Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Sistemas Distribuídos

Projeto Prático: Sistema de Gestão de Séries Temporais

Grupo 9

Número	**Nome**
A106902	Francisco Martins
A107293	Hugo Soares
A107372	Nuno Rebelo
A106807	Marco Sèvegrand

Braga, 21 de dezembro de 2025

1 Introdução

Este trabalho descreve o desenvolvimento de um motor de base de dados especializado no armazenamento e processamento de séries temporais de vendas. O sistema foi desenhado para suportar o registo massivo de eventos e a consulta de agregações estatísticas, garantindo a integridade dos dados num ambiente multi-utilizador concorrente. O foco principal incidiu na conformidade com os requisitos de persistência em disco, gestão de memória RAM e na implementação de um protocolo binário eficiente sobre TCP.

2 Arquitetura do Sistema

A arquitetura do sistema foi planeada para maximizar a reusabilidade de código e o isolamento de componentes. No servidor, o processamento de pedidos é delegado a uma *thread pool* global, permitindo que a receção de dados no *socket* nunca seja interrompida pela execução de operações lógicas pesadas. No cliente, a biblioteca abstrai a complexidade da rede, oferecendo uma interface síncrona para o utilizador enquanto gere a assincronia das respostas em segundo plano.

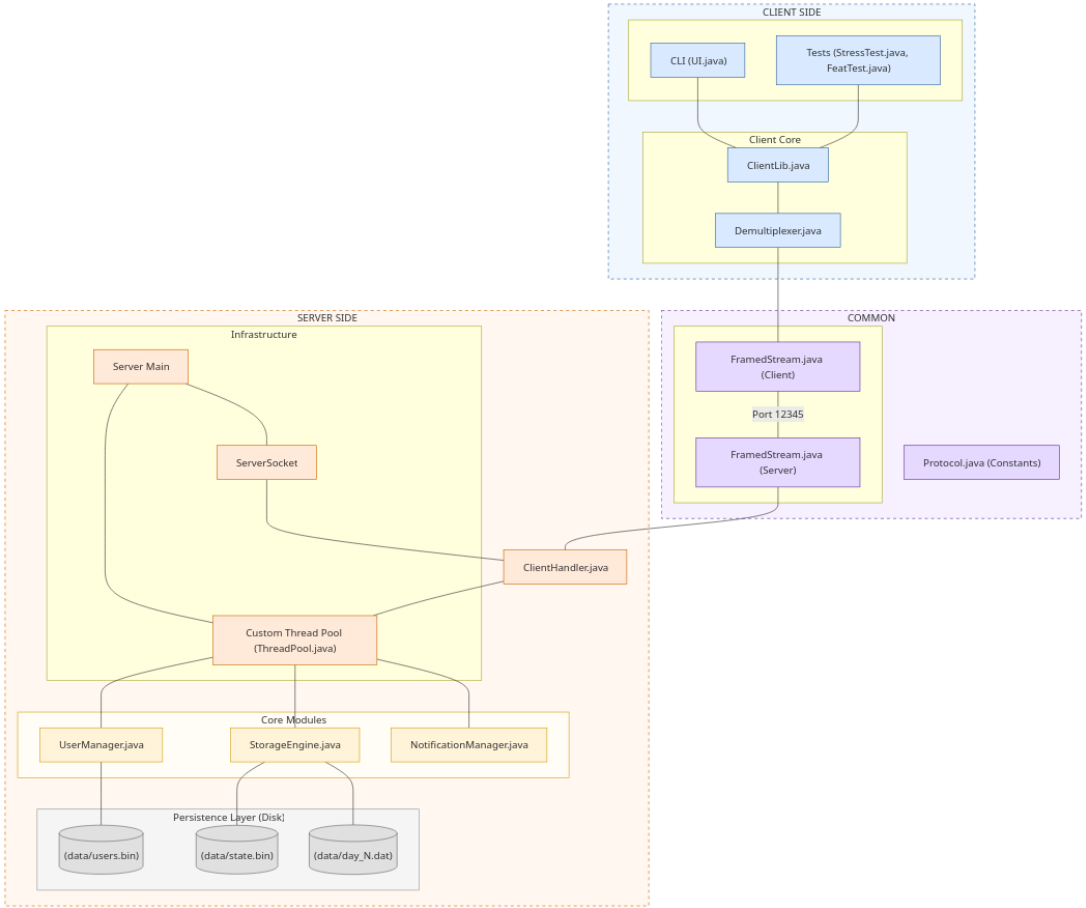

Figura 1: Diagrama da arquitetura do sistema e fluxo de mensagens entre componentes.

3 Implementação dos Requisitos

3.1 Autenticação e Gestão de Sessão

Em conformidade com o enunciado, o sistema exige que qualquer interação (exceto registo e login) seja precedida de autenticação. O grupo optou por centralizar esta lógica no `UserManager`, que utiliza um `HashMap` protegido por `ReentrantLock`. Como decisão de implementação, os utilizadores são persistidos num ficheiro binário após cada registo, garantindo que as contas não se percam entre reinicializações do servidor.

3.2 Registo de Eventos e Simulação Temporal

O registo de vendas no dia corrente é efetuado de forma volátil num buffer, conforme previsto. A operação `NEW_DAY`, exigida para simular a passagem do tempo, despoleta o fecho do dia atual e a sua escrita física no disco. O grupo decidiu implementar esta operação como um comando administrativo que sinaliza simultaneamente o motor de armazenamento e o gestor de notificações, assegurando uma transição de estado atómica.

