**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, 3 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 para processar pedidos de forma não bloqueante. No lado do cliente, a arquitetura foi desenhada para satisfazer o requisito de uma ligação TCP única, utilizando um `Demultiplexer` que gere múltiplas threads através de um protocolo de etiquetas (*tags*). A comunicação é feita num formato binário, utilizando `DataInputStream` e `DataOutputStream` sobre um sistema de *framing* que garante que as mensagens são transmitidas e lidas na íntegra.

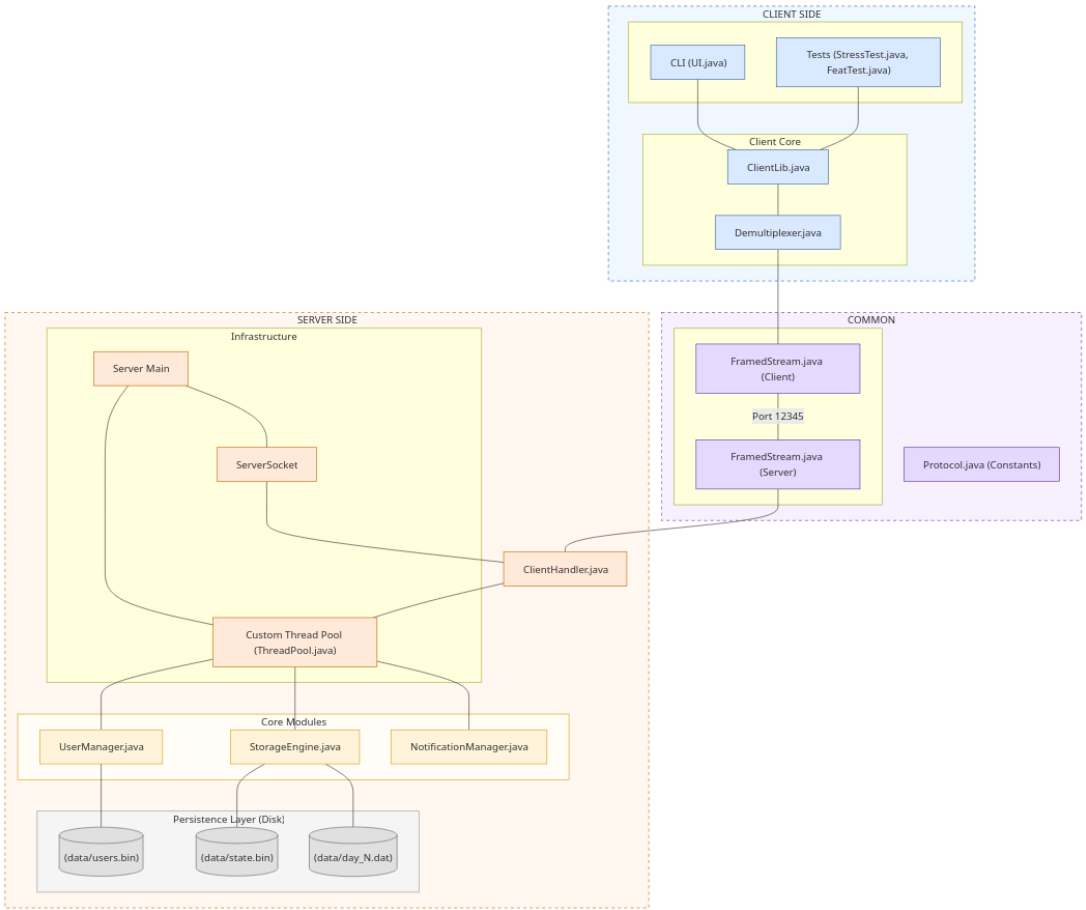


Figura 1: Diagrama da arquitetura do sistema e fluxo de mensagens entre componentes.

# 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 mecanismos de exclusão mútua (*locks*) para proteger o mapa de utilizadores. Para garantir a persistência, os dados das contas são guardados em disco (`users.bin`), permitindo que as credenciais sobrevivam a reinicializações do serviço.

## 3.2 Registo de Eventos e Transição Temporal

Os eventos do dia corrente são acumulados em memória para maximizar o desempenho de escrita. A operação `NEW_DAY` encerra o ciclo temporal: os dados são serializados para ficheiros `.dat` numerados e as threads bloqueadas em notificações são acordadas. Esta transição é coordenada de forma atómica para que nenhum evento seja perdido durante a mudança de estado.

