**Saga Pattern**

* **Introduction:**

Saga’s are mechanism to maintain data consistency in a microservice architecture without having to use distributed transactions.

You define a Saga for each system command that needs to update data in multiple services.

A Saga is a series of local transactions. Each local transaction updates data within a single service using the familiar ACID transaction frameworks and libraries.

The system operation initiates the first step of the Saga. The completion of local transaction triggers the execution of the next local transaction.

Sagas differ from ACID transactions in a couple of important ways.

They lack the isolation property of ACID transactions. As a result, an application must use what are known as “countermeasures” design techniques that prevent or reduce the impact of concurrency anomalies caused by lock of isolations.

Each local transaction commits its changes, a Saga must be rolled back using compensating transactions.

A screenshot of a cell phone

Description automatically generated

This saga consists of the following local transactions:

1 Order Service—Create an Order in an APPROVAL\_PENDING state.

2 Consumer Service—Verify that the consumer can place an order.

3 Kitchen Service—Validate order details and create a Ticket in the CREATE

\_PENDING.

4 Accounting Service—Authorize consumer’s credit card.

5 Kitchen Service—Change the state of the Ticket to AWAITING\_ACCEPTANCE.

6 Order Service—Change the state of the Order to APPROVED.

* **Sagas use compensating transaction to roll back changes:**

If any one of the transactions fails, we have to rollback the earlier transaction committed so far, to achieve this we need a “compensating transaction”.

If the transaction fails at T(n+1) this has to revert the commit made for previous transactions (Tn…T1). This is achieved by “compensating transaction”.

The saga executes the compensating transactions in reverse order of the forward transactions Cn….C1.

Pivot transactions - the transaction which followed by the steps which never fail.

Retriable transactions - The transactions which always succeeds

* **Coordinating Sagas:**

A Saga’s implementation consists of logic that coordinates the step of Saga.

When a Saga is initiated by a system command, the coordination logic must select and tell the first Saga participant to execute a local transaction. Once the transaction completes, the Saga’s sequencing coordination selects and invokes the next Saga participant. This process continues until the Saga has executed all the steps.

If any local transaction fails, the Saga must execute the compensating transaction in reverse order.

There are couple of different ways to structure a saga’s coordination logic.

