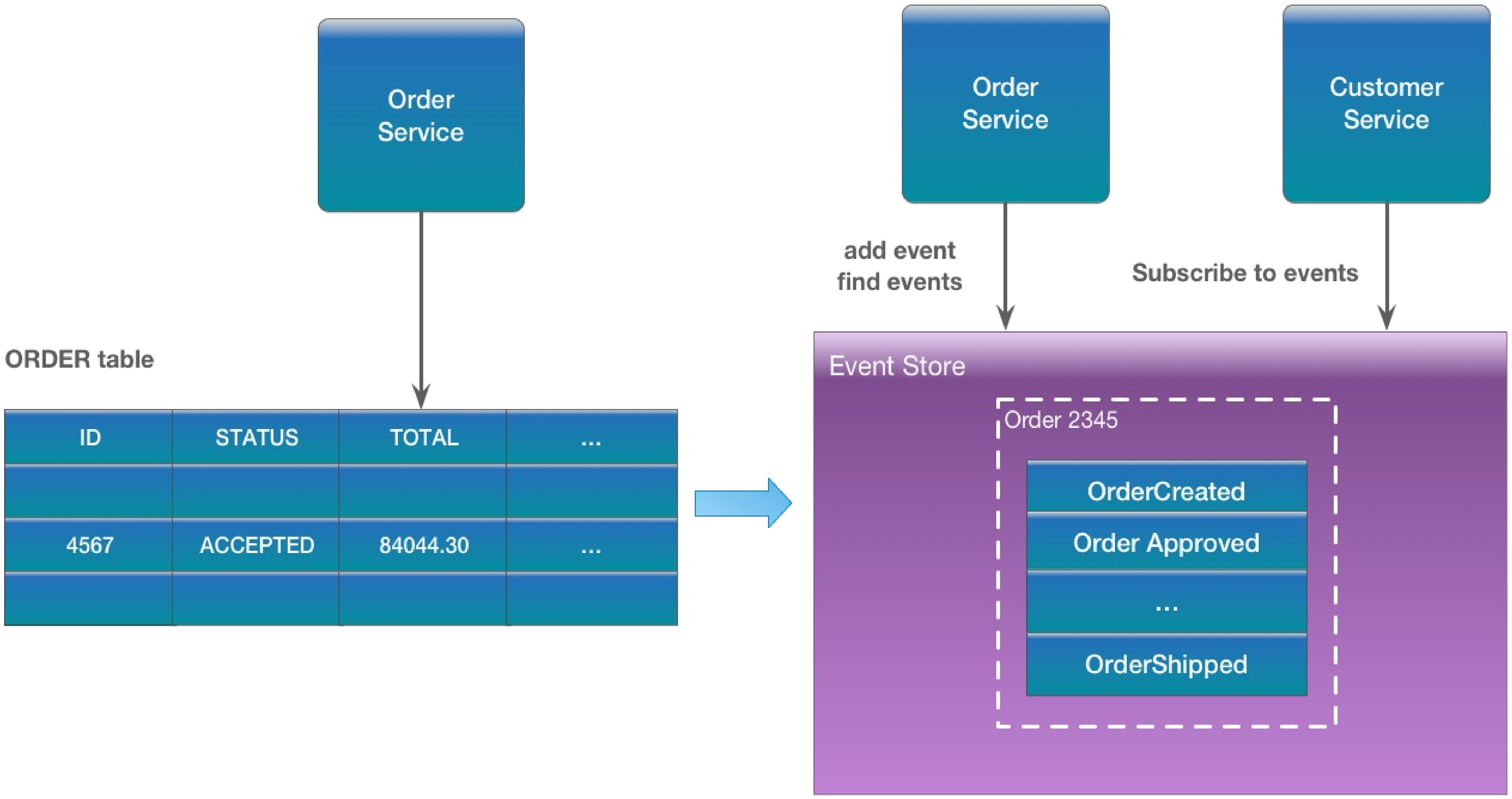
- In the context of a Saga, for each local transaction, there are <u>two actions that must be done atomically</u>:

