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, 9 de janeiro de 2026

1 Introdução

Este projeto implementa um motor de base de dados para o armazenamento e processamento de séries temporais de vendas. O sistema permite que múltiplos clientes registem eventos em tempo real, consultem agregações estatísticas (quantidade, volume, média, máximo) e subscrevam notificações de padrões específicos. O desenho da solução focou-se na persistência de grandes quantidades de dados, na garantia de integridade em ambientes concorrentes e na redução da latência através de uma hierarquia de cache em memória.

2 Arquitetura do Sistema

O sistema adota uma estrutura em camadas para isolar as responsabilidades de rede, lógica de negócio e armazenamento. O servidor utiliza um *pool* de threads personalizado (`ThreadPool`) para processar pedidos de forma concorrente, evitando a criação excessiva de threads.

No lado do cliente, a arquitetura foi desenhada para satisfazer o requisito de uma ligação TCP única por cliente. O `Demultiplexer` centraliza a leitura do socket numa thread dedicada, distribuindo as respostas pelas threads de aplicação com base em *tags*. Esta abordagem permite que múltiplas threads do cliente submetam pedidos concorrentemente sem bloqueio de cabeça de linha.

A comunicação é feita num formato binário, utilizando `DataInputStream` e `DataOutputStream` sobre um sistema de *framing* (`FramedStream`) que garante que as mensagens são transmitidas e lidas na íntegra, prefixando cada frame com o seu tamanho.

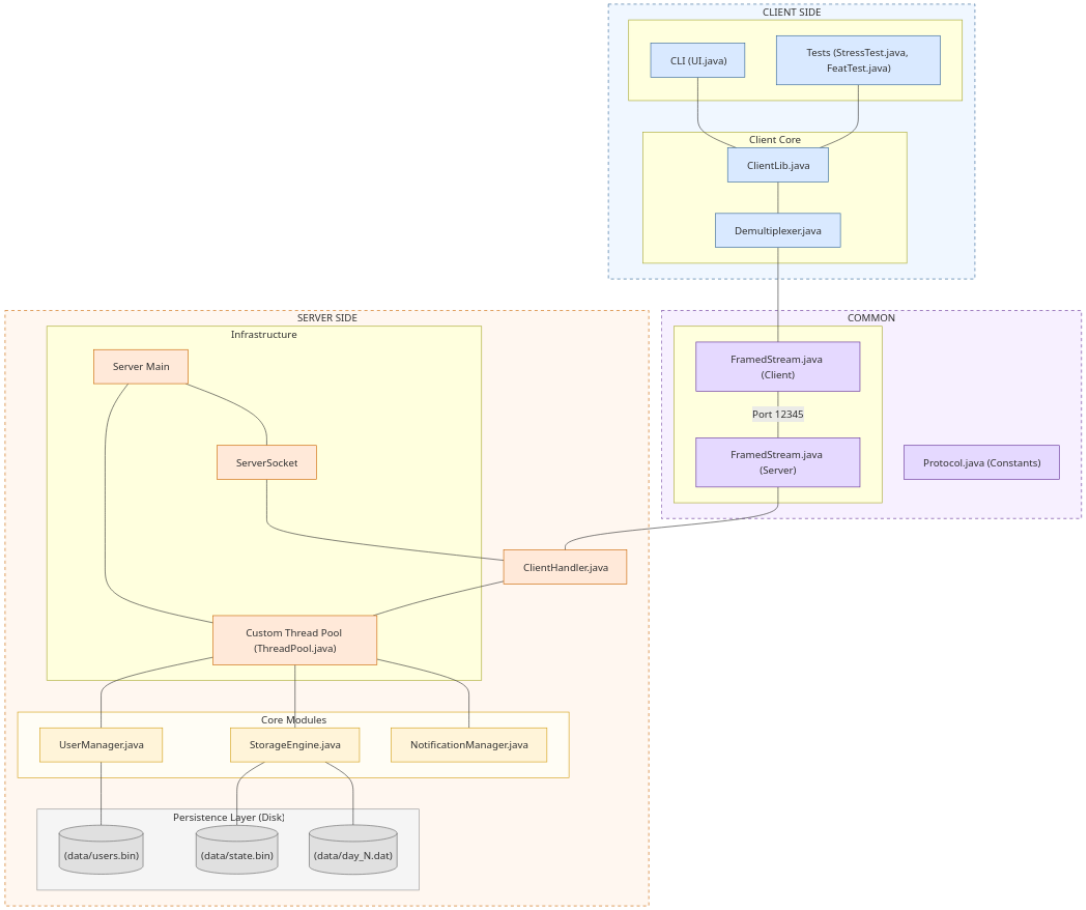

Figura 1: Diagrama da arquitetura do sistema e fluxo de mensagens entre componentes.

