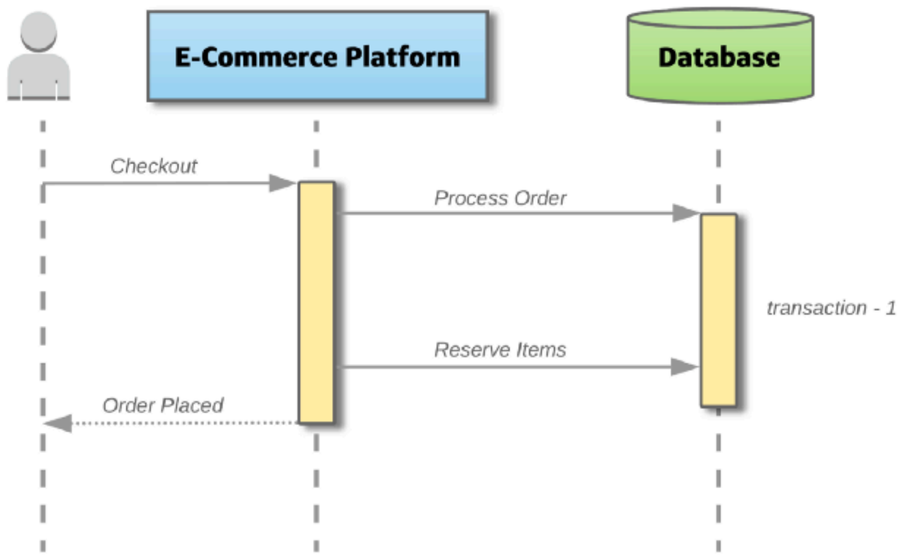
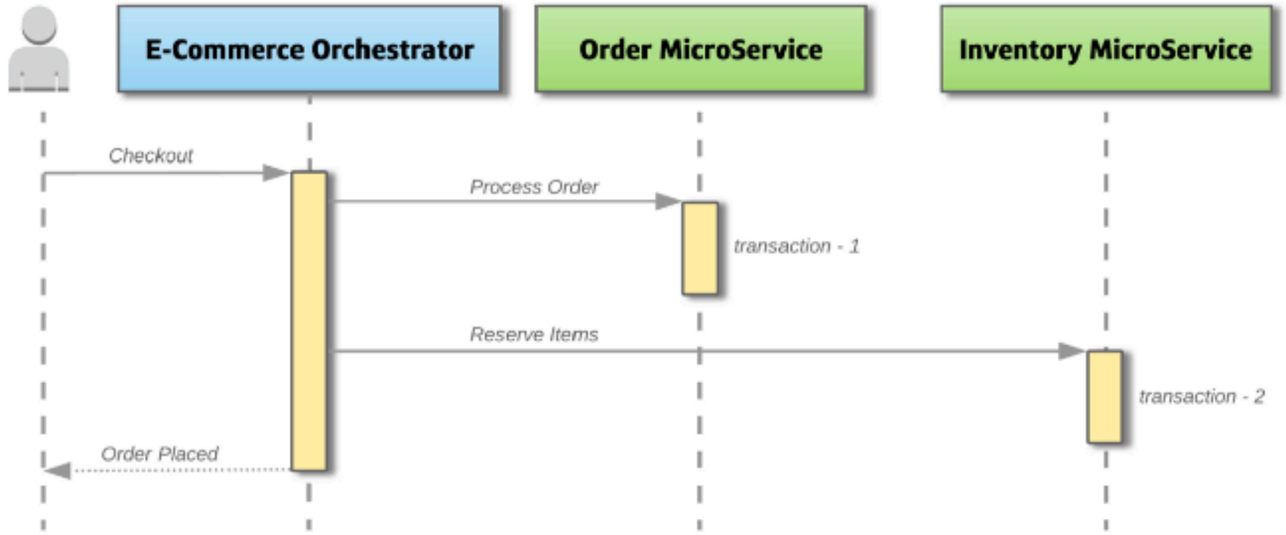
Chapter 4
Transaction Models
: How to Handle Distributed Transactions in MSA

Mijin An
meeeeejin@gmail.com

What Is A Distributed Transaction?



