Projeto: Sistema de Comércio Eletrônico Distribuído

Computação Distribuída

Table of contents

1 Visão Geral do Projeto **2**
- 1.1 Objetivo . . . 2
- 1.2 Contexto: O Fluxo de Checkout . . . 2

2 Especificação Funcional **2**

3 Parte 1: Implementação com Arquitetura Síncrona (RPC) **3**
- 3.1 Etapa A: gRPC . . . 3
- 3.2 Etapa B: Web Services REST . . . 4
- 3.3 Banco de Dados . . . 4

4 Parte 2: Implementação com Arquitetura Assíncrona (Publish-Subscribe) **4**
- 4.1 Banco de Dados . . . 4

5 Geração de Dados Sintéticos **5**

6 Guia Detalhado de Implementação **9**
- 6.1 1. Configuração do Ambiente . . . 9
- 6.2 2. Implementação da Parte 1 (RPC) . . . 9
  - 6.2.1 Exemplo Mínimo (gRPC - Pagamento) . . . 10
- 6.3 3. Implementação da Parte 2 (Kafka) . . . 11
  - 6.3.1 Exemplo Mínimo (Kafka - Consumidor de Pagamentos e Produtor para Estoque) . . . 12
- 6.4 4. Testes e Comparação . . . 15

7 Relatório do Projeto **16**

8 Avaliação **16**

1 Visão Geral do Projeto

1.1 Objetivo

Este projeto tem como objetivo principal proporcionar uma experiência prática com diferentes paradigmas de comunicação em sistemas distribuídos. Você irá implementar um subsistema de checkout de um comércio eletrônico utilizando duas abordagens distintas:

1. **Comunicação Síncrona baseada em RPC:** Utilizando gRPC ou Web Services REST.
2. **Comunicação Assíncrona baseada em Publish-Subscribe:** Utilizando Apache Kafka.

Ao final, deverá ser feita uma comparação das duas arquiteturas, analisando suas características, vantagens, desvantagens e, principalmente, seu comportamento em relação à escalabilidade e ao tratamento de altas cargas de trabalho.

1.2 Contexto: O Fluxo de Checkout

Imagine um sistema de comércio eletrônico. Quando um cliente finaliza a compra (checkout), diversas etapas precisam ser executadas em sequência ou em paralelo:

1. **Criação do Pedido:** O pedido inicial é registrado no sistema.
2. **Processamento do Pagamento:** O pagamento é verificado e autorizado.
3. **Separação de Estoque:** Os itens do pedido são localizados e separados no estoque.
4. **Emissão da Nota Fiscal:** Uma nota fiscal é gerada para o pedido.
5. **Envio para Transportadora:** O pedido é despachado para a empresa de logística/transportadora.

Este fluxo envolve múltiplos "serviços" ou responsabilidades que precisam se comunicar.

> **Atenção**
>
> Todos os códigos ou trechos de códigos deste documento podem conter alguns erros (foram reutilizados de tutoriais, blogs, repositórios); portanto, preste mais atenção às especificações para entender e identificar possíveis erros nos códigos de exemplo.

2 Especificação Funcional

Você deverá implementar os seguintes serviços (simplificados):

1. **Serviço de Pedidos:** Responsável por receber a requisição inicial de checkout e iniciar o fluxo. Persiste informações básicas do pedido.
2. **Serviço de Pagamentos:** Responsável por simular o processamento de pagamento. Recebe detalhes do pedido, atualiza o status do pagamento e notifica o próximo passo.
3. **Serviço de Estoque:** Responsável por simular a verificação e separação de itens no estoque. Recebe a confirmação de pagamento, atualiza o status do estoque e notifica o próximo passo.
4. **Serviço Fiscal:** Responsável por simular a emissão da nota fiscal. Recebe a confirmação da separação do estoque, gera a nota e notifica o próximo passo.
5. **Serviço de Logística:** Responsável por simular o envio do pedido para a transportadora. Recebe a nota fiscal, agenda o envio e finaliza o processo (para este escopo).
6. **Serviço de Catálogo:** Responsável por enviar ao cliente a lista de produtos em estoque, contendo ID e Estoque de cada produto.
    - REST: um endpoint GET /produtos que retorna um JSON array, ex: `[{"id": 1, "nome": "Produto A", "estoque": 10}, {"id": 5, "nome": "Produto B", "estoque": 0}, {"id": 12, "nome": "Produto C", "estoque": 5}]`
    - gRPC: um serviço Catalogo com um método ListarProdutos que retorna uma mensagem contendo uma lista de produtos com seus IDs e talvez estoque. Ex: `message ProdutoInfo { int32 id = 1; string nome = 2; int32 estoque = 3; } message ListaProdutosResponse { repeated ProdutoInfo produtos = 1; }`

Cada serviço deve, no mínimo, registrar a operação que realizou (por exemplo, em um log ou no seu próprio banco de dados SQLite).

