ACID Is So Yesterday: Maintaining Data Consistency with Sagas

Chris Richardson

Founder of Eventuate.io

Founder of the original CloudFoundry.com

Author of POJOs in Action

@crichardson

chris@chrisrichardson.net

http://eventuate.io

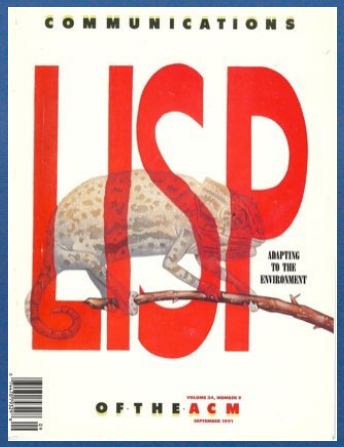
QCon SAN FRANCISCO

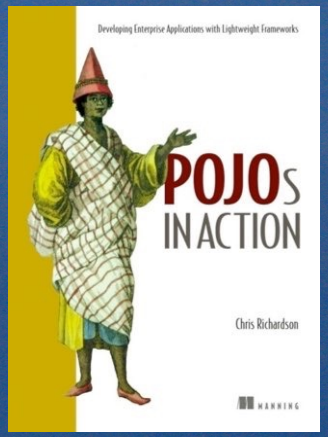
Presentation goal

Distributed data management challenges in a microservice architecture

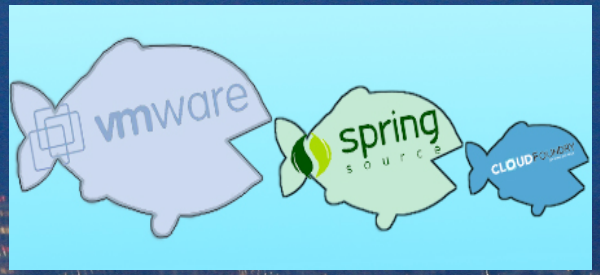
Sagas as the transaction model

About Chris


COMMUNICATIONS
LISP
ADAPTING TO THE ENVIRONMENT
OF THE ACM


Developing Enterprise Applications with Lightweight Frameworks
POJOs IN ACTION
Chris Richardson
MANNING


CLOUDFOUNDRY
SIGN IN
SIGN UP
SYSTEM ALERT. PLEASE READ: Cloud Foundry will be moving to a new URL.
The Enterprise Java Cloud
Real Java Applications Deployed in Minutes
Built for Spring and Grails Web Applications
Most Widely Used Technologies Delivered as a Platform
SIGN UP
LEARN MORE
APPLICATION DEMO
Deploying Web Applications To Amazon EC2 with Cloud Foundry


vmware
spring
source
CLOUDFOUNDRY

About Chris

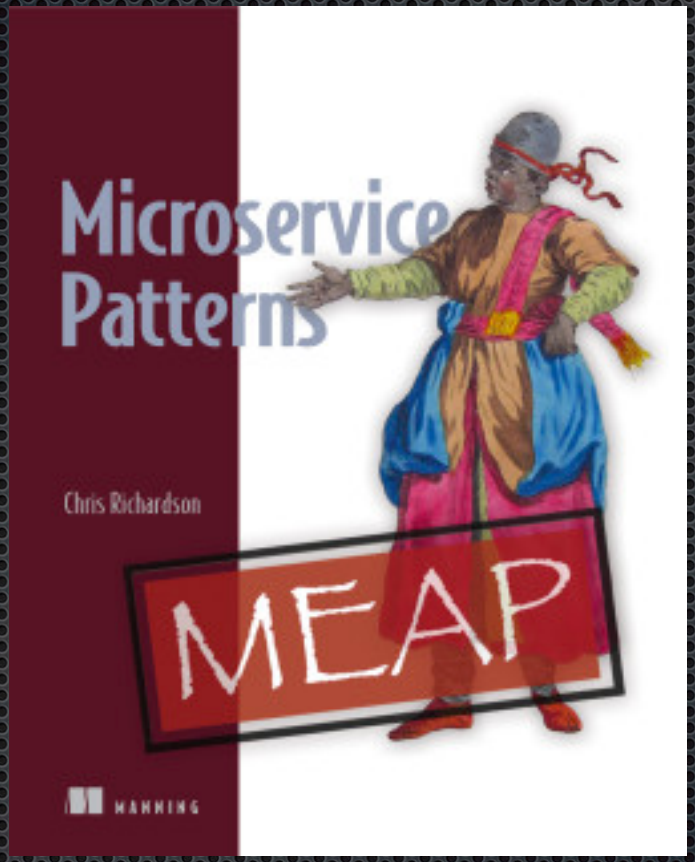
Consultant and trainer focusing on modern application architectures including microservices (http://www.chrisrichardson.net/)

