Sistemas Distribuídos - Grupo 1

26 de novembro de 2023

Daniel Pereira	**Duarte Ribeiro**	**Francisco Ferreira**	**Rui Lopes**
A100545	A100764	A100660	A100643

Introdução

Este relatório tem como objetivo apresentar o trabalho prático desenvolvido para a unidade curricular de Sistemas Distribuídos. O trabalho consiste no desenvolvimento de um sistema distribuído que permite a execução de funções remotamente. Deste modo, iremos apresentar a arquitetura do sistema, a sua implementação e as decisões tomadas pelo grupo durante o desenvolvimento do mesmo.

1. Arquitetura

Desde o início da realização do projeto, tivemos a ambição de realizar todos as funcionalidades pedidas. Assim, o sistema foi desenvolvido de forma distribuída, a partir de um servidor que reencaminha funções para vários *workers* que as executam.

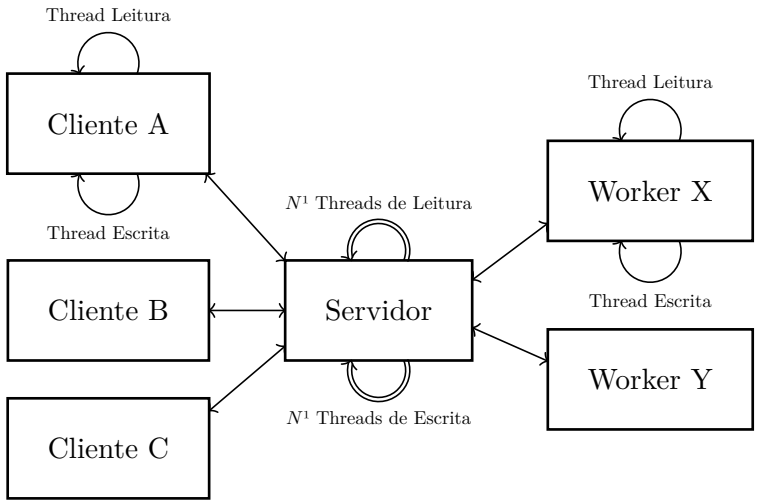

Figura 1: Visão geral da arquitetura

Para organização e separação de responsabilidades, o projeto foi desenvolvido com vários módulos[2]:

- Módulo `client-api`: implementação de lógica do cliente;
- Módulo `client`: interface do cliente sobre o módulo `client-api`;
- Módulo `server`: implementação do programa do servidor;
- Módulo `worker`: implementação do programa do worker;
- Módulo `commons`: módulo de utilitários ou classes comuns aos vários módulos.

O módulo `commons` tem presente a implementação de estruturas de dados *thