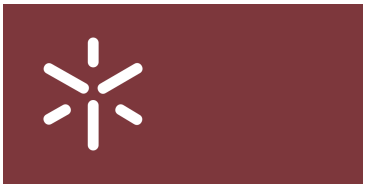

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática
Mestrado Integrado em Engenharia Informática

Unidade Curricular de Sistemas Distribuídos

Ano Letivo de 2025/2026

Trabalho Prático - Grupo 3

Gabriel Dantas
A107291

Luís Ferreira
A98286

José Fernandes
A106937

Simão Oliveira
A107322

9 janeiro 2026

Índice

1. Introdução 1

2. Arquitetura do Sistema 1

3. Funcionalidades 2

3.1. Autenticação e registo do utilizador 2

3.2. Registo de eventos 2

3.3. Agregação de informação 2

3.4. Filtrar eventos de uma série temporal 3

3.5. Notificação de ocorrências 3

3.5.1. Vendas simultâneas 3

3.5.2. Vendas consecutivas 3

3.6. Multi-Threaded 3

3.7. Persistência de séries temporais e limite destas em memória 4

4. Programa Servidor 4

5. Biblioteca do cliente 5

6. Interface do utilizador 5

7. Avaliação de desempenho 6

7.1. Testes e resultados 6

7.1.1. Cenário Favorável (20 clientes) 7

7.1.2. Cenário Médio (40 clientes) 7

7.1.3. Cenário Extremo (70 clientes) 8

7.2. Análise dos resultados 9

8. Conclusão 9

1. Introdução

Este projeto tem como objetivo a implementação e um serviço de registo de eventos em séries temporais e de agregação de informação, relativos a venda de produtos, em que a informação é mantida num servidor e acedida remotamente. A comunicação entre os servidor e os clientes é realizada através de *sockets* TCP, de forma a inserir e consultar informação. O servidor encontra-se preparado para lidar com várias conexões ao mesmo tempo de forma eficiente, garantindo acessos concorrentes.

2. Arquitetura do Sistema

A arquitetura do sistema foi construída seguindo um modelo cliente - servidor, onde o servidor é o núcleo central de armazenamento, processamento e coordenação da informação, enquanto os clientes conectam-se para efetuar pedidos de leitura e escrita, o servidor foi implementado para ser acedido de forma concorrente, tendo em atenção a múltiplos clientes simultaneamente recorrendo a mecanismos de concorrência baseados em *threads* dedicadas.

A comunicação entre o cliente e o servidor é feita através de um socket *TCP* por conexão, permitindo que os clientes efetuem diversas operações que são processadas pelo servidor, este protocolo é realizado em formato binário, onde todos os pedidos e respostas são serializados e deserializados recorrendo exclusivamente a *Data[Input|Output]Stream*. Este protocolo de comunicação foi considerado tendo em conta que o *TCP* garante uma comunicação estável e confiável entre o cliente e o servidor. De modo a abstrair a comunicação realizada sobre o *TCP*, foi desenvolvida a interface partilhada *IAmazUM*, implementada tanto no cliente como no servidor.

No lado do cliente, foi desenvolvido o *ClientUI*, que permite o utilizador testar o serviço após a sua autenticação com o servidor, uma vez autenticado, o cliente passa a ter acesso a todas as funcionalidades disponibilizadas, que por sua vez a cada pedido efetuado, o *ClientStub* encapsula toda a lógica de comunicação, sendo responsável por serializar os pedidos, enviá-los ao servidor e aguardar pelas respetivas respostas de forma transparente para o utilizador.

No servidor, cada conexão cliente é associada a uma instância de *ServerWorker*, que é responsável pela receção e validação dos pedidos, incluindo a verificação do estado de autenticação. Para operações que não envolvem autenticação ou registo, os pedidos são executados de forma concorrente através de uma *TaskPool*, cujo objetivo é gerir o trabalho das *threads*, garantindo um número máximo de *threads* ativas no sistema.

A lógica funcional dos serviços encontra-se centralizada no *ServerSkeleton*, que implementa a interface *IAmazUM* e recorre ao *ServerDatabase* para aceder à informação mantida em memória, em *cache* ou persistida em ficheiros. Esta divisão do servidor em vários componentes contribui para uma melhor separação de responsabilidades e para a simplificação da arquitetura.

