Managing data consistency in a microservice architecture using Sagas

Chris Richardson

Founder of eventuate.io

Author of Microservices Patterns

Founder of the original CloudFoundry.com

Author of POJOs in Action

@crichardson

chris@chrisrichardson.net

http://eventuate.io http://learn.microservices.io

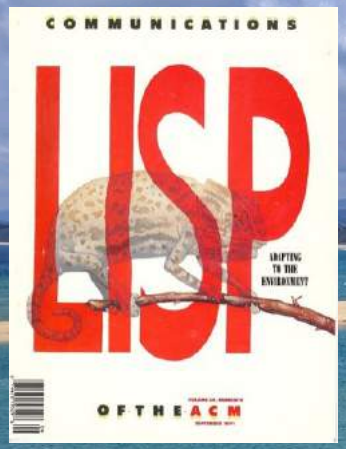
SATURN 2018

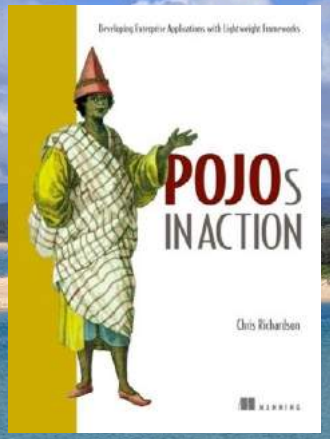
Presentation goal

Distributed data management challenges in a microservice architecture

Sagas as the transaction model

About Chris


COMMUNICATIONS
LISP
OF THE ACM


POJOs
IN ACTION
Chris Richardson


CLOUDFOUNDRY
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


vmware
spring
source

About Chris

Consultant and trainer focusing on modern application architectures including microservices

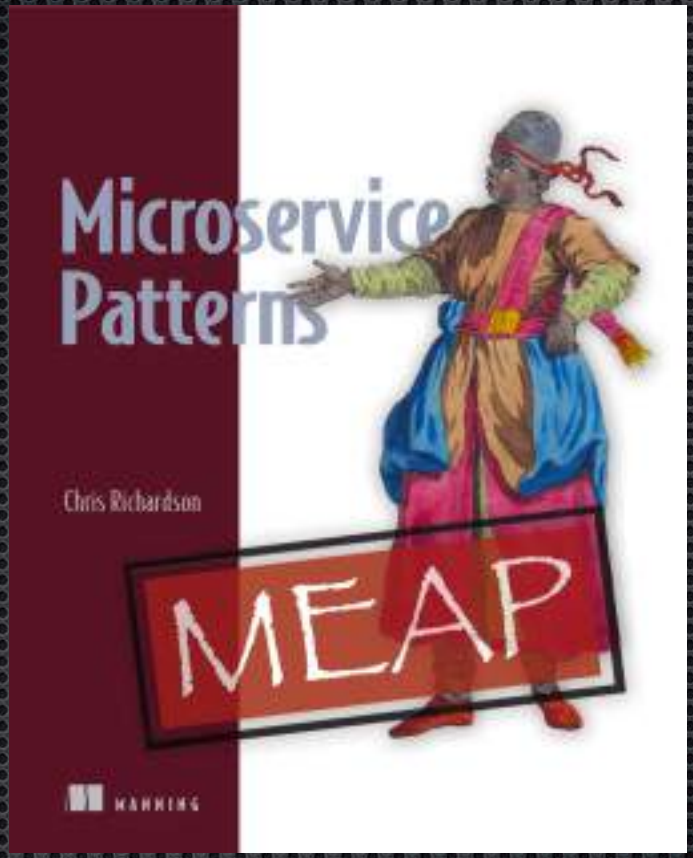
(http://www.chrisrichardson.net/)

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



40%
discount
with code
ctwsaturn18

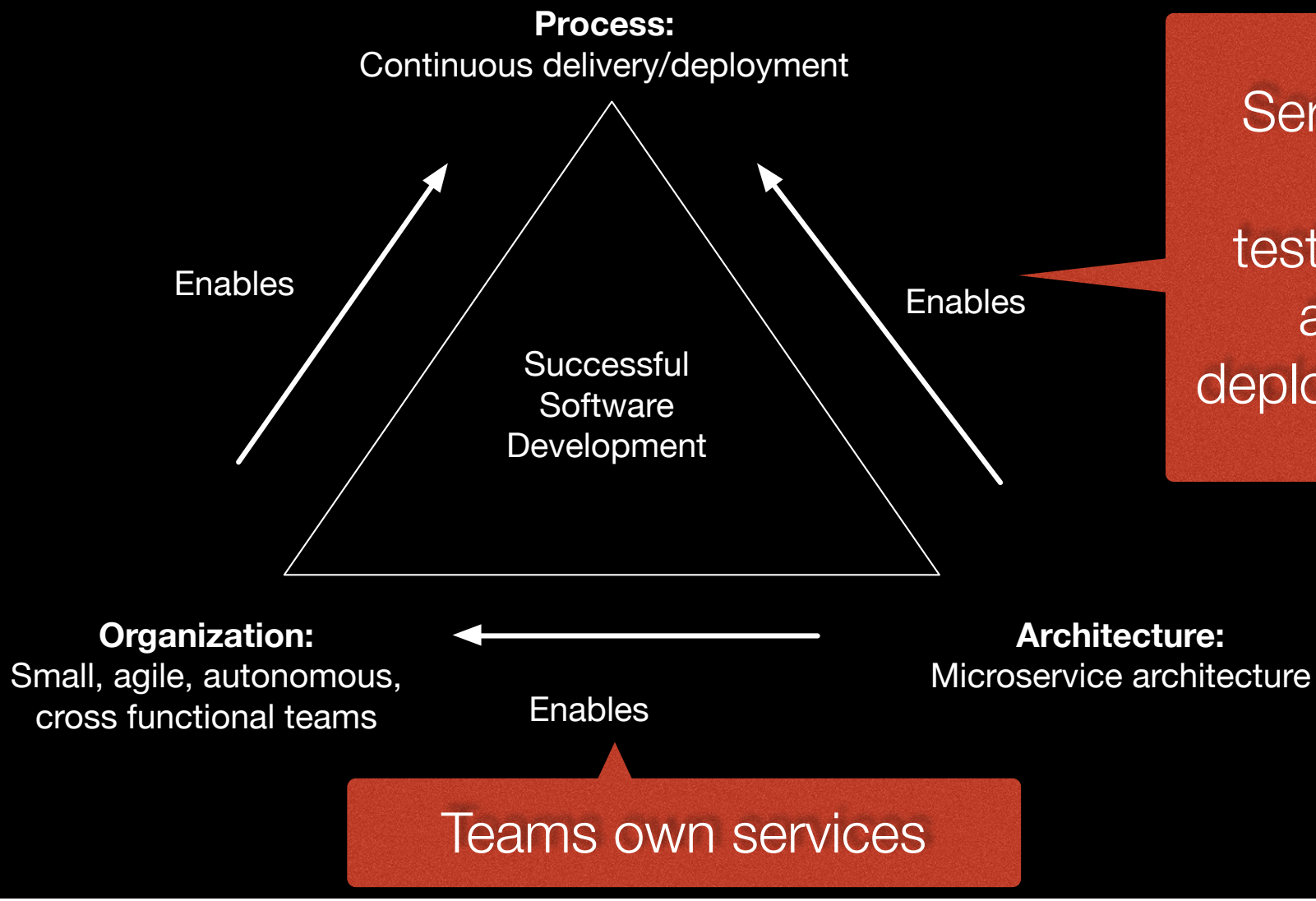
http://learn.microservices.io

Agenda

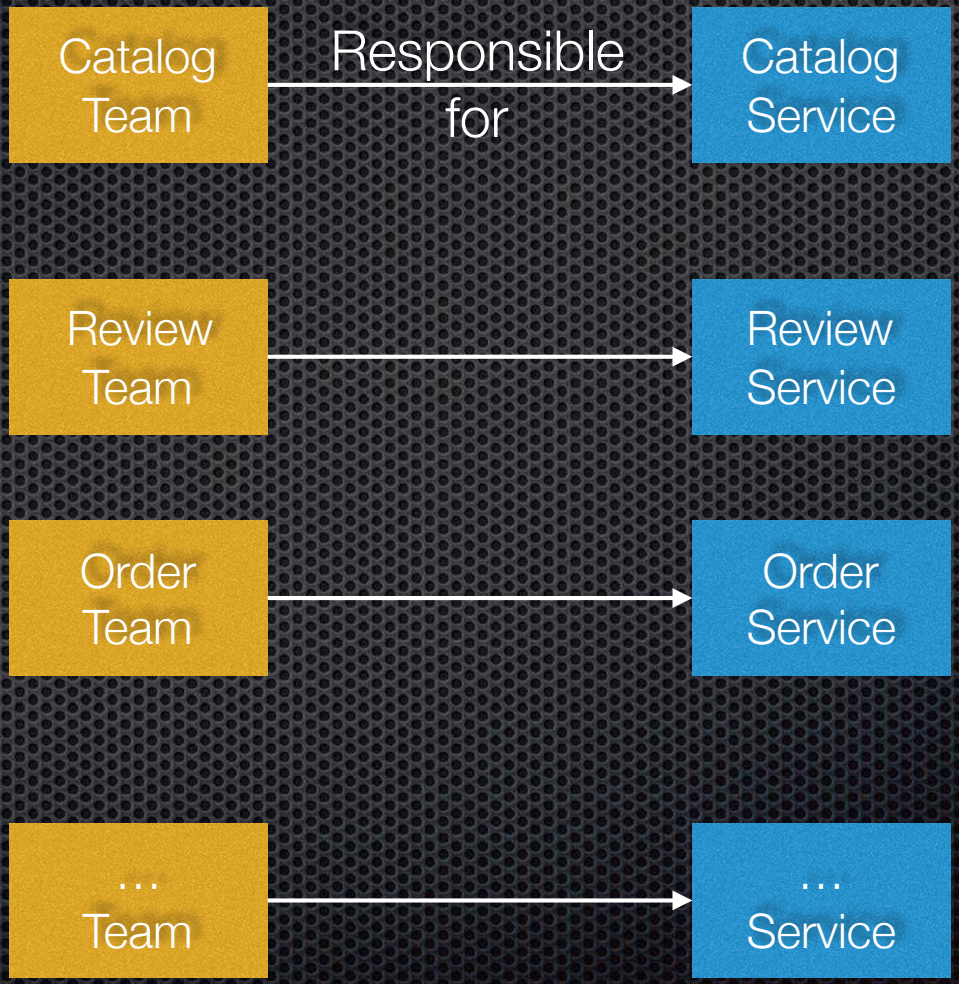
- ACID is not an option
- Overview of sagas
- Coordinating sagas
- Countermeasures for data anomalies
- Using reliable and transactional messaging

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

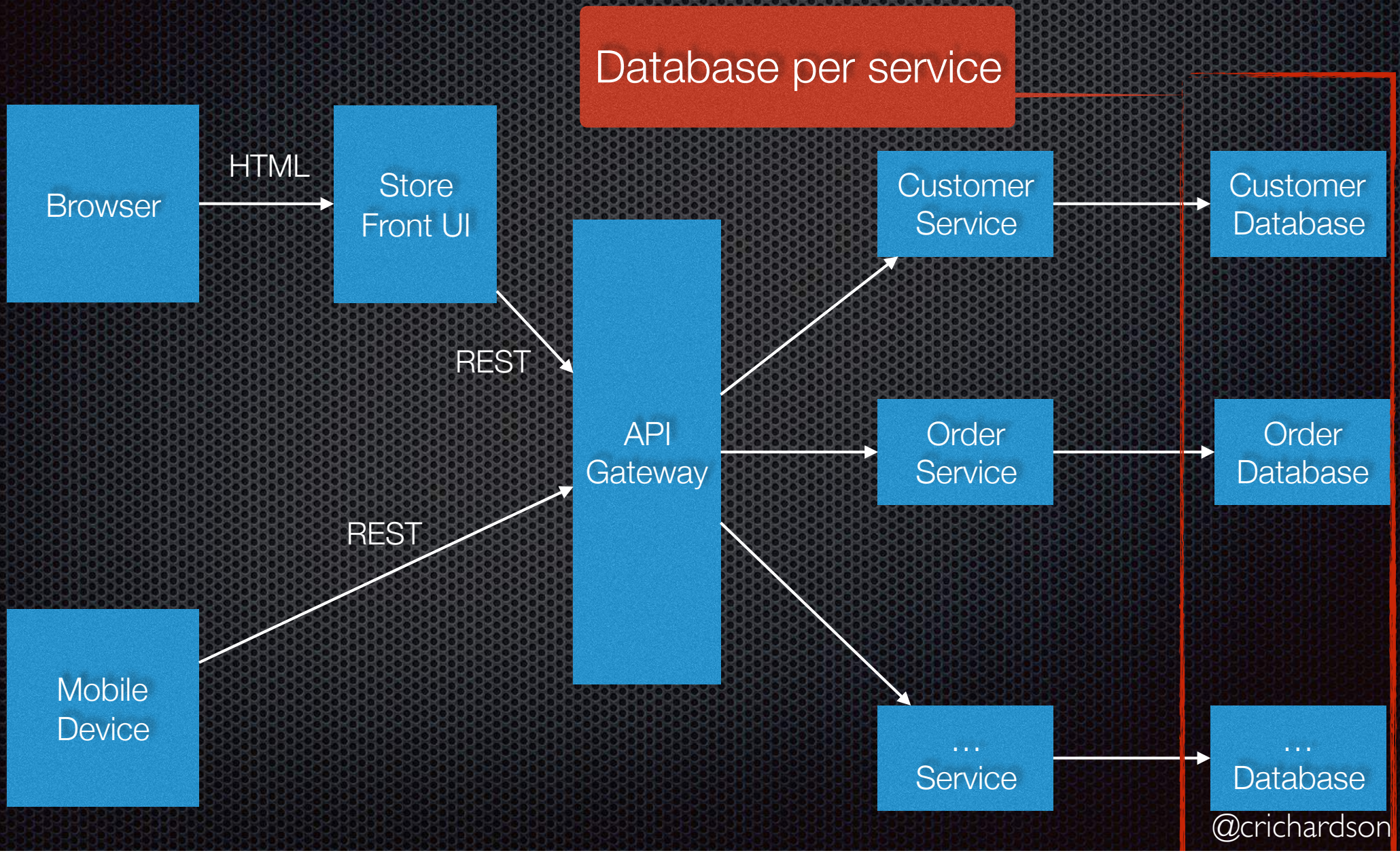
Microservices enable continuous delivery/deployment



