



Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Sistemas Distribuídos

Ano Letivo de 2025/2026

Grupo 01

João Teixeira
A106836

Nelson Mendes
A106884

Simão Mendes
A106928

Tomás Machado
A104186

December 27, 2025

SD

Índice

1. Introdução	1
2. Arquitetura do Sistema e Protocolo	1
2.1. Visão Geral do Modelo Cliente-Servidor	1
2.2. Protocolo de Comunicação	1
2.2.1. Estrutura das Mensagens e Serialização	1
2.2.2. <i>Multiplexing</i> e Identificação de Pedidos	2
2.3. Gestão de Conexões e Autenticação	2
2.3.1. Fluxo de Autenticação	2
2.3.2. Gestão de Sessão	2
3. Gestão de Dados e Persistência	3
3.1. Estrutura de Dados em Memória	3
3.2. Persistência e Formato de Ficheiros	3
3.3. Algoritmo de Gestão da Cache (LRU)	3
4. Processamento e Notificações	4
4.1. Agregação de Dados e Estratégia de <i>Caching</i>	4
4.2. Gestão de Notificações Bloqueantes	5
5. Avaliação Experimental	5
5.1. Metodologia	5
5.2. Análise	5
5.3. Discussão	6
6. Conclusão	6
7. Bibliografia	7
8. Anexos	7

1. Introdução

Foi proposto o desenvolvimento de um sistema cliente-servidor para a gestão de séries temporais massivas, focado no domínio de vendas. O objetivo central deste projeto é a **construção de uma infraestrutura robusta capaz de suportar acessos concorrentes e operações de agregação de dados** em tempo real, utilizando exclusivamente as primitivas base da linguagem Java para comunicação e sincronização.

O desafio técnico principal residiu na **implementação de um sistema distribuído de raiz**, com auxílio de mecanismos de serialização de alto nível para garantir a máxima eficiência no tráfego de rede. A arquitetura da solução baseia-se num modelo *Multiplexing* de conexões, onde a gestão de pedidos é orquestrada por componentes de *Middleware* (como o Demultiplexer e TaggedConnection), permitindo desacoplar a lógica de negócio da complexidade da comunicação assíncrona.

Para garantir a **consistência** e o **desempenho do servidor**, foram adotadas estratégias avançadas de concorrência, nomeadamente a **utilização de ReentrantReadWriteLocks para otimizar o acesso a estruturas de dados partilhadas** e a implementação de mecanismos de cache para acelerar consultas frequentes. A persistência dos dados é assegurada através de um sistema de ficheiros *log-based*, que garante durabilidade entre execuções.

O trabalho aqui realizado foi auxiliado, em algumas partes do seu desenvolvimento, com inteligência artificial. Os *prompts* utilizados serão apresentados no capítulo Secção 8 (Anexos).

2. Arquitetura do Sistema e Protocolo

2.1. Visão Geral do Modelo Cliente-Servidor

O sistema implementa uma arquitetura cliente-servidor baseada em *sockets TCP*, onde cada cliente mantém uma única conexão persistente com o servidor. O servidor segue o padrão *Thread-Per-Connection*, garantindo o isolamento entre clientes.

No lado do cliente, a arquitetura é mais complexa. O *SalesClient* utiliza uma **única conexão física** (TaggedConnection), mas implementa um **mecanismo de demultiplexação** que permite um processamento assíncrono de múltiplas operações concorrentes.

2.2. Protocolo de Comunicação

2.2.1. Estrutura das Mensagens e Serialização

O protocolo utiliza uma estrutura de *frames* etiquetadas onde cada mensagem contém três componentes: um **identificador numérico** (*tag*) de 4 bytes, o **comprimento dos dados** (4 bytes) e o **payload** propriamente dito. Esta estrutura está implementada na classe TaggedConnection.Frame.

A **serialização binária** é realizada através das classes `DataInputStream` e `DataOutputStream`, que permitem escrita e leitura tipo-por-tipo (UTF para *strings*, int para inteiros, double para valores decimais).

2.2.2. Multiplexing e Identificação de Pedidos

O suporte para um cliente *multi-threaded* é assegurado pela classe `Demultiplexer`. Este mecanismo funciona da seguinte forma:

1. Cada operação do cliente é associada a uma **tag** única que identifica o tipo de pedido;
2. O `Demultiplexer` cria uma **thread de receção** que monitoriza continuamente a conexão;
3. Quando uma resposta chega, o `Demultiplexer` verifica a **tag** e encaminha os dados para a fila correspondente;
4. A **thread** que aguardava por essa resposta específica é **notificada e pode processar os dados**.

Esta abordagem, visível no Anexo 3, permite que **múltiplas threads no cliente** façam pedidos simultâneos através da mesma conexão TCP, sendo cada resposta corretamente direcionada ao solicitante original. O `Demultiplexer` mantém um mapa (`buf`) que associa cada **tag** a uma `Entry` contendo uma fila de mensagens e uma variável de condição para sincronização.

2.3. Gestão de Conexões e Autenticação

2.3.1. Fluxo de Autenticação

A autenticação, visível no Anexo 4, segue o seguinte fluxo:

1. **Registo:** O cliente invoca um método que envia a **tag REGISTER** com as credenciais serializadas;
2. O servidor **verifica no UserManager se o utilizador já existe**;
3. **Login:** Após um registo bem-sucedido, o cliente usa faz **login** com a **tag LOGIN**;
4. O **ServerWorker valida as credenciais** e marca a sessão como autenticada.

2.3.2. Gestão de Sessão

Uma vez autenticado, o `ServerWorker` mantém o estado `isAuthenticated` e `currentUsername` para a sessão. **Todas as operações subsequentes verificam primeiro este estado**. Caso o cliente não esteja autenticado, o servidor rejeita a operação com uma mensagem de erro apropriada.

