



Universidade do Minho

Braga, Portugal

TRABALHO PRÁTICO - RELATÓRIO

SISTEMAS DISTRIBUÍDOS

Base de dados para séries temporais

Grupo 12

Engenharia Informática 2025/26

Equipa de Trabalho:

A106936 - Duarte Escairo Brandão Reis Silva

A106932 - Luís António Peixoto Soares

A106856 - Tiago Silva Figueiredo

A104704 - Inês Ferreira Ribeiro

9 Janeiro 2026

Índice

1. Introdução	1
2. Arquitetura	1
2.1. Modelo de Comunicação	1
2.1.1. Modelo Geral	1
2.1.2. Serialização das Mensagens	1
2.2. Arquitetura do Cliente	1
2.2.1. Threads e Estruturas Auxiliares	1
2.2.2. Ciclo de Vida da Ligação	2
2.3. Arquitetura do Servidor	3
2.3.1. Inicialização	3
2.3.2. Estruturas Auxiliares	3
2.3.2.1. Cache	3
2.3.2.2. Bases de Dados	4
2.3.2.3. BoundedBuffer	4
2.3.2.4. GestorLogins	5
2.3.2.5. GestorSeries	5
2.3.2.6. GestorNotificacoes	5
2.3.3. Threads	5
2.3.3.1. ServerWorkers	5
2.3.3.2. ServerNotifier	6
2.3.3.3. ServerSimulator	6
2.3.3.4. ServerReader	6
2.3.3.5. ServerWriter	6
3. Resultados de Testes e Observações	7
4. Decisões Técnicas e Justificações	8
4.1. Modelo de Threads no Servidor	8
4.2. Separação entre Leitura e Processamento de Mensagens	8
4.3. Envio de Respostas e Evitamento de Bloqueios	8
4.4. Buffers de Respostas por Cliente	8
5. Conclusão	9

1. Introdução

No âmbito da unidade curricular de Sistemas Distribuídos, foi proposto o desenvolvimento de um projeto cujo objetivo era a implementação de um serviço de registo de eventos em séries temporais e de agregação de informação, acessível remotamente através de um servidor.

Ao longo do desenvolvimento do projeto, pensamos, desenhamos e implementamos código, com especial foco na criação de uma solução que garantisse o correto funcionamento do serviço num ambiente concorrente. Para tal, foram também desenvolvidos testes que permitem demonstrar a capacidade de paralelização e a correta coordenação das várias funcionalidades exigidas.

Neste relatório, será descrita, de forma sucinta e objetiva a arquitetura adotada, abordando as *threads*, estruturas de dados concorrentes, modelo de comunicação cliente-servidor e os principais mecanismos de sincronização e tratamento de erros utilizados.

2. Arquitetura

2.1. Modelo de Comunicação

2.1.1. Modelo Geral

O sistema adota um modelo de comunicação cliente-servidor centralizado, onde múltiplos clientes estabelecem ligações independentes a um servidor através de sockets TCP. Cada cliente inicia a comunicação criando uma ligação única e persistente com o servidor, permitindo o envio concorrente de mensagens e a receção das respetivas respostas.

No lado do servidor, cada ligação é tratada de forma concorrente, e permite que vários clientes sejam atendidos em simultâneo. A comunicação baseia-se na troca de mensagens serializadas, que encapsulam o tipo de pedido e os dados associados. O servidor valida as mensagens recebidas e responde de acordo com o protocolo definido, incluindo o envio de mensagens de erro em caso de pedidos inválidos ou corrompidos.

2.1.2. Serialização das Mensagens

A estrutura das mensagens foi desenvolvida com o objetivo de garantir flexibilidade e extensibilidade, permitindo a introdução de novos tipos de pedidos sem comprometer o funcionamento do sistema existente. Cada mensagem é composta por:

- um inteiro - utilizado como identificador pelo Demultiplexer no cliente (ver Secção 2.2.1);
- um valor `TipoMsg` - identifica o tipo da Mensagem (autenticação, registo, notificações e agregações);
- um *array* de *bytes* - contém o conteúdo da mensagem serializado.

