



**INSTITUTO
FEDERAL**
Paraíba

Campus
Cajazeiras

PROGRAMAÇÃO P/ WEB 2

5. GERENCIANDO TRANSAÇÕES COM SAGAS

PROF. DIEGO PESSOA

✉ DIEGO.PESSOA@IFPB.EDU.BR

 @DIEGOEP



CST em Análise e
Desenvolvimento de
Sistemas

PROBLEMA

- ▶ Problema: gerenciar transação em operações realizadas por múltiplos serviços.
 - ▶ Solução: aplicação do padrão SAGA, sequência orientada a mensagens de transações locais para manter a consistência de dados.
- ▶ Desafio: sagas são ACD (falta o I de isolation).
 - ▶ Aplicações devem usar técnicas que reduzem o impacto de anomalias de concorrência provenientes da falta de isolamento.
- ▶ Primeiro obstáculo: mover de um single database com transações ACID para uma arquitetura multi-database com ACD sagas.
 - ▶ Na prática, mesmo aplicações monolíticas muitas vezes não usam ACID ao pé da letra, já que diminuem o nível de isolamento da transação para melhorar o desempenho.

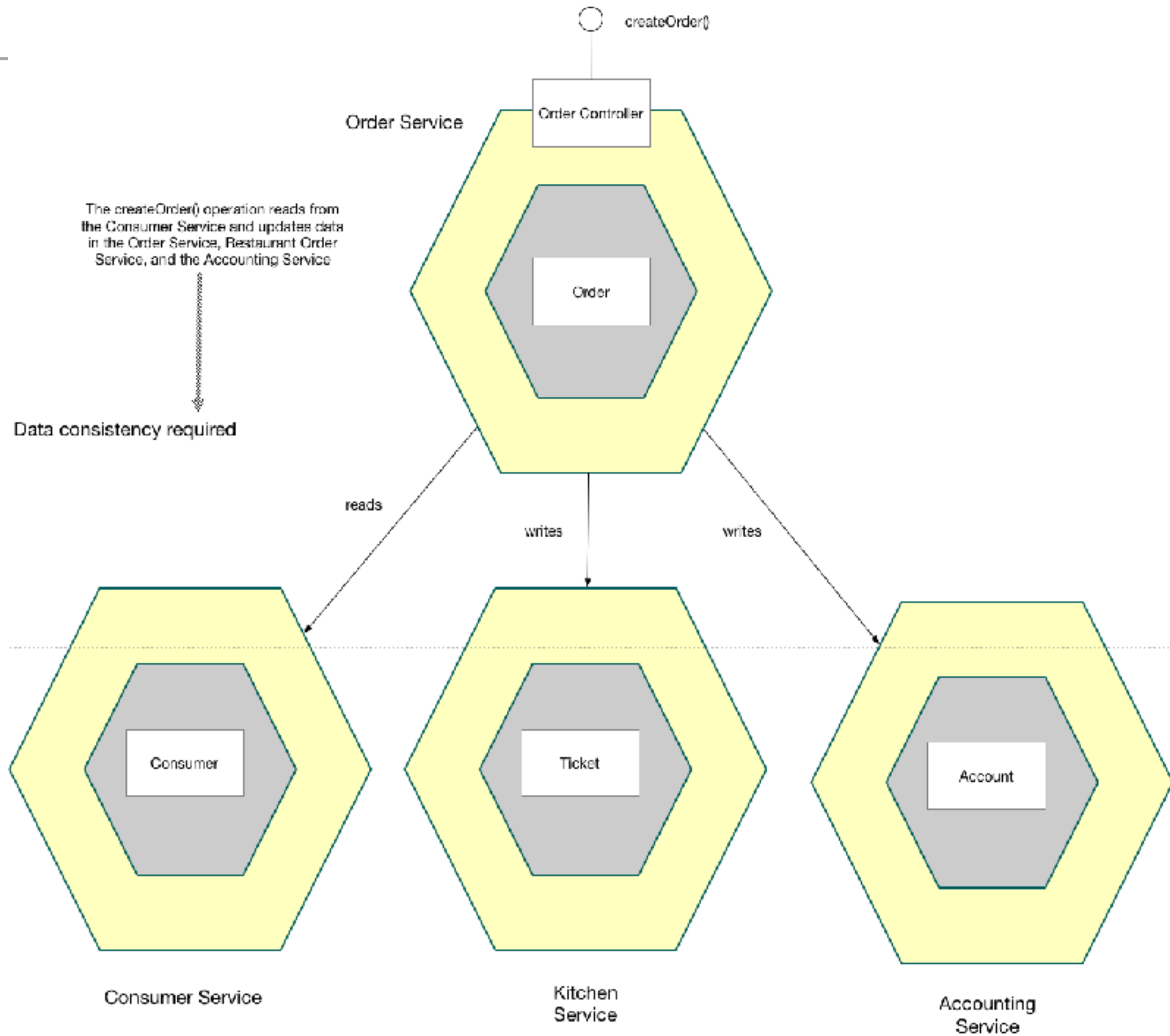
RESUMO

- ▶ Desafios de gerenciamento de transações em microsserviços.
- ▶ Como manter consistência de dados usando sagas
- ▶ Apresentação dos dois tipos diferentes de coordenação de sagas: coreografia e orquestração
- ▶ Como lidar com a concorrência causada pela falta de isolamento.
- ▶ Implementação de um exemplo de saga

GERENCIAMENTO DE TRANSAÇÕES EM MICROSERVIÇOS

DESAFIOS DE GERENCIAMENTO DE TRANSAÇÕES EM MICROSERVIÇOS.

- ▶ Transações em aplicações monolíticas é fácil. Principalmente pelos recursos disponibilizados pelos frameworks (ex.: @Transactional no Spring)
- ▶ Em um ambiente multi-database com microserviços, o desafio é maior, exigindo a criação de sagas para garantir a consistência dos dados.
- ▶ Exemplo: createOrder()



O PROBLEMA COM AS ABORDAGENS TRADICIONAIS

- ▶ O problema com transações distribuídas:
 - ▶ X/open Distributed Transaction Processing (DTP) Model (X/Open XA), JTA, etc.
 - ▶ Não suportam novas tecnologias (ex.: NoSQL)
 - ▶ Não se integram a sistemas de pubsub (ex.: Kafka)
 - ▶ Demandam que os serviços estejam disponíveis no momento da transação

USANDO O PADRÃO SAGA PARA MANTER A CONSISTÊNCIA

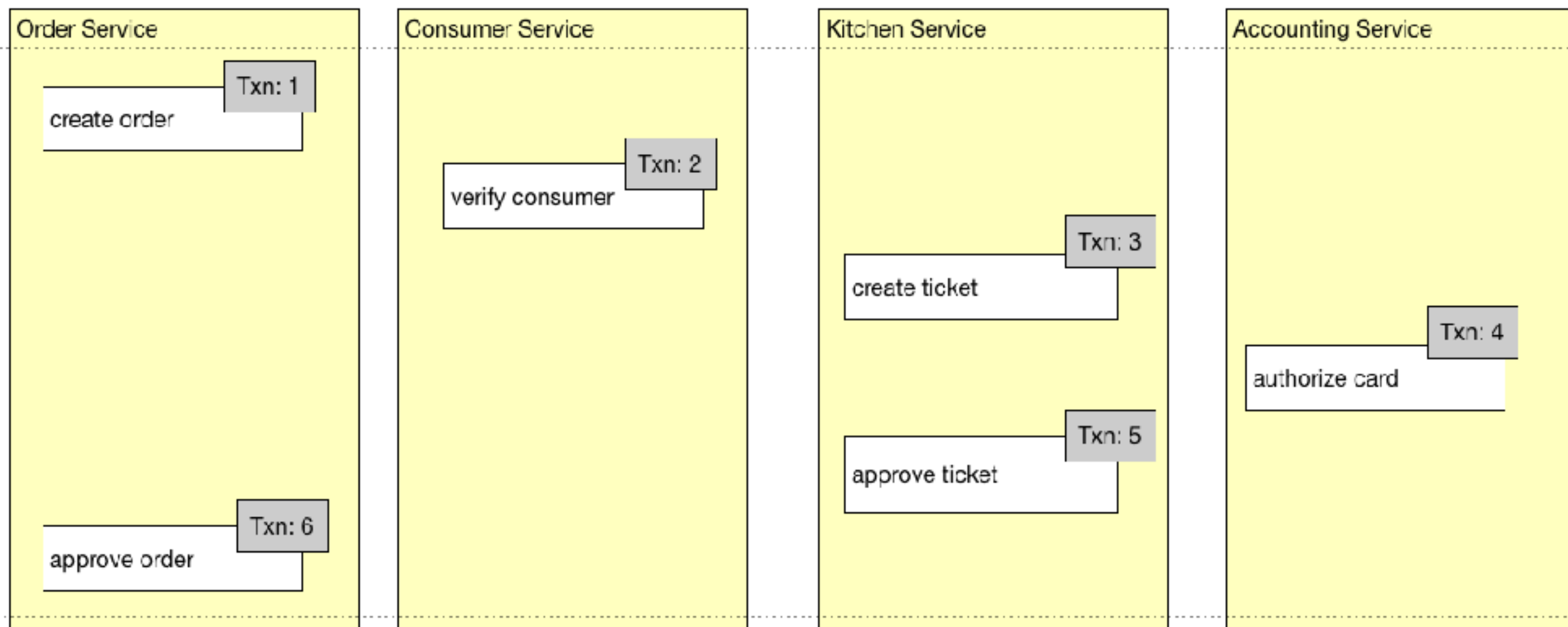
PATTERN: SAGA

MAINTAIN DATA CONSISTENCY ACROSS SERVICES USING A SEQUENCE OF LOCAL TRANSACTIONS THAT ARE COORDINATED USING ASYNCHRONOUS MESSAGING.

