Sistemas Distribuídos

Relatório do Projeto: Servidor de Séries Temporais

Hugo Soares (A107293)
Francisco Martins (A106902)
Marco Sèvegrand (A106807)

27 de novembro de 2025

Conteúdo

1 Introdução

Este relatório descreve a implementação de um sistema cliente-servidor para gestão e análise de séries temporais de vendas, desenvolvido no âmbito da unidade curricular de Sistemas Distribuídos.

O objetivo principal do projeto foi desenvolver um servidor capaz de receber eventos de vendas de múltiplos clientes simultaneamente, armazená-los de forma eficiente, permitir consultas estatísticas e garantir a persistência dos dados, assegurando a consistência através de mecanismos de sincronização adequados.

2 Arquitetura do Sistema

O sistema segue uma arquitetura clássica Cliente-Servidor utilizando sockets TCP para comunicação.

2.1 Servidor

O servidor (`TimeSeriesServer`) é o componente central. Ao iniciar, ele cria um `ServerSocket` e utiliza um `ExecutorService` (Thread Pool) para aceitar conexões. Para cada cliente conectado, uma nova tarefa `ClientHandler` é submetida à pool, permitindo o processamento concorrente de múltiplos clientes.

2.2 Cliente

O cliente (`TimeSeriesClient`) fornece uma API para comunicar com o servidor, abstraindo a complexidade do protocolo de rede. Existe também uma interface de texto (`ClientUI`) que permite aos utilizadores interagir com o sistema.

3 Estruturas de Dados e Domínio

3.1 Séries Temporais (TimeSeries)

A classe `TimeSeries` representa os dados de um dia específico. Ela armazena uma lista de objetos `Event`. Para otimizar o desempenho das consultas, a classe implementa um sistema de *Lazy Caching*. Os resultados de agregações (soma de quantidades, volume de vendas, estatísticas de preço) são armazenados em mapas (`quantityCache`, `volumeCache`, etc.). Estes caches são invalidados sempre que um novo evento é adicionado, garantindo a consistência dos dados.

3.2 Eventos (Event)

Cada evento de venda é imutável e contém:

- Nome do produto (String)
- Quantidade (long)

- Preço unitário (double)
- Timestamp (long)

4 Concorrência e Sincronização

A gestão da concorrência é um dos aspetos mais críticos deste projeto. Foram utilizados diferentes mecanismos para garantir a integridade dos dados e a eficiência.

4.1 ReentrantReadWriteLock

Na classe `TimeSeries`, utilizou-se `ReentrantReadWriteLock`. Esta escolha justifica-se pelo padrão de acesso aos dados:

- **Leitura (Read Lock):** Múltiplas threads podem calcular estatísticas simultaneamente (ex: `calculateQuantity`), desde que não haja escritas a ocorrer.
- **Escrita (Write Lock):** Apenas uma thread pode adicionar um evento (`addEvent`) de cada vez, bloqueando leituras e outras escritas.

O servidor principal também usa este lock para proteger o mapa de séries temporais (`timeSeriesMap`) e o mapa de utilizadores.

4.2 NotificationManager e Condition Variables

Para implementar as funcionalidades de espera por eventos (vendas simultâneas ou consecutivas), foi criada a classe `NotificationManager`. Utiliza-se um `ReentrantLock` e uma `Condition` (`eventOccurred`).

```
public void recordSale(String productName) {
    lock.lock();
    try {
        productsWithSales.add(productName);
        // ... logica de consecutivos ...
        eventOccurred.signalAll(); // Acorda threads    espera
    } finally {
        lock.unlock();
    }
}
```

Listing 1: Exemplo de uso de Condition no NotificationManager

As threads que aguardam por vendas (comando `WAIT_SIMULTANEOUS`) ficam bloqueadas no método `await()` da variável de condição, sendo acordadas apenas quando ocorre uma nova venda, evitando *busy-waiting*.

5 Persistência e Gestão de Memória

O sistema implementa uma lógica de "Janela Deslizante"para gestão de memória.

5.1 Rotação de Dias

O servidor mantém em memória apenas um número fixo de dias (`maxSeriesInMemory`). Quando o comando `NEXT_DAY` é invocado:

1. O dia corrente é incrementado.
2. Uma nova `TimeSeries` é criada.
3. Se o limite de memória for excedido, a série mais antiga é removida da memória e persistida em disco.

5.2 Serialização

Os dados são guardados na pasta `data/`.

- **Utilizadores:** Ficheiro de texto `users.dat` (formato `user:hash`).
- **Séries:** Ficheiros binários `series_<dia>.dat` usando `ObjectOutputStream`.

6 Protocolo de Comunicação

Foi desenvolvido um protocolo binário personalizado (`BinaryProtocol`) para minimizar o tráfego de rede.

6.1 Estrutura das Mensagens

Todas as mensagens iniciam com um inteiro representando o ID do comando (definido em `ProtocolCommands`), seguido pelos argumentos específicos.

6.2 Comandos Principais

- **REGISTER (1) / LOGIN (2):** Autenticação de utilizadores.
- **ADD_EVENT (10):** Envio de dados de venda.
- **NEXT_DAY (11):** Avanço do tempo no servidor.
- **GET_QUANTITY (20) / GET_VOLUME (21):** Consultas agregadas.
- **WAIT_SIMULTANEOUS (30):** Bloqueia até dois produtos serem vendidos no mesmo dia.

7 Conclusão

O projeto permitiu aplicar na prática os conceitos de programação concorrente e distribuída. A utilização de `ReadWriteLocks` permitiu otimizar o acesso aos dados, favorecendo leituras paralelas, enquanto as variáveis de condição permitiram implementar notificações eficientes entre threads. A arquitetura modular facilita a manutenção e a extensão futura do sistema.