A **persistência de utilizadores é garantida através da classe UserManager**, que carrega e armazena os dados com recurso a serialização binária. O mecanismo de *shutdown* no `ServerMain` assegura que os dados sejam persistidos mesmo em encerramentos abruptos.

3. Gestão de Dados e Persistência

A arquitetura de gestão de dados foi desenhada para garantir **eficiência no acesso à informação** recente (dia corrente) e **escalabilidade no acesso ao histórico** (D dias), de modo a cumprir o requisito de manter em memória apenas um subconjunto limitado de séries temporais (S).

3.1. Estrutura de Dados em Memória

A representação fundamental dos dados reside na classe `TimeSeries`, que encapsula todos os eventos de vendas ocorridos num único dia. A estruturação dos dados tem o seguinte formato:

- **Organização Hierárquica:** Internamente, a `TimeSeries` utiliza um **Mapa de Hash** (`HashMap<String, ProductEvent>`) para indexar os eventos por produto. Esta escolha privilegia a velocidade de acesso ($O(1)$) para operações de escrita (adicionar venda) e leitura (consultar produto específico);
- **Aggregação por Produto:** Cada entrada no mapa aponta para um objeto `ProductEvent`, que mantém a lista de vendas individuais (`SalesEvent`) e pré-calculta agregações simples (como totais) para otimizar consultas futuras;
- **Separação de Responsabilidades:** O estado global do servidor (`ServerState`) mantém explicitamente duas referências distintas:
 1. **currentSeries:** A série do dia atual, mantida permanentemente em memória e aberta para escritas concorrentes;
 2. **cache:** Um gestor que intermedeia o acesso aos dias históricos, decidindo o que reside em RAM e o que deve ser carregado do disco.

3.2. Persistência e Formato de Ficheiros

A **persistência de dados** é gerida pelo `SeriesFileManager`, que fragmenta o histórico em ficheiros binários diários (`series_YYYY-MM-DD.dat`), permitindo o **carregamento seletivo** (*Lazy Loading*) apenas dos dias necessários para as operações.

Optou-se por um **formato binário customizado sequencial** (via `DataOutputStream`), evitando a sobrecarga da serialização nativa e utilizando *buffers* para otimização de I/O. O ficheiro estrutura-se em: **cabeçalho** (contagem), **corpo** (eventos por produto) e **metadados**. Adicionalmente, a persistência de credenciais e registo de clientes é assegurada de forma análoga pela classe `UserManager` num ficheiro dedicado (`users.dat`).

3.3. Algoritmo de Gestão da Cache (LRU)

A **gestão da memória RAM** é assegurada por duas estruturas na classe `ServerCache`, ambas seguem a política **LRU** (*Least Recently Used*).

Estratégia de Implementação: As *caches* utilizam `LinkedHashMap com accessOrder = true`. Esta configuração reordena os elementos a cada acesso, mantendo os itens recentes no final e os menos usados no início para remoção. Para garantir a consistência sob concorrência, todos os

acessos **são protegidos por um writeLock**, necessário devido às modificações estruturais que o LinkedHashMap sofre mesmo em leituras.

Gestão de Séries Temporais (Limite S): O método `removeEldestEntry()` impõe o limite rígido S de séries em memória. A remoção de elemento da *cache* segue a seguinte lógica de **Write-Back**:

1. Ao atingir o limite, **identifica-se a série LRU**;
2. **Persistência Condicional:** Se a série estiver no conjunto `modifiedSeries`, é serializada para disco antes da remoção, caso contrário, é apenas descartada.

A estratégia utilizada pela *cache* de séries pode ser consultada no Anexo 5.

Gestão de Agregações (Limite de Resultados): A `aggregationCache` também implementa um limite LRU para evitar o crescimento ilimitado da *heap*. Por serem dados derivados, as entradas **são descartadas sem persistência quando o limite é atingido**. Esta *cache* é **totalmente limpa via `clearAggregations()` no início de cada dia**, invalidando resultados de janelas temporais obsoletas.

A estratégia utilizada pela *cache* de agregações pode ser consultada no Anexo 6.

4. Processamento e Notificações

Este módulo constitui **o núcleo lógico do servidor**, onde são centralizados e manipulados dados em duas estruturas especializadas: o `ServerState`, que **gera a persistência e consulta de séries temporais**, e o `NotificationManager`, **responsável pela orquestração de eventos em tempo real**.

4.1. Agregação de Dados e Estratégia de *Caching*

O cálculo de estatísticas sobre janelas deslizantes de D dias utiliza ***Lazy Evaluation*** com ***Caching para mitigar a latência de I/O*** (Anexo 1). O processo de consulta foi otimizado através de dois vetores:

1. **Recuperação Eficiente:** Cada pedido gera uma chave única (`Op_Produto_Janela`). Se o resultado existir em *cache*, é devolvido imediatamente, evitando a reconstrução dispensiosa do estado a partir das `TimeSeries`;
2. **Consistência Temporal:** A operação `startNewDay` atua como ponto de sincronização, invalidando a *cache* para garantir que as agregações refletem a nova janela temporal relativa.

A integridade é **assegurada** por um `ReentrantReadWriteLock`, preferido ao `synchronized` para permitir que múltiplos leitores (agregações) operem em paralelo, bloqueando apenas durante escritas críticas.