  1) **update** the database and

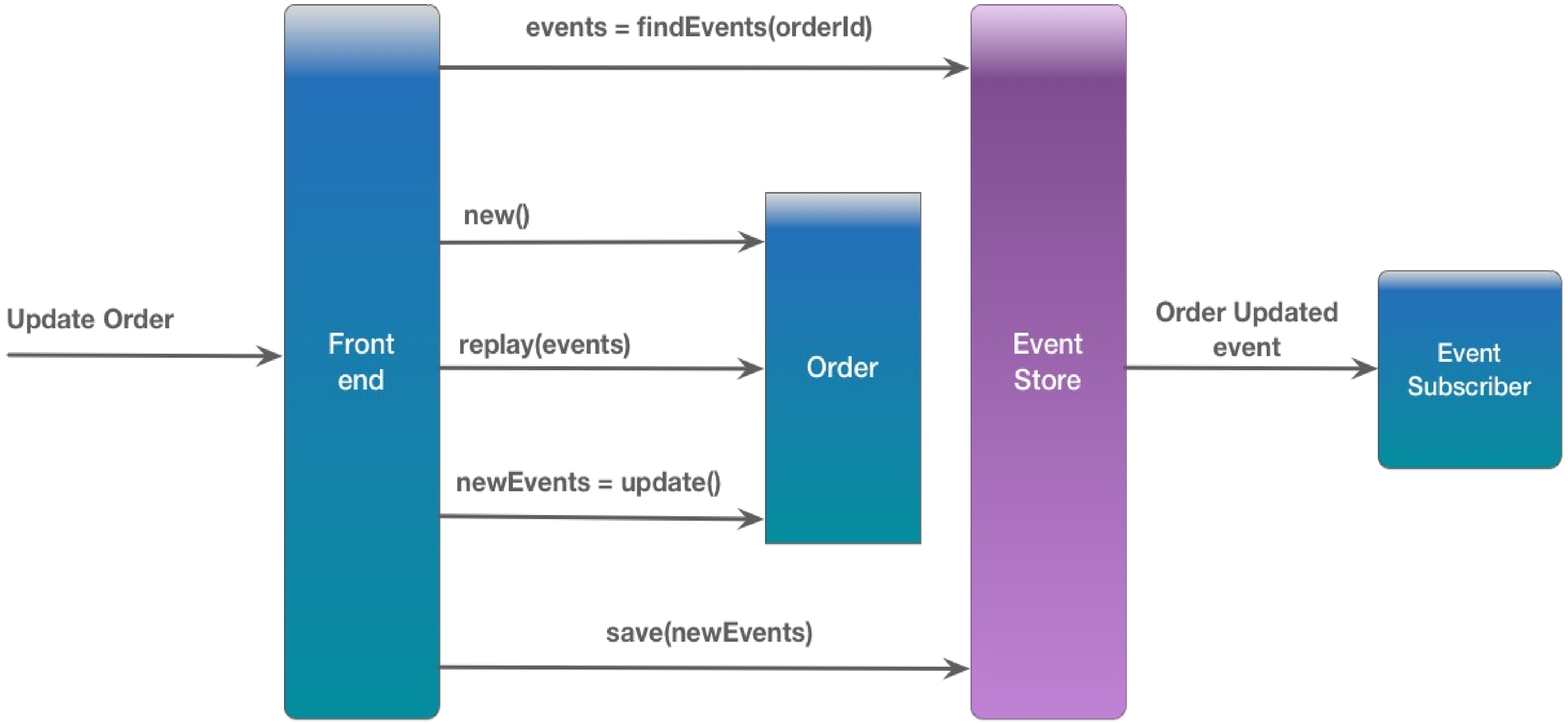
  2) **publish** events

- How to maintain the ACID requirements?
- **Event sourcing** refers to persisting the state of a business entity as a sequence of state-changing events or incremental updates.
  - Whenever the state of a business entity changes, a new event is appended to the list of events (**domain event**).
  - Since that is one operation it is inherently atomic.
  - A entity's current state is reconstructed by replaying its events.