Each team owns one or more services



Microservice architecture

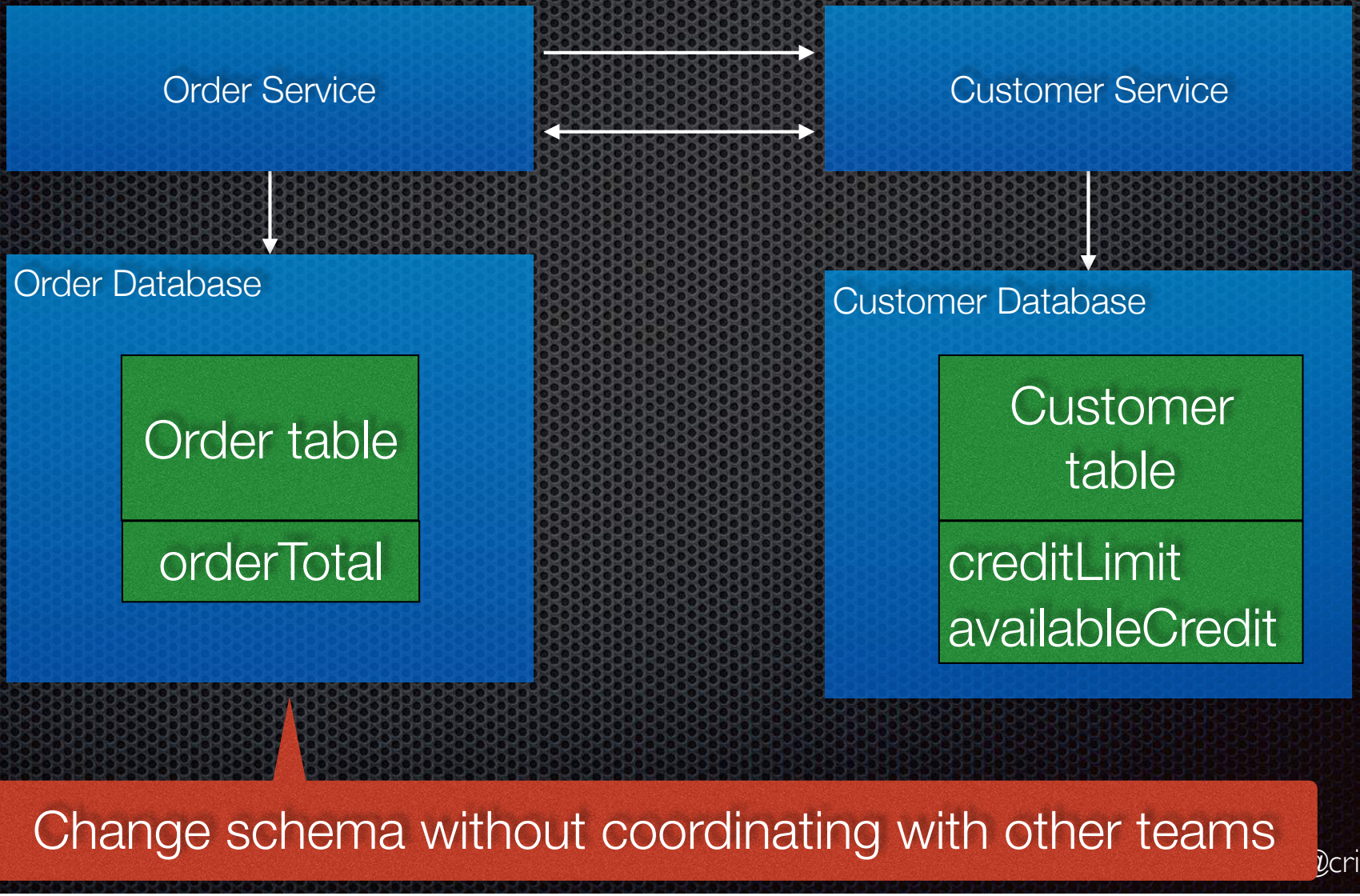


Private database

!=

private database **server**

Loose coupling = encapsulated data



But…

How to maintain data consistency?

How to implement queries?

How to maintain data consistency?

createOrder(customerId, orderTotal)

Pre-conditions:

- customerId is valid

Post-conditions

- Order was created
- Customer.availableCredit -= orderTotal

Spans services

Customer class

Invariant:

availableCredit >= 0
availableCredit <= creditLimit

Cannot use ACID transactions that span services

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
- Countermeasures for data anomalies
- Using reliable and transactional messaging

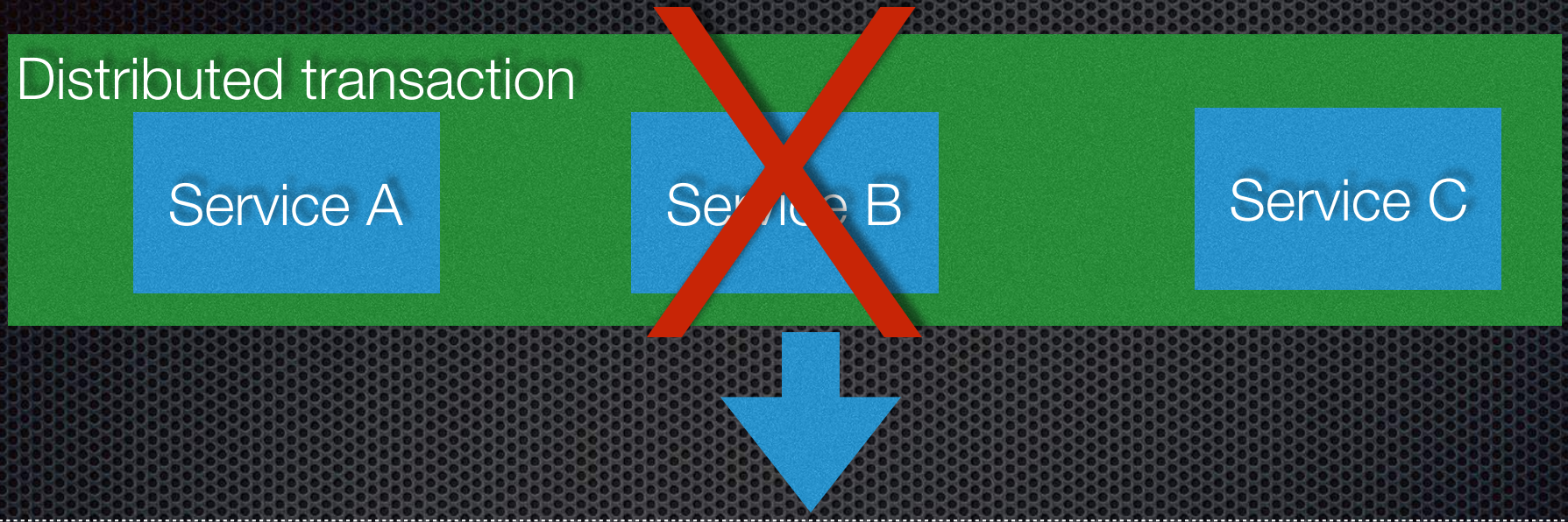
From a 1987 paper

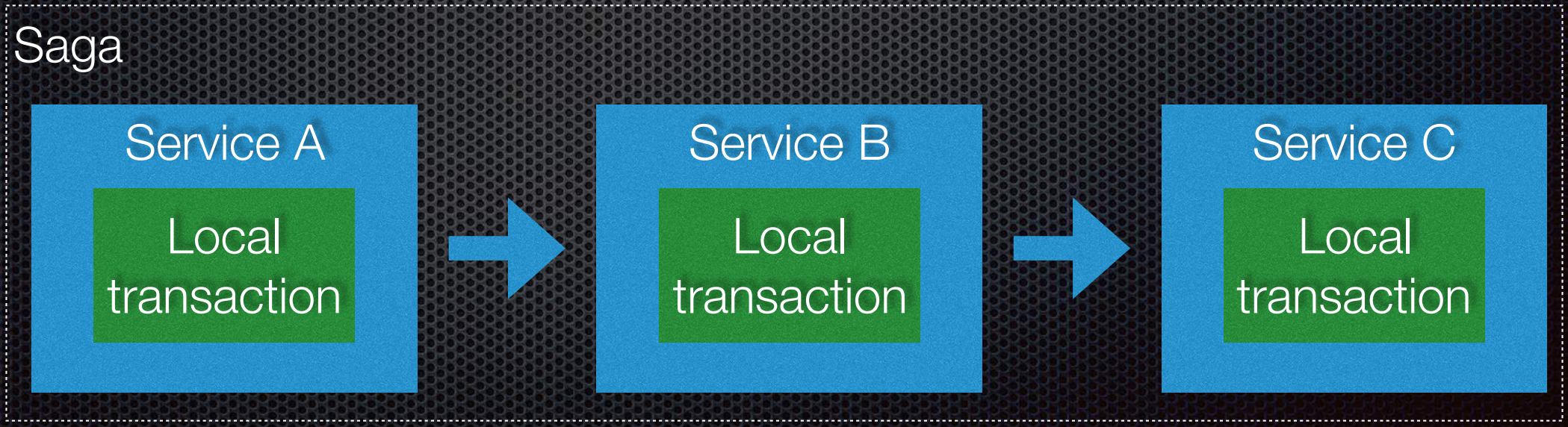
SAGAS

Hector Garcia-Molina
Kenneth Salem

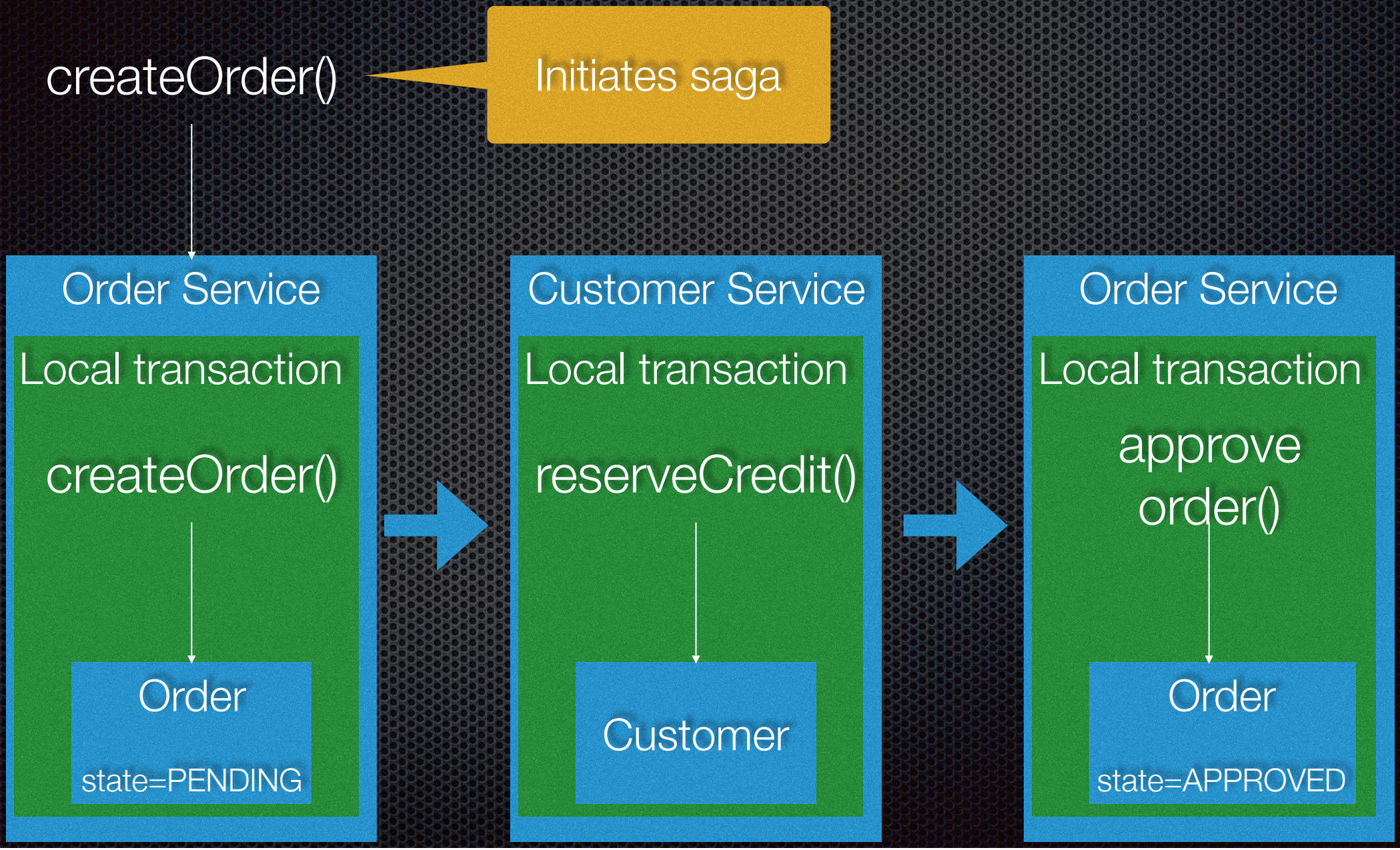
Department of Computer Science
Princeton University
Princeton, N J 08544

Use Sagas instead of 2PC





Create Order Saga



But what about rollback?

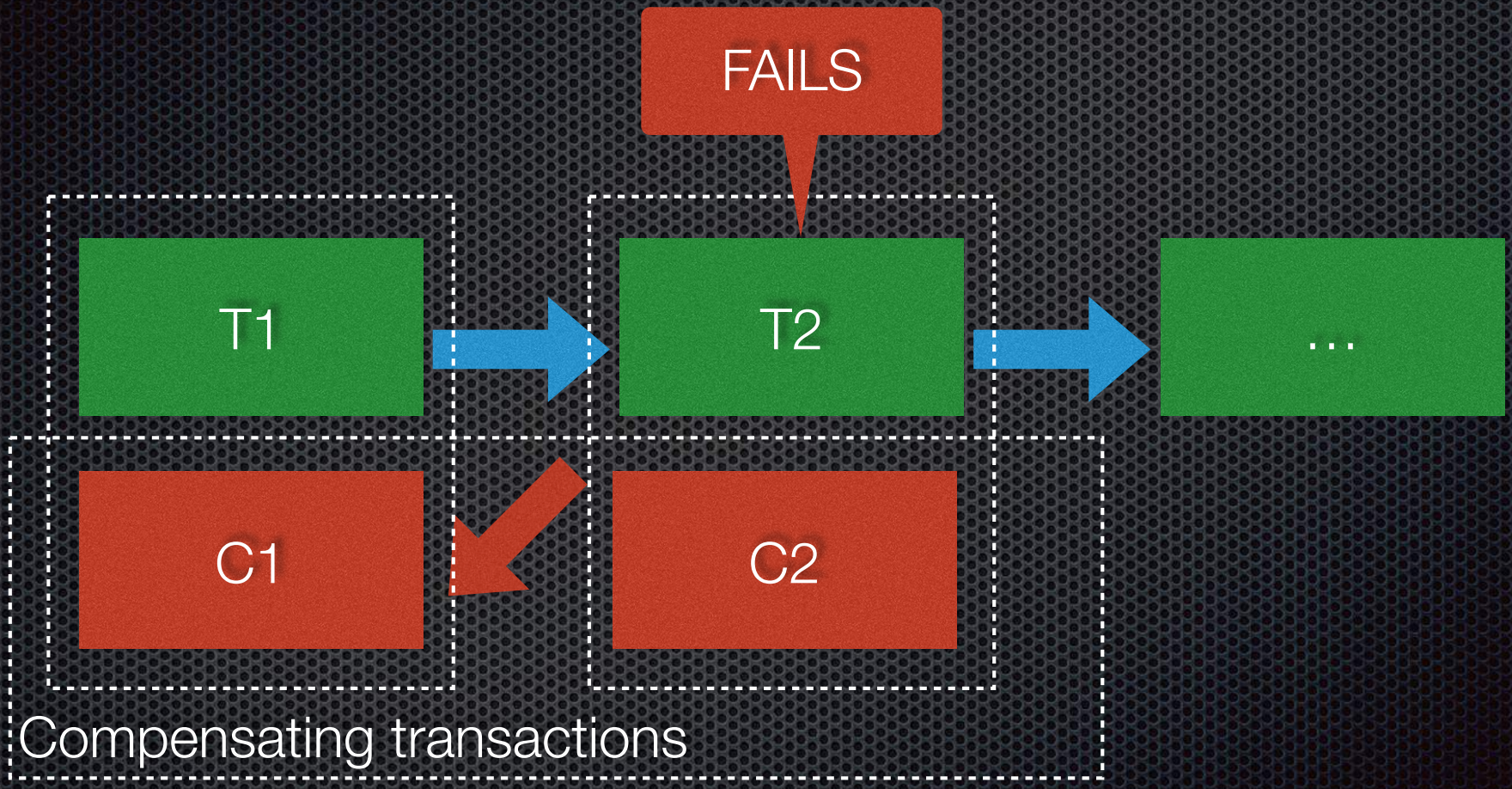
```
BEGIN TRANSACTION
...
UPDATE ...
...
INSERT ....
...
.... BUSINESS RULE VIOLATED!!!!
...
ROLLBACK TRANSACTION
```

Really simple!

