Universidade do Minho
Escola de Engenharia

Sistemas Distribuídos — Trabalho Prático
Base de dados para séries temporais

Trabalho realizado por:
Gonçalo Sá(A107376), João Sousa(A106900),
José Machado(A106926) e José Rocha(A106887)

Engenharia Informática
Universidade do Minho
2025/2026

1.Introdução

Este projeto consiste no desenvolvimento de um serviço distribuído para o registo e agregação de eventos em séries temporais utilizando a linguagem Java. O sistema permite que múltiplos clientes registem dados de vendas num servidor central, que gere o histórico dos últimos D dias. Para garantir a eficiência, implementou-se um controlo de memória RAM através de um limite de S séries em cache, recorrendo à persistência em disco para os dados extras.

Respeitando os requisitos propostos, a comunicação assenta em sockets TCP com suporte a multiplexagem (uma única conexão por cliente para múltiplas threads) e um protocolo binário proprietário para minimizar o tráfego de rede. A arquitetura foca-se na redução da contenção através de locks granulares e na minimização de threads suspensas por notificações seletivas por produto.

2.Protocolo

O sistema utiliza um protocolo de comunicação binário proprietário, assente na troca de pacotes estruturados que garantem a integridade e a correta interpretação dos dados entre as partes.

2.1. Estrutura e Serialização de Dados

A unidade base de troca de informação é o Frame. Para lidar com a natureza de *stream* do TCP, cada pacote inicia-se com um campo de tamanho, permitindo ao recetor delimitar as mensagens corretamente. A serialização é feita via DataInputStream/OutputStream, assegurando a precisão dos tipos primitivos.

FRAME:

Size-4bytes	Tag-4 bytes	Opcode-1 byte	Payload(N bytes)

Figura 1: Estrutura do Frame.

```java
public static Frame deserialize(DataInputStream in) throws IOException {
    int totalSize = in.readInt();
    int tag = in.readInt();
    int opCode = in.readInt();
    byte[] payload = new byte[totalSize - 8]; // Removemos os 8 bytes da t
    in.readFully(payload);
    return new Frame(tag, opCode, payload);
}
```

```java
public void serialize(DataOutputStream out) throws IOException {
    int totalSize = 4 + 4 + payload.length; // Tag(4) + OpCode(4)
    out.writeInt(totalSize);
    out.writeInt(tag);
    out.writeInt(opCode);
    out.write(payload);
    out.flush();
}
```

2.2. Multiplexagem e Gestão de Conexões

Para cumprir o requisito de uma única ligação TCP para múltiplas threads, implementou-se um Demultiplexer. Cada pedido leva uma Tag gerada pelo cliente; ao receber a resposta, uma thread de leitura global encaminha o Frame para a fila correspondente àquela Tag. Isto permite que threads fiquem em espera (await) apenas pelos seus dados, permitindo que respostas de processamento rápido ultrapassem pedidos mais lentos.

```
public byte[] send(int opCode, byte[] data) throws IOException, InterruptedException {
    int tag;
    Entry e;
    lock.lock();
    try {
        // 1. Gerar Tag única (pode ser um contador global)
        tag = getNextTag();
        // 2. Criar entrada no mapa
        e = new Entry(lock.newCondition());
        waiters.put(tag, e);
    } finally {
        lock.unlock();
    }
    // 3. Enviar o frame (usando o lock de escrita da TaggedConnection)
    conn.send(tag, opCode, data);
    lock.lock();
    try {
        // 4. Esperar pela resposta (Ciclo contra Spurious Wakeups)
        while (e.frame == null && exception == null) {
            e.cond.await();
        }
        if (exception != null) throw exception;
        // 5. Limpar e retornar
        waiters.remove(tag);
        return e.frame.payload;
    } finally {
        lock.unlock();
    }
}
```

2.3. Eficiência na Filtragem de Dados

Dada a potencial dimensão das listas de eventos, utilizámos uma técnica de compressão por dicionário no retorno do comando de filtragem.

