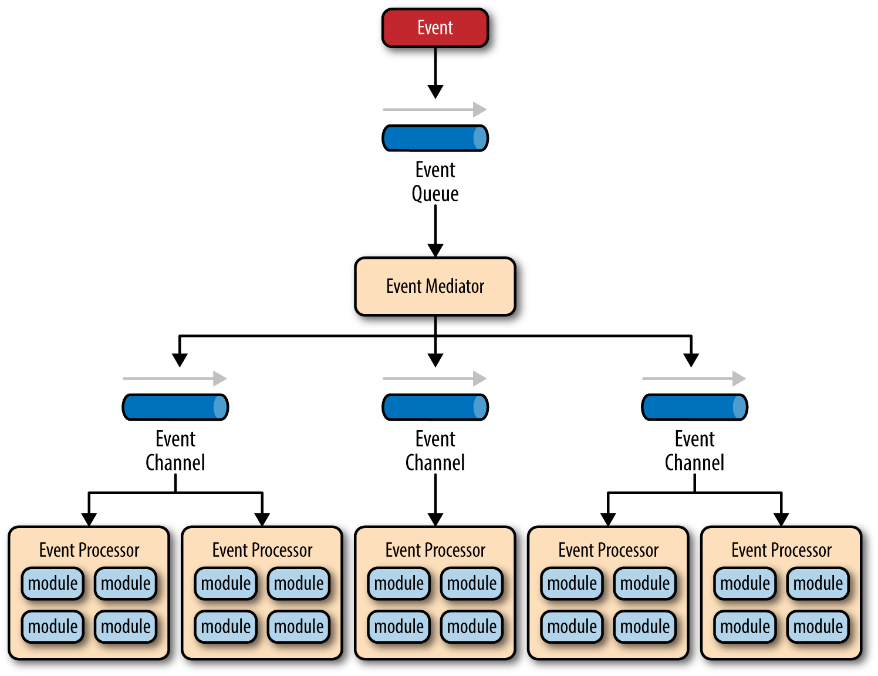
**Event Driven Architecture:**

The event-driven architecture pattern is a popular distributed asynchronous architecture pattern used to produce highly scalable applications. **The event-driven architecture pattern consists of two main topologies, the mediator and the broker.** The mediator topology is commonly used when you need to orchestrate multiple steps within an event through a central mediator, whereas the broker topology is used when you want to chain events together without the use of a central mediator. Because the architecture characteristics and implementation strategies differ between these two topologies, it is important to understand each one to know which is best suited for your particular situation.