Seguindo as especificações definidas no enunciado, as mensagens são transmitidas através dos sockets em formato binário, sendo os seus diferentes campos escritos diretamente no fluxo de saída através dos métodos `writeInt()` e `write()`, e lidos no fluxo de entrada com recurso aos métodos `readInt()` e `readFully()`, garantindo uma comunicação eficiente.

2.2. Arquitetura do Cliente

2.2.1. Threads e Estruturas Auxiliares

A implementação do cliente teve como principal objetivo permitir o envio concorrente e não bloqueante de mensagens para o servidor. Para tal, o envio de mensagens do cliente é feito usando uma estrutura intermediária, o `Stud`. Este, disponibiliza a API de envio e receção de mensagens que é usada pela interface gráfica e pelos *scripts* de teste desenvolvidos. Para cada

mensagem enviada, é criada uma *thread* do tipo *Sender*, responsável por efetuar o envio da mensagem e aguardar a respectiva resposta. Quando a resposta chega, adiciona-a à lista de respostas do cliente.

Uma vez que múltiplas *threads* podem aceder simultaneamente ao mesmo socket, tornou-se necessário um mecanismo de coordenação que garantisse a correta associação entre pedidos e respostas. Para esse efeito, foi utilizada a classe *Demultiplexer*, desenvolvida nas aulas teórico-práticas da unidade curricular.

O *Demultiplexer* permite enviar vários pedidos sobre uma única ligação TCP, mantendo um mapa de entradas associado às *threads* ativas. Para cada mensagem enviada, é criada uma nova entrada, e quando a resposta correspondente é recebida, esta é colocada na entrada correta e a *thread* em espera é sinalizada.

Adicionalmente, o cliente implementa um mecanismo de controlo associado ao envio de notificações, garantindo que apenas uma notificação de cada tipo pode estar pendente em simultâneo. Enquanto não é recebida a resposta do servidor, a possibilidade de reenvio de uma nova notificação do mesmo tipo é bloqueado, assegurando a coerência do protocolo de comunicação.

2.2.2. Ciclo de Vida da Ligação

Para iniciar o cliente é necessário executar:

```
make cliente <HOST> <PORT>
```

Fazendo isto, o cliente tem uma ligação estabelecida com o servidor através do socket que é criado e, a partir daqui, pode enviar todo o tipo de mensagens que desejar usando a interface gráfica que, por sua vez, utiliza a API disponibilizada pelo Stud:

void	<code>send(Mensagem mensagem)</code>	Envia uma mensagem.
void	<code>sendAGREGACAO(TipoMsg tipo, String^g produto, int dias)</code>	Envia uma mensagem de agregação.
void	<code>sendEVENTO(TipoMsg tipo, String^g produto, int quantidade, double preco)</code>	Envia uma mensagem de evento.
void	<code>sendEVENTO_INVALIDO(TipoMsg tipo)</code>	Envia uma mensagem de evento com valores inválidos.
void	<code>sendFILTRAR(TipoMsg tipo, List^g<String^g> produtos, int dias)</code>	Envia uma mensagem de filtragem.
boolean	<code>sendLOGIN(TipoMsg tipo, String^g username, String^g password)</code>	Envia uma mensagem de login/registo e espera pela resposta.
void	<code>sendNotificacaoVC(TipoMsg tipo, int n)</code>	Envia uma mensagem de notificação VC.
void	<code>sendNotificacaoVS(TipoMsg tipo, String^g prod1, String^g prod2)</code>	Envia uma mensagem de notificação VS.

