Transaction Management in Microservices

In a microservice architecture, transactions that are within a single service can still use ACID transactions. The challenge, however, lies in implementing transactions for operations that update data owned by multiple services.

For example, as described in chapter 2, the createOrder() operation spans numerous services, including Order Service, Kitchen Service, and Accounting Service. Operations such as **these need a transaction management mechanism that works across services.**

**the traditional approach to distributed transaction management isn’t a good choice for modern applications**.

Instead of an ACID transactions, an operation that spans services must use what’s known as a *saga*, **a message-driven sequence of local transactions, to maintain data consistency.**

**One challenge with sagas is that they are ACD (Atomicity, Consistency, Durability). They lack the isolation feature of traditional 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 the lack of isolation.**

*Probably the good news is:*

in reality, even monolithic applications such as the FTGO application typically don’t use textbook ACID transactions. For example,

* **many applications use a lower transaction isolation level in order to improve performance.**
* **Also, many important business processes, such as transferring money between accounts at different banks, are eventually consistent.** Not even Starbucks uses two-phase commit ([www.enterpriseintegrationpatterns.com/ramblings/18\_starbucks.html](http://www.enterpriseintegrationpatterns.com/ramblings/18_starbucks.html)).

# The trouble with distributed transactions

The traditional approach to maintaining data consistency across multiple services, databases, or message brokers is to use distributed transactions. The de facto standard for distributed transaction management is the X/Open Distributed Transaction Processing (DTP) Model (X/Open XA—see https://en.wikipedia.org/wiki/X/Open\_XA).

XA uses *two-phase commit* (2PC) to ensure that all participants in a transaction either commit or rollback. An XA-compliant technology stack **consists of XA-compliant databases** **and message brokers**, **database drivers,** **and messaging APIs**, and **an inter-process communication mechanism that propagates the XA global transaction ID.** Most SQL databases are XA compliant, as are some message brokers. Java EE applications can, for example, use JTA to perform distributed transactions.

As simple as this sounds, there are a variety of problems with distributed transactions.

* One problem is that **many modern technologies, including NoSQL databases** such as MongoDB and Cassandra, **don’t support them**.

Also**,** distributed transactions aren’t supported by modern message brokers such as RabbitMQ and Apache Kafka(now I think it does). As a result, if you insist on using distributed transactions, you can’t use many modern technologies.

* Another problem with distributed transactions is that **they are a form of synchronous IPC, which reduces availability.** In order for a distributed transaction to commit, all the participating services must be available. As described in chapter 3, the availability is the product of the availability of all of the participants in the transaction. If a distributed transaction involves two services that are 99.5% available, then the overall availability is 99%, which is significantly less. Each additional service involved in a distributed transaction further reduces availability.

*Get back to this paragraph later:*

There is even Eric Brewer’s CAP theorem, which states that a system can only have two of the following three properties:

consistency, availability, and partition tolerance (https://en.wikipedia.org/wiki/CAP\_theorem). Today, architects prefer to have a system that’s available rather than one that’s consistent.

On the surface, distributed transactions are appealing. From a developer’s perspective, they have the same programming model as local transactions.

But because of the problems mentioned so far, distributed transactions aren’t a viable technology for

modern applications.

Chapter 3 described how to send messages as part of a database transaction **withou**t using distributed transactions.

To solve the more complex problem of maintaining data consistency in a microservice architecture, an application must use a different mechanism that **builds on the concept of loosely coupled, asynchronous services. This is where sagas come in.**

# Using the Saga pattern to maintain data consistency

***Sagas* are mechanisms to maintain data consistency in a microservice architecture using asynchronous messaging 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 sequence of local transactions.** Each local transaction updates data within a single service using the familiar ACID transaction frameworks and libraries mentioned earlier.

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

An important benefit of asynchronous messaging is that it ensures that all the steps of a saga are executed, even if one or more of the saga’s participants is temporarily unavailable.