Para complementar a funcionalidade do projeto, foi também desenvolvido um programa de testes que permite avaliar o comportamento do serviço sob diferentes cargas de trabalho, incluindo testes de escalabilidade, concorrência e validação de segurança, nomeadamente a correta rejeição de pedidos efetuados por clientes não autenticados.

3. Funcionalidades

O sistema oferece um conjunto de funcionalidades focadas no armazenamento de manipulação de dados de forma eficiente e segura, dando especial atenção à concorrência e ao desempenho. Todas as operações realizadas no servidor são atómicas, o que garante a integridade dos dados.

3.1. Autenticação e registo do utilizador

Para ter acesso às funcionalidades principais do servidor, os utilizadores devem primeiramente efetuar o registo ou autenticação com o servidor, apenas após uma autenticação bem sucedida que é permitido ao utilizador inserir e consultar informação.

3.2. Registo de eventos

Para adicionar um evento à serie temporal o cliente envia a informação relativa ao evento (produto, quantidade e preço), o pedido é tratado pelo *ServerWorker*, associado ao cliente autenticado, que invoca os métodos necessários para o registo do evento na base de dados do servidor. A escrita do evento é realizada pelo *ServerDatabase* de forma atómica garantindo a integridade dos dados na presença de múltiplos pedidos concorrentes.

3.3. Agregação de informação

As operações de agregação de informação dizem respeito a dias anteriores, já concluídos, sendo implementadas de forma lazy (on demand), de modo a evitar processamento desnecessário. Quando uma agregação é solicitada, pelo utilizador, pela primeira vez para um determinado produto e intervalo de dias, o servidor calcula o resultado conforme disponível, à informação existente na cache, às séries temporais mantidas em memória ou aos dados persistidos em ficheiro. O resultado obtido é posteriormente guardado na cache para reutilização em pedidos futuros equivalentes.

As operações de agregação de informação - quantidade de vendas, volume de vendas, preço médio de venda e preço máximo de venda - seguem todas o mesmo fluxo de processamento. Para qualquer uma destas operações o utilizador submete um pedido ao servidor, que é tratado pelo *ServerWorker* associado ao cliente. Este encaminha o pedido para o *ServerSkeleton*, que é responsável pelo processamento da operação.

Para o cálculo da agregação, o *ServerSkeleton* recorre ao *ServerDatabase* para obter a informação relativa aos últimos *d* dias, utilizando dados que estejam na memória, na cache ou persistidos em ficheiros. Após o processamento, o resultado é devolvido ao *ServerWorker*, que envia a resposta ao utilizador.

O acesso à informação, quer proveniente da cache, da memória ou da leitura de ficheiros, é efetuado de forma atómica, garantindo a integridade dos dados e a consistência dos pedidos concorrentes.

3.4. Filtrar eventos de uma série temporal

Como a lista de eventos a devolver ao cliente é potencialmente grande e com nomes de produtos aparecendo repetidos, foi implementado um mecanismo de serialização eficiente baseado num dicionário de produtos por utilizador. Para cada utilizador autenticado o servidor mantém uma cópia do dicionário pessoal que associa o nome de cada produto a um identificador único, sendo esta mesma estrutura mantida no cliente.

Sempre o utilizador efetua um pedido de filtragem de eventos, o servidor verifica se o cliente já possui o mapeamento necessário no seu dicionário pessoal. Caso exista, os eventos são enviados apenas utilizando os identificadores inteiros dos produtos; caso contrário, o servidor envia adicionalmente as entradas em falta do dicionário, para que deste modo o cliente atualize o seu mapeamento local.

Esta abordagem reduz significativamente o custo da serialização e de comunicação, evitando o envio repetido de nomes de produtos (*strings*) e garantindo uma representação mais compacta da lista de eventos.

3.5. Notificação de ocorrências

Para as notificações de ocorrências solicitadas pelo cliente, o servidor recorre a um componente dedicado, o *NotificationManager*, responsável pela gestão e sincronização das notificações associadas às vendas. Este componente permite que instâncias do *ServerWorker* aguardem notificações de forma segura, sem recorrer a *busy waiting*. Cada pedido de notificação resulta numa espera bloqueante sobre uma condição associada ao evento pretendido, sendo o cliente acordado no momento em que condição é satisfeita, quando o dia termina ou o quando servidor entra em estado de encerramento.