3.3 Agregações Estatísticas e Modelo Lazy

As operações de agregação (quantidade, volume, média e máximo) são processadas de modo *lazy*, uma imposição do enunciado para otimizar o uso de recursos. O grupo implementou uma cache de segundo nível (`aggCache`) que armazena resultados parciais por dia e produto. Em vez de re-processar ficheiros históricos, o sistema combina estes resultados em cache, reduzindo drasticamente o I/O de disco em pedidos repetidos.

3.4 Filtragem e Serialização Compacta

Para a listagem de eventos históricos, o enunciado exige uma serialização eficiente. O grupo optou por um algoritmo de **compressão por dicionário**, onde os nomes repetidos de produtos são substituídos por identificadores inteiros no fluxo de dados. Esta técnica permite que séries temporais com milhares de eventos sejam transmitidas com um consumo de largura de banda significativamente inferior ao de uma serialização convencional.

3.5 Notificações Bloqueantes

As notificações de vendas simultâneas e consecutivas requerem que a *thread* do pedido fique suspensa. O grupo utilizou as primitivas de sincronização `Condition.await()` para este fim. Uma decisão crítica de projeto foi o uso do contador de dias como sentinela: caso o dia termine (`NEW_DAY`) enquanto uma *thread* aguarda, a condição é invalidada e o cliente recebe uma resposta negativa, cumprindo o comportamento de tempo real exigido.

3.6 Ligação Única e Suporte Multi-threaded

O requisito de manter uma única ligação TCP por cliente, suportando pedidos concorrentes, foi satisfeito através de um protocolo de *tags*. O grupo implementou um `Demultiplexer` no cliente que mapeia as respostas vindas do servidor de volta à *thread* correta. Esta abordagem resolve o problema do bloqueio em cabeça de linha (*head-of-line blocking*), permitindo que pedidos rápidos não fiquem retidos atrás de agregações pesadas.

3.7 Persistência e Limites de Memória

Respeitando os parâmetros S (séries em RAM) e D (janela de retenção), o motor de armazenamento utiliza uma política de substituição **LRU (Least Recently Used)**. O grupo implementou esta lógica através de uma `LinkedHashMap`, garantindo que apenas as S séries mais consultadas ocupem memória RAM. Séries excedentes são processadas diretamente do disco em fluxo, conforme exigido pelo enunciado para garantir a escalabilidade do sistema.

4 Realização de Testes

A validação funcional confirmou que o sistema gere corretamente as notificações e a transição de dias. No que diz respeito à performance, foi executado um teste de stress com 25.000 inserções simultâneas. A concordância absoluta entre os contadores locais dos testes e os valores devolvidos pelo servidor demonstra a fiabilidade do sistema sob carga elevada e a ausência de condições de corrida.

5 Utilização de Ferramentas de IA

Conforme estipulado no enunciado do projeto, o uso de ferramentas de Inteligência Artificial foi limitado a tarefas que não constituem o núcleo das competências de Sistemas Distribuídos avaliadas. O grupo utilizou modelos de linguagem para a geração de código auxiliar, sendo cada componente validado manualmente.

5.1 Interface de Utilizador (CLI)

A interface `UI.java` foi gerada para proporcionar um ambiente interativo. O *prompt* focou-se na interpretação de comandos e na ligação à biblioteca cliente:

> *"Gera uma interface CLI interativa em Java que utilize a classe ClientLib para comunicar com um servidor de séries temporais. A interface deve suportar e explicar os comandos: ajuda, registar, entrar, sair, limpar, evento, dia, novodia, qtd, vol, media, max, filtrar, simul e consec. Deve tratar erros de input e manter o estado de login do utilizador."*

5.2 Teste de Funcionalidades

Para garantir que cada método da `ClientLib` operava corretamente, utilizou-se o seguinte *prompt* para o `FeatTest.java`:

> *"Com base na classe ClientLib, cria um script de teste funcional em Java que valide individualmente cada funcionalidade (registo, login, adição de eventos, agregações e notificações bloqueantes). O teste deve utilizar um cliente multi-threaded para verificar se a receção de notificações em background não interfere com os pedidos síncronos."*

5.3 Teste de Carga e Stress

O `StressTest.java` foi gerado para validar a robustez do servidor sob carga elevada:

> *"Cria um teste de carga em Java para validar a consistência de dados num fluxo significativo de registos. Simula 5 clientes, cada um com 5 threads concorrentes, realizando 1000 inserções cada. Calcula os valores esperados localmente e, no fim, compara-os com os resultados de agregação devolvidos pelo servidor."*

5.4 Consulta de Documentação

Foi também utilizada IA para consultas rápidas à documentação das APIs de Java (*e.g.*, `ReentrantLock` e `LinkedHashMap`), acelerando a resolução de dúvidas sintáticas e de utilização de primitivas de sincronização.

6 Conclusão

O sistema desenvolvido cumpre integralmente os requisitos de eficiência e escalabilidade. A arquitetura modular, aliada a técnicas como o *demultiplexing* no cliente e o processamento assíncrono no servidor, permite uma utilização otimizada dos recursos. As estratégias de cache e a serialização compacta garantem que a base de dados mantenha tempos de resposta adequados mesmo perante volumes de dados massivos.