Sagas differ from ACID transactions in a couple of important ways. **they lack the isolation property of ACID transactions**. Also, **because each local transaction commits its changes, a saga must be rolled back using compensating transactions.**

## Example, Create Order Saga

The saga’s first local transaction is initiated by the external request to create an order. The other five local transactions are each triggered by completion of the previous one.

This saga consists of the following local transactions:

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

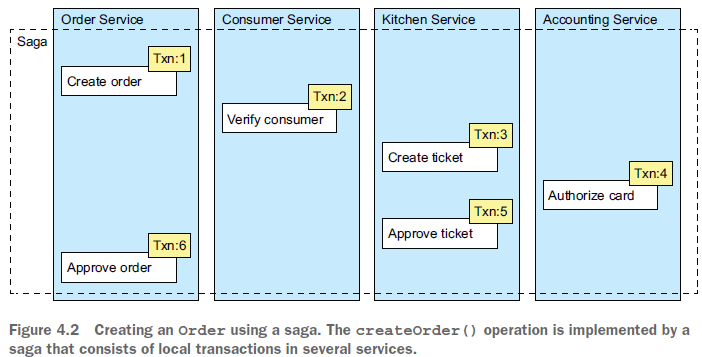
2 Consumer Service—Verify that the consumer can place an order.

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.



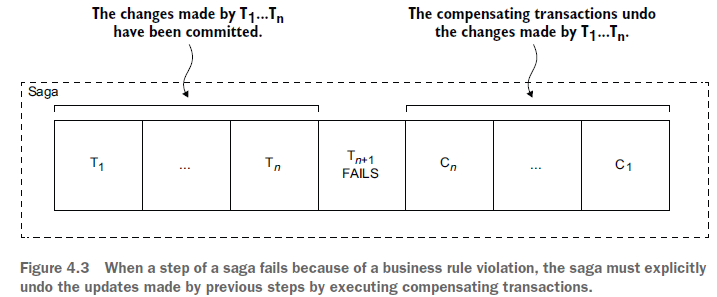
Later, in section 4.2, I describe how the services that participate in a saga communicate using asynchronous messaging. A service publishes a message when a local transaction completes. This message then triggers the next step in the saga. Not only does using messaging ensure the saga participants are loosely coupled, it also guarantees that a saga completes. That’s because if the recipient of a message is temporarily unavailable, the message broker buffers the message until it can be delivered.

On the surface, sagas seem straightforward, but there are a few challenges to using them. One challenge is the lack of isolation between sagas. Section 4.3 describes how to handle this problem. Another challenge is rolling back changes when an error occurs. Let’s take a look at how to do that.

## Using Compensation Transactions to Rollback Changes

A great feature of traditional ACID transactions is that the business logic can easily roll back a transaction if it detects the violation of a business rule. It executes a ROLLBACK statement, and the database undoes all the changes made so far. Unfortunately, sagas can’t be automatically rolled back, because each step commits its changes to the local database. This means, for example, that if the authorization of the credit card fails in the fourth step of the Create Order Saga, the FTGO application must explicitly undo the changes made by the first three steps. You must write what are known as *compensating transactions*.

Suppose that the (*n* + 1)th transaction of a saga fails. The effects of the previous *n* transactions must be undone. Conceptually, each of those steps, Ti, has a correspondingcompensating transaction, Ci, which undoes the effects of the Ti. To undo theeffects of those first *n* steps, the saga must execute each Ci in reverse order. **Thesequence of steps is T1 … Tn, Cn … C1, as shown in figure 4.3. In this example, Tn+1fails, which requires steps T1 … Tn to be undone.**



The saga executes the compensation transactions in reverse order of the forward transactions: Cn … C1. The mechanics of sequencing the Cis aren’t any different than sequencing the Tis. **The completion of Ci must trigger the execution of Ci-1.**

Consider, for example, the Create Order Saga. This saga can fail for a variety of reasons:

 The consumer information is invalid or the consumer isn’t allowed to create orders.