Storing Events



Request Flow



Benefits of Event Sourcing

- Accurate **audit logging**
  - Each state change corresponds to one or more events
- Easy **temporal queries**
  - Because event sourcing maintains the complete history of each business object, implementing temporal queries and reconstructing the historical state
- In order to improve the performance of reconstructing the state, we may use **snapshots**, which are reconstructed states, from different points in time.
- Similar to commit logs (write-ahead logs) from DBMS

Event Sourcing Implementations

- **Eventsourced** (https://github.com/eligosource/eventsourced)
- **Jdon** (http://www.jdon.org/)
- **EventStore** (http://www.geteventstore.com.)
- **Axon** (http://www.axonframework.org/)
- **Akka toolkit** (https://akka.io/)
- **Eventuate** (https://eventuate.io/)
- **Lagom** (https://www.lagomframework.com/)

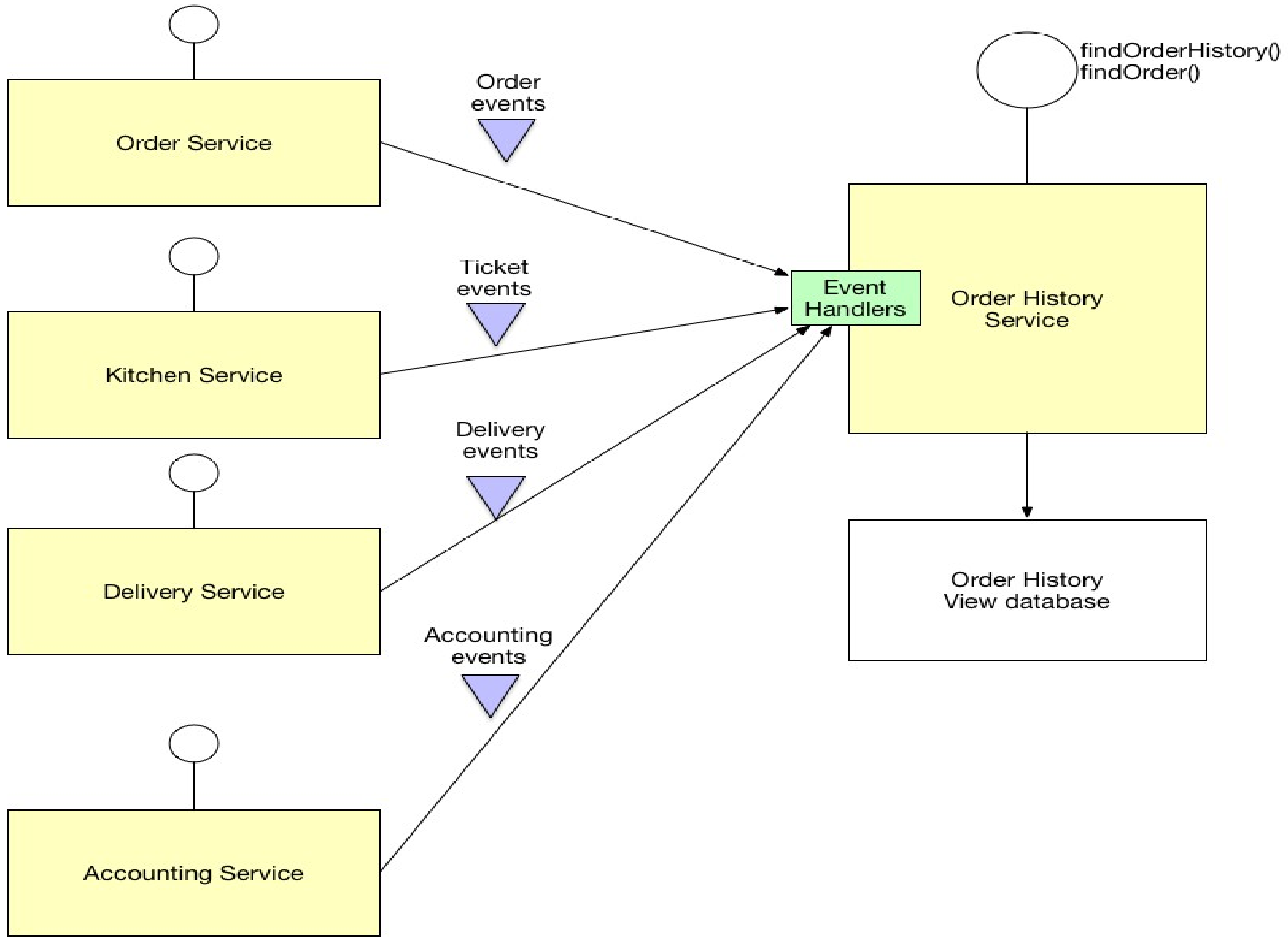
Querying

- **API Composer:** invoke the services that own the data and perform an in-memory join of the results.
- Simple
- May lead to inefficient, in-memory joins of large datasets.
- Other solution:
  - CQRS Pattern