Figura 1: API de envio de mensagens do Stud

Quando o cliente desejar terminar a sua execução, basta navegar para o menu inicial e usar a opção “0 -> Sair”. Quando isto é feito, as *threads* *Demultiplexer* e *Senders* que tenham sido lançadas, são corretamente terminadas. Esquematizando, o funcionamento do cliente pode ser visto da seguinte maneira:

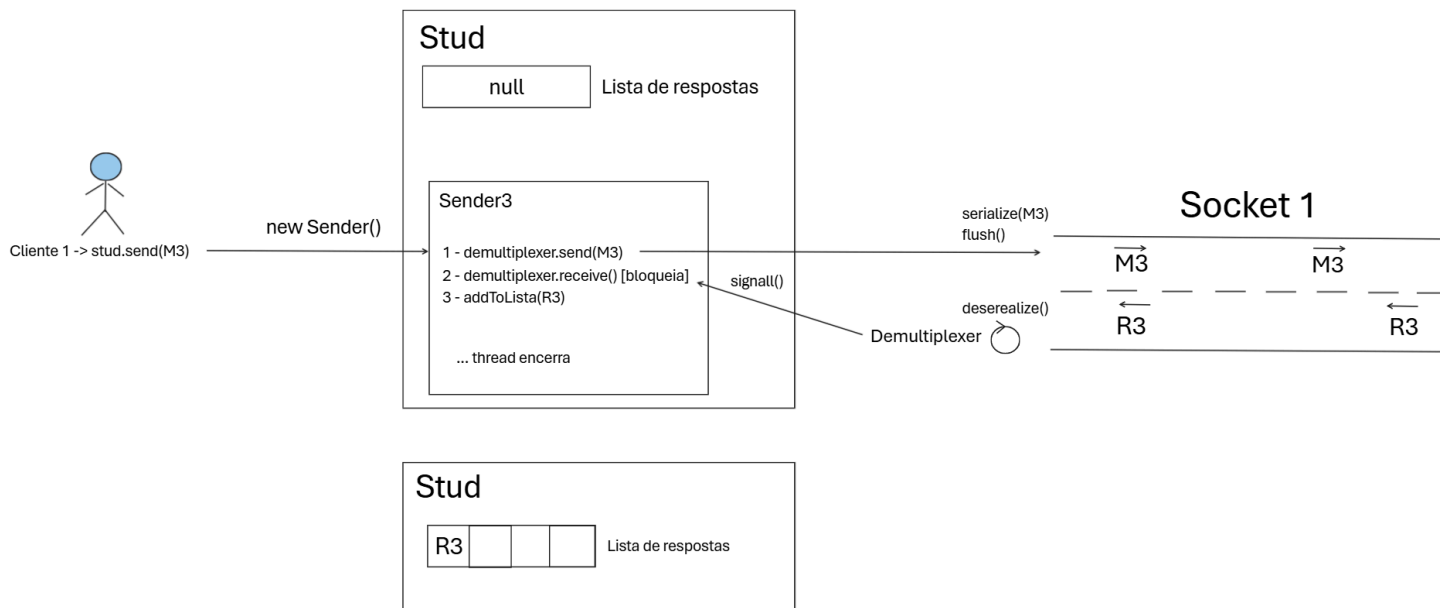


Figura 2: Esquema de envio e recepção de uma mensagem no cliente

2.3. Arquitetura do Servidor

Tendo em conta que o servidor teve de ser organizado de forma a separar responsabilidades e operações, é necessário explicar as principais funcionalidades e os momentos do ciclo de ligação com os clientes. De uma forma geral, o servidor segue uma arquitetura centralizada, utilizando uma *ThreadPool* para a execução dos pedidos e por cada ligação que é feita com um cliente, lança 2 *threads*: o *ServerReader* (ver Secção 2.3.3.4) e o *ServerWriter* (ver Secção 2.3.3.5).