Rolling back sagas

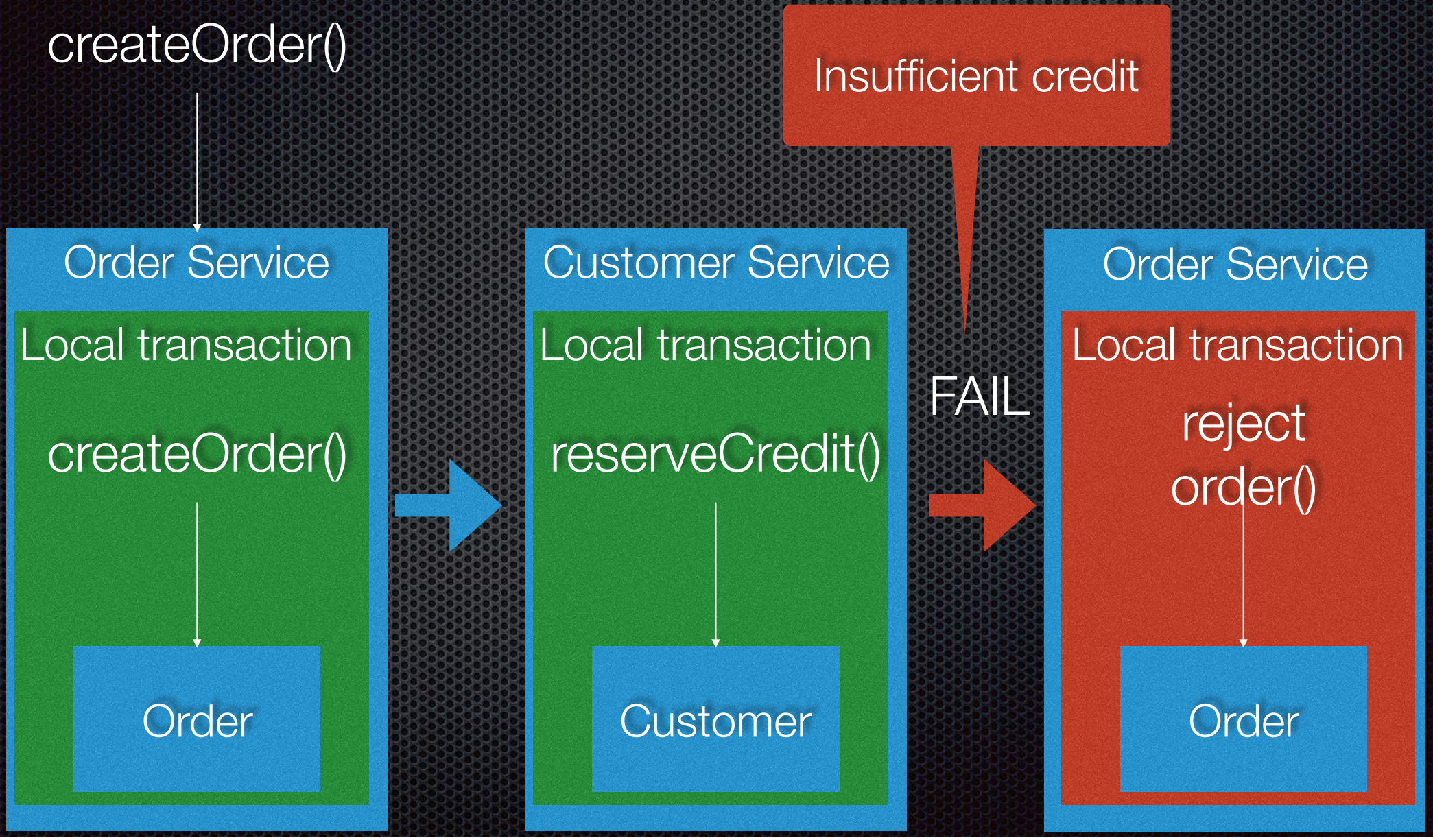
- Use compensating transactions
- Developer must write application logic to “rollback” eventually consistent transactions
- Careful design required!

Saga: Every Ti has a Ci



T1 ⇒ T2 ⇒ C1

Create Order Saga - rollback



Writing compensating transactions isn't always easy

- Write them so they will always succeed
  - If a compensating transaction fails => no clear way to recover
- Challenge: Undoing changes when data has already been changed by a different transaction/saga
  - More on this later

Non-compensatable actions

- For example, sending an email can’t be unsent.
- Move actions that can’t be undone to the end of the saga
- More on this later.

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

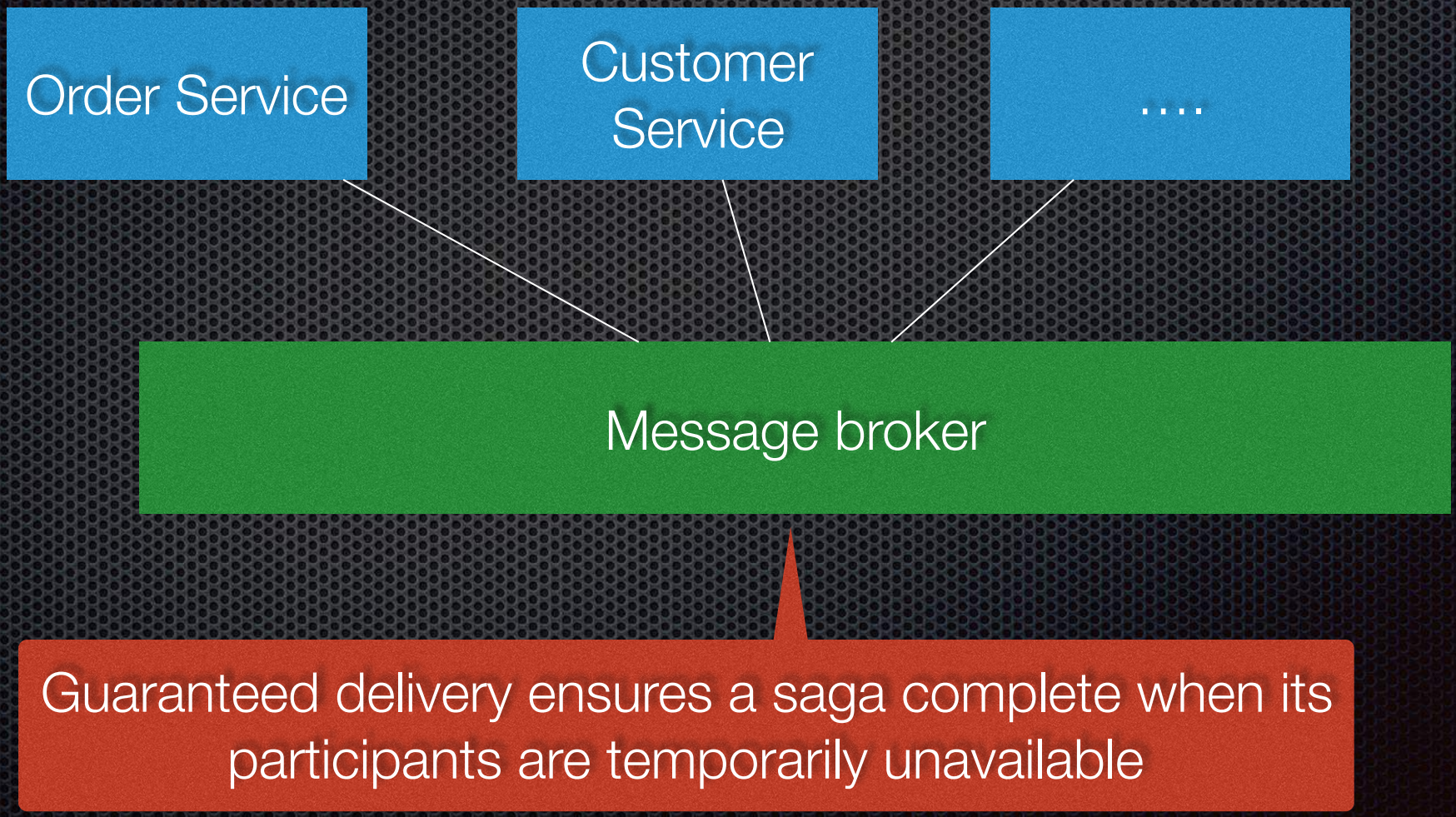
Agenda

- ACID is not an option
- Overview of sagas
- Coordinating sagas
- Countermeasures for data anomalies
- Using reliable and transactional messaging

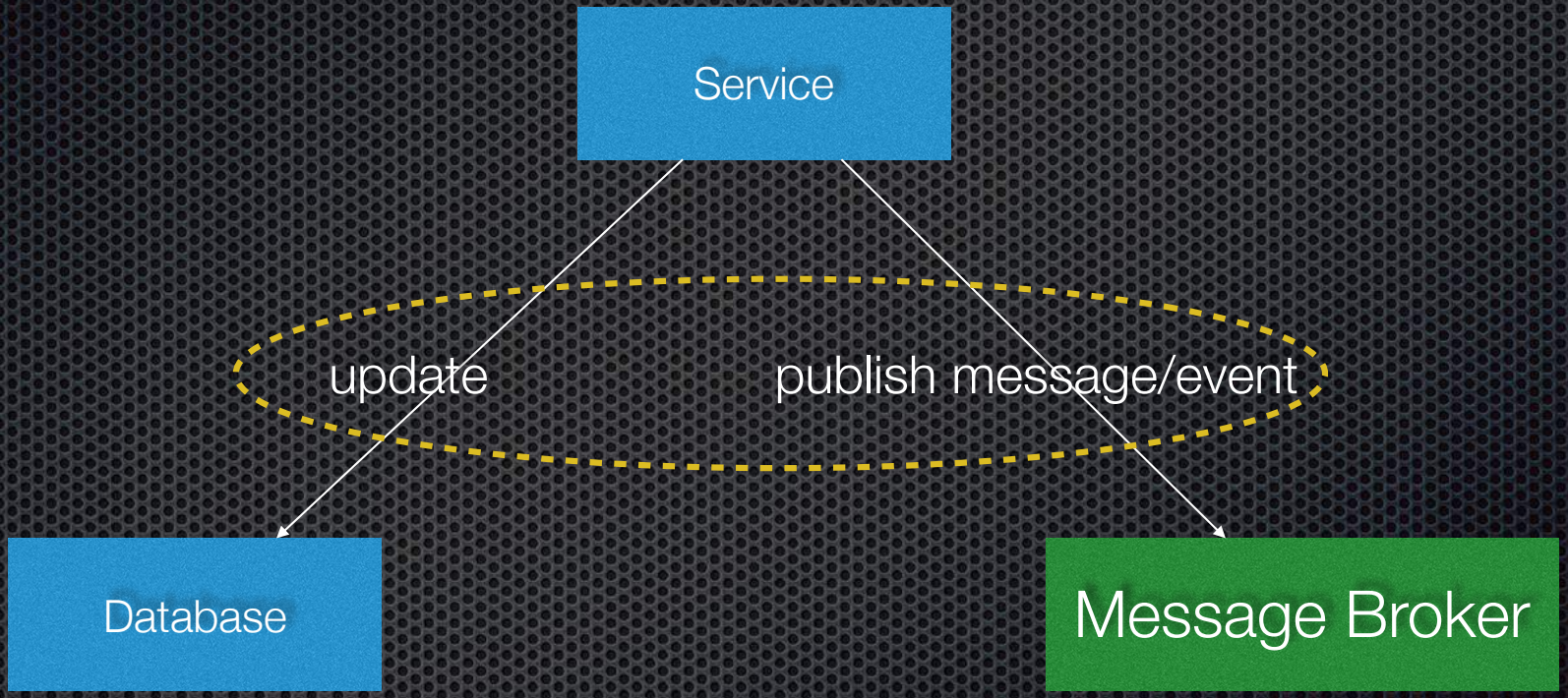
How to sequence the saga transactions?