Transaction in a monolith



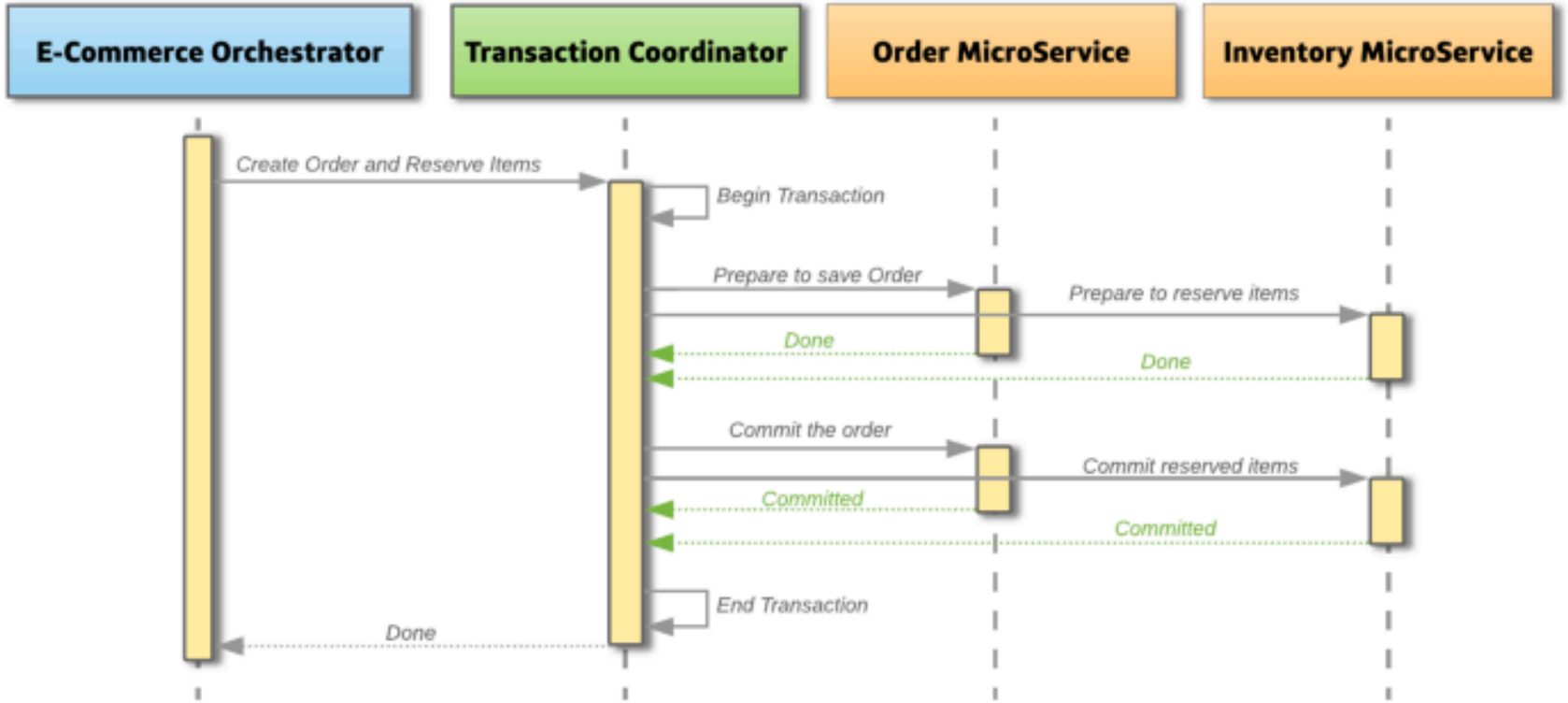
Transactions in a microservice

The Problem with Distributed Transactions in MSA

- **How do we keep the transaction atomic?**
  - Atomicity means “ALL or NOTHING”
  - For example, if the `Reserve Items` fails, how do we rollback the `Process Order`?