4.2. Gestão de Notificações Bloqueantes

Para as subscrições de eventos, implementou-se um modelo de espera passiva com `ReentrantLock` e `Condition`, eliminando o custo de *busy waiting* (Anexo 2). O sistema distingue dois tipos de monitorização:

- **Vendas Simultâneas:** O servidor verifica se o conjunto de produtos vendidos no dia corrente satisfaz algum par subscrito, notificando a condição específica desse par;
- **Vendas Consecutivas:** Mantém-se um registo do último produto e contadores de ocorrências, de modo a despertar apenas as *threads* cujos limiares específicos foram atingidos.

Um mecanismo crítico implementado foi a validação de **Épocas** (Epochs). Cada bloqueio regista o dia de entrada (`currentDayEpoch`). **Se a thread acordar e a época tiver mudado**, a operação aborta, de forma a prevenir notificações inconsistentes baseadas em estados de dias anteriores.

5. Avaliação Experimental

5.1. Metodologia

A avaliação do sistema visou **validar o comportamento integrado das camadas de comunicação, concorrência e armazenamento sob condições de carga**. Utilizou-se um ambiente isolado em `localhost` para medir o desempenho intrínseco do servidor, com intuito de eliminar variáveis externas de rede e garantir a repetibilidade dos testes.

A metodologia baseou-se em **três eixos fundamentais que cobrem a totalidade da aplicação**:

- **Eficiência do Middleware e Protocolo:** Teste de débito de mensagens para avaliar se o Demultiplexer e a serialização binária permitem o processamento fluído de múltiplos pedidos simultâneos sobre uma única ligação TCP, quantificando o *throughput* máximo suportado;
- **Contenção e Concorrência:** Simulação de fluxos paralelos de escrita (registo de vendas) e leitura (notificações e agregações), avaliando o impacto dos mecanismos de bloqueio (`ReadWriteLock` e `ReentrantLock`) na latência percebida pelo cliente sob carga intensa;
- **Gestão de Persistência e Memória:** Monitorização do ciclo de vida das séries temporais entre a RAM e o disco, validando a eficácia da política de remoção LRU quando o volume de dados excede o parâmetro S e o impacto das otimizações de I/O do sistema operativo.

5.2. Análise

Os testes confirmaram que o protocolo binário **reduz significativamente o overhead de comunicação face a formatos textuais**, permitindo ao `ServerWorker` despachar pedidos com elevada rapidez. O sistema atingiu débitos de pico superiores a **68.000 operações por segundo**, demonstrando-se assim uma escalabilidade vertical eficaz. No domínio da concorrência, a estratégia de **locks de granularidade fina provou ser robusta**: os bloqueios de escrita na série do dia corrente não impediram a execução de operações de leitura, resultando numa **latência média estável na ordem dos 0,018 ms**.

Relativamente à persistência, o sistema manteve **um consumo de memória estável**. A análise

comparativa demonstrou que, embora a política LRU force a libertação de memória (com $S = 1$), o tempo de recuperação de dados do disco (*cache miss*) foi estatisticamente idêntico ao acesso à memória (*cache hit*), situando-se ambos nos $\sim 0,029\text{ ms}$. Este comportamento valida o aproveitamento eficiente da *Page Cache* do sistema operativo, que mantém os ficheiros .dat ativos em memória física, mitigando a latência de I/O mecânico em acessos frequentes.

A análise quantitativa dos tempos de execução, a evolução da latência e a comparação de acesso a memória encontram-se documentados em Anexo 7, Anexo 8 e Anexo 9, respetivamente.

5.3. Discussão

O fenómeno descrito anteriormente deve-se à **eficaz cooperação entre a gestão da aplicação e a Page Cache do sistema operativo**. Ao serializar uma série para o disco, o sistema operativo mantém esses dados em cache na memória física, efetivamente criando uma camada de *caching* secundária e transparente. Assim, uma *cache miss* na aplicação frequentemente resulta num *cache hit* na *Page Cache*, mitigando drasticamente a penalidade de I/O, tradicionalmente associada à persistência em disco.

Conclui-se, portanto, que a arquitetura de persistência adotada oferece uma **robustez notável face à restrição de memória**. O parâmetro S atua primariamente como um controlador de consumo de RAM pela aplicação, sem degradar a performance percebida, desde que o volume de dados ativos não exceda a memória física total disponível para a *Page Cache*.

6. Conclusão

O projeto desenvolvido resultou **num sistema distribuído de gestão de séries temporais funcional e eficiente**, capaz de cumprir todos os requisitos funcionais e não-funcionais propostos. A arquitetura baseada num protocolo binário customizado revelou-se uma decisão acertada, permitindo atingir débitos elevados **na ordem das 68.000 operações por segundo em regime de stress**, como demonstrado nos testes de carga.

A análise dos resultados experimentais valida as principais opções de desenho:

- **Eficiência do Protocolo:** A baixa latência média observada (consistentemente abaixo de 0.020 ms) confirma que a sobrecarga da serialização manual é mínima comparada com abordagens baseadas em texto ou objetos;
- **Concorrência:** A utilização de ReentrantReadWriteLocks provou ser eficaz na gestão de múltiplos clientes, permitindo que operações de leitura (agregações) ocorram em paralelo sem serem bloqueadas, o que é crítico para um sistema deste tipo;
- **Middleware:** A abstração criada pelas classes TaggedConnection e Demultiplexer simplificou significativamente o desenvolvimento, garantindo a entrega ordenada e correta das mensagens mesmo em cenários de alta concorrência.