**Mediator Topology**

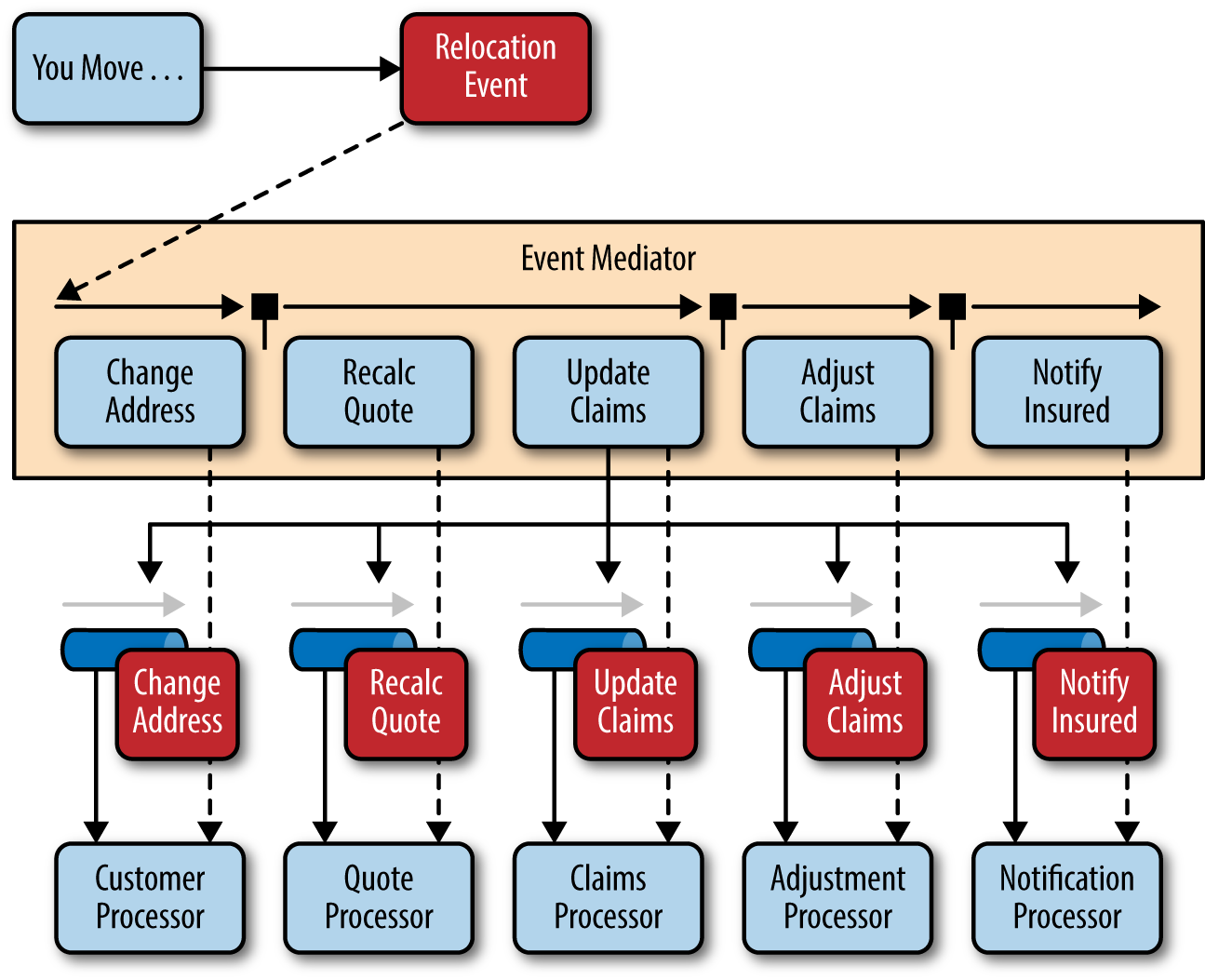
There are four main types of architecture components within the mediator topology: event queues, an event mediator, event channels, and event processors. The event flow starts with a client sending an event to an event queue, which is used to transport the event to the event mediator. The event mediator receives the initial event and orchestrates that event by sending additional asynchronous events to event channels to execute each step of the process. Event processors, which listen on the event channels, receive the event from the event mediator and execute specific business logic to process the event. The simplest and most common implementation of the event mediator is through open source integration hubs such as Spring Integration, Apache Camel, or Mule ESB



**Broker Topology**

The broker topology differs from the mediator topology in that there is no central event mediator; rather, the message flow is distributed across the event processor components in a chain-like fashion through a lightweight message broker (e.g., ActiveMQ, HornetQ, etc.). This topology is useful when you have a relatively simple event processing flow and you do not want (or need) central event orchestration.

There are two main types of architecture components within the broker topology: a broker component and an event processor component. The broker component can be centralized or federated and contains all of the event channels that are used within the event flow. The event channels contained within the broker component can be message queues, message topics, or a combination of both.



