Communication Between Microservices

@zmerta

Types of Communication

- Synchronous
- Asynchronous

In synchronous communication,
the service consumer makes a request
and waits until the operation completes
and response is received.

Synchronous Communication

- SOAP
- REST
- RPC

Synchronous Communication

Pros

- Simple
- The result is immediately available
- Well-known

Cons

- High coupling
- Calling service may be impacted by errors in the called service
- Blocking
- Doesn’t scale well

In asynchronous communication
the calling service does not wait
for a response from the called service.

Asynchronous Communication

- Notifications
- Request/async response
- Message-based

Asynchronous Communication

Pros

- Loose coupling
- Non-blocking
- Highly scalable

Cons

- The result is not immediately available
- Difficult error handling
- Needs infrastructure

In microservice architecture we aim for the autonomy of the microservices.

Try to avoid synchronous communication between your microservices.

Use asynchronous message-based communication.

Asynchronous communication provides temporal decoupling while increasing robustness.

But the asynchronous message-based communication is difficult.

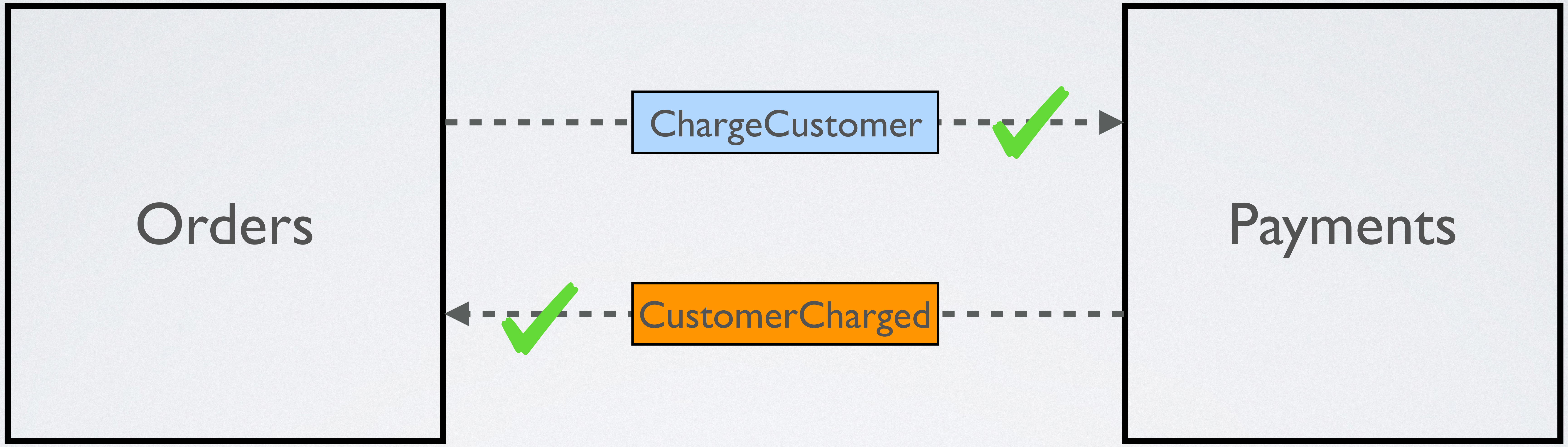
We have to deal with a lot of challenges.

Communication Challenges

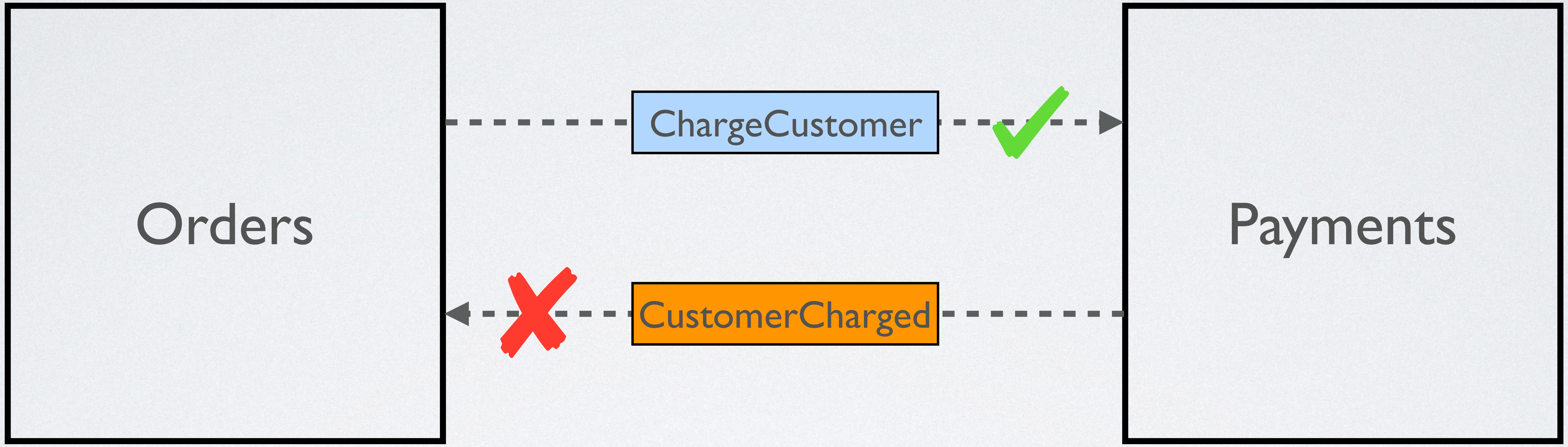
- Lost messages
- Duplicate messages
- Participants failing and losing state
- Error conditions
- Concurrent actions