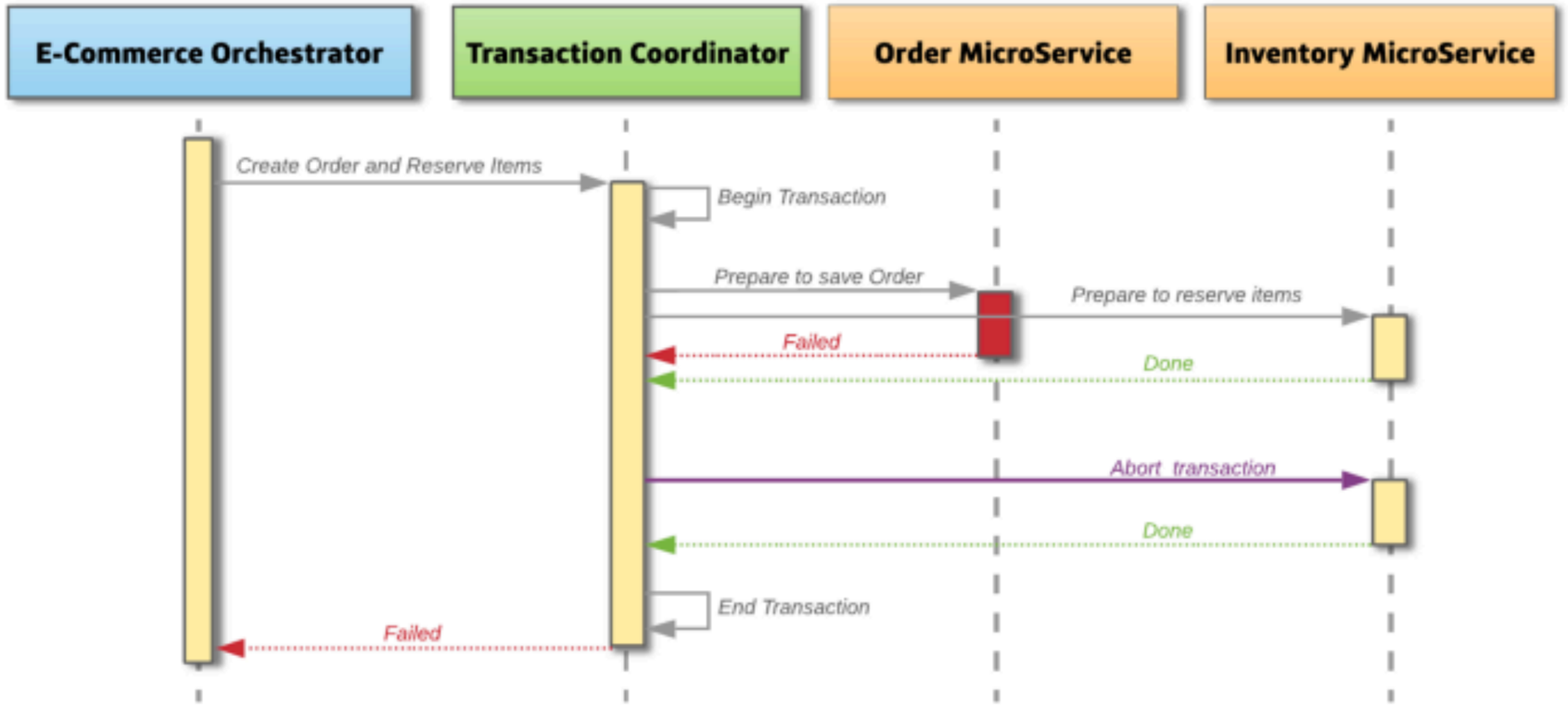
- **How do we handle concurrent requests?**
  - If an object is updated and at the same time, another request reads the same object, should the service return the old data or new?

Possible Solution 1: Two-Phase Commit (1)



- This method has two stages: a **prepare** phase and a **commit** phase
- Also, there is a **Transaction Coordinator** which maintains the lifecycle of the transaction

Possible Solution 1: Two-Phase Commit (2)



