Using sagas to maintain data consistency in a microservice architecture

Chris Richardson

Founder of Eventuate.io

Founder of the original CloudFoundry.com

Author of POJOs in Action

@crichardson

chris@chrisrichardson.net

http://eventuate.io

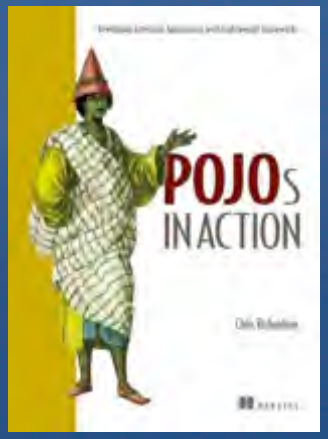
Presentation goal

Distributed data management challenges in a microservice architecture

Sagas as the transaction model

About Chris


COMMUNICATIONS
LISP
OF THE ACM


POJOs
IN ACTION


CLOUDFOUNDRY
The Enterprise Java Cloud
SIGN UP
LEARN MORE
CLOUDFOUNDRY
APPLICATION DEMO


vmware
spring

About Chris

Consultant and trainer focusing on modern application architectures including microservices (http://www.chrisrichardson.net/)

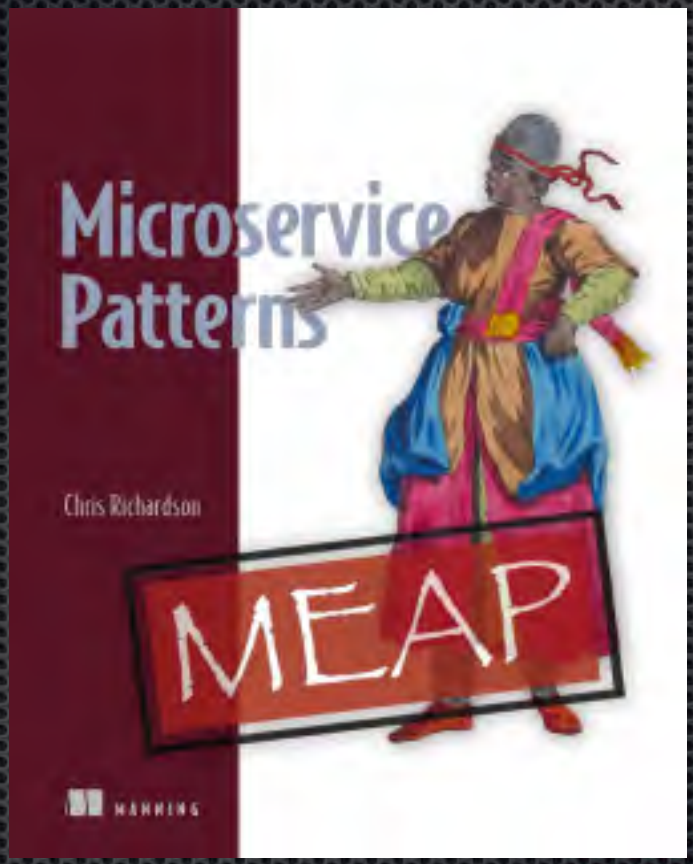
Chris Richardson ArchSummit

About Chris

Founder of a startup that is creating an open-source/SaaS platform that simplifies the development of transactional microservices

(http://eventuate.io)

For more information



http://learnmicroservices.io

Agenda

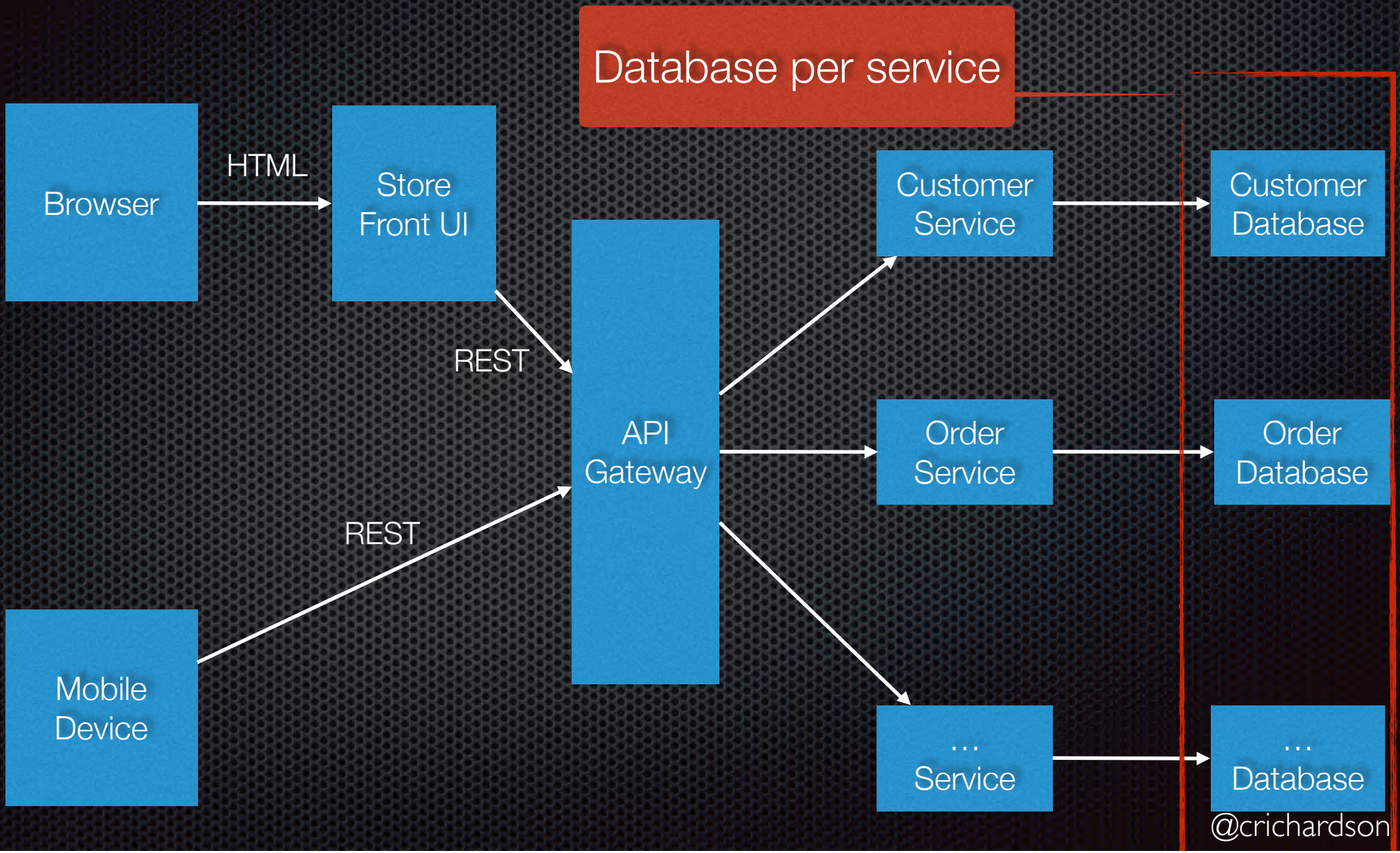
- ACID is not an option
- Overview of sagas
- Coordinating sagas

The microservice architecture
structures
an application as a
set of loosely coupled services