Let’s do a simple exercise…

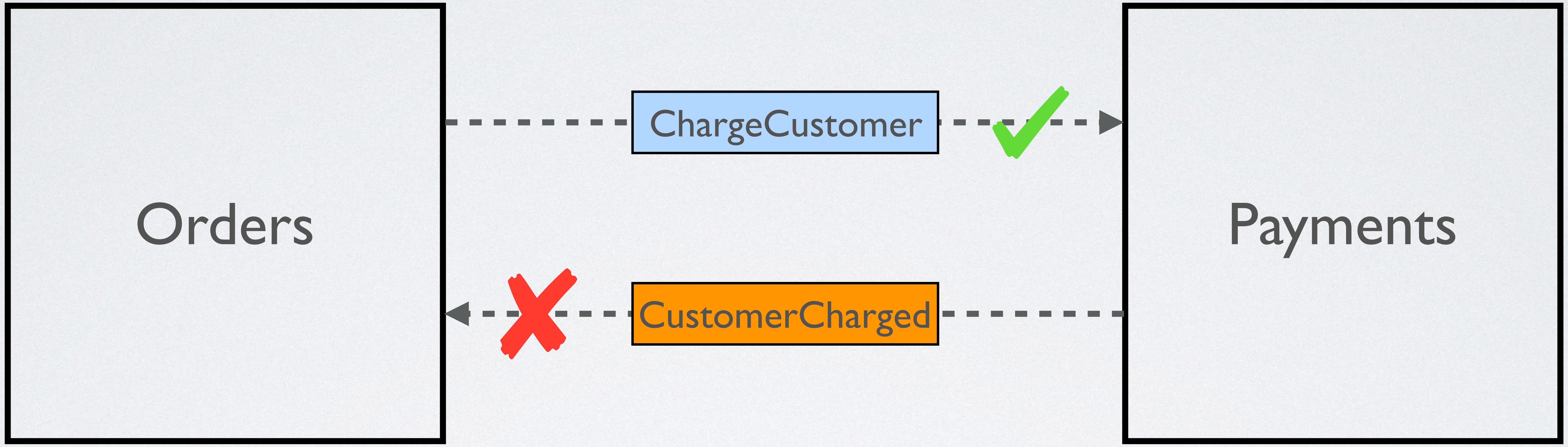
Happy Path



What if the **CustomerCharged** message **never arrives**?



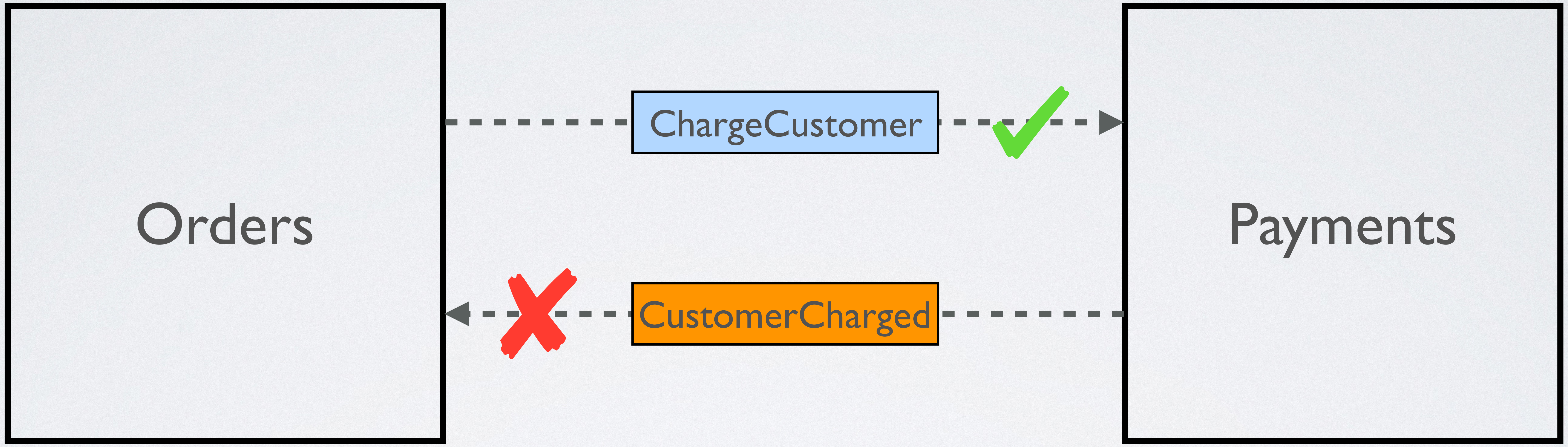
What if the **CustomerCharged** message **never arrives**?