**Choreography:**

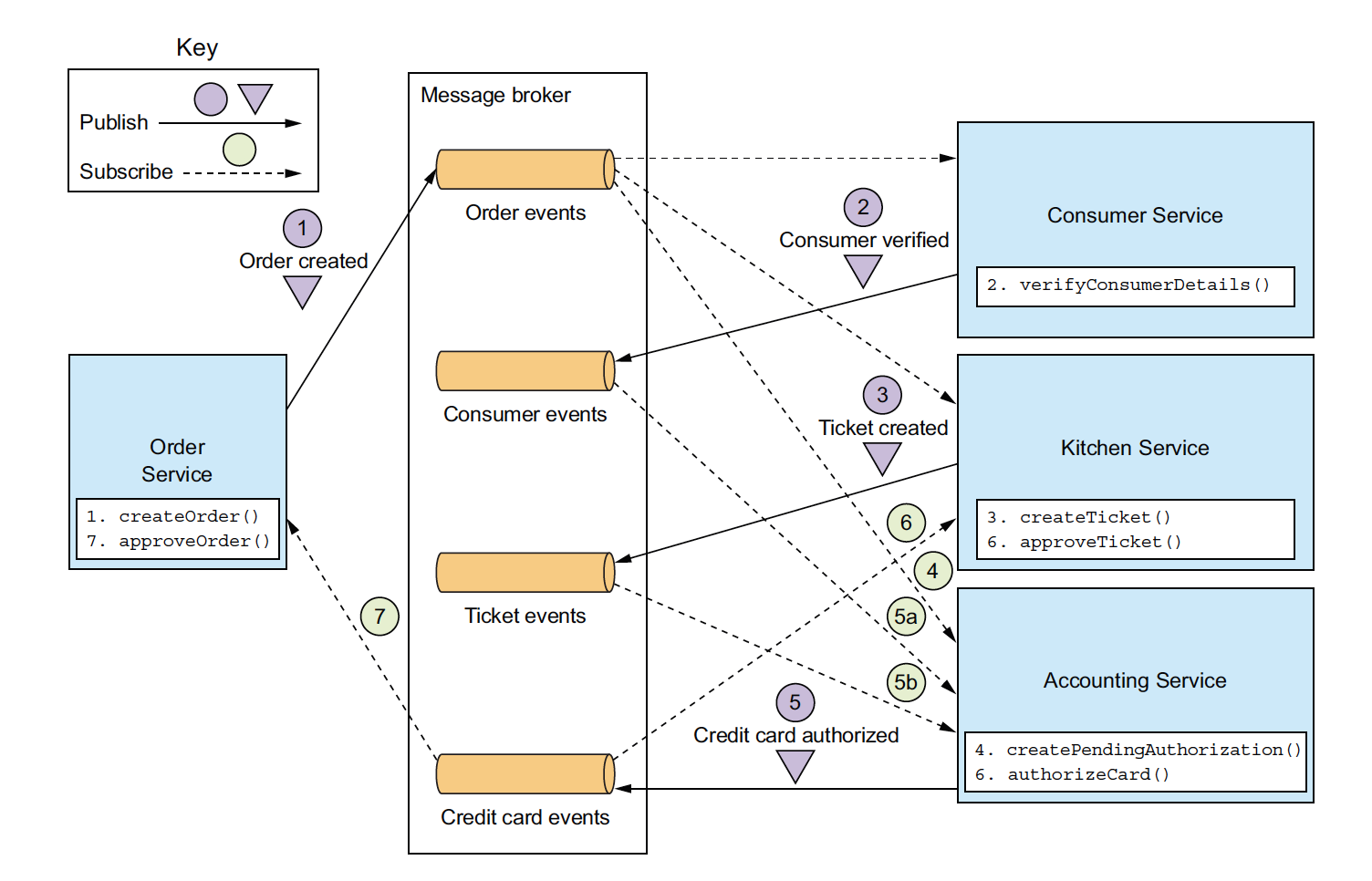
Distribute the decision making and sequencing among the saga participants. They primarily communicate by exchanging events.

**Orchestration:**

Centralize a saga’s coordination logic in a saga orchestrator class. A saga orchestrator sends command messages to saga participants telling them which operation to perform.

* **Choreography-based Sagas:**

When using choreography, there’s no central coordinator telling the Saga participants what to do. Instead, the Saga participants subscribe to each other’s events and respond accordingly.



**IMPLEMENTING THE CREATE ORDER SAGA USING CHOREOGRAPHY:**

Above figure shows the design of choreography-based version create order saga.

The participants communicate by exchanging events.

Each participant, starting with the OrderService, updates it database and publish an event that triggers the next participant.

The happy path through this saga is as follows:

1. OrderService creates an order in the APPROVAL\_PENDING state and publishes an OrderCreated event.
2. Consumer service consumes the OrderCreated event, verifies that the consumer can place the order, and publishes a ConsumerVerified event.
3. KitchenService consumes the OrderCreated event, validates the Order, creates a Ticket in a CREATE\_PENDING state, and publishes the TicketCreated event.
4. AccountinService consumes the OrderCreated event and creates a CreditCardAuthorization in a pending state.
5. AccountingService consumes the TicketCreated and ConsumerVerified events, charge the consumer’s credit card and publishes the CreditCardAuthorized event.
6. KitchenService consumes the CreditCardAuthorized event and changes the state of the Ticket to AWAITING\_ACCEPTANCE
7. OrderService received the CreditCardAuthorized events, changes the state of the order to APPROVED and publish an OrderApproved event.

The create order saga must also handle the scenario where a saga participant rejects the Order and publishes some kind of failure event.

For example, the authorization of the consumer credit card might fail. The saga must execute the compensating transactions to undo what’s already been done.

Below figure shows the flow of events when the AccountingService can’t authorize the consumer’s credit card.