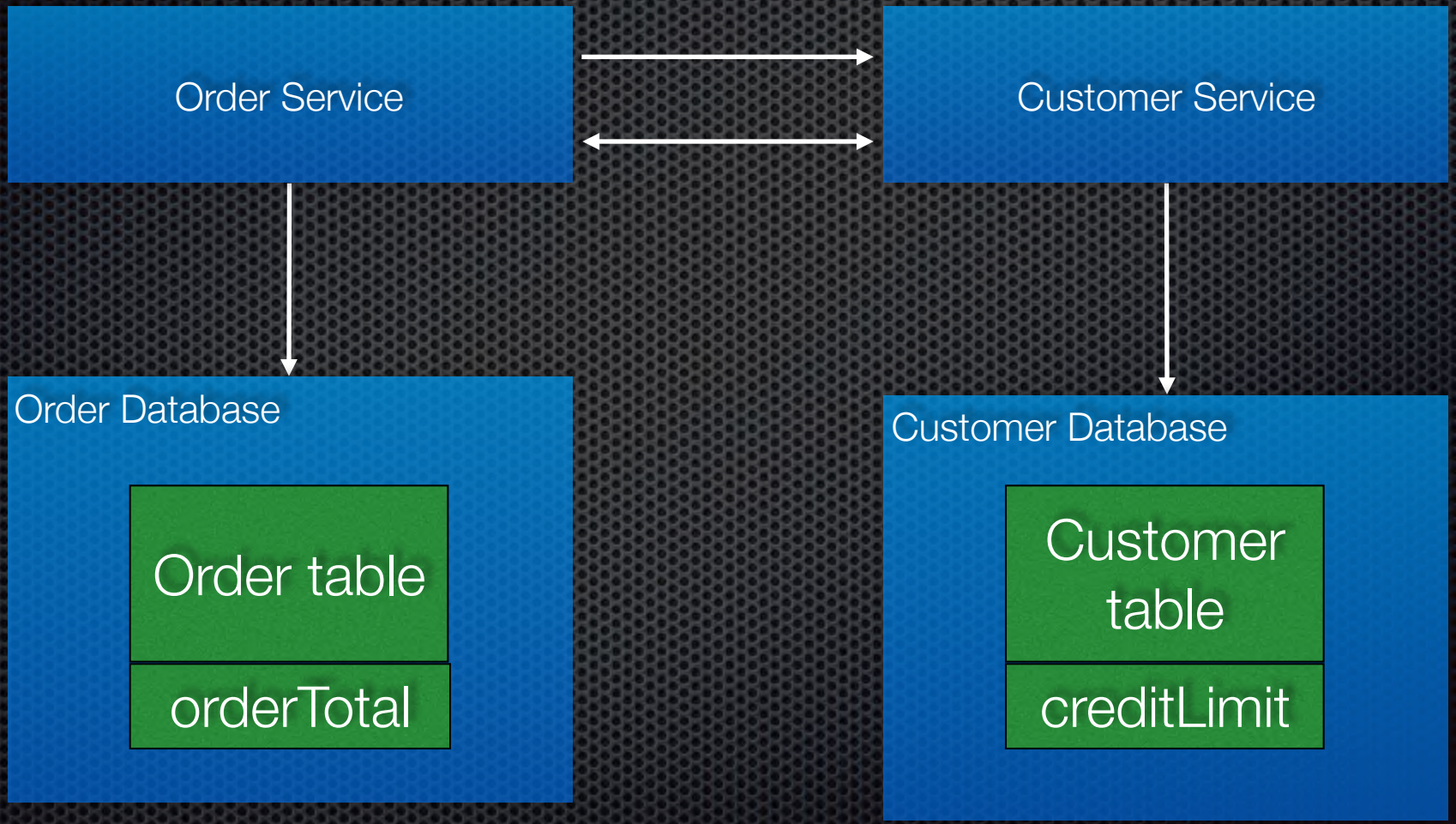
The microservice architecture tackles complexity through modularization

* might improve scalability too

Microservice architecture



Loose coupling = encapsulated data



How to maintain data consistency?!?!?

Invariant:
sum(open order.total) <= customer.creditLimit

Cannot use ACID transactions

Distributed transactions

```
BEGIN TRANSACTION
...
SELECT ORDER_TOTAL
 FROM ORDERS WHERE CUSTOMER_ID = ?
...
SELECT CREDIT_LIMIT
FROM CUSTOMERS WHERE CUSTOMER_ID = ?
...
INSERT INTO ORDERS …
...
COMMIT TRANSACTION
```

Private to the Order Service

Private to the Customer Service

2PC is not an option

- Guarantees consistency

BUT

- 2PC coordinator is a single point of failure
- Chatty: at least O(4n) messages, with retries O(n^2)
- Reduced throughput due to locks
- Not supported by many NoSQL databases (or message brokers)
- CAP theorem ⇒ 2PC impacts availability
- ….

