



Published in OLX Brasil Tech



Raphael Amoedo

Follow

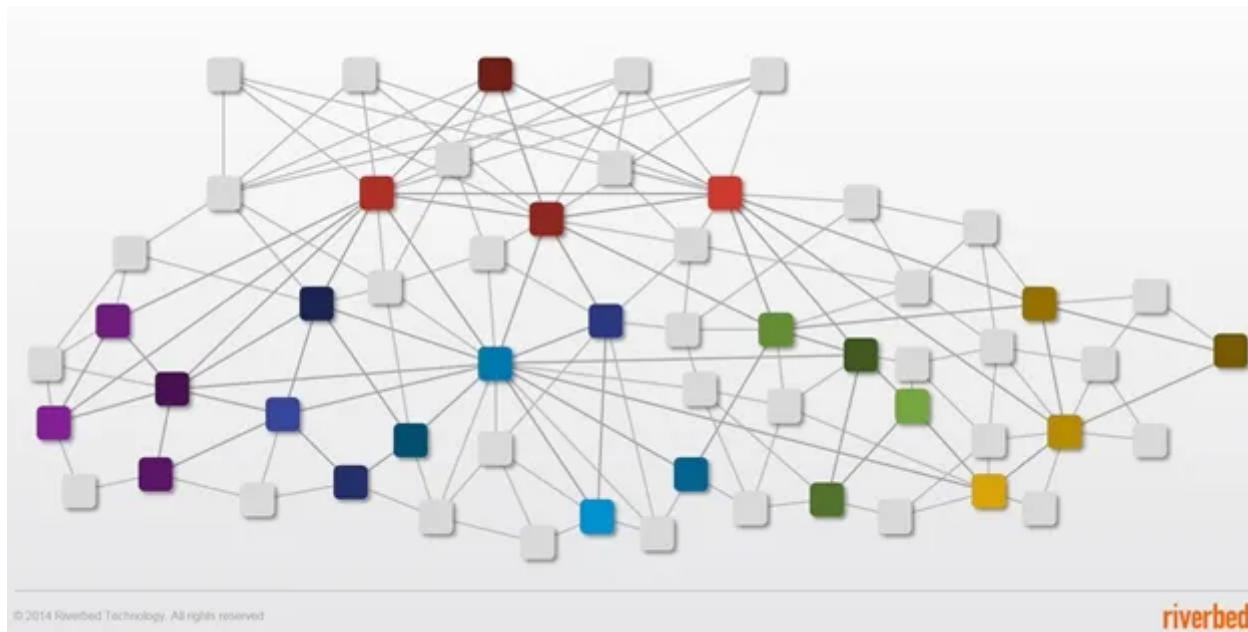
Oct 31, 2018 · 5 min read



Save



Transações distribuídas em microserviços



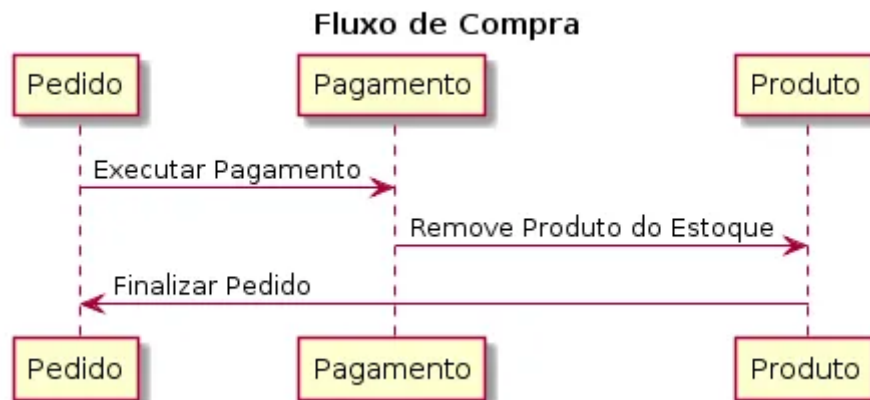
<https://www.riverbed.com/wp-content/uploads/2017/10/microservices.jpg>

O principal objetivo de uma transação é garantir a integridade e a consistência dos dados. Dentro de um contexto de microserviços, é comum que cada microserviço possua seu próprio banco de dados. Como garantir a consistência dos dados entre bancos e aplicações/serviços diferentes?