Diagram

Description automatically generated

The sequence of events as follows:

1. OrderService creates an order in the APPROVAL\_PENDING state and publishes an OrderCreated event.
2. Consumer service consumes the OrderCreated event, verifies that the consumer can place the order, and publishes a ConsumerVerified event.
3. KitchenService consumes the OrderCreated event, validates the Order, creates a Ticket in a CREATE\_PENDING state, and publishes the TicketCreated event.
4. AccountinService consumes the OrderCreated event and creates a CreditCardAuthorization in a pending state.
5. AccountingService consumes the TicketCreated and ConsumerVerified events, charge the consumer’s credit card and publishes the CreditCardAuthorizationFailed event.
6. KitchenService consumes the CreditCard AuthorizationFailedEvent and changes the state of the Ticket to REJECTED.
7. OrderService consumes the CreditCard AuthorizationFailed event and changes the state of the Order to REJECTED

As you can see, the participants of choreography-based sagas interact using publish/subscribe. Let’s take a closer look at some issues you’ll need to consider when implementing publish/subscribe based communications for your sagas.

**RELIABLE EVENT-BASED COMMUNICATION:**

There are couple of interservice communication-related issues that you must consider when implementing choreography-based sagas.

1. The first issue is ensuring that a saga participant updates its database and publishes an event as part of database transaction.

Each step of choreography-based sagas updates the database and publishes an event.

It is essential that updating database and publishing an event happen atomically. To communicate reliably, the saga participants must use transactional messaging.

1. The second issue you need to consider is ensuring that a saga participant must be able to map each event that it receives to its own data.

For example, when OrderService receives a CreditCardAuthorized event, it must be able to lookup the corresponding order. The solution is for a saga participant to publish an events containing a correlation-id, which is data that enables other participants to perform the mapping.

For example, the participants of the create order saga can used orderId as a correlation-id that’s passed from one participant to the next.

**BENEFITS AND DRAWBACKS OF CHOREOGRAPHY-BASED SAGAS:**

Choreography-based sagas have several benefits:

Simplicity:

Services publish events when they create, update or delete business objects.

Loose coupling:

The participants subscribe to events and don’t have direct knowledge on each other.

And it has following drawbacks:

More difficult to understand:

Unlike with orchestration, there isn’t a single place in the code that defines saga. Instead, choreography distributes the implementation of saga among the services. Consequently, it’s sometimes difficult for a developer to understand how a given saga works.

Cycling dependencies between services:

The saga participants subscribe to each other’s events, which often creates cyclic dependencies. In the example OrderService saga, we can see a cyclic dependency.

OrderService -> AccountingService -> OrderService

Note: This is not necessarily a problem, cyclic dependencies are considered as design smell.

Risk of tight coupling:

Each saga participant need to subscribe to all events that affect them. For example, AccountingService must subscribe to all events that cause the consumer’s credit card to be charged or refunded. As a result, there’s a risk that it would need to be updated in lockstep with the order lifecycle implemented by OrderService.

Note: Choreography can work well for simple sagas, but because of these drawbacks it’s often better for more complex sagas to use orchestration.

* **Orchestration-based sagas:**

Orchestration is another way to implement sagas.

When using orchestration, you define an orchestrator class whose sole responsibility is to tell the saga participants what to do.

The saga orchestrator communicates with the participants using command / async reply-style interaction.

To execute a saga step, it sends a command message to a participant telling it what operation to perform. After the Saga implementation has performed the operation, it sends a reply message to the orchestrator. The orchestrator then process the message and determines which saga step to perform next.