 The restaurant information is invalid or the restaurant is unable to accept orders.

 The authorization of the consumer’s credit card fails.

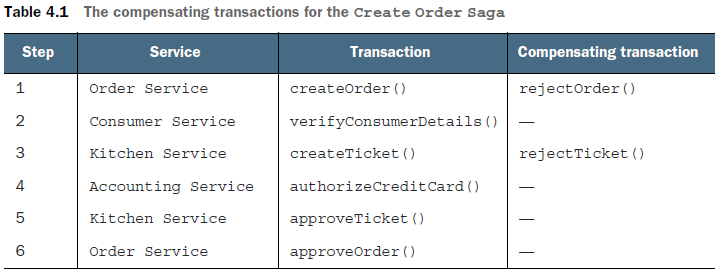
If a local transaction fails, the saga’s coordination mechanism must execute compensating transactions that reject the Order and possibly the Ticket. Table 4.1 shows the compensating transactions for each step of the Create Order Saga.

It’s important to note that **not all steps need compensating transactions**. Read-only steps, such as verify-ConsumerDetails(), don’t need compensating transactions. **Nor do steps such as authorizeCreditCard() that are followed by steps that always succeed**.

**How about when a data base Constraint is violated in those two steps?**

Section 4.3 discusses how:

* the first three steps of the Create Order Saga are termed ***compensatable transactions***because **they’re followed by steps that can fail**
* how the fourth step is termed the saga’s ***pivot transaction* because it’s followed by steps that never fail**
* and how the last two steps are termed ***retriable transactions***because they always succeed.



To see how compensating transactions are used, imagine a scenario where the authorization of the consumer’s credit card fails. In this scenario, the saga executes 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 state.

4 Accounting Service—Authorize consumer’s credit card, which fails.

5 Kitchen Service—Change the state of the Ticket to CREATE\_REJECTED.

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

The fifth and sixth steps are compensating transactions that undo the updates made by Kitchen Service and Order Service, respectively**. A saga’s coordination logic is responsible for sequencing the execution of forward and compensating transactions**. Let’s look at how that works.

## Coordinating Sagas

A saga’s implementation consists of logic that coordinates the steps of the saga.

When a saga is initiated by system command, the coordination logic must select and

tell the first saga participant to execute a local transaction. Once that 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 transactions in

reverse order. There are a 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 operations 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.**

**Each participant**, starting with the Order Service, **updates its database and publishes an event that triggers the next participant**.

The happy path through this saga is as follows:

1 Order Service 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 Kitchen Service consumes the OrderCreated event, validates the Order, creates a Ticket in a CREATE\_PENDING state, and publishes the TicketCreated event.

4 Accounting Service consumes the OrderCreated event and creates a CreditCardAuthorization in a PENDING state.

5 Accounting Service consumes the TicketCreated and ConsumerVerified events, charges the consumer’s credit card, and publishes the CreditCard-Authorized event.

6 Kitchen Service consumes the CreditCardAuthorized event and changes the state of the Ticket to AWAITING\_ACCEPTANCE.

7 Order Service receives the CreditCardAuthorized events, changes the state of

the Order to APPROVED, and publishes 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’s credit card might fail. The saga must execute the compensating transactions to undo what’s already been done. Figure 4.5 shows the flow of events when the AccountingService can’t authorize the consumer’s credit card.

The sequence of events is as follows:

1 Order Service 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 Kitchen Service consumes the OrderCreated event, validates the Order, creates

a Ticket in a CREATE\_PENDING state, and publishes the TicketCreated event.

4 Accounting Service consumes the OrderCreated event and creates a Credit-

CardAuthorization in a PENDING state.