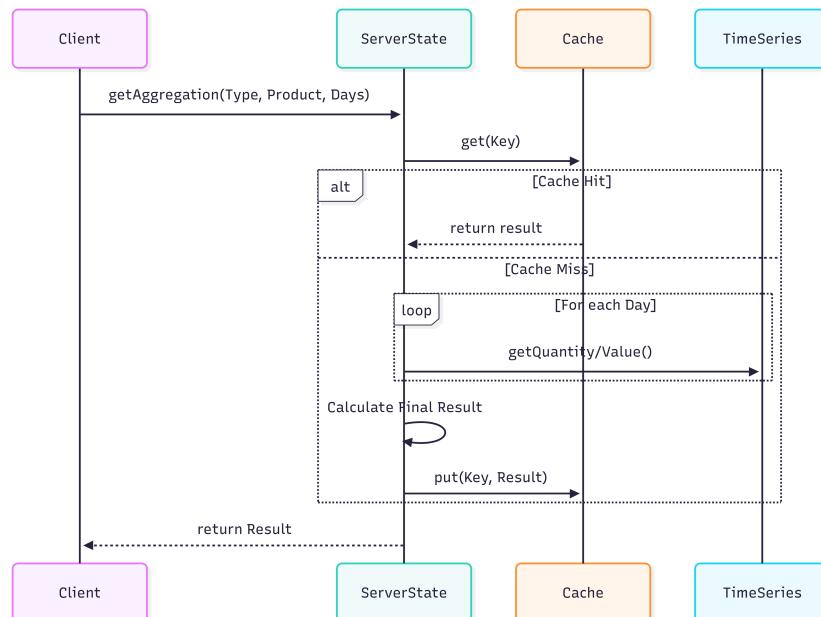
Apesar do desempenho positivo, **identificaram-se limitações no modelo de persistência** simples com um ficheiro único por utilizador, que poderá tornar-se um gargalo com o crescimento exponencial do volume de dados. **Futuras iterações do sistema** poderiam beneficiar da implementação de índices em disco ou **de uma estratégia de sharding** dos ficheiros de logs. **Em suma**, o trabalho permitiu consolidar competências críticas na engenharia de sistemas distribuídos, demonstrando que é **possível construir soluções de alto desempenho em Java** através de uma gestão cuidadosa dos recursos de rede e de processamento.

7. Bibliografia

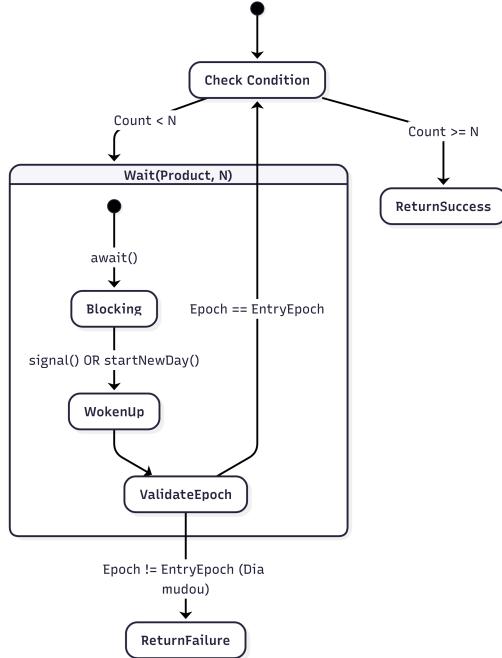
- Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2011). Distributed Systems: Concepts and Design (5th ed.). Addison-Wesley.
- Oracle. (2024). Java Platform, Standard Edition & Java Development Kit Version 17 API Specification. Disponível em: <https://docs.oracle.com/en/java/javase/17/docs/api/index.html>
- Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2006). Java Concurrency in Practice. Addison-Wesley Professional.

8. Anexos

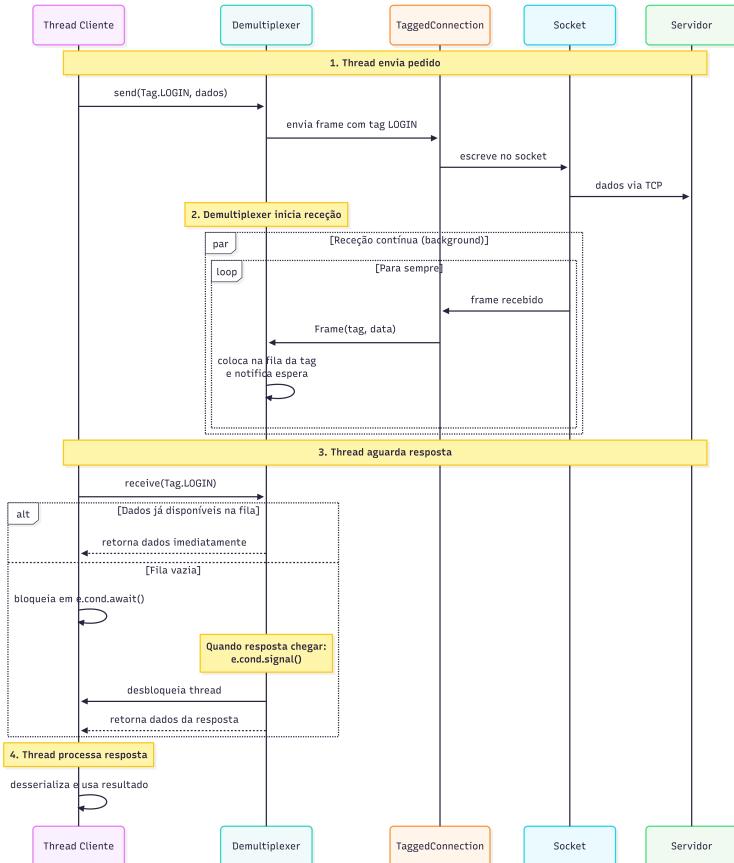
Apresentam-se abaixo os diagramas de atividade e de estados que detalham os fluxos de execução descritos na secção Secção 4.