3 Parte 1: Implementação com Arquitetura Síncrona (RPC)

Nesta parte, a comunicação entre os serviços será síncrona. Um serviço chama o próximo diretamente e espera pela resposta antes de continuar. Será realizado em etapas, iniciando com **gRPC** e depois **Web Services REST**.

3.1 Etapa A: gRPC

- **Definição:** Utilizar Protocol Buffers (`.proto`) para definir as mensagens e as assinaturas dos serviços (ex: `ProcessarPagamento`, `SepararEstoque`).
- **Implementação:** Criar servidores gRPC para cada serviço e clientes gRPC dentro dos serviços que precisam chamar outros.
- **Comunicação:** Chamadas RPC diretas e síncronas. Ex: Serviço de Pedidos chama `ProcessarPagamento` no Serviço de Pagamentos.

3.2 Etapa B: Web Services REST

- **Definição:** Definir APIs RESTful para cada serviço (endpoints, métodos HTTP - ex: `POST /pagamentos`, `POST /estoque/separar` - e formato JSON para os dados).
- **Implementação:** Utilizar um framework web como Flask ou FastAPI para criar os servidores REST. Utilizar bibliotecas como `requests` para fazer as chamadas HTTP entre os serviços.
- **Comunicação:** Requisições HTTP síncronas. Ex: Serviço de Pedidos faz um `POST` para o endpoint de pagamento.

3.3 Banco de Dados

Cada serviço pode utilizar um arquivo SQLite separado para persistir seu estado (ex: `pedidos.db`, `pagamentos.db`, `estoque.db`, etc.).

4 Parte 2: Implementação com Arquitetura Assíncrona (Publish-Subscribe)

Nesta parte, a comunicação será assíncrona, utilizando Apache Kafka como message broker.

- **Tópicos Kafka:** Definir tópicos para representar os eventos de negócio (ex: `pedidos_criados`, `pagamentos_processados`, `estoque_separado`, `nf_emitida`).
- **Produtores:** Quando um serviço conclui sua tarefa, ele publica uma mensagem (evento) no tópico apropriado. Ex: Serviço de Pagamentos publica no tópico `pagamentos_processados` após processar um pagamento.
- **Consumidores:** Os serviços se inscrevem (subscribe) nos tópicos que lhes interessam para iniciar seu trabalho. Ex: Serviço de Estoque consome mensagens do tópico `pagamentos_processados`.
- **Comunicação:** Os serviços são desacoplados. Um produtor não sabe (e não precisa saber) quais ou quantos consumidores receberão a mensagem.

4.1 Banco de Dados

Similar à Parte 1, cada serviço mantém seu próprio estado em um banco de dados SQLite.