3.5.1. Vendas simultâneas

Para uma notificação de ocorrência de vendas simultâneas, o utilizador efetua o pedido para que ele seja reportado mal tenham sido vendidos dois produtos específicos, *p1* e *p2*, no dia corrente. O *ServerWorker* dedica então uma thread para notificar o cliente, caso o dia seja terminado, a thread notifica ao cliente que não foi verificada uma venda simultânea dos respetivos pedidos *p1* e *p2*

3.5.2. Vendas consecutivas

Para uma notificação de ocorrência de vendas consecutivas, o utilizador efetua o pedido para que ele seja reportado mal tenham sido efetuadas *n* vendas consecutivas de um mesmo produto no dia corrente. O *ServerWorker* dedica então uma thread para notificar o cliente, caso o dia seja terminado, a thread notifica ao cliente que não foi verificada uma venda consecutiva.

3.6. Multi-Threaded

O servidor suporta múltiplos clientes concorrentes, recorrendo a uma *TaskPool* que tem como objetivo a execução das funcionalidades do serviço, com exceção da autenticação e registo. Cada cliente autenticado é associado a um ServerWorker dedicado, o qual pode submeter vários pedidos concorrentes sem interferir com os pedidos de outros clientes.

Para evitar uma possível sobrecarga do sistema por um único cliente, o número de tarefas concorrentes que cada *ServerWorker* pode submeter é limitado. Esta abordagem permite um equilíbrio entre concorrência e controlo de recursos, embora não elimine totalmente a possibilidade de degradação de desempenho caso sejam configurados limites excessivamente elevados.

3.7. Persistência de séries temporais e limite destas em memória

Uma vez que uma cada série temporal pode ter um volume elevado de eventos, não é viável manter em memória todas as séries correspondentes aos *D* dias anteriores. Deste modo, o servidor mantém em memória apenas a série do dia corrente e, adicionalmente, no máximo *S* séries relativas a dias anteriores, sendo $S < D$ um parâmetro de inicialização do servidor. As restantes séries são persistidas no disco, organizadas por dia.

Quando um dia é terminado, a sua série temporal é serializada e armazenada em ficheiro, que pode ser posteriormente removida da memória caso o limite *S* seja atingido. Durante a execução de operações de agregação, o servidor utiliza preferencialmente dados na *cache* ou séries presentes em memória; caso a informação necessária não esteja disponível, a série correspondente é lida do disco de forma sequencial.

Quando é requisitada uma agregação que requer informação que não se encontra na *cache*, o *ServerDatabase* tenta obter os dados a partir das séries mantidas em memória, caso estes dados não se encontram na memória, a série correspondente é lida do disco. O *ServerDatabase* mantém em memória até um número máximo de *S* séries de dias anteriores, sempre que o limite é ultrapassado, o servidor descarta automaticamente o dia mais antigo. Desta forma, o processamento de agregações pode recorrer a dados persistidos sem violar as restrições de memória impostas pelo sistema.

4. Programa Servidor

O programa implementa um servidor *multi-threaded* que é responsável por aceitar conexões de clientes, processar pedidos concorrentemente e manter toda a informação necessária para suportar as funcionalidades do sistema. O servidor foi repartido em 3 componentes principais: *Server*, *ServerWorker* e *ServerSkeleton*, apoiados por uma base de dados (*ServerDatabase*) e mecanismos auxiliares de *cache* e persistência.

O *Server* é responsável pela inicialização do sistema e pela confirmação de novas conexões TCP até um limite máximo, para cada cliente autenticado, é criada uma instância de *ServerWorker* dedicado ao cliente, garantindo que cada cliente mantém apenas uma única conexão ativa com o servidor.

Cada *ServerWorker* executa numa thread dedicada e é responsável por receber, interpretar (com auxílio do *ServerSkeleton*) e responder aos pedidos do cliente associado. A comunicação é realizada através de um protocolo binário, utilizando *DataInputStream* e *DataOutputStream*. As operações potencialmente dispendiosas, como agregações e escrita em disco, são submetidas a uma *thread pool* partilhada, para evitar o bloqueio do atendimento a outros pedidos do mesmo cliente.