3 Implementação dos Requisitos

3.1 Autenticação e Gestão de Sessão

O sistema impõe autenticação para todas as operações de negócio. A lógica reside no `UserManager`, que utiliza um `ReentrantLock` para proteger o mapa de utilizadores durante acessos concorrentes. Os dados das contas são guardados em disco (`users.bin`) no formato binário, permitindo que as credenciais persistam a reinicializações do serviço. O `ClientHandler` verifica a flag `authenticated` (declarada `volatile`) antes de processar qualquer operação de negócio.

3.2 Registo de Eventos e Transição Temporal

Os eventos do dia corrente são acumulados em memória numa lista para maximizar o desempenho de escrita. A operação `NEW_DAY` encerra o ciclo temporal: os dados são serializados para ficheiros `.dat` numerados (ex: `day_0.dat`, `day_1.dat`), o contador de dias é incrementado, e as threads bloqueadas em notificações são acordadas através do `NotificationManager`. Esta transição é coordenada com locks para garantir atomicidade.

3.3 Agregações Estatísticas e Cache Lazy

As agregações são processadas de forma *lazy*: o cálculo só ocorre quando solicitado pelo cliente. O `StorageEngine` mantém uma cache de resultados (`aggCache`) indexada por par (dia, produto). Quando uma estatística é solicitada, o sistema verifica primeiro a cache; caso não exista, processa a série temporal do dia correspondente e armazena o resultado para consultas futuras.

A cache é invalidada automaticamente quando os dias saem da janela de retenção D, através da remoção proativa em `persistDay()`.

3.4 Gestão de Memória com Cache LRU

Para respeitar o limite S de séries em memória, implementámos uma cache LRU utilizando `LinkedHashMap` com o parâmetro `accessOrder=true` e sobrescrita do método `removeEldestEntry`. Quando uma série é acedida, move-se para o final da lista; quando o limite é excedido, a entrada menos recentemente utilizada é automaticamente removida.

Esta implementação carrega séries inteiras em memória. Para conjuntos de dados extremamente grandes (biliões de eventos por série, conforme mencionado no enunciado), seria necessário implementar processamento em streaming, onde os eventos são lidos, processados e descartados incrementalmente. A implementação atual é adequada para a maioria dos cenários práticos.

3.5 Filtragem Eficiente de Dados

Na listagem de eventos de um dia específico, implementámos compressão por dicionário para otimizar a transmissão. O servidor identifica os produtos únicos no conjunto de resultados, cria um mapeamento de nomes para índices inteiros, e envia primeiro o dicionário seguido dos eventos com referências aos índices. Esta abordagem reduz significativamente o tamanho das mensagens quando os mesmos produtos aparecem múltiplas vezes.

3.6 Notificações Bloqueantes

O `NotificationManager` coordena as notificações em tempo real utilizando variáveis de condição (`Condition`).

Para **vendas simultâneas**, mantém-se um `Set<String>` dos produtos vendidos no dia corrente. Threads que aguardam verificam se ambos os produtos estão no conjunto; caso contrário, bloqueiam em `await()` até serem sinalizadas por uma nova venda ou pelo fim do dia.

Para **vendas consecutivas**, rastreia-se o último produto vendido e o contador de vendas consecutivas. Adicionalmente, um `Map<Integer, Set<String>>` regista quais produtos atingiram cada contagem consecutiva durante o dia, permitindo que threads que iniciem a espera após a condição já ter sido satisfeita sejam imediatamente notificadas.

3.7 Suporte a Clientes Multi-threaded

A concorrência no cliente é gerida pela classe `Demultiplexer`, que implementa um padrão de multiplexagem sobre uma única conexão TCP:

1. Cada pedido recebe uma *tag* única gerada atomicamente.
2. A thread regista-se numa tabela de pendentes antes de enviar o pedido.
3. Uma thread de leitura dedicada recebe frames e distribui-os às threads corretas com base na tag.
4. A thread original bloqueia em `await()` até a sua resposta chegar.

Este design garante que um pedido lento (ex: notificação bloqueante) não impede outros pedidos do mesmo cliente.

3.8 Persistência e Limpeza de Dados

O `StorageEngine` persiste o estado global (dia atual e marcador de limpeza) em `state.bin`, permitindo que o servidor retome operações após reinício sem sobrescrever dados existentes. Ficheiros de dias fora da janela de retenção D são removidos de forma eficiente, iterando apenas sobre os dias ainda não limpos.

4 Validação e Teste

A robustez do sistema foi validada através de três suites de testes complementares:

O `RobustTest` foca-se na correção funcional, testando autenticação (registo, login, rejeições), registo de eventos, agregações (quantidade, volume, média, máximo), notificações simultâneas e consecutivas, concorrência do cliente (múltiplas threads numa conexão) e casos de fronteira (nomes longos, caracteres UTF-8, valores extremos).

