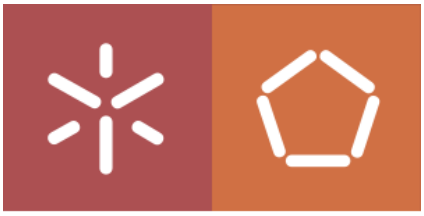


Universidade do Minho
Escola de Engenharia

Sistemas Distribuídos

Trabalho prático

Grupo 48

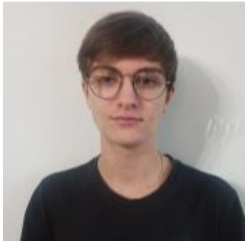
Eduarda Mafalda Martins Vieira, a104098

Maria Inês da Rocha Pinto Gracias Fernandes, a104522

Pedro Manuel Macedo Rebelo, a104091

a104098

a104522



a104091

9 de Janeiro, 2026

Índice

Introdução

Este relatório descreve o trabalho prático realizado na unidade curricular de Sistemas Distribuídos, do curso de Engenharia Informática na Universidade do Minho, referente ao ano letivo 2025/2026. O objetivo deste projeto consistiu na implementação de um serviço distribuído para o registo e agregação de eventos em séries temporais, relacionados com a venda de produtos, permitindo o acesso remoto por clientes através de *sockets* TCP.

O sistema desenvolvido suporta operações como autenticação e registo de utilizadores, inserção de eventos, consultas agregadas sobre dias anteriores e notificações de ocorrências específicas. Foi dada especial atenção à concorrência e à eficiência do processamento, garantindo que pedidos demorados não bloqueiem outras operações. A persistência dos dados e a limitação de memória também foram consideradas, assegurando que séries temporais antigas são armazenadas no disco e apenas um número limitado de dias permanece em memória.

O presente documento apresenta a arquitetura do sistema, detalha as decisões técnicas adotadas e descreve as estratégias utilizadas para comunicação, concorrência, agregação e persistência de dados. Inclui ainda os resultados dos testes realizados e uma análise do impacto das escolhas de implementação no desempenho e na escalabilidade do sistema.

Arquitetura do Sistema

O sistema foi desenvolvido seguindo uma arquitetura cliente-servidor, em que o servidor central é responsável pelo armazenamento, processamento e agregação de eventos em séries temporais, enquanto os clientes interagem remotamente através de *sockets* TCP. Esta divisão permite separar a lógica de negócios do acesso do utilizador, facilitando a escalabilidade e a manutenção do sistema.

O servidor gere todas as operações críticas, incluindo autenticação de utilizadores, registo de eventos, cálculo de agregações e envio de notificações. Para lidar com múltiplos clientes simultaneamente, o servidor utiliza *threads*, garantindo que pedidos demorados não bloqueiem outras operações. As séries temporais são armazenadas em memória apenas para os dias mais recentes, sendo os restantes persistidos em disco, garantindo eficiência na utilização de recursos.

A biblioteca cliente abstrai a complexidade da comunicação e da serialização, oferecendo uma API simples para operações como login, inserção de eventos, consultas agregadas e notificações. Por fim, a interface de utilizador destina-se a testes e demonstrações, permitindo interagir com o sistema de forma intuitiva sem interferir na lógica subjacente.

Esta arquitetura modular permite que cada componente evolua de forma independente e assegura um serviço eficiente, seguro e escalável, alinhado com os requisitos do trabalho prático.