Em vez de repetir o nome do produto em cada entrada da lista, o servidor envia primeiro um mapeamento (ID para Nome). No corpo da resposta, cada evento utiliza apenas o ID (4 bytes), reduzindo drasticamente o *overhead* de rede e o tempo de processamento de strings repetidas.

3.Implementação

O núcleo do sistema foi desenhado para garantir a integridade dos dados e a eficiência no acesso, mesmo perante grandes volumes de informação e múltiplos clientes concorrentes.

3.1. Arquitetura do Servidor e Concorrência

O servidor adota uma estratégia de uma thread por cliente através da classe ServerWorker, o que permite o atendimento simultâneo de múltiplos utilizadores.

Para evitar que o sistema se torne um gargalo, implementámos uma hierarquia de locks granulares:

- **TSDB:** Utiliza um ReentrantReadWriteLock para controlar o acesso ao histórico global e à lista de utilizadores.
- **SerieDia:** Cada série diária possui o seu próprio lock de leitura/escrita, permitindo que consultas estatísticas (leituras) em diferentes dias ocorram em paralelo, sem interferirem umas com as outras.

3.2. Notificações e Condições de Bloqueio

Para minimizar o desperdício de ciclos de CPU (requisito de evitar threads acordadas desnecessariamente), a classe Notificador utiliza um mapeamento de Condition por produto.

- **Funcionamento:** Uma thread que aguarda "Produto A" fica suspensa apenas na variável de condição associada a esse nome.
- **Garantia de Terminação:** No encerramento do dia, o sistema sinaliza todas as condições para evitar que threads fiquem retidas indefinidamente se as condições de venda não forem atingidas.

```
public void esperarSimultaneo(String p1, String p2, Map<String, List<Evento>> dia) throws Exception {
    lock.lock();
    try {
        Condition cond = lock.newCondition();
        registarInteresse(p1, cond);
        registarInteresse(p2, cond);

        while (!diaTerminou && (!dia.containsKey(p1) || !dia.containsKey(p2))) {
            cond.await();
        }
        removerInteresse(p1, cond);
        removerInteresse(p2, cond);
        // Se o loop parou mas um dos produtos não existe, o dia acabou antes
        if (!dia.containsKey(p1) || !dia.containsKey(p2)) {
            throw new Exception("O dia terminou sem que a venda ocorresse.");
        }
    } finally {
        lock.unlock();
    }
}
```

3.3. Gestão de Séries Temporais (S vs D)

Implementámos um mecanismo de swapping para garantir que a RAM não excede o limite S.

- **Controlo de Memória:** A TSDB rastreia as séries em memória. Quando uma nova série é carregada e o limite S é atingido, a série mais antiga (FIFO) é descarregada para um ficheiro binário.
- **Agregação Lazy e Caching:** O cálculo de métricas é feito *on-demand*. Os resultados são armazenados num objeto Stats que permanece em cache na RAM mesmo após os eventos brutos serem movidos para disco. Isto garante que consultas repetidas ao histórico sejam quase instantâneas.

```java
private void garantirSerieNaMemoria(int diaID) {
    histLock.writeLock().lock();
    try {
        SerieDia serie = historico.get(diaID);
        if (serie == null) return;
        if (serie.estaEmMemoria()) {
            diasEmMemoria.remove((Integer) diaID);
            diasEmMemoria.add(diaID);
            return;
        }
        if (diasEmMemoria.size() >= S) {
            int diaParaRemover = diasEmMemoria.remove( index: 0);

            SerieDia antiga = historico.get(diaParaRemover);
            if (antiga != null) antiga.descarregarEventos();
        }
        try {
            serie.carregarDoDisco();
            diasEmMemoria.add(diaID);
        } catch (IOException e) {
            e.printStackTrace();
        }
    } finally {
        histLock.writeLock().unlock();
    }
}
```

3.4. Abstração na Biblioteca de Acesso (API)

A interação com o servidor é totalmente mediada por uma biblioteca de acesso (ClientAPI), que encapsula a gestão de sockets e a lógica do Demultiplexer. Esta camada de abstração permite que o utilizador final interaja com o serviço através de chamadas de métodos simples (ex: addEvent), sem necessidade de manipular bytes ou gerir tags manualmente.