5 Geração de Dados Sintéticos

Para simular um ambiente com dados preexistentes e permitir testes mais realistas, utilize o script Python abaixo para gerar um banco de dados SQLite (`ecommerce.db`) com dados sintéticos.

```
import sqlite3
import random
from faker import Faker
from datetime import datetime
import os

fake = Faker('pt_BR')

def criar_conexao(db_file):
    """Cria uma conexão com o banco de dados SQLite especificado por db_file"""
    conn = None
    try:
        conn = sqlite3.connect(db_file)
        print(f"Conectado ao SQLite versão {sqlite3.sqlite_version}")
        return conn
    except sqlite3.Error as e:
        print(f"Erro ao conectar ao banco de dados: {e}")
    return conn

def criar_tabela(conn, create_table_sql):
    """Cria uma tabela a partir da instrução SQL create_table_sql"""
    try:
        c = conn.cursor()
        c.execute(create_table_sql)
    except sqlite3.Error as e:
        print(f"Erro ao criar tabela: {e}")

# --- Definições das Tabelas ---
sql_create_produtos_table = """ CREATE TABLE IF NOT EXISTS produtos (
                                    id INTEGER PRIMARY KEY,
                                    nome TEXT NOT NULL,
                                    preco REAL NOT NULL,
                                    quantidade_estoque INTEGER NOT NULL
                                ); """

sql_create_clientes_table = """ CREATE TABLE IF NOT EXISTS clientes (
                                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                                    nome TEXT NOT NULL,
                                    email TEXT NOT NULL UNIQUE
                                ); """

sql_create_pedidos_table = """ CREATE TABLE IF NOT EXISTS pedidos (
                                id INTEGER PRIMARY KEY AUTOINCREMENT,
                                cliente_id INTEGER NOT NULL,
                                data_criacao TEXT NOT NULL,
                                status TEXT NOT NULL, -- Ex: 'pendente', 'pagamento_aprovado',
                                     'em_separacao', 'nf_emitida', 'enviado', 'cancelado'
                                valor_total REAL,
                                FOREIGN KEY (cliente_id) REFERENCES clientes (id)
                            ); """

sql_create_itens_pedido_table = """ CREATE TABLE IF NOT EXISTS itens_pedido (
```

```
                                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                                    pedido_id INTEGER NOT NULL,
                                    produto_id INTEGER NOT NULL,
                                    quantidade INTEGER NOT NULL,
                                    preco_unitario REAL NOT NULL,
                                    FOREIGN KEY (pedido_id) REFERENCES pedidos (id),
                                    FOREIGN KEY (produto_id) REFERENCES produtos (id)
                                ); """

sql_create_pagamentos_table = """ CREATE TABLE IF NOT EXISTS pagamentos (
                                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                                    pedido_id INTEGER NOT NULL UNIQUE, -- Geralmente 1 pagamento por pedido
                                    data_processamento TEXT NOT NULL,
                                    status TEXT NOT NULL, -- Ex: 'aprovado', 'rejeitado', 'pendente'
                                    metodo TEXT,
                                    FOREIGN KEY (pedido_id) REFERENCES pedidos (id)
                                ); """

sql_create_notas_fiscais_table = """ CREATE TABLE IF NOT EXISTS notas_fiscais (
                                        id INTEGER PRIMARY KEY AUTOINCREMENT,
                                        pedido_id INTEGER NOT NULL UNIQUE,
                                        numero TEXT NOT NULL UNIQUE,
                                        data_emissao TEXT NOT NULL,
                                        chave_acesso TEXT,
                                        FOREIGN KEY (pedido_id) REFERENCES pedidos (id)
                                    ); """

sql_create_envios_table = """ CREATE TABLE IF NOT EXISTS envios (
                                id INTEGER PRIMARY KEY AUTOINCREMENT,
                                pedido_id INTEGER NOT NULL UNIQUE,
                                nota_fiscal_id INTEGER NOT NULL UNIQUE,
                                data_despacho TEXT,
                                codigo_rastreamento TEXT,
                                status TEXT NOT NULL, -- Ex: 'aguardando_envio', 'enviado', 'entregue'
                                FOREIGN KEY (pedido_id) REFERENCES pedidos (id),
                                FOREIGN KEY (nota_fiscal_id) REFERENCES notas_fiscais (id)
                            ); """


def popular_dados(conn, num_clientes=100, num_produtos=50, num_pedidos=200):
    """Popula o banco de dados com dados sintéticos"""
    cursor = conn.cursor()

    print("Populando tabela clientes...")
    clientes_ids = []
    for i in range(num_clientes):
        nome = fake.name()
        email = f"cliente_{i}_{fake.user_name()}@exemplo.com" # Garante unicidade
        cursor.execute("INSERT INTO clientes (nome, email) VALUES (?, ?)", (nome, email))
        clientes_ids.append(cursor.lastrowid)

    print("Populando tabela produtos...")
    produtos_ids = []
    for i in range(num_produtos):
        nome = f"Produto {fake.word().capitalize()} {random.choice(['Eletrônico', 'Vestuário',
            'Livro', 'Alimento'])}"
        preco = round(random.uniform(10.5, 500.99), 2)
        estoque = random.randint(0, 100)
        cursor.execute("INSERT INTO produtos (id, nome, preco, quantidade_estoque)
            VALUES (?, ?, ?, ?)", (i + 1, nome, preco, estoque))
```

```
        produtos_ids.append(i + 1)

    print("Populando tabelas pedidos e itens_pedido...")
