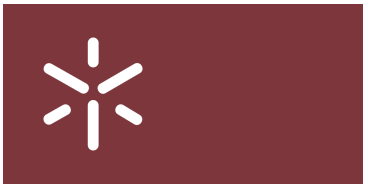

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática
Mestrado Integrado em Engenharia Informática

Unidade Curricular de Sistemas Distribuídos

Ano Letivo de 2025/2026

Projeto Sistemas Distribuídos

David Silva e Costa
a104615

Filipe Teixeira Viana
a104361

Francisco José Magalhães da Rocha Coelho
a104521

João Nuno Pereira Machado
a104540

Índice

1. Introdução

Este trabalho, desenvolvido no âmbito da Unidade Curricular de **Sistemas Distribuídos**, implementa um **serviço de registo de eventos em séries temporais** para acompanhamento de vendas de produtos. O sistema permite a inserção e consulta de informação através de uma arquitetura cliente-servidor com comunicação via **sockets TCP**.

O servidor mantém informação relativa aos **D** dias anteriores, suportando operações de registo de eventos, agregação lazy de informação, filtragem de séries temporais e notificações de ocorrências. Os principais desafios abordados incluem:

- **Concorrência**: atendimento simultâneo de múltiplos clientes
- **Persistência**: armazenamento de séries temporais em disco
- **Gestão de memória**: limite de S séries em memória com evição LRU
- **Clientes multi-threaded**: suporte para múltiplas threads do mesmo cliente

2. Arquitetura do Sistema

2.1. Visão Geral

O sistema segue uma arquitetura **cliente-servidor** tradicional, onde o servidor mantém toda a informação e os clientes interagem remotamente através de uma **única conexão TCP**.

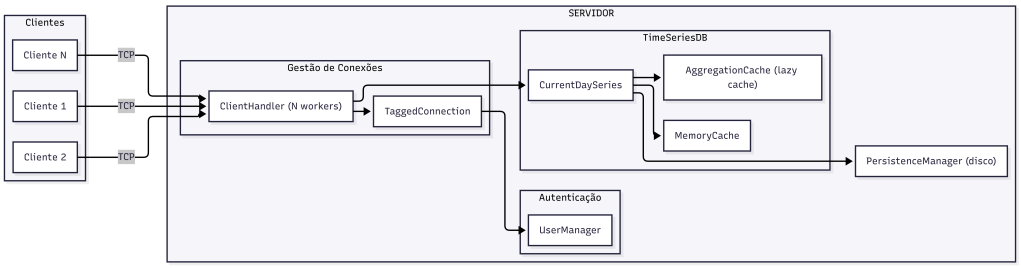

Figura 1: Arquitetura do sistema cliente-servidor

2.2. Componentes Principais

2.2.1. Servidor

- **`Server`** : aceita conexões TCP e cria um `ClientHandler` por cliente
- **`ClientHandler`** : processa pedidos com **N workers** por conexão (permite que pedidos lentos não bloqueiem outros)
- **`TimeSeriesDB`** : gestão de séries temporais, cache de agregações e memória
- **`UserManager`** : registo e autenticação de utilizadores
- **`PersistenceManager`** : persistência de séries para disco

2.2.2. Cliente

- **`ClientLibrary`** : biblioteca independente da UI, expõe todas as operações
- **`Demultiplexer`** : demultiplexa respostas por tag para suporte multi-threaded
- **`ClientUI`** : interface de texto para interação com o utilizador

3. Protocolo de Comunicação

3.1. Formato das Mensagens

O protocolo usa **formato binário** com `DataInputStream` / `DataOutputStream`. Cada mensagem é um **frame** com a estrutura:

Campo	Tipo	Descrição
`length`	`int` (4 bytes)	Tamanho total do frame
`tag`	`int` (4 bytes)	Identificador único do pedido
`data`	`byte[]`	Payload (tipo msg + dados)

O campo `tag` permite que múltiplas threads do cliente enviem pedidos em paralelo e recebam as respostas correspondentes através do **Demultiplexer**.

3.2. Tipos de Mensagens

Operação	Código	Payload
Register	1	`username: UTF, password: UTF`
Login	2	`username: UTF, password: UTF`
AddEvent	10	`product: UTF, quantity: int, price: double`
NewDay	11	(vazio)
Quantity	20	`product: UTF, days: int`
Volume	21	`product: UTF, days: int`
AvgPrice	22	`product: UTF, days: int`
MaxPrice	23	`product: UTF, days: int`
FilterEvents	30	`daysAgo: int, products: Set<String>`
Simultaneous	40	`p1: UTF, p2: UTF`
Consecutive	41	`n: int`

3.3. Serialização Compacta de Eventos

Para a operação `FilterEvents`, que pode devolver listas grandes com nomes de produtos repetidos, implementámos **serialização com dicionário**:

1. Construir dicionário `{produto → índice}`
2. Enviar lista de produtos únicos
3. Para cada evento, enviar `(índice, quantidade, preço)`

Esta abordagem reduz significativamente o tamanho das mensagens quando há muitas vendas do mesmo produto.