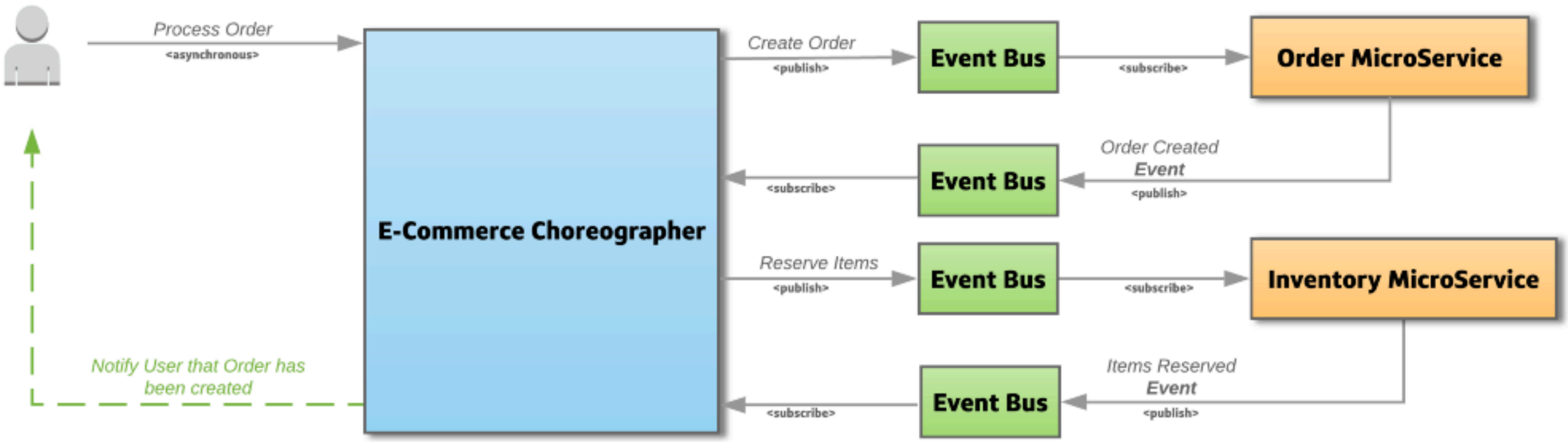
- Above is a **failure** scenario:
  - If at any point a single microservice fails to prepare, the coordinator will abort the transaction and begin the **rollback** process

Possible Solution 1: Two-Phase Commit (3)

- **Advantages:**
  - It guarantees the atomicity of the transaction
  - It is a synchronous call

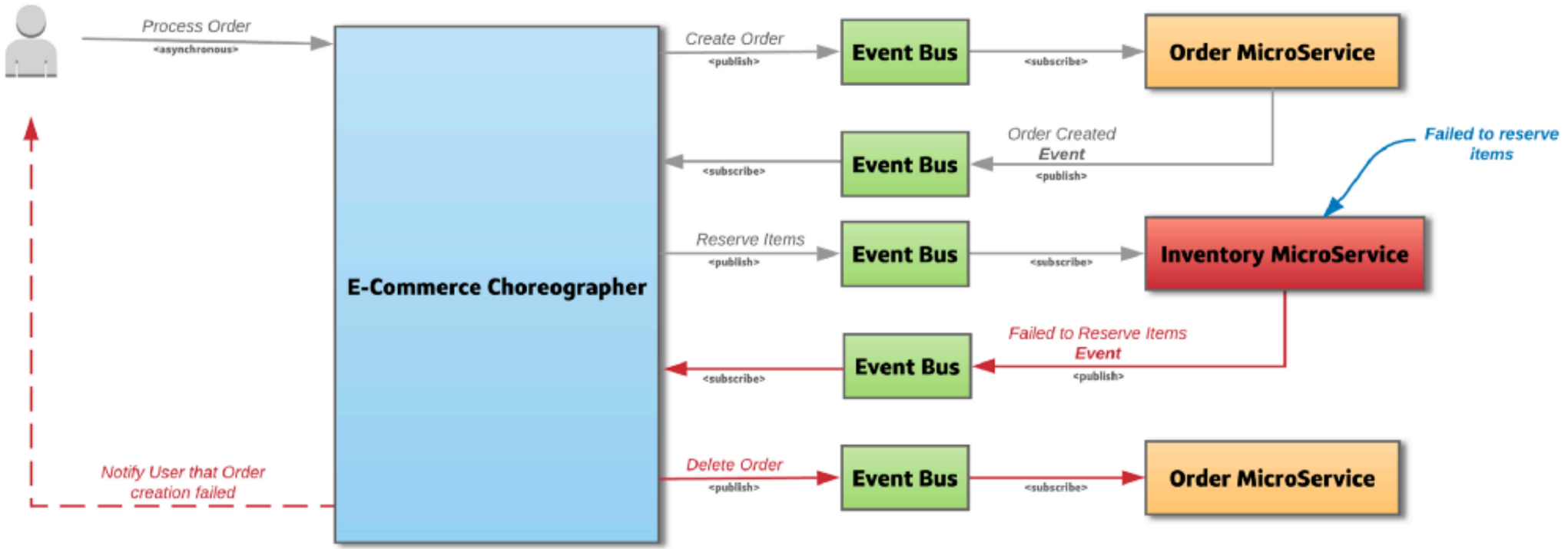
- **Disadvantages:**
  - It is quite slow compared to the time for operation of a single microservice
    - i.e., Highly dependent on the transaction coordinator
  - The lock could become a performance bottleneck
    - Possible to have a deadlock