    pedido_ids_criados = []
    for i in range(num_pedidos):
        cliente_id = random.choice(clientes_ids)
        data_criacao = fake.date_time_between(start_date="-90d", end_date="now").isoformat()
        status_inicial = 'pendente'
        cursor.execute("INSERT INTO pedidos (cliente_id, data_criacao, status) VALUES (?, ?, ?)",
                       (cliente_id, data_criacao, status_inicial))
        pedido_id = cursor.lastrowid
        pedido_ids_criados.append(pedido_id)

        # Adiciona itens ao pedido
        num_itens = random.randint(1, 5)
        valor_total_pedido = 0
        for _ in range(num_itens):
            produto_id = random.choice(produtos_ids)
            # Busca preço do produto para consistência
            cursor.execute("SELECT preco FROM produtos WHERE id = ?", (produto_id,))
            resultado = cursor.fetchone()
            if resultado:
                 preco_unitario = resultado[0]
                 quantidade = random.randint(1, 3)
                 valor_total_pedido += quantidade * preco_unitario
                 cursor.execute("""INSERT INTO itens_pedido
                                   (pedido_id, produto_id, quantidade, preco_unitario)
                                   VALUES (?, ?, ?, ?)""",
                                   (pedido_id, produto_id, quantidade, preco_unitario))
            else:
                print(f"Alerta: Produto ID {produto_id} não encontrado ao adicionar item ao pedido
                       {pedido_id}")

        # Atualiza valor total do pedido
        cursor.execute("UPDATE pedidos SET valor_total = ? WHERE id = ?", (round(valor_total_pedido, 2),
             pedido_id))

    # Simula alguns status posteriores (Pagamentos, NFs, Envios) para alguns pedidos
    print("Simulando status posteriores...")
    for pedido_id in pedido_ids_criados:
        if random.random() < 0.8: # 80% dos pedidos terão pagamento processado
             data_pagamento = fake.date_time_between(start_date="-80d", end_date="now").isoformat()
             status_pagamento = random.choice(['aprovado', 'rejeitado'])
             metodo = random.choice(['cartao_credito', 'boleto', 'pix'])
             cursor.execute("""INSERT INTO pagamentos
                               (pedido_id, data_processamento, status, metodo)
                               VALUES (?, ?, ?, ?)""",
                               (pedido_id, data_pagamento, status_pagamento, metodo))

             if status_pagamento == 'aprovado':
                 cursor.execute("UPDATE pedidos SET status = 'pagamento_aprovado' WHERE id = ?", (pedido_id,))

                 if random.random() < 0.9: # 90% dos aprovados terão NF emitida
                     cursor.execute("UPDATE pedidos SET status = 'em_separacao' WHERE id = ?",
                           (pedido_id,)) # Simula passagem pelo estoque

                     data_nf = fake.date_time_between(start_date="-70d", end_date="now").isoformat()
                     numero_nf = f"NF{random.randint(10000, 99999)}-{pedido_id}"
                     chave_nf = fake.ean(length=13) + fake.ean(length=13) + fake.ean(length=13) # Simula chave longa
                     cursor.execute("""INSERT INTO notas_fiscais
```

```
                                    (pedido_id, numero, data_emissao, chave_acesso)
                                    VALUES (?, ?, ?, ?)""",
                                    (pedido_id, numero_nf, data_nf, chave_nf))
                    nf_id = cursor.lastrowid
                    cursor.execute("UPDATE pedidos SET status = 'nf_emitida' WHERE id = ?", (pedido_id,))

                    if random.random() < 0.95: # 95% dos com NF serão enviados
                        data_envio = fake.date_time_between(start_date="-60d", end_date="now").isoformat()
                        rastreio = f"BR{random.randint(100000000, 999999999)}PY"
                        status_envio = random.choice(['aguardando_envio', 'enviado', 'entregue'])
                        cursor.execute("""INSERT INTO envios
                                       (pedido_id, nota_fiscal_id, data_despacho, codigo_rastreamento, status)
                                       VALUES (?, ?, ?, ?, ?)""",
                                       (pedido_id, nf_id, data_envio, rastreio, status_envio))
                        if status_envio in ['enviado', 'entregue']:
                             cursor.execute("UPDATE pedidos SET status = ? WHERE id = ?", (status_envio, pedido_id,))


    conn.commit()
    print("População de dados concluída.")

# --- Função Principal ---
def main():
    database = "ecommerce.db"

    # Apaga o banco de dados antigo, se existir, para começar do zero
    if os.path.exists(database):
        os.remove(database)
        print(f"Banco de dados '{database}' existente removido.")

    # Cria uma conexão com o banco de dados (isso criará o arquivo se não existir)
    conn = criar_conexao(database)

    # Cria as tabelas
    if conn is not None:
        print("Criando tabelas...")
        criar_tabela(conn, sql_create_clientes_table)
        criar_tabela(conn, sql_create_produtos_table)
        criar_tabela(conn, sql_create_pedidos_table)
        criar_tabela(conn, sql_create_itens_pedido_table)
        criar_tabela(conn, sql_create_pagamentos_table)
        criar_tabela(conn, sql_create_notas_fiscais_table)
        criar_tabela(conn, sql_create_envios_table)
        print("Tabelas criadas com sucesso.")

        # Popula as tabelas com dados
        popular_dados(conn, num_clientes=50, num_produtos=30, num_pedidos=100) # Ajuste os números conforme necessário

        conn.close()
        print(f"Conexão com '{database}' fechada.")
    else:
        print("Erro! Não foi possível criar a conexão com o banco de dados.")

if __name__ == '__main__':
    # Instale as dependências se necessário: pip install Faker
    main()
```