Agenda

- ACID is not an option
- Overview of sagas
- Coordinating sagas

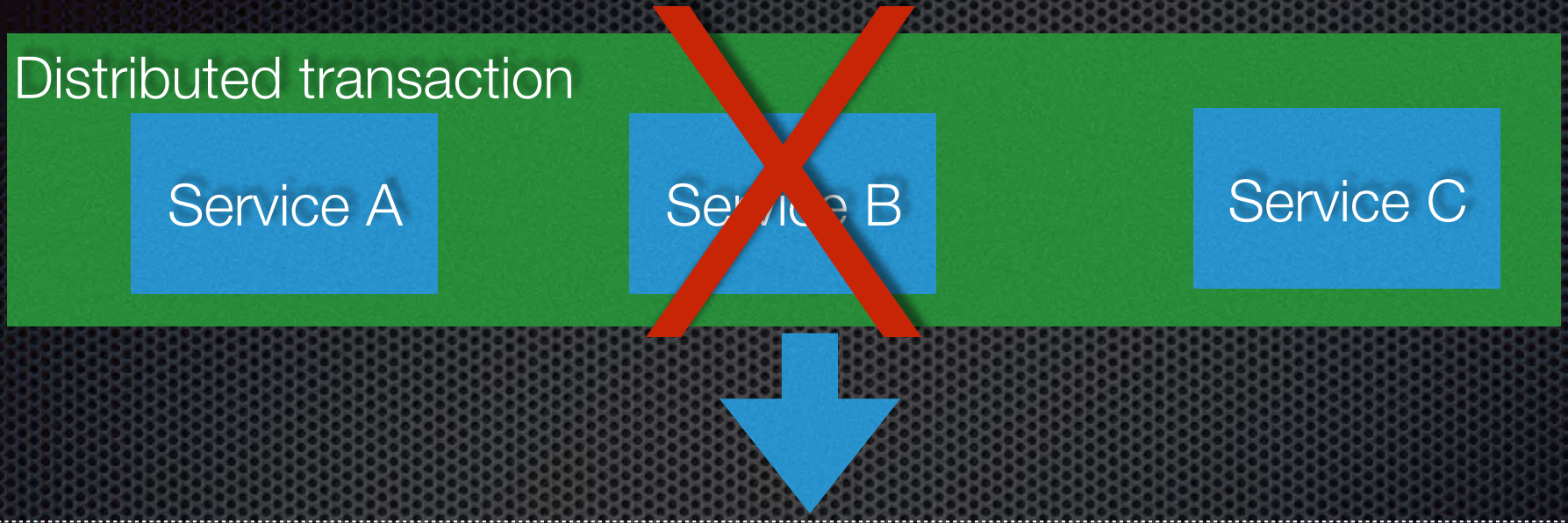
From a 1987 paper

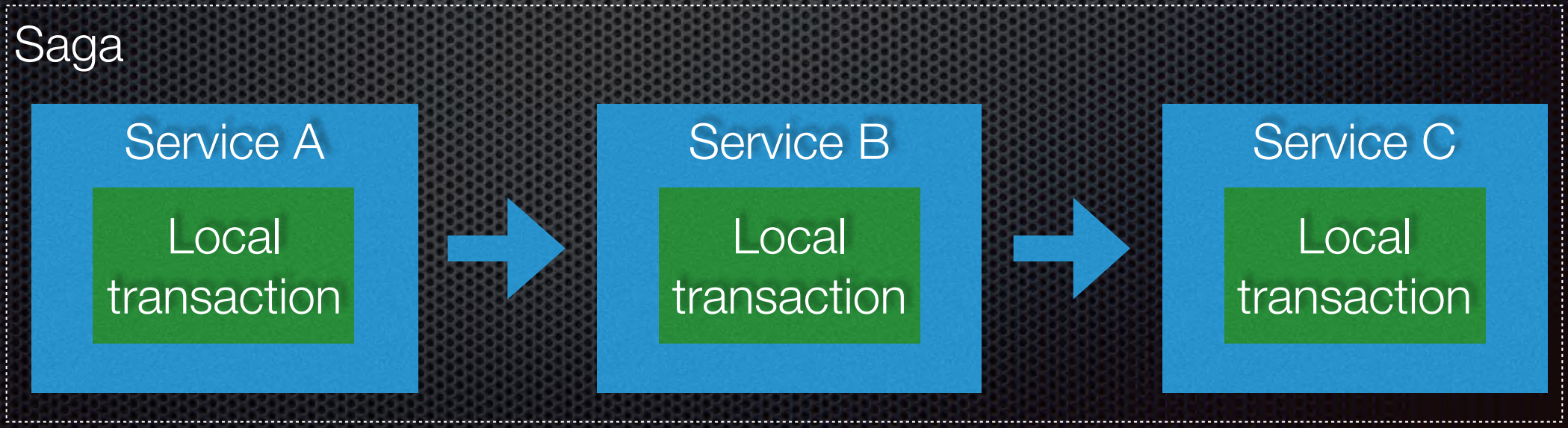
SAGAS

Hector Garcia-Molina
Kenneth Salem

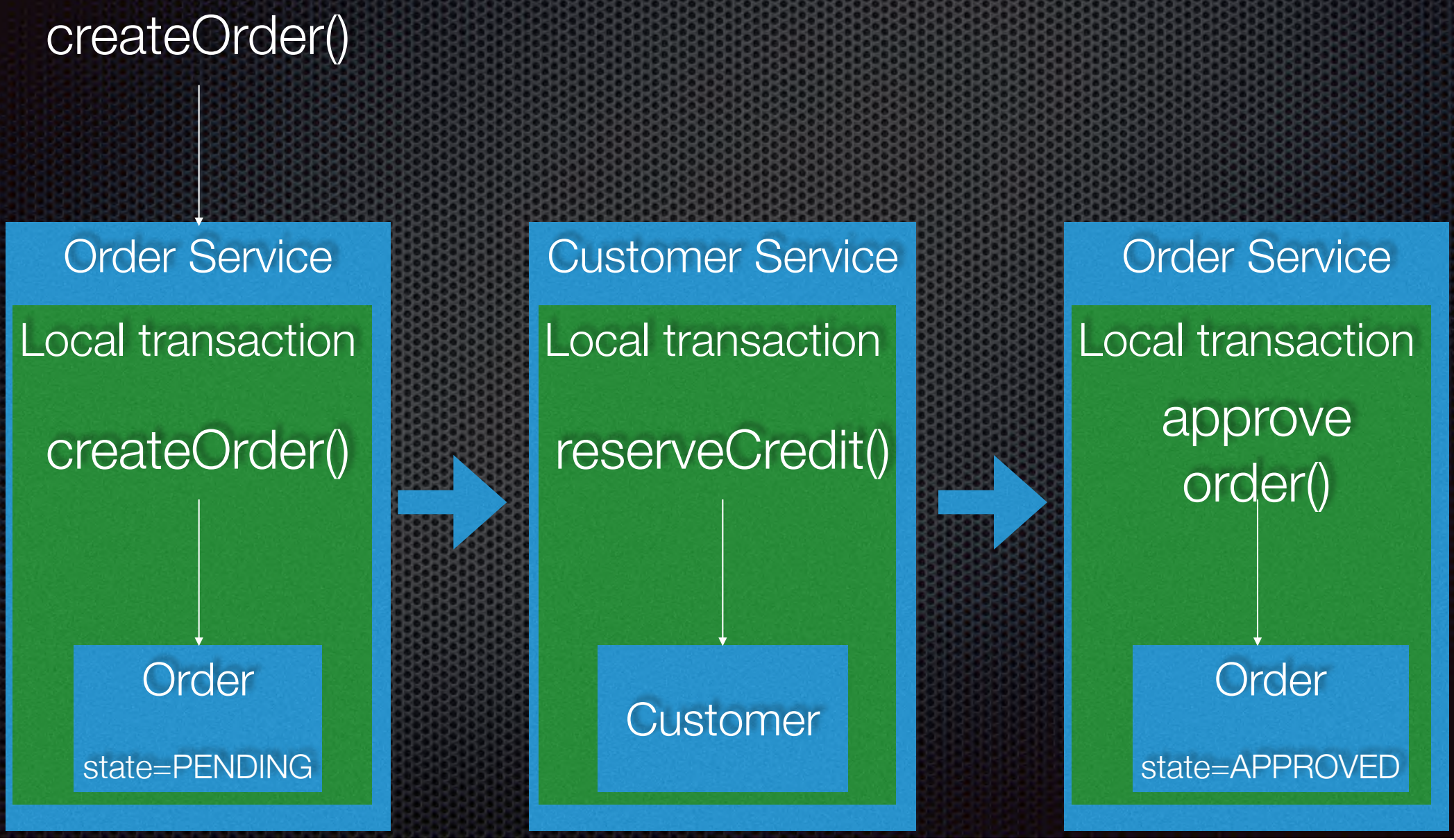
Department of Computer Science
Princeton University
Princeton, N J 08544

Use Sagas instead of 2PC





Create Order Saga



If only it were this easy…

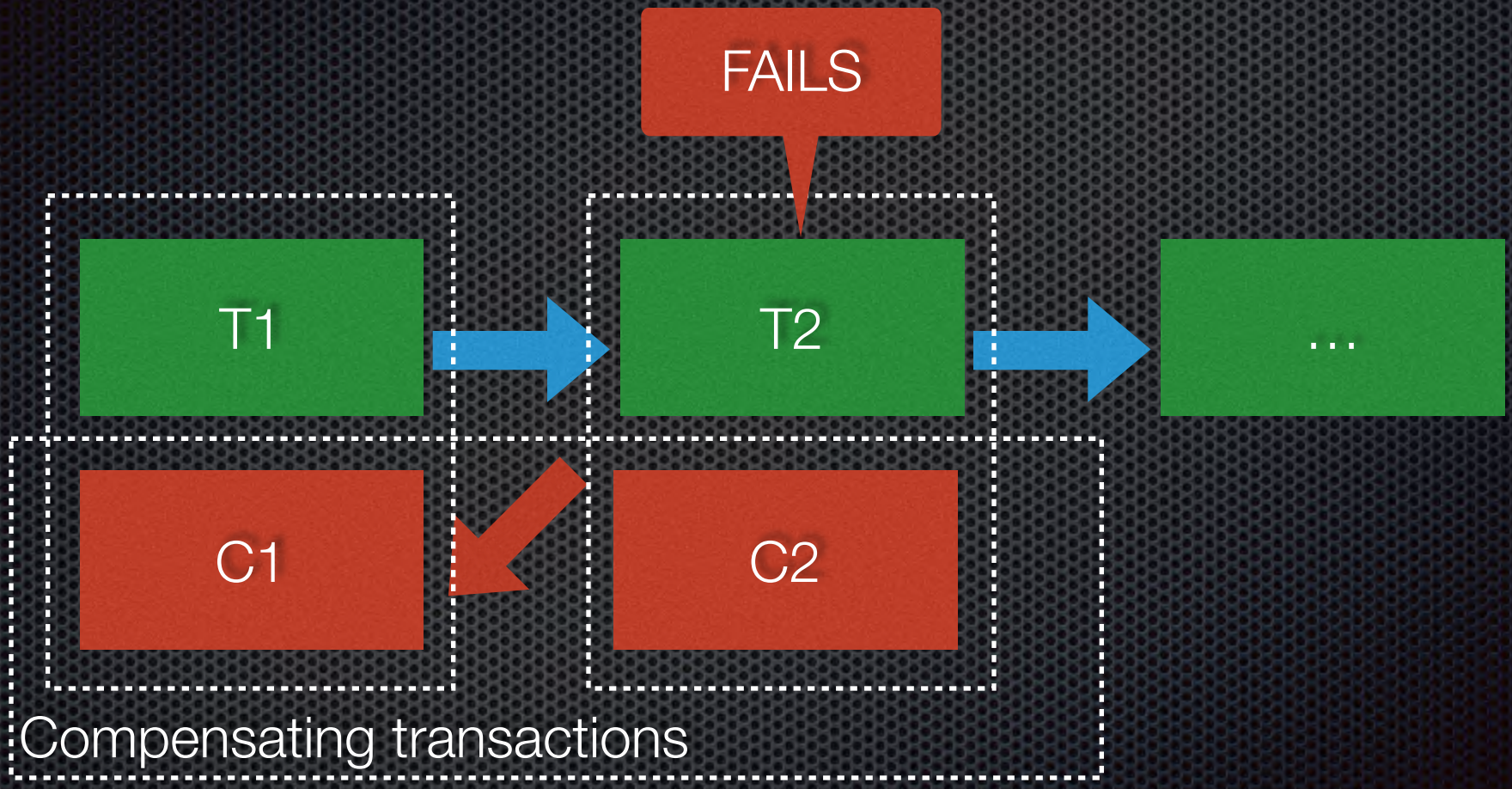
Rollback using compensating transactions