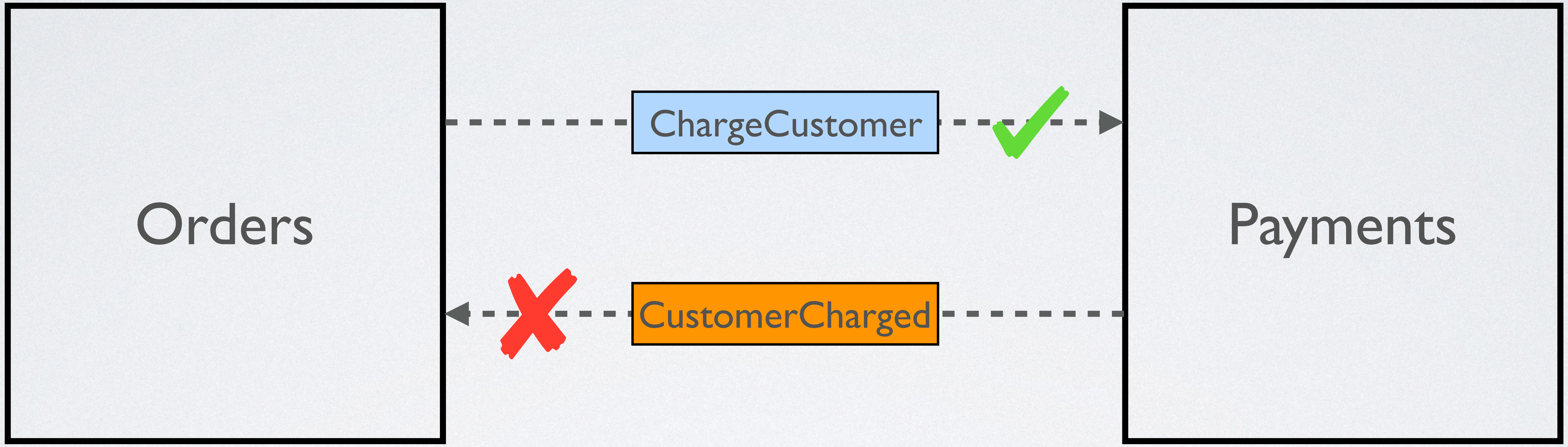
The Orders service can **Give it up**

or **Resend** the **ChargeCustomer** message.

What if the Orders service will **Give it up** but the customer was already charged?

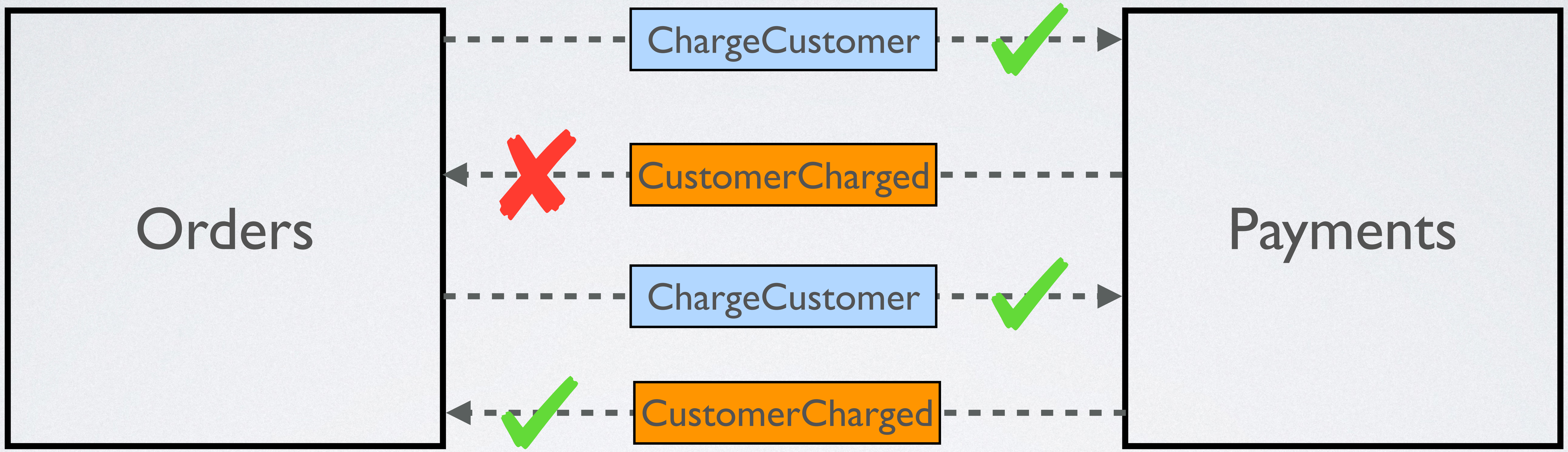


What if the Orders service will **Give it up** but the customer was already charged?