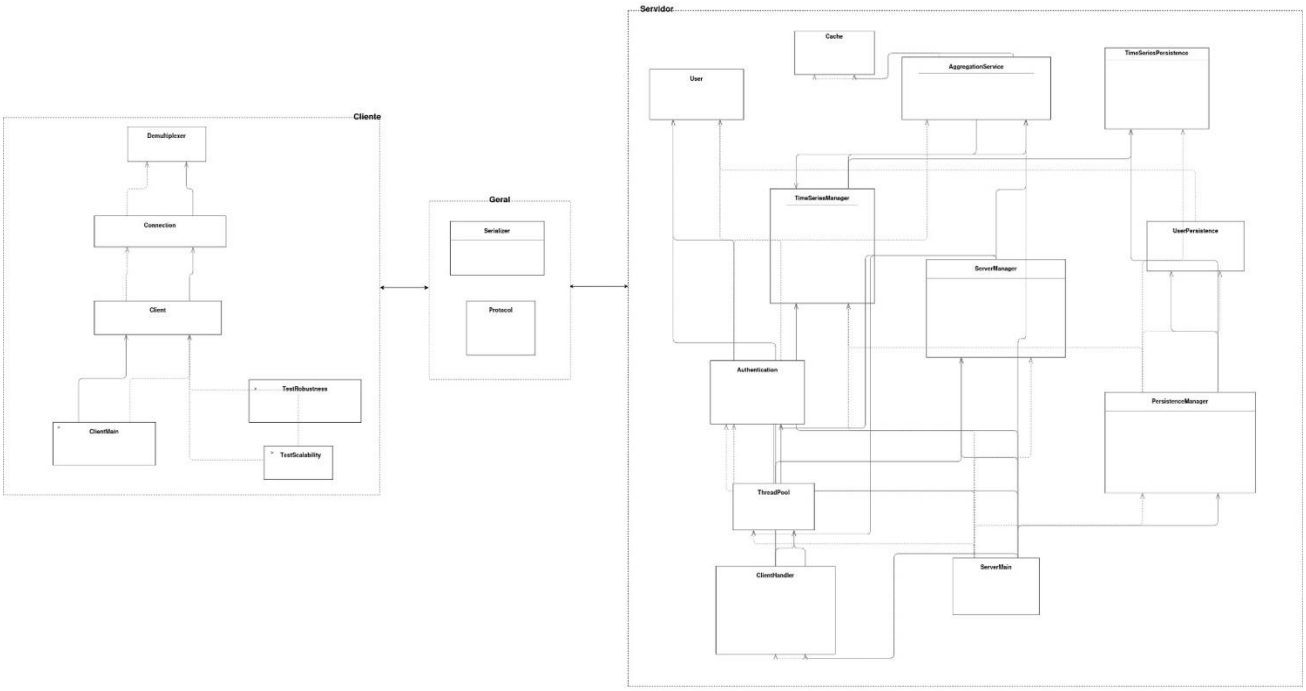


Figura 1 - Arquitetura para o sistema a desenvolver

O sistema está dividido em três packages principais que refletem a separação de responsabilidades:

- ***geral***: código partilhado para conexão entre cliente e servidor;
- ***client***: biblioteca e interface de acesso aos serviços;
- ***server***: lógica de negócio, gestão de estado e persistência.

Classes Implementadas

1. Geral

Este *package* garante que o cliente e o servidor "falam a mesma língua" (camada de comunicação).

1.1. Protocol

- Define as constantes e códigos de operação (OpCodes).
- Exemplo: Códigos inteiros para LOGIN, PUT_EVENT, GET_AGGREGATION, NEW_DAY, etc.
- Garante que quando o cliente envia um 1, o servidor sabe que é um pedido de login.

1.2. Serializer

- Garante que o protocolo de comunicação está num formato binário.
- Contém métodos para transformar objetos complexos (como uma lista de eventos vendida) em *bytes* de forma compacta e eficiente, usando apenas DataInput e DataOutput.

2. Client

Este *package* garante a funcionalidade 6 “Suporte a clientes *multi-threaded*” através de uma técnica chamada Demultiplexagem (*Tagging*).

2.1. Client

- Biblioteca do Cliente exigida no enunciado.
- Expõe métodos como login(), insertEvent(), getAverage() do lado do cliente que tratam da informação que é enviada no pedido ao servidor.
- Esconde a complexidade de rede da aplicação final.

2.2. ClientMain

- A interface de utilizador (consola) para testes.
- Lê o input do teclado e chama a biblioteca.

2.3. Connection

- Gere o Socket TCP.
- Responsável por enviar mensagens para a rede.

2.4. Demultiplexer

- Evita que as respostas do servidor às várias *threads* do mesmo cliente (que usam a mesma conexão TCP), cheguem fora de ordem.
- Cada pedido enviado leva uma ID (Tag) única. O Demultiplexer tem uma *thread* que está sempre a ler do *socket*. Quando chega uma resposta, ele vê a ID e entrega a resposta à *thread* específica que fez aquele pedido (usando Map<Integer, Condition> ou wait/notify).

3. Server

Este *package* gere autenticação, dados em memória, disco e concorrência.

A. Entrada e Conexão

3.1 ServerMain

- Inicia o *ServerSocket* e aceita conexões.

3.2 ThreadPool

- Minimiza o número de *threads*.
- Em vez de criar uma *thread* por cliente infinitamente, usa um *pool* controlado para executar as tarefas.

3.3 ClientHandler

- Representa a *thread* que atende um cliente específico.
- Lê o pedido, descodifica-o (usando Protocol), invoca a lógica de negócio e devolve a resposta.

B. Lógica de Negócio e Gestão

3.4 ServerManager

- É a classe que orquestra tudo.
- Contém referências para os gestores de utilizadores e de séries temporais.

3.5 User

- Uma biblioteca de um User expõe métodos de busca de *username* e *password*.

3.6 Authentication

- Verifica *passwords*.
- Gere o login e o registo do utilizador através de Map<String, User> (funcionalidade 1).

3.7 TimeSeriesManager

- Gere os dados do Dia Corrente (*currentDay*) e o acesso aos dias passados.
- Controla a mudança de dia (funcionalidade 2 – "Registo de Eventos", simular passagem do tempo).
- Gere os bloqueios para as Notificações (funcionalidade 5 – "Notificação de ocorrências", Vendas simultâneas/consecutivas). Usa variáveis de condição onde as *threads* ficam à espera que os eventos ocorram.