- ACID transactions can simply rollback

BUT

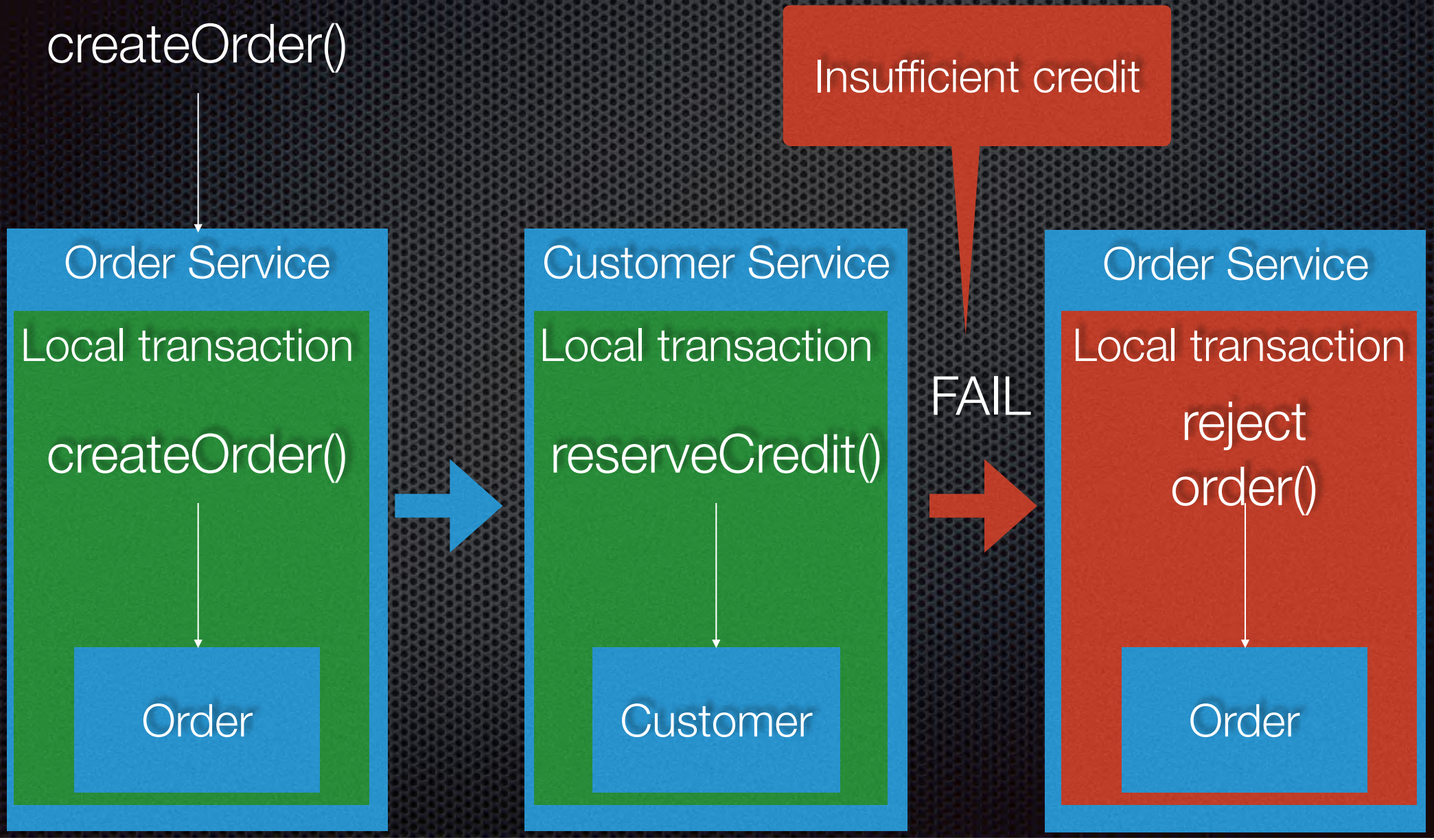
- Developer must write application logic to “rollback” eventually consistent transactions
- Careful design required!

Saga: Every Ti has a Ci



T1 ⇒ T2 ⇒ C1

Create Order Saga - rollback



Sagas complicate API design

- Request initiates the saga. When to send back the response?
- Option #1: Send response when saga completes:

  + Response specifies the outcome

  - Reduced availability

- Option #2: Send response immediately after creating the saga **(recommended)**:

  + Improved availability

  - Response does not specify the outcome. Client must poll or be notified

Revised Create Order API

- createOrder()
  - returns id of newly created order
  - **NOT** fully validated
- getOrder(id)
  - Called periodically by client to get outcome of validation

Minimal impact on UI

- UI hides asynchronous API from the user
- Saga will usually appear instantaneous (<= 100ms)
- If it takes longer ⇒ UI displays “processing” popup
- Server can push notification to UI

Sagas complicate the business logic

- Changes are committed by each step of the saga
- Other transactions see “inconsistent” data, e.g. Order.state = PENDING ⇒ more complex logic
- Interaction between sagas and other operations
  - e.g. what does it mean to cancel a PENDING Order?
  - “Interrupt” the Create Order saga
  - Wait for the Create Order saga to complete?