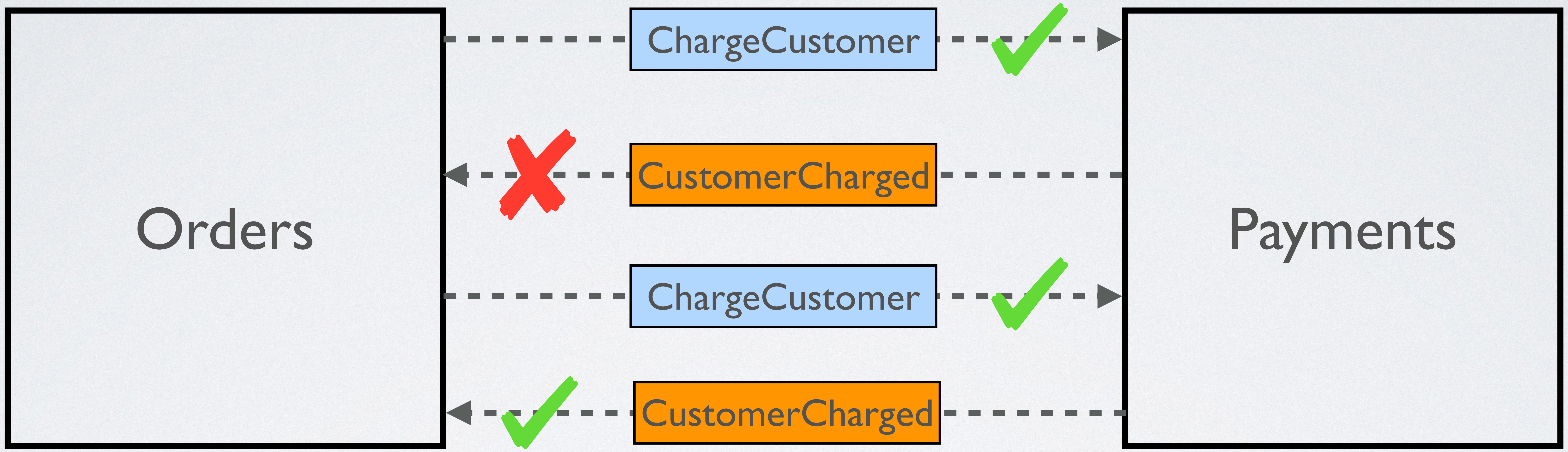


Yes, easy money :-) See you in a prison ;-)
But seriously, we have to somehow deal with it.

What if the Orders service will **Resend** the **ChargeCustomer** message and the customer has been charged twice?

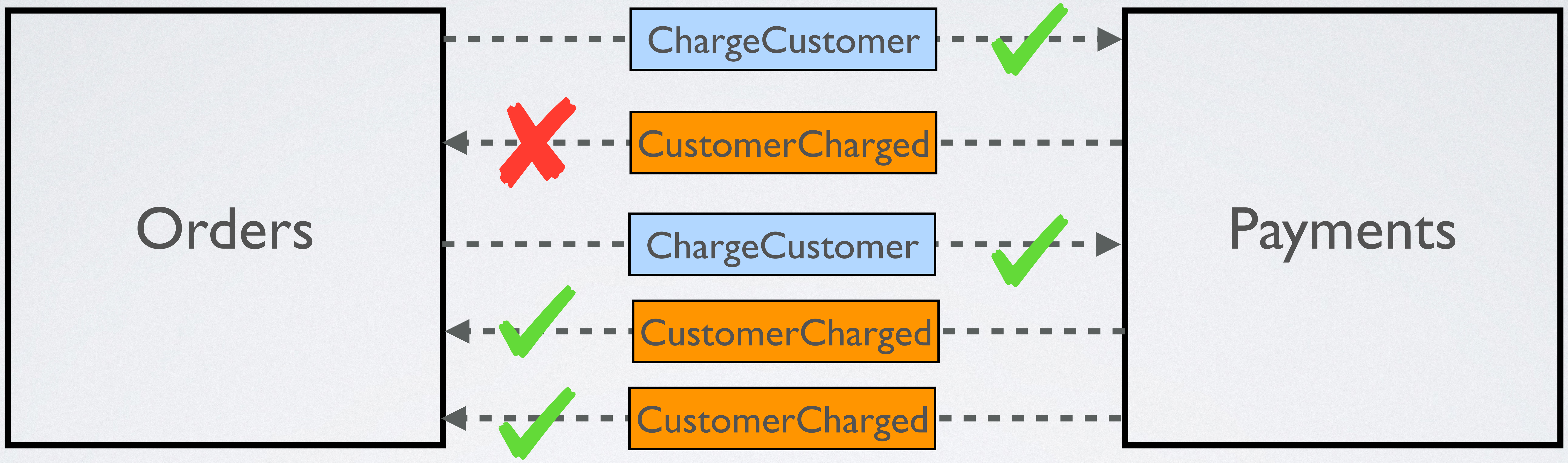


What if the Orders service will **Resend** the **ChargeCustomer** message and the customer has been charged twice?