O *ServerSkeleton* implementa a interface *IAmazUM* e centraliza a lógica funcional do servidor, este componente coordena o acesso à base de dados, à *cache* de agregações e aos mecanismos de persistência, abstraindo estes detalhes do *ServerWorker*. As operações de agregação são im-

plementadas de forma *lazy*, recorrendo à *cache* sempre que possível e calculando os resultados apenas quando necessário.

A informação persistente e o estado do sistema são geridos pelo *ServerDatabase*, que mantém os dados do dia corrente e um número limitado de séries temporais de dias anteriores em memória através do PersistenceManager que faz a leitura e escrita em disco. O acesso aos dados é protegido por mecanismos de sincronização adequados recorrendo a bloqueios, garantindo a integridade da informação e a consistência dos resultados na presença de pedidos concorrentes.

Esta separação clara de responsabilidades permite reduzir a contenção entre threads, melhorar a escalabilidade do servidor e manter uma arquitetura modular e extensível.

5. Biblioteca do cliente

O *ClientStub* foi desenvolvido para abstrair a comunicação entre o cliente e o servidor, utilizando uma arquitetura baseada em requisições assíncronas e mensagens com *tags* únicas. Para este efeito, o *ClientStub* recorre a uma *TaggedConnection* em conjunto com um mecanismo de multiplexagem, possibilitando a coexistência de múltiplos pedidos ativos sobre a mesma conexão. A comunicação entre o cliente e o servidor é sincronizada através do uso de bloqueios, garantindo que o cliente aguarde corretamente pelas respostas do servidor sem interferir com outros pedidos concorrentes.

Além disso, o *ClientStub* expõe funcionalidades como autenticação e registo de utilizadores, registo de vendas, operações de agregação de informação, filtragem de eventos em séries temporais, notificações, finalização do dia corrente e *shutdown* do servidor. Adicionalmente, a biblioteca mantém um dicionário pessoal de produtos no lado do cliente, atualizado conforme a informação recebida pelo servidor, para que seja reduzido o custo de serialização e transmissão de strings repetidas por parte do servidor. A biblioteca foi projetada para ser eficiente e escalável, recorrendo a múltiplas *threads* para suportar vários pedidos concorrentes de forma não bloqueante. O cliente pode ainda terminar a ligação com o servidor de forma limpa, através do envio explícito de uma mensagem de desconexão, assegurando a libertação correta dos recursos no servidor.

6. Interface do utilizador

A *interface* do utilizador foi desenvolvida com o objetivo de permitir uma interação simples, intuitiva e robusta com o serviço, abstraindo os detalhes da comunicação com o servidor através da utilização da biblioteca do cliente (*ClientStub*).

O programa inicia com a conexão ao servidor e é apresentado um menu inicial que permite o utilizador efetuar o registo, autenticação ou terminar a execução do programa, após uma autenticação bem sucedida é disponibilizado um menu com acesso a todas as funcionalidades do sistema. A *interface* suporta as funcionalidades anteriormente descritas sobre o *ClientStub*. Para melhorar a experiência do utilizador, as operações potencialmente bloqueantes são executadas em threads dedicadas, para permitir que o utilizador continue a interagir com o menu enquanto os pedidos são processados de forma assíncrona.

Inclusive, o programa verifica continuamente os dados introduzidos pelo utilizador, prevenindo a introdução de valores inválidos. Além disso, a interface trata explicitamente de erros de

comunicação e exceções de rede, apresentando mensagens ao utilizador sempre que ocorrem falhas na interação com o servidor. No final de cada sessão, a interface assegura uma terminação controlada da aplicação, aguardando a conclusão das *threads* ainda em execução e enviando uma mensagem explícita de desconexão ao servidor, garantindo a libertação correta dos recursos e a estabilidade do sistema.

7. Avaliação de desempenho

Para a avaliação de desempenho do sistema, foi solicitada ao Copilot (modelo Claude Sonnet 4.5) a criação de uma classe de testes, através do seguinte prompt:

```
Cria um ambiente de testes onde sejam testadas, de forma aleatória, todas as
operações do sistema. A operação de end day deve ser consideravelmente mais rara
do que as restantes. No final, devem ser recolhidas métricas relevantes, tais como
o tempo médio de resposta, o tempo máximo de resposta, entre outras.
```

Como resultado, foi criada uma classe denominada ChaosMonkey.java. Esta classe tem como objetivo simular um cenário de carga concorrente e não determinística, gerando aleatoriamente diferentes operações do sistema para um número N de clientes, sendo este valor passado como argumento aquando da execução dos testes.