Command Query Responsibility Segregation (CQRS)

- How to implement a query that retrieves data from multiple services in a microservice architecture?
- API Composition may be inefficient
- **CQRS: Splitting the update from the query**
  - Define a **view database**, which is a read-only replica that is designed to support that query.
  - The application **keeps the replica up to date** by subscribing to Domain events published by the service that own the data.
  - *Command* side (create, update, delete) vs. *Query* side

Quey Side Service



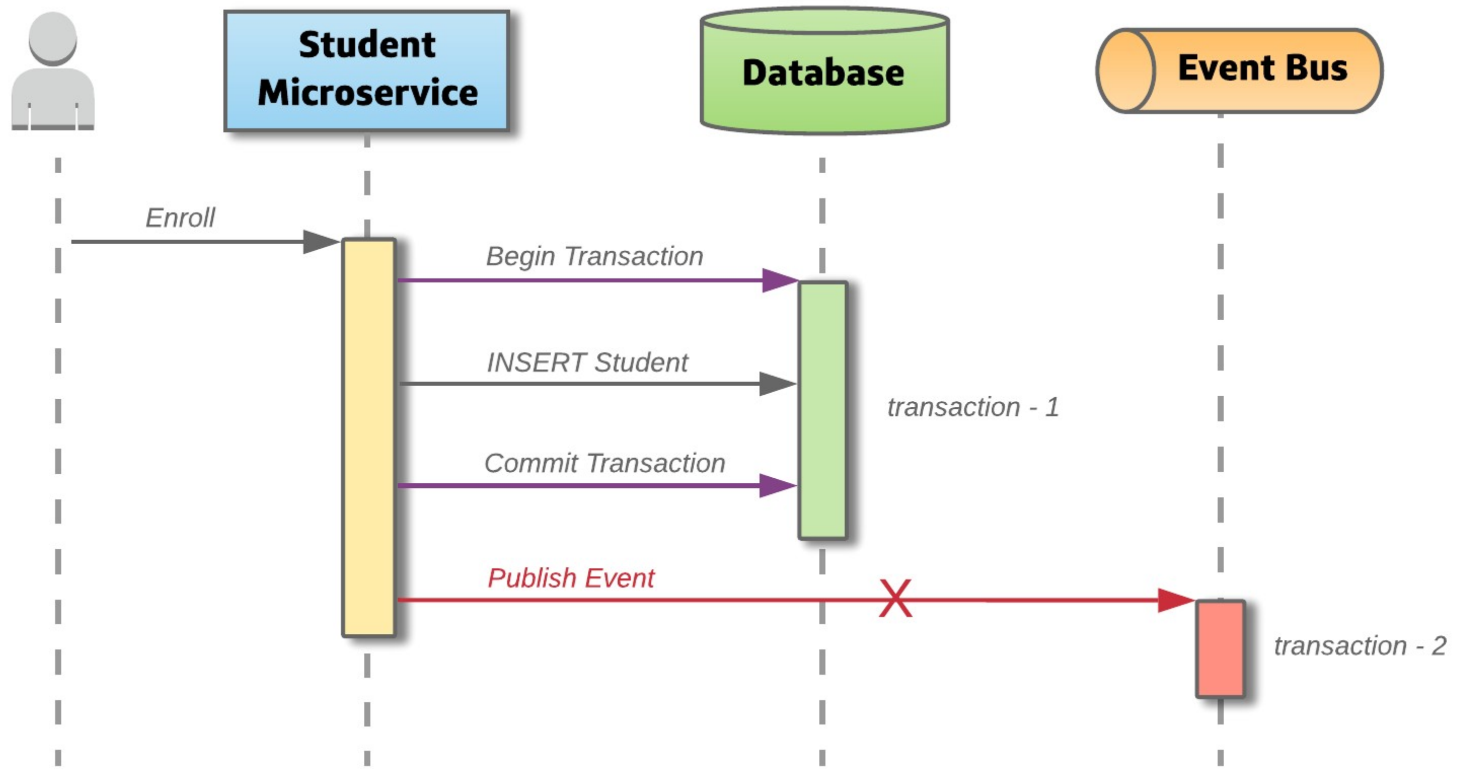
The CQRS pattern is often used along with the Event Sourcing pattern.