Uma transação é basicamente uma sequência de operações e visa garantir que todas sejam executadas, ou nenhuma, caso uma das operações falhe. Uma transação se torna distribuída quando ativa operações em vários servidores diferentes.

Imagina o seguinte cenário: temos 3 microserviços — serviço de Pedido, serviço de Produto e serviço de Pagamento — e precisamos garantir a consistência entre eles. Uma pequena descrição do fluxo desejado no nosso exemplo seria:

- Ao criar um pedido, deve-se processar o pagamento
- Quando o pagamento for processado, o estoque do produto deve ser atualizado
- Quando o estoque do produto for atualizado, o pedido deve ser finalizado



Exemplo do fluxo de compra que queremos

Dentre algumas das maneiras de resolver esse problema, apresentarei o padrão **Saga**, que é uma sequência de transações que representam um processo de negócio.

Existem duas formas mais comuns de implementação Saga:

Coreografia (Choreography) e Orquestração (Orchestration)

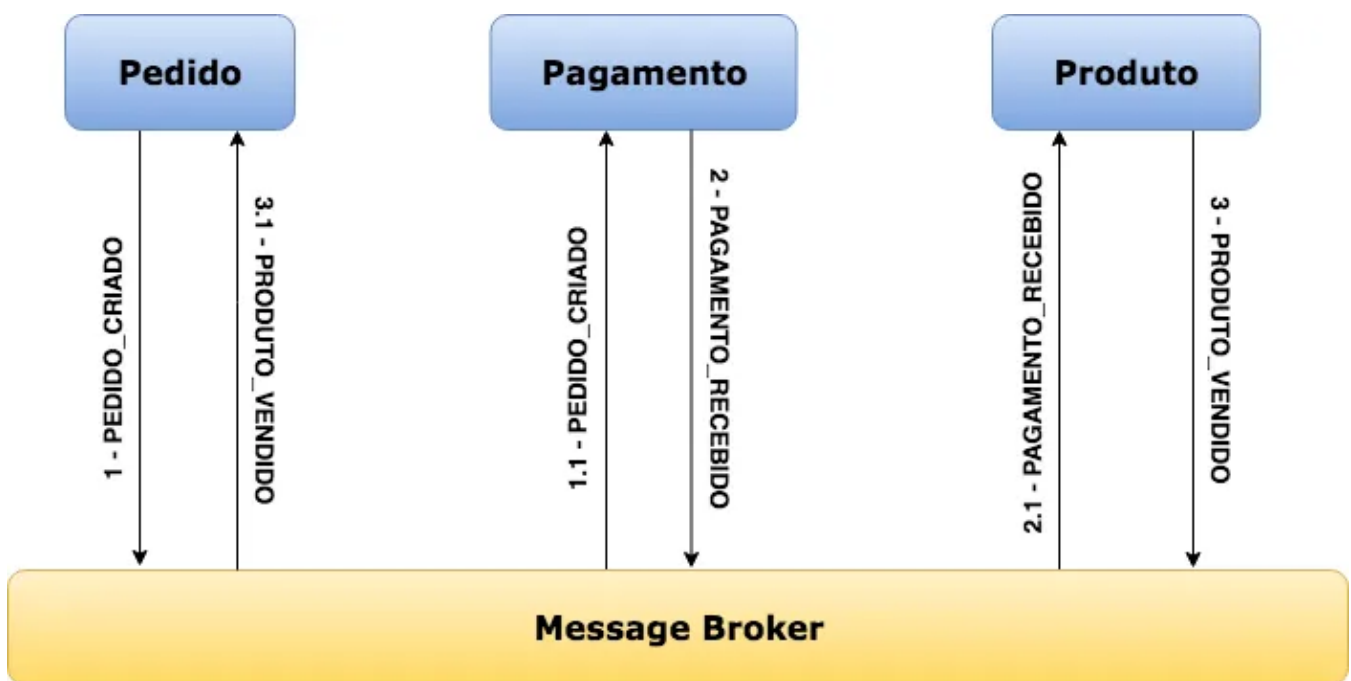
Coreografia

É baseada em eventos. Cada serviço sabe qual evento disparar e qual evento ouvir para que a saga seja completa.

Seguindo o nosso exemplo acima, utilizando coreografia o fluxo seria:

- Ao criar um pedido, o serviço Pedido dispararia um evento que vamos chamar de **PEDIDO_CRIADO**.
- O serviço de Pagamento ouve esse evento.
- Pagamento recebe o evento de **PEDIDO_CRIADO** e processa o pagamento.