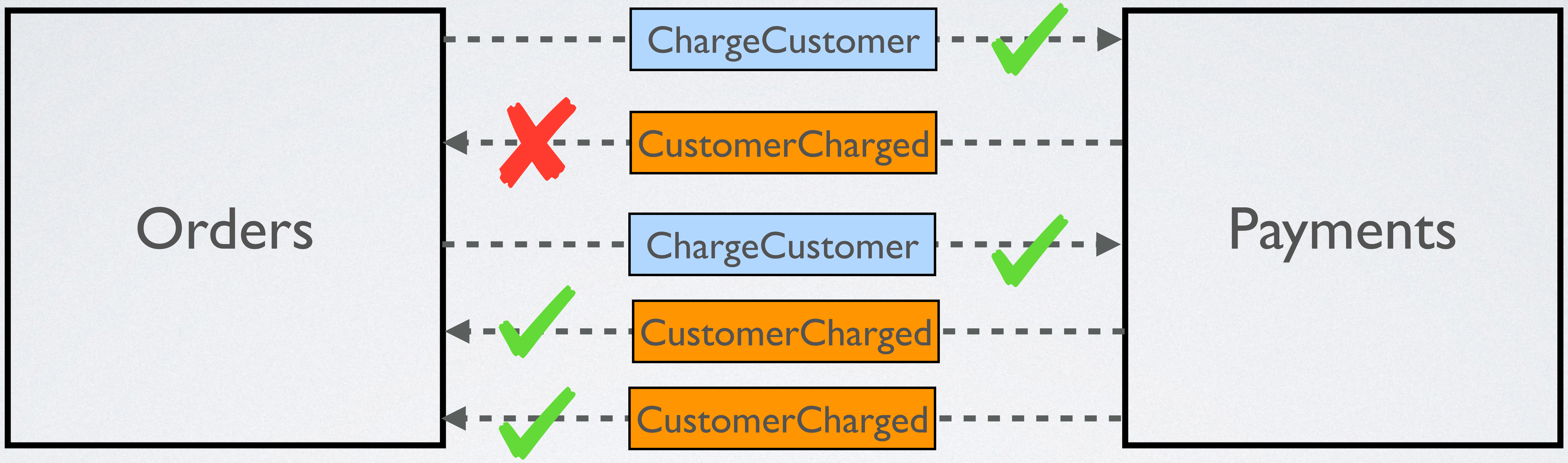


We need an **Idempotent Receiver**.

What if the Orders service will **Resend** the **ChargeCustomer** message and then the **CustomerCharged** message will arrive twice?



What if the Orders service will **Resend** the **ChargeCustomer** message and then the **CustomerCharged** message will arrive twice?



We need an **Idempotent Sender**.

The communication quickly become quite complex.

The two microservices are engaged in a conversation.

CONVERSATIONS

A conversation is an exchange of related messages over time.

DESCRIBING CONVERSATIONS

Conversation Policy

- Participant roles
- Message types
- Protocol

Participant Roles

- Define who is involved in the conversation
- Each participant could be in one or more roles

Message Types

- Define different kinds of messages and how they impact the conversation
- All participants must clearly understand the intent of the messages

Protocol

- Defines "legal" message exchanges between participants

MESSAGE TYPES IN CONVERSATIONS

Message Types in Conversations

- Command
- Event

Command

- Captures the intent
- Single receiver
- Models personal communication

Event

- Informs about something that happened in the past
- Intentless
- Multiple receivers
- Models broadcast

CONVERSATION STATE

Conversations are Stateful

- Global conversation state
- Participant state

CONVERSATION TYPES

Conversation Types

- Static Conversation
- Dynamic Conversation

CONVERSATION CONSISTENCY

Achieving a conversation consistency could be difficult in a Microservice architecture.

BALANCE THESE CONSIDERATIONS

- Reducing uncertainty
- Detecting errors
- Mitigating risk
- Optimistic vs. Pessimistic
- Idempotency
- Certainty vs. Complexity
- Layered Protocols

Reducing Uncertainty

- Think about how to reduce the uncertainty

- Is it even possible?

- Ex. If a consumer does not receive a response to a request, it cannot be certain whether the provider processed the request or not

Detecting Errors

- It could be difficult due to the inherent uncertainty
- Ex. How does one detect that a letter sent with regular mail did not arrive?

Mitigating Risk

- Participants have to accept that consistency can’t be achieved in all cases
- The conversation should minimize the probability of inconsistency

Optimistic vs Pessimistic

Optimistic

- Optimizing for happy-path
- Complex failure scenarios
- High cost

Pessimistic

- Minimizing the frequency and severity of failure scenarios

Idempotency

- Simple strategy - retry operation
- Avoid duplicate execution
- Typically using correlation identifier

Certainty vs Complexity

- More complex conversation can increase certainty but also complexity

LAYERED PROTOCOLS

- Lower protocol levels could help to deal with some failure scenarios, so the application layer doesn not have to worry about this

- Ex. Reliable messaging could help with
  - Retrying
  - Idempotency
  - Message Delivery Reliability

CONSISTENCY STRATEGIES

Consistency Strategies

- Ignore Error
- Isolate Error
- Retry
- Compensating Action
- Start Over
- Tentative Operation
- Coordinated Agreement

Ignore Error

- A conversation is optimistic, it consider only the happy path
- Use the strategy when
  - The impact of the error is negligible (financial, reputation, …)
  - Error correction is expansive

Isolate Error

- Ignore the error in the context of the current conversation
- Handle all errors afterwards

Retry

- If the operation doesn't succeed at first, try again
- Only if the retry is meaningful (ex. technical error)

Compensating Action

- If the action fail, use a second action that undoes a prior action to regain a consistent state
- Two types of compensation
  - Perfect Compensation
  - Imperfect Compensation

Start Over

- If you cannot undo an action, revert to the beginning and rebuild the desired state

Tentative Operation

- A conversation has multiple participants and a coordinator
- A coordinator asks participants to execute a tentative operations
- The participants send results of the tentative operations
- According to the results a coordinator will confirm the tentative operations or cancel them

Tentative Operation

- Explicit Cancellation
- Implicit Cancellation

Coordinated Agreement

- A conversation has multiple participants and a coordinator
- A coordinator asks participants to execute the operations or tentative operations
- The participants send results of the operations or tentative operations
- According to the results a coordinator will confirm operations or execute compensating actions and cancel the tentative operations

MITIGATION STRATEGIES

Mitigation Strategies

- Perform most likely to fail action first
- Perform hardest to revert action last

Always ask

- Will it pay off?
- How big is the impact of a potencial problem?
- How often can a problem occur?
- How difficult is to fix the problem?

