Universidade do Minho
Escola de Engenharia

Departamento de Informática

Licenciatura em Engenharia Informática

Sistemas Distribuídos

Trabalho Prático

Grupo n.º 55

Carlos Daniel Lopes Cunha, nº A106910

Rui Mario da Silva Costa, nº A107316

Francisco Queirós Ribeiro da Costa Maia, nº A108962

Índice

1 Introdução

No âmbito da cadeira de Sistemas Distribuídos, o projeto proposto desafia-nos a desenvolver um sistema para o registo e consulta de vendas de produtos, organizado por séries temporais. O objetivo principal do projeto é permitir que vários utilizadores possam, em simultâneo, enviar dados de vendas para um servidor central e consultar métricas estatísticas sobre essa informação de forma eficiente.

O sistema foi construído com base numa arquitetura cliente-servidor através de sockets TCP, utilizando um protocolo de comunicação binário desenhado especificamente para este trabalho. O servidor é responsável por garantir a persistência dos dados em ficheiros, gerir uma cache em memória para acelerar as consultas e implementar um sistema de notificações para alertar os clientes sobre determinados padrões de vendas

2 Arquitetura do sistema

2.1 Modelo

A solução foi estruturada seguindo um modelo cliente-servidor. A comunicação entre as duas entidades é realizada através de sockets TCP, o que garante a entrega das mensagens por ordem e sem erros, fundamental para a integridade dos registos de vendas. O sistema foi desenhado de forma modular, separando as responsabilidades de autenticação, gestão do dia atual, persistência em disco, cálculos estatísticos e filtragem de histórico em componentes independentes, tentando assim satisfazer todos os desafios propostos no enunciado

2.2 Gestão de conexões e concorrência

No servidor, a gestão de múltiplos utilizadores é feita através da criação de threads individuais para cada ligação no *MainServer*. Dentro de cada ligação, o *ConnectionHandler* utiliza um processamento concorrente onde cada pedido do cliente é executado numa nova thread. Esta abordagem permite que o servidor continue a ler novas mensagens do socket sem ter de esperar que o processamento do pedido anterior termine, aumentando a disponibilidade do serviço para o o utilizador.

2.3 Sincronização e segurança de dados

Para evitar conflitos quando vários utilizadores tentam aceder ou modificar os mesmos dados ao mesmo tempo, utilizámos *ReentrantLock*. Estes mecanismos de exclusão mútua garantem que operações como o registo de um evento ou a mudança de dia ocorram de

forma segura. Além disso, o sistema de notificações utiliza variáveis de condição para permitir que as threads dos clientes fiquem a aguardar eficientemente por eventos específicos no *NotificationManager*, como o atingir de um volume de vendas, sem desperdício de recursos.

2.4 Persistência, cache e filtragem

O *PersistenceManager* assegura que os eventos são guardados em ficheiros binários, prevenindo a perda de dados em caso de falha do sistema. Paralelamente, o *AggregationManager* gere uma cache que armazena os resultados das séries temporais mais consultadas. Esta componente respeita um limite máximo de séries guardadas, descartando as menos utilizadas para garantir que o servidor não excede a memória RAM disponível. Por fim, a filtragem de eventos de uma série temporal é assegurada pelo *FilterManager*, que percorre os ficheiros binários de forma eficiente. Este componente permite aplicar filtros por produto ou data sem carregar a totalidade dos ficheiros para a memória, permitindo consultas complexas sobre o histórico sem sobrecarregar a memória.