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



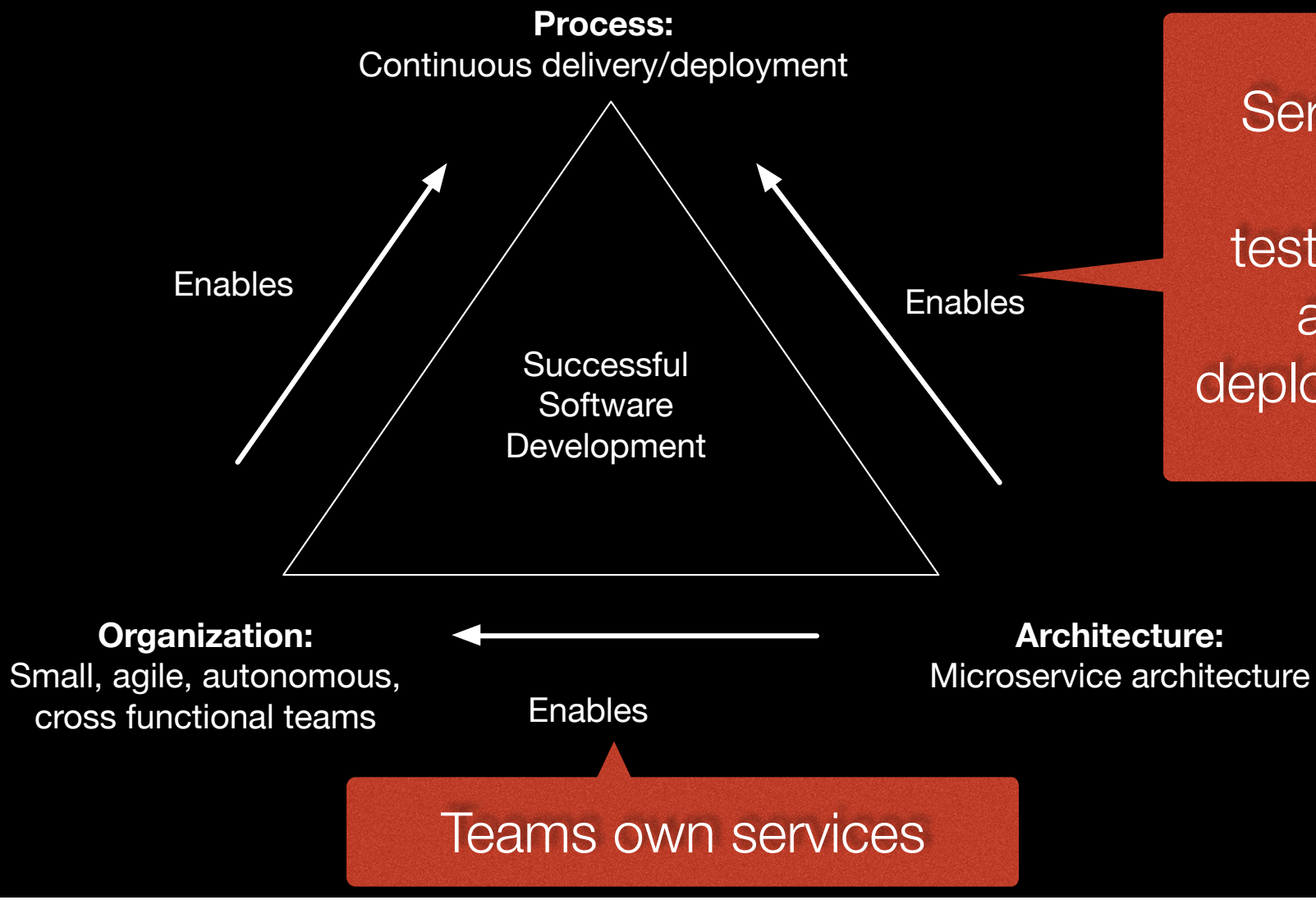
http://learnmicroservices.io

Agenda

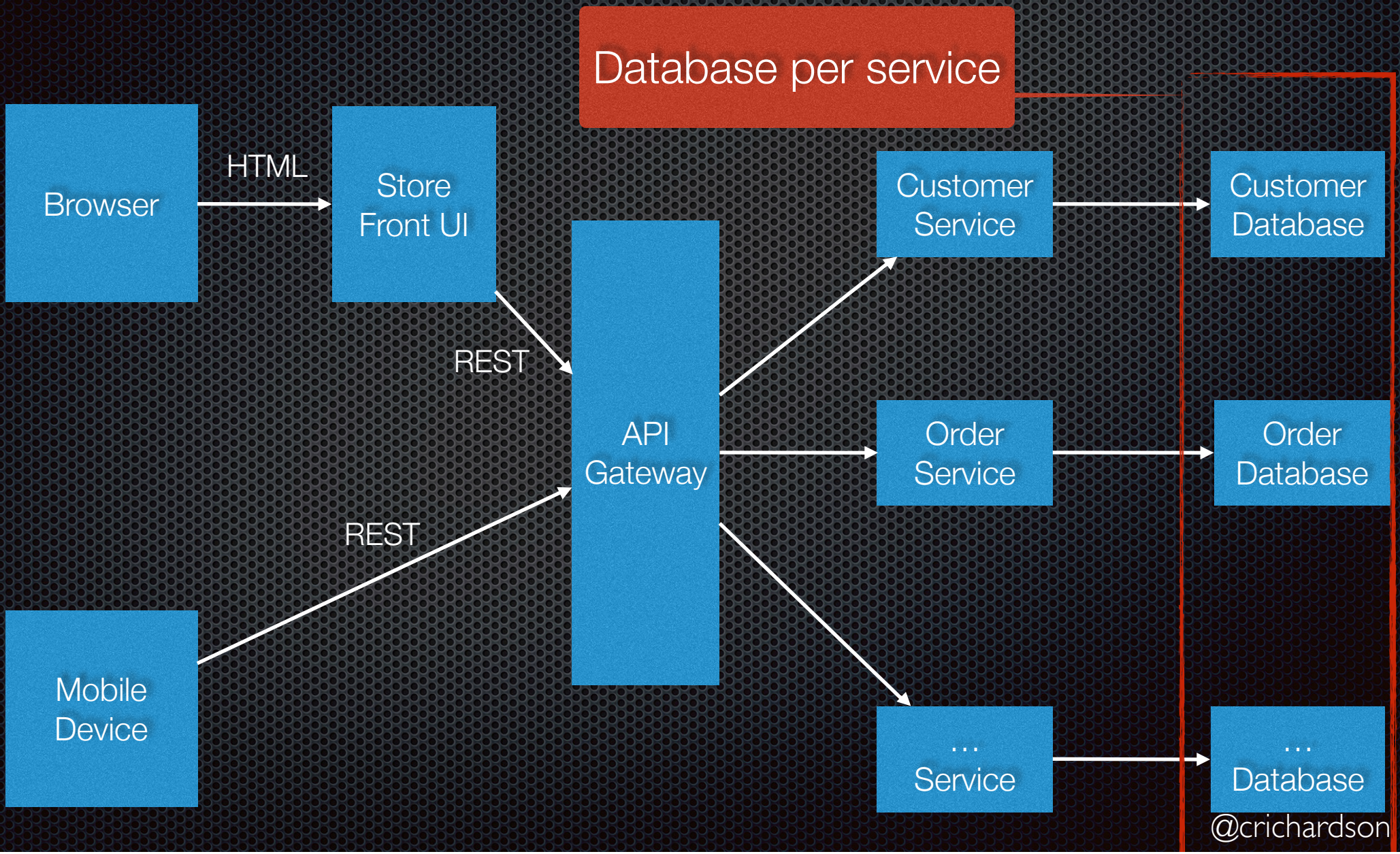
- ACID is not an option
- Overview of sagas
- Coordinating sagas
- Sagas and inter-service communication

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

Microservices enable continuous delivery/deployment



Microservice architecture

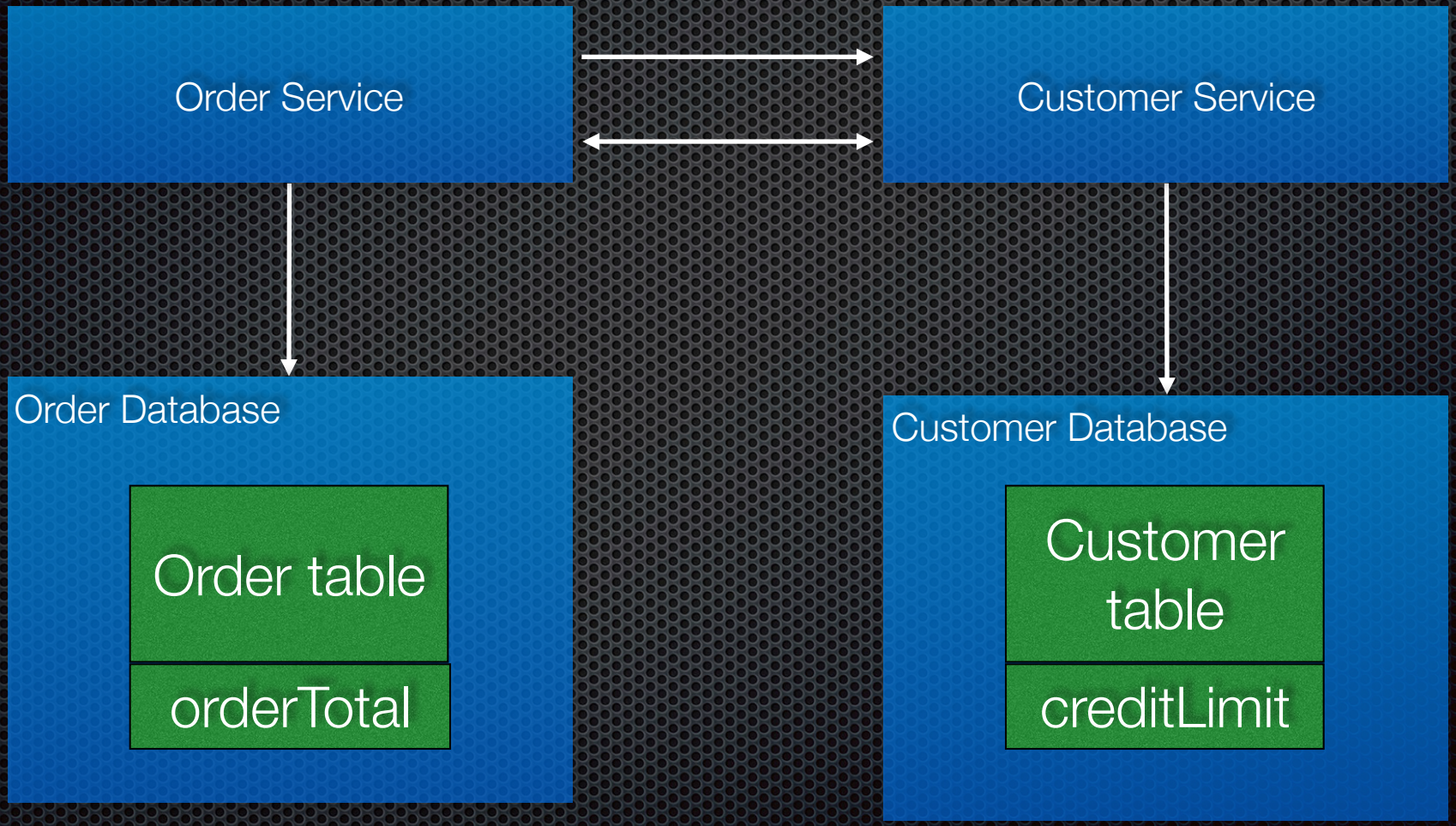


Private database

!=

private database **server**

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
- ….