- Após o pagamento ser processado, o serviço Pagamento dispara um evento que vamos chamar de **PAGAMENTO_RECEBIDO**.
- Produto ouve e recebe o evento de **PAGAMENTO_RECEBIDO** e remove o produto do estoque.
- Após a remoção do produto do estoque, Produto dispara um evento que vamos chamar de **PRODUTO_VENDIDO**.
- O serviço Pedido ouve esse evento, recebe o evento e finaliza o pedido.



Bem fácil, bem simples e intuitivo. Entretanto, como citamos no começo: *O principal objetivo de uma transação é garantir a integridade e a consistência dos dados*. Sendo assim, como esse simples exemplo deveria se comportar, caso o pagamento falhe ou o produto não exista no estoque? Como funciona o **rollback** nesse caso?

- Caso o pagamento falhe, devemos cancelar o pedido.
- Caso o produto não exista no estoque, devemos desfazer o pagamento feito e cancelar o pedido.

Sendo assim, para que isso aconteça: caso o pagamento falhe, devemos disparar um evento como por exemplo **PAGAMENTO_NAO_RECEBIDO**. O serviço Pedido deve ouvir esse evento e cancelar o pedido.

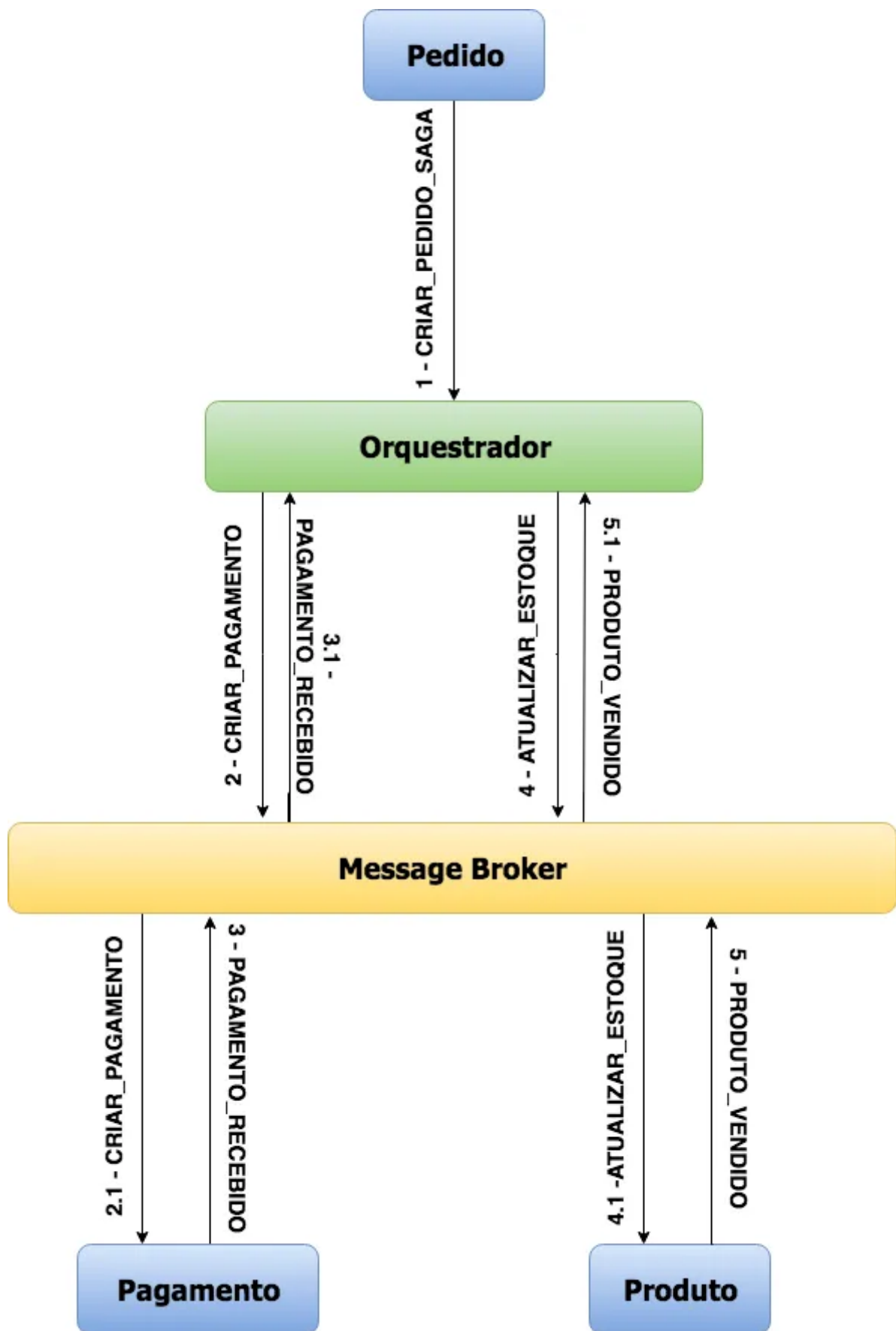
Da mesma forma, caso o produto não exista no estoque, o serviço Produto deve disparar um evento como por exemplo **PRODUTO_SEM_ESTOQUE**. O serviço Pagamento e o serviço Pedido devem ouvir esse evento e desfazer(reembolsar) o pagamento e cancelar o pedido, respectivamente.