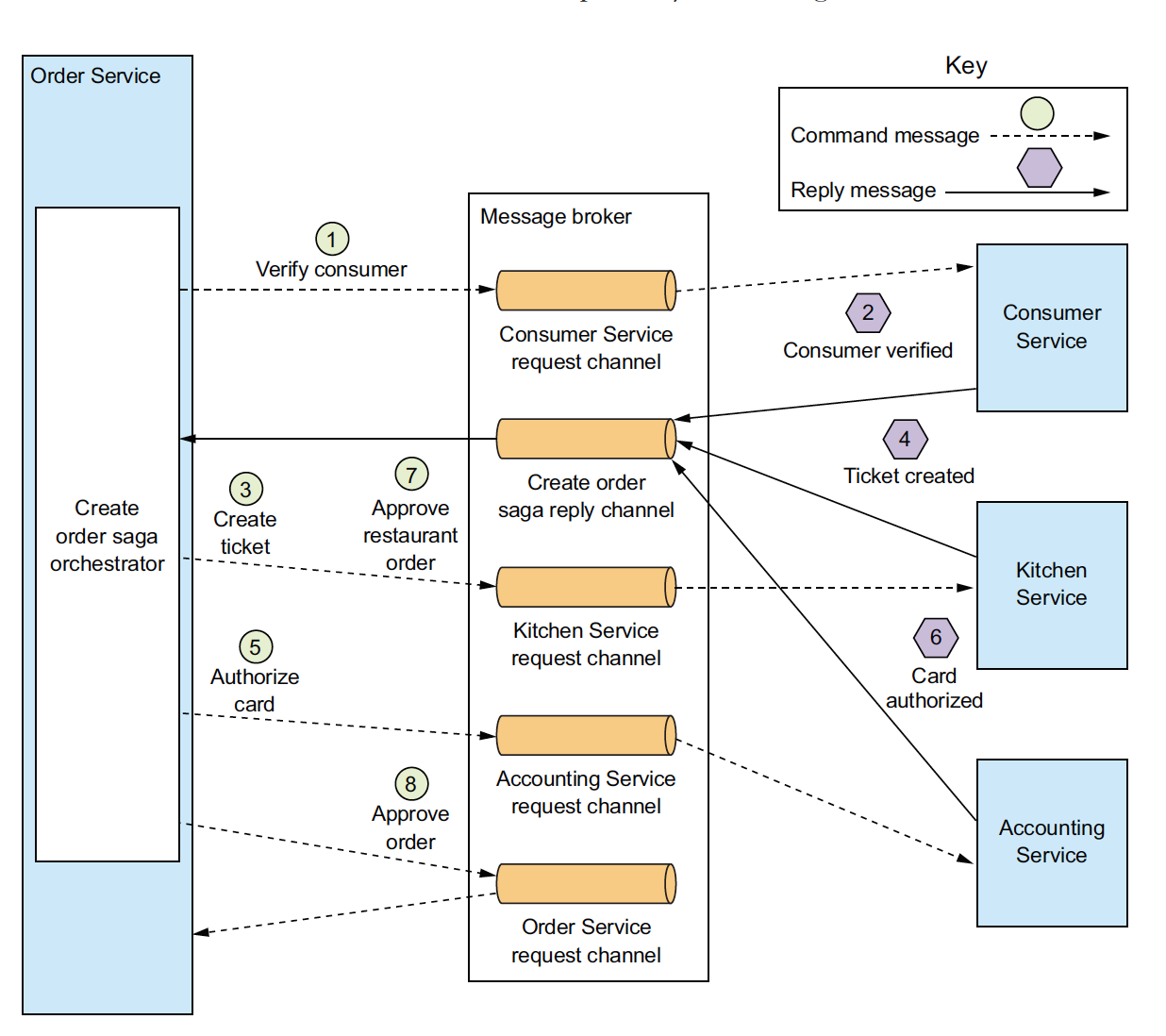
**IMPLEMENTING THE CREATE ORDER SAGA USING ORCHESTRATION:**

Below diagram show the design of orchestration-based version of CreateOrder saga.

The Saga is orchestrated by the CreateOrderSaga class, which invokes the saga participants using asynchronous request/response.

This class keeps track of the process and sends command message to saga participants, such as KitchenService and ConsumerService.

The CreateOrderSaga reads reply messages from its reply-channel and then determines the next step, if any, in the saga.



Order service first creates an order and CreateOrderSaga orchestrator. After that, the flow for the happy path is as follows.

1. The Saga orchestrator sends a Verify Consumer command to ConsumerService.
2. ConsumerService replies with a ConsumerVerified message.
3. The Saga Orchestrator sends a CreateTicket command to KitchenService.
4. KitchenService replies with a TicketCreated message.
5. The Saga orchestrator sends an Authorize Card message to AccountingService.
6. AccountingService replies with a CardAuthorized message.
7. The Saga orchestrator sends an ApproveTicket command to KitchenService.
8. The Saga orchestrator sends an ApproveOrder command to OrderService.

Note that in final step, the saga orchestrator sends a command message to OrderService, even though it is a component of OrderService. In principle, the CreateOrderSaga could approve the order by updating it directly. But in order to be consistent, the Saga treats the OrderService as just another participant.