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

Use asynchronous, broker-based messaging



Saga step = a transaction local to a service



Transactional DB: BEGIN … COMMIT

DDD: Aggregate

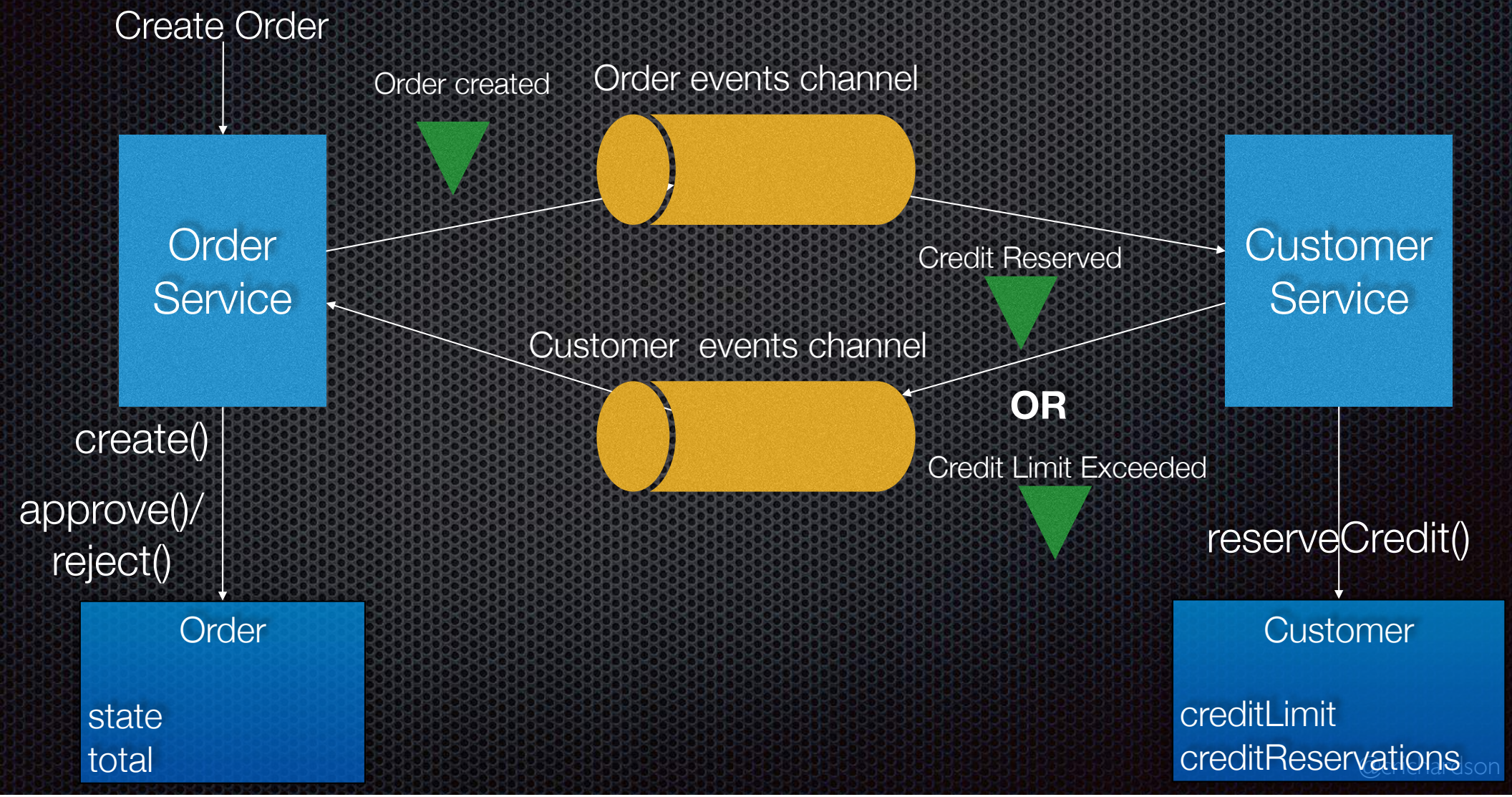
NoSQL: single ‘record’

Choreography: **distributed** decision making

vs.

Orchestration: **centralized** decision making

Option #1: Choreography-based coordination using events



Order Service: publishing domain events

```
public class OrderService {

  @Autowired
  private DomainEventPublisher domainEventPublisher;

  @Autowired
  private OrderRepository orderRepository;

  @Transactional
  public Order createOrder(OrderDetails orderDetails) {
    Order order = Order.createOrder(orderDetails);
    orderRepository.save(order);
    domainEventPublisher.publish(Order.class,
            order.getId(),
            singletonList(new OrderCreatedEvent(order.getId(), orderDetails)));
    return order;
  }
}
```

Create order

Publish event

Customer Service: consuming domain events…

```
public class OrderEventConsumer {

  @Autowired
  private CustomerRepository customerRepository;

  @Autowired
  private DomainEventPublisher domainEventPublisher;

  public DomainEventHandlers domainEventHandlers() {
    return DomainEventHandlersBuilder
            .forAggregateType("io.eventuate.examples.tram.ordersandcustomers.orders.domain.Order")
            .onEvent(OrderCreatedEvent.class, this::orderCreatedEventHandler)
            .build();
  }

  private void orderCreatedEventHandler(DomainEventEnvelope<OrderCreatedEvent> domainEventEnvelope) {...}
}
```

Subscribe to event

https://github.com/eventuate-tram/eventuate-tram-examples-customers-and-orders

Customer Service: consuming domain events

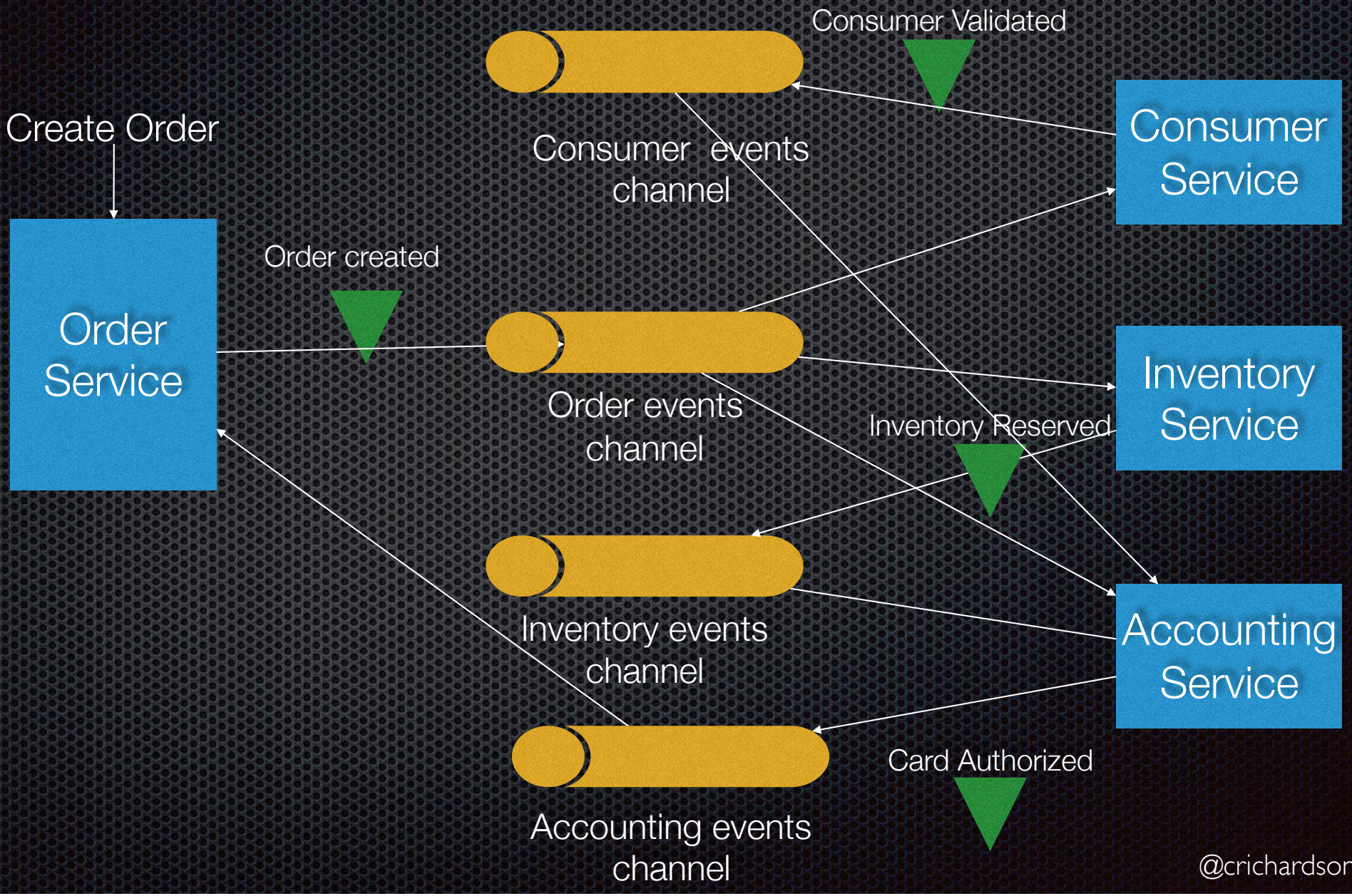
```
private void orderCreatedEventHandler(DomainEventEnvelope<OrderCreatedEvent> domainEventEnvelope) {

  OrderCreatedEvent orderCreatedEvent = domainEventEnvelope.getEvent();

  Customer customer = customerRepository
          .findOne(orderCreatedEvent.getOrderDetails().getCustomerId());

  try {
    customer.reserveCredit(orderCreatedEvent.getOrderId(),
            orderCreatedEvent.getOrderDetails().getOrderTotal());

    CustomerCreditReservedEvent customerCreditReservedEvent =
            new CustomerCreditReservedEvent(orderCreatedEvent.getOrderId());

    domainEventPublisher.publish(Customer.class,
            customer.getId(),
            Collections.singletonList(customerCreditReservedEvent));

  } catch (CustomerCreditLimitExceededException e) {

    CustomerCreditReservationFailedEvent customerCreditReservationFailedEvent =
            new CustomerCreditReservationFailedEvent(orderCreatedEvent.getOrderId());

    domainEventPublisher.publish(Customer.class,
            customer.getId(),
            Collections.singletonList(customerCreditReservationFailedEvent));
  }
}
```

Attempt to Reserve credit

Publish event on success

Publish event on failure

More complex choreography example



Benefits and drawbacks of choreography

Benefits

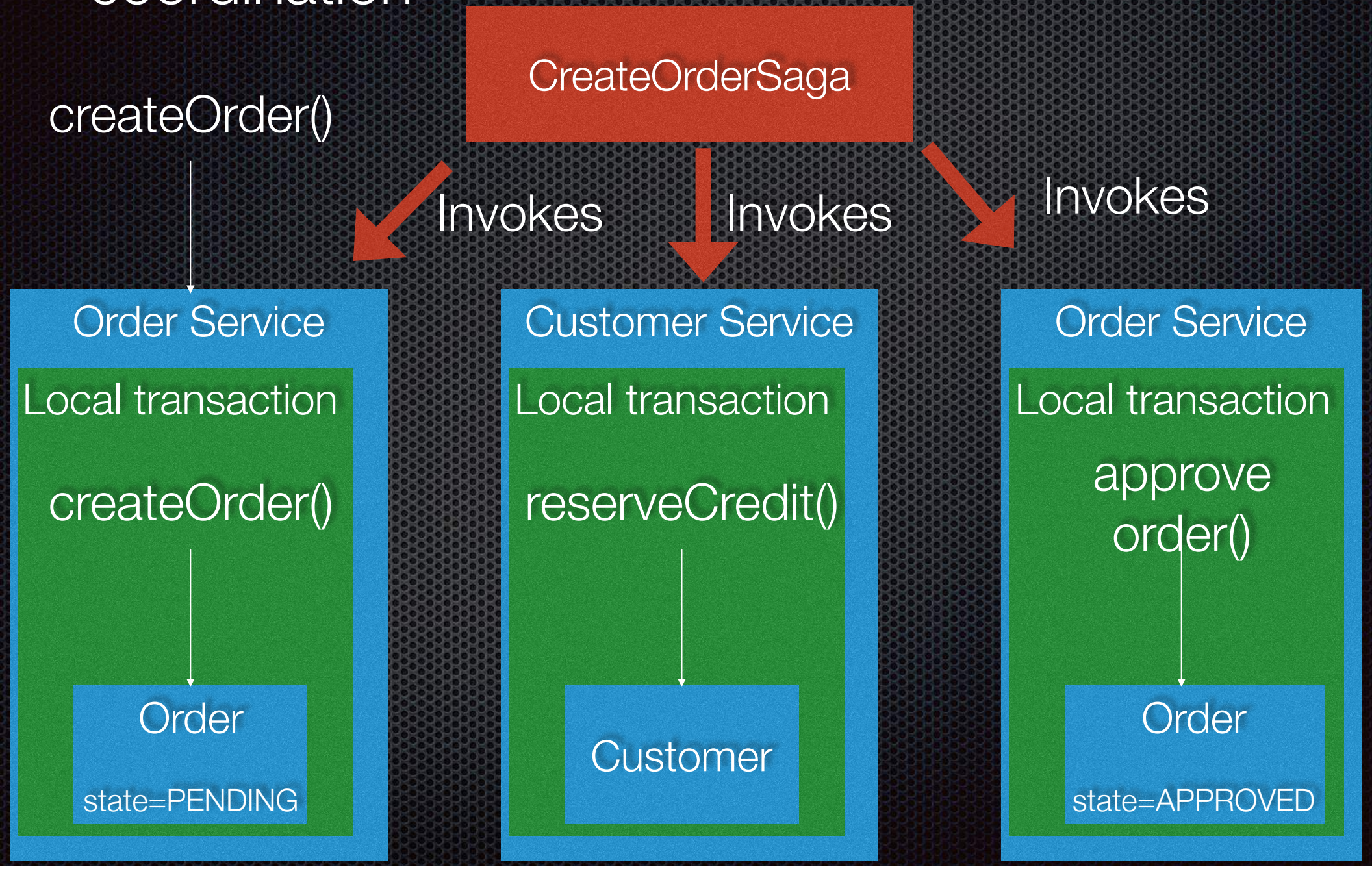
- Simple, especially when using event sourcing
- Participants are loosely coupled

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

https://github.com/eventuate-examples/eventuate-examples-java-customers-and-orders

Option #2: Orchestration-based saga coordination

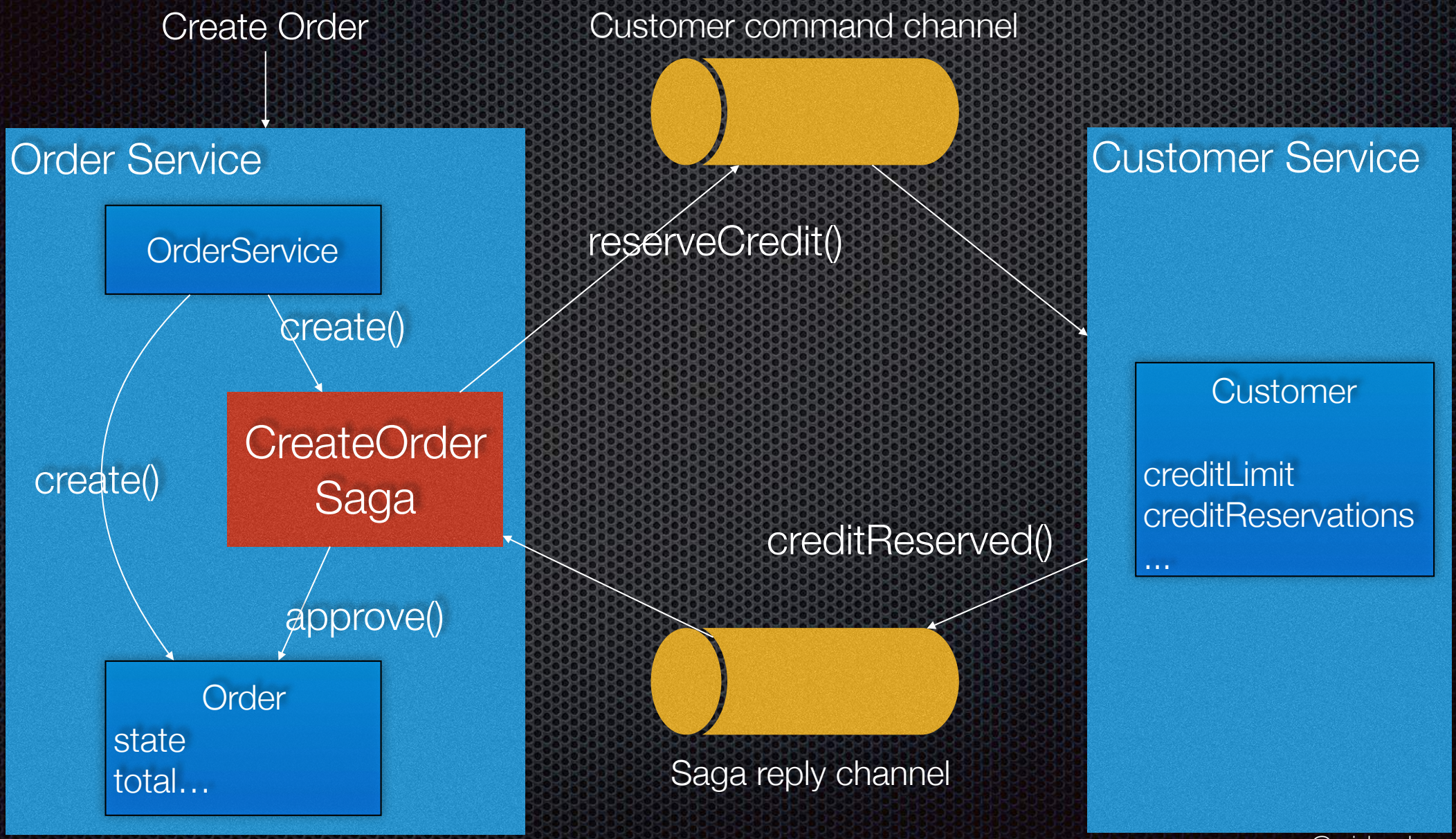


A saga (orchestrator)
is a **persistent object**
that
tracks the state of the saga
and
invokes the participants