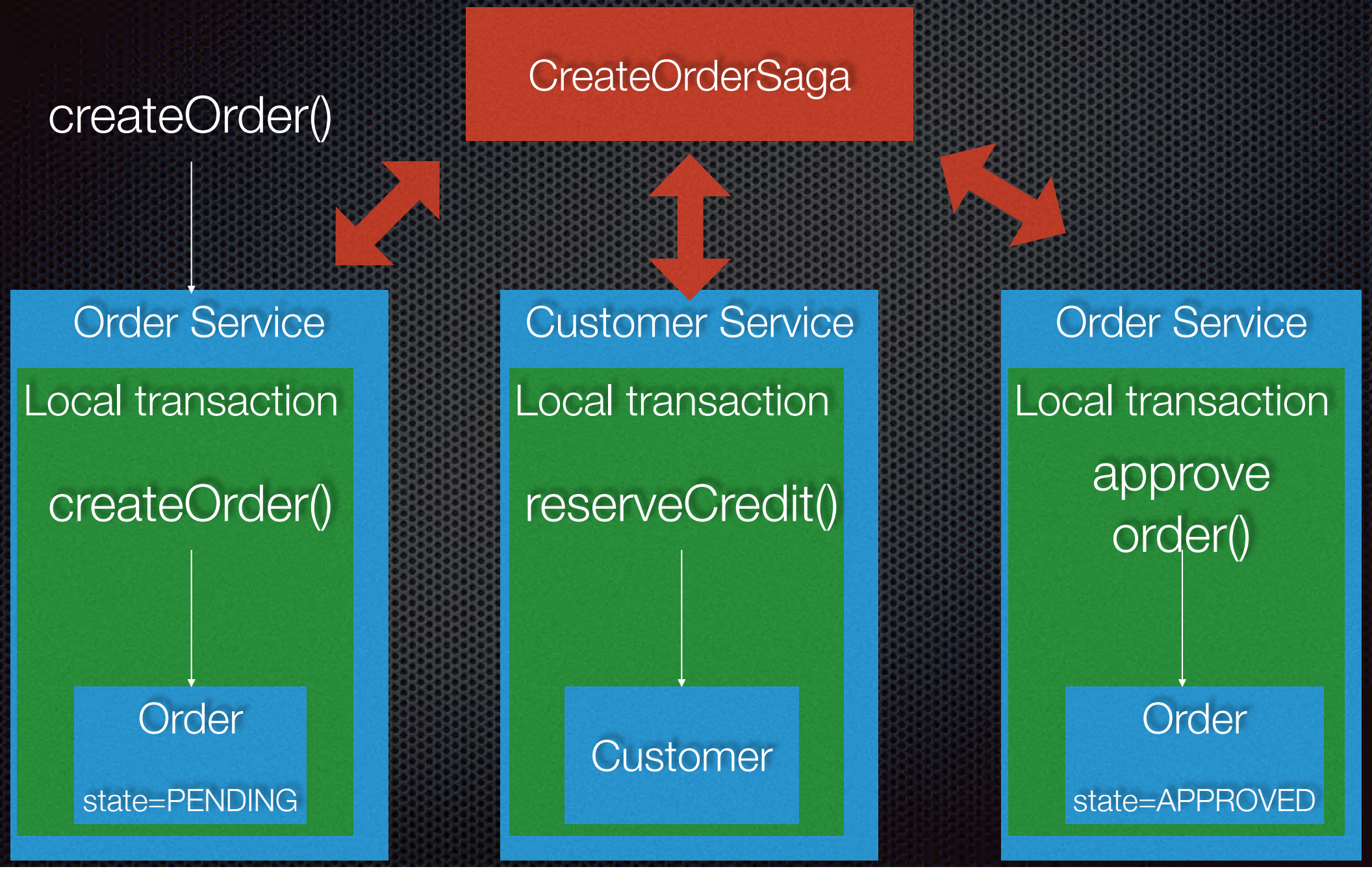
Agenda

- ACID is not an option
- Overview of sagas
- Coordinating sagas

How to sequence the saga transactions?

- After the completion of transaction Ti “something” must decide what step to execute next
- Success: which T(i+1) - branching
- Failure: C(i - 1)