- ▶ Saga = Sequência de transações locais (usando ACID em cada uma) executadas em múltiplos serviços.
- ▶ Exemplo de saga para o método createOrder()
- ▶

USANDO O PADRÃO SAGA PARA MANTER A CONSISTÊNCIA

► Exemplo de saga para o método createOrder()



Desafios:

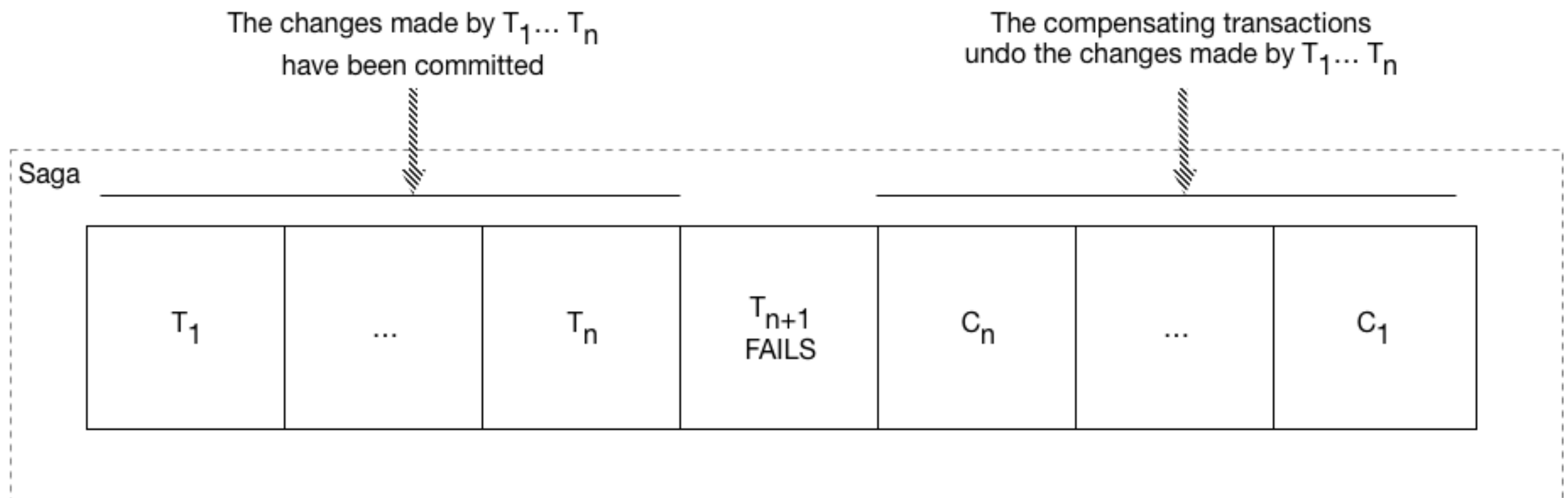
- Lidar com a falta de isolamento
- Rollbacks

SAGAS

- ▶ Sagas usam compensação de transações para fazer rollback de mudanças
- ▶ Em transações ACID é possível fazer facilmente rollback quando alguma regra de negócio é violada.
- ▶ Sagas não podem ser rolled back automaticamente porque a cada etapa as mudanças são comitadas no banco.
- ▶ Exemplo: se a autorização do cartão de crédito falha (Txn4), a aplicação precisa explicitamente desfazer as mudanças que foram feitas nas fases anteriores.
- ▶ Isso é conhecido por "transações de compensação"

CONTROLE DE FALHAS

- ▶ Ao falhar, as sagas são executadas em ordem inversa.



CONTROLE DE FALHAS

- ▶ Ao falhar, as sagas são executadas em ordem inversa.

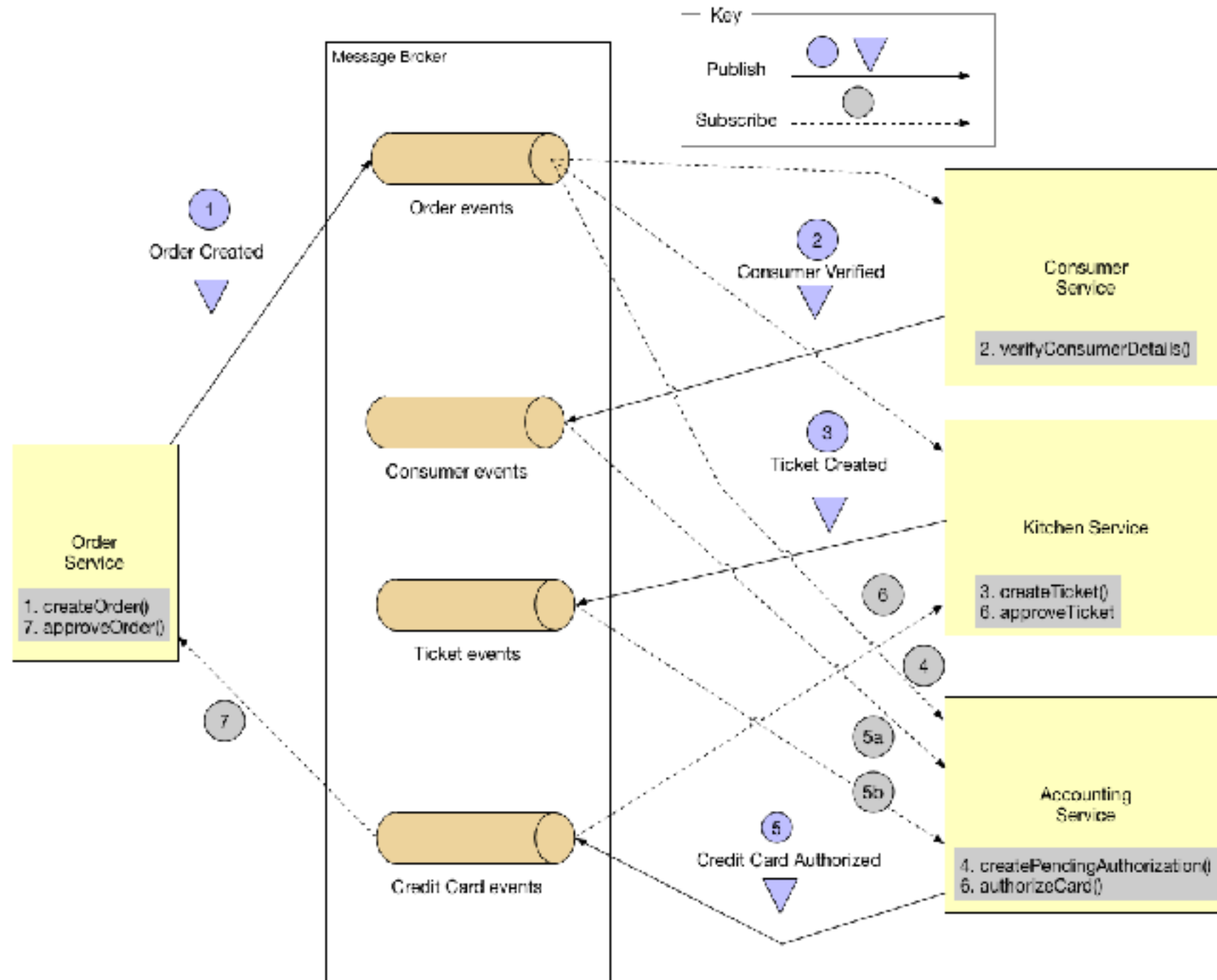
Step	Service	Transaction	Compensating transaction
1	Order Service	createOrder()	rejectOrder()
2	Consumer Service	verifyConsumerDetails()	-
3	Kitchen Service	createTicket()	rejectTicket()
4	Accounting Service	authorizeCreditCard()	-
5	Kitchen Service	approveTicket()	-
5	Order Service	approveOrder()	-

COORDENANDO SAGAS

COORDENANDO SAGAS

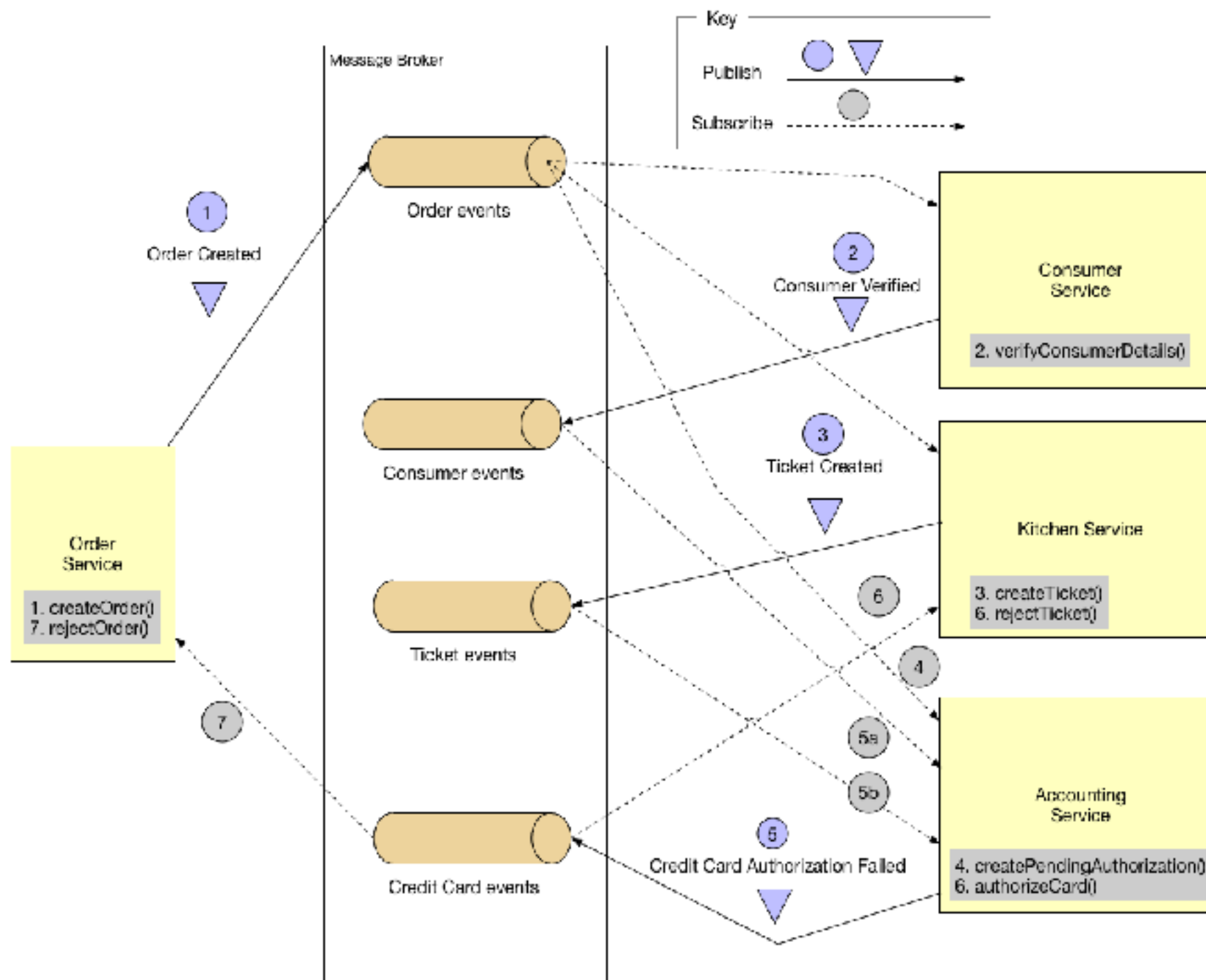
- ▶ Estratégias para estruturar a lógica de coordenação:
- ▶ **Coreografia** - distribui a tomada de decisão e o sequenciamento entre os participantes. Eles se comunicam trocando eventos.
- ▶ **Orquestração** - centralizar a lógica de coordenação numa classe orquestradora. O orquestrador envia mensagens para os participantes da saga indicando que operações realizar.

