Pattern: Saga

Context

You have applied the [Database per Service](https://microservices.io/patterns/data/database-per-service.html) pattern. Each service has its own database. Some business transactions, however, span multiple service so you need a mechanism to ensure data consistency across services. For example, lets imagine that you are building an e-commerce store where customers have a credit limit. The application must ensure that a new order will not exceed the customer’s credit limit. Since Orders and Customers are in different databases the application cannot simply use a local ACID transaction.

Problem

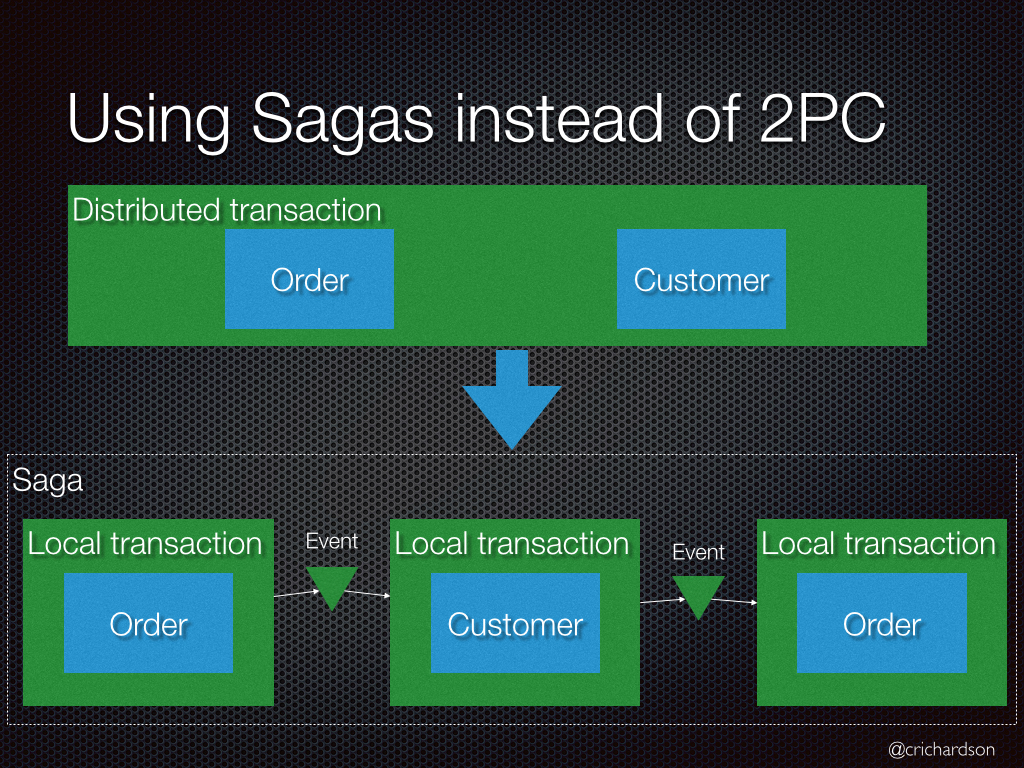
How to maintain data consistency across services?

Forces

* 2PC is not an option

Solution

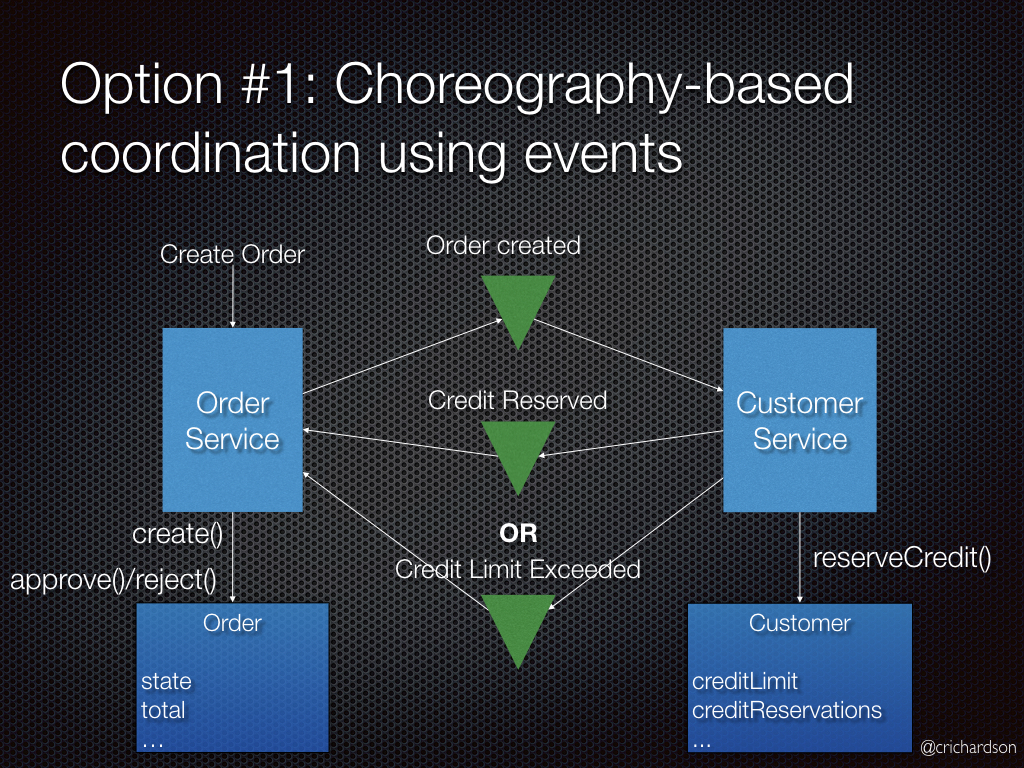
Implement each business transaction that spans multiple services as a saga. A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.