O `StressTest` avalia o comportamento do sistema sob carga, submetendo o servidor a milhares de inserções simultâneas distribuídas por múltiplos clientes e threads. Este teste confirma a fiabilidade dos mecanismos de sincronização e a integridade dos dados, validando que os totais reportados pelo servidor coincidem com os eventos enviados.

O `CacheTest` valida especificamente a gestão de memória, forçando evicções da cache LRU através de acessos a mais de S séries e verificando que os dados são corretamente recuperados do disco.

5 Utilização de Ferramentas de IA

Conforme o enunciado, o uso de IA foi direcionado para tarefas de suporte e produtividade, não para a implementação das primitivas fundamentais de concorrência e comunicação.

5.1 Geração da Interface de Utilizador

A interface de utilizador foi desenvolvida recorrendo ao seguinte prompt:

Utilizando a API definida no ficheiro ClientLib.java anexo, cria uma interface de linha de comando em Java. A CLI deve permitir ao utilizador registar-se, fazer login e executar todos os comandos de negócio, como registar vendas, consultar agregações, filtrar eventos e aguardar notificações. Garante que existe um indicador visual do estado da sessão e que são apresentadas mensagens de ajuda sobre a sintaxe de cada comando.

Anexámos o ficheiro `ClientLib.java` ao prompt para assegurar que a interface invocava corretamente os métodos de negócio e respeitava a lógica de comunicação implementada.

5.2 Geração das Classes de Teste

Para a geração das três classes de teste do projeto, utilizámos o seguinte prompt, anexando os ficheiros `ClientLib.java` e `Protocol.java` para contexto:

Preciso de uma suite completa de testes em Java para validar um servidor de séries temporais. A biblioteca cliente está definida no ficheiro ClientLib.java anexo, e os códigos de operação estão em Protocol.java. Gera três ficheiros de teste separados:

1. RobustTest.java *– Testes de correção funcional organizados por secções: (a) Autenticação: registo de utilizador novo, rejeição de duplicados, login com credenciais corretas e incorretas, bloqueio de operações sem autenticação; (b) Registo de eventos: inserção simples, múltiplos eventos, verificação da transição de dia com getCurrentDay(); (c) Agregações: testar AGGR_QTY, AGGR_VOL, AGGR_AVG, AGGR_MAX com valores conhecidos, verificar produto inexistente retorna 0; (d) Notificações: testar waitSimultaneous() com sucesso e timeout quando o dia termina, testar waitConsecutive() com sequência completa e interrompida; (e) Concorrência do cliente: lançar 20 threads a fazer operações simultâneas numa única conexão ClientLib, medir tempo e verificar que todas completam; (f) Casos de fronteira: nomes de produtos longos (200+ caracteres), caracteres UTF-8 e especiais, quantidades e preços zero, valores numéricos grandes.*

2. StressTest.java *– Teste de carga e integridade: criar 10 clientes com 5 threads cada, cada thread insere 20000 eventos. Implementar um padrão onde 30% das operações de escrita se concentram num único produto (o mais popular), enquanto os outros 70% das escritas são espalhados por 100 produtos diferentes. Acumula os valores esperados localmente. No final, chamar newDay() e comparar os totais esperados com os valores retornados pelas agregações do servidor. Calcular e imprimir throughput (ops/seg).*

3. CacheTest.java *– Testes específicos para a cache LRU: criar dados para mais dias do que o parâmetro S (ex: 6 dias com S=3), aceder aos dias em ordem que force evicções, verificar que após evicção os dados são corretamente recuperados do disco.*

Testar também que agregações repetidas são mais rápidas (cache hit) e que grandes volumes de dados (10000 eventos/dia) são processados corretamente.

Cada ficheiro deve ter um método main() executável independentemente, imprimir relatório final com contagem de testes passados/falhados, e usar apenas as APIs padrão do Java (sem JUnit).

O código gerado foi validado manualmente, ajustando detalhes de sincronização nos testes de notificações e corrigindo a lógica de tracking de dias nos testes de fronteira.

5.3 Acesso Rápido a Documentação de API

Utilizámos a IA para agilizar a consulta das APIs das bibliotecas padrão do Java. Foi através desta exploração que identificámos o método `removeEldestEntry()` da classe `LinkedHashMap`, que permitiu implementar a política de substituição LRU de forma nativa e eficiente.

6 Conclusão

O sistema desenvolvido responde aos requisitos técnicos e de desempenho especificados. A arquitetura modular separa claramente as responsabilidades de rede, lógica de negócio e persistência. As otimizações implementadas, como a multiplexagem no cliente, a cache LRU para séries temporais, e a cache de agregações calculadas, garantem um serviço eficiente e capaz de gerir fluxos intensos de dados e consultas complexas.

A única limitação identificada relaciona-se com o processamento de séries extremamente grandes (biliões de eventos por dia): a implementação atual carrega ficheiros inteiros em memória, quando um processamento em streaming seria mais apropriado. Esta é uma área de melhoria para trabalho futuro.