Saga orchestrator behavior

- On create:
  - Invokes a saga participant
  - Persists state in database
  - Wait for a reply
- On reply:
  - Load state from database
  - Determine which saga participant to invoke next
  - Invokes saga participant
  - Updates its state
  - Persists updated state
  - Wait for a reply
  - …

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 orchestration framework
- Currently for Java
- https://github.com/eventuate-tram/eventuate-tram-sagas
- https://github.com/eventuate-tram/eventuate-tram-sagas-examples-customers-and-orders

Benefits and drawbacks of orchestration

Benefits

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

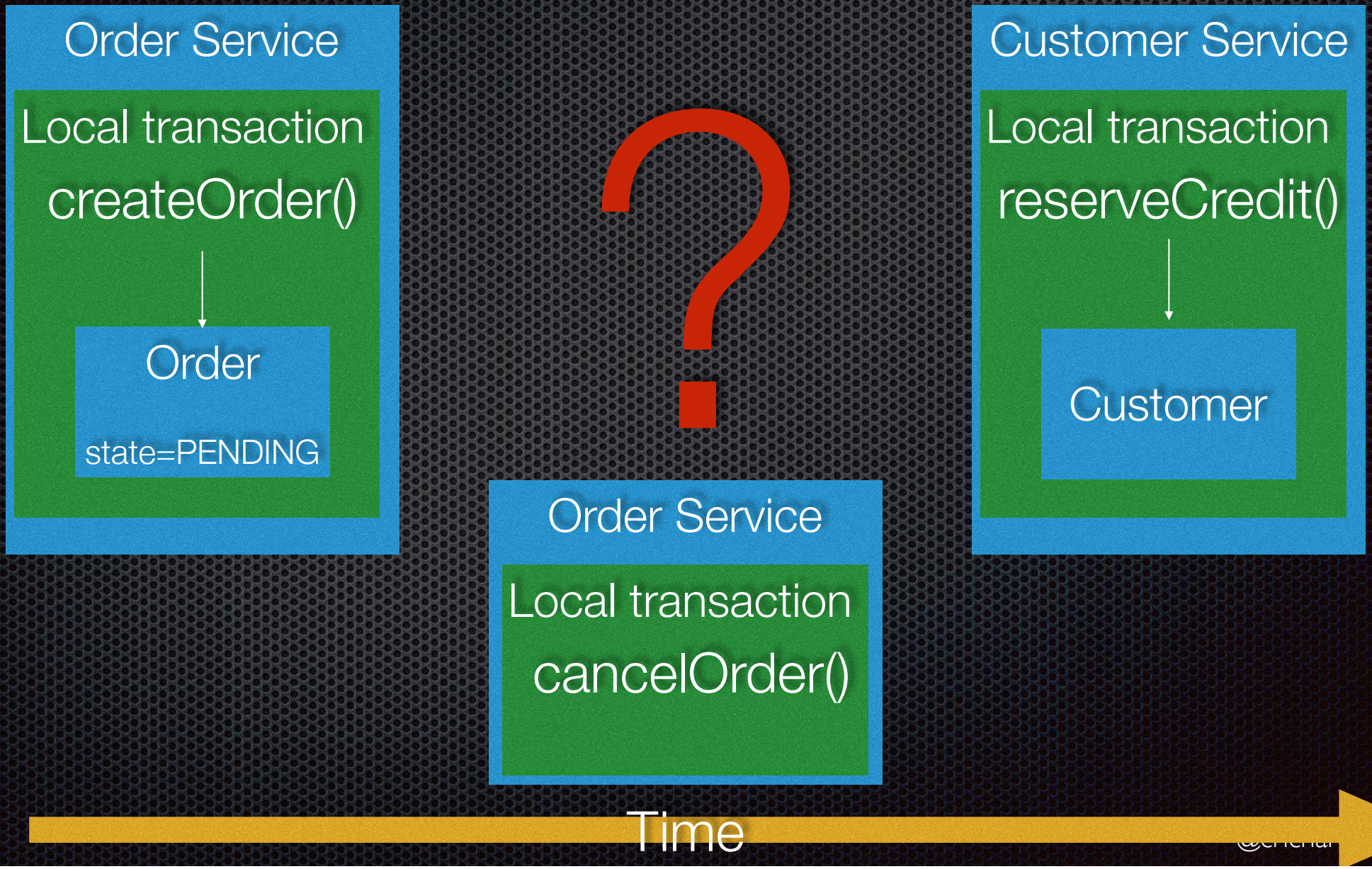
Drawbacks

- Risk of smart sagas directing dumb services

Agenda

- ACID is not an option
- Overview of sagas
- Coordinating sagas
- Countermeasures for data anomalies
- Using reliable and transactional messaging

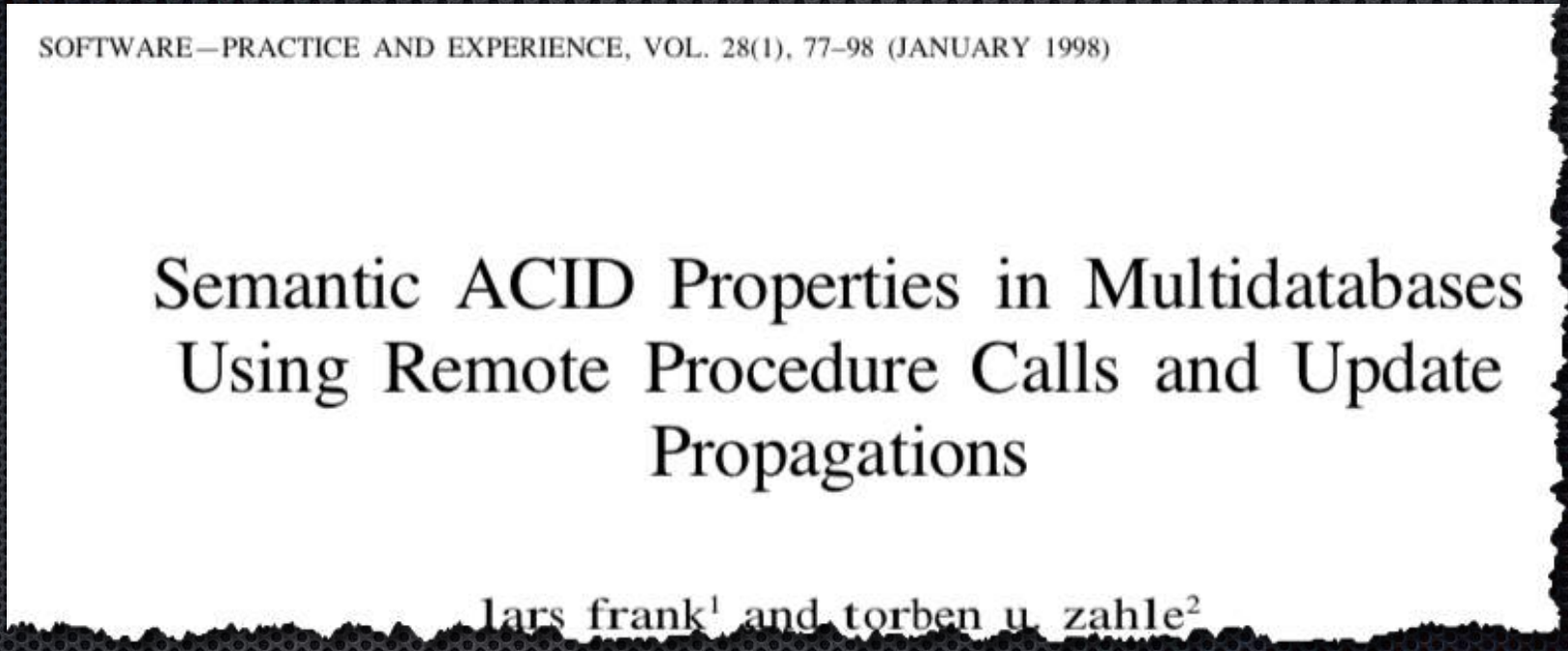
Lack of isolation ⇒ complicates business logic



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]

http://bit.ly/semantic-acid-ctm - paywall 😢

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

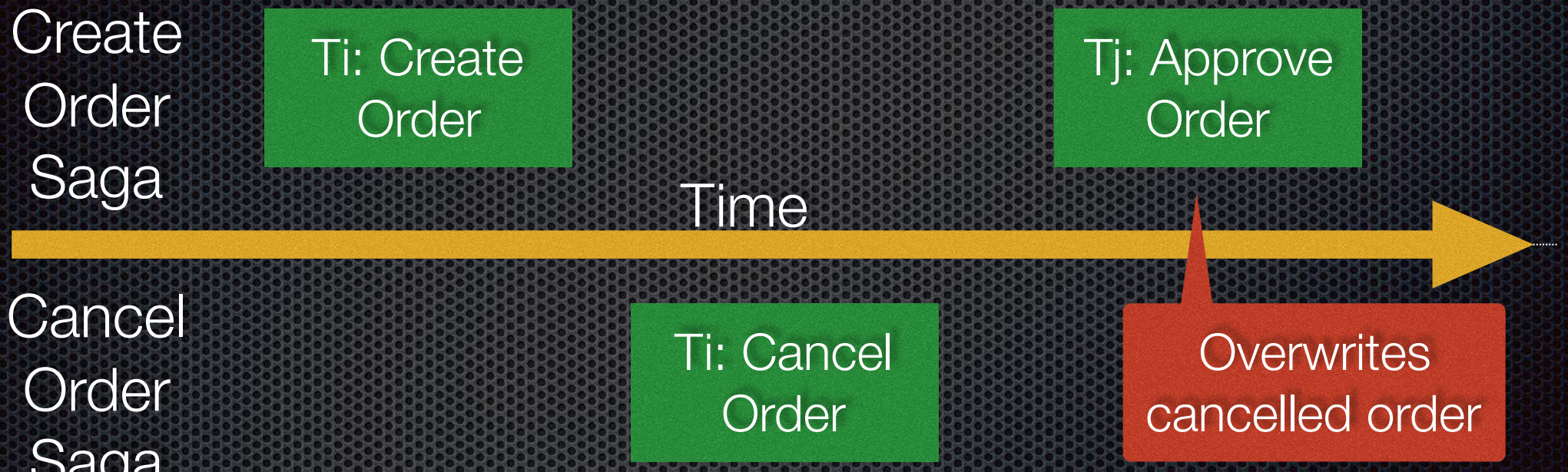
Outcome of
concurrent execution
!=
a sequential execution