ACID ⇒

Basically
Available
Soft state
Eventually consistent

http://queue.acm.org/detail.cfm?id=1394128

Agenda

- ACID is not an option
- Overview of sagas
- Coordinating sagas
- Sagas and inter-service communication

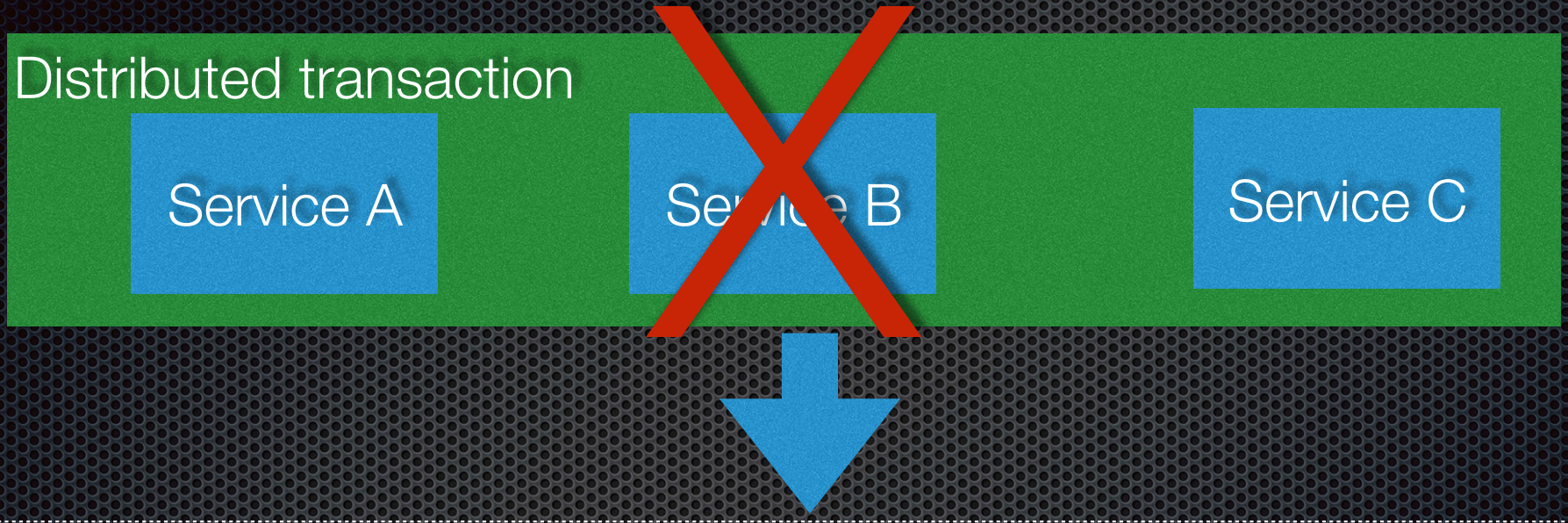
From a 1987 paper

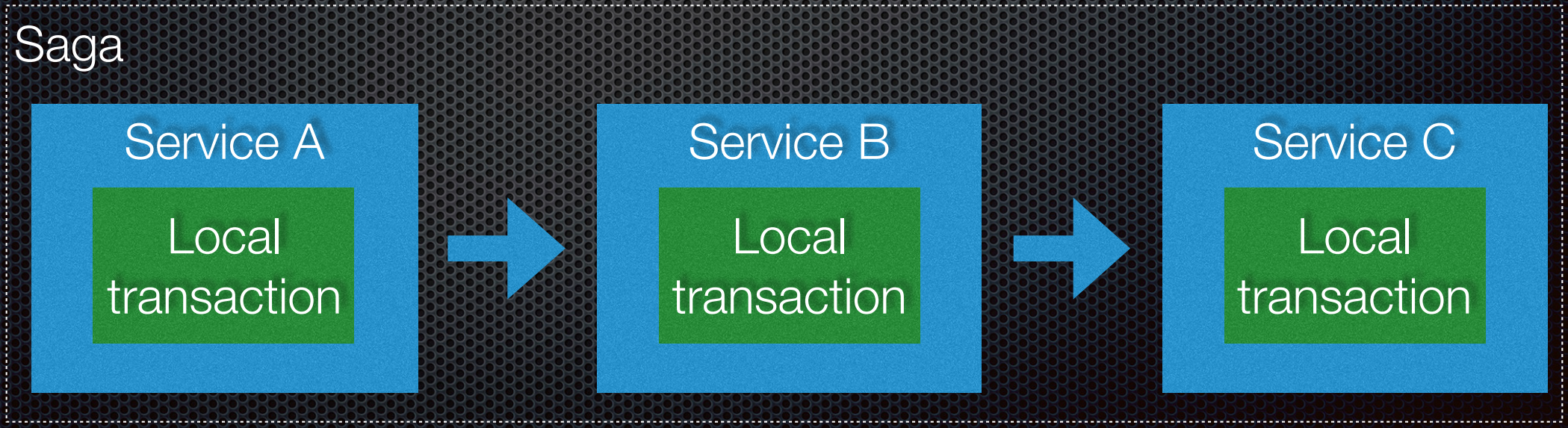
SAGAS

Hector Garcia-Molina
Kenneth Salem

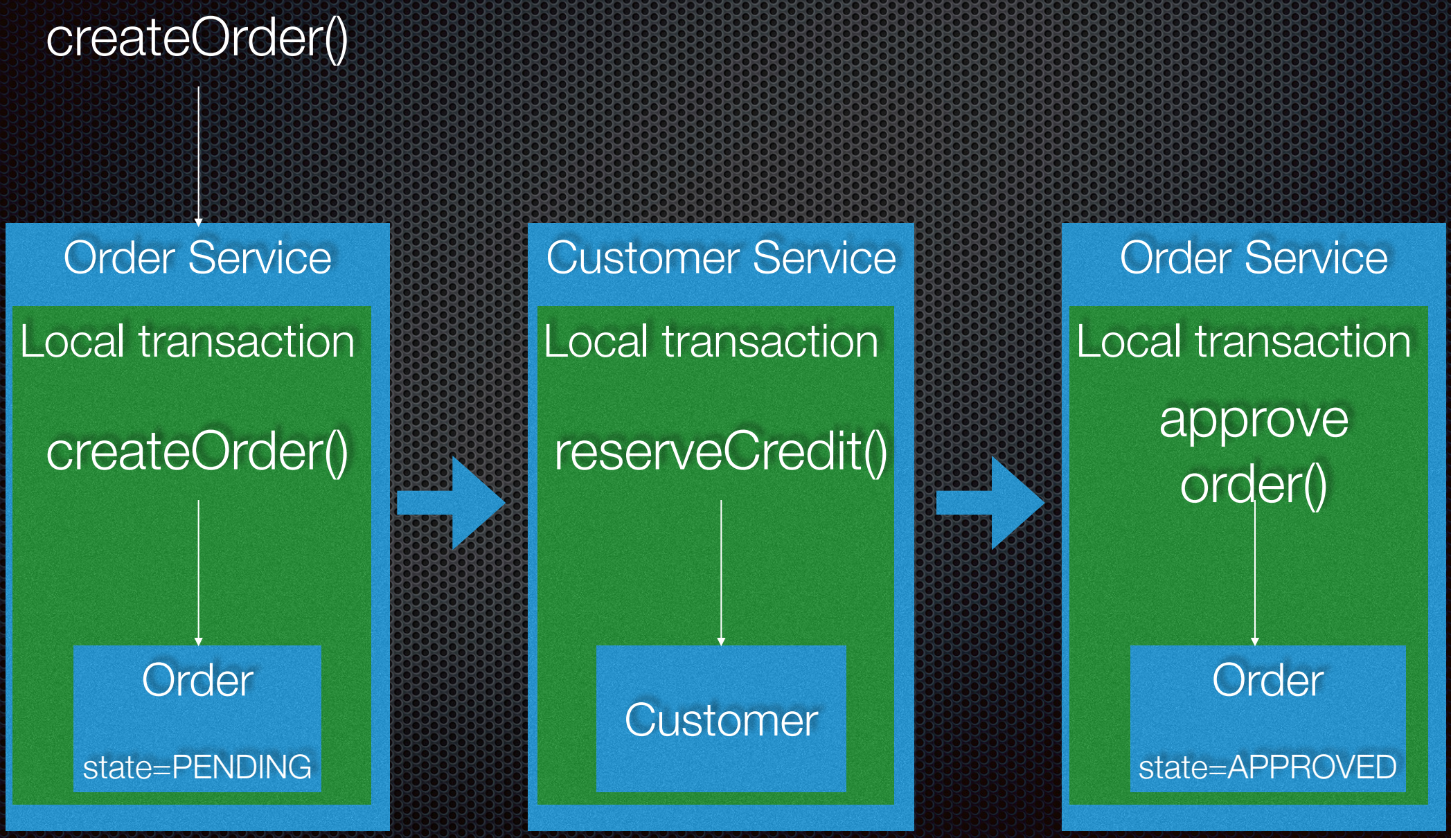
Department of Computer Science
Princeton University
Princeton, N J 08544

Use Sagas instead of 2PC





Create Order Saga



If only it were this easy....