Orquestração

É baseada em comandos. Existe uma espécie de coordenador de execução de sagas e ele é que sabe quais eventos disparar dada uma determinada saga. Ele pode ser um objeto, um serviço à parte, etc.

Ainda seguindo o nosso exemplo acima, utilizando orquestração o fluxo seria:

- Ao criar um pedido, o serviço Pedido acionaria a saga que vamos chamar de **CRIAR_PEDIDO_SAGA** no orquestrador/coordenador.
- O orquestrador sabe quais os passos devem ser executados para que a saga **CRIAR_PEDIDO_SAGA** seja completa, então ele dispara o comando de **CRIAR_PAGAMENTO**.
- O serviço Pagamento recebe esse comando, cria o pagamento e retorna a resposta em um canal (channel) de mensageria.
- O orquestrador recebe e sabe que o próximo passo é **ATUALIZAR_ESTOQUE**, então ele dispara esse comando.
- O serviço Produto recebe esse comando, atualiza o estoque e retorna a resposta em um canal (channel) de mensageria.
- O orquestrador recebe, sabe que o próximo passo é encerrar o Pedido (e por aí vai).



Como funcionaria o rollback nesse caso? Então... Seguiria o mesmo exemplo utilizado na coreografia:

Caso o pagamento falhe, devemos retornar a resposta do **PAGAMENTO_NAO_RECEBIDO**. O **orquestrador** deve receber essa resposta e

cancelar o pedido. A mesma lógica com o **PRODUTO_SEM_ESTOQUE**.

Benefícios e Desvantagens

Como não poderia deixar de ser, tudo tem seus prós e contras.

Coreografia:

Prós:

- Mais fácil de entender
- Fácil implementação
- Serviços desacoplados sem conhecer diretamente um ao outro
- Não possui um único ponto de falha

Contras:

- Quanto maior os passos, mais difícil o tracking
- Pode acabar adicionando uma dependência cíclica entre serviços — uma vez que eles subscrevem nos eventos entre si

Open in app ↗

Sign up

Sign In



Prós:

- Evita dependência cíclica (o orquestrador depende dos participantes mas os participantes não dependem do orquestrador)
- Centralização da orquestração da transação distribuída
- Reduz a complexidade dos participantes — uma vez que eles precisam apenas executar e responder comandos
- Mais fácil de ser testado e implementado
- A complexidade da transação se mantém linear mesmo quando novos passos são adicionados
- Rollbacks são mais fáceis de gerenciar




- Se existe uma segunda transação que quer mudar o mesmo objeto você pode colocar em espera no orquestrador até a primeira transação acabar

Contras:

- Corre o risco do orquestrador ter muita lógica
- Aumenta a complexidade da infraestrutura pois tem que lidar com um serviço extra
- Ponto único de falha
- Algumas novas funcionalidades podem significar mudanças em múltiplos serviços e então ordenar releases

Ferramentas

Algumas ferramentas encontradas para facilitar o uso desse padrão:

- [EventFlow](#)
- [Axon Framework](#)
- [MementoFX](#)
- [Eventuate.io](#)
- [AWS Step Functions](#) (pode s  150 |  2 |  1)

A seguir um exemplo de como seria a orquestração usando uma ferramenta do Eventuate chamada [eventuate-tram-sagas](#):

```
// A definição do CreateOrderSaga
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();
...

```

E a seguir o exemplo de um participante da Saga:

```

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) {
        ...
    }
    ...
}

```

Referências

<https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part/>

<https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part-2/>