Delft University of Technology

Distributed Data Systems

System design document - Group 15

Jan Bryczkowski (5489903)
Stanisław Howard (5533368)
Filip Błaszczyk (5575958)
Daniel Lihotský (5518199)
Christiaan Molier (6334385)
March 21, 2025

1 System Design - SAGA Choreography Design Pattern

We chose the SAGA pattern to handle distributed transactions across our Order, Stock, and Payment services. We chose this over solutions like 2PC for performance reasons. If the aim is to have 10k transactions per minute, we need an asynchronous event-driven solution. SAGA is perfect for this because it allows for multiple distributed transactions happening at the same time and compensation in case of failures without blocking other operations. By implementing the compensating transactions, we ensure that if a payment fails, stock can be restored, the order can be canceled, etc. This will give us an eventually consistent system, which is sufficient in our use case as the stock and account balance can get below the real amount for a brief period and then go back in case of failure as long as it is not below 0 at any point.

We opted for the choreography-based SAGA, where each microservice listens for relevant events and reacts accordingly, eliminating the need for a central orchestrator. We made this decision purely because we did not want to have a single point of failure and a bottleneck in the orchestrator. Below is an example of a successful order checkout step-by-step:

1. An order checkout event is triggered at the **Order Service**.
2. The **Order Service** sends a stock subtract event to the **Stock Servic**e and a pay event to the **Payment Service**.
3. The **Stock Service** receives the stock subtract event and, if possible, subtracts the stock for the given items in the order. If successful, it sends back a completed event to the **Order Service**.
4. The **Payment Service** receives the pay event and, if possible, subtracts the balance for that order's owner. If successful, it sends back a completed event to the **Order Service**.
5. After the **Order Service** receives both completion events, only then can it mark the order status as completed.

   Note: Because of the asynchronous nature of this transaction, the user does not get an immediate response back from the server. Instead, the order will get marked as completed or failed at some point in the future, and the user can poll the server for the order's status and receive an immediate processing/completed/failed response.

This concludes a successful order checkout. If at step 3 or 4 the transaction cannot go through a compensation must be sent out. For example if the balance is too low and the transaction cannot be completed at the payment service, the payment service sends a compensation event to the stock service - to get back the stock that did not get paid for and it sends a transaction unsuccessful event back to the order service. The order service then marks the order as failed.

In the case of a database failure during a transaction the affected service also sends a compensation. For example if the payment service tries to subtract the money and the payment database is down it will do the same as if the money was not there.

In the case of a microservice failure, nothing will happen right away but once the service goes back up it will know at what point it died and it will continue/compensate depending on the situation. Because of our use of Kafka (which we will go into more detail later in this document) no events will be lost in the case of microservice failure.

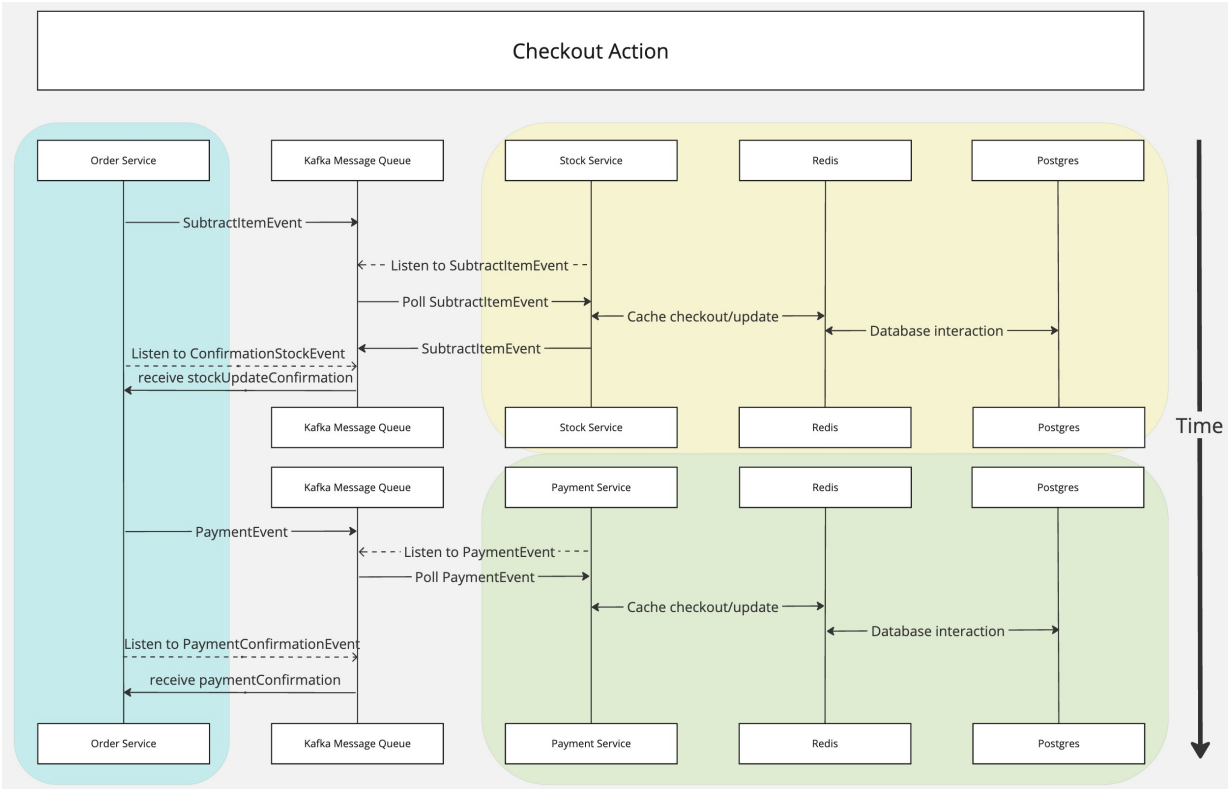

Figure 1: Checkout action flow between components

2 Storing Data in the Databases - PostgreSQL & Redis

In our project, we chose PostgreSQL as the main database because it provides strong consistency guarantees and supports multi-partition transactions. Since our system involves multiple microservices working together, we need a way to ensure that all database updates remain consistent even when they span different services.