**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**

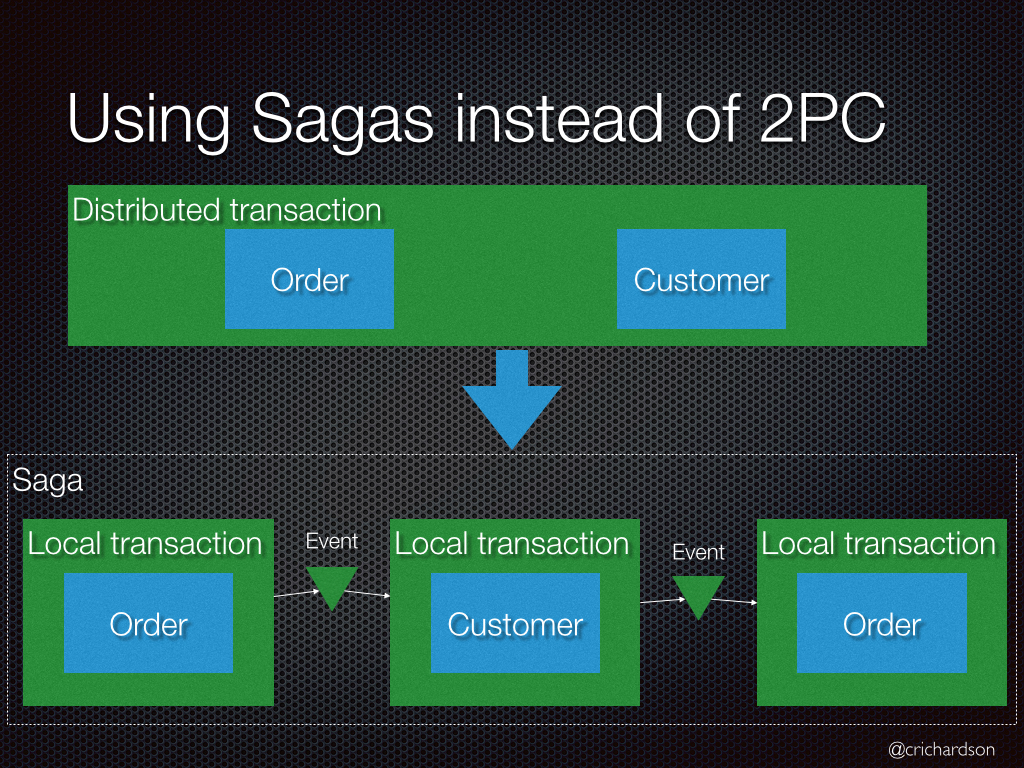
How to maintain data consistency across services?

**Forces**

* 2PC is not an option

**Solution**

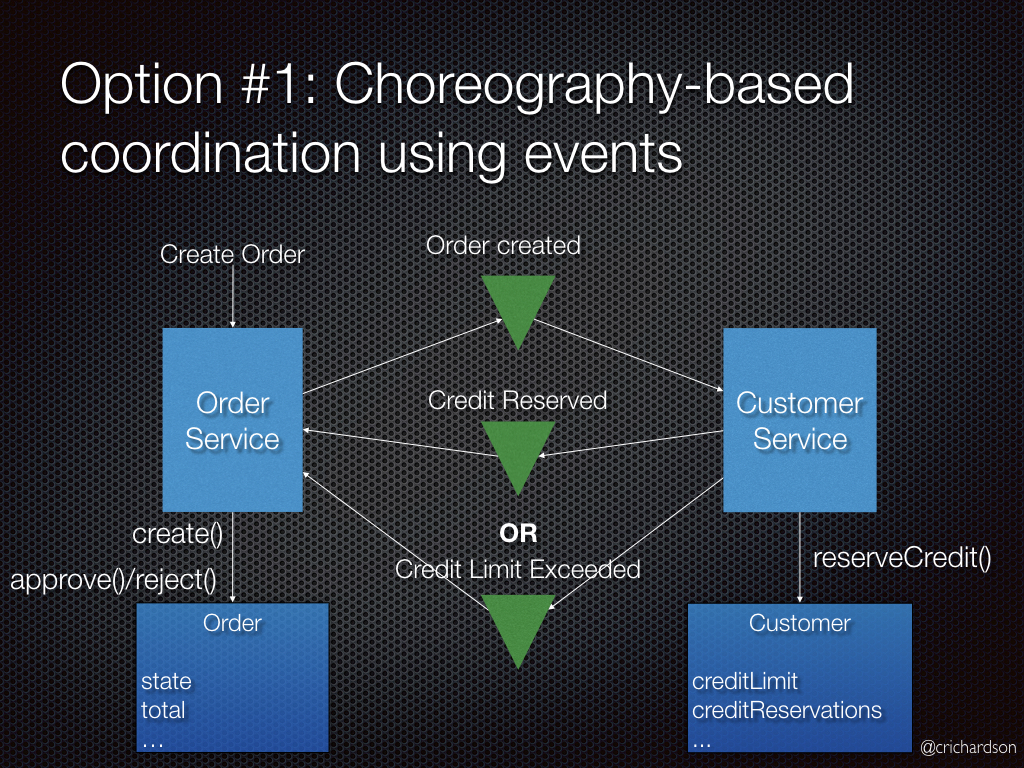
Implement each business transaction that spans multiple services as a saga. A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business 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