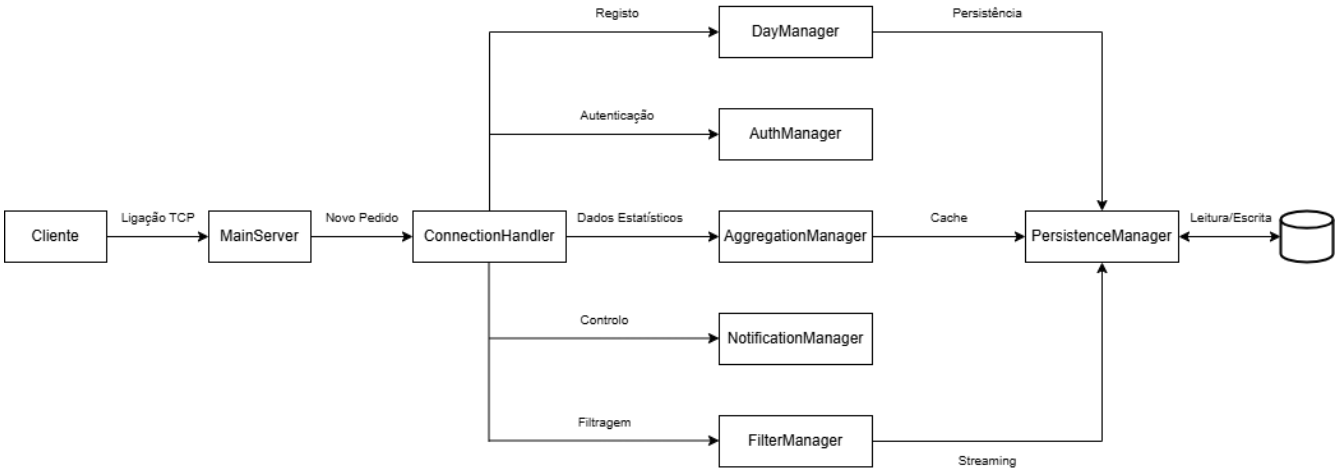

Imagem 1- Arquitetura do Sistema

3 Protocolo de comunicação

3.1 Estrutura das mensagens

Cada mensagem trocada utiliza um sistema de framing (delimentação) baseado no tamanho, onde os primeiros 4 bytes indicam o comprimento total do conteúdo que se segue. Esta estrutura é composta por:

- **Total Length (4 bytes):** Indica o tamanho do resto da mensagem
- **Request ID (4 bytes):** Um identificador único para cada pedido
- **OpCode (1 byte):** Um código numérico que identifica a operação pretendida (Login, Adicionar Evento,...).
- **Payload:**Os dados específicos de cada operação, cujo tamanho é variável.

3.2 Serialização de dados

A serialização de dados é feita utilizando as classes *DataInputStream* e *DataOutputStream*. Para evitar erros de leitura, foi criada a classe *IOUtils*, que normaliza a escrita de tipos complexos. As Strings, por exemplo, são mandadas enviando primeiro o seu comprimento em bytes e depois o conteúdo em formato UTF-8, permitindo que o recetor saiba exatamente quanto deve ler do socket. Se uma operação falhar (por exemplo, credenciais inválidas), o servidor envia uma mensagem de erro que inclui o código correspondente e uma descrição textual, facilitando a depuração e a apresentação de mensagens claras ao utilizador na interface.

Para a operação de filtragem de eventos, implementou-se um mecanismo de serialização compacta para garantir a eficiência na comunicação de séries temporais longas. Em vez de repetir campos redundantes em cada evento, o protocolo envia apenas os dados que sofreram alterações face ao evento anterior. Através de sinalizadores binários, o servidor indica se o nome do produto ou a sua descrição permanecem os mesmos, reduzindo drasticamente o volume de bytes transmitidos em resultados com muitos registos repetidos.

4 Mecanismos de sincronização e controlo

4.1 Estratégia de sincronização e exclusão mútua

A principal decisão de desenho para garantir a segurança dos dados foi a utilização de exclusão mútua através da classe *ReentrantLock*. Em vez de blocos *synchronized*, optámos

por locks explícitos em todos os gestores (Managers) do sistema, garantindo que apenas uma thread de cada vez pode modificar estruturas sensíveis, como a base de utilizadores no *AuthManager* ou o registo de eventos no *DayManager*. No *ConnectionHandler*, utilizámos um lock específico (outLock) para assegurar que as respostas enviadas para o cliente não se misturam no canal de comunicação quando existem vários pedidos pendentes e concorrentes para a mesma ligação