SAGAS BASEADAS EM “COREOGRAFIA” (CAMINHO FELIZ)



1. The OrderService creates an Order in the APPROVAL_PENDING state and publishes an OrderCreated event.
2. The ConsumerService consumes the OrderCreated event, verifies that the consumer can place the order, and publishes an ConsumerVerified event
3. The KitchenService consumes the OrderCreated event, validates the Order, creates a Ticket in a CREATE_PENDING state, and publishes TicketCreated event
4. The AccountingService consumes the OrderCreated event and creates a CreditCardAuthorization in a PENDING state
5. The AccountingService consumes the TicketCreated and ConsumerVerified events, charges the consumer's credit card and publishes CreditCardAuthorized event
6. The KitchenService consumes the CreditCardAuthorized event and changes the state of the Ticket to AWAITING_ACCEPTANCE
7. The OrderService receives the CreditCardAuthorized events, changes the state of the Order to APPROVED and publishes an OrderApproved event.

SAGAS BASEADAS EM “COREOGRAFIA” (CASO DE ERRO NA AUTORIZAÇÃO DO CARTÃO)



1. The OrderService creates an Order in the APPROVAL_PENDING state and publishes an OrderCreated event.
2. The ConsumerService consumes the OrderCreated event, verifies that the consumer can place the order, and publishes an ConsumerVerified event
3. The KitchenService consumes the OrderCreated event, validates the Order, creates a Ticket in a CREATE_PENDING state, and publishes TicketCreated event
4. The AccountingService consumes the OrderCreated event and creates a CreditCardAuthorization in a PENDING state
5. The AccountingService consumes the TicketCreated and ConsumerVerified events, charges the consumer's credit card and publishes a Credit Card Authorization Failed event
6. The Kitchen Service consumes the Credit Card Authorization Failed event and changes the state of the Ticket to REJECTED.
7. The Order Service consumes the Credit Card Authorization Failed event and changes the state of the Order to REJECTED

SAGAS BASEADAS EM “COREOGRAFIA” – DESAFIOS

▶ #1 - Comunicação confiável baseada em eventos

- ▶ Garantir que um participante da saga atualize o banco de dados e publique um evento como parte de uma transação
- ▶ Exemplo: na saga "Create Order", o Kitchen Service recebe um evento "Consumer Verified", cria um "Ticket" e publica um evento "Ticket Created".
- ▶ É essencial que o banco seja atualizado e que a aplicação do evento ocorra de maneira atômica.
- ▶ A comunicação deve usar Transacional Messaging (o Kafka [suporta](#)).

SAGAS BASEADAS EM “COREOGRAFIA” – DESAFIOS

- ▶ **#2 - Mapeamento de eventos para modelo de negócio local**
- ▶ Garantir que um participante de uma saga esteja apto a mapear cada evento que ele recebe para seus próprios dados
- ▶ Exemplo: quando o "Order Service" recebe um evento "Credit Card Authorized", ele deve estar apto a recuperar o pedido (Order).

SAGAS BASEADAS EM “COREOGRAFIA” – DESAFIOS

▶ #2 - Mapeamento de eventos para modelo de negócio local

- ▶ A solução é: para cada participante da saga, publicar eventos contendo um "correlation id", que é o dado que permite que outros participantes façam o mapeamento.
- ▶ Exemplo:
 - ▶ Os participantes da saga "Create Order" podem usar o "orderId" como um correlation id, que é passado de um participante para o outro.
 - ▶ O "Accounting Service" publica um evento "Credit Card Authorized" contendo o "orderId" do evento "TicketCreated"
 - ▶ Quando o "Order Service" recebe um evento "Credit Card Authorized", ele usa o "orderId" para recuperar o pedido (Order) correspondente.
 - ▶ Similarmente, o Kitchen Service usa o "orderId" do evento para recuperar o Ticket correspondente.

SAGAS BASEADAS EM “COREOGRAFIA” – VANTAGENS

- ▶ **Simplicidade** - serviços somente publicam eventos quando eles criam, atualizam ou deletam objetos de negócio.
- ▶ **Baixo acoplamento entre serviços** - os participantes somente se inscrevem a eventos, não possuindo conhecimento direto sobre os outros

SAGAS BASEADAS EM “COREOGRAFIA” – DESVANTAGENS

- ▶ **Mais difícil de entender** - não há um único local para definir a saga. A implementação é distribuída entre os serviços. Pode ser difícil pro desenvolvedor entender como uma determinada saga funciona.
- ▶ **Dependências cíclicas entre os serviços** - os participantes da saga se inscrevem cada um aos eventos do outro, o que irá gerar uma dependência cíclica. Exemplo: Order Service → Accounting Service → Order Service. Isto não é necessariamente um problema, mas é considerado um “design smell”.
- ▶ **Risco de alto acoplamento entre eventos** - cada participante precisa se inscrever a todos os eventos que os afetam. Exemplo: Accounting Service precisa se inscrever para todos os eventos, visto que o cartão de crédito pode ser liberado ou rejeitado. Como resultado, há um risco de que ele precise ser atualizado em confronto com o que foi implementado no “Order Service”.

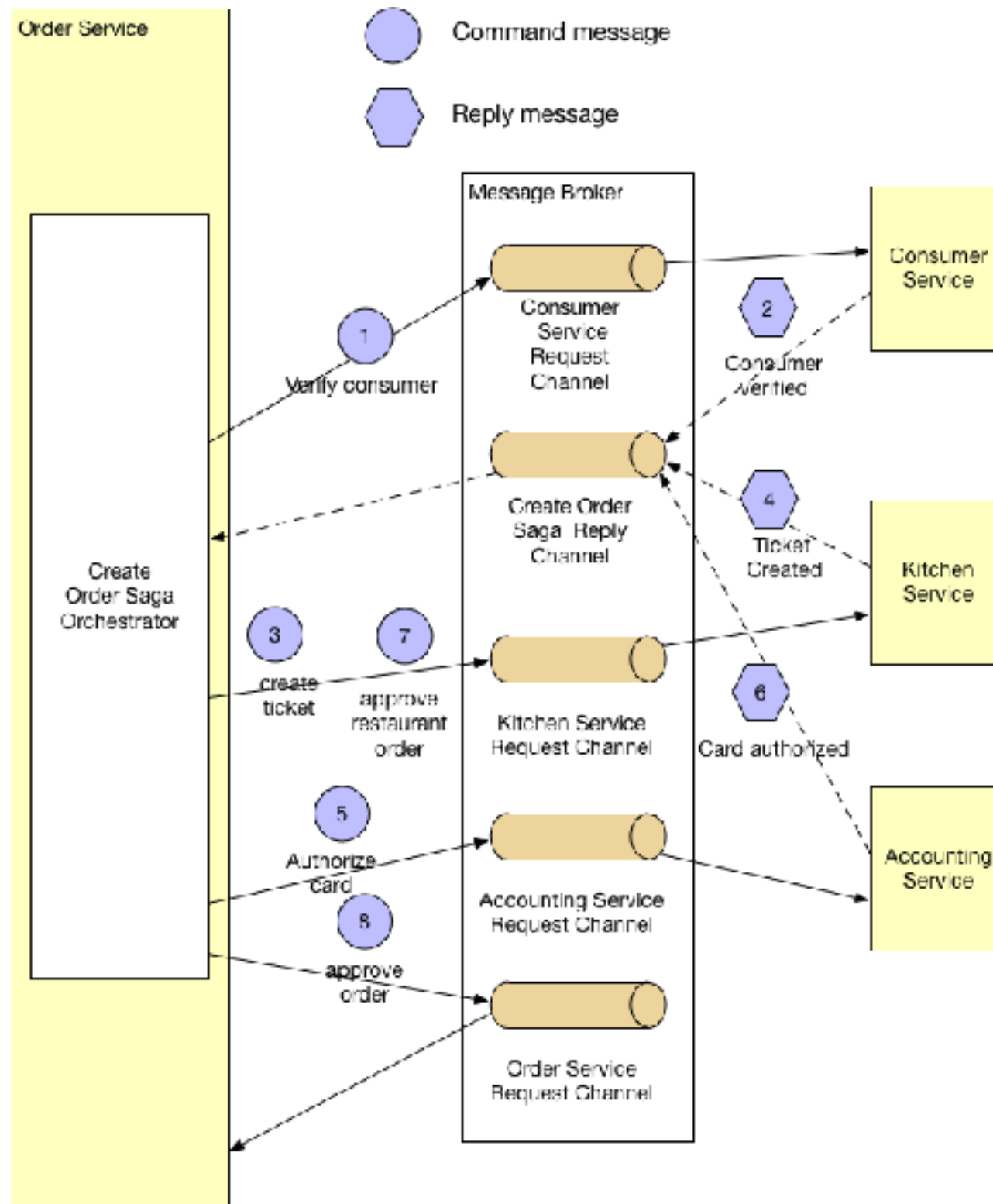
SAGAS BASEADAS EM “COREOGRAFIA” – RESUMO

- ▶ Podem funcionar bem para sagas simples, mas por conta das desvantagens, para sagas mais complexas, geralmente é melhor usar orquestração.

SAGAS BASEADAS EM “ORQUESTRAÇÃO”

- ▶ Ao usar orquestração, é definido uma classe “Orquestradora” cuja responsabilidade é dizer aos participantes da saga o que deve ser feito
- ▶ O orquestrador se comunica com os participantes usando o estilo de interação de comandos assíncronos com resposta.
- ▶ Para executar uma etapa da saga, ele envia uma mensagem do tipo comando para um participante dizendo a operação que deve ser feita.
- ▶ Após o participante realizar a operação, ele envia uma mensagem de resposta ao orquestrador.
- ▶ O orquestrador processa a mensagem e determina qual etapa da saga realizar em seguida