2.3.1. Inicialização

make server <D> <S> <W> <I> [RESET]

- <D> – número de séries que o servidor deve contabilizar para as suas operações
- <S> - número de séries que o servidor deve manter em memória
- <W> - número de *threads* responsáveis pela execução de tarefas (tamanho da *ThreadPool*)
- <I> - tempo de intervalo entre a passagem de dias
- [RESET] - *flag* opcional que apaga todas as entradas das bases de dados (com exceção da dos utilizadores).

Depois disto, são criadas as estruturas auxiliares com os valores passados, nomeadamente: as caches de séries e agregações são criadas com tamanho <S> (ver Secção 2.3.2.1), a *ThreadPool* com <W> *ServerWorkers* (ver Secção 2.3.3.1) e o *GestorSéries* (ver Secção 2.3.2.5) com a data carregada da *BDServerDay* (ver Secção 2.3.2.2) .

2.3.2. Estruturas Auxiliares

2.3.2.1. Cache

Como o enunciado do projeto requeria a implementação de uma cache para séries de dias anteriores e outra para resultados de agregações, desenvolvemos uma estrutura genérica que permite um acesso controlado e eficiente à informação.

Tendo em conta que as operações sobre as séries de dias anteriores são as mais pesadas, mas predominantemente de leitura, optámos por uma implementação baseada em *locks* de leitura e escrita. Esta abordagem permite minimizar a contenção, garantindo que múltiplas *threads* possam consultar a cache de forma concorrente.

```
1 public class Cache<K,V>{
2     private final int capacidade;
3     private final Map<K,V> map;
4     private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
5     private final ReentrantReadWriteLock.WriteLock writelock = lock.writeLock();
6     private final ReentrantReadWriteLock.ReadLock readLock = lock.readLock();
```

Figura 3: Variáveis de instância da classe Cache

2.3.2.2. Bases de Dados

Como as caches estão em memória, tornou-se necessário garantir a persistência das séries (na BDSeries). Adicionalmente, o grupo considerou que é importante persistir também os clientes registados (na BDUsers) e o dia atual do servidor (na BDServerDay), permitindo que, ao reiniciar, o servidor retome o último dia em que se encontrava.

Tal como as caches, as bases de dados são acessadas por múltiplas *threads*, pelo que a sua implementação teve em conta a exclusão mútua e a minimização da contenção. Para tal, utilizou-se um *lock* de leitura nas operações de consulta e um *lock* de escrita nos métodos que alteram a informação persistida. É de notar que na BDServerDay, esta estratégia não foi aplicada visto que é usada por uma *thread* apenas.

```
1 public class BDSeries implements Map<String, Serie> {
2     private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
3     private final ReentrantReadWriteLock.WriteLock writelock = lock.writeLock();
4     private final ReentrantReadWriteLock.ReadLock readLock = lock.readLock();
5     private static BDSeries singleton = null;
```

Figura 4: Variáveis de instância da classe BDSeries

2.3.2.3. BoundedBuffer

Para suportar a comunicação concorrente entre as várias *threads* do sistema, foi desenvolvida uma estrutura genérica de sincronização BoundedBuffer, que permite fazer:

- **pool():Object** - Aguardar de uma forma não ativa por um objeto;
- **add(o:Object)** - Inserir um objeto de uma forma concorrente, mas controlada na fila e, posteriormente, sinalizar uma *thread* que se encontre à espera de uma inserção nessa fila.

No contexto do servidor, o `BoundedBuffer` armazena objetos do tipo `ServerData`, que encapsulam a mensagem e o identificador do cliente que a enviou.

2.3.2.4. GestorLogins

É no `GestorLogins` que os nomes e passwords dos clientes são registados e persistidos. Para isto, é usada a `BDUsers` para realizar as verificações de login ou as operações de registo de novos clientes. É relevante notar que como a `BDUsers` é *thread-safe*, esta estrutura não necessita de mecanismos adicionais de sincronização.