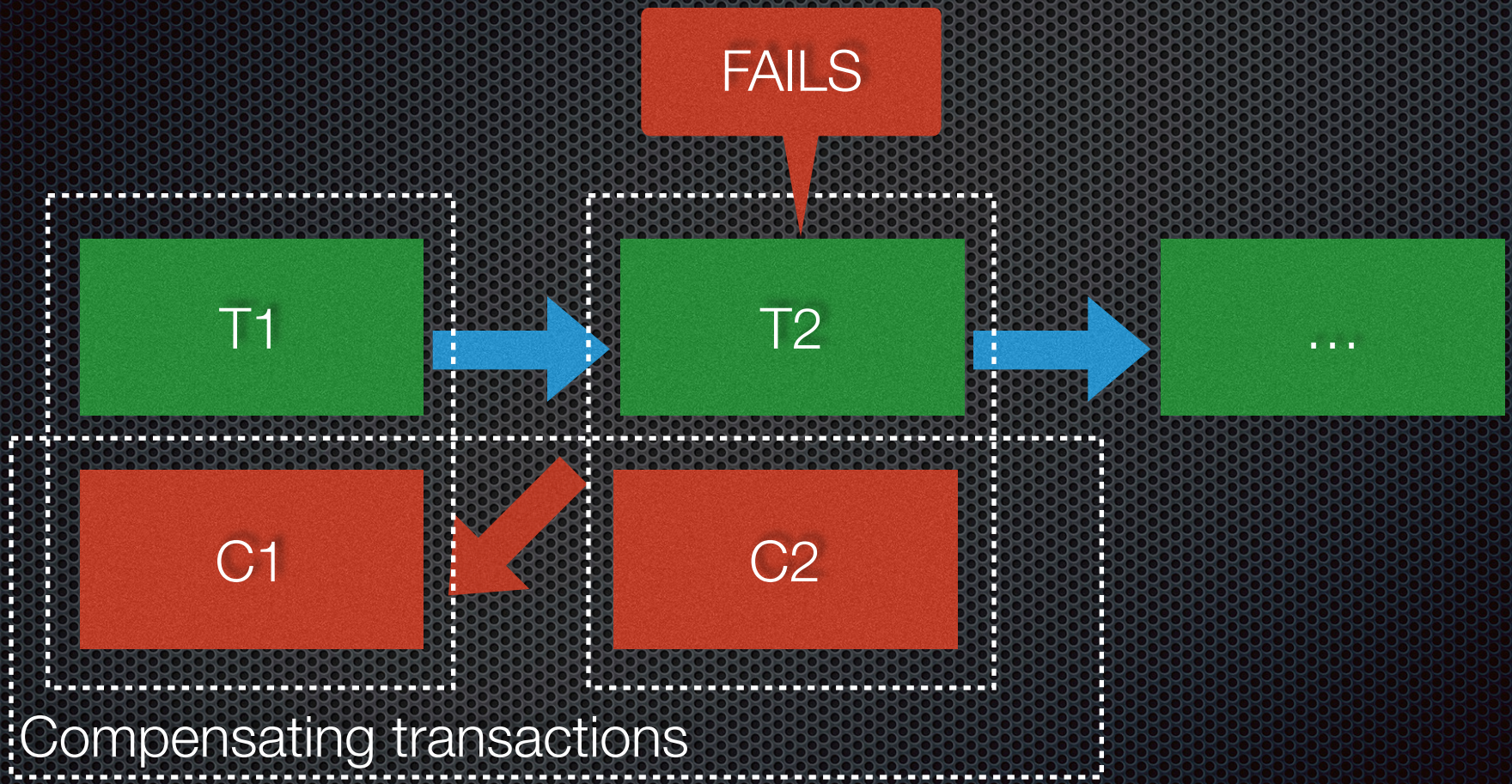
Rollback using compensating transactions

- ACID transactions can simply rollback

BUT

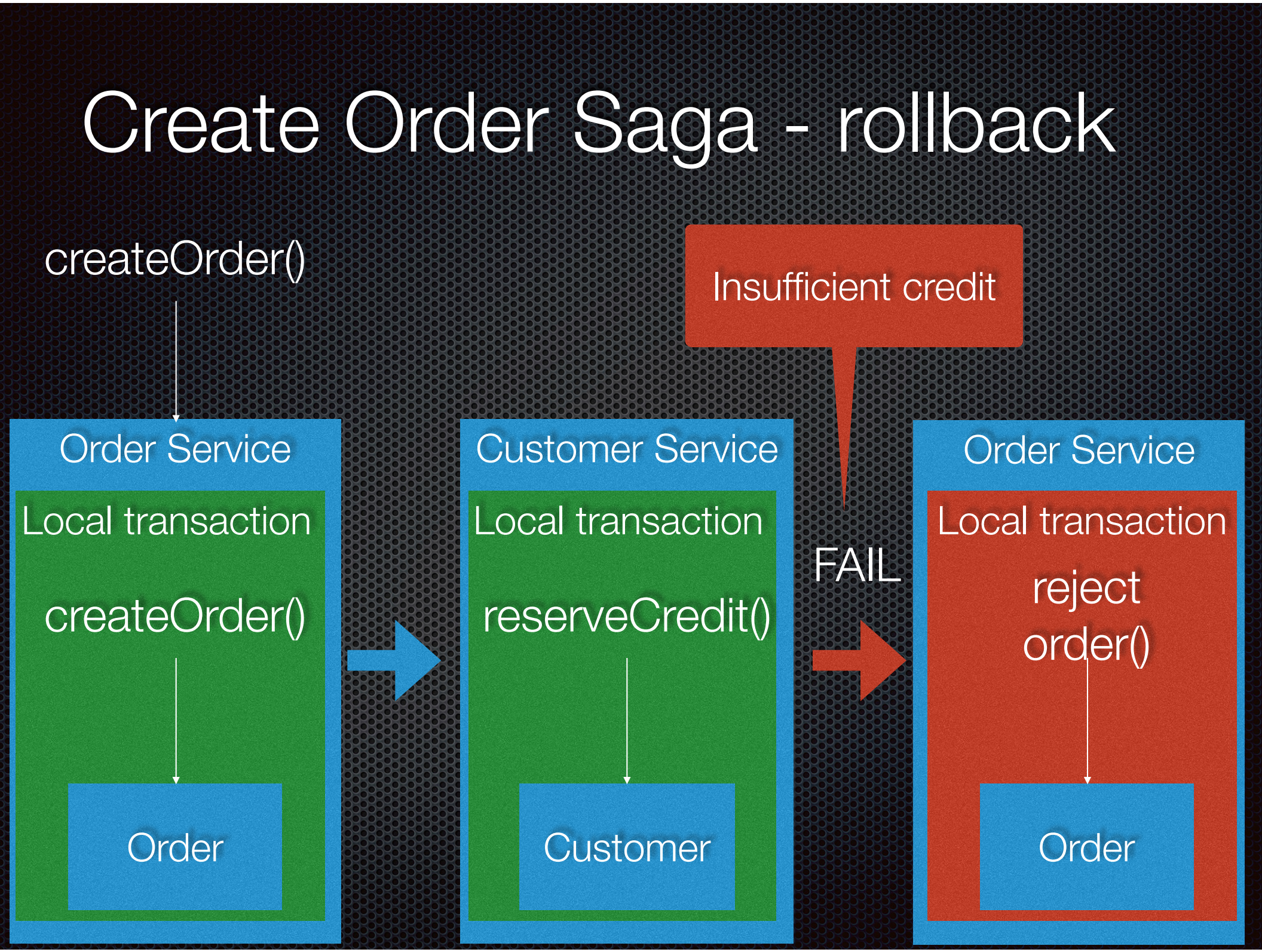
- Developer must write application logic to “rollback” eventually consistent transactions
- Careful design required!

Saga: Every Ti has a Ci



T1 ⇒ T2 ⇒ C1

Create Order Saga - rollback



Sagas complicate API design

- Synchronous API vs Asynchronous Saga
- 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