5 Accounting Service consumes the TicketCreated and ConsumerVerified

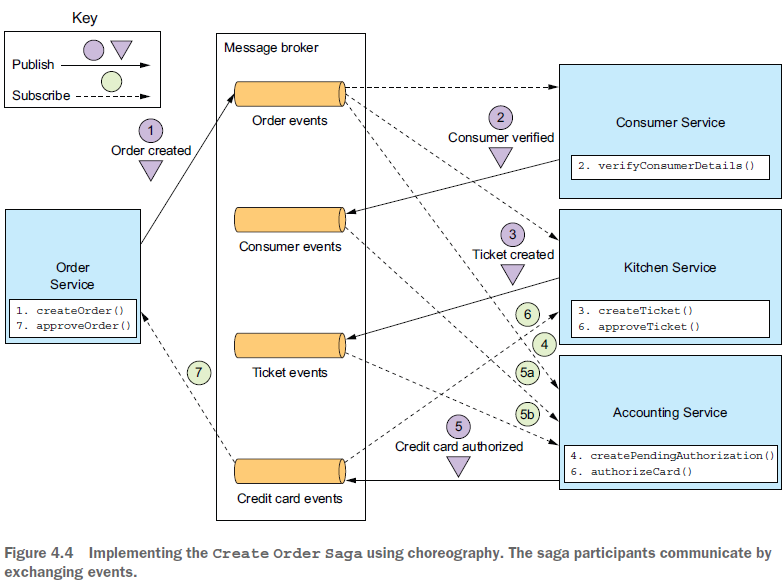
events, charges the consumer’s credit card, and publishes a Credit Card Authorization Failed event.

6 Kitchen Service consumes the Credit Card Authorization Failed event and changes the state of the Ticket to REJECTED.

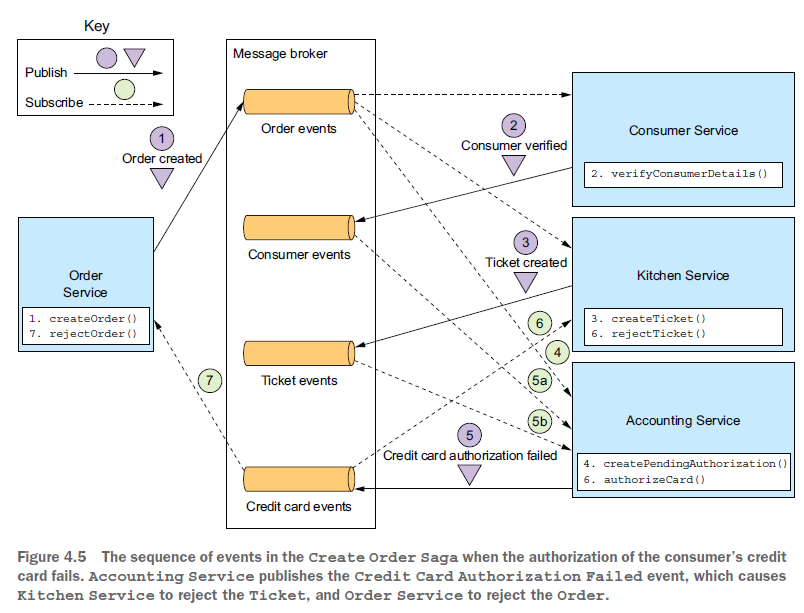
7 Order Service consumes the Credit Card Authorization Failed event and

changes the state of the Order to REJECTED.

The figure below shows the design of the choreography-based version of the Create Order Saga. The participants communicate by exchanging events**.**



And below is the sequence for when the credit card authorization fails:



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 communication for your sagas:

**RELIABLE EVENT-BASED COMMUNICATION**

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

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

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

For example, in the Create Order Saga, Kitchen Service receives a Consumer Verified event, creates a Ticket, and publishes a Ticket Created event.

It’s essential that the database update and the publishing of the event happen atomically. Consequently, to communicate reliably, the saga participants must use transactional messaging, described in chapter 3.