Cada cliente executa operações de forma independente, respeitando uma distribuição aleatória que garante que a operação end day ocorre com uma frequência significativamente inferior às restantes, aproximando o comportamento do sistema a um cenário realista de utilização.

Adicionalmente, cada operação executada possui 30% de probabilidade de introduzir uma pausa aleatória entre 0 e 50 ms.

7.1. Testes e resultados

Inicializamos o servidor com uma capacidade de **5 dias mantidos em memória**, uma **taskpool com 50 threads** e **35 dias armazenados em disco** (cada cliente pedirá agregações entre 1 a 30 dias anteriores) e **30 agregações em cache** e desenvolvemos 3 cenários diferentes, todos com duração de **60 segundos**:

- **Favorável** — 20 clientes
- **Normal** — 40 clientes
- **Extremo** — 70 clientes

7.1.1. Cenário Favorável (20 clientes)

Métrica	Valor
Duração Total	60,08 segundos
Total de Operações	156.198
Throughput	2.599,96 ops/s
Tempo Médio de Resposta	0,31 ms
Total de Erros	0 (0,00%)
Erros de Rede	0
Timeouts	0

Tabela 1: Métricas Gerais — Cenário Favorável

Operação	Count	Avg (ms)	Min (ms)	Max (ms)	Erros (%)
`addSale`	39.423	0,08	0,03	27,03	0,00
`getSalesQuantity`	23.641	0,33	0,03	8,92	0,00
`getSalesVolume`	23.713	0,34	0,03	12,99	0,00
`getSalesAveragePrice`	23.513	0,56	0,03	13,44	0,00
`getSalesMaxPrice`	23.633	0,34	0,03	12,83	0,00
`filterEvents`	22.100	0,37	0,03	8,18	0,00
`endDay`	175	0,89	0,46	6,17	0,00

Tabela 2: Métricas por Tipo de Operação — Cenário Favorável

Operação	Percentagem	Contagem
`addSale`	25,2%	39.423
`getSalesQuantity`	15,1%	23.641
`getSalesVolume`	15,2%	23.713
`getSalesAveragePrice`	15,1%	23.513
`getSalesMaxPrice`	15,1%	23.633
`filterEvents`	14,1%	22.100
`endDay`	0,1%	175

Tabela 3: Distribuição de Operações — Cenário Favorável

7.1.2. Cenário Médio (40 clientes)

Métrica	Valor
Duração Total	60,15 segundos
Total de Operações	312.878
Throughput	5.202,06 ops/s
Tempo Médio de Resposta	0,31 ms
Total de Erros	0 (0,00%)
Erros de Rede	0
Timeouts	0

Tabela 4: Métricas Gerais — Cenário Normal

Operação	Count	Avg (ms)	Min (ms)	Max (ms)	Erros (%)
addSale	79.067	0,07	0,02	26,51	0,00
getSalesQuantity	47.684	0,33	0,03	28,77	0,00
getSalesVolume	47.184	0,33	0,03	28,57	0,00
getSalesAveragePrice	47.138	0,57	0,03	27,53	0,00
getSalesMaxPrice	47.294	0,34	0,03	28,84	0,00
filterEvents	44.215	0,39	0,03	43,13	0,00
endDay	296	0,85	0,36	6,06	0,00

Tabela 5: Métricas por Tipo de Operação — Cenário Normal

Operação	Percentagem	Contagem
addSale	25,3%	79.067
getSalesQuantity	15,2%	47.684
getSalesVolume	15,1%	47.184
getSalesAveragePrice	15,1%	47.138
getSalesMaxPrice	15,1%	47.294
filterEvents	14,1%	44.215
endDay	0,1%	296

Tabela 6: Distribuição de Operações — Cenário Normal

7.1.3. Cenário Extremo (70 clientes)

Métrica	Valor
Duração Total	60,23 segundos
Total de Operações	513.499
Throughput	8.526,06 ops/s
Tempo Médio de Resposta	0,77 ms
Total de Erros	0 (0,00%)
Erros de Rede	0
Timeouts	0