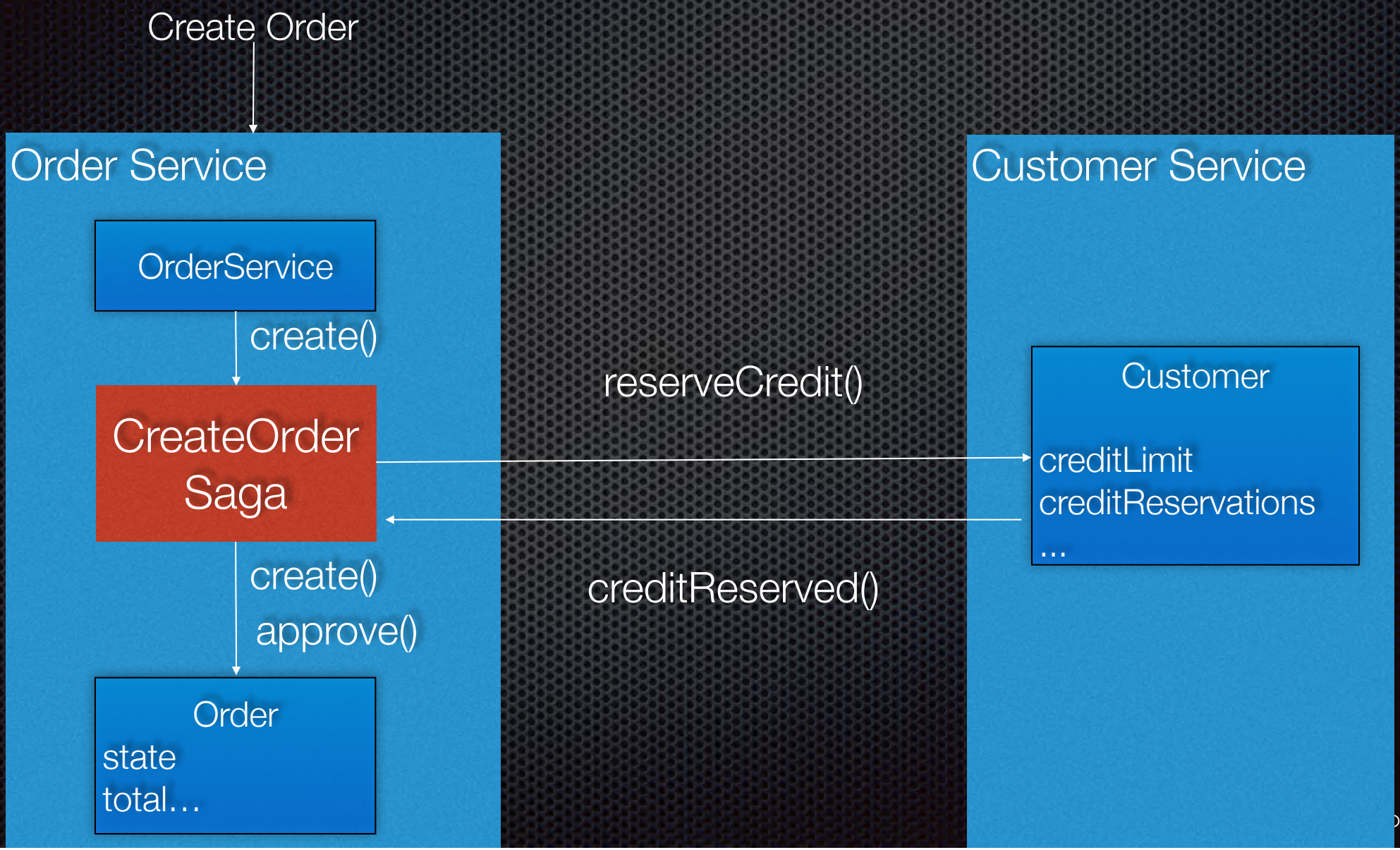
Orchestration-based saga coordination



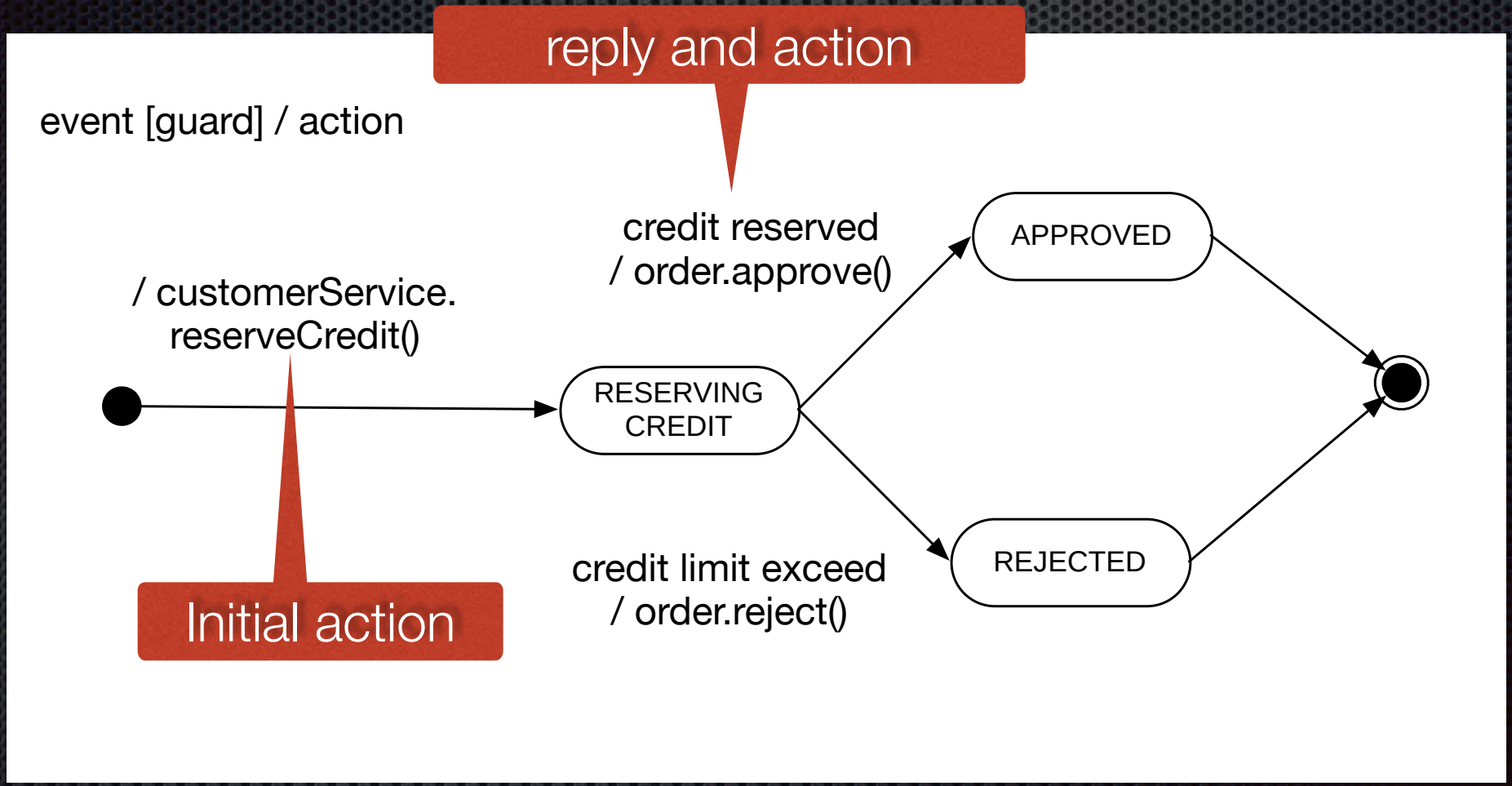
Complex coordination logic is centralized 

Services expose APIs that are invoked by saga 

CreateOrderSaga orchestrator



Saga orchestrators are state machines



Create Order Saga code

State

Enum

Persistent data

```
public class CreateOrderSaga implements Saga<CreateOrderSagaState, CreateOrderSagaData> {
```

Stateless singleton: Behavior

Eventuate Saga framework

Create Order Saga

State machine definition

```
public class CreateOrderSaga
        implements Saga<CreateOrderSagaState, CreateOrderSagaData> {

  private SagaStateMachine<CreateOrderSagaState, CreateOrderSagaData> stateMachine;

  @PostConstruct
  public void initializeStateMachine() {
    this.stateMachine = starting(this::initialize)
            .inState(CreateOrderSagaState.RESERVING_CREDIT)
            .replyFrom(customerService.reserveCredit)
            .onReplyDo(Success.class, this::handleCreditReserved)
            .andTransitionToEndState(CreateOrderSagaState.APPROVED)
            .onReplyDo(Failure.class, this::handleCreditLimitExceeded)
            .andTransitionToEndState(CreateOrderSagaState.REJECTED);

  }

  @Override
  public SagaStateMachine<CreateOrderSagaState, CreateOrderSagaData>
          getStateMachine() { return stateMachine; }
```

Initializing the saga

```
this.stateMachine = starting(this::initialize)
        .inState(CreateOrderSagaState.RESERVING_CREDIT)
```

Create order

```
private SagaActions<CreateOrderSagaState, CreateOrderSagaData> initialize(CreateOrderSagaData data) {
  ResultWithEvents<Order> oe = Order.createOrder(data.getOrderDetails());
  orderRepository.save(oe.result);

  data.setOrderId(oe.result.getId());

  ReserveCreditCommand cmd =
          new ReserveCreditCommand(data.getOrderId(), data.getOrderDetails().getOrderTotal());

  return sending(customerService.reserveCredit
          .makeCommand(singletonMap("id",
                  Long.toString(data.getOrderDetails().getCustomerId())), cmd))
          .using(data);
}
```

Invoke saga participant

Handling a reply

```
this.stateMachine = onStartDo(this::initialize)
        .withStartState(CreateOrderSagaState.RESERVING_CREDIT)
        .inState(CreateOrderSagaState.RESERVING_CREDIT)
        .replyFrom(customerService.reserveCredit)
        .onReplyDo(Success.class, this::handleCreditReserved)
```

```
private void handleCreditReserved(CreateOrderSagaData data, Success reply) {
  Order order = orderRepository.findOne(data.getOrderId());
  order.noteCreditReserved();
}
```