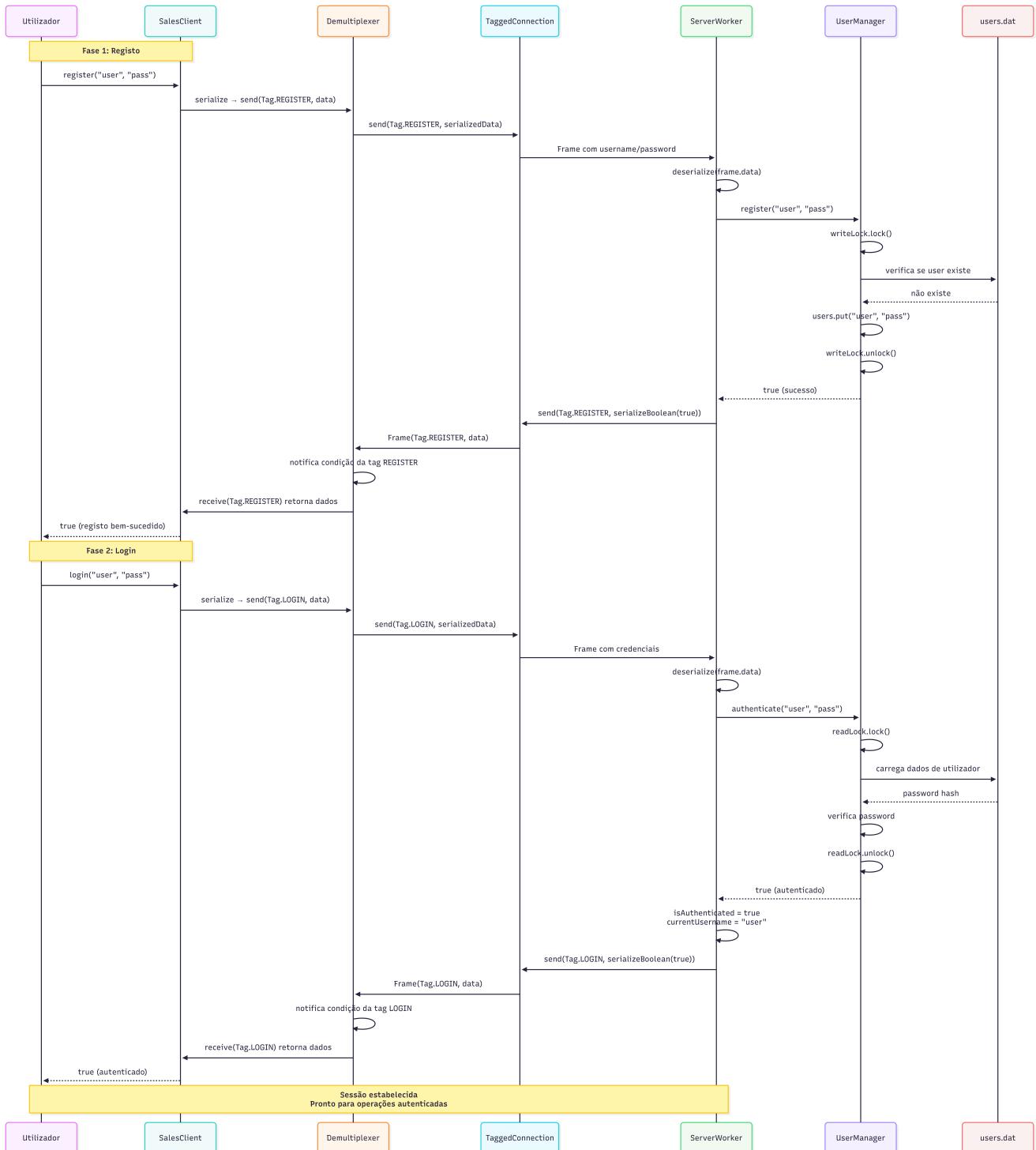
Anexo 1: Fluxo de decisão da estratégia de *Lazy Evaluation*. O sistema prioriza a recuperação rápida via cache, recorrendo ao cálculo histórico apenas quando necessário, garantindo eficiência no acesso a dados frequentes.