Como Usar o Gerador:

1. Certifique-se de ter Python instalado.
2. Instale a biblioteca `Faker`: `pip install Faker`.
3. Salve o código acima como um arquivo Python (ex: `gerador_dados.py`).
4. Execute o script: `python gerador_dados.py`.
5. Um arquivo `ecommerce.db` será criado (ou substituído) no mesmo diretório, contendo as tabelas e os dados sintéticos.

6 Guia Detalhado de Implementação

6.1 1. Configuração do Ambiente

- **Python:** Instale Python 3.8 ou superior.
- **Bibliotecas:**
  - **Comum:** `pip install Faker` (para o gerador), `sqlite3` (já vem com Python).
  - **Opção gRPC:** `pip install grpcio grpcio-tools protobuf`
  - **Opção REST:** `pip install Flask requests` ou `pip install fastapi uvicorn requests`
  - **Opção Kafka:** `pip install confluent-kafka`
- **Apache Kafka:** Para a Parte 2, você precisará de um cluster Kafka em execução. Siga esse tutorial para instalar https://kafka.apache.org/quickstart

6.2 2. Implementação da Parte 1 (RPC)

- **Estrutura:** Crie diretórios separados para cada serviço (ex: `servico_pedidos`, `servico_pagamentos`, etc.). Cada diretório conterá o código do servidor e, se necessário, o código do cliente para chamar outros serviços.
- **gRPC:**
  1. **Defina os `.proto`:** Crie arquivos `.proto` definindo as mensagens (requests/responses) e os `service`s com seus métodos RPC.
  2. **Gere o Código Python:** Use `python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. *.proto` para gerar os stubs.
  3. **Implemente os Servidores:** Crie classes que herdam do `Servicer` gerado e implemente a lógica de negócio para cada método RPC. Interaja com o banco SQLite local do serviço.
  4. **Implemente os Clientes:** Nos serviços que precisam chamar outros, use os stubs gerados para criar canais (`grpc.insecure_channel`) e fazer as chamadas RPC.
- **REST:**

1. **Implemente os Servidores (Flask/FastAPI):** Defina rotas (endpoints) que correspondem às ações (ex: `/pedidos`, `/pagamentos/{pedido_id}`). Implemente a lógica nessas rotas, interagindo com o SQLite local. Use JSON para entrada e saída.
2. **Implemente os Clientes:** Nos serviços que precisam chamar outros, use a biblioteca `requests` para fazer chamadas HTTP (GET, POST, etc.) para os endpoints dos outros serviços.