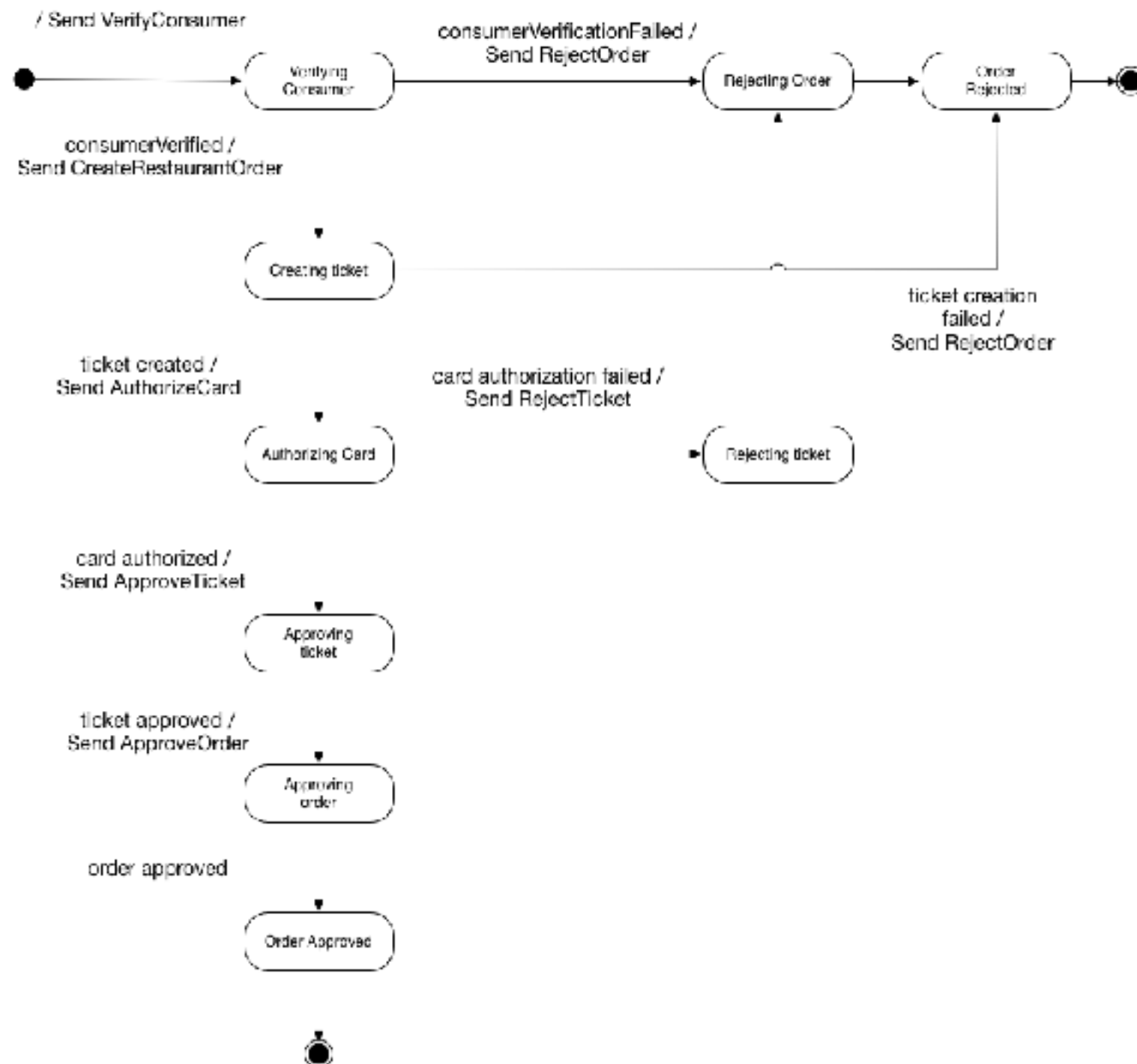
SAGAS BASEADAS EM “ORQUESTRAÇÃO”



1. The saga orchestrator sends a **Verify Consumer** command to the **Consumer Service**
2. The **Consumer Service** replies with a **Consumer Verified** message.
3. The saga orchestrator sends a **Create Ticket** command to the **Kitchen Service**
4. The **Kitchen Service** replies with a **Ticket Created** message
5. The saga orchestrator sends a **Authorize Card** message to the **Accounting Service**
6. The **Accounting Service** replies with a **Card Authorized** message
7. The saga orchestrator sends an **Approve Ticket** command to the **Kitchen Service**
8. The saga orchestrator sends an **Approve Order** command to the **Order Service**.

MODELANDO SAGAS COMO “STATE MACHINES”

Reply from Participant / send CommandMessage



- **Verifying Consumer** - the initial state When in this state, the saga is waiting for the Consumer Service verify that the consumer can place the order
- **Creating Ticket** - the saga is waiting for a reply to Create Ticket command
- **Authorizing Card**- waiting for the Accounting Service to authorize the consumer's credit card
- **Order Approved** - a final state indicating that saga completed successfully
- **Order Rejected** - a final state indicating that the Order was rejected by one of the participants

ORQUESTRAÇÃO DE SAGA E MENSAGENS TRANSACIONAIS

- ▶ Cada etapa de uma saga baseada em orquestração consiste em um serviço atualizar um banco de dados e publicar uma mensagem.
- ▶ O serviço precisa usar mensagens transacionais para atualizar atomicamente o banco de dados e publicar as mensagens

ORQUESTRAÇÃO DE SAGAS – BENEFÍCIOS

- ▶ **Dependências mais simples** - Não introduz dependências cíclicas. O orquestrador invoca os participantes, mas o contrário não é permitido. Ou seja, o orquestrador depende dos participantes, mas não vice-versa.
- ▶ **Menor acoplamento** - serviço implementa apenas API que é chamada pelo orquestrador. Logo, não precisa conhecer sobre todos os outros eventos.
- ▶ **Melhora separação de responsabilidades e simplifica a lógica de negócio** - objetos de domínio são mais simples e não têm conhecimento das sagas em que participam.

ORQUESTRAÇÃO DE SAGAS – DESVANTAGENS

- ▶ **Risco de centralização excessiva de lógica de negócio no orquestrador** - Isto resulta num projeto em que o "smart" orquestrador diz a todos os serviços que operação fazer.
- ▶ É possível evitar este problema projetando orquestrados que são somente responsáveis pela sequência de eventos e não conter qualquer outra lógica de negócio.

ORQUESTRAÇÃO DE SAGAS – RESUMO

- ▶ Recomenda-se usar orquestração para todas as sagas, exceto as mais simples.
- ▶ A implementação de uma lógica de coordenação para as sagas é um dos problemas que precisa ser resolvido
- ▶ Outro problema, talvez o maior deles, é tratar a falta de isolamento.

TRATANDO A FALTA DE ISOLAMENTO

UMA SAGA É ACD

- ▶ **Atomicidade** - a implementação da saga garante que todas as transações são executadas ou todas as mudanças são desfeitas.
- ▶ **Consistência** - integridade referencial dentro de um serviço é tratada pelas bases de dados locais. A integridade referencial entre serviços é tratada pelos serviços.
- ▶ **Durabilidade** - tratado pelos bancos de dados locais

LIDANDO COM A FALTA DE ISOLAMENTO

- ▶ ACID, (i) = isolamento: garante que a saída de múltiplas transações concorrentes é a mesma se elas forem executadas em sequência.
- ▶ O banco provê a ilusão de que cada transação ACID tem acesso exclusivo ao dado.
- ▶ Isolamento torna fácil escrever lógica de negócio que executa de maneira concorrente
- ▶ O desafio em usar sagas é que não há a propriedade de isolamento das transações ACID

LIDANDO COM A FALTA DE ISOLAMENTO

- ▶ Isto ocorre porque as atualizações feitas por cada transação local são visíveis imediatamente para outras sagas, visto que elas são comitadas.
- ▶ Este comportamento causa dois problemas:
 - ▶ 1) Outras sagas podem mudar os dados acessados pela saga enquanto ela está executando
 - ▶ 2) Sagas podem ler seus dados antes de outra saga completar suas atualizações e consequentemente serem expostos à dados inconsistentes (leitura suja)

LIDANDO COM A FALTA DE ISOLAMENTO

- ▶ A falta de isolamento potencialmente causa o que a literatura chama de "anomalias".
- ▶ Uma anomalia é quando uma transação lê ou escreve dados de maneira que não seria se fossem executadas uma por vez.
- ▶ Quando uma anomalia ocorre, a saída das sagas concorrentes é diferente do que seria se fossem executadas em sequência.

LIDANDO COM A FALTA DE ISOLAMENTO

- ▶ Superficialmente, a falta de isolamento tornar o modelo impraticável
- ▶ Mas, na prática, é comum aceitar um isolamento reduzido em troca de maior desempenho.
- ▶ Transações de mundo real geralmente são diferentes da definição seguida ao pé da letra

VISÃO GERAL DAS ANOMALIAS

- ▶ Atualizações perdidas - uma saga sobrescreve sem ler as mudanças feitas por outra saga
- ▶ Leituras sujas - uma transação ou uma saga lê a atualização feita por uma saga que não completou ainda as suas atualizações
- ▶ Leituras não repetidas - duas diferentes etapas de uma saga leem o mesmo dado e recebe resultados diferentes porque uma outra saga fez atualizações

ANOMALIAS – ATUALIZAÇÕES PERDIDAS

- ▶ Exemplo:
 - ▶ 1. O primeiro passo da saga "Create Order" cria um pedido (Order)
 - ▶ 2. Enquanto a saga está executando, a saga "Cancel Order" cancela o pedido
 - ▶ 3. A etapa final da saga "Create Order" aprova o pedido
- ▶ Neste cenário, a saga "Create Order" ignora as mudanças feitas pela saga "Cancel Order", sobrescrevendo-as. O que vai fazer com que a aplicação entregue um pedido que foi cancelado.

ANOMALIAS – LEITURAS SUJAS

- ▶ Imagine um cenário que intercala a execução das sagas "Cancel Order" e "Create Order" e a saga "Cancel order" é rolled back porque é muito tarde para cancelar o pedido. É possível que uma sequência de transações que invocam o Consumer Service como:
 - ▶ 1. Cancel Order saga - aumenta o crédito disponível
 - ▶ 2. Create Order saga - reduz crédito disponível
 - ▶ 3. Cancel Order saga - uma transação de compensação reduz o crédito disponível
- ▶ Neste cenário, a saga "Create Order" faz uma leitura suja do crédito disponível, que permite ao consumidor exceder o seu limite de crédito.

CONTRAMEDIDAS PARA LIDAR COM FALTA DE ISOLAMENTO

- ▶ Que estratégia pode ser pensada para lidar com as anomalias?
- ▶ Uma estratégia seria o uso dos estados `*_PENDING`, exemplo: `APPROVAL_PENDING`
- ▶ Neste caso, uma saga que atualiza pedidos, inicia atribuindo o estado do pedido para `*_PENDING`. Esse estado indica para as outras transações que o pedido está sendo atualizado por outra saga
- ▶ Essa estratégia foi introduzida por [Frank and Zahle, 1998] e se chama "Semantic Lock"

CONTRAMEDIDAS PARA LIDAR COM A FALTA DE ISOLAMENTO

- ▶ O paper de [Frank and Zahle, 1998] descreve as seguintes contramedidas:
 - ▶ **Semantic lock** - lock no nível da aplicação
 - ▶ **Commutative updates** - projetar para que a atualização de operações seja executada em qualquer ordem
 - ▶ **Pessimistic view** - reordenar as etapas de uma saga para minimizar o risco de negócio
 - ▶ **Reread value** - se prevenir contra escrita suja através da re-leitura do dado para verificar se ele sofreu mudanças antes de sobrescreve-lo
 - ▶ **Version file** - gravar as atualizações em um registro (ex.: número de versão), de maneira que elas possam ser reordenadas
 - ▶ **By value** - usa cada risco de negócio da requisição para dinamicamente selecionar o mecanismo de concorrência