Documenting Conversations

- Document a Conversation Policy and a Conversation State
- From the point of view of each participant of the conversation
- You can use UML and BPML

CONTROLLING CONVERSATIONS

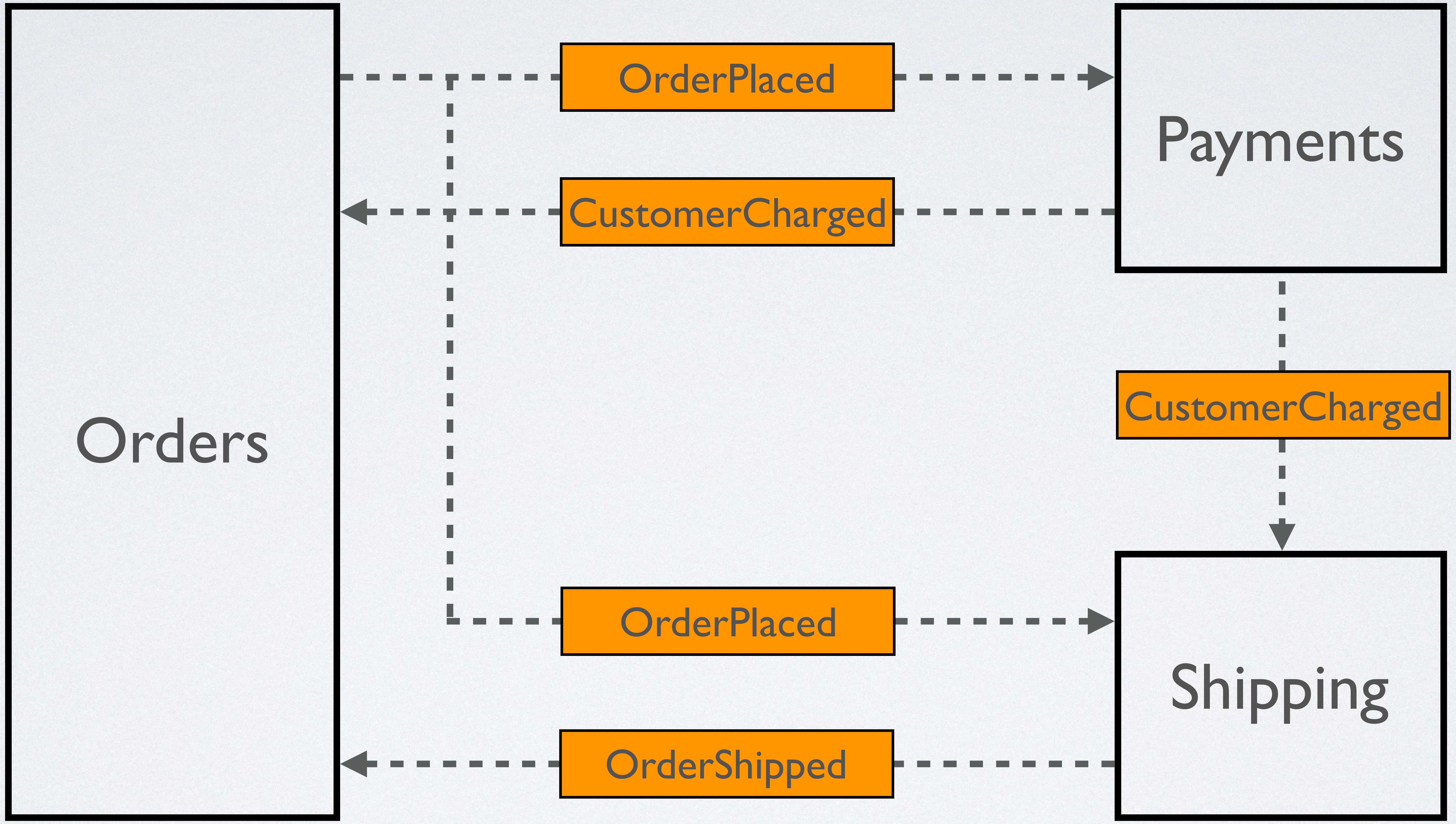
Controlling Styles

- Choreography
- Orchestration

Choreography

- Conversation is autonomous
- Conversation is not explicitly controlled
- All participants are independent, everyone knows what to do
- Conversation is handled using the Event messages

Choreography



Choreography

Pros

- Autonomous
- No central coordinator
- No central point of failure
- Easily changeable