CQRS Implementations

- **Axon** (http://www.axonframework.org/)
- **Lagom** (https://www.lagomframework.com/)
- **Eventuate** (https://eventuate.io/)
- **Jdon** (https://en.jdon.com/)
- **NCQRS** (github.com/ncqrs)
- **Lokad** (github.com/Lokad/lokad-cqrs)

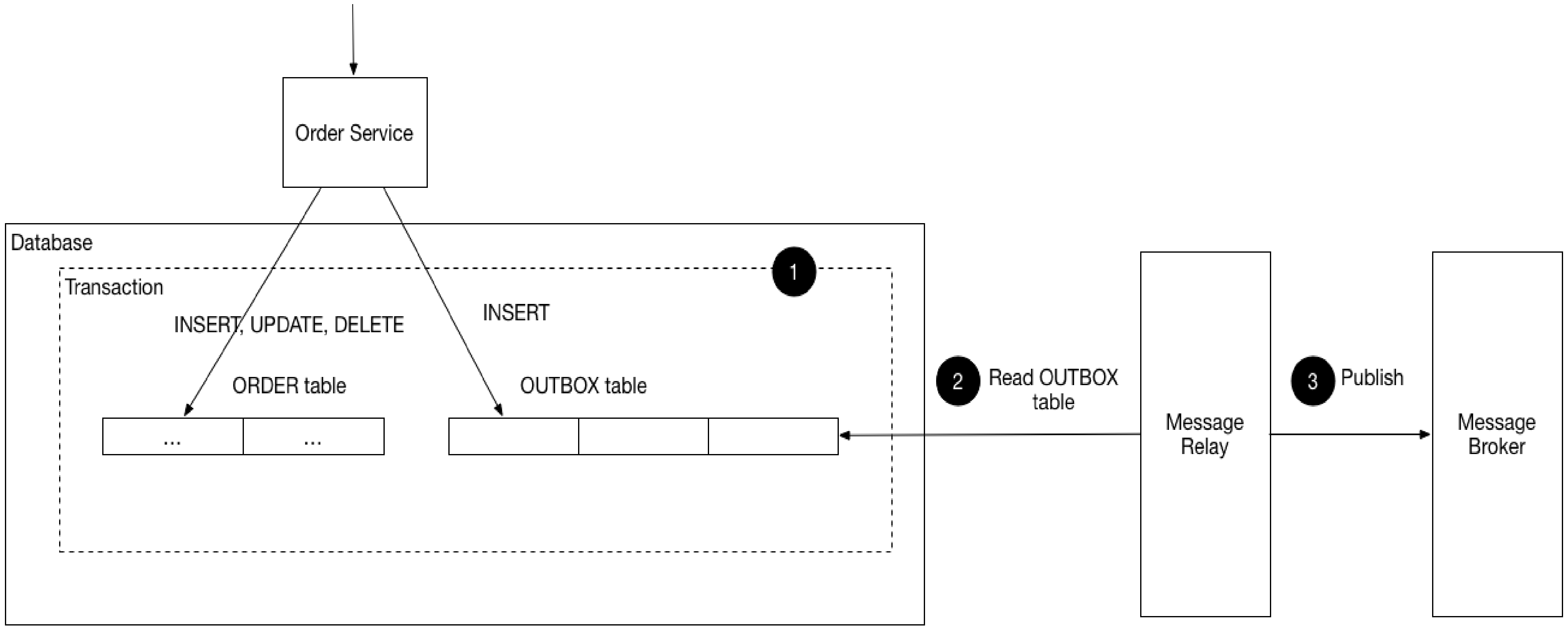
Reliable Event Publication

- How to reliably/atomically update the database and publish messages/events?