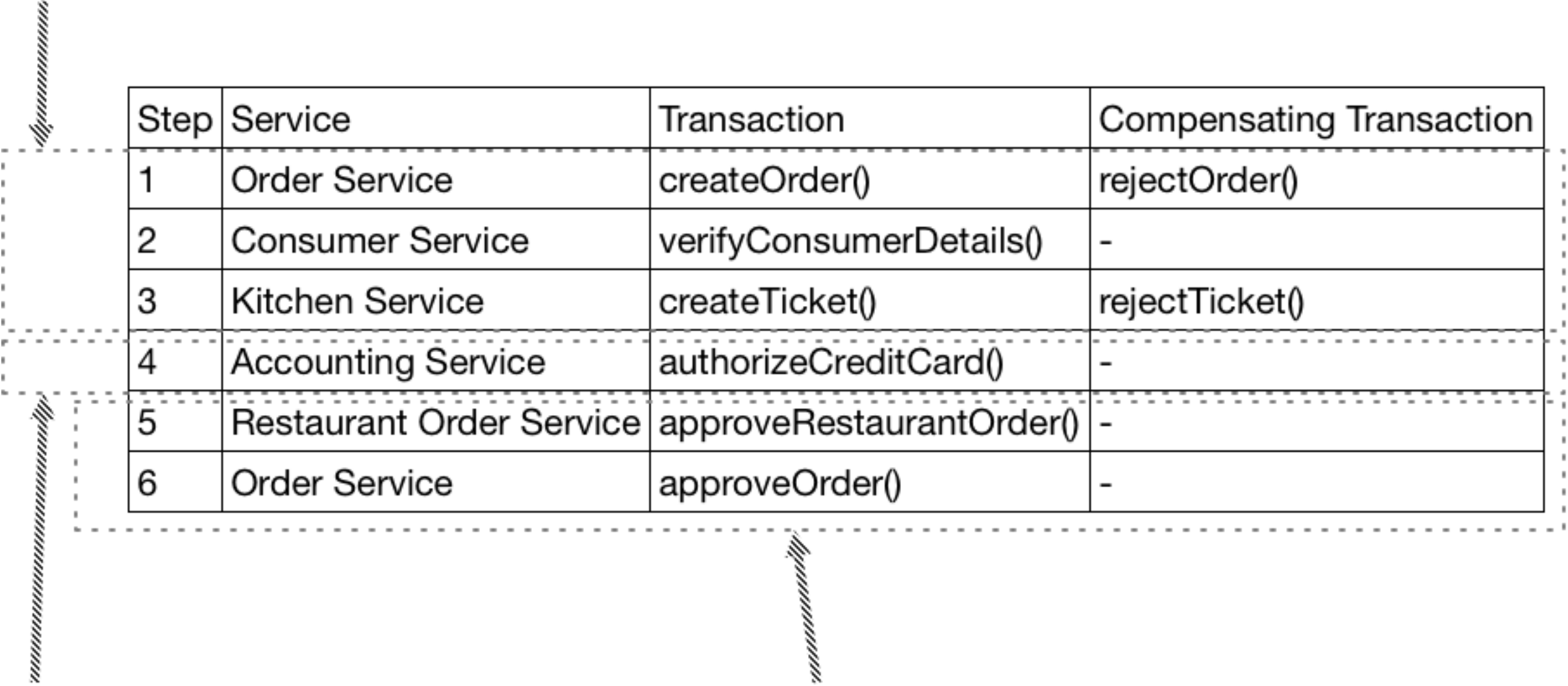
ESTRUTURA DE UMA SAGA

- ▶ Seguindo o modelo das contramedidas, podemos definir os seguintes tipos de transação para uma saga:
- ▶ **Compensatable transactions** - transações que podem potencialmente ser "rolled back" usando uma transação de compensação;
- ▶ **Pivot transaction** - o ponto crítico de uma saga (go/no-go point). Se a transação é commitada, então a saga irá seguir até ser completada.
- ▶ **Retriable transactions** - transações que seguem uma "pivot transaction" e garantem a execução completa da saga.

ESTRUTURA DE UMA SAGA

Compensatable Transactions:

Must support roll back



Step	Service	Transaction	Compensating Transaction
1	Order Service	createOrder()	rejectOrder()
2	Consumer Service	verifyConsumerDetails()	-
3	Kitchen Service	createTicket()	rejectTicket()
4	Accounting Service	authorizeCreditCard()	-
5	Restaurant Order Service	approveRestaurantOrder()	-
6	Order Service	approveOrder()	-

Pivot Transaction:

The saga's go/no-go transaction. If it succeeds, then the saga runs to completion.

Retriable Transactions:

Guaranteed to complete

COUNTERMEASURE: SEMANTIC LOCK

- ▶ Uma saga compensatable transaction atribui uma flag a qualquer registro que ela cria ou atualiza.
- ▶ A flag indica que o registro ainda não está "comittado" e pode sofrer mudanças
- ▶ A flag pode ser também um "lock" que impede outras transações de acessar o registro ou um "aviso" que indica que outra transação deve tratar o registro com cautela.
- ▶ Este estado é limpo tanto por uma "retriable transaction" - quando a saga é completada com sucesso - ou por uma "compensating transaction" - quando a saga é rolled back.

COUNTERMEASURE: SEMANTIC LOCK

- ▶ Exemplo: campo `order.state`
- ▶ Os estados `*_PENDING` (`APPROVAL_PENDING`, `REVISION_PENDING`) implementam um semantic lock.
- ▶ Dizem para outras sagas que o acesso ao pedido que uma saga está processando está em processo de atualização.
- ▶ Por exemplo, o primeiro passo da saga "Create Order", que é uma "compensatable transaction", cria um pedido no estado "APPROVAL_PENDING". A etapa final, que é uma "retriable transaction", muda o campo para "APPROVED". Transações de compensação mudam o campo para "REJECTED".

COUNTERMEASURE: SEMANTIC LOCK

- ▶ No entanto, o gerenciamento do lock é apenas metade do problema. É preciso decidir caso a caso baseado em como a saga deve lidar com o registro que foi travado.
- ▶ Por exemplo: considere o comando `cancelOrder()`. Um cliente pode invocar essa operação para cancelar um pedido que está no estado `APPROVAL_PENDING`
 - ▶ Uma opção é retornar uma mensagem de falha e dizer pro cliente tentar novamente. O principal benefício desta abordagem é que é simples de implementar. A desvantagem é que isto torna o cliente mais complexo de implementar, visto que ele precisa implementar a lógica para tentar novamente.
 - ▶ Outra opção seria bloquear o método `cancelOrder()` até o lock ser liberado.
- ▶ O benefício de usar semantics locks é que isto praticamente recria o isolamento provido por transações ACID.
- ▶ A desvantagem é que a aplicação precisa gerenciar os locks. Também pode ser necessário implementar um algoritmo de detecção de deadlocks para realizar o rollback de uma saga para quebrar um deadlock e re-executá-lo.

COUNTERMEASURE: COMMUTATIVE UPDATES

- ▶ Operações são comutativas se elas podem ser executadas em qualquer ordem
- ▶ Operações de debito() e credito() são comutativas (se for ignorado o cheque especial). Esta medida é útil porque elimina atualizações perdidas.
- ▶ Exemplo:
 - ▶ Considere um cenário em que uma saga precisa passar por um roll back após uma transação de compensação ter debitado (ou creditado) uma conta.
 - ▶ A transação somente pode creditar (ou debitar) a conta para desfazer a atualização
 - ▶ Desta forma, não há possibilidade de sobrescrever atualizações feitas por outras sagas.

COUNTERMEASURE: PESSIMISTIC VIEW

- ▶ Reordena as etapas de uma saga para minimizar o risco de uma leitura suja.
- ▶ Exemplo:
 - ▶ Cenário em que uma leitura suja do crédito disponível permite exceder limite
 - ▶ Para reduzir esse risco, podemos reordenar a saga "Cancel Order" de:
 - ▶ 1. Consumer Service - aumenta o crédito disponível
 - ▶ 2. Order Service - muda o estado do pedido para "CANCELADO"
 - ▶ 3. Delivery Service - cancela a entrega
 - ▶ Para:
 - ▶ 1. Order Service: muda o estado do pedido para "CANCELADO"
 - ▶ 2. Delivery Service : cancela a entrega
 - ▶ 3. Customer Service : aumenta o crédito disponível
 - ▶ Na reordenada, o crédito é aumentado numa "reliable transaction", o que elimina a possibilidade de uma leitura suja

COUNTERMEASURE: REREAD VALUE

- ▶ A releitura de values previne atualizações perdidas.
- ▶ Uma saga que usa esta estratégia sempre vai reler um registro antes de atualizá-lo, verificando se houve mudança para então atualizá-lo. Se houve mudança no registro, a saga aborda e possivelmente reinicia
- ▶ Essa medida é uma forma do padrão [Lock Otimista Offline](#).
- ▶ A saga "Create Order" poderia usar essa medida para tratar o cenário em que um pedido é cancelado enquanto ele está no processo de ser aprovado.
- ▶ A transação que aprova o pedido verifica que se o pedido sofreu mudanças desde que ele foi criado no início da saga.
- ▶ Se não houve mudança, a transação aprova o pedido. Do contrário, se o pedido foi cancelado, a transação aborta a saga, o que faz com que as transações de compensação sejam executadas.

COUNTERMEASURE: VERSION FILE

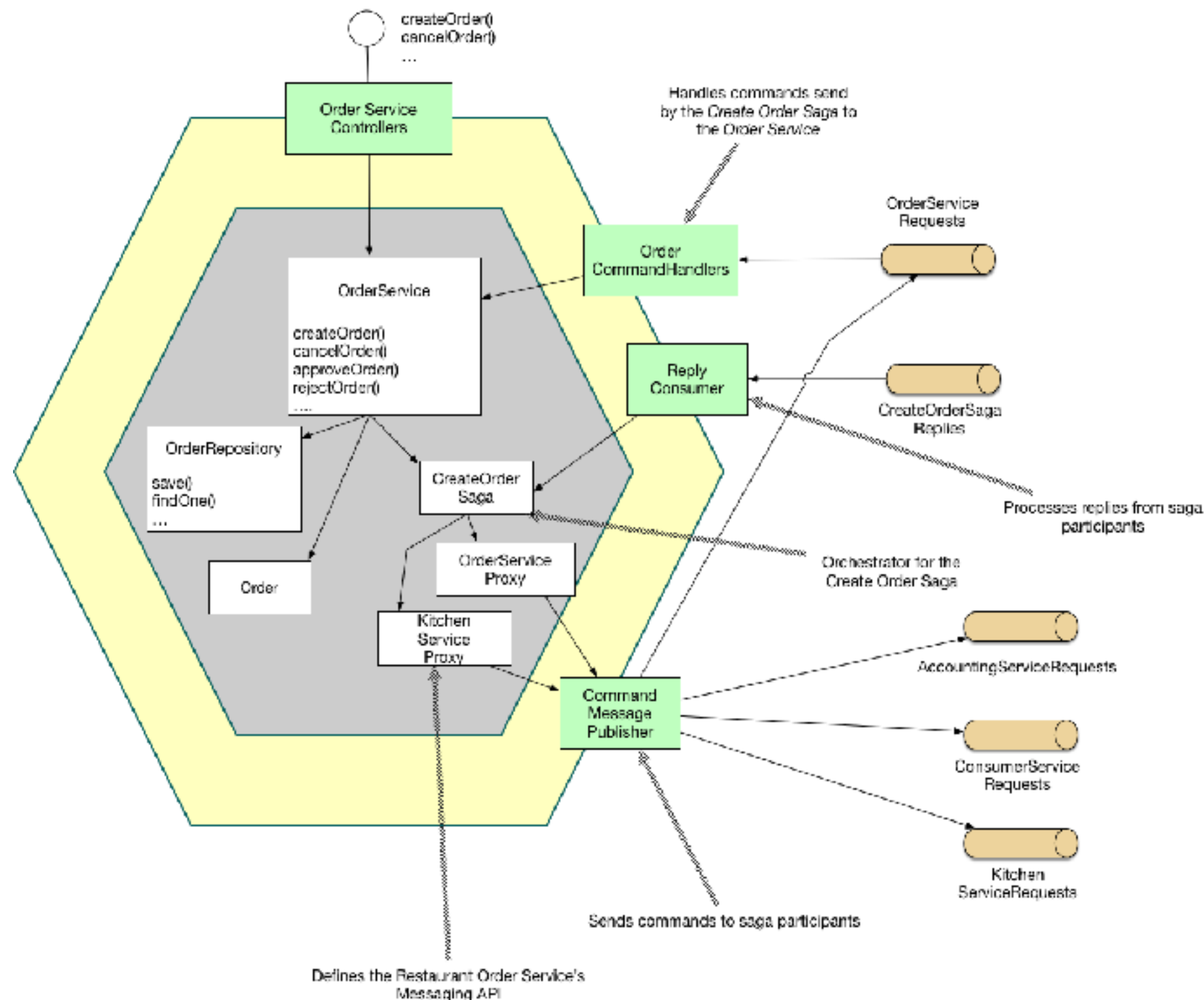
- ▶ Grava operações que são realizadas em um registro de maneira que se possível reordená-las.
- ▶ É uma forma de tornar operações não-comutativas em operações comutativas
- ▶ Exemplo: considere um cenário em que a saga "Create Order" executa de maneira concorrente com uma saga "Cancel Order".
 - ▶ A menos que a saga use "semantic lock", é possível que a saga "Cancel Order" cancele o pedido antes da saga "Create Order" ainda estiver na etapa de autorizar o cartão.
 - ▶ Uma forma de tratar essas requisições fora de ordem é registrar as operações na ordem que elas chegam e executá-las na ordem correta.
 - ▶ Neste cenário, seria registrada primeiro a operação "Cancel Authorization" (demandada pela saga "Cancel Order") e então a ação "Authorize Card" seria invalidada, já que não poderia ser executada nesta ordem.