4.2 Variáveis de condição

Para implementar os requisitos de notificação (vendas simultâneas e consecutivas), utilizámos a interface *Condition* associada aos nossos locks. No *NotificationManager*, as threads dos clientes que aguardam por uma condição não ficam a consumir recursos do processador, em vez disso, executam um *await()* e libertam o lock, ficando suspensas até que o *DayManager* sinalize, através de um *signalAll()*, que um novo evento relevante foi registado. Para evitar o problema de despertares espontâneos e garantir que um cliente não fica bloqueado caso o dia mude, implementámos um sistema (*dayGeneration*) que valida se a espera ainda é válida após cada sinalização.

4.3 Controlo de concorrência na filtragem e cache

Para lidar com a filtragem, a sincronização garante que leituras extensas no disco não bloqueiam o sistema. O *FilterManager* utiliza mecanismos de leitura que operam de forma segura em simultâneo com a persistência de novos dados. Adicionalmente, o *AggregationManager* utiliza um lock dedicado para gerir a cache de resultados e o limite S de séries temporais em memória. Esta separação de lcoks permite que consultas ao histórico, cálculos de agregados e registos de novos eventos ocorram concorrentemente sempre que possível, maximizando o desempenho do servidor sob carga.

5 Testes implementados

Nesta secção vamos apresentar os testes que nos ajudaram a validar o código feito para o projeto. Vale realçar que a implementação dos mesmos contou com o auxílio da inteligência artificial **Gemini**, sendo que todo o código gerado foi validado, adaptado e integrado manualmente por nós.

5.1 Teste de escalabilidade

Para validar a robustez e a eficiência da solução desenvolvida, foi implementada um teste de carga (*TesteStress.java*). Este teste simula um cenário onde 50 clientes concorrentes tentam aceder ao servidor em simultâneo, realizando cada um 100 pedidos de inserção de eventos. Este teste foi fundamental para demonstrar que o servidor consegue gerir múltiplas threads em simultâneo sem a ocorrência de corrupção de dados ou impasses deadlocks, confirmando a eficácia da nossa estratégia.

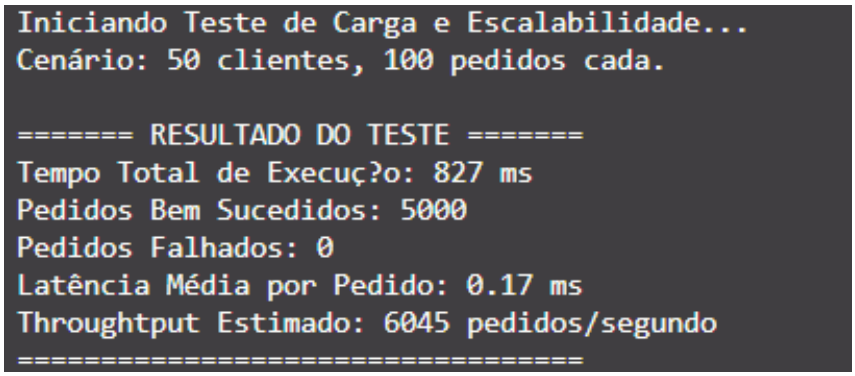

Imagem 2- Resultados do Teste de Escalabilidade

Durante a execução dos testes, o servidor processou com sucesso os 5000 pedidos num tempo total de apenas 827 ms, sem registar qualquer falha de comunicação ou processamento. A arquitetura do *ConnectionHandler* permitiu manter uma latência média reduzida, de aproximadamente 0,17 ms por pedido, atingindo um throughput estimado de 6045 pedidos por segundo. Estes resultados comprovam que a utilização de *ReentrantLocks* e a gestão eficiente do protocolo binário garantem a integridade e a rapidez das respostas. Além disso, a estabilidade demonstrada valida a resiliência do *PersistenceManager* na serialização de eventos e confirma que o sistema é capaz de lidar com picos de tráfego mantendo o consumo de recursos e o desempenho em níveis bastante bons.