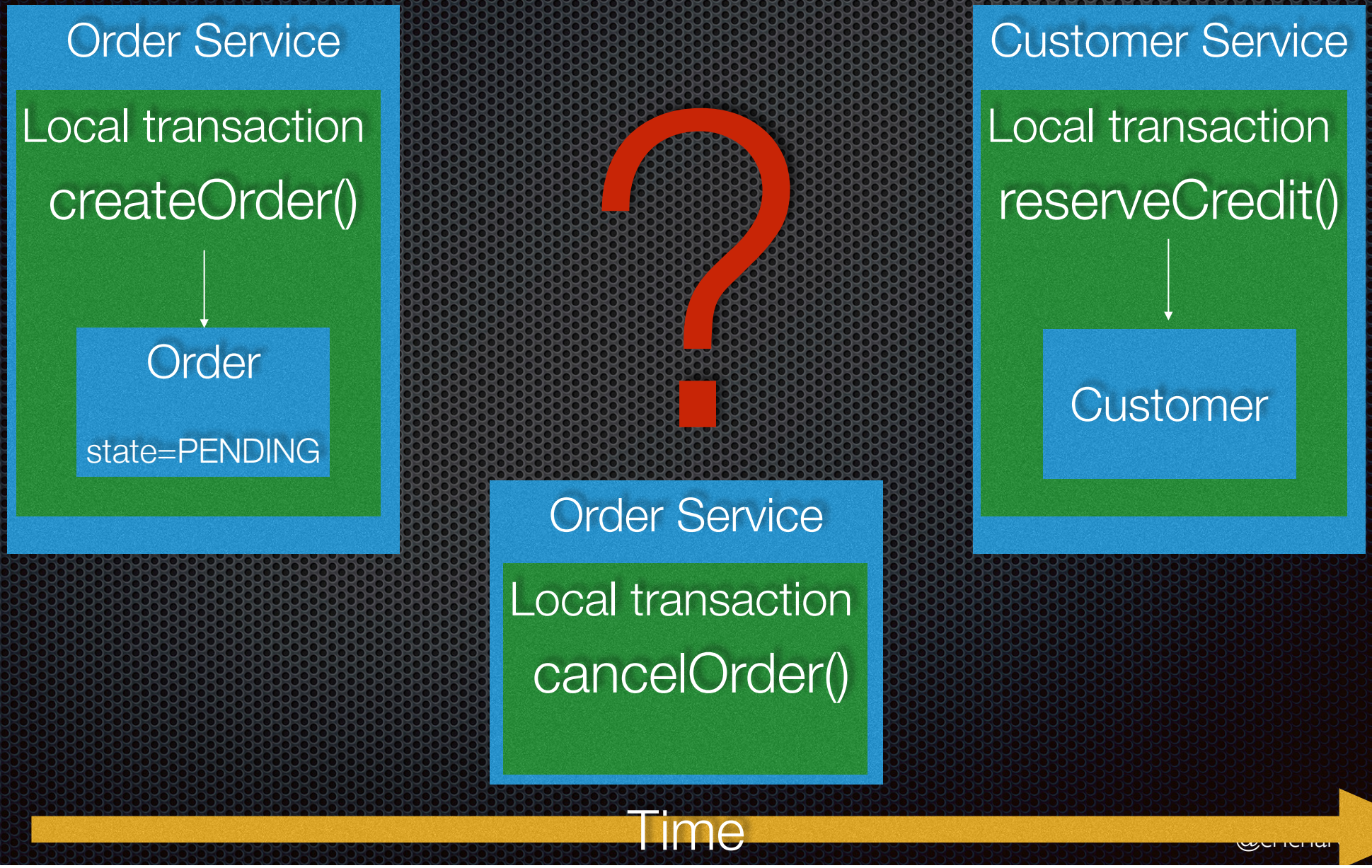
Revised Create Order API

- 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

Lack of isolation ⇒ complicates business logic



How to cancel a PENDING Order?

- Don't ⇒ throw an OrderNotCancellableException
  - Questionable user experience
- "Interrupt" the Create Order saga?
  - Cancel Order Saga: set order.state = CANCELLED
  - Causes Create Order Saga to rollback
  - But is that enough to cancel the order?
- Cancel Order saga waits for the Create Order saga to complete?
  - Suspiciously like a distributed lock
  - But perhaps that is ok

Countermeasure Transaction Model

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 28(1), 77–98 (JANUARY 1998)

Semantic ACID Properties in Multidatabases Using Remote Procedure Calls and Update Propagations

lars frank[1] and torben u. zahle[2]

Saga structure

- Series of compensatable transactions (Ti,Ci)
- Pivot transaction (Ti)
  - “not compensatable or retriable”
  - Execute compensating transactions if it fails
  - **GO/NO GO point**
- Set of retriable transactions (Ti)
  - Can't fail

Sagas are ACD

- **Atomicity**
  - Saga implementation ensures that all transactions are executed **OR** all are compensated
- **Consistency**
  - Referential integrity within a service handled by local databases
  - Referential integrity across services handled by application
- **Durability**
  - Durability handled by local databases

Lack of I ⇒ anomalies

- Lost update
  - Ti reads ⇒ other transaction writes ⇒ Tj (or Ci) writes
- Dirty reads
  - Ti writes ⇒ other transaction reads ⇒ Ci writes
- non-repeatable/fuzzy read
  - Ti reads ⇒ other transaction writes ⇒ Tj reads

Countermeasures for reducing impact of isolation anomalies...

- Commutative updates
  - e.g. debit account can compensate for a credit account
- Version file
  - Record history of changes
  - Use them to make updates commutative
  - e.g. record cancel reservation so that create/cancel = cancel/create
  - Sounds suspiciously like event sourcing

…Countermeasures for reducing impact of isolation anomalies…

- Re-read value
  - Before modifying value, Ti re-reads value that was read by a previous Ti
  - Abort if the value has changed (and possibly restart)
- Pessimistic view
  - Minimize the business risk
  - Reduce available credit in compensatable transaction
  - Increase available credit in retriable transaction, which will never be compensated

...Countermeasures for reducing impact of isolation anomalies

- Countermeasures by value
  - Business risk determine strategy
  - High risk => use 2PC/distributed transaction
- Semantic lock
  - Compensatable transaction sets flag, retriable transaction releases it
  - Flag = lock - prevents other transactions from accessing it
  - Flag = warning - treat the data differently, e.g. a pending deposit
  - Require deadlock detection, e.g. timeout

Agenda

- ACID is not an option
- Overview of sagas
- Coordinating sagas
- Sagas and inter-service communication

How to sequence the saga transactions?

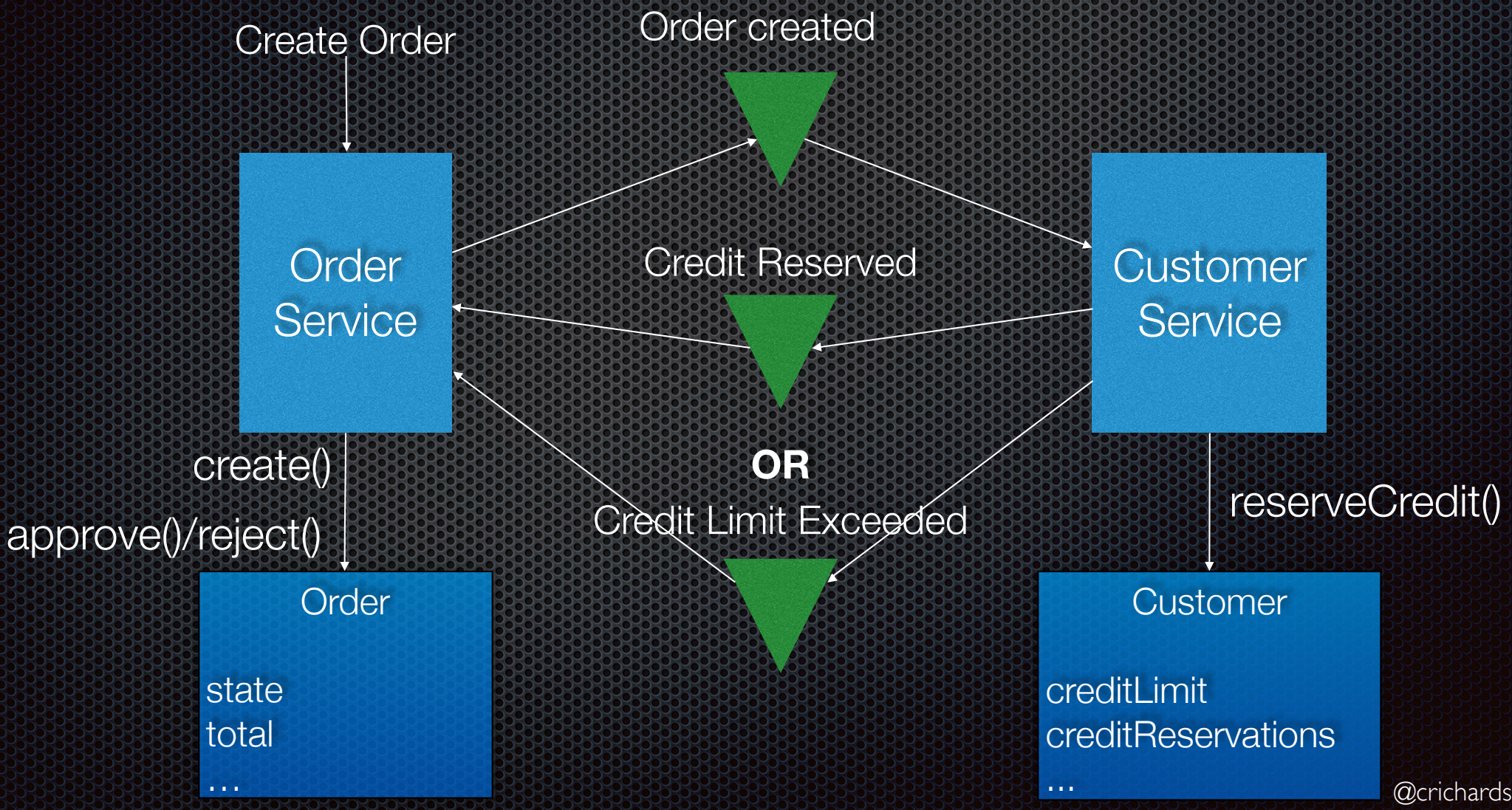
- After the completion of transaction Ti “something” must decide what step to execute next
- Success: which T(i+1) - branching
- Failure: C(i - 1)

Choreography: **distributed** decision making

vs.