Possible Solution 2: SAGA (1)



- **Choreography-based SAGA**:
  - Each service publishes an **event** whenever it updates its data
  - Other service subscribe to events
  - When an event is received, a service updates its data

Possible Solution 2: SAGA (2)



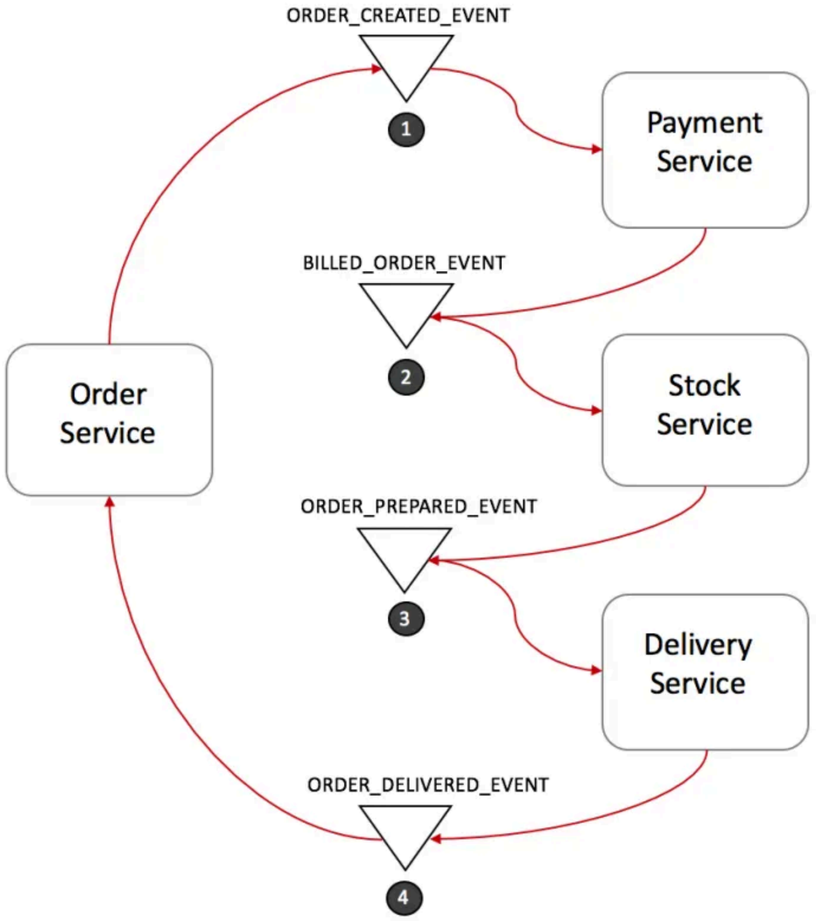
- If any microservice **fails** to reserve the items, it emits a `Failed to Reserve Items` event
- The **choreographer** listens for this event and starts a **compensating transaction** by emitting a `Delete Order` event

Possible Solution 2: SAGA (3)