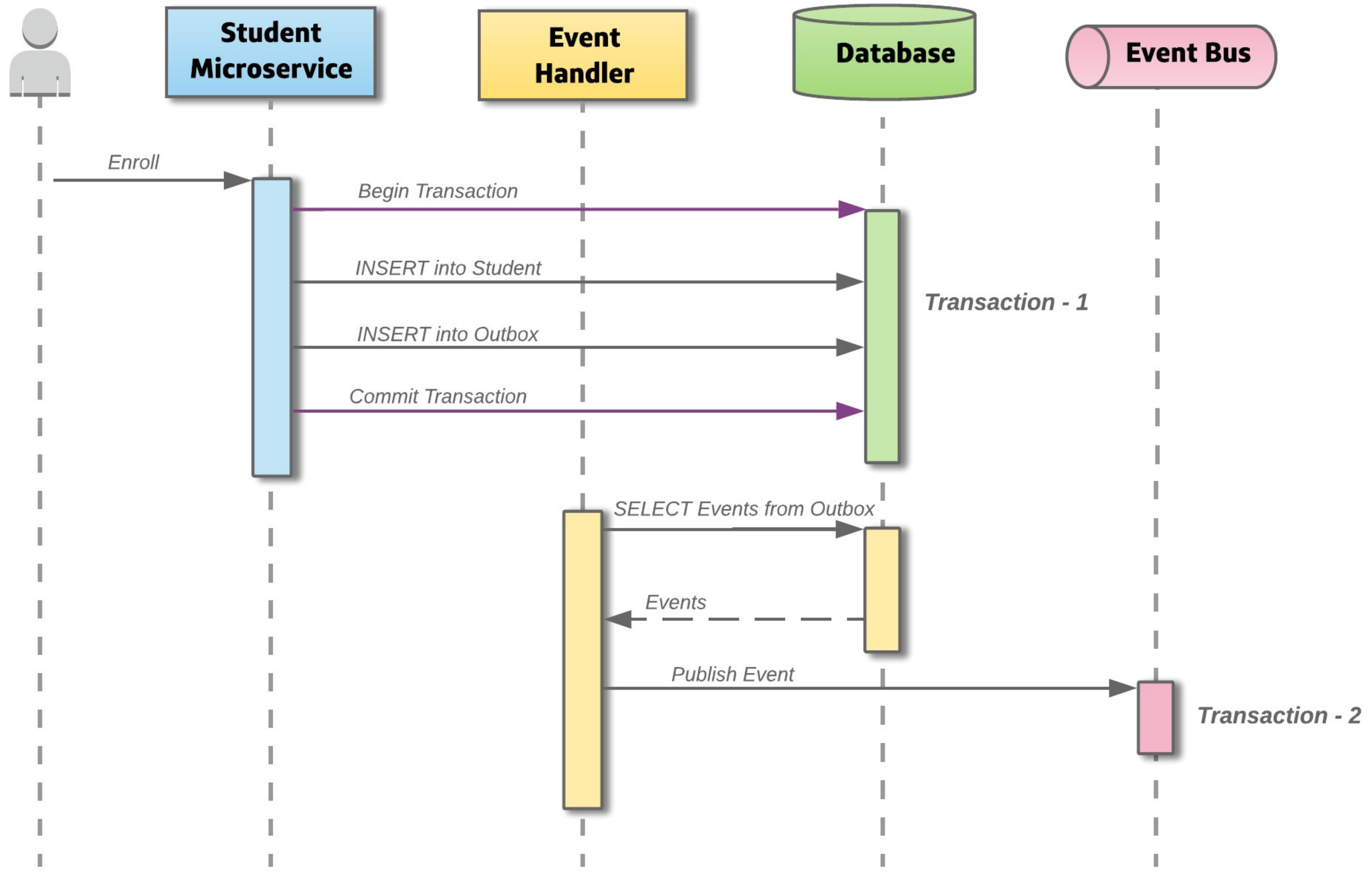


Transactional Outbox

- Use an additional **outbox table** as part of the local transaction.



Transactional Outbox



Implementing Outbox Pattern

- **Polling publisher**
  - Publish messages by polling the database's outbox table. Tricky to publish events in order.
- **Transaction log tailing**
  - Tail the database transaction log (using a miner) and publish each message/event inserted into the outbox to the message broker.
  - Requires database specific solutions: MySQL binlog, Postgres WAL (write-ahead logging), AWS DynamoDB table streams, etc.
  - Tricky to avoid duplicate publishing

Messaging

Next week...

- Microservices need to collaborate.
- Synchronous communication results in tight runtime coupling, both the client and service must be available for the duration of the request.
- **Asynchronous messaging** allows inter-service communication, by exchanging messages over messaging channels.
- Messaging suppliers:
  - Apache Kafka
  - RabbitMQ
  - OpenMQ, ActiveMQ (JMS implementations)
  - etc.