A interface de utilizador foi desenhada como uma consola interativa que permite realizar operações de forma intuitiva, suportando a visualização de notificações em tempo real. A API garante que, mesmo que a interface realize múltiplas chamadas concorrentes, a integridade da comunicação é mantida pela multiplexagem

4. Funcionamento

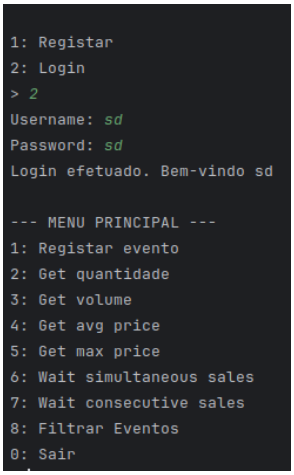
4.1 Inicializar cliente e do servidor

```
### Bem-vindo à TSDB Store ###

1: Registar
2: Login
```

```
########################################
###   SERVIDOR TSDB STORE - ATIVO     ###
###   Porto: 12345                    ###
###   Config: D=10, S=5               ###
########################################
Dica: Prime ENTER para mudar para o próximo dia.
```

4.2 Registo/login e listagem de funcionalidades



5. Avaliação de Desempenho.

Nesta parte do trabalho, focámo-nos em validar se as soluções que desenhámos — como a multiplexagem e a gestão de memória — funcionam como esperado quando o sistema é levado ao limite. Onde tentámos perceber o comportamento do servidor perante situações reais de utilização.

5.1. Escalabilidade e Multiplexagem

Testámos a capacidade do sistema em lidar com várias threads de cliente a usar a mesma ligação TCP. O objetivo era confirmar se o Demultiplexer conseguia gerir as Tags sem confundir as respostas.

> Resultado: O sistema aguentou bem a carga concorrente. Notámos que existe um pequeno custo de processamento para gerir as Conditions e o mapa de tags no cliente, mas isso é compensado pela vantagem de não termos de abrir e fechar sockets constantemente. A arquitetura permitiu que as threads não ficassem bloqueadas à espera umas das outras para usar o canal de comunicação.

5.2. Robustez

Seguindo o que era pedido no enunciado, criamos um cenário onde um cliente pede dados (através de um filtro grande), mas deixa de ler a resposta a meio.

> **Resultado:** Este teste foi util para validar a nossa estrutura de threads no servidor. Enquanto a thread ServerWorker do cliente problemático ficou ocupada a tentar gerir o buffer, os outros clientes continuaram a receber respostas sem qualquer atraso. Isto confirma que o isolamento que criámos no servidor evita que um utilizador lento ou com problemas de rede consiga interromper o serviço para os restantes.

5.3. Gestão de Memória (S vs D)

Para testar a persistência, baixámos o limite S no servidor para forçar a escrita de ficheiros em disco e a limpeza da memória RAM.

> Resultado: Ao forçarmos a mudança de dia no servidor, confirmámos que os dados foram corretamente persistidos em ficheiros .bin. Quando consultámos o histórico, o servidor foi capaz de localizar e carregar esses dados de volta para a memória de forma transparente. A organização binária que utilizámos permitiu que a recuperação de dados históricos fosse eficiente, provando que o sistema consegue gerir grandes volumes de histórico (até ao limite D) sem comprometer a estabilidade da memória RAM

6.Conclusão

O projeto permitiu a implementação de um serviço de séries temporais robusto, cumprindo integralmente os requisitos de comunicação binária e suporte multi-threaded através de uma única ligação TCP. A arquitetura assente no Demultiplexer e na gestão dinâmica de memória (S vs D) provou ser eficaz no processamento de volumes massivos de dados, garantindo que o limite de RAM não é excedido sem comprometer a persistência e integridade do histórico total.

A utilização de locks granulares por série diária e de um sistema de notificações seletivas por produto reduziu significativamente a contenção e o desperdício de ciclos de CPU, enquanto a estratégia de agregação *lazy* com cache estatístico otimizou o desempenho das consultas recorrentes.