6.2.1 Exemplo Mínimo (gRPC - Pagamento)

```
// pagamento.proto
syntax = "proto3";

package pagamento;

message PagamentoRequest {
  int32 pedido_id = 1;
  float valor = 2;
}

message PagamentoResponse {
  int32 pedido_id = 1;
  bool sucesso = 2;
  string mensagem = 3;
}

service Pagamento {
  rpc ProcessarPagamento (PagamentoRequest) returns (PagamentoResponse);
}
```

```
import grpc
from concurrent import futures
import time
import random
import sqlite3

# Importar stubs gerados
import pagamento_pb2
import pagamento_pb2_grpc

DATABASE = 'pagamentos.db' # Banco específico do serviço

def get_db():
    conn = sqlite3.connect(DATABASE)
    conn.execute('''
        CREATE TABLE IF NOT EXISTS pagamentos (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            pedido_id INTEGER NOT NULL UNIQUE,
            status TEXT NOT NULL,
            data_processamento TEXT
        )
    ''')
    conn.commit()
    return conn
```

```
class PagamentoService(pagamento_pb2_grpc.PagamentoServicer):
    def ProcessarPagamento(self, request, context):
        print(f"Recebido pedido de pagamento para Pedido ID: {request.pedido_id},
             Valor: {request.valor}")

        # Simula processamento
        time.sleep(random.uniform(0.5, 1.5))
        sucesso = random.random() > 0.1 # 90% de chance de sucesso
        status = "aprovado" if sucesso else "rejeitado"
        data = time.strftime('%Y-%m-%d %H:%M:%S')

        try:
            conn = get_db()
            cursor = conn.cursor()
            cursor.execute("INSERT INTO pagamentos (pedido_id, status, data_processamento)
                VALUES (?, ?, ?)",
                            (request.pedido_id, status, data))
            conn.commit()
            conn.close()
            print(f"Pagamento para Pedido ID: {request.pedido_id} registrado como {status}")

            # Chama o próximo serviço (Estoque) - NÃO MOSTRADO AQUI
            # if sucesso:
            #   chamar_servico_estoque(request.pedido_id)

            return pagamento_pb2.PagamentoResponse(pedido_id=request.pedido_id, sucesso=sucesso,
                mensagem=f"Pagamento {status}")
        except sqlite3.Error as e:
             print(f"Erro SQLite: {e}")
             context.set_code(grpc.StatusCode.INTERNAL)
             context.set_details(f"Erro ao salvar pagamento: {e}")
             return pagamento_pb2.PagamentoResponse(pedido_id=request.pedido_id, sucesso=False,
                 mensagem="Erro interno no servidor de pagamento")


def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    pagamento_pb2_grpc.add_PagamentoServicer_to_server(PagamentoService(), server)
    server.add_insecure_port('[::]:50051') # Porta para o serviço de pagamento
    print("Servidor de Pagamento gRPC iniciado na porta 50051...")
    server.start()
    server.wait_for_termination()

if __name__ == '__main__':
    # Garante que a tabela exista ao iniciar
    conn = get_db()
    conn.close()
    serve()
```

6.3 3. Implementação da Parte 2 (Kafka)

- **Estrutura:** Semelhante à Parte 1, com diretórios por serviço.
- **Tópicos:** Crie os tópicos necessários no seu cluster Kafka (pode ser feito via linha de comando ou programaticamente, mas para o projeto, criar manualmente pode ser mais simples).

- **Produtores:**
    1. Instancie um `Producer` da `confluent_kafka`.
    2. Quando um evento ocorrer (ex: pagamento aprovado), serialize os dados relevantes (JSON é uma boa escolha) e use `producer.produce(topic, key=key, value=message, callback=delivery_report)` para enviar. A chave (`key`) é importante para garantir que mensagens relacionadas a um mesmo pedido vão para a mesma partição, mantendo a ordem *dentro* daquela partição.
    3. Use `producer.flush()` periodicamente ou ao final para garantir o envio.
- **Consumidores:**
    1. Instancie um `Consumer` da `confluent_kafka`, especificando `bootstrap.servers` e um `group.id`. O `group.id` permite que múltiplas instâncias do mesmo serviço trabalhem em conjunto, consumindo partições diferentes do tópico.
    2. Use `consumer.subscribe([topic1, topic2])` para ouvir os tópicos de interesse.
    3. Em um loop, use `consumer.poll(timeout)` para buscar novas mensagens.
    4. Se uma mensagem for recebida, deserialize-a (ex: `json.loads(msg.value().decode('utf-8'))`).
    5. Processe a mensagem (sua lógica de negócio, interação com SQLite).
    6. **Importante:** O Kafka gerencia os offsets (quais mensagens já foram lidas por um `group.id`). Por padrão (com `enable.auto.commit=true`), o commit é feito automaticamente em intervalos. Para maior controle, pode-se desabilitar o auto commit e fazer commits manuais (`consumer.commit()`) após o processamento bem-sucedido da mensagem.