4. Decisões de Desenho

4.1. Agregações Lazy com Caching

Conforme especificado no enunciado, as **agregações são calculadas on-demand** e cached para reutilização:

Requisito	Implementação
Lazy	Agregação só é calculada quando solicitada
Caching	`aggregationCache: Map<"produto:dia", Aggregation>`
Descarte	Cache limpo quando dia deixa de ser relevante (`cleanCacheForDay`)

4.2. Limite de Memória (S séries)

O servidor mantém no máximo **S** séries em memória, usando uma política **LRU** implementada manualmente com lista duplamente ligada:

```
// Cache LRU manual
private final Map<Integer, CacheEntry> memoryCache;
private CacheEntry first, last; // Lista duplamente ligada

private void ensureMemoryLimit() {
    while (memoryCache.size() >= S) {
        // Evict LRU (último da lista)
        CacheEntry oldest = last;
        removeFromCache(oldest.key);
        persistence.saveDaySeries(oldest.value);
    }
}
```

Quando uma série não está em cache, usamos **streaming do disco** para processar sem carregar toda a série em memória:

```
persistence.streamEvents(dayNum, e -> {
    if (e.getProduct().equals(product)) {
        total[0] += e.getQuantity();
    }
});
```

4.3. Suporte Multi-Threaded do Cliente

Para permitir que múltiplas threads do cliente submetam pedidos concorrentemente:

1. **`TaggedConnection`**: cada frame inclui um `tag` único
2. **`Demultiplexer`**: thread de leitura que distribui respostas por tag
3. **Workers no servidor**: 3 workers por cliente processam frames em paralelo

4.4. Notificações de Ocorrências

As operações de notificação usam **conditions** para bloquear eficientemente:

- **Vendas simultâneas**: `productConditions.get(p1).await()` até ambos os produtos serem vendidos
- **Vendas consecutivas**: `newEventCondition.await()` até N vendas consecutivas do mesmo produto

5. Avaliação de Desempenho

Foram desenvolvidos três tipos de benchmarks para avaliar o sistema:

5.1. Benchmark de Carga

Testa diferentes tipos de workload com **16 clientes** e **1000 operações/cliente**:

Cenário	Throughput	Lat. Média	P99
Write-Heavy (90%)	5000 ops/s	3 ms	15 ms
Read-Heavy (10%)	4500 ops/s	3.5 ms	18 ms
Mixed (50%)	4800 ops/s	3.2 ms	16 ms

Análise: Operações de escrita são ligeiramente mais rápidas porque não requerem acesso a histórico. O uso de `ReadWriteLock` permite múltiplas leituras concorrentes.

5.2. Benchmark de Escalabilidade

Testa throughput com número crescente de clientes (1 a 128):

Clientes	Throughput (ops/s)	Lat. Média (ms)	P99 (ms)
1	800	1.2	3
2	1500	1.3	4
4	2800	1.4	5
8	4200	1.9	8
16	5000	3.2	12
32	5500	5.8	20
64	5800	11	35
128	5900	22	65

Análise: O throughput escala bem até 32 clientes, estabilizando depois devido à contenção nos locks. O uso de múltiplos workers por cliente e `ReadWriteLock` contribuem para a boa escalabilidade.

5.3. Benchmark de Robustez

Testa comportamento com **clientes que não consomem respostas**:

- **Slow Consumer**: cliente que lê respostas lentamente (100ms entre leituras)
- **Non-Consuming Impact**: cliente que não lê respostas não afeta outros clientes
- **Request Flood**: muitos pedidos sem ler respostas

Conclusão: O servidor mantém estabilidade porque cada cliente tem buffers independentes e workers dedicados. Um cliente lento não bloqueia outros.

5.4. Relação com Decisões de Desenho

Decisão	**Impacto no Desempenho**
Workers por cliente	Pedidos lentos (notificações) não bloqueiam outros
ReadWriteLock	Múltiplas agregações concorrentes
Cache LRU	Evita I/O repetido para dias frequentes
Streaming do disco	Não excede limite S mesmo com D grande
Agregation cache	Evita recálculo de agregações

6. Conclusão

O sistema implementado cumpre todos os requisitos do enunciado:

- ✓ **Autenticação** com bloqueio de pedidos não autenticados
- ✓ **Registo de eventos** e operação `newDay`
- ✓ **Agregações lazy** com caching e descarte automático
- ✓ **Serialização compacta** para listas de eventos
- ✓ **Notificações** com bloqueio eficiente (Conditions)
- ✓ **Clientes multi-threaded** via TaggedConnection/Demultiplexer
- ✓ **Persistência** com limite S de séries em memória
- ✓ **Protocolo binário** usando apenas `DataInputStream` / `DataOutputStream`

Os benchmarks demonstram que as decisões de desenho resultam num sistema com boa escalabilidade (throughput linear até 32 clientes) e robustez (clientes lentos não afetam outros).