COUNTERMEASURE: BY VALUE

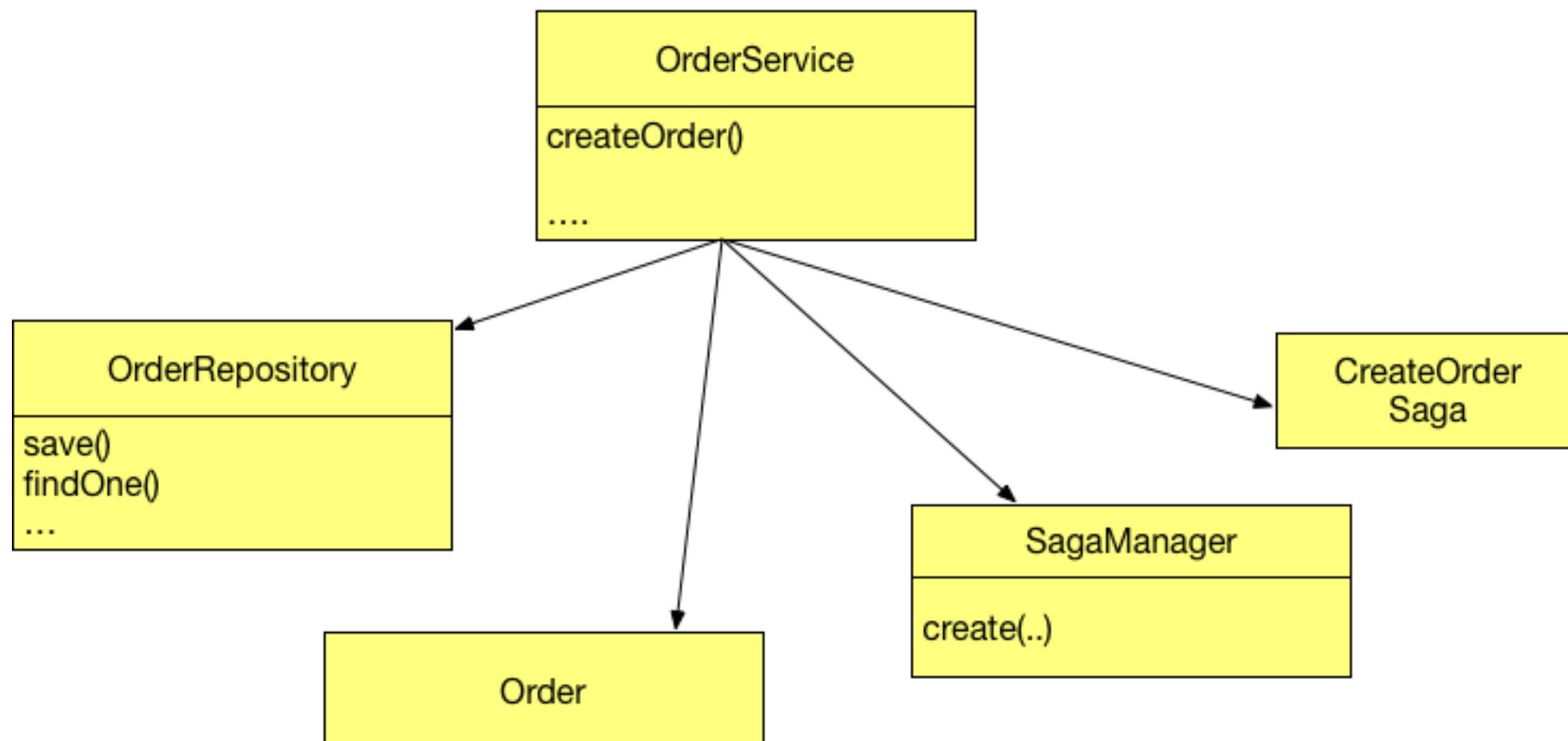
- ▶ Usa as propriedades de cada requisição para decidir entre usar sagas e transações distribuídas.
- ▶ Executa requisições de baixo-risco usando sagas, talvez aplicando as contramedidas vistas até então.
- ▶ No entanto, executa requisições de alto risco, envolvendo, por exemplo, altos valores monetários, usando transações distribuídas.
- ▶ Essa estratégia permite que a aplicação faça os trade-offs considerando risco de negócio, disponibilidade e escalabilidade.

PROJETANDO O “ORDER SERVICE” E A SAGA “CREATE ORDER”

ORDERSERVICE E A SAGA “CREATE ORDER”



A CLASSE ORDER SERVICE



A CLASSE ORDERSERVICE E O MÉTODO CREATEORDER()

```
@Transactional
public class OrderService {

    @Autowired
    private SagaManager<CreateOrderSagaState> createOrderSagaManager;

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private DomainEventPublisher eventPublisher;

    public Order createOrder(OrderDetails orderDetails) {
        ...
        ResultWithEvents<Order> orderAndEvents = Order.createOrder(...);
        Order order = orderAndEvents.result;
        orderRepository.save(order);

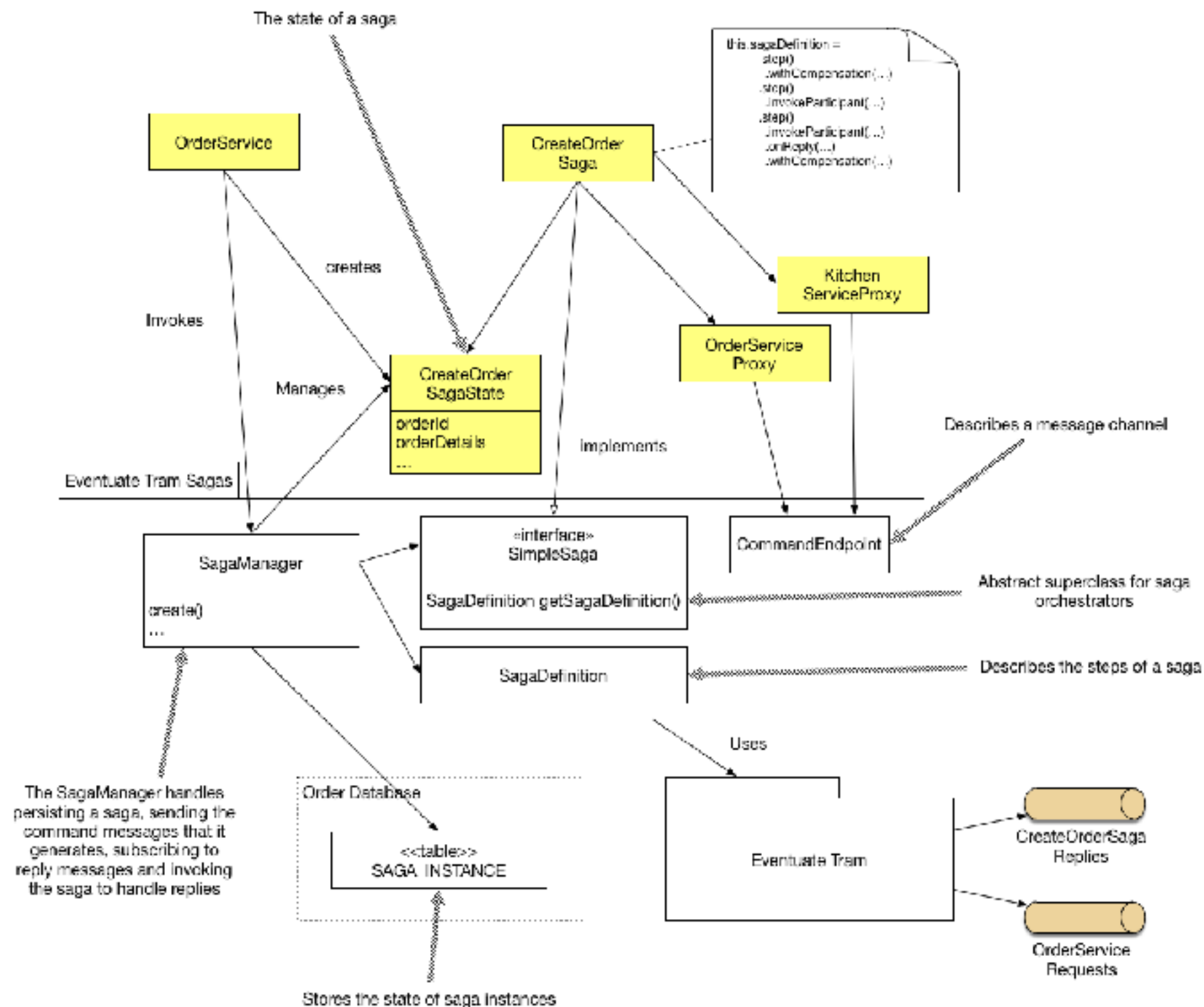
        eventPublisher.publish(Order.class,
                               Long.toString(order.getId()),
                               orderAndEvents.events);

        CreateOrderSagaState data =
            new CreateOrderSagaState(order.getId(), orderDetails);
        createOrderSagaManager.create(data, Order.class, order.getId());

        return order;
    }

    ...
}
```