* 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 Order Service receives a Credit Card Authorized event, it must be able to look up the corresponding Order. **The solution is for a saga participant to publish 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 use the orderId as a

correlation ID that’s passed from one participant to the next. Accounting Service publishes

a Credit Card Authorized event containing the orderId from the Ticket-Created event. When Order Service receives a Credit Card Authorized event, it uses the orderId to retrieve the corresponding Order. Similarly, Kitchen Service uses the orderId from that event to retrieve the corresponding Ticket.

**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 of each other.

And there are some drawbacks:

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

 *Cyclic dependencies between the services*—The saga participants subscribe to each other’s events, which often creates cyclic dependencies. For example, if you carefully examine figure 4.4, you’ll see that there are cyclic dependencies, such as Order Service  Accounting Service  Order Service. Although this isn’t necessarily a problem, cyclic dependencies are considered a design smell.**(Why is it a smell here?)**

 *Risk of tight coupling*—Each saga participant needs to subscribe to all events that affect them. For example, **Accounting Service 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 Order Service**.

Choreography can work well for simple sagas, but because of these drawbacks it’s often better for more complex sagas to use orchestration. Let’s look at how orchestration works.

### 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 participant has performed the operation, it sends a reply message to the orchestrator. The orchestrator then processes the message and determines which saga step to perform next.

**IMPLEMENTING THE CREATE ORDER SAGA USING ORCHESTRATION**

Figure 4.6 shows the design of the orchestration-based version of the Create Order 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 messages to saga participants, such as Kitchen Service and Consumer Service. The CreateOrderSaga class 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 a Create Order Saga orchestrator. After that,

the flow for the happy path is as follows:

1 The saga orchestrator sends a Verify Consumer command to Consumer Service.

2 Consumer Service replies with a Consumer Verified message.

3 The saga orchestrator sends a Create Ticket command to Kitchen Service.

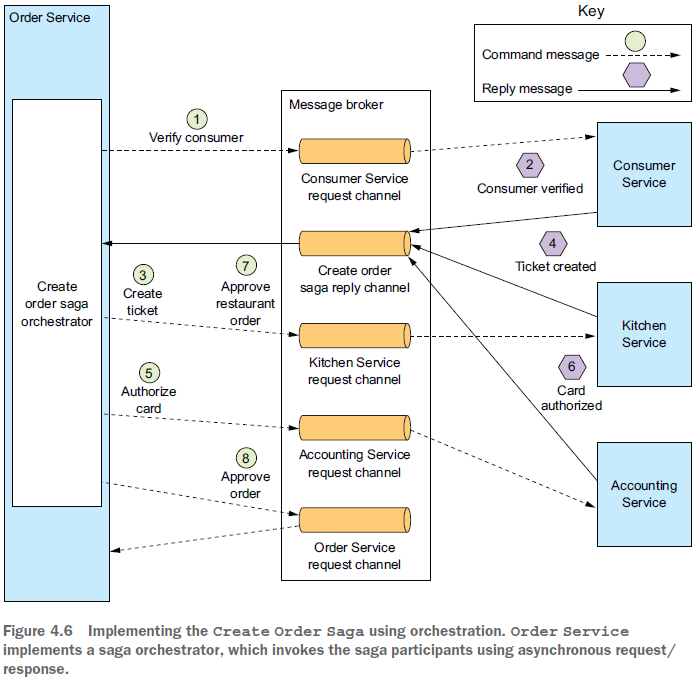
4 Kitchen Service replies with a Ticket Created message.