To make our system faster and reduce the load on PostgreSQL, we use Redis as a caching layer. Redis is beneficial because it stores frequently accessed data in memory, allowing us to retrieve it much faster than querying the database every time. For example, when a user checks stock availability or views an order, instead of repeatedly querying PostgreSQL, we first check Redis. This significantly improves response times and helps reduce unnecessary database queries.

Cached data expires after a certain time to prevent it from becoming outdated. Whenever an important change is made in PostgreSQL, we update or remove the corresponding Redis entry to keep data accurate. Additionally, Redis helps with real-time updates by using its *pub/sub* system, which allows different parts of our system to immediately react to stock changes or payment updates.

We also use Redis for distributed locking to avoid race conditions when multiple users try to update the same data at the same time. When data is updated in PostgreSQL, we ensure that Redis reflects those changes so that cached information remains accurate. For critical updates like payment transactions, we always query PostgreSQL to ensure we do not rely on potentially outdated information.

3 Communication Between Services - Kafka

As mentioned above, our goal is to be able to handle asynchronous actions that are triggered by each microservice. We chose Kafka for two main reasons:

- **Fault tolerance** - This allows us to abstract the logic of saving the events that have been triggered but not yet processed by the microservices. It guarantees that all events sent to a topic will be processed in the correct order.
- **Good scalability** - For our scenario of 10,000 calls per minute, Kafka should be ideal to store the events and allow the services to poll them at their own pace.

The general idea of utilizing Kafka is to ensure reliable, scalable, and event-driven communication between our microservices. We plan to have a separate topic for each event type triggered. This way each service has its own queue from which it polls for events

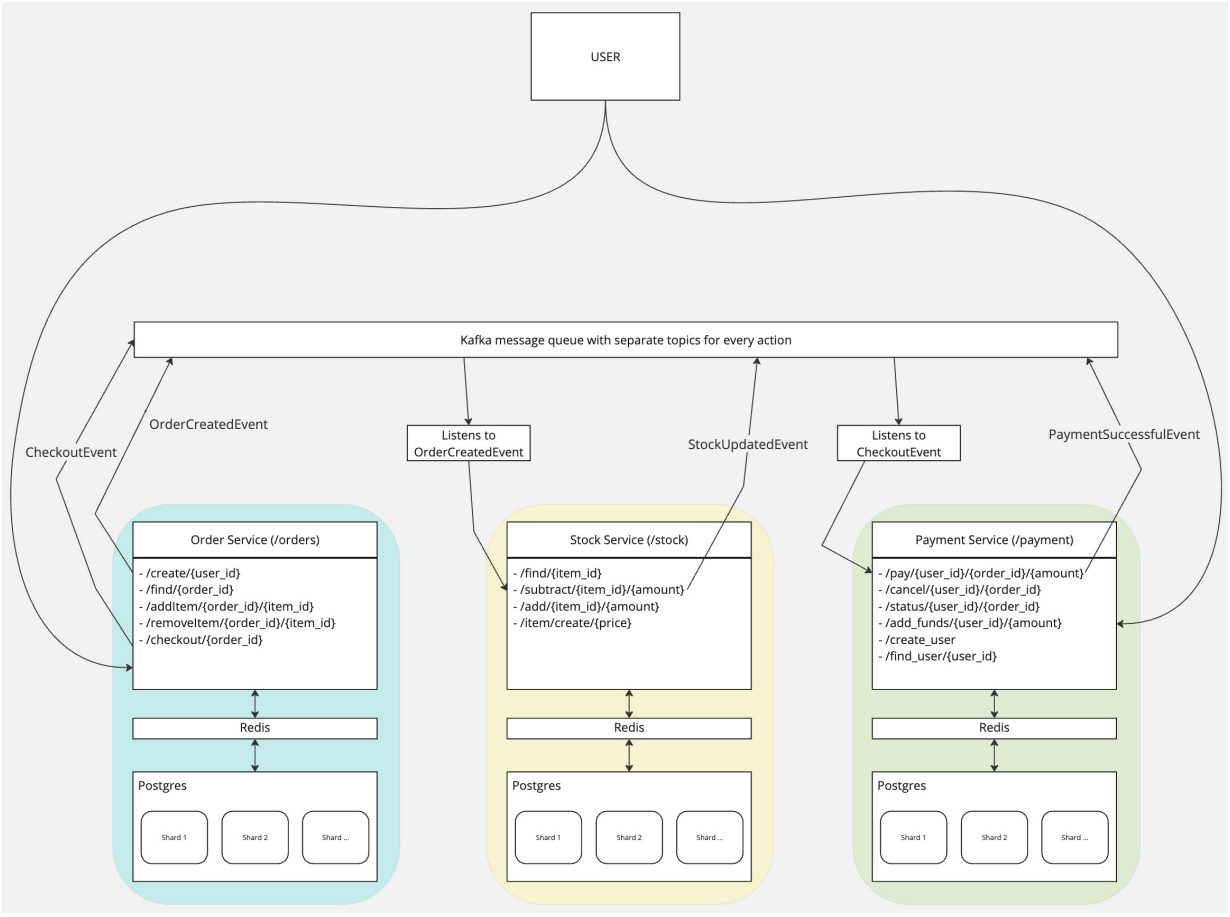

Figure 2: System interactions