Orchestration: **centralized** decision making

Option #1: Choreography-based coordination using events



Benefits and drawbacks of choreography

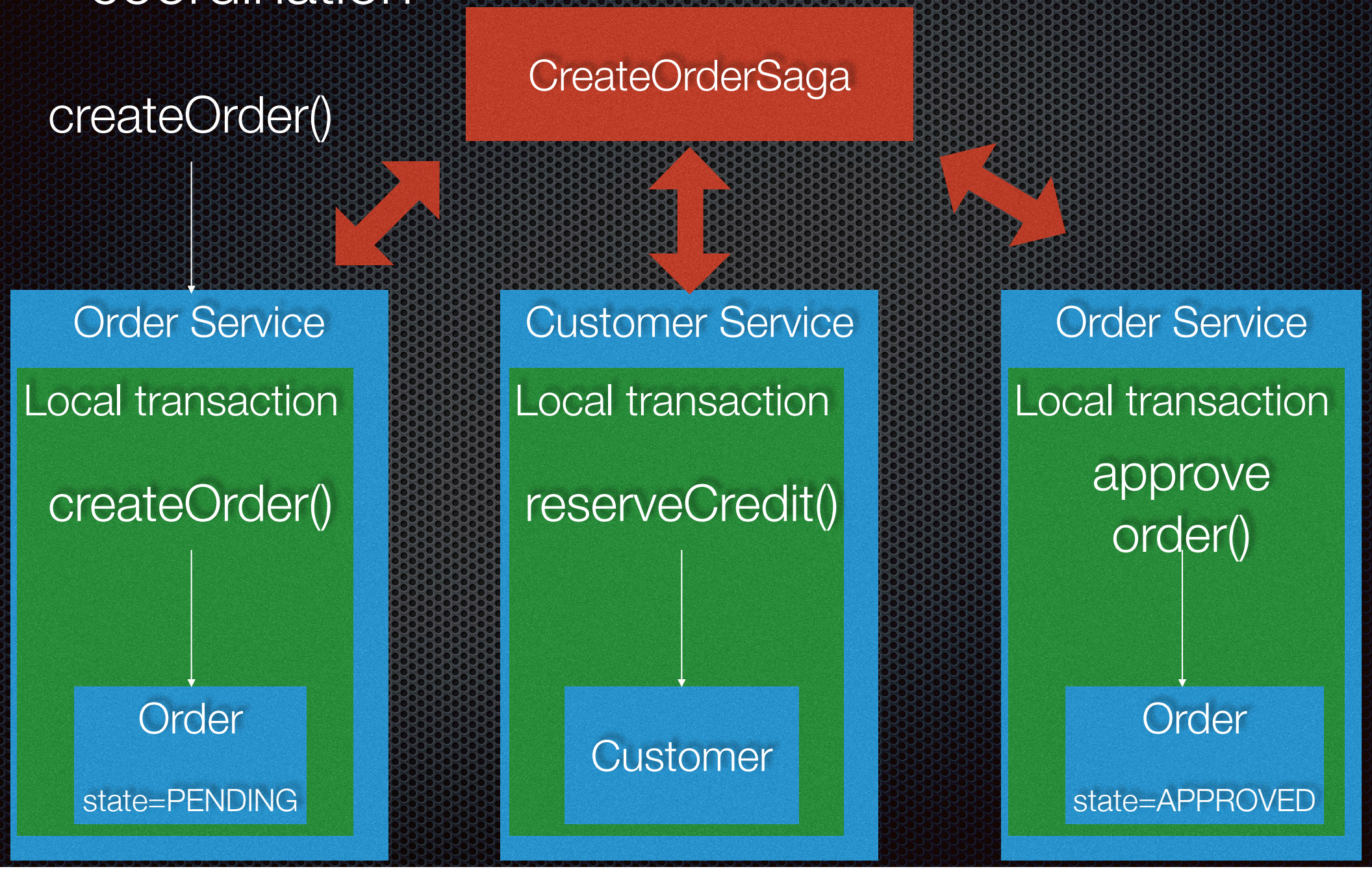
Benefits

- Simple, especially when using event sourcing
- Participants are loosely coupled

Drawbacks

- Cyclic dependencies - services listen to each other's events
- Overloads domain objects, e.g. Order and Customer **know** too much
- Events = indirect way to make something happen

Option #2: Orchestration-based saga coordination

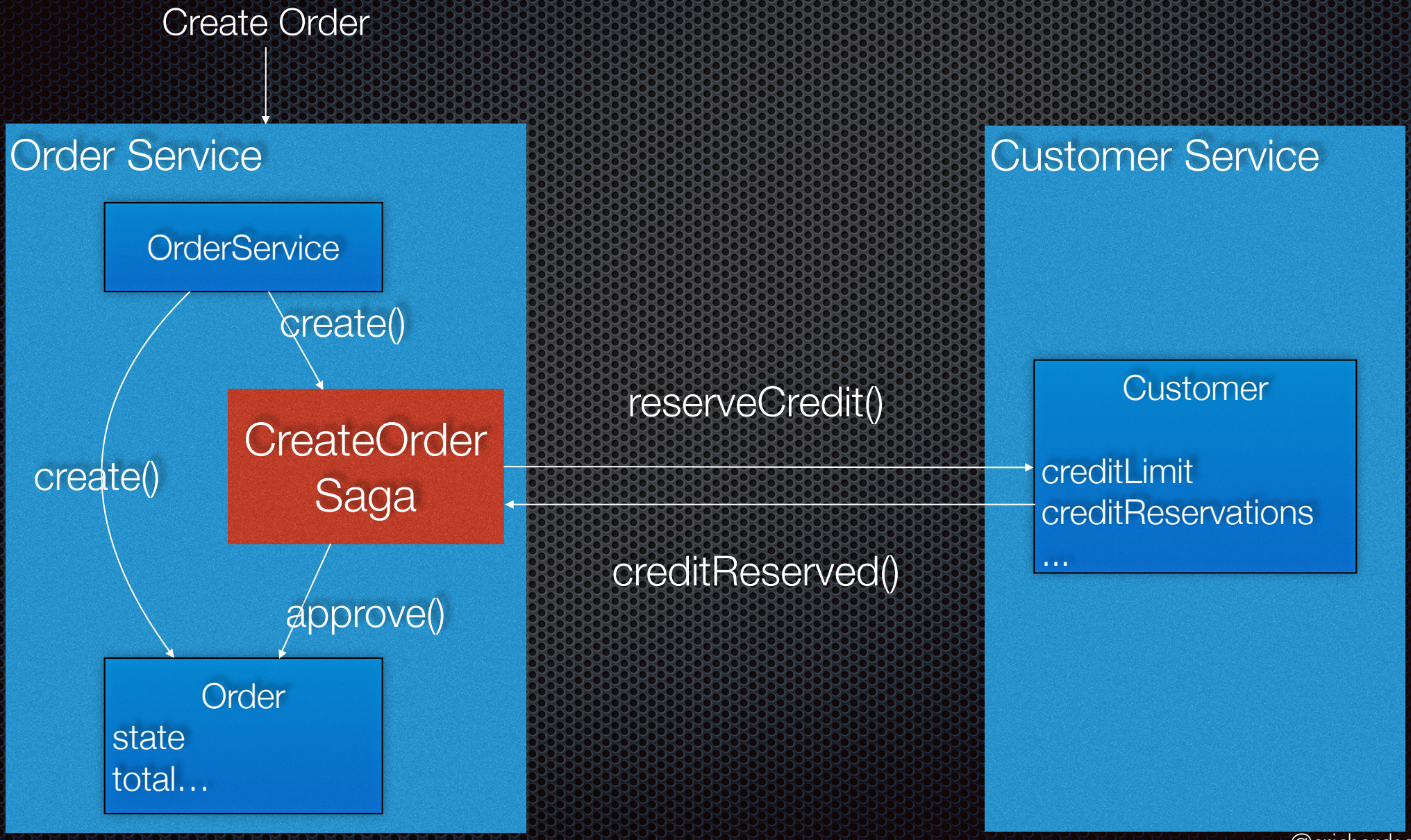


A saga (orchestrator)
is a **persistent object**
that
tracks the state of the saga
and
invokes the participants

Saga behavior

- On create:
  - Invokes a saga participant
- On reply:
  - Determine which saga participant to invoke next
  - Invokes saga participant
  - Updates its state
  - …

CreateOrderSaga orchestrator



CreateOrderSaga definition

```
public class CreateOrderSaga implements SimpleSaga<CreateOrderSagaData> {

  private SagaDefinition<CreateOrderSagaData> sagaDefinition =
          step()
            .withCompensation(this::reject)
          .step()
            .invokeParticipant(this::reserveCredit)
          .step()
            .invokeParticipant(this::approve)
          .build();


  @Override
  public SagaDefinition<CreateOrderSagaData> getSagaDefinition() { return this.sagaDefinition; }


  private CommandWithDestination reserveCredit(CreateOrderSagaData data) {
    long orderId = data.getOrderId();
    Long customerId = data.getOrderDetails().getCustomerId();
    Money orderTotal = data.getOrderDetails().getOrderTotal();
    return send(new ReserveCreditCommand(customerId, orderId, orderTotal))
            .to("customerService")
            .build();
  }
```