6.3.1 Exemplo Mínimo (Kafka - Consumidor de Pagamentos e Produtor para Estoque)

```
from confluent_kafka import Consumer, Producer, KafkaError
import json
import time
import sqlite3
import os

KAFKA_BROKERS = os.getenv('KAFKA_BROKERS', 'localhost:9092')
PAGAMENTOS_APROVADOS_TOPIC = 'pagamentos_aprovados' # Tópico que este serviço consome
ESTOQUE_SEPARADO_TOPIC = 'estoque_separado' # Tópico que este serviço produz
CONSUMER_GROUP_ID = 'estoque-group'
DATABASE = 'estoque.db' # Banco específico do serviço

def get_db():
    conn = sqlite3.connect(DATABASE)
    # Tabela para rastrear quais pedidos já tiveram estoque separado
    conn.execute('''
        CREATE TABLE IF NOT EXISTS separacao_estoque (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            pedido_id INTEGER NOT NULL UNIQUE,
            status TEXT NOT NULL, -- 'sucesso', 'falha_sem_estoque'
            data_separacao TEXT
        )
```

```
    ''')
    # Tabela simplificada de produtos (poderia vir do DB geral ou ser sincronizada)
    conn.execute('''
        CREATE TABLE IF NOT EXISTS produtos_local (
            produto_id INTEGER PRIMARY KEY,
            quantidade INTEGER NOT NULL
        )
    ''')
    conn.commit()
    return conn

def delivery_report(err, msg):
    """ Callback para reportar o resultado da produção da mensagem. """
    if err is not None:
        print(f"Falha ao entregar mensagem para {msg.topic()}: {err}")
    else:
        print(f"Mensagem entregue em {msg.topic()} [{msg.partition()}] @ offset {msg.offset()}")

def simular_separacao_estoque(pedido_id):
    """Simula a lógica de separação e retorna True se sucesso."""
    print(f"[Estoque] Iniciando separação para Pedido ID: {pedido_id}")
    # Lógica simplificada: assume que sempre há estoque.
    # Em um cenário real: buscar itens do pedido, verificar estoque em `produtos_local`,
    # decrementar quantidade, tratar falta de estoque.
    time.sleep(random.uniform(0.8, 2.0))
    sucesso = random.random() > 0.05 # 95% de chance de ter estoque
    status_db = 'sucesso' if sucesso else 'falha_sem_estoque'
    data = time.strftime('%Y-%m-%d %H:%M:%S')

    conn = get_db()
    try:
        cursor = conn.cursor()
        cursor.execute("INSERT INTO separacao_estoque (pedido_id, status, data_separacao) VALUES (?, ?, ?)",
                       (pedido_id, status_db, data))
        conn.commit()
        print(f"[Estoque] Resultado da separação para Pedido ID {pedido_id}: {status_db}")
        return sucesso
    except sqlite3.IntegrityError:
         print(f"[Estoque] Pedido ID {pedido_id} já processado anteriormente.")
         return False # Evita reprocessamento e envio duplicado para Kafka
    except sqlite3.Error as e:
        print(f"[Estoque] Erro SQLite ao salvar separação: {e}")
        return False # Considera falha se não conseguir salvar
    finally:
        if conn:
            conn.close()


def main():
    # Garante que tabelas existam
    conn = get_db()
    # Popular estoque local inicial (exemplo)
    cursor = conn.cursor()
    cursor.execute("INSERT OR IGNORE INTO produtos_local (produto_id, quantidade)
        VALUES (101, 50), (102, 30)")
    conn.commit()
    conn.close()

    consumer_conf = {
        'bootstrap.servers': KAFKA_BROKERS,
```

```
    'group.id': CONSUMER_GROUP_ID,
    'auto.offset.reset': 'earliest', # Ler desde o início se for um novo grupo
    # 'enable.auto.commit': False # Descomentar para commit manual
}
consumer = Consumer(consumer_conf)

producer_conf = {'bootstrap.servers': KAFKA_BROKERS}
producer = Producer(producer_conf)

try:
    consumer.subscribe([PAGAMENTOS_APROVADOS_TOPIC])
    print(f"Consumidor de Estoque inscrito no tópico '{PAGAMENTOS_APROVADOS_TOPIC}'...")

    while True:
        msg = consumer.poll(timeout=1.0) # Espera por 1 segundo

        if msg is None:
            continue
        if msg.error():
            if msg.error().code() == KafkaError._PARTITION_EOF:
                # Fim da partição, não é um erro real
                continue
            else:
                print(f"Erro no consumidor: {msg.error()}")
                break

        try:
            # Assume que a mensagem do pagamento é JSON com 'pedido_id'
            pagamento_info = json.loads(msg.value().decode('utf-8'))
            pedido_id = pagamento_info.get('pedido_id')

            if pedido_id:
                print(f"\n[Estoque] Recebido evento de pagamento aprovado para
                    Pedido ID: {pedido_id}")

                # Processa o estoque
                separacao_ok = simular_separacao_estoque(pedido_id)

                if separacao_ok:
                    # Produz evento para o próximo serviço (Fiscal)
                    evento_estoque = {
                        'pedido_id': pedido_id,
                        'data_separacao': time.strftime('%Y-%m-%d %H:%M:%S'),
                        'status': 'concluido'
                    }
                    producer.produce(
                        ESTOQUE_SEPARADO_TOPIC,
                        key=str(pedido_id), # Usa pedido_id como chave para particionamento
                        value=json.dumps(evento_estoque).encode('utf-8'),
                        callback=delivery_report
                    )
                    producer.poll(0) # Tenta enviar mensagens pendentes sem bloquear
                else:
                     print(f"[Estoque] Separação falhou ou já processada para
                         Pedido ID: {pedido_id}. Nenhuma mensagem produzida.")
                     # Aqui poderia ter lógica para tratar falha (ex: notificar, retry)
            else:
                 print("[Estoque] Mensagem recebida sem 'pedido_id'. Ignorando.")

            # Se usando commit manual:
```

```
                # consumer.commit(asynchronous=False)

            except json.JSONDecodeError:
                 print(f"[Estoque] Erro ao decodificar mensagem JSON: {msg.value()}")
            except Exception as e:
                 print(f"[Estoque] Erro inesperado ao processar mensagem: {e}")


    except KeyboardInterrupt:
        print("Encerrando consumidor de estoque...")
    finally:
        consumer.close()
        producer.flush(timeout=5) # Garante envio das mensagens pendentes
        print("Consumidor e Produtor de Estoque finalizados.")

if __name__ == '__main__':
    main()
```