Tabela 7: Métricas Gerais — Cenário Extremo

Operação	Count	Avg (ms)	Min (ms)	Max (ms)	Erros (%)
addSale	130.031	0,24	0,04	109,62	0,00
getSalesQuantity	77.948	0,85	0,05	86,65	0,00
getSalesVolume	77.536	0,85	0,05	86,45	0,00
getSalesAveragePrice	77.190	1,43	0,04	101,40	0,00
getSalesMaxPrice	77.843	0,86	0,04	63,32	0,00
filterEvents	72.419	0,77	0,05	99,14	0,00
endDay	532	1,33	0,52	37,57	0,00

Tabela 8: Métricas por Tipo de Operação — Cenário Extremo

Operação	Percentagem	Contagem
`addSale`	25,3%	130.031
`getSalesQuantity`	15,2%	77.948
`getSalesVolume`	15,1%	77.536
`getSalesAveragePrice`	15,0%	77.190
`getSalesMaxPrice`	15,2%	77.843
`filterEvents`	14,1%	72.419
`endDay`	0,1%	532

Tabela 9: Distribuição de Operações — Cenário Extremo

7.2. Análise dos resultados

A análise dos resultados obtidos permite concluir que o sistema apresenta um desempenho globalmente eficiente, não tendo sido detetado qualquer tipo de erro durante a execução dos testes. Em todos os cenários analisados, os tempos médios de resposta revelaram-se bastante satisfatórios.

Os valores observados nos cenários favorável e normal são muito semelhantes, o que indica que o servidor consegue lidar de forma eficaz com cargas moderadas, mantendo um desempenho estável. Este comportamento sugere que o sistema possui uma boa capacidade de gestão de pedidos concorrentes.

No cenário extremo, verifica-se um aumento significativo dos tempos de resposta, aproximadamente três vezes superior ao dos restantes cenários. Ainda assim, o sistema continua funcional e estável, o que demonstra que, apesar de a carga adicional já ser claramente percetível, o servidor mantém um comportamento aceitável sob condições de stress elevado.

Um aspeto particularmente relevante é a eficiência da operação de filtragem de eventos. Esta operação apresenta tempos de resposta significativamente inferiores quando comparada com algumas operações de agregação, evidenciando a eficácia da estrutura de dados baseada em dicionário utilizada na sua implementação.

Por motivos de análise comparativa, esta funcionalidade foi temporariamente removida, passando o servidor a enviar sempre o nome completo do produto. Nesta configuração, observou-se um aumento de aproximadamente 25% no tempo médio de resposta para o pedido de filtros, o que reforça a importância da solução adotada e o impacto positivo da utilização de estruturas de dados eficientes no desempenho global do sistema.

8. Conclusão

Consideramos que o projeto desenvolvido cumpre os requisitos propostos no enunciado do trabalho prático, assegurando uma comunicação eficiente e fiável entre clientes e servidor. A arquitetura adotada permitiu uma clara separação de responsabilidades entre os diferentes componentes do sistema, esta divisão contribuiu para uma maior modularidade, facilitando a manutenção, extensão e compreensão da solução desenvolvida.

A introdução de um dicionário pessoal por cliente para o mapeamento de identificadores de produtos revelou-se particularmente relevante para a eficiência do sistema, sobretudo na operação de filtro de eventos em séries temporais. Esta abordagem permitiu reduzir significativamente o volume de dados transmitidos, evitando o envio repetido de strings de nomes de produtos, o que seria ineficiente em cenários com grandes volumes de eventos

Os testes realizados demonstraram que o sistema se comporta corretamente sob diferentes cenários de carga e concorrência.

Em suma, o trabalho desenvolvido evidência a aplicação prática dos conceitos fundamentais de sistemas distribuídos abordados nas aulas, constituindo uma base sólida para futuras extensões e melhorias, como a possibilidade do cliente enviar vários pedidos ao mesmo tempo, tal poderia ser facilmente implementado como uma possível extensão, uma possível melhoria seria a integração o dicionário pessoal do cliente em todas as operações do sistema, isto é, não apenas à operação de filtro de eventos, permitindo que a comunicação entre o cliente e o servidor seja realizada exclusivamente com identificadores de produtos, esta abordagem poderia contribuir para uma maior eficiência e uniformização do protocolo de comunicação