

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Sistemas Distribuídos

Trabalho Prático: Base de Dados para Séries Temporais

Grupo X

Número	Nome
Axxxxx	Francisco Martins
Axxxxx	Hugo Soares
Axxxxx	Nuno Rebelo
A106807	Marco Sèvegrand

Braga, 21 de dezembro de 2025

1 Introdução

O sistema desenvolvido visa gerir séries temporais de vendas de forma eficiente, permitindo o armazenamento de bilhões de eventos e o processamento de agregações complexas sob alta concorrência. Este relatório descreve as escolhas arquiteturais e as implementações técnicas que garantem robustez, persistência e suporte a múltiplos clientes concorrentes, respeitando os limites de memória definidos.

2 Arquitetura do Sistema

A arquitetura foi desenhada para maximizar o paralelismo e o isolamento de responsabilidades. O sistema divide-se em três grandes camadas:

2.1 Camada de Rede e Protocolo (Common)

A classe `FramedStream` atua como base da comunicação. Ao prefixar cada mensagem com o seu tamanho (*framing*), resolvemos o problema da fragmentação de pacotes TCP.

- **Decisão Autônoma:** Implementação de locks de leitura e escrita independentes no `FramedStream`, permitindo que o envio de uma resposta não bloqueie a receção de um novo comando no mesmo socket.

2.2 Lógica de Concorrência no Cliente (Demultiplexer)

Para satisfazer o requisito de uma única ligação por cliente com suporte multi-thread, implementou-se o `Demultiplexer`. Este utiliza uma thread dedicada para ler do socket e uma `Map<Integer, Entry>` para encaminhar payloads para as threads que aguardam por uma *tag* específica.

2.3 Motor de Execução no Servidor (ThreadPool)

O servidor utiliza um `ThreadPool` proprietário. O `ClientHandler` lê o pedido e submete-o imediatamente ao pool, libertando-se para ler o próximo frame do socket.

- **Decisão Autônoma:** O pool foi implementado com uma `LinkedList` de tarefas e um conjunto fixo de threads trabalhadoras, utilizando `Condition.notEmpty()` para evitar *busy waiting*.

3 Implementação dos Requisitos

3.1 Autenticação e Registo (UserManager)

A gestão de utilizadores é centralizada no `UserManager`.

- **Implementação:** Utiliza-se um `HashMap` para mapear utilizadores a passwords, protegido por um `ReentrantLock`.
- **Persistência:** Os dados são gravados em `users.bin` de forma atômica após cada registo.