2.3.2.5. GestorSeries

Uma das partes da lógica de negócio do servidor encontra-se na estrutura `GestorSeries`. É nesta componente que os `ServerWorkers` executam os pedidos recebidos, de acordo com o seu tipo, nomeadamente operações de registo, agregação e filtragem, produzindo os respetivos resultados. Para isto, ele guarda:

- a Cache de Séries;
- a `BDSeries`;
- a Cache de resultados de agregações;
- a Série do dia corrente;
- a Data do dia corrente;

As características de maior relevância sobre o `GestorSeries` são:

- o uso de um *lock* nos métodos que alteram o estado do dia corrente, para garantir que: novos registos nunca são adicionados numa série nova que ainda não foi corretamente inicializada e agregações que precisem de verificar o dia atual não obtenham um dia mal inicializado.
- o uso de uma *thread* para escrever na `BDSeries` a série do dia quando o `ServerSimulator` (ver Secção 2.3.3.3) deseja passar o dia, já que se a série do dia que se está a guardar tiver milhares ou milhões de eventos, a operação de I/O poderá demorar.

2.3.2.6. GestorNotificacoes

A outra parte da lógica de negócio do servidor encontra-se no `GestorNotificacoes`. É nele em que as notificações do dia são guardadas e a cada inserção que é feita no servidor, usam-se os dados guardados para verificar se existe algum cliente que deva ser notificado. Caso se chegue ao fim de um dia e exista alguma notificação na estrutura, é criada uma *thread* que envia a resposta “null” (para as notificações de vendas consecutivas) ou “false” (para as notificações de vendas simultâneas) para os `ServerWriters` correspondentes. A API disponibilizada que é usada pelos `ServerWorkers` e pelo `ServerNotifier` é :

<code>void</code>	<code>addVC(int id, NotificacaoVC noti, int clienteID)</code>	Adiciona uma notificação do tipo Vendas Consecutivas
<code>void</code>	<code>addVS(int id, NotificacaoVS noti, int clienteID)</code>	Adiciona uma notificação do tipo Vendas Simultâneas
<code>List<ServerData></code>	<code>clear()</code>	Obtem todas as notificações pendentes e limpa as listas
<code>List<ServerData></code>	<code>processarProdutoVendido(String produto)</code>	Processa a venda de um produto

Figura 5: API do `GestorNotificacoes`

2.3.3. Threads

2.3.3.1. ServerWorkers

Os `ServerWorkers`, criados aquando da inicialização do servidor, consomem mensagens do `BoundedBuffer` de mensagens pendentes, executam-nas de acordo com a lógica de negócio definida

e encaminham a resposta produzida para o BoundedBuffer do ServerWriter correspondente ao cliente que originou o pedido.

2.3.3.2. ServerNotifier

A *thread* ServerNotifier é responsável por atualizar o estado das notificações no GestorNotificacoes sempre que ocorre um novo evento. Quando um ServerWorker processa uma mensagem do tipo registo (ou seja, o registo de um novo evento), ele sinaliza o ServerNotifier adicionando ao seu BoundedBuffer a string do produto envolvido. O ServerNotifier consome estas strings do buffer e invoca o método:

```
processarProdutoVendido(produto: String): List<ServerData>
```

do GestorNotificacoes. Este método retorna todas as notificações prontas a serem enviadas, encapsuladas como ServerData. Em seguida, o ServerNotifier distribui cada ServerData para o BoundedBuffer do ServerWriter correspondente, garantindo que todos os clientes afetados recebem a notificação corretamente.

2.3.3.3. ServerSimulator

A *thread* ServerSimulator é responsável por simular a passagem do tempo no sistema. Em intervalos regulares, invoca o método que faz avançar o dia no servidor, que inclui: a persistência da série do dia atual, a atualização da data corrente e a limpeza das notificações pendentes. Para evitar bloqueios prolongados, as operações potencialmente demoradas, como a escrita da série do dia corrente na BDSeries e envio de notificações com resultados “null” e “false”, são delegadas a *threads* auxiliares.