Update Order

Customer Service - command handling

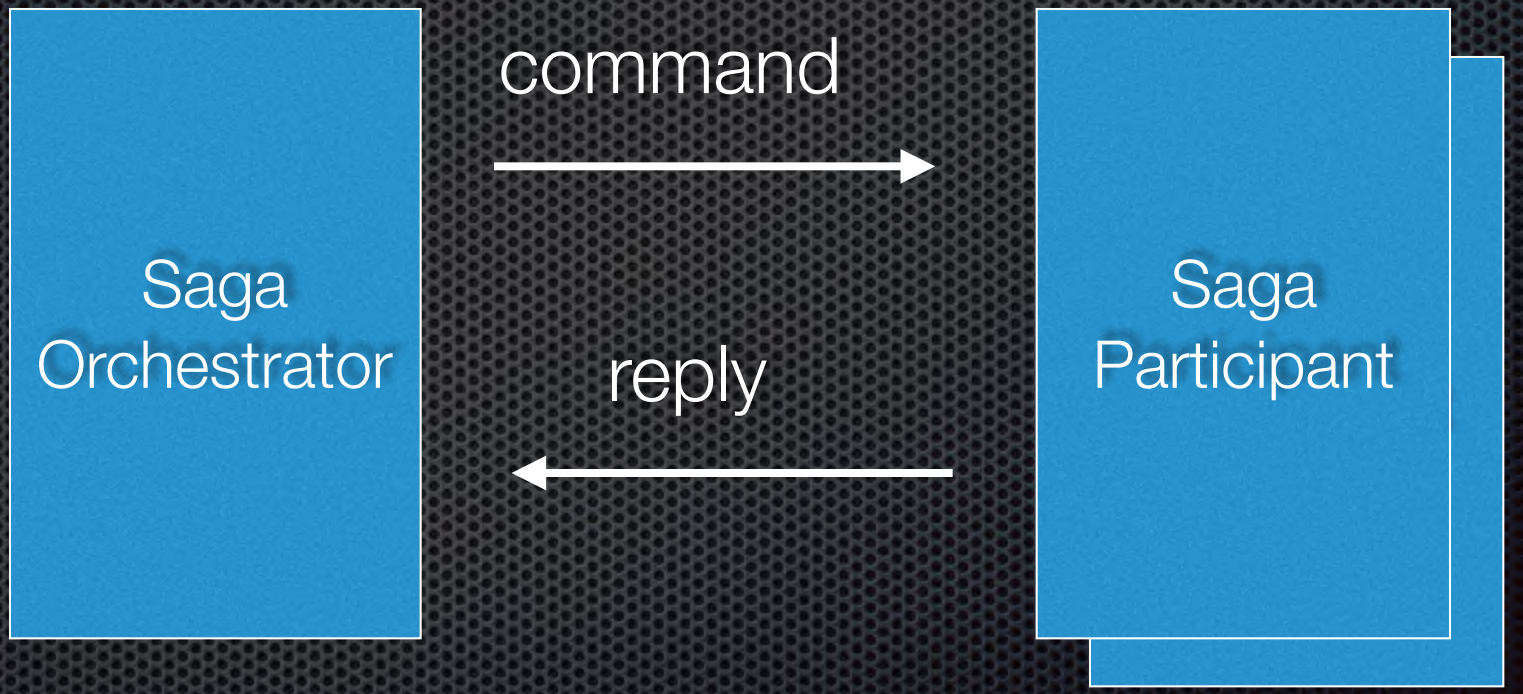
```
@CommandHandler("customerService")
public class CustomerCommandHandler {

  @Autowired
  private CustomerRepository customerRepository;

  @CommandHandlerMethod(path="/customers/{customerId}",
          replyChannel = "'CustomerAggregate'", partitionId="path['id']")
  public Success reserveCredit(@PathVariable("customerId") String customerId,
                               CommandMessage<ReserveCreditCommand> cm) {
    ReserveCreditCommand cmd = cm.getCommand();
    Customer customer = customerRepository.findOne(Long.parseLong(customerId));
    customer.reserveCredit(cmd.getOrderId(), cmd.getOrderTotal());
    return new Success();
  }
```

Reserve credit

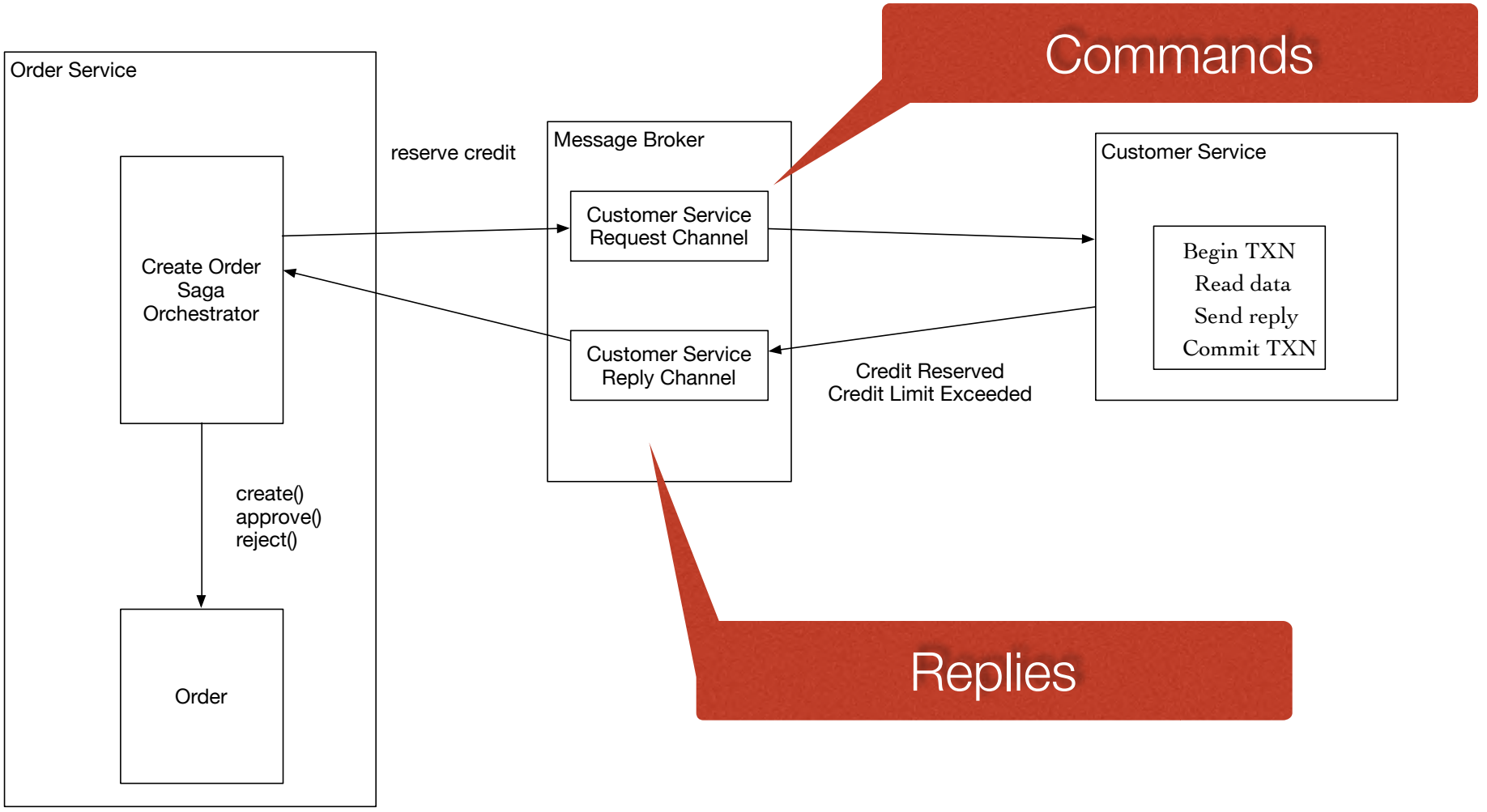
About Saga orchestrator ⇔ participant communication