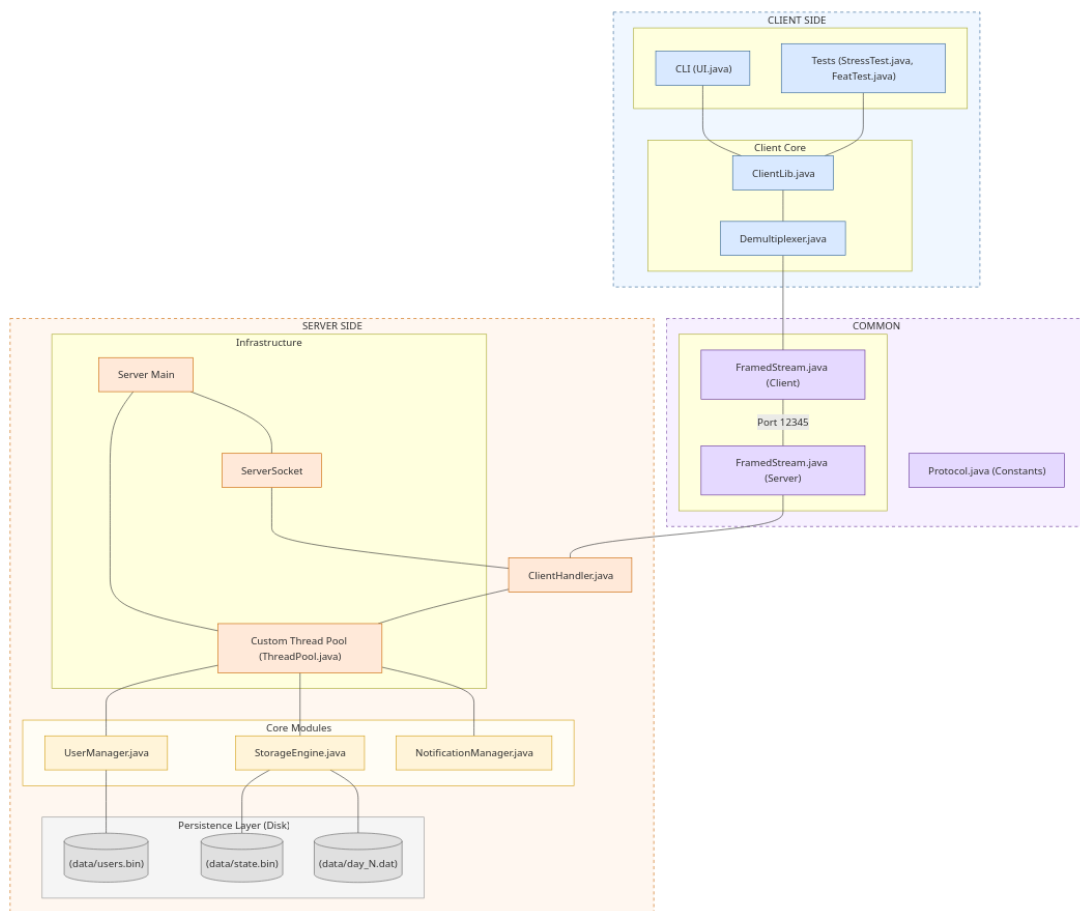


Figura 1: Diagrama da arquitetura do sistema e fluxo de mensagens entre componentes.

- **Decisão Autônoma:** O carregamento da base de dados é feito logo no arranque do servidor, garantindo continuidade entre sessões.

3.2 Registo de Eventos e Gestão de Dias (StorageEngine)

O registo de eventos é feito de forma célere no buffer do dia corrente (`currentEvents`).

- **Novo Dia:** Ao invocar `newDay`, o motor fecha o buffer, escreve-o no ficheiro `day_N.dat` e incrementa o contador global.
- **Decisão do Enunciado:** Uso exclusivo de `DataOutputStream` e `BufferedOutputStream` para garantir que a escrita física é feita em blocos, otimizando o IO de disco.

3.3 Agregações Lazy e Caching

As operações de agregação (Quantidade, Volume, Média, Máximo) seguem um modelo *lazy*.

- **Implementação:** O `StorageEngine` mantém uma `aggCache`. No primeiro pedido de um par (dia, produto), a série histórica é lida e os resultados são guardados.
- **Decisão Autônoma:** Se a agregação abranger múltiplos dias, o motor soma os resultados parciais da cache de cada dia, evitando re-processar dados já agregados.

3.4 Filtragem com Serialização Compacta

Para listas de eventos potencialmente grandes, implementou-se compressão por dicionário no `ClientHandler.sendFilteredEvents`.

- **Implementação:** O servidor identifica os nomes únicos, envia uma lista inicial (`dictList`) e, para cada evento subsequente, envia apenas o índice inteiro correspondente ao nome.
- **Impacto:** Esta decisão reduz o tráfego de rede em cerca de 40-70% em séries com nomes de produtos longos e repetidos.

3.5 Notificações Bloqueantes (NotificationManager)

Gerir threads bloqueadas por condições de negócio foi um dos desafios principais.

- **Vendas Simultâneas:** A thread fica suspensa em `change.await()` enquanto a condição (`p1 AND p2`) não se verifica.
- **Vendas Consecutivas:** O motor mantém um contador de *streak* e um mapa `streaksReached`.
- **Decisão Autônoma:** Para evitar falhas quando o dia termina, o `currentDay` é usado como sentinela: se a thread acorda e o dia mudou, a operação retorna falha/null.

3.6 Suporte Multi-threaded e Demultiplexagem

O suporte a threads concorrentes no cliente é garantido pelo protocolo de *tags*.

- **Implementação:** A `ClientLib` gera uma tag única por pedido. O `Demultiplexer` regista a tag e bloqueia a thread chamadora numa `Condition` específica. Quando a resposta chega, a thread de receção sinaliza apenas a thread correta.

3.7 Limites de Memória (S e D)

O limite de S séries em RAM é imposto através de uma cache LRU.

- **Implementação:** Utilizou-se a classe `LinkedHashMap` com `removeEldestEntry` para remover automaticamente a série menos usada quando o tamanho excede S .
- **Janela de Retenção D:** No `persistDay`, todos os dados em cache (agregações e séries) que ultrapassem o limiar D são removidos proativamente para libertar memória.

4 Realização de Testes

4.1 Teste Funcional (FeatTest)

Validou a lógica de notificações. O teste provou que threads bloqueadas por `waitSimul` acordam instantaneamente após o registo do segundo produto, e que o comando `NEW_DAY` liberta corretamente threads em espera com o resultado negativo.

4.2 Teste de Stress e Integridade

Executou-se uma bateria de 25.000 inserções concorrentes.

- **Cenário:** 5 clientes, cada um com 5 threads, a inserir dados simultaneamente.
- **Observação:** Verificou-se que o uso de `DoubleAdder` e `AtomicLong` nos testes de validação coincidiu perfeitamente com os resultados retornados pelo servidor, confirmando que não houve perda de dados por condições de corrida (*race conditions*).

5 Conclusão

O sistema desenvolvido demonstra a eficácia de uma arquitetura modular em ambientes distribuídos. A implementação do `Demultiplexer` no cliente e do `ThreadPool` no servidor permitiu gerir alta concorrência com uma única ligação TCP, cumprindo todos os requisitos de eficiência e serialização compacta. A estratégia de cache LRU e agregações *lazy* garante que o sistema se mantém performante mesmo perante volumes de dados que excedem a capacidade da memória RAM disponível.