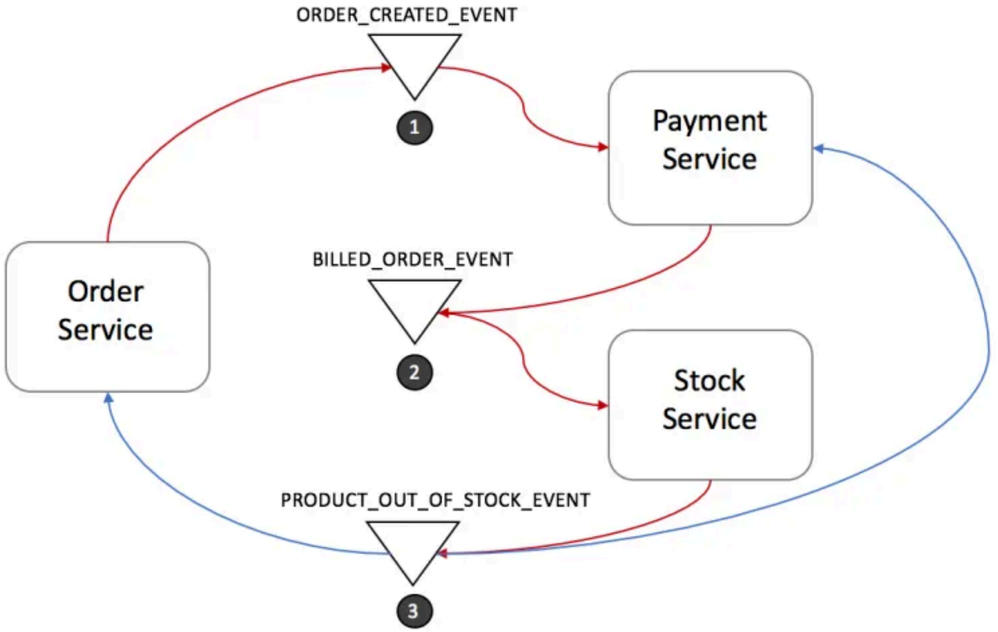
- **Advantages:**
  - Each microservice focuses only on its own atomic transaction
  - There is no database lock required
  - Due to its asynchronous event based solution, it makes the system highly scalable under heavy load

- **Disadvantages:**
  - It does not have read isolation
  - It might add a cyclic dependency between services
  - When the number of microservices increase, it becomes harder to debug and maintain