Above diagram depict one scenario of Saga, but a Saga is likely to have numerous scenarios. For example, the create order saga has four scenarios. In addition to the happy path, the saga can fail due to a failure in either ConsumerService, KitchenService, or AccountingService. It is uselful, to model a saga as a state machine, because it describes all possible scenarios.

* **MODELING SAGA ORCHESTRATORS AS STATE MACHINES:**

A good way to model a Saga orchestrator is as a state machine.

A state machine consists of a set of states and a set of transitions between states that are triggered by events.

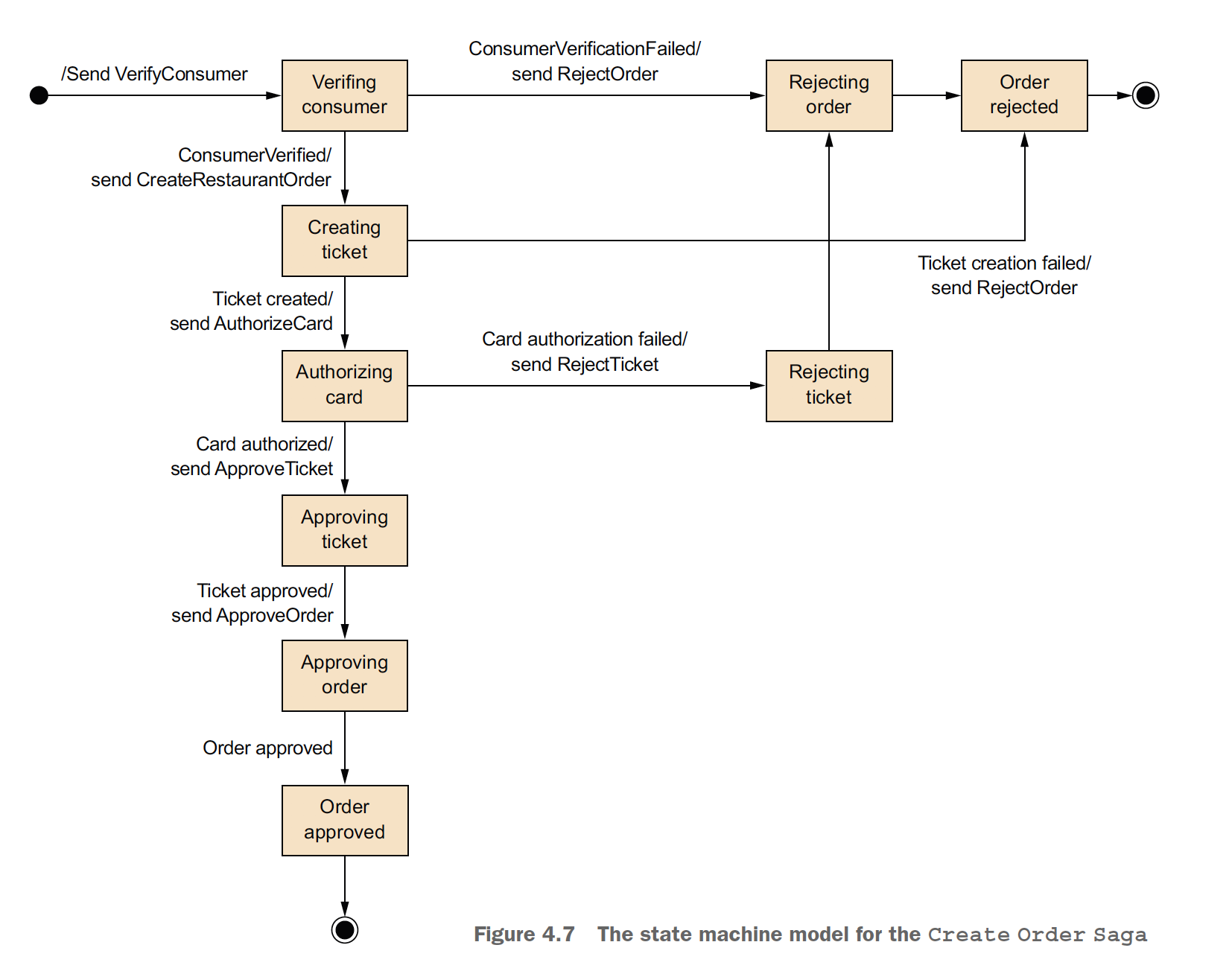
Each transition can have an action, which for a saga is the invocation of a saga participant.