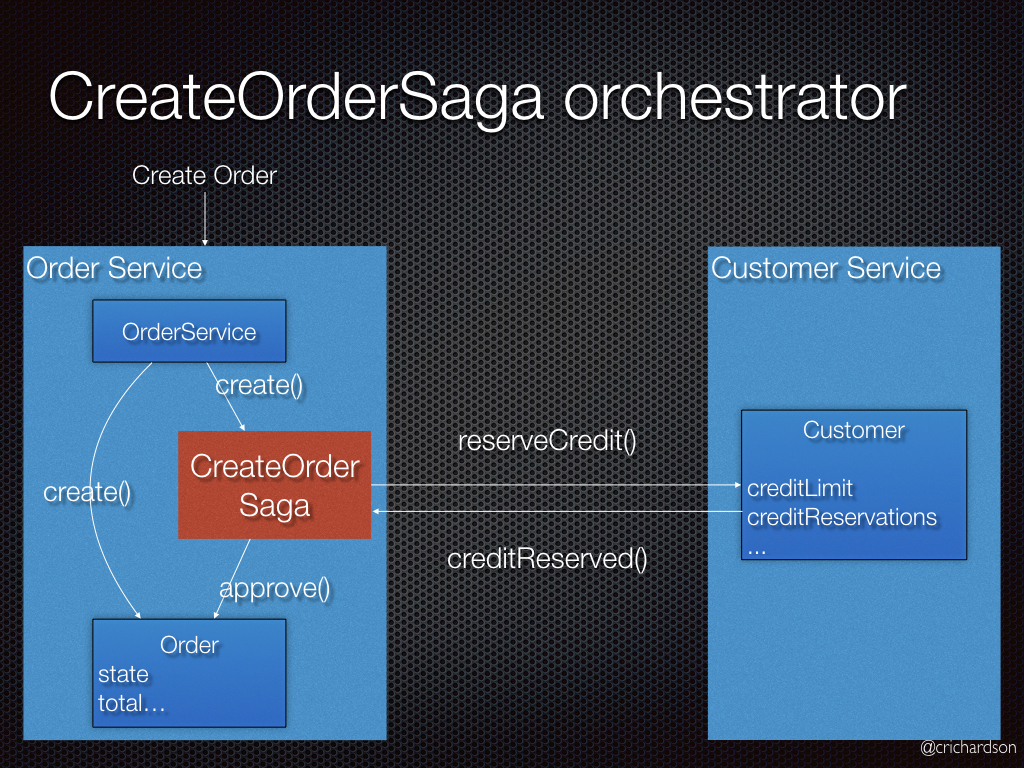
**Example: Choreography-based saga**



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 an 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.

**Related patterns**

* The [Database per Service pattern](https://microservices.io/patterns/data/database-per-service.html) creates the need for this pattern
* The following patterns are ways to *atomically* update state *and* publish events:
  + [Event sourcing](https://microservices.io/patterns/data/event-sourcing.html)
  + [Application events](https://microservices.io/patterns/data/application-events.html)
  + [Database triggers](https://microservices.io/patterns/data/database-triggers.html)
  + [Transaction log tailing](https://microservices.io/patterns/data/transaction-log-tailing.html)

Feign aims at simplifying HTTP API clients. Simply put, the developer needs only to declare and annotate an interface while the actual implementation will be provisioned at runtime.