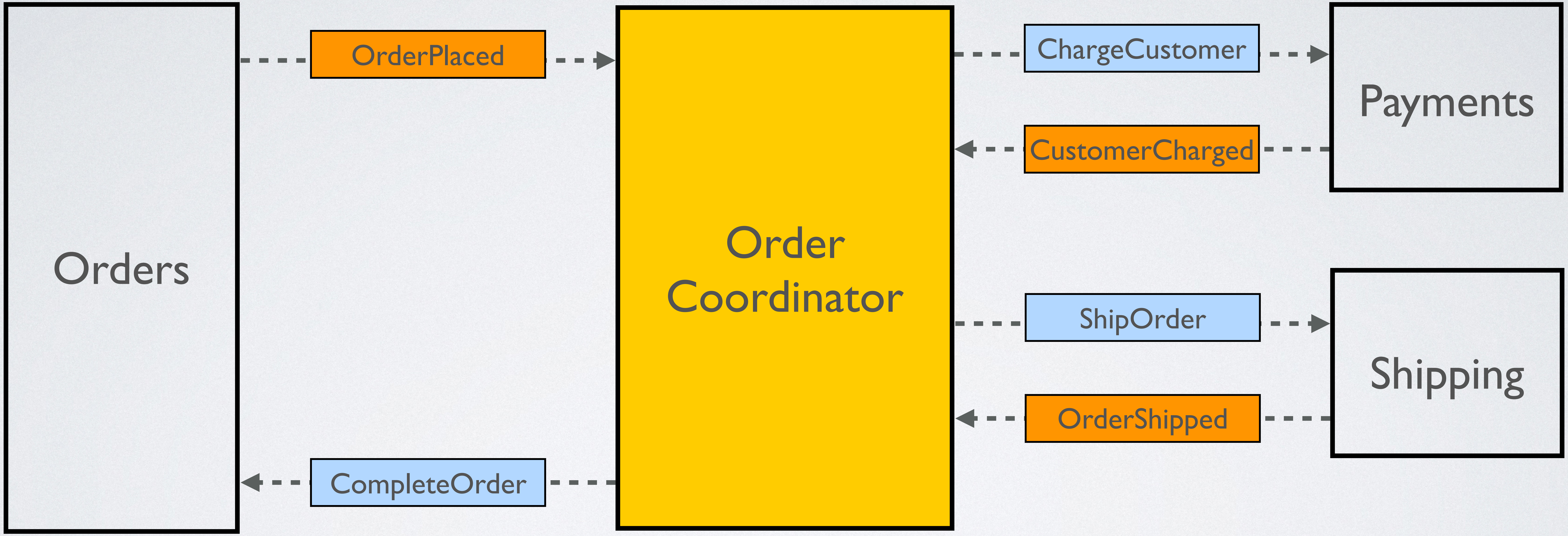
Cons

- A conversation isn’t explicitly visible
- Difficult to monitor

Orchestration

- Conversation is autocratic
- Conversation is controlled by the central coordinator
- Conversation is handled using Command and Event messages

ORCHESTRATION



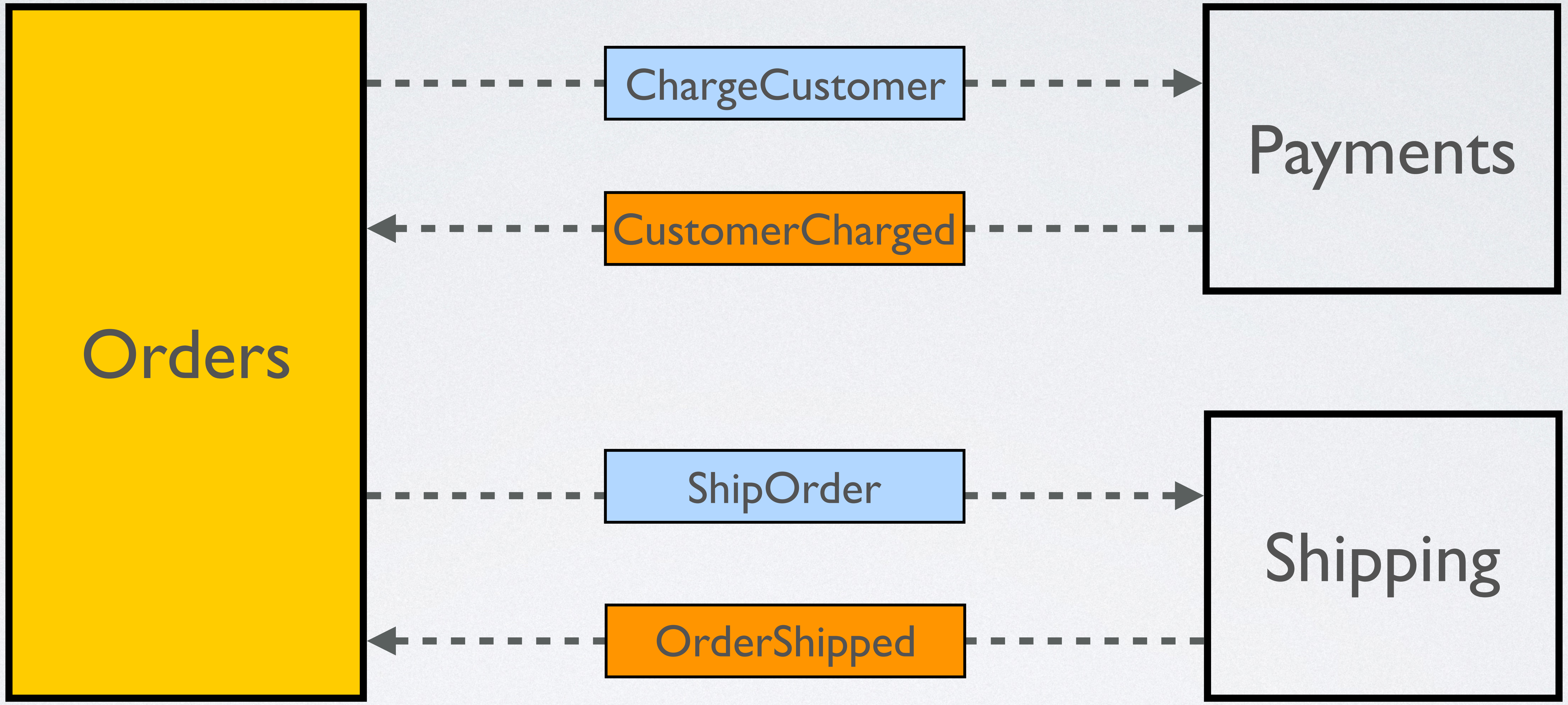
Don’t do this!