SAGA: Choreography vs. Orchestration

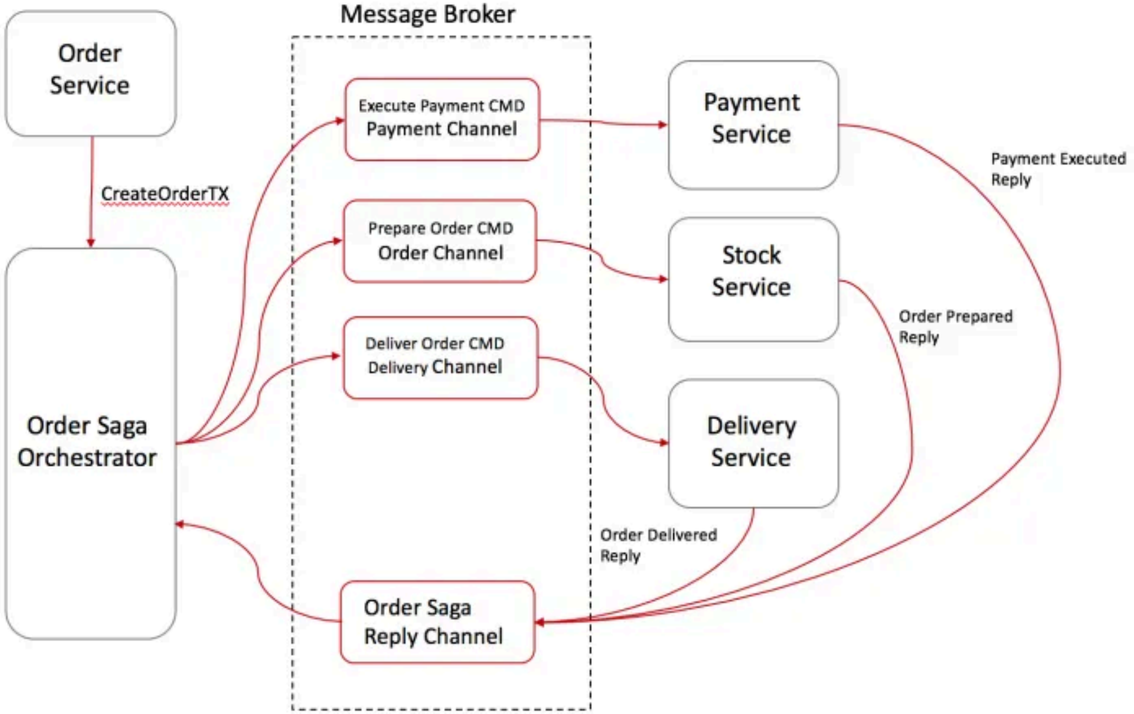


Success

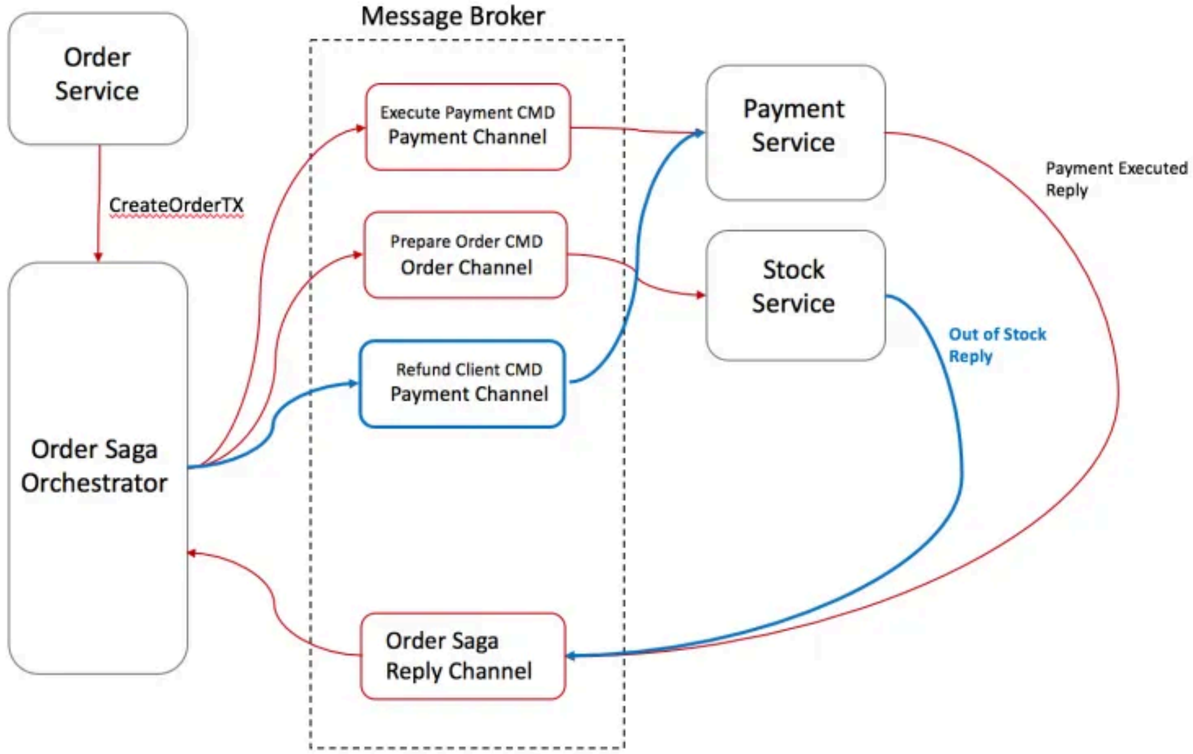


Failure

SAGA: Choreography vs. Orchestration



Success



Failure

Reference

[1] Jim Gray and Andreas Reuter, “Transaction Processing: Concepts and Techniques”, Morgan Kaufmann, San Mateo, CA (1993)

[2] Sohan Ganapathy, “Handling Distributed Transactions in the Microservice world”, Medium, https://medium.com/swlh/handling-transactions-in-the-microservice-world-c77b275813e0

[3] Hector Garcaa-Molrna, Kenneth Salem, “SAGAS”, https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf (1987)

[4] Denis Rosa, “Saga Pattern | How to implement business transactions using Microservices - Part I”, The Couchbase Blog, https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part/

[5] Denis Rosa, “Saga Pattern | How to implement business transactions using Microservices - Part II”, The Couchbase Blog, https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part-2/