5 The saga orchestrator sends an Authorize Card message to Accounting Service.

6 Accounting Service replies with a Card Authorized message.

7 The saga orchestrator sends an Approve Ticket command to Kitchen Service.

8 The saga orchestrator sends an Approve Order command to Order Service.



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

I add : *I think the reason behind this for example could be that when publishing to the broker, each instance of the Order Service then can process the message maybe as opposed to the orchestrator itself doing that every time.*

Diagrams such as figure 4.6 each depict one scenario for a 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 Consumer

Service, Kitchen Service, or Accounting Service. **It’s useful, therefore, 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 transitions 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**.(We could know more about these testing strategies in the future though)**

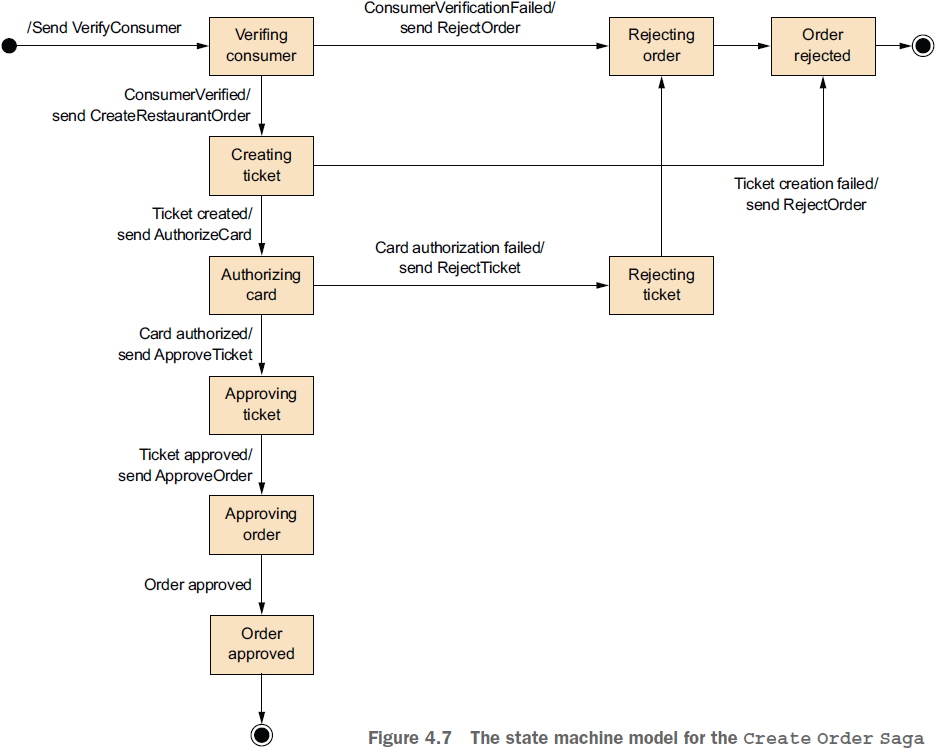


Figure 4.7 shows the state machine model for the Create Order Saga. This state machine consists of numerous states, including the following:

 Verifying Consumer—The initial state. When in this state, the saga is waiting for the Consumer Service to verify that the consumer can place the order.

 Creating Ticket—The saga is waiting for a reply to the Create Ticket command.

 Authorizing Card—Waiting for Accounting Service to authorize the consumer’s credit card.

 Order Approved—A final state indicating that the saga completed successfully.

 Order Rejected—A final state indicating that the Order was rejected by one of the participants.

The state machine also defines numerous state transitions. For example, the state machine transitions from the Creating Ticket state to either the Authorizing Card or the Rejected Order state. It transitions to the Authorizing Card state when it receives a successful reply to the Create Ticket command. Alternatively, if Kitchen Service couldn’t create the Ticket, the state machine transitions to the Rejected Order state.

The state machine’s initial action is to send the VerifyConsumer command to Consumer Service. The response from Consumer Service triggers the next state transition.

If the consumer was successfully verified, the saga creates the Ticket and transitions to the Creating Ticket state. But if the consumer verification failed, the saga rejects the Order and transitions to the Rejecting Order state.

The state machine undergoes numerous other state transitions, driven by the responses from saga participants, until it reaches a final state of either Order Approved or Order Rejected.

**SAGA ORCHESTRATION AND TRANSACTIONAL MESSAGING**