2.3.3.4. ServerReader

Quando uma nova ligação é estabelecida, é lançada uma *thread* ServerReader, cuja responsabilidade é ler mensagens do socket do cliente de forma bloqueante, validar a sua integridade e inseri-las no BoundedBuffer de mensagens pendentes. Caso seja detetada uma mensagem corrompida, é lançada uma MensagemCorrompidaException, sendo adicionada uma mensagem de erro ao buffer de respostas do cliente. Qualquer exceção ou término da ligação por parte do cliente, conduz ao encerramento ordenado da ligação, garantindo também a terminação do respetivo ServerWriter.

2.3.3.5. ServerWriter

Tal como o ServerReader, esta *thread* é lançada quando uma nova ligação chega ao servidor. Tem como função consumir, de forma bloqueante, mensagens do seu BoundedBuffer de respostas e escrevê-las no socket do cliente. Quando recebe uma mensagem do tipo POISON_PILL, interpreta-a como um sinal de término da ligação, que evita que esta *thread* fique bloqueada indefinidamente à espera de novas mensagens no seu BoundedBuffer. Para mensagens de resposta válidas, escreve os dados no socket e executa o respetivo **flush()**.

De uma forma esquemática, o funcionamento do servidor pode ser resumido a:

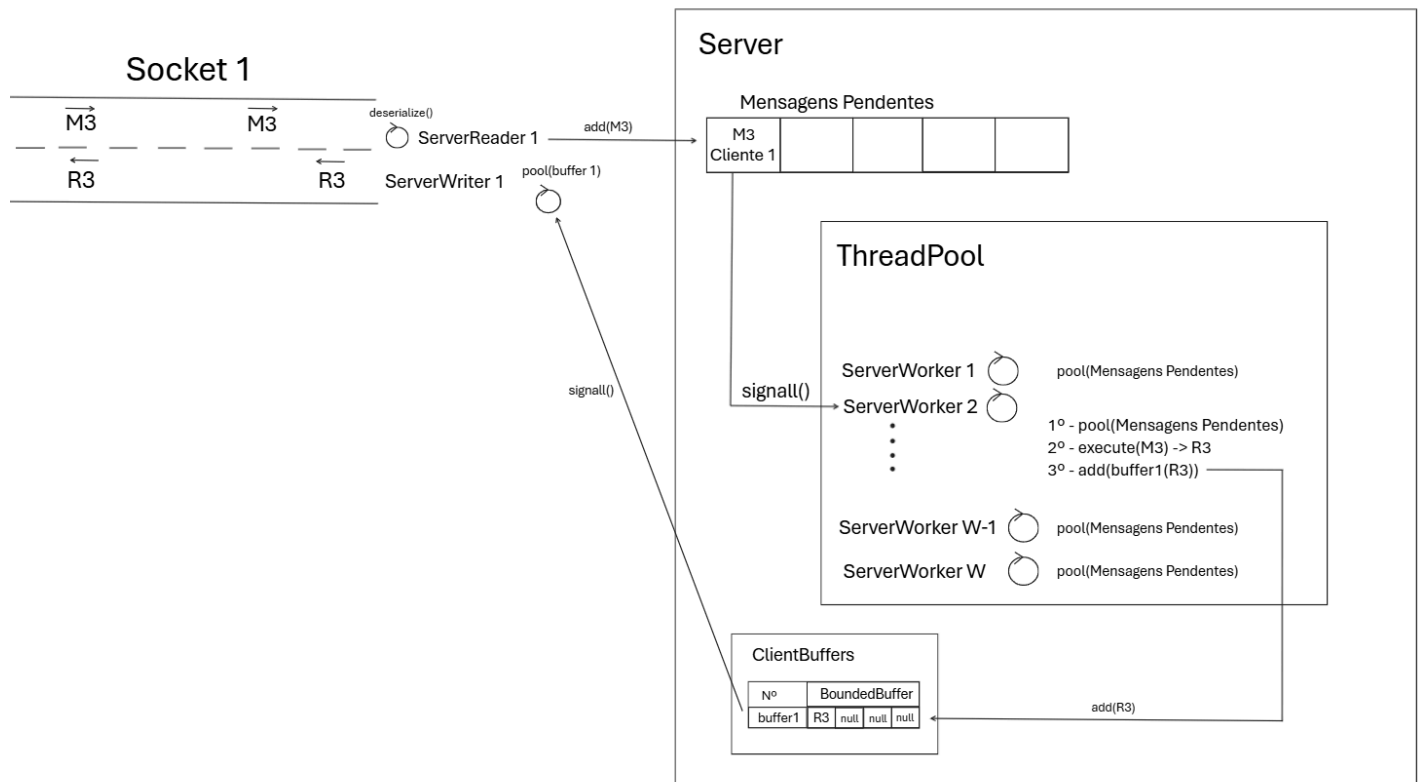


Figura 6: Esquema de recepção e envio de uma mensagem no servidor

3. Resultados de Testes e Observações

Durante a realização de testes, o grupo tomou várias observações. Por um lado, o servidor é capaz de atender vários clientes *multi-threaded* de forma eficiente e rápida, devido à arquitetura usada, já que, mesmo com um número elevado e crescente de clientes, este é capaz de realizar todas as operações das mensagens que recebe num intervalo de tempo ínfimo (na ordem dos ms). Mesmo que o servidor seja usado por um número elevado de clientes, se estes usarem o serviço de forma manual, ele deverá conseguir lidar com as mensagens facilmente.

Contudo, foi possível observar que algumas limitações quer do hardware, quer do software utilizados, tornaram algumas funcionalidades do servidor pontos de gargalo. A principal envolve as operações de I/O: operações de login e registo demoram sempre, em média, mais do que as restantes e operações de agregação cujos resultados não se encontrem na cache de agregações, demoram também bastante tempo já que têm de consultar a BDSeries.