There are two ways of coordination sagas:

* Choreography - each local transaction publishes domain events that trigger local transactions in other services
* Orchestration - an orchestrator (object) tells the participants what local transactions to execute

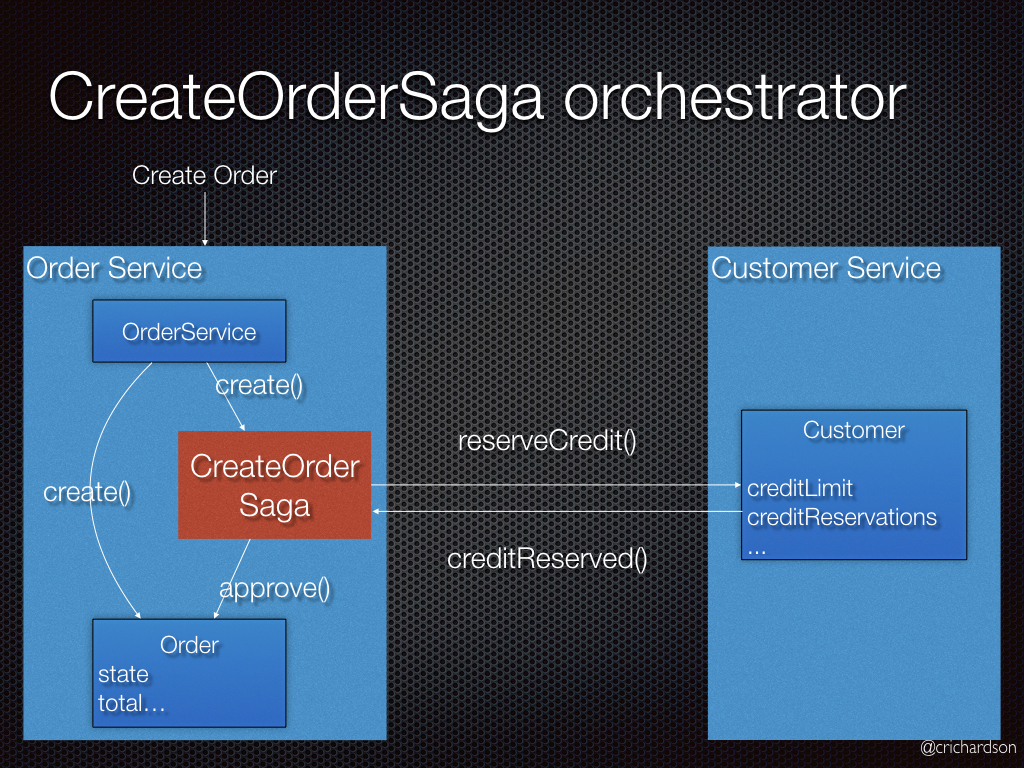
Example: Choreography-based saga



An e-commerce application that uses this approach would create an order using a choreography-based saga that consists of the following steps:

1. The Order Service creates an Order in a *pending* state and publishes an OrderCreated event
2. The Customer Service receives the event attempts to reserve credit for that Order. It publishes either a Credit Reserved event or a CreditLimitExceeded event.
3. The Order Service receives the event and changes the state of the order to either *approved* or *cancelled*

Example: Orchestration-based saga



An e-commerce application that uses this approach would create an order using an orchestration-based saga that consists of the following steps:

1. The Order Service creates an Order in a *pending* state and creates a CreateOrderSaga
2. The CreateOrderSaga sends a ReserveCredit command to the Customer Service
3. The Customer Service attempts to reserve credit for that Order and sends back a reply
4. The CreateOrderSaga receives the reply and sends either an ApproveOrder or RejectOrder command to the Order Service
5. The Order Service changes the state of the order to either *approved* or *cancelled*

Resulting context

This pattern has the following benefits:

* It enables an application to maintain data consistency across multiple services without using distributed transactions

This solution has the following drawbacks:

* The programming model is more complex. For example, a developer must design compensating transactions that explicitly undo changes made earlier in a saga.

There are also the following issues to address:

* In order to be reliable, a service must atomically update its database *and* publish a message/event. It cannot use the traditional mechanism of a distributed transaction that spans the database and the message broker. Instead, it must use one of the patterns listed below