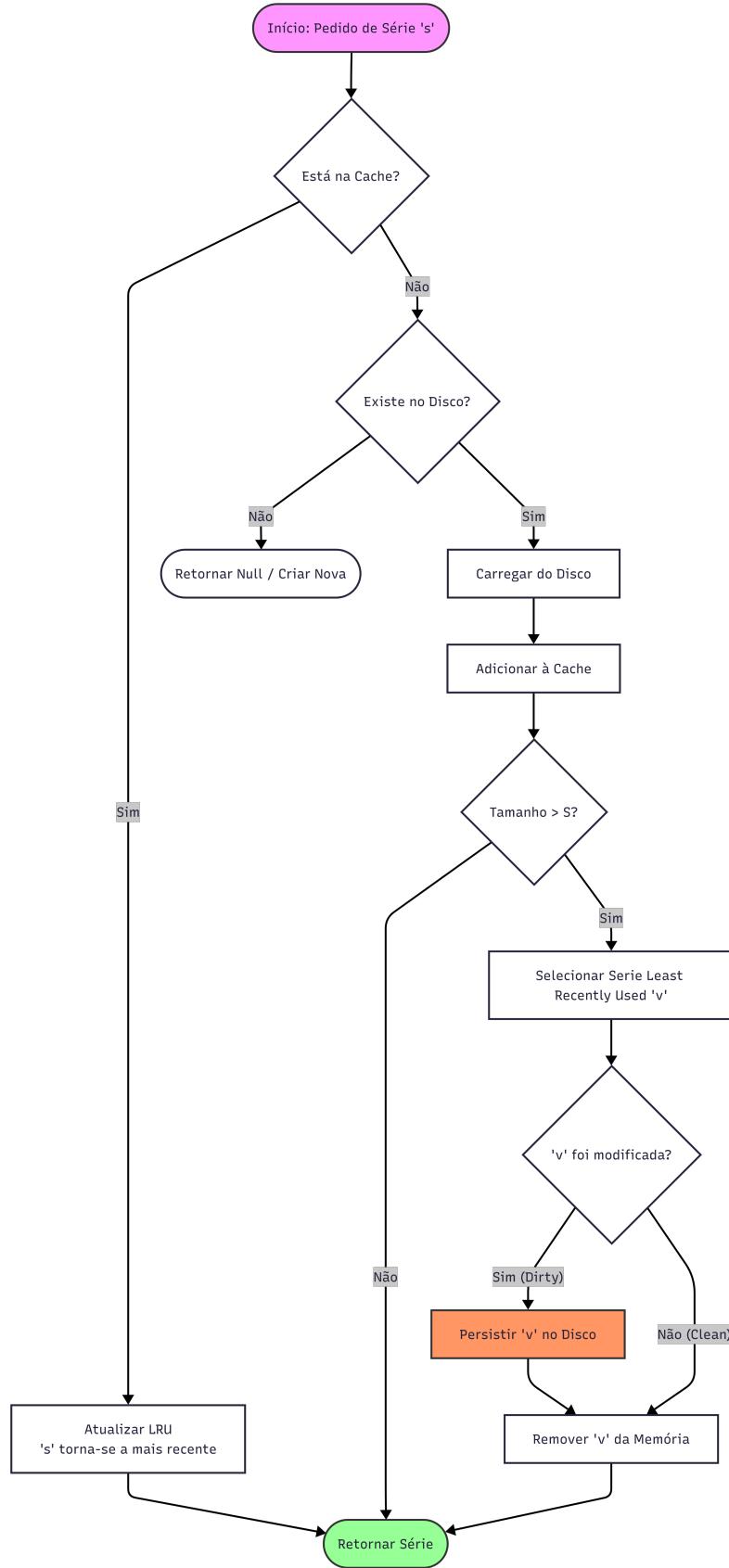
Anexo 2: Diagrama de estados do mecanismo de notificação. Destaca-se a validação de *Epochs* após o despertar da *thread* (estado *ValidateEpoch*), crucial para garantir que a condição cumprida pertence ao dia lógico correto.



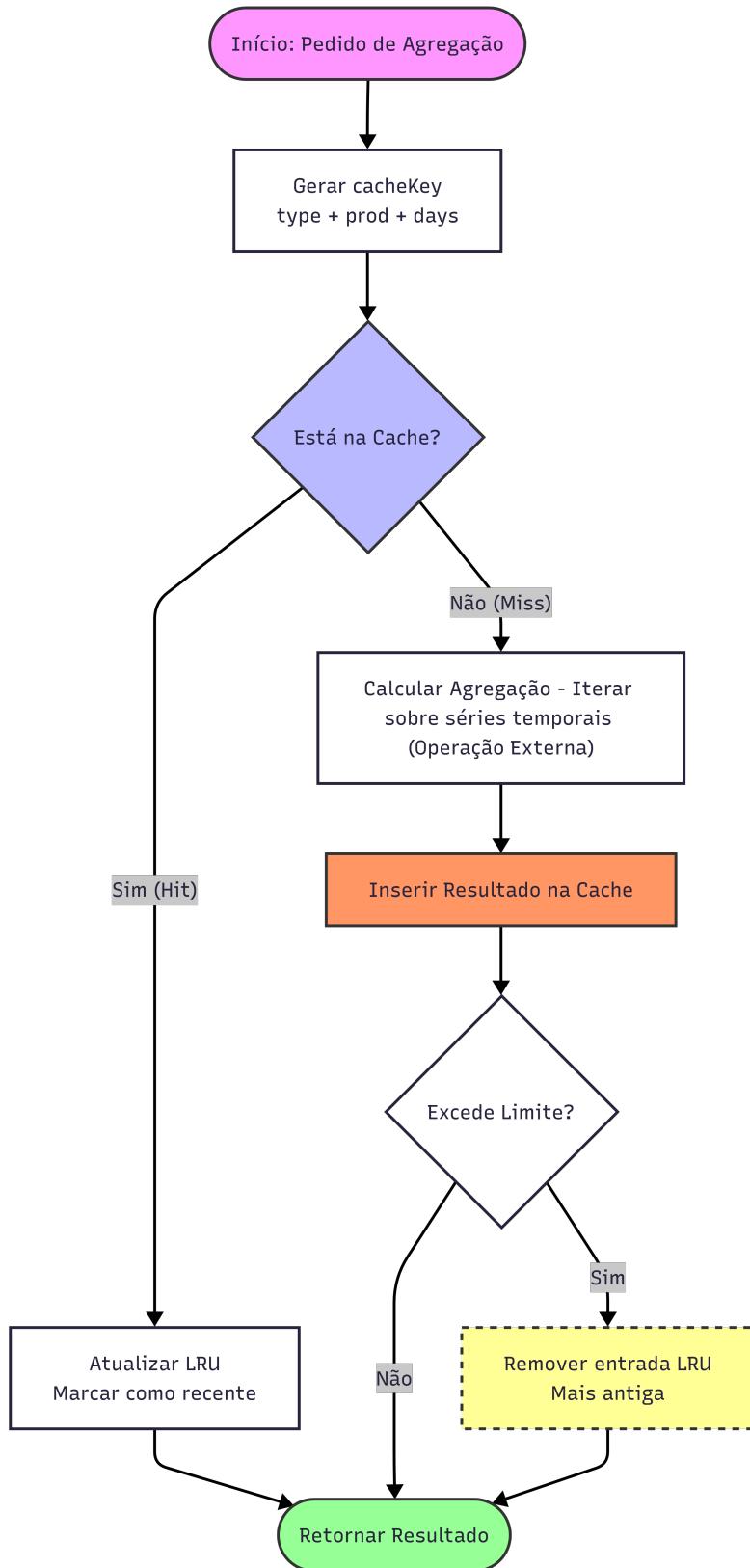
Anexo 3: Diagrama de Sequência, exemplificando o funcionamento da classe **Demultiplexer** e a **Identificação de Pedidos**, incluindo os métodos e parâmetros utilizados no código.