Assim, a alteração dos valores de inicialização do servidor têm um impacto bastante grande na capacidade de resposta deste. Incrementando o número de ServerWorkers, é possível diminuir o tempo de resposta das mensagens de uma forma geral. Contudo, operações de I/O continuam a ser as mais custosas, e, por isso, deverá existir um equilíbrio no número de séries e agregações que se deseja ter em cache, e a memória que se quer usar (já que as séries poderão conter milhares ou milhões de eventos). Por isto mesmo, os parâmetros de inicialização do servidor deverão ter em conta o uso que este terá e a capacidade do hardware que o irá executar.

4. Decisões Técnicas e Justificações

Durante a realização deste projeto, o grupo teve de tomar várias decisões arquiteturais, nomeadamente sobre a forma como o servidor lida com novas ligações, como organiza a informação e quando deve lançar *threads*. Para tal, foi necessário equilibrar a capacidade de resposta do sistema com a contenção e o uso de recursos, tendo sempre em consideração as funcionalidades exigidas no enunciado.

Deste modo, as principais decisões tomadas incidiram sobre o modelo de concorrência adotado no servidor e sobre a forma como as mensagens são processadas e encaminhadas.

4.1. Modelo de Threads no Servidor

Uma das primeiras decisões a tomar prendeu-se com o número de *threads* a utilizar no servidor. Numa fase inicial, o grupo procurou minimizar o número de *threads* criadas, tendo optado por uma abordagem em que, para cada cliente que se conectava, era lançada uma única *thread* responsável por tratar, de forma sequencial, todas as mensagens enviadas por esse cliente.

4.2. Separação entre Leitura e Processamento de Mensagens

Na nova arquitetura, passou a existir uma *thread* dedicada à leitura do socket de cada cliente (ServerReader) e uma ou mais *threads* responsáveis pela execução das mensagens de acordo com a lógica de negócio (ServerWorkers da ThreadPool). Esta separação permite que a leitura das mensagens seja desacoplada do seu processamento, aumentando a capacidade de resposta do servidor.