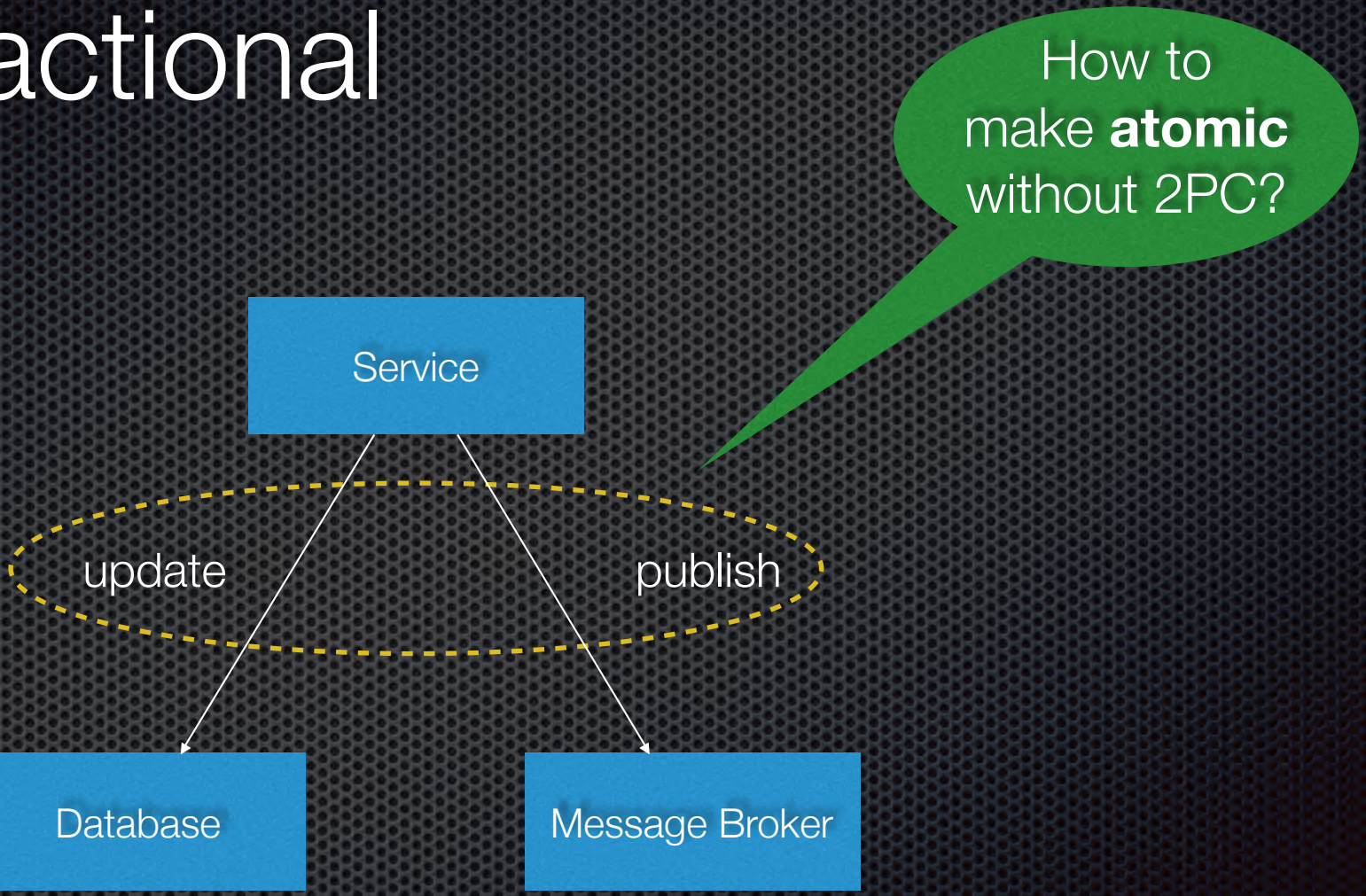
Saga must complete even if there are transient failures

Use asynchronous messaging

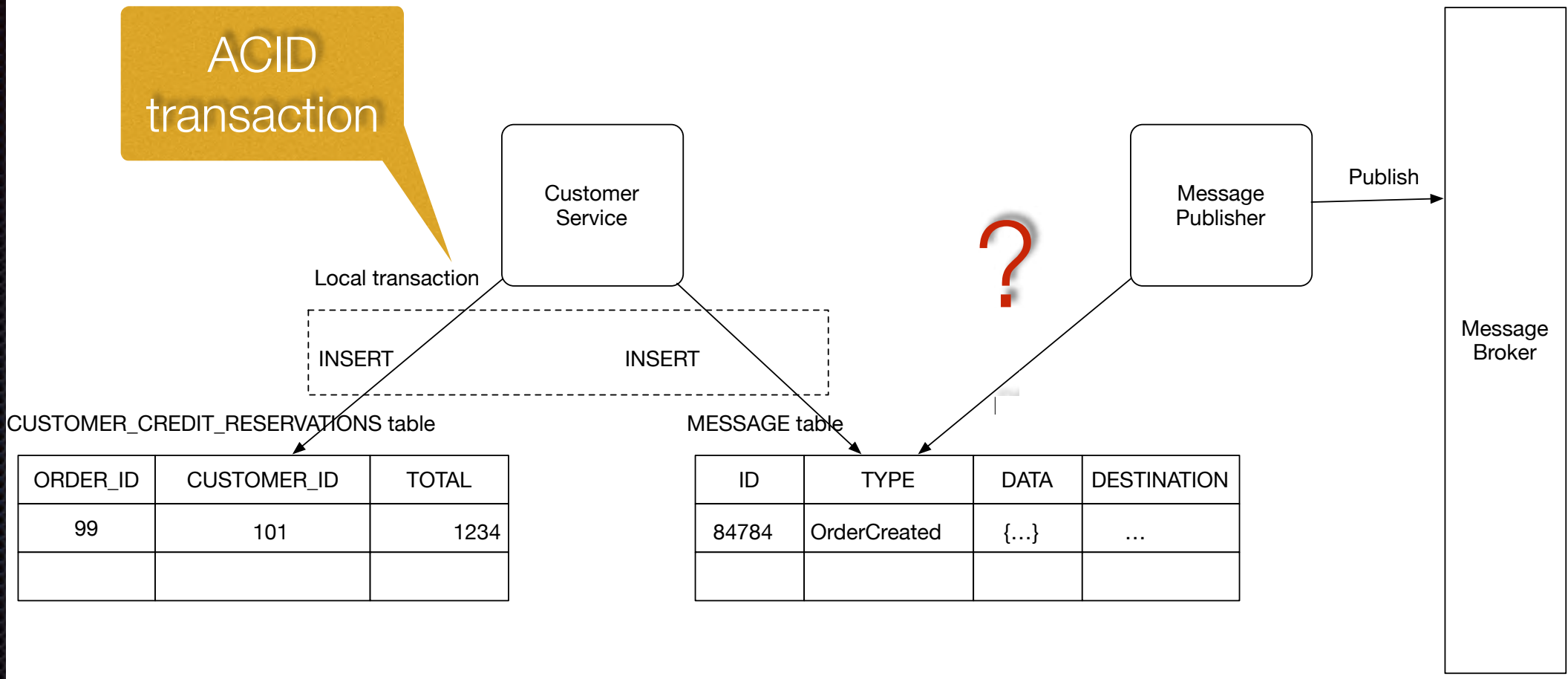
Create Order Saga messaging



Messaging must be transactional



Option #1: Use database table as a message queue



- See BASE: An Acid Alternative, http://bit.ly/ebaybase

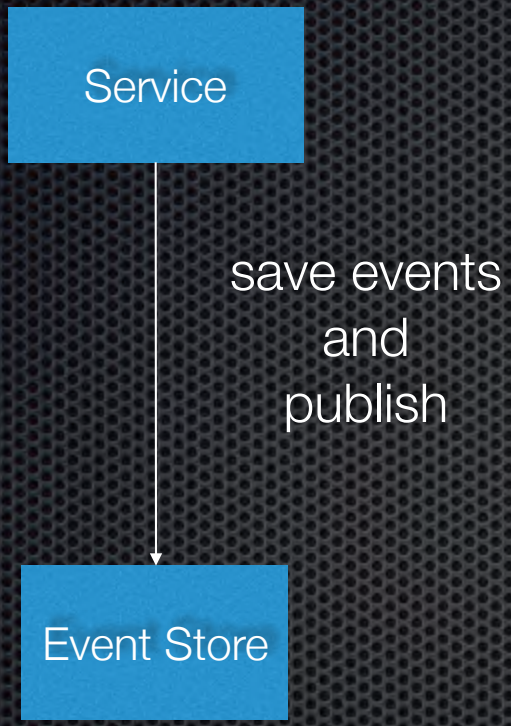
Publishing messages

Poll the MESSAGE table

OR

Tail the database transaction log

Option #2: Event sourcing: event-centric persistence



Event table

Entity id	Entity type	Event id	Event type	Event data
101	Order	901	OrderCreated	…
101	Order	902	OrderApproved	…
101	Order	903	OrderShipped	…

Every state change ⇒ event

Summary

- Microservices tackle complexity and accelerate development
- Database per service is essential for loose coupling
- Use sagas to maintain data consistency across services
- Use transactional messaging to make sagas reliable


@crichardson chris@chrisrichardson.net
Questions?
http://learnmicroservices.io