Don’t build a central coordinating service.

There is a better way…

Make a coordinator part of an appropriate microservice.

Orchestration



Orchestration

Pros

- A conversation is explicitly visible
- Easy to monitor

Cons

- Lower autonomy
- A central coordinator is often complex
- A single point of failure

SAGA

Sagas (1987)

by

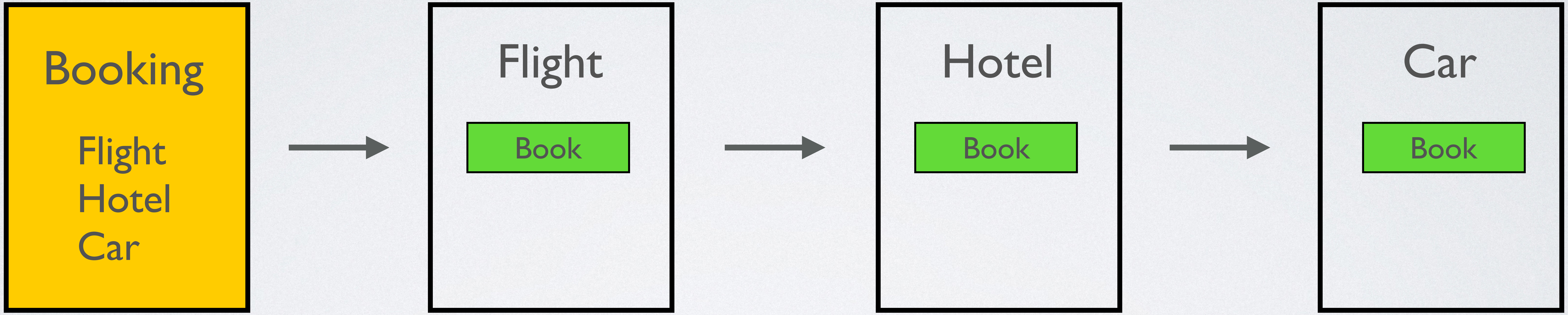
Hector Garcia-Molina and Kenneth Salem

Saga is as a long running transaction divided to a sequence of independent transactions.

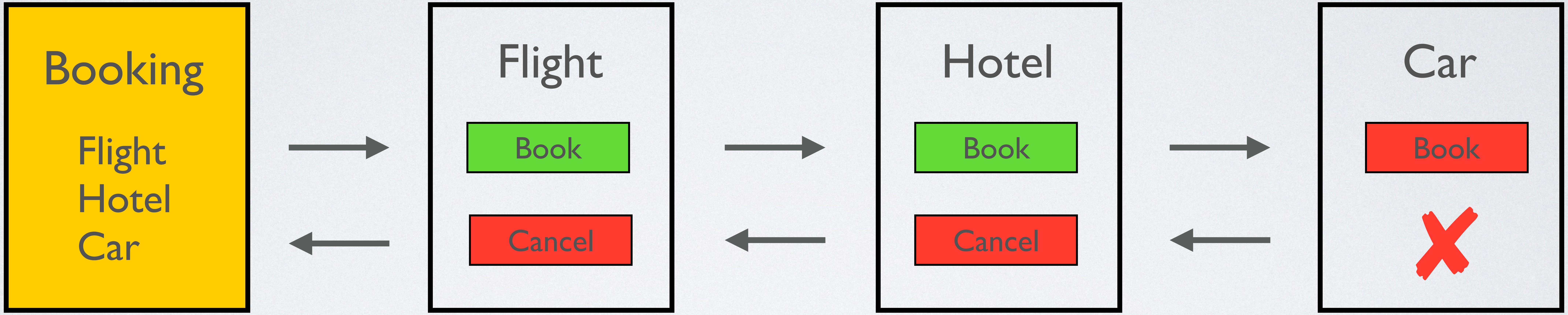
All transactions and their order are known in advance.

If an error occurs, the compensation actions of the completed transactions are performed.

SAGA



SAGA



Booking

Flight
Hotel
Car

Now

Saga is a universal pattern for managing complex business transactions.

Saga types

- Active

  Controls the conversation

- Passive

  Observes the conversation

Implementation Styles

- Orchestration

  Using a Process Manager pattern

- Choreography

  Using a Routing Slip pattern

Existing Tools

- Axon Framework

  Java Domain-Driven Design and CQRS framework

- Camunda

  A platform for workflow and business process management

Saga Example

- Event-driven implementation
- Using Axon Framework

That’s all…