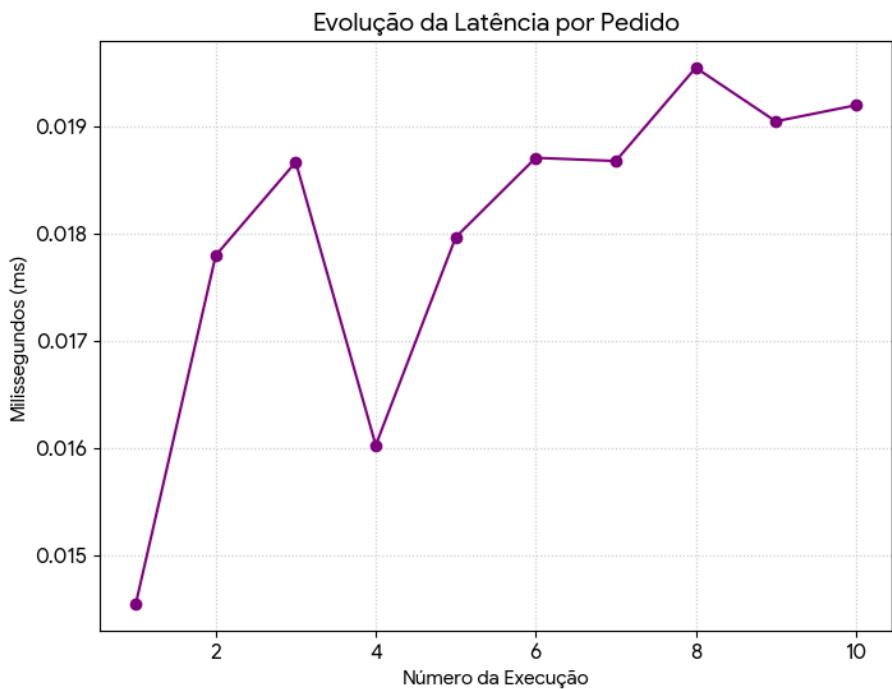
Anexo 4: Diagrama de Sequência, demonstrando o processo de **Autenticação**, incluindo os métodos e parâmetros utilizados no código.



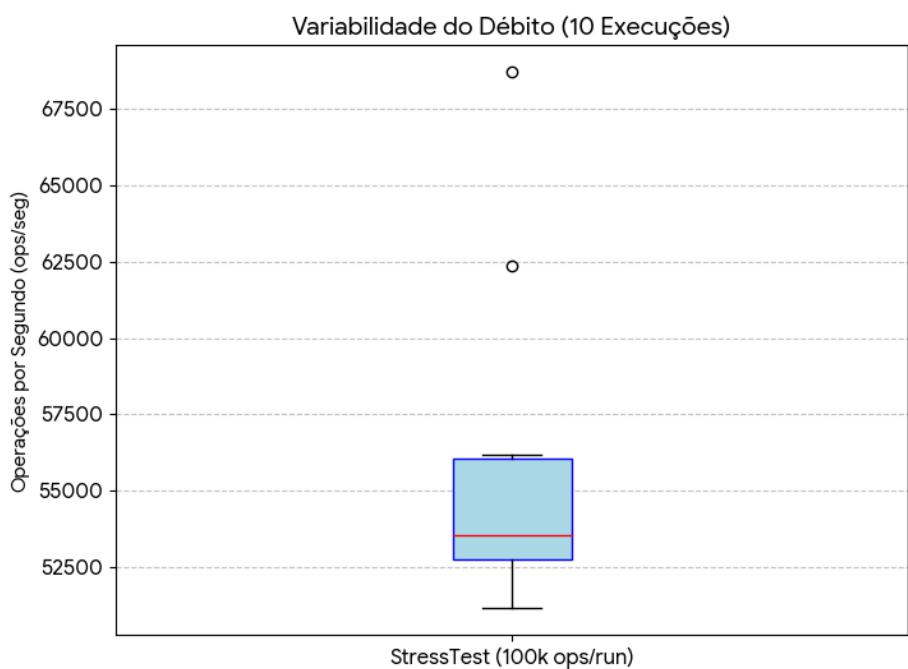
Anexo 5: Diagrama de Atividade do funcionamento da *cache* de Séries temporais.