Sounds scary

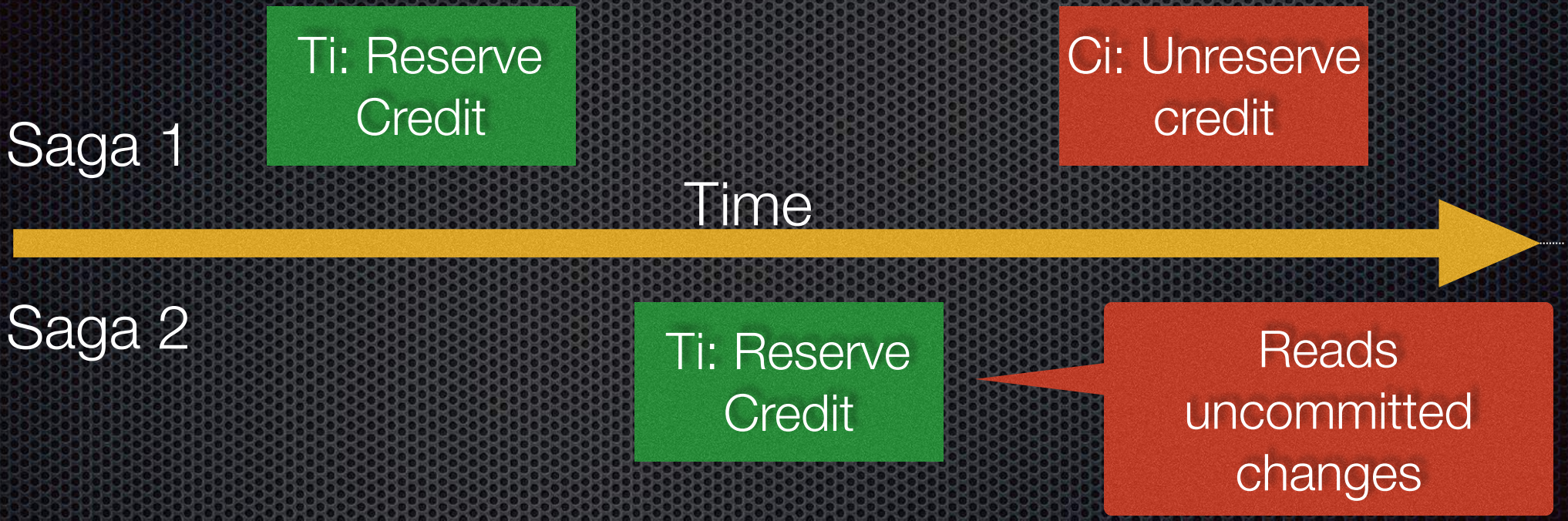
BUT

It's common to relax isolation to improve performance

Anomaly: Lost update

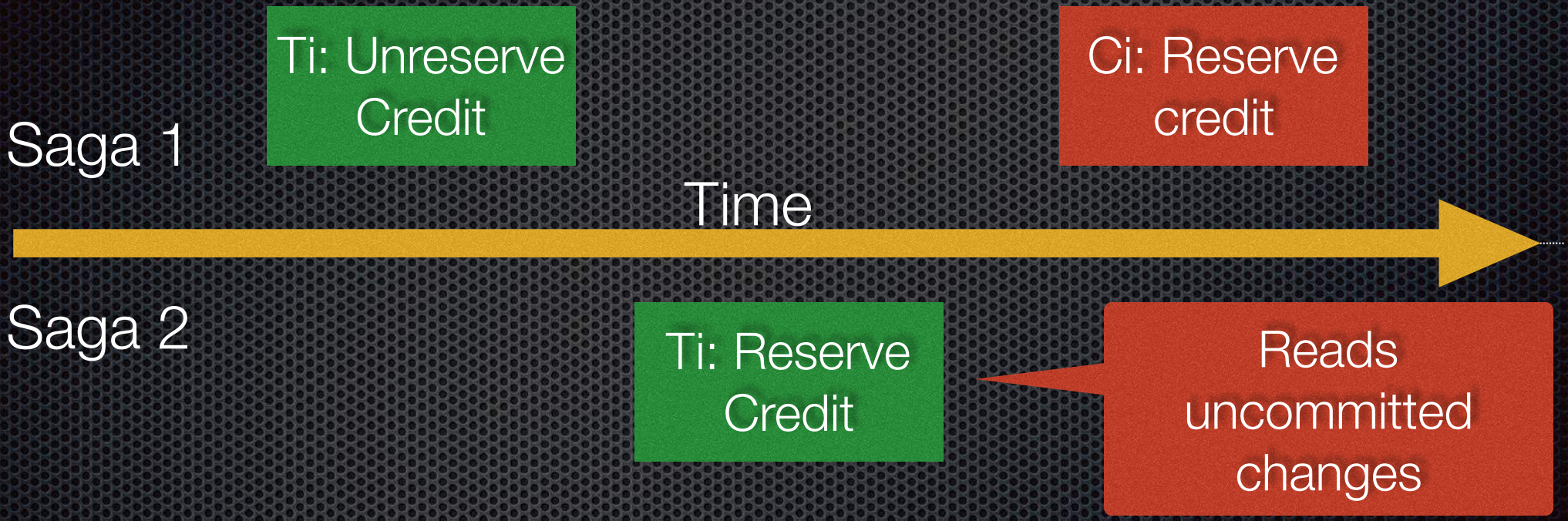


Anomaly: Dirty reads …



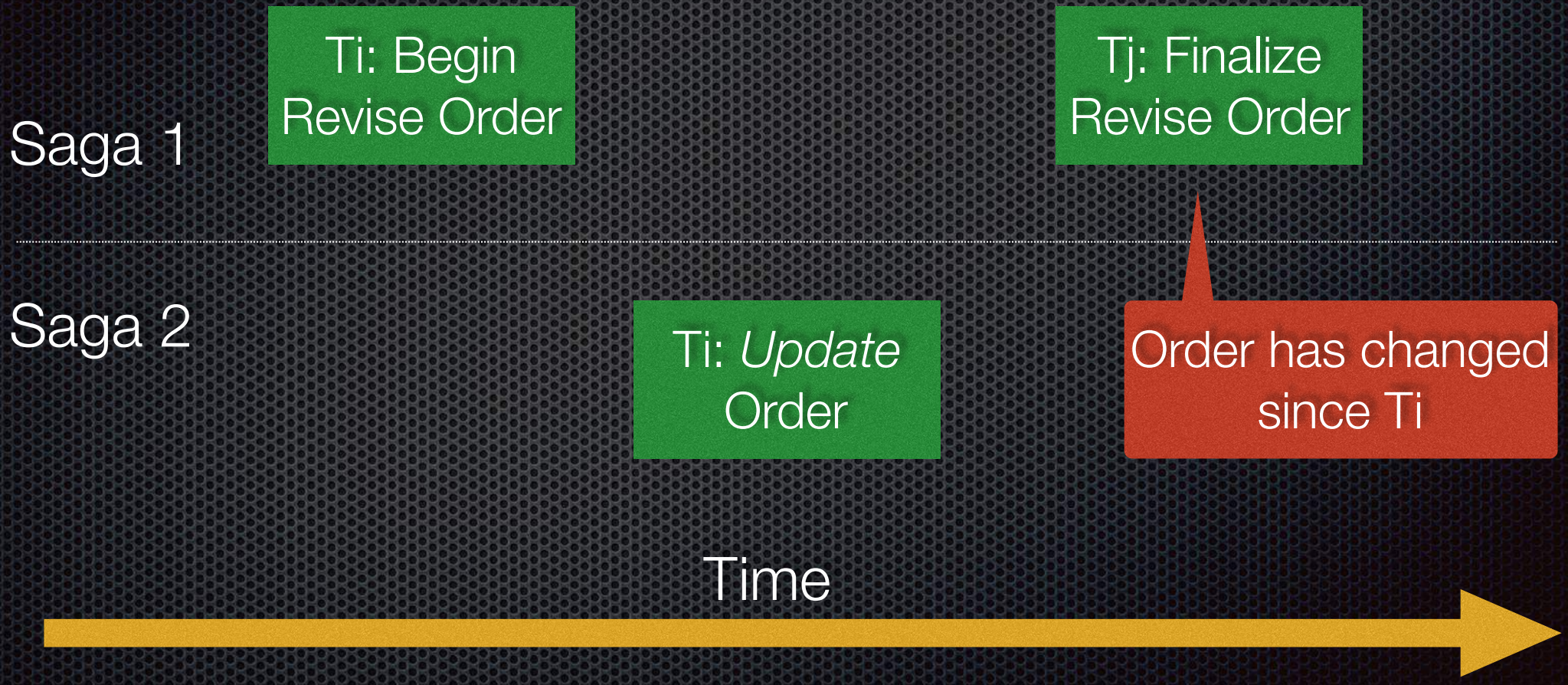
Order rejected unnecessarily

… Anomaly: Dirty reads



Credit limit exceeded!

Anomaly: non-repeatable/ fuzzy read

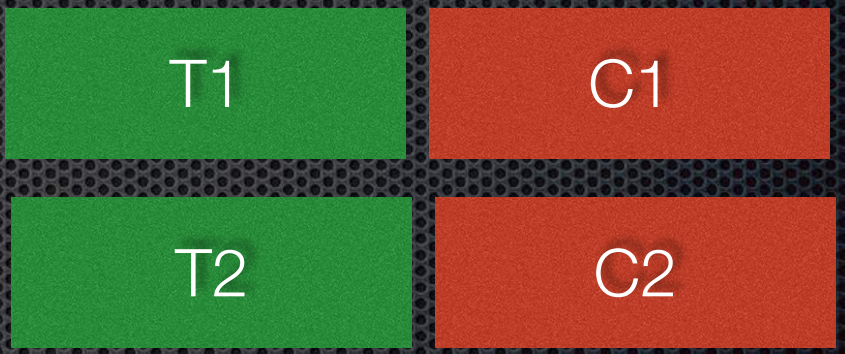


Countermeasures for **reducing*** impact of isolation anomalies…

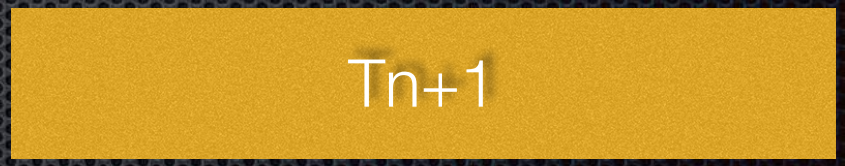
i.e. good enough, eventually consistent application

Saga structure

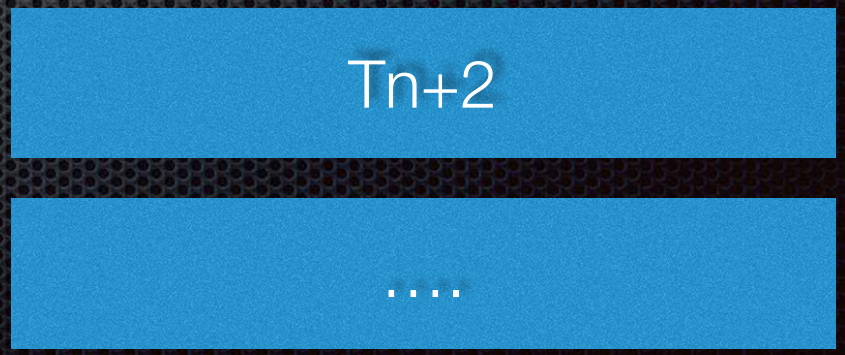
Compensatable transactions



Pivot transaction = GO/NO GO

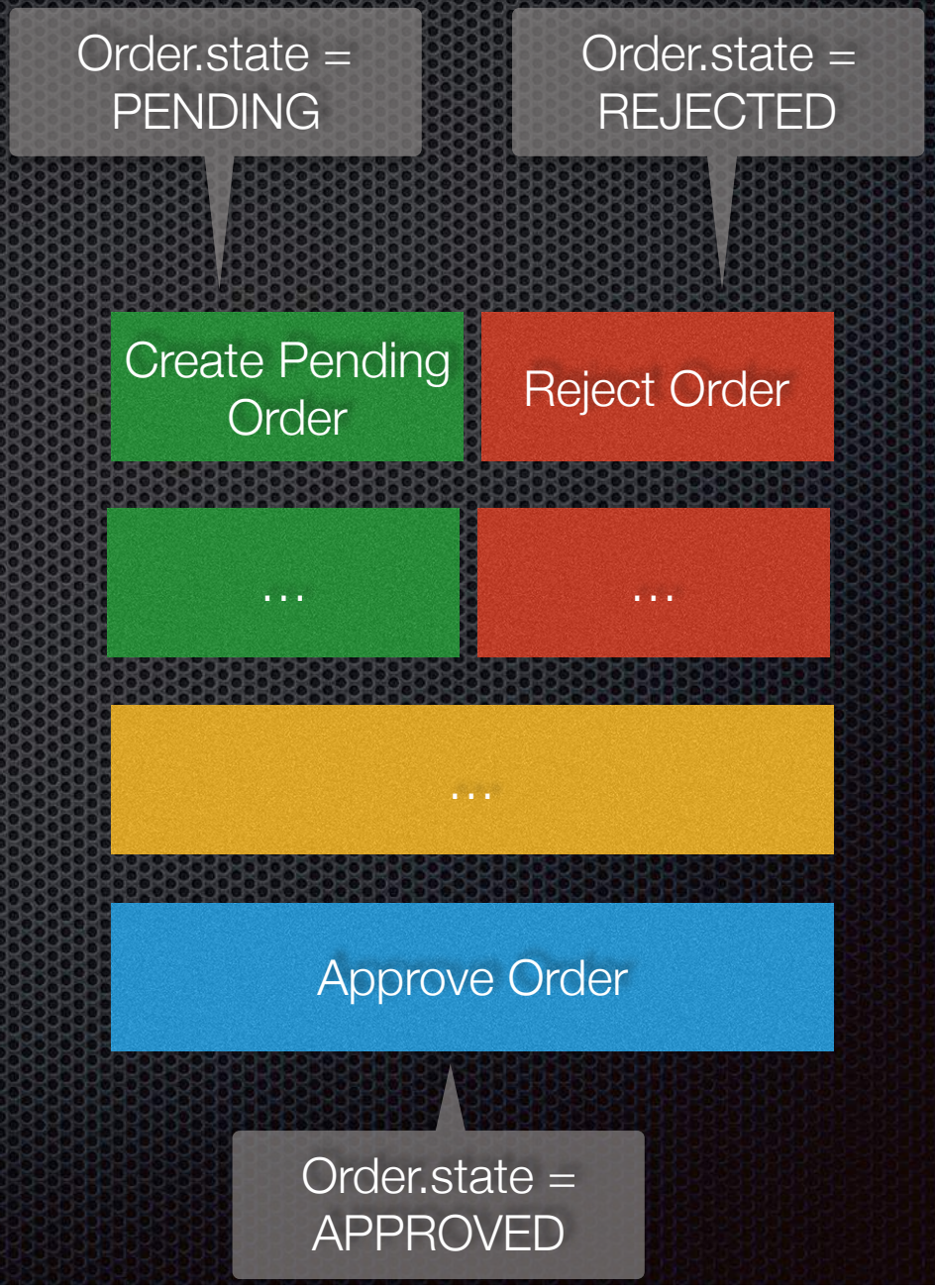


Retriable transactions that can’t fail



Countermeasure: Semantic lock

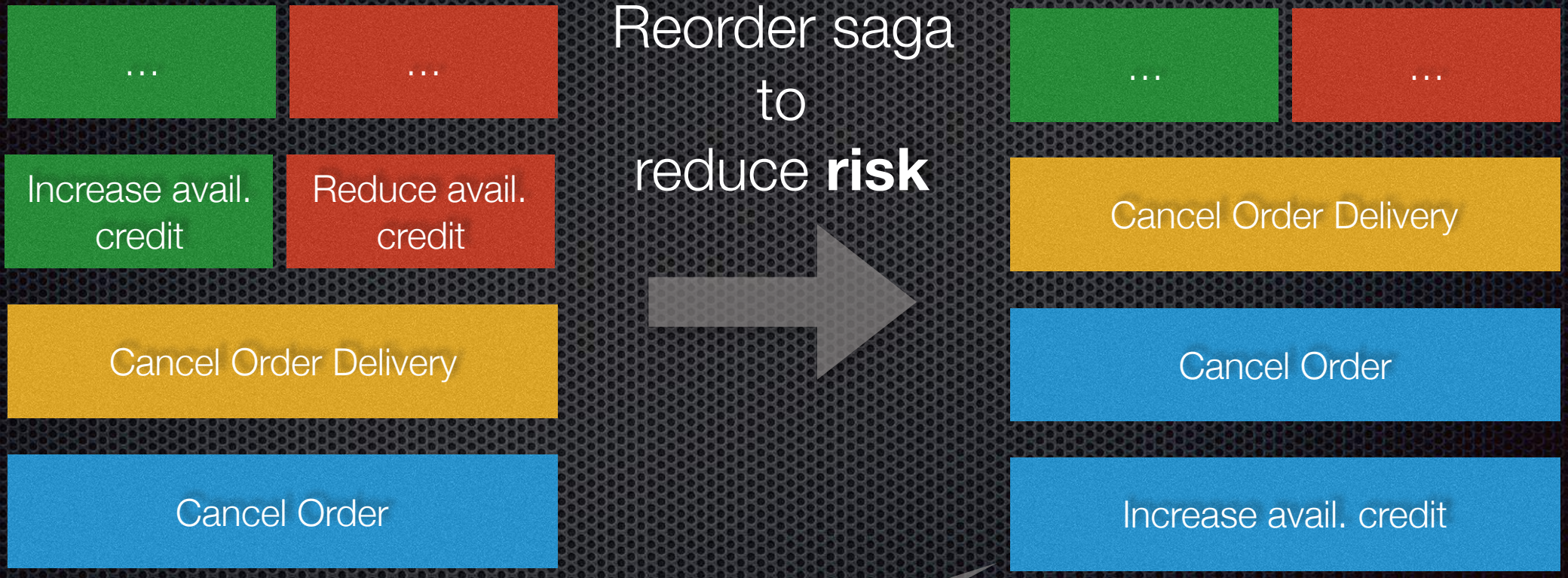
- Compensatable transaction sets flag, retriable transaction releases it
- Indicates a possible dirty read
- Flag = lock:
  - prevents other transactions from accessing it
  - Require deadlock detection, e.g. timeout
- Flag = warning - treat the data differently, e.g.
  - Order.state = PENDING
  - a pending deposit