IMPLEMENTAÇÃO DA SAGA “CREATE ORDER” USANDO EVENTUATE



DEFINIÇÃO DA SAGA “CREATEORDER”

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

    private SagaDefinition<CreateOrderSagaState> sagaDefinition;

    public CreateOrderSaga(OrderServiceProxy orderService,
                           ConsumerServiceProxy consumerService,
                           KitchenServiceProxy kitchenService,
                           AccountingServiceProxy accountingService) {

        this.sagaDefinition =
            step()
                .withCompensation(orderService.reject,
                                CreateOrderSagaState::makeRejectOrderCommand)
            .step()
                .invokeParticipant(consumerService.validateOrder,
                                CreateOrderSagaState::makeValidateOrderByConsumerCommand)
            .step()
                .invokeParticipant(kitchenService.create,
                                CreateOrderSagaState::makeCreateTicketCommand)
                .onReply(CreateTicketReply.class,
                        CreateOrderSagaState::handleCreateTicketReply)
                .withCompensation(kitchenService.cancel,
                                CreateOrderSagaState::makeCancelCreateTicketCommand)
            .step()
                .invokeParticipant(accountingService.authorize,
                                CreateOrderSagaState::makeAuthorizeCommand)
            .step()
                .invokeParticipant(kitchenService.confirmCreate,
                                CreateOrderSagaState::makeConfirmCreateTicketCommand)
            .step()
                .invokeParticipant(orderService.approve,
                                CreateOrderSagaState::makeApproveOrderCommand)
            .build();

    }

    @Override
    public SagaDefinition<CreateOrderSagaState> getSagaDefinition() {
        return sagaDefinition;
    }
}
```


A CLASSE CREATEORDERSagaSTATE

```
public class CreateOrderSagaState {

    private Long orderId;

    private OrderDetails orderDetails;
    private long ticketId;

    public Long getOrderId() {
        return orderId;
    }

    private CreateOrderSagaState() {
    }

    public CreateOrderSagaState(Long orderId, OrderDetails orderDetails) {
        this.orderId = orderId;
        this.orderDetails = orderDetails;
    }

    CreateTicket makeCreateTicketCommand() {
        return new CreateTicket(getOrderDetails().getRestaurantId(),
                                getOrderId(), makeTicketDetails(getOrderDetails()));
    }

    void handleCreateTicketReply(CreateTicketReply reply) {
        logger.debug("getTicketId {}", reply.getTicketId());
        setTicketId(reply.getTicketId());
    }

    CancelCreateTicket makeCancelCreateTicketCommand() {
        return new CancelCreateTicket(getOrderId());
    }

    ...
}
```

A CLASSE KITCHENSERVICEPROXY

- ▶ Define os endpoints de mensagens de comandos para o Kitchen Service:
 - ▶ create - cria um ticket
 - ▶ confirmCreate - confirma a criação
 - ▶ cancel - cancela um ticket
- ▶ Cada CommandEndPoint especifica o type do comando, o canal de destino das mensagens de comando e os tipos esperados de resposta

A CLASSE KITCHENSERVICEPROXY

```
public class KitchenServiceProxy {

    public final CommandEndpoint<CreateTicket> create =
        CommandEndpointBuilder
            .forCommand(CreateTicket.class)
            .withChannel(
                KitchenServiceChannels.kitchenServiceChannel)
            .withReply(CreateTicketReply.class)
            .build();

    public final CommandEndpoint<ConfirmCreateTicket> confirmCreate =
        CommandEndpointBuilder
            .forCommand(ConfirmCreateTicket.class)
            .withChannel(
                KitchenServiceChannels.kitchenServiceChannel)
            .withReply(Success.class)
            .build();

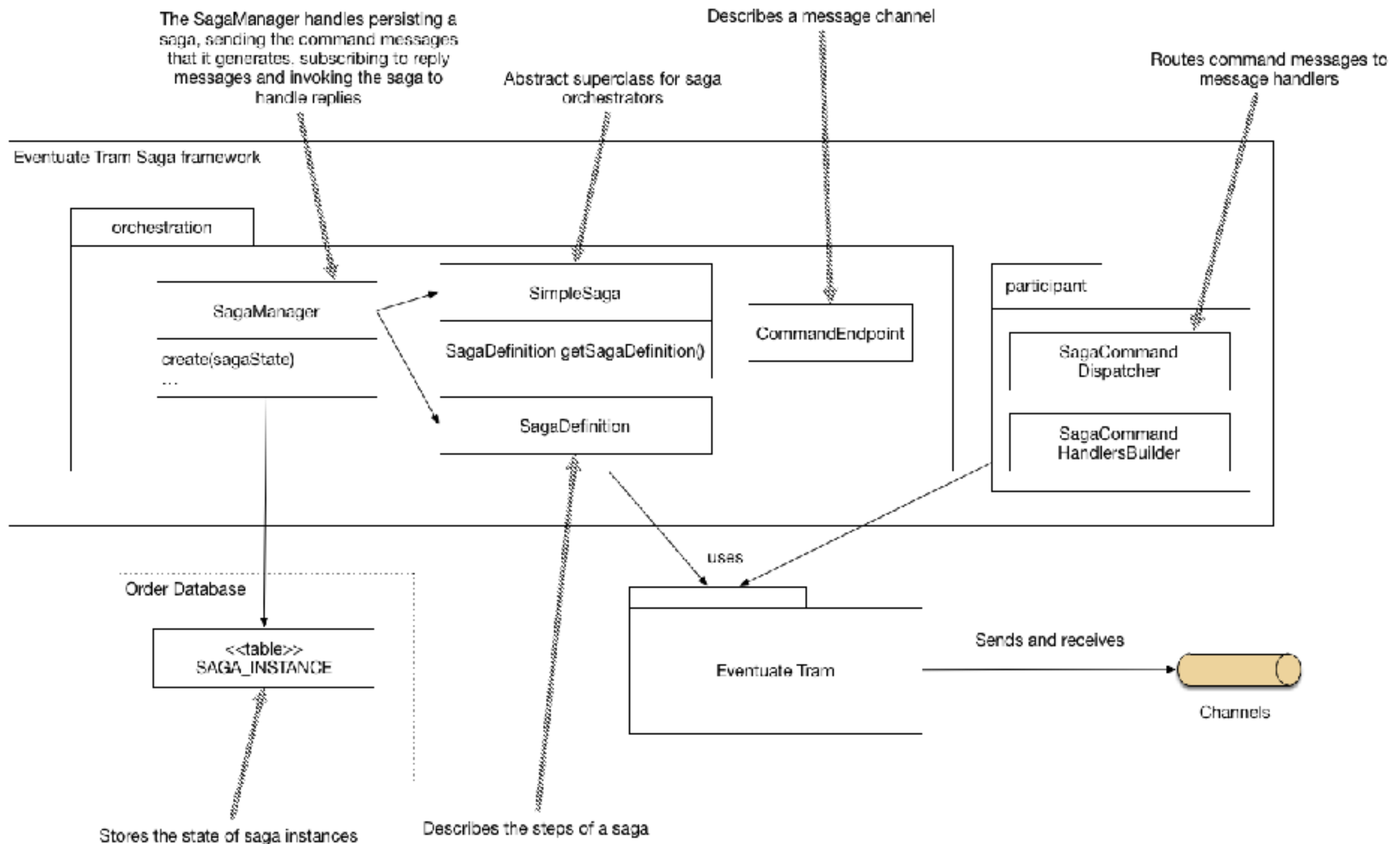
    public final CommandEndpoint<CancelCreateTicket> cancel =
        CommandEndpointBuilder
            .forCommand(CancelCreateTicket.class)
            .withChannel(
                KitchenServiceChannels.kitchenServiceChannel)
            .withReply(Success.class)
            .build();