C. Agregação e Otimização

3.8 AggregationService

- Calcula as estatísticas (Soma, Média, Max).
- Implementa o modo *Lazy* (funcionalidade 3) implementando uma cache que guarda pedidos anteriores do dia corrente e os seus resultados.
- Só calcula quando pedido.

3.9 Cache

- Armazena resultados de agregações já calculadas (ex: "Média do Produto X nos últimos 3 dias").
- Evita reler do disco ou recalcular memória.

D. Persistência (sub-*package* persistence)

Responde à funcionalidade 7 – "Persistência de séries temporais e limite destas em memória" (séries com biliões de eventos e limite S em memória).

3.10 PersistenceManager

- Coordena a escrita e a leitura.

3.11 UserPersistence

- Guarda utilizadores e senhas em ficheiro.

3.12 TimeSeriesPersistence

- Escrita: Quando o dia acaba, faz *flush* dos eventos do dia corrente para um ficheiro (ex: dia_1.dat).
- Leitura: Quando é preciso uma agregação antiga, carrega o ficheiro.
- Gestão de Memória: Garante que apenas S dias estão carregados em memória. Se tentar carregar o S+1, ele descarta (faz *flush*) um dia antigo da memória.

Aplicação de exclusão mútua e variáveis de condição

A sincronização do sistema baseia-se no uso de *locks* e variáveis de condição para garantir consistência dos dados, evitar condições de corrida e minimizar a contenção entre *threads*. No servidor, o ThreadPool implementa o padrão produtor–consumidor recorrendo a ReentrantLock e Condition, permitindo que as *threads* trabalhadoras aguardem eficientemente por novos pedidos sem recorrer a *busy waiting*, assegurando simultaneamente acesso exclusivo à fila de tarefas.

A gestão das séries temporais é assegurada pelo TimeSeriesManager, que utiliza um ReentrantReadWriteLock para distinguir operações de leitura e escrita. Esta abordagem permite que múltiplas consultas e agregações sejam executadas em simultâneo, enquanto as operações de registo de eventos e mudança de dia são realizadas em exclusividade, garantindo consistência sem comprometer o desempenho.

As funcionalidades de notificação de vendas simultâneas e consecutivas são implementadas através de variáveis de condição associadas ao mesmo mecanismo de sincronização. As *threads* clientes bloqueiam em await() até que novos eventos sejam registados, sendo acordadas através de signalAll() invocado no método de inserção de vendas, garantindo que todas as condições são reavaliadas sempre que ocorre uma nova venda. Esta solução evita ciclos de espera ativa e assegura uma implementação eficiente dos requisitos de notificação.

Os módulos de cache e autenticação recorrem igualmente a *read-write locks* para proteger estruturas de dados partilhadas, garantindo atomicidade e consistência nas operações de leitura e escrita. No lado do cliente, o Demultiplexer coordena múltiplos pedidos concorrentes sobre uma única ligação TCP

através de *locks* e variáveis de condição, assegurando que cada *thread* recebe a resposta correspondente ao seu pedido, viabilizando o suporte a clientes *multi-threaded*.

Conclusões

O trabalho desenvolvido permitiu implementar um sistema distribuído completo para o registo e agregação de eventos em séries temporais, cumprindo a maioria dos requisitos propostos no enunciado. Foram implementadas com sucesso as funcionalidades de autenticação e registo de utilizadores, inserção de eventos, agregações lazy com caching, notificações bloqueantes, suporte a clientes multi-threaded e persistência de dados com limitação de séries em memória. A arquitetura adotada demonstrou ser escalável, modular e adequada à gestão concorrente de múltiplos clientes.

Destaca-se a implementação das agregações sob demanda com processamento iterativo de dados em disco, garantindo o cumprimento do limite de memória, bem como o uso de mecanismos de sincronização avançados que permitem concorrência eficiente sem comprometer a consistência dos dados. O suporte a múltiplas threads no cliente, através de um mecanismo de multiplexagem/ demultiplexagem, permitiu explorar de forma prática os desafios da comunicação concorrente sobre uma única ligação TCP.

Relativamente aos requisitos não totalmente cumpridos, a operação de filtragem de eventos encontra-se funcional, mas não implementa ainda uma serialização compacta conforme solicitado no enunciado. Adicionalmente, os testes de robustez evidenciaram uma limitação na arquitetura atual do servidor quando um cliente não consome as respostas enviadas, podendo levar ao bloqueio do processamento. Estas limitações foram identificadas e analisadas, constituindo oportunidades claras de melhoria futura.

Em suma, o projeto atingiu os principais objetivos propostos e permitiu aplicar de forma integrada conceitos fundamentais de sistemas distribuídos, nomeadamente concorrência, sincronização, comunicação entre processos, persistência e escalabilidade. As dificuldades encontradas e as limitações identificadas contribuíram igualmente para uma compreensão mais profunda dos desafios inerentes ao desenvolvimento de sistemas distribuídos reais.