Serviço Público Federal
Ministério da Educação
Fundação Universidade Federal de Mato Grosso do Sul

# RELATÓRIO TÉCNICO: ARQUITETURA DE SISTEMAS DISTRIBUÍDOS

## ACME/SA

**Atividade Avaliativa**

Gustavo Cortez de Paula

TRÊS LAGOAS (MS)
2025

**SUMÁRIO**

## 1. Introdução e Escopo

Este documento descreve a implementação técnica da plataforma ACME/SA, uma solução de *backend* distribuído projetada para simular o ecossistema de múltiplas filiais de uma organização. O objetivo principal do sistema é demonstrar conceitos práticos de sistemas distribuídos, especificamente focados em concorrência, consistência eventual e tolerância a falhas em um ambiente de microsserviços.

A solução foi desenvolvida utilizando Python com o framework FastAPI, operando sobre bancos de dados SQLite independentes para cada instância (nó), garantindo que não haja compartilhamento de armazenamento físico entre as réplicas, o que força a necessidade de sincronização via rede.

## 2. Arquitetura de Software

A arquitetura segue o padrão de microsserviços descentralizados. Cada instância da aplicação (definida em config.py através da variável NODE_NAME) atua como uma filial autônoma.

### 2.1 Componentes Principais

A estrutura do código foi modularizada para separar responsabilidades críticas:

- **Gerenciamento de Estado (state.py):** Encapsula as regras de negócio, transações atômicas e o controle de *locks* assíncronos. É aqui que garantimos a integridade local dos dados.
- **Persistência (database.py):** Utiliza SQLite em modo WAL (*Write-Ahead Logging*) com chaves estrangeiras ativadas. A escolha do modo WAL foi estratégica para melhorar a performance de concorrência de leitura e escrita em disco local.
- **Sincronização (sync.py):** Gerencia as filas de mensagens e a lógica de replicação para os nós pares (PEERS).
- **Segurança (security.py):** Centraliza a lógica de *hashing* de senhas (bcrypt) e a emissão/validação de tokens JWT.

## 3. Estratégia de Sincronização e Consistência

Um dos maiores desafios deste projeto foi equilibrar a consistência dos dados entre nós que operam independentemente. Adotamos uma abordagem híbrida de **Consistência Forte Local** e **Consistência Eventual Global**.

### 3.1 Consistência Local (ACID)

Para operações críticas, como a criação de pedidos e movimentação de estoque, o sistema impõe consistência forte. Quando uma requisição chega:

1. Um *lock* específico para o produto é adquirido no state.py.
2. Uma transação de banco de dados é aberta.
3. O sistema valida o saldo e insere o pedido.
4. Se houver sucesso, o saldo é decrementado e a transação sofre *commit*.

Isso impede que dois pedidos simultâneos no **mesmo nó** vendam o mesmo item se o estoque for insuficiente (concorrência tratada via HTTP 409 Conflict).

### 3.2 Replicação Baseada em Eventos (Eventual)

A comunicação entre nós não ocorre via banco compartilhado, mas via eventos HTTP assíncronos.

- **Idempotência via Snapshots:** Ao invés de enviar comandos relativos (ex: "diminua 1 do estoque"), optamos por enviar o estado final do objeto (*snapshot*). Eventos como order_created e stock_update carregam o objeto completo e sua versão. Isso permite que, mesmo se um nó receber o mesmo evento duas vezes, o resultado final seja consistente e não duplicado.
- **Fluxo de Atualização:** Quando um pedido é criado no Nó A, o estoque é atualizado localmente de imediato. O evento é então enfileirado para o Nó B. O Nó B, ao receber o stock_update, aplica a mudança em seu banco, garantindo que todas as filiais convirjam para o mesmo saldo ao longo do tempo.

## 4. Tolerância a Falhas e Resiliência

Considerando que falhas de rede são inevitáveis em sistemas distribuídos, implementamos mecanismos robustos no módulo sync.py para lidar com indisponibilidade dos pares.

### 4.1 Filas de Retentativa (Retry Pattern)

A classe ReplicaSynchronizer mantém uma fila FIFO (*First-In, First-Out*) em memória para cada nó par configurado.

- Se um nó destino estiver offline (simulado pelo desligamento da instância), os eventos não são perdidos; eles permanecem em estado pending.

- Um *loop* de verificação periódico (configurado via REPLICATION_RETRY_SECONDS) tenta reenviar os eventos pendentes até receber uma confirmação de sucesso (HTTP 2xx).

### 4.2 Recuperação Automática

Graças à idempotência dos eventos descrita na seção anterior, o sistema suporta o reinício dos nós sem corrupção de dados. Ao restabelecer a conexão, a fila é drenada automaticamente, atualizando o nó que estava offline com as últimas versões dos produtos e pedidos. O endpoint /status foi implementado para permitir auditoria operacional, exibindo o tamanho do *backlog* de replicação em tempo real.

## 5. Protocolos de Segurança

A segurança foi desenhada em camadas para proteger tanto o acesso dos usuários quanto a integridade da replicação entre servidores.

### 5.1 Autenticação de Usuários (Camada Externa)

Clientes e operadores interagem com a API através de tokens JWT (*JSON Web Tokens*), assinados com um JWT_SECRET. As senhas dos usuários são armazenadas apenas como *hashes* (bcrypt). O sistema implementa controle de acesso (RBAC), onde apenas perfis autorizados (Admin/Operador) podem realizar cadastros sensíveis.

### 5.2 Segurança Entre Réplicas (Camada Interna)

Para evitar que um usuário mal-intencionado (com um JWT válido) forje eventos de sincronização para alterar estoques em outras filiais, implementamos um mecanismo de **Mutual Trust** simplificado. O tráfego entre os nós nos endpoints /replica/event é protegido por um cabeçalho exclusivo: X-Replica-Token. Este token é distinto do JWT dos usuários e é conhecido apenas pelos servidores (definido nas variáveis de ambiente). Requisições de replicação sem esse token específico são rejeitadas imediatamente, isolando o tráfego de infraestrutura do tráfego de clientes.

## 6. Conclusão

O projeto ACME/SA demonstra uma implementação funcional de um sistema distribuído resiliente. Através do uso de filas de retentativa, bancos de dados isolados e eventos portando *snapshots* de estado, conseguimos atingir os objetivos de consistência

eventual e alta disponibilidade, mantendo a integridade dos dados mesmo em cenários de falha de comunicação entre os nós.