}
```

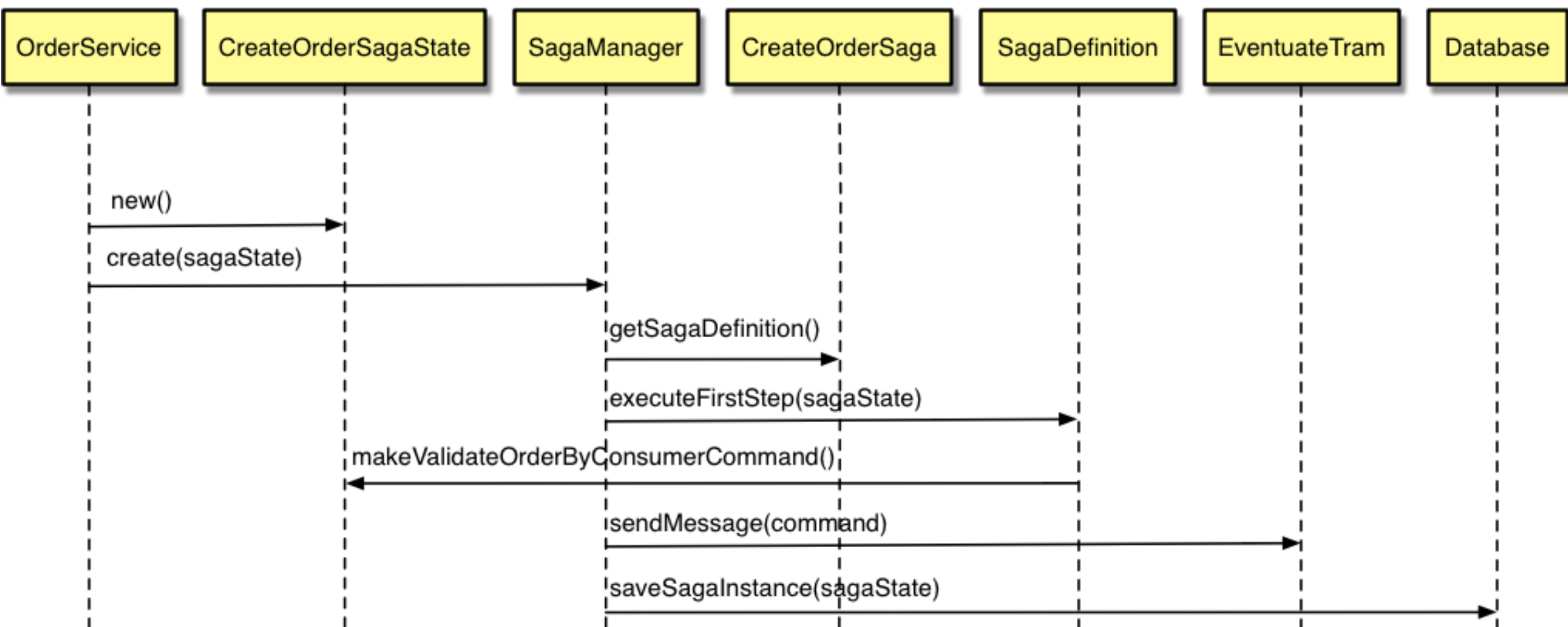
A CLASSE KITCHENSERVICEPROXY

- ▶ Classes não são obrigatórias, a saga poderia simplesmente enviar as mensagens de comando diretamente para os participantes.
- ▶ Porém, há dois importantes benefícios em usar classes proxy:
 - ▶ 1) Uma classe proxy define endpoints estaticamente tipados, o que reduz a chance da saga enviar a mensagem errada para um serviço;
 - ▶ 2) Uma classe proxy é uma API bem-definida para invocar um serviço, o que faz o código mais fácil de entender e testar.

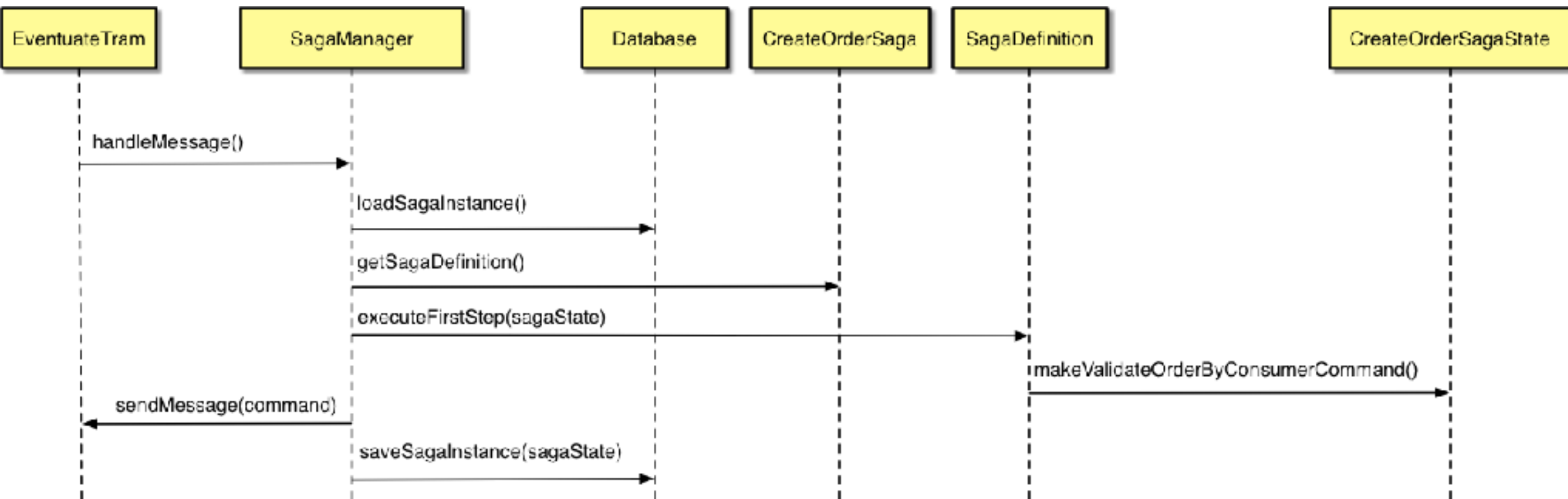
0 FRAMEWORK EVENTUATE TRAM SAGA



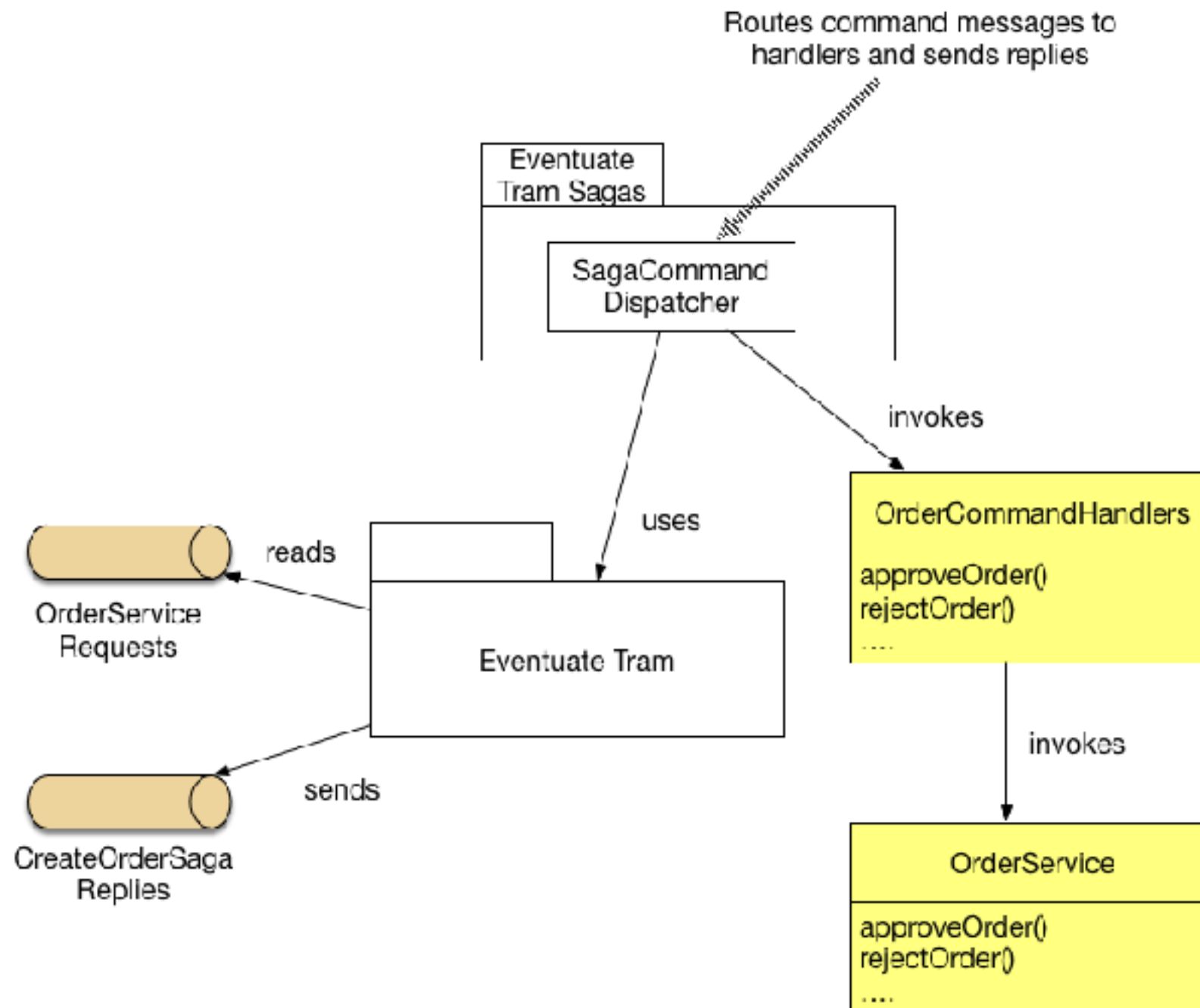
SEQUÊNCIA DE EVENTOS PARA CRIAÇÃO DA SAGA “CREATE ORDER”



SEQUÊNCIA DE EVENTOS QUANDO O SAGAMANAGER RECEBE UMA RESPOSTA



A CLASSE ORDERCOMMANDHANDLERS



A CLASSE ORDERCOMMANDHANDLERS

```
public class OrderCommandHandlers {

    @Autowired
    private OrderService orderService;

    public CommandHandlers commandHandlers() {
        return SagaCommandHandlersBuilder
            .fromChannel("orderService")
            .onMessage(ApproveOrderCommand.class, this::approveOrder)
            .onMessage(RejectOrderCommand.class, this::rejectOrder)
            ...
            .build();
    }

    public Message approveOrder(CommandMessage<ApproveOrderCommand> cm) {
        long orderId = cm.getCommand().getOrderId();
        orderService.approveOrder(orderId);
        return withSuccess();
    }

    public Message rejectOrder(CommandMessage<RejectOrderCommand> cm) {
        long orderId = cm.getCommand().getOrderId();
        orderService.rejectOrder(orderId);
        return withSuccess();
    }
}
```

A CLASSE ORDERSERVICECONFIGURATION

@Configuration

```
public class OrderServiceConfiguration {
```

@Bean

```
public OrderService orderService(RestaurantRepository restaurantRepository,
    ...
    SagaManager<CreateOrderSagaState>
        createOrderSagaManager,
    ...) {
    return new OrderService(restaurantRepository,
        ...
        createOrderSagaManager
        ...);
}
```

@Bean

```
public SagaManager<CreateOrderSagaState> createOrderSagaManager(CreateOrderSaga saga) {
    return new SagaManagerImpl<>(saga);
}
```

@Bean

```
public CreateOrderSaga createOrderSaga(OrderServiceProxy orderService,
    ConsumerServiceProxy consumerService,
    ...) {
    return new CreateOrderSaga(orderService, consumerService, ...);
}
```

@Bean

```
public OrderCommandHandlers orderCommandHandlers() {
    return new OrderCommandHandlers();
}
```

@Bean

```
public SagaCommandDispatcher orderCommandHandlersDispatcher(OrderCommandHandlers orderCommandHandlers) {
    return new SagaCommandDispatcher("orderService", orderCommandHandlers.commandHandlers());
}
```

A CLASSE ORDERSERVICECONFIGURATION

```
...  
  
@Bean  
public KitchenServiceProxy kitchenServiceProxy() {  
    return new KitchenServiceProxy();  
}  
  
@Bean  
public OrderServiceProxy orderServiceProxy() {  
    return new OrderServiceProxy();  
}  
  
...  
}
```

RESUMO

- ▶ Algumas operações envolvem atualização de dados em múltiplos serviços.
- ▶ Abordagens tradicionais baseadas em transações distribuídas X2/2PC não se encaixam em aplicações modernas.
- ▶ Uma melhor abordagem é usar o padrão Saga. Uma saga é uma sequência de transações locais coordenadas pelo uso de mensagens.
- ▶ Cada transação local atualiza os dados em um serviço específico.
- ▶ Como cada transação local comita suas mudanças, se uma saga precisar fazer roll back por alguma violação de regra de negócio, ela deve executar transações de compensação para explicitamente desfazer as mudanças.

RESUMO

- ▶ É possível usar coreografia ou orquestração para coordenar as etapas de uma saga.
- ▶ Na coreografia, uma transação local publica eventos, que dispara outros participantes a executar transações locais.
- ▶ Na orquestração, um orquestrador envia mensagens para os participantes dizendo o quais transações locais eles devem executar.
- ▶ É possível desenvolver e usar orquestradores como máquinas de estados.
- ▶ Sagas simples podem usar coreografia, mas orquestração é geralmente uma melhor escolha para sagas complexas.

RESUMO

- ▶ Modelar lógica de negócio baseada em sagas pode ser desafiador porque as sagas não são isoladas umas das outras.
- ▶ Geralmente é preciso usar contramedidas, que são estratégias de *design* para evitar anomalias de concorrência causadas pelo modelo transacional ACD.
- ▶ Uma aplicação deve considerar o uso de locking no sentido de simplificar a lógica de negócio, mesmo que haja o risco de deadlocks.