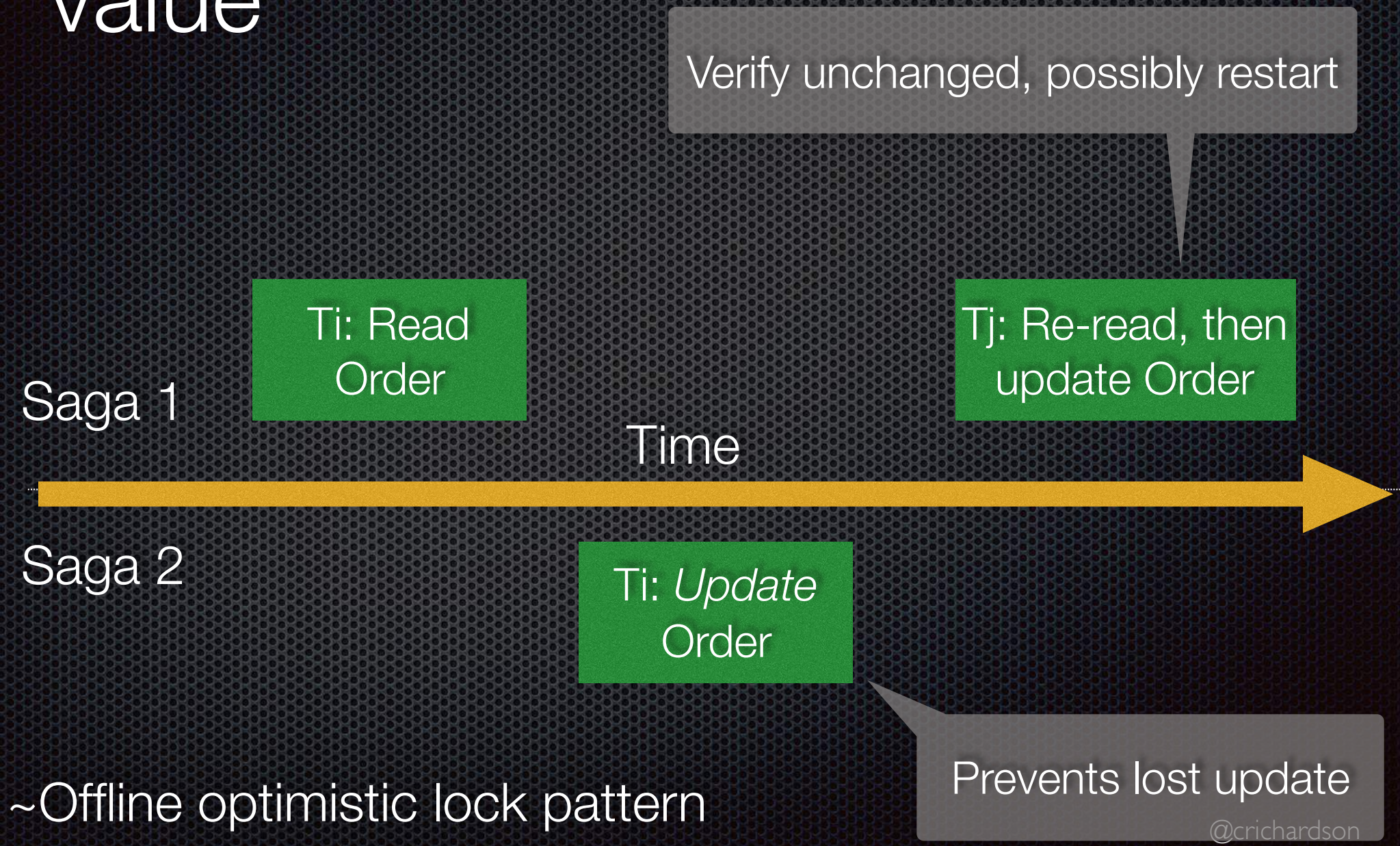
Countermeasure: Commutative updates

- Commutative: g(f(x)) = f(g(x))
- For example:
  - Account.debit() compensates for Account.credit()
  - Account.credit() compensates for Account.debit()
- Avoids lost updates

Countermeasure: Pessimistic view

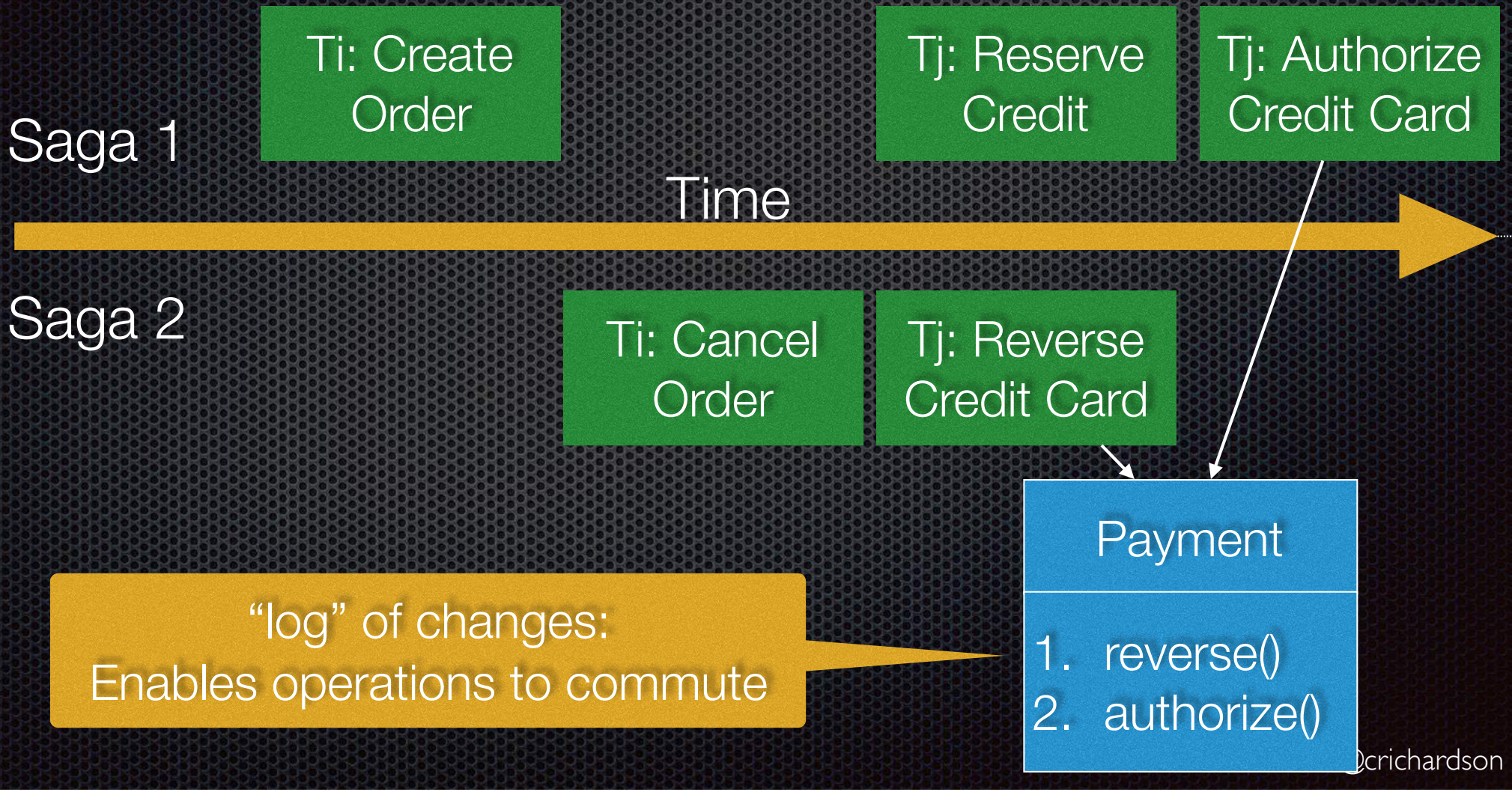


Countermeasure: Re-read value



~Offline optimistic lock pattern

Countermeasure: version file



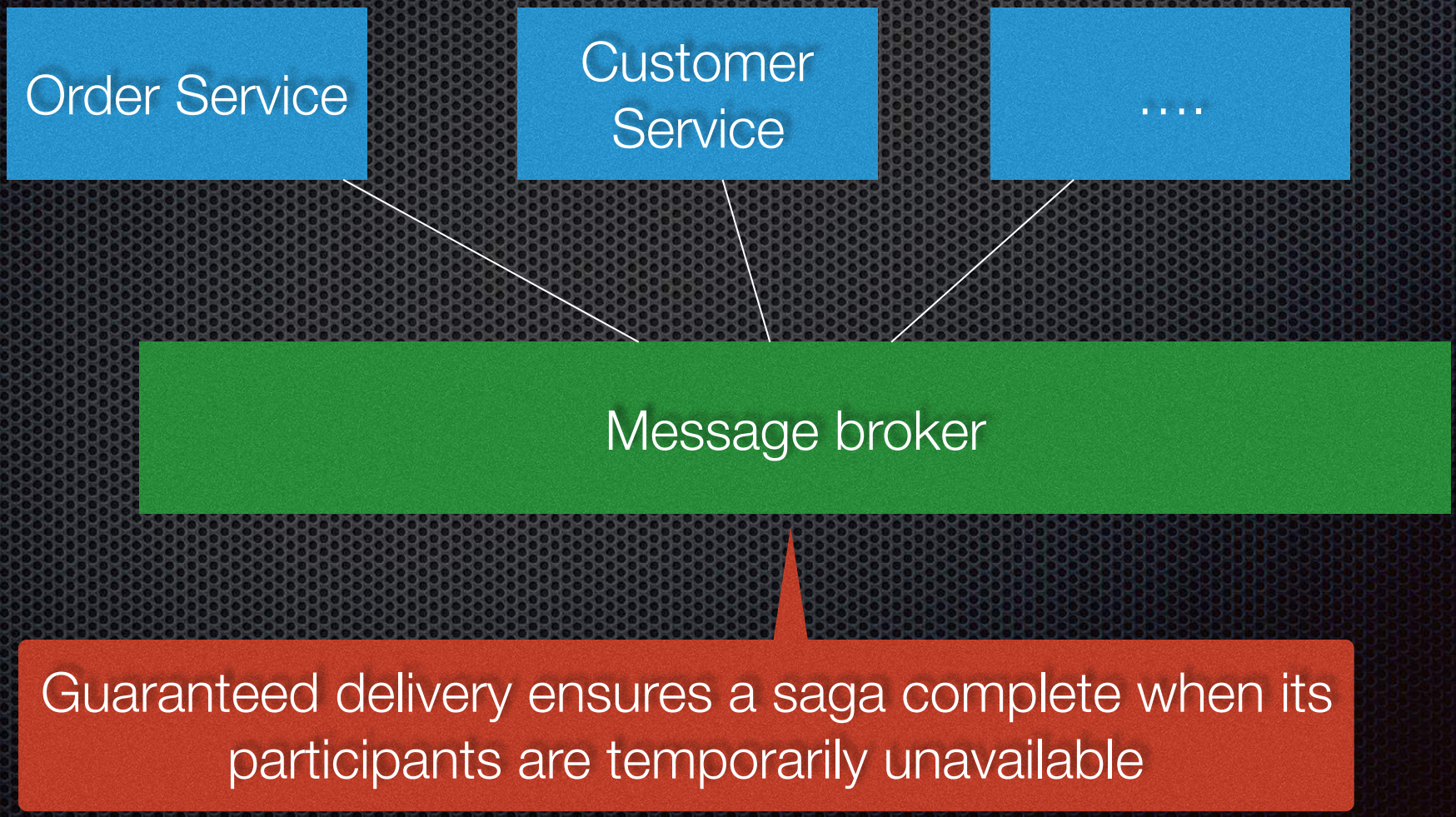
Countermeasure: By value

- Business risk determines strategy
- Low risk => semantic ACID
- High risk => use 2PC/distributed transaction

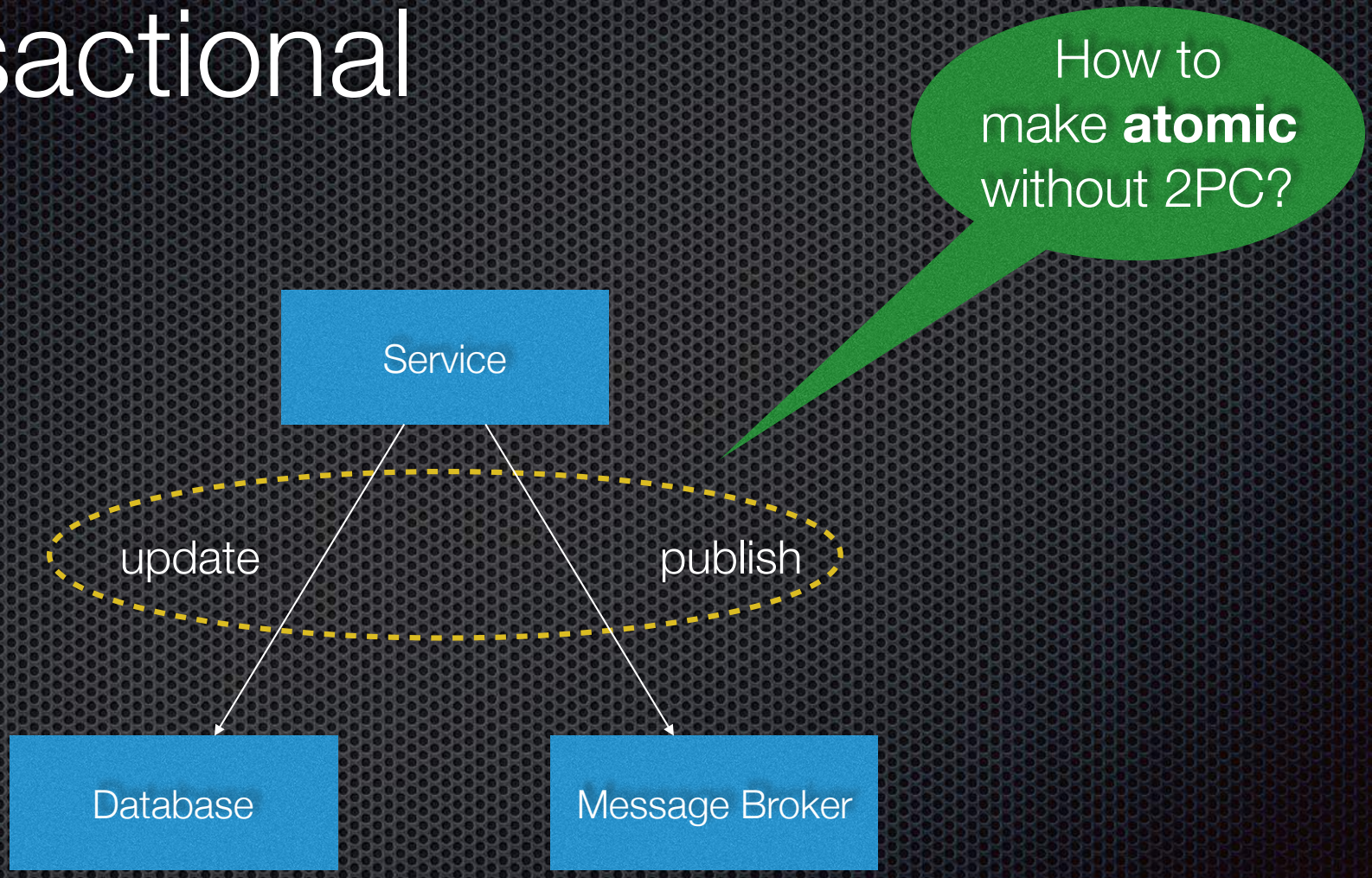
Agenda

- ACID is not an option
- Overview of sagas
- Coordinating sagas
- Countermeasures for data anomalies
- Using reliable and transactional messaging