Saga's Data

Sequence of steps

step = (T_i, C_i)

Build command to send

Customer Service command handler

```
public class CustomerCommandHandler {

  @Autowired
  private CustomerRepository customerRepository;

  public CommandHandlers commandHandlerDefinitions() {
    return SagaCommandHandlersBuilder
            .fromChannel("customerService")
            .onMessage(ReserveCreditCommand.class, this::reserveCredit)
            .build();
  }

  public Message reserveCredit(CommandMessage<ReserveCreditCommand> cm) {
    ReserveCreditCommand cmd = cm.getCommand();
    long customerId = cmd.getCustomerId();
    Customer customer = customerRepository.findOne(customerId);
    try {
      customer.reserveCredit(cmd.getOrderId(), cmd.getOrderTotal());
      return withSuccess(new CustomerCreditReserved());
    } catch (CustomerCreditLimitExceededException e) {
      return withFailure(new CustomerCreditReservationFailed());
    }
  }
```

Route command to handler

Reserve credit

Make reply message

Eventuate Tram Sagas

- Open-source Saga framework
- Currently for Java
- https://github.com/eventuate-tram/eventuate-tram-sagas

Benefits and drawbacks of orchestration

Benefits

- Centralized coordination logic is easier to understand
- Reduced coupling, e.g. Customer knows less
- Reduces cyclic dependencies

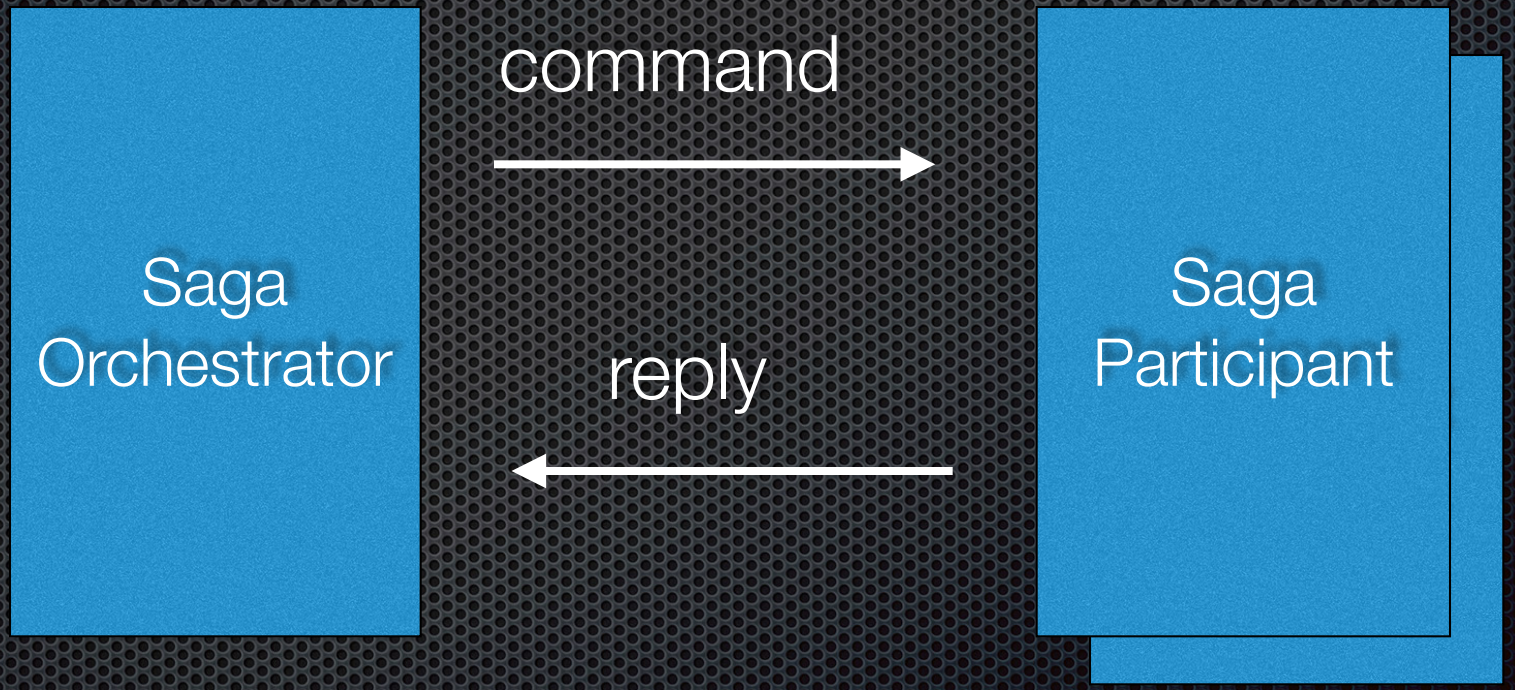
Drawbacks

- Risk of smart sagas directing dumb services

Agenda

- ACID is not an option
- Overview of sagas
- Coordinating sagas
- Sagas and inter-service communication