The transition between states are triggered by the completion of a local transaction performed by a saga participant.

The current state and the specific outcome of the local transaction determine the state transition and what action, if any, to perform.

There are also effective testing strategies for state machines. As a result using a state machine model makes designing, implementing, and testing sagas easier.

Below figure shows the state machine model for the Create order saga. This state machine consists of numerous states, including the following.



1. Verifying consumer - The initial state. When in this state, the saga is waiting for the ConsumerService to verify that the consumer can place the order.
2. Creating Ticket – The saga is waiting for a reply to the Create Ticket command.
3. Authorizing Card – Waiting for AccountingService to authorize the consumer’s credit card.
4. Order Approved – A final state indicating that the saga completed successfully.
5. Order Rejected – A final state indicating that the Order was rejected by one of the participants.

**SAGA ORCHESTRATION AND TRANSACTIONAL MESSAGING:**

Each step of orchestration-based saga consist of service updating a database and publishing a message.

As described earlier a service must use the transactional messaging in order to automatically update the database and publish the message.

**BENEFITS AND DRAWBACKS OF ORCHESTRATION-BASED SAGA:**

Simpler dependencies:

One benefit of orchestration is that it doesn’t introduce cyclic dependencies.

The saga orchestrator invokes saga participants, but the participants don’t invoke the orchestrator. Because of this there is no cyclic dependencies.

Less Coupling:

Each service implements an API that is invoked by the orchestrator, so it doesn’t need to know about the events published by the saga participants.

Improves separation of concerns and simplifies the business logic:

The saga coordination logic is localized in the saga orchestrator. The domain objects are simpler and have no knowledge of the saga that they participate in.

* **Handling the lack of isolation:**

The I in ACID stands for isolation. The isolation property of ACID transactions ensures that the outcome of executing multiple transactions concurrently is the same as if they were executed in some serial order.

The database provides the illusion that each ACID transaction has exclusive access to data.

Isolation makes it a lot easier to write business logic that executes concurrently.

The challenge with using sagas is that they lack the isolation property of ACID transactions. That’s because the updates made by each of saga’s local transactions are immediately visible to other sagas once that transactions commit.

This behavior can cause two problems:

1. Other sagas can change the data accessed by the saga while its executing. And other sagas can read its data before the saga has completed its update, and consequently can be exposed to inconsistent data. We can consider a saga to be ACD.

Atomicity:

The saga implementation ensures that all transactions are executed, or all changes are undone.

Consistency:

Referential integrity within a service is handled by local databases. Referential integrity across services is handled by the services.

Durability:

Handled by local databases.

This lack of isolation potentially causes what the database literature calls anomalies. An anomaly is when a transaction reads or writes data in a way that it wouldn’t if transactions were executed one at time. When an anomaly occurs, the outcome of executing sagas concurrently is different than if they were executed serially.

We will discuss a set of design strategies that deal with the lack of isolation. These strategies are known as “countermeasures”.

Some countermeasures implement isolation at application level. Other countermeasures reduce the business risk of the lack of isolation. By using countermeasures, you can write saga-based business logic that works correctly.

* **Overview of Anomalies:**

The lack of isolation can cause the following three anomalies.

Lost updates:

One saga overwrites without reading changes made by another saga.

Dirty reads:

A transaction or a saga reads the updates made by a saga that has not yet completed those updates.

Fuzzy / nonrepeatable reads:

Two different steps of saga read the same data and get different results because another saga has made updates.

All three anomalies can occur, but the first two are the most common and most challenging.

Lost Updates:

A lost update anomaly occurs when one saga overwrites an update made by another saga. Example scenario;

1. The first step of the create order saga creates the order.
2. While that saga is executing, the cancel order saga cancels the Order.
3. The final step of the create order saga approves the order.

In this scenario, the create order saga ignores the update made by the cancel order saga and overwrites it. As a result, the FTGO application will ship an order that the customer had cancelled.

Dirty Reads:

A dirty read occurs when one saga reads data that’s in the middle of being updated by another saga.