## 3.3 Agregações Estatísticas e Cache Lazy

As agregações são processadas de forma *lazy*: o cálculo só ocorre quando solicitado. Implementámos uma cache de resultados (`aggCache`) que armazena estatísticas por par dia/produto. Em consultas subsequentes, o sistema reutiliza estes valores, evitando leituras redundantes do disco. Esta estratégia permite que o servidor responda instantaneamente a pedidos sobre períodos históricos longos.

## 3.4 Filtragem Eficiente de Dados

No requisito de listagem de eventos, implementámos compressão por dicionário. O servidor identifica produtos únicos no conjunto de resultados e envia um mapeamento de índices inteiros. Esta abordagem reduz o tamanho das mensagens transmitidas, uma vez que os nomes dos produtos não são repetidos na rede, otimizando o uso da largura de banda.

## 3.5 Notificações Bloqueantes com Timeout Implícito

As notificações utilizam variáveis de condição (`Condition.await()`). Se o critério de venda for atingido, a thread é sinalizada. Caso o dia termine antes da satisfação do padrão, a transição para um novo dia força o despertar das threads com um retorno nulo ou negativo, cumprindo a semântica de tempo real exigida.

## 3.6 Ligação TCP Única com Multiplexagem

A concorrência no cliente é gerida pela classe `Demultiplexer`. Esta centraliza a leitura do socket numa thread dedicada, distribuindo as respostas pelas threads de aplicação com base na *tag* de correlação. Isto impede o bloqueio de cabeça de linha, permitindo que uma consulta estatística pesada não impeça o envio simultâneo de novos eventos de venda.

### 3.7 Persistência com Gestão de Memória

O motor respeita os limites $S$ (séries em RAM) e $D$ (janela histórica). Utilizamos uma cache LRU baseada em `LinkedHashMap` para gerir as séries carregadas. Se uma série necessária não estiver em memória e o limite $S$ for atingido, os dados são lidos do disco e integrados na cache, forçando a expulsão da entrada menos utilizada. Este mecanismo garante que o motor opera sempre dentro dos limites de memória definidos.

## 4 Validação e Teste

A robustez do sistema foi validada através de dois cenários principais de teste. O `FeatTest` foca-se na correção funcional, testando a integridade das notificações simultâneas e consecutivas, bem como a transição entre ciclos diários e o cálculo básico de agregadores.

O `StressTest` avalia o comportamento do sistema sob carga, submetendo o servidor a milhares de inserções simultâneas distribuídas por múltiplos clientes e threads de trabalho. Este teste confirmou a fiabilidade dos mecanismos de sincronização e a integridade dos dados, garantindo que os totais reportados pelo servidor coincidem com os eventos enviados pelos clientes, mesmo em cenários de elevada concorrência.

## 5 Utilização de Ferramentas de IA

Conforme o enunciado, o uso de IA foi direcionado para tarefas de suporte e produtividade. Gerámos a estrutura da interface de utilizador e os componentes de teste automático. Todo o código produzido foi revisto e integrado manualmente para garantir a conformidade com o protocolo definido e a estabilidade da aplicação.

### 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

#### 5.2.1 Testes Funcionais (FeatTest.java)

Para a validação funcional, o prompt utilizado foi:

> *Cria um programa de teste em Java para validar o funcionamento das notificações. Deves lançar threads que fiquem a aguardar por vendas simultâneas e consecutivas, enquanto outras threads inserem eventos de venda. No final, verifica se o sistema liberta corretamente as threads quando o dia é encerrado e se as notificações refletem os eventos registados.*

#### 5.2.2 Teste de Carga (StressTest.java)

Para o teste de carga, utilizou-se o seguinte prompt:

> *Gera um programa que submeta o servidor a um elevado número de pedidos concorrentes. Cria vários clientes, cada um com múltiplas threads, para registar milhares de vendas em simultâneo. No final, o programa deve comparar os totais enviados pelos clientes com os resultados devolvidos pelos agregadores do servidor para garantir que não houve perda de dados durante o processo.*

### 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`. Esta descoberta permitiu implementar a política de substituição LRU de forma nativa e eficiente, simplificando significativamente a lógica da camada de armazenamento.

## 6 Conclusão

O sistema desenvolvido responde aos requisitos técnicos e de desempenho. A arquitetura modular e as otimizações implementadas, como a multiplexagem e a gestão inteligente de memória, garantem um serviço resiliente e capaz de gerir fluxos intensos de dados e consultas complexas com eficácia.