About Saga orchestrator ⇔ participant communication

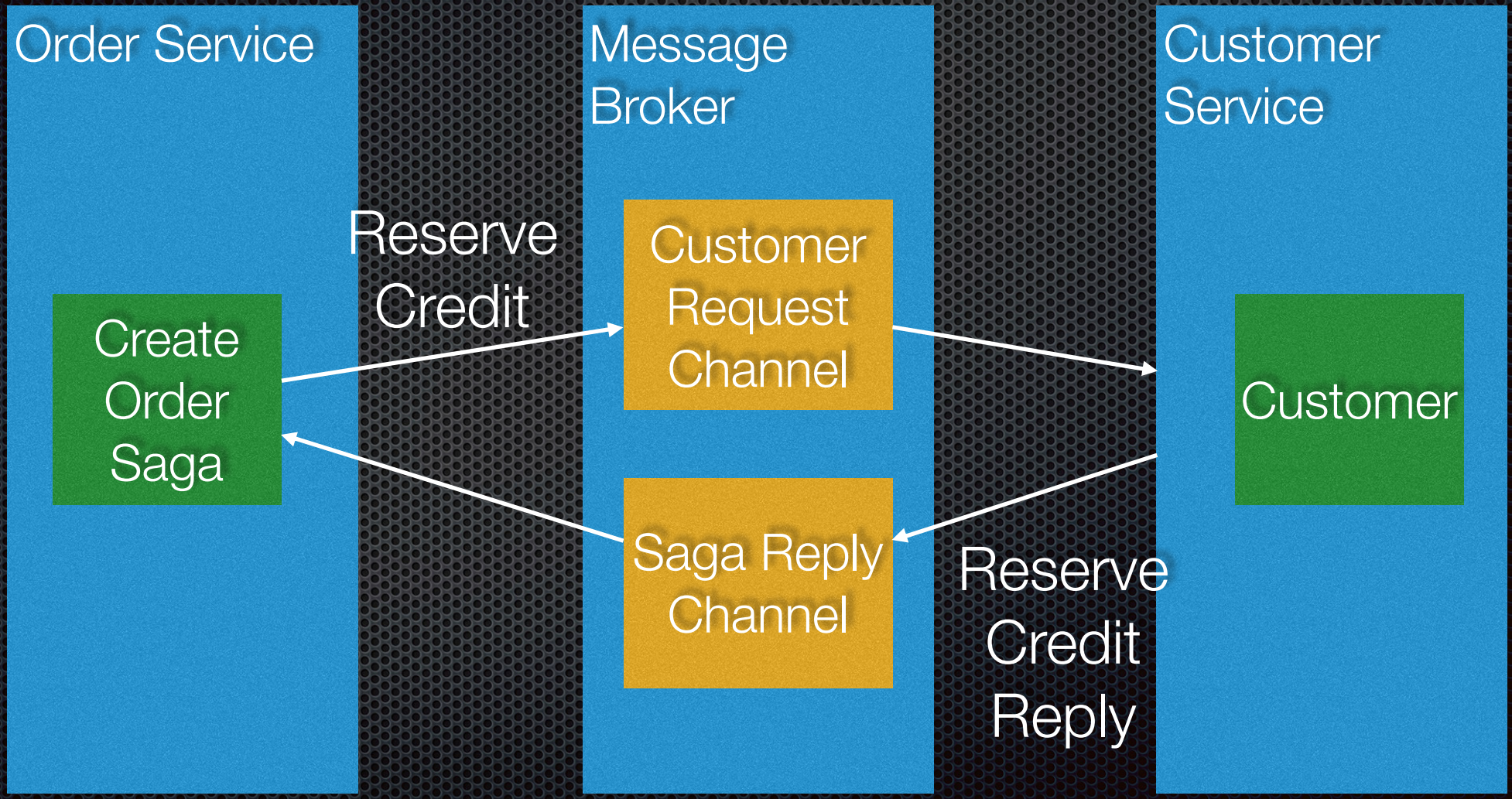


Saga must complete even if there are transient failures

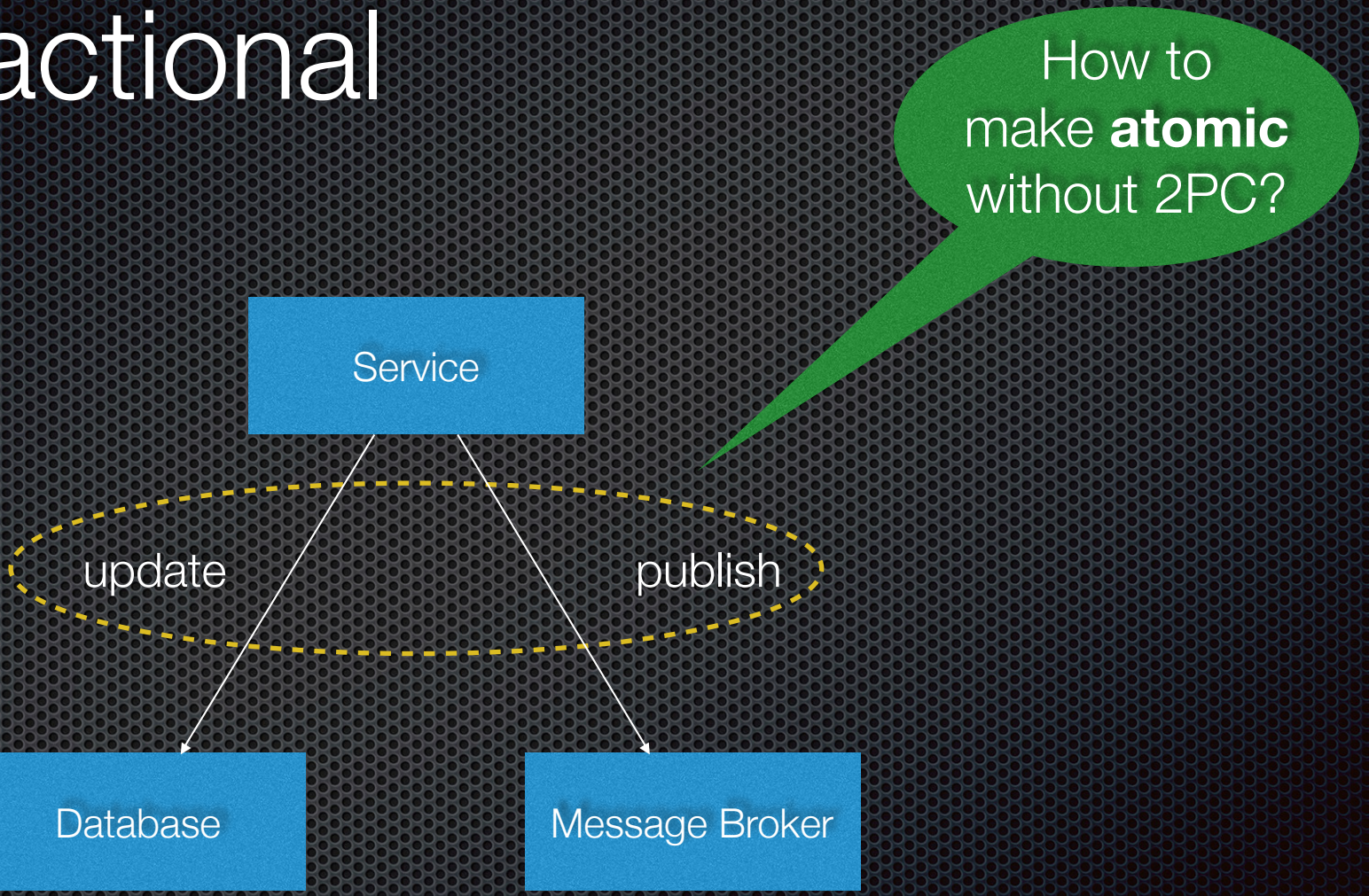
Use asynchronous messaging

Ensures sagas complete when participants are temporarily unavailable

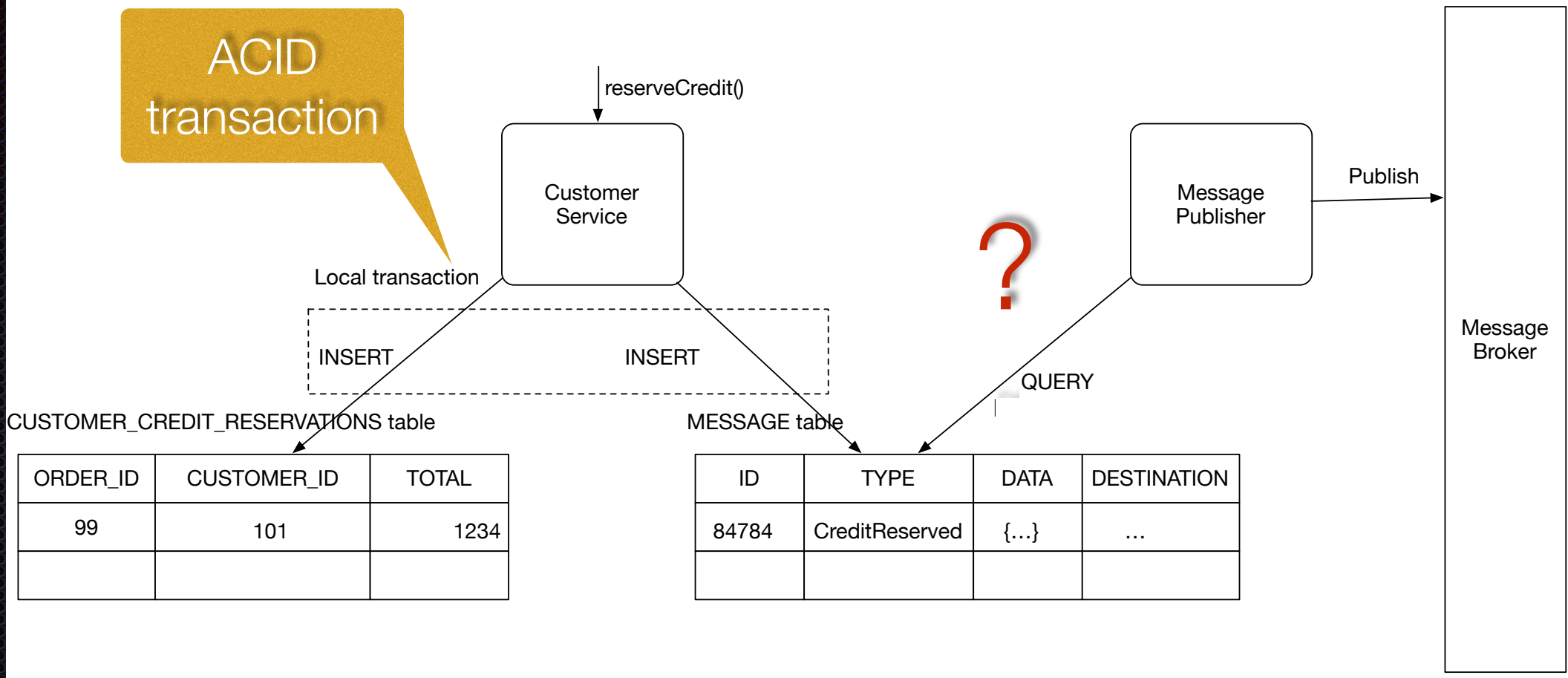
Create Order Saga - messaging



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 (ok)

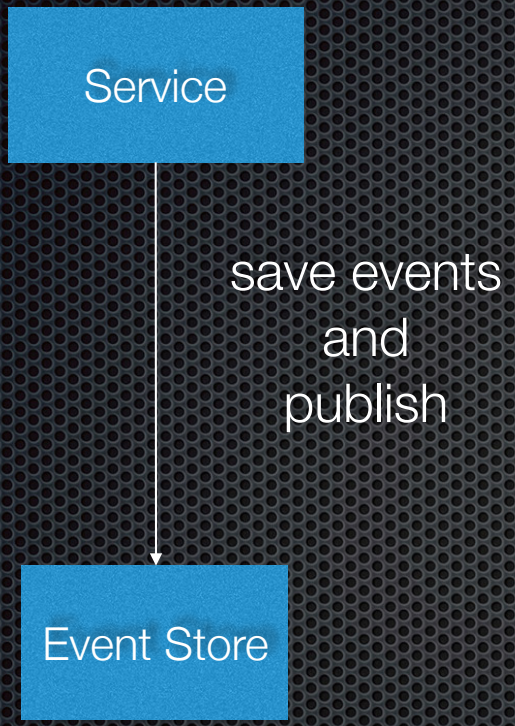
OR

Tail the database transaction log (better)

Eventuate Tram

- Open-source framework for transactional messaging
  - Send and receive messages
  - Publish and subscribe to domain events
  - Send commands and replies
- Currently, for Java
- https://github.com/eventuate-tram/eventuate-tram-core

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