Para permitir a partilha das mensagens lidas pelos vários ServerReaders com os ServerWorkers, foi necessária a criação de uma estrutura bloqueante para os consumidores, que minimizasse o número de *threads* acordadas aquando da inserção de novas mensagens. Para esse efeito, foi desenvolvida a estrutura BoundedBuffer, que recorre à operação **await()** no método de consumo e à operação **signal()** no método de inserção, reduzindo assim o número de *threads* desnecessariamente acordadas.

4.3. Envio de Respostas e Evitamento de Bloqueios

Ainda assim, tornou-se necessário decidir de que forma deveria ser efetuado o envio das respostas aos clientes. Caso essa responsabilidade fosse atribuída aos ServerWorkers, poderia ocorrer uma situação em que, se um cliente fosse lento a consumir as mensagens do socket — ou, no pior dos casos, deixasse de o fazer —, o ServerWorker ficaria bloqueado ao tentar realizar o **flush()** no socket. Tal situação levaria à perda dessa *thread*, possivelmente de forma permanente.

Se este cenário se repetisse para todas as *threads* do servidor, o sistema ficaria incapaz de responder a novos pedidos. Para evitar este problema, o grupo decidiu introduzir uma separação adicional, criando uma *thread* dedicada à escrita no socket de cada cliente (o ServerWriter).

4.4. Buffers de Respostas por Cliente

De forma a garantir que um número mínimo de *threads* é acordado quando um ServerWorker termina a execução de uma mensagem, optou-se por não utilizar um BoundedBuffer global de respostas partilhado por todos os ServerWriters. Em alternativa, cada ServerWriter passou a possuir o seu próprio BoundedBuffer de respostas.

Assim, sempre que uma nova ligação é estabelecida com o servidor, são criados um ServerReader e um ServerWriter, bem como um novo buffer associado a este último. A referência a esse buffer é armazenada (com recurso a um *lock*) numa estrutura do tipo Map, permitindo que, quando um ServerWorker conclui o processamento de uma mensagem, consiga encaminhar a resposta para o BoundedBuffer correspondente ao cliente correto.

5. Conclusão

O desenvolvimento deste trabalho prático permitiu implementar com sucesso um serviço de registo e agregação de eventos em séries temporais, conforme proposto no enunciado da unidade curricular. O sistema resultante suporta múltiplos clientes concorrentes, garantindo a correta associação entre pedidos, respostas e notificações, mesmo em cenários de elevada concorrência e carga.

A arquitetura adotada, baseada no modelo cliente-servidor, foi inspirada nos conceitos e estruturas abordados ao longo das aulas, tendo sido cuidadosamente adaptada às exigências específicas do problema. A separação clara de responsabilidades — nomeadamente entre leitura, processamento e escrita de mensagens — aliada ao uso de uma `ThreadPool`, `BoundedBuffers` e estruturas de cache, revelou-se fundamental para assegurar uma boa capacidade de resposta e evitar bloqueios indesejados.

Ao longo do projeto, foram tomadas diversas decisões técnicas com o objetivo de equilibrar desempenho, contenção e uso de recursos. A aplicação criteriosa de mecanismos de sincronização, como locks de leitura e escrita, permitiu garantir a integridade dos dados partilhados sem comprometer desnecessariamente a paralelização do sistema. Adicionalmente, a introdução de caches em memória contribuiu para reduzir o impacto das operações de I/O, identificadas como um dos principais fatores limitativos do desempenho global.

De forma geral, este projeto constituiu uma oportunidade relevante para consolidar conhecimentos fundamentais sobre programação concorrente e sistemas distribuídos, permitindo aplicar, de forma integrada, conceitos como comunicação por sockets, sincronização de threads, coordenação entre produtores e consumidores e desenho de arquiteturas robustas e escaláveis.