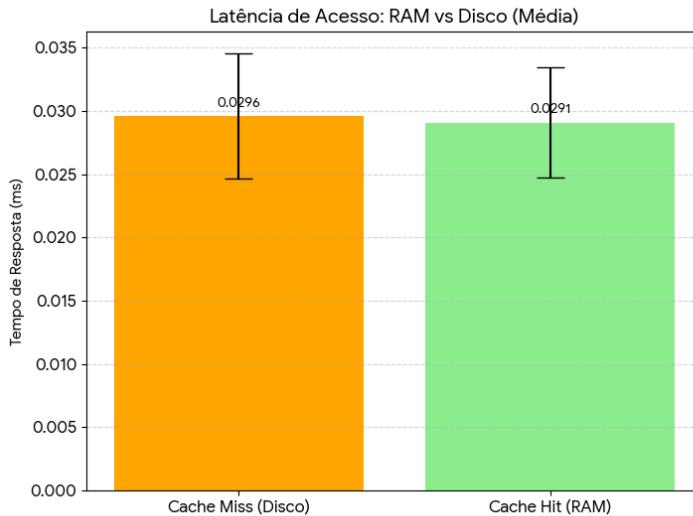
Anexo 6: Diagrama de Atividade do funcionamento da *cache* de agregações de dados.



Anexo 7: Débito de mensagens em regime de stress (10 execuções de 100.000 ops). O sistema demonstrou uma capacidade de processamento elevada, com picos de 68.734 ops/seg, validando o baixo *overhead* do protocolo binário e a eficiência do Demultiplexer.



Anexo 8: Evolução da latência média por pedido ao longo de 10 execuções independentes. A manutenção dos tempos de resposta consistentemente abaixo dos 0,020 ms comprova a eficácia dos *ReadWriteLocks* em minimizar a contenção entre *threads*.



Anexo 9: Comparaçāo entre latēncia de *Cache Miss* (leitura lógica de disco) e *Cache Hit* (leitura em RAM). A paridade dos tempos ($\sim 0,029\text{ ms}$) evidencia a ligação entre a *cache* aplicacional e o *Page Cache* do sistema operativo, que garante alta performance mesmo com um limite de memória restrito ($S = 1$).

Aqui são apresentados os *prompts* utilizados nas ferramentas de inteligēncia artificial para algumas partes do desenvolvimento deste trabalho.

- **1º Prompt:**

- ▶ “De forma a suplementar a parte do Multiplexing e Identificação dos Pedidos e o Fluxo de Autenticação, não achas que poderíamos fazer uns diagramas de sequênciā? Se sim, gera o código a descrever estes dois fluxos para meter no *Mermaid Chart*.”

- **2º Prompt:**

- ▶ “Cria um programa Java chamado `StressTestClient` que utilize o meu `SalesClient` para realizar testes de performance. O programa deve primeiro realizar um povoamento massivo de 100.000 eventos para criar peso real nos ficheiros de disco. Em seguida, deve medir o débito (ops/s) através de *threads* concorrentes em operações de leitura e escrita simultâneas. Deve também comparar a latēncia de *Cache Miss* contra *Cache Hit* e, no final, gerar uma tabela formatada que apresente os resultados de cada execução, incluindo o débito, as latēncias e o ganho de eficiēncia da *cache*.”

- **3º Prompt:**

- ▶ “Usa o código atual do `SalesClient` como base e melhora apenas a `UserInterface` para torná-la visualmente mais apelativa. Adiciona cores ANSI simples e símbolos de consola (como `[+]`, `[!]`, `>>>`) para destacar as ações. O objetivo é que a interação via terminal seja mais clara e profissional, mantendo toda a lógica funcional original.”

- 4º **Prompt:**

- ▶ “Act as a Senior Java Developer. I will provide you with the raw Java source code for a distributed systems project (a Sales Tracking System). Your task is to generate comprehensive Javadoc comments in English for all classes, fields, and methods.

Guidelines:

- **Concurrency:** Explicitly explain how thread safety is achieved (e.g., usage of ReentrantLock, Condition, and synchronized blocks);
- **Architecture:** Describe the role of middleware components like Demultiplexer and TaggedConnection in abstracting network complexity;
- **Protocol:** Detail the binary framing format (Tags/Length/Data) where applicable;
- **Style:** Use a formal, academic tone suitable for a technical report.

Please apply these comments directly to the code provided.”

- 5º **Prompt:**

- ▶ “Age como um *Tech Lead* ou Engenheiro de *Software* Sénior. Com base no código e no relatório do projeto que desenvolvemos – um **Sistema Distribuído de Gestão de Vendas e Séries Temporais** em Java –, a tua tarefa é gerar um ficheiro README.md completo, profissional e bem estruturado em *Markdown*. O documento deve seguir uma estrutura técnica e incluir os seguintes pontos:

- **Título e Introdução:** Um resumo claro do projeto, destacando que é um sistema cliente-servidor que utiliza primitivas de sincronização Java (sem bases de dados externas) e um protocolo binário customizado para alta performance.
- **Arquitetura do Sistema:** Explica brevemente o modelo de *Middleware* implementado (classes Demultiplexer e TaggedConnection) e como ele gera o *multiplexing* de conexões. Menciona a estratégia de concorrência no servidor (uso de ReentrantReadWriteLock e modelo *Thread-Per-Connection*) e a gestão de memória (Cache LRU).
- **Instalação e Compilação:** Instruções passo-a-passo para compilar o projeto (menciona o uso do *Gradle/gradlew*) e requisitos prévios (Java 21).
- **Como Executar:** Exemplo de comando para iniciar o **Servidor** (expli-cando os argumentos: <cache_size_S> e [port]). Exemplo de comando para iniciar o **Cliente**.
- **Funcionalidades:** Lista as principais operações suportadas (Registo/ Login, inserção de vendas, consultas de agregação e subscrição de notifi-cações).
- **Estrutura do Projeto:** Uma breve árvore de ficheiros ou descrição dos pacotes principais (`sd.server`, `sd.client`, `sd.middleware`, `sd.common`).

Mantém um tom formal e académico, adequado para submissão num contexto universitário de Engenharia Informática, mas com a clareza de um repositório *open-source* profissional.”