Use asynchronous, broker-based messaging

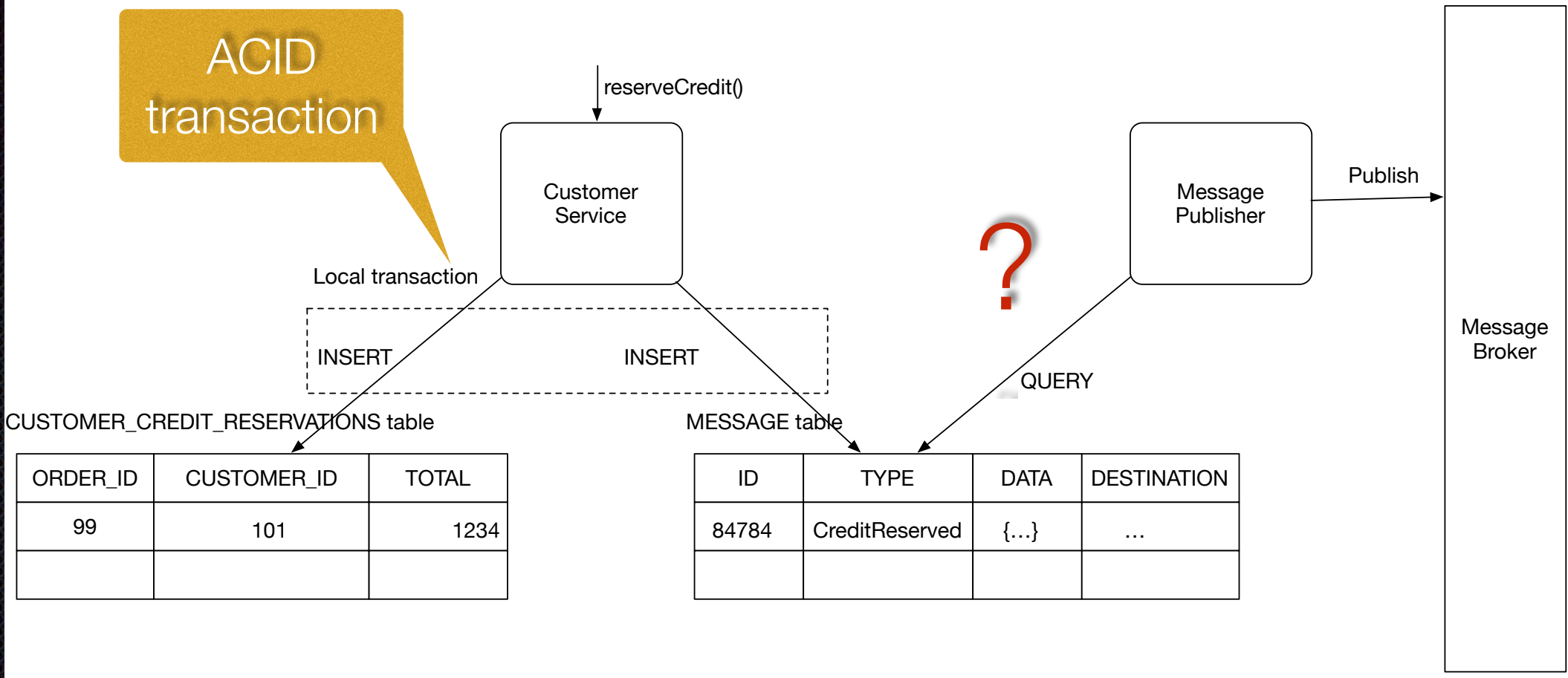


Messaging must be transactional



Option #1: Use database table as a message queue

Option #1: Use database table as a message queue

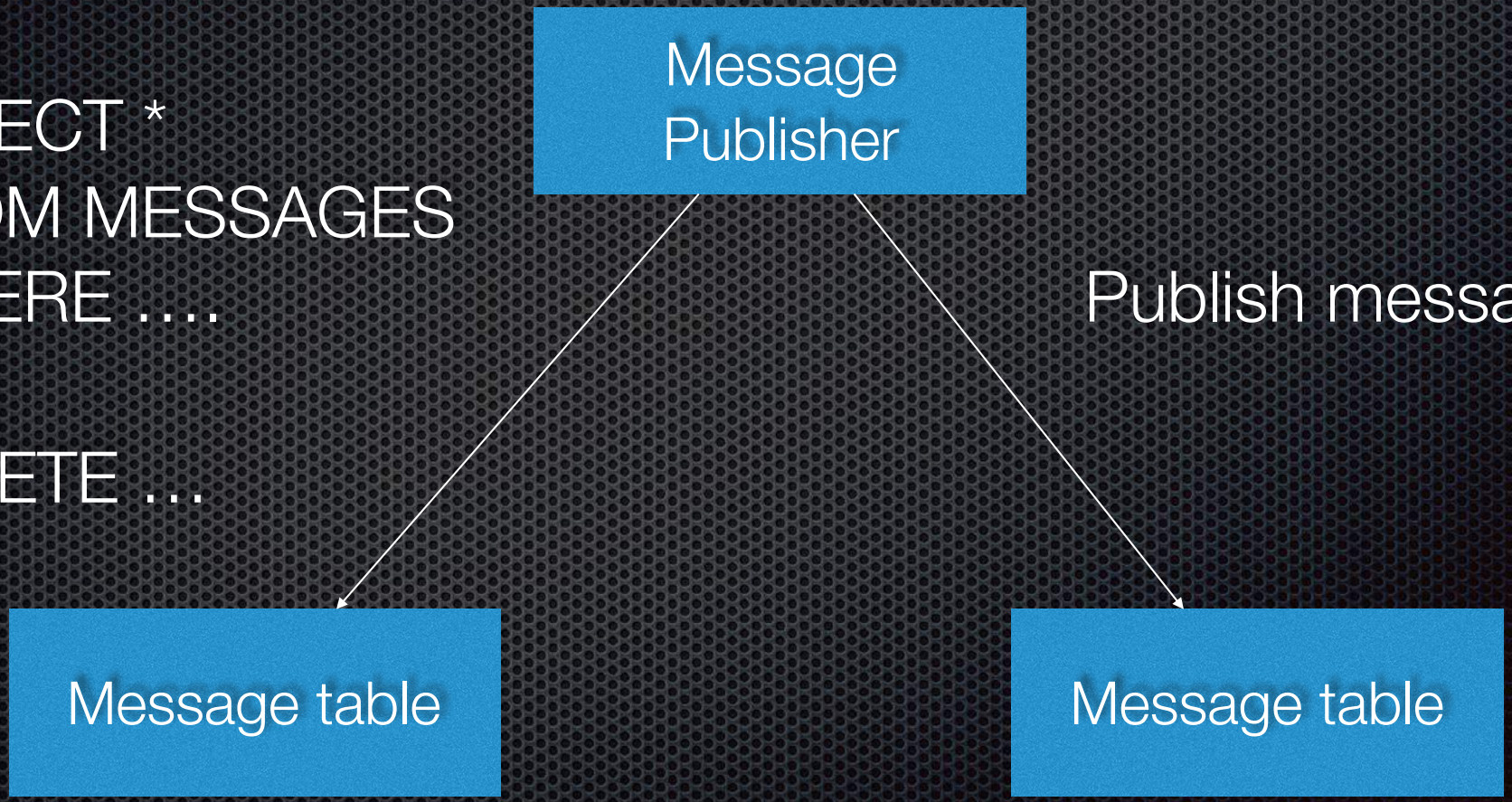


- See BASE: An Acid Alternative, http://bit.ly/ebaybase

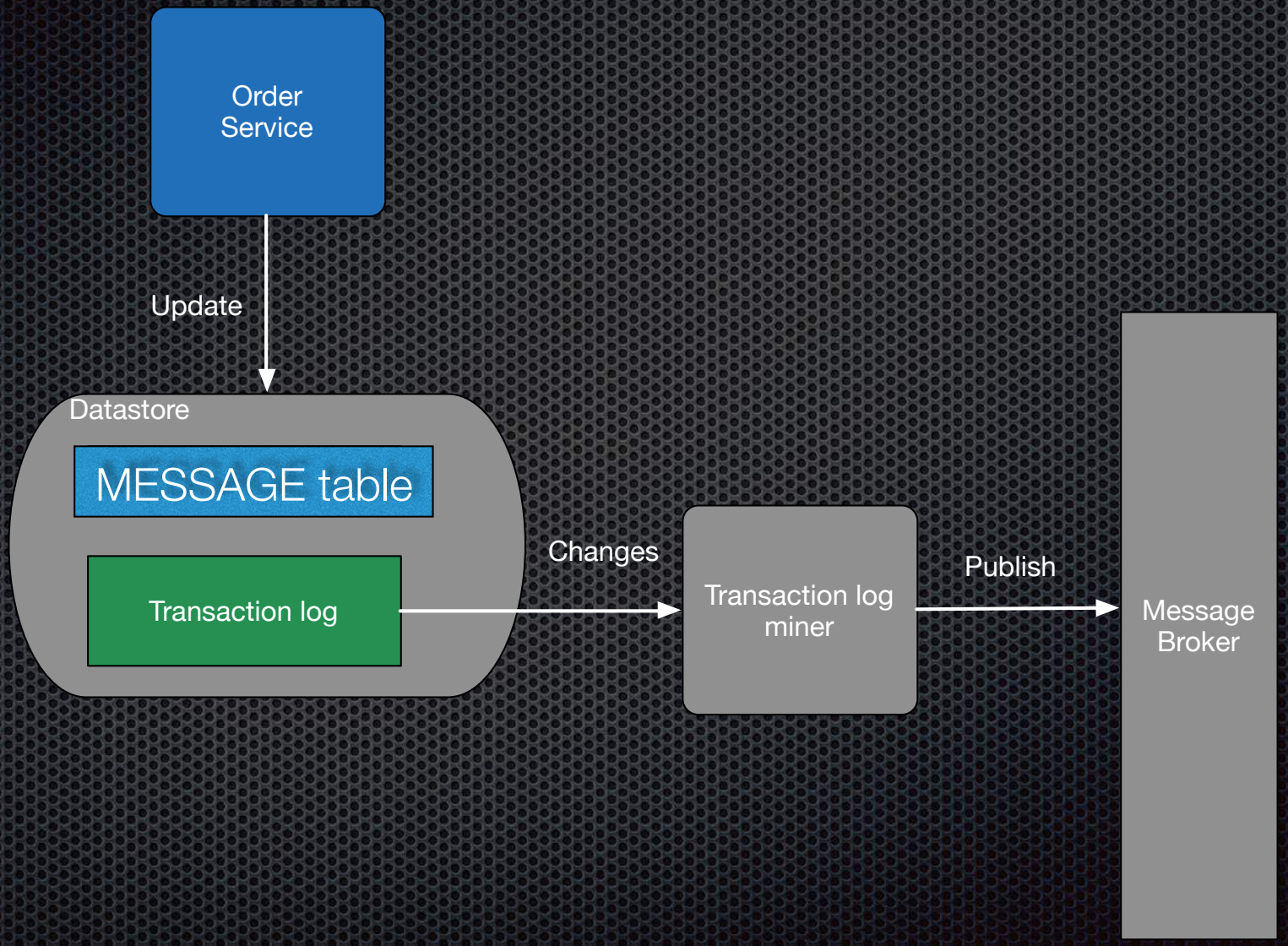
Publishing messages by polling the MESSAGE table

SELECT *
FROM MESSAGES
WHERE ….

DELETE …



Transaction log tailing



Transaction log tailing

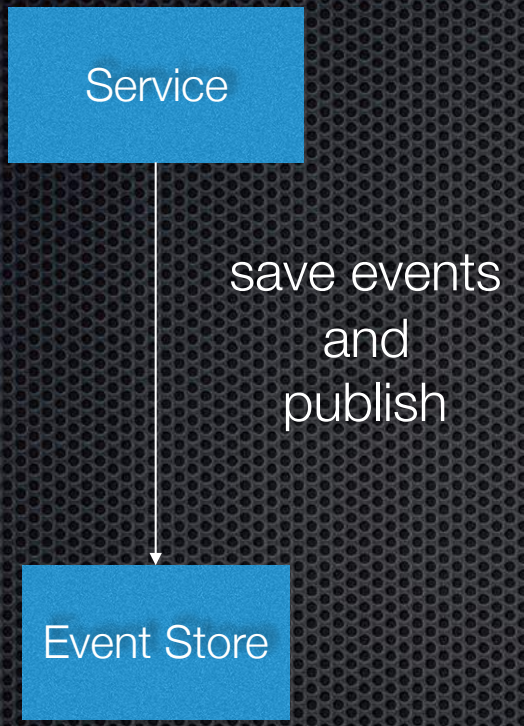
- Eventuate Local - reads MySQL binlog
- Oracle Golden Gate
- AWS DynamoDB streams
- MongoDB - Read the oplog

Transaction log tailing: benefits and drawbacks

- **Benefits**
  - No 2PC
  - No application changes required
  - Guaranteed to be accurate
- **Drawbacks**
  - Obscure
  - Database specific solutions
  - Tricky to avoid duplicate publishing

Option #2: Event sourcing: event-centric persistence

Event sourcing: persists an object as a sequence of events



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

Guarantees:

state change

->

event is published

Implementing choreography-based sagas is straightforward

Preserves history of domain objects

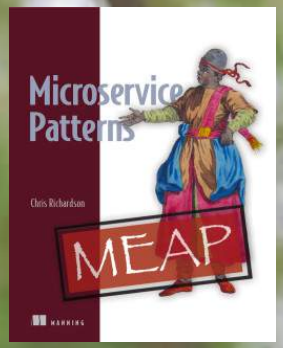
Supports temporal queries

Built-in auditing

Summary

- Microservices tackle complexity and accelerate development
- Database per service is essential for loose coupling
- Use ACID transactions **within** services
- Use orchestration-based or choreography-based sagas to maintain data consistency **across** services
- Use countermeasures to reduce impact of anomalies caused by lack of isolation

@crichardson chris@chrisrichardson.net



40% discount with code ctwsaturn18

Questions?

http://learn.microservices.io