6.4 4. Testes e Comparação

- **Testes Funcionais:** Verifique se um pedido passa corretamente por todas as etapas em ambas as arquiteturas.
- **Testes de Carga:**
  - Crie um script (ou use ferramentas como `locust`, `k6` ou `ApacheBench`) para simular a criação de um grande número de pedidos em um curto período de tempo.
  - Monitore:
    - **Latência:** Tempo total para um pedido ser processado ponta a ponta.
    - **Taxa de Transferência (Throughput):** Quantos pedidos por segundo o sistema consegue processar.
    - **Uso de Recursos:** CPU, memória dos serviços e do broker Kafka.
    - **Tratamento de Erros/Falhas:** Injete falhas (ex: derrube um serviço temporariamente). Como cada arquitetura reage? A arquitetura Kafka deve mostrar maior resiliência, pois as mensagens podem ser processadas quando o serviço voltar.
- **Análise:** Compare os resultados. Espera-se que:
  - A arquitetura RPC síncrona tenha menor latência inicial para baixo volume, mas sofra com gargalos e acoplamento sob alta carga (um serviço lento impacta toda a cadeia).
  - A arquitetura Kafka tenha uma latência *por etapa* potencialmente maior (devido à indireção do broker), mas ofereça maior throughput geral, desacoplamento e resiliência, escalando melhor horizontalmente (adicionando mais instâncias de consumidores).

7 Relatório do Projeto

Prepare um relatório contendo:

1. **Introdução:** Objetivos do projeto.
2. **Arquiteturas:** Diagramas e descrição detalhada das duas implementações (RPC e Kafka).
3. **Implementação:** Detalhes sobre a implementação de cada serviço, tecnologias utilizadas, desafios encontrados.
4. **Resultados dos Testes:** Apresentação clara dos resultados dos testes de carga (gráficos de latência vs carga, throughput, etc.).
5. **Análise Comparativa:** Discussão aprofundada comparando as duas abordagens nos quesitos:
    - Complexidade de implementação e manutenção.
    - Acoplamento entre serviços.
    - Escalabilidade (horizontal/vertical).
    - Resiliência a falhas.
    - Performance (latência, throughput) sob diferentes cargas.
    - Observabilidade e Monitoramento.
6. **Conclusão:** Síntese dos aprendizados e quando cada abordagem seria mais adequada.
7. **Código Fonte:** Todo o código fonte no repositório GitHub Classroom.

O Relatório deverá ser entregue no repositório GitHub Classroom do projeto.

8 Avaliação

A avaliação levará em conta:

- **Funcionalidade:** Correção e completude das implementações.
- **Qualidade do Código:** Organização, clareza, boas práticas, tratamento básico de erros.
- **Relatório:** Clareza da escrita, profundidade da análise comparativa, qualidade dos diagramas e resultados apresentados.
- **Compreensão dos Conceitos:** Demonstração de entendimento dos princípios de RPC, Publish-Subscribe, Kafka e suas implicações em sistemas distribuídos.