5.2 Teste de robustez

Complementarmente ao teste de escalabilidade, realizámos um teste de robustez para verificar a capacidade do servidor em manter a sua disponibilidade enquanto lida com operações que o "bloqueiam". Neste cenário, um cliente "Zombi" invocou a operação de notificação de vendas simultâneas, ficando a sua thread suspensa no servidor à espera de uma condição. Simultaneamente, um segundo cliente interagiu com o sistema realizando operações de escrita. Os resultados demonstraram a eficácia do nosso modelo de concorrência, uma vez que o cliente normal obteve resposta em apenas 4 ms, confirmando que o uso de variáveis de condição e a gestão de threads por pedido evitam o bloqueio do serviço. Este teste validou que o servidor não sofre de encadeamento de bloqueios, mantendo-se totalmente funcional para novos utilizadores mesmo quando existem pedidos pendentes de longa duração.

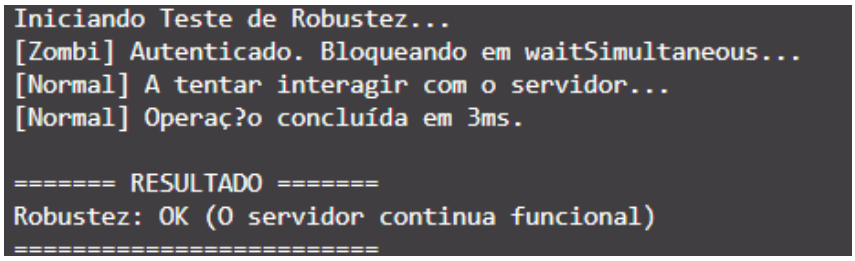

Imagem 3- Resultados do Teste de Robustez

5.3 Teste de persistência

Também foi feito um teste de persistência e agregação para validar a integridade do ciclo de armazenamento e recuperação de dados. Neste cenário, foram inseridos eventos no dia atual que, após o comando de avanço de dia, foram forçados a ser persistidos em disco pelo *PersistenceManager*. De seguida, ao solicitar uma agregação histórica de quantidades para um produto específico, confirmámos que o *AggregationManager* foi capaz de carregar o ficheiro binário do disco de forma transparente, processar a informação e devolver o resultado exato de 15 unidades. Este teste foi crucial para demonstrar que a persistência não compromete a correção dos cálculos estatísticos e que o sistema consegue transitar dados entre a memória RAM e o armazenamento persistente.

```
Iniciando Teste de Persistência e Agregaçao...
-> Inserindo eventos no dia atual...
-> Avançando o dia (Persistindo dados)...
   Dia 0 foi guardado no disco.
-> Solicitando agregaçao (Quantidade de Televisoes nos últimos 2
dias)...

======= RESULTADO =======
Persistência e Agregaçao: OK
Total detectado: 15 unidades.
=========================
```

Imagem 4- Resultados do Teste de persistência

5.4 Teste de filtragem

Por último, foi realizado um teste de *filtragem* para validar a precisão do FilterManager e a eficiência do protocolo na transmissão de subconjuntos de dados históricos. Neste cenário, foram registados múltiplos eventos relativos a diferentes produtos e, após a persistência do dia em disco, foi solicitada uma consulta filtrada por identificadores específicos. O servidor demonstrou precisão ao processar os ficheiros binários e devolver exclusivamente os registos correspondentes aos critérios de busca, ignorando os dados irrelevantes.

```
Iniciando Teste de Filtragem...
-> Dados persistidos no Dia 0
-> Solicitando filtro para 'Prod_A' e 'Prod_C'...

======= RESULTADO =======
Conteúdo recebido:
Prod_A,Prod_C



Filtragem: OK (Apenas os produtos solicitados foram retornados)
=========================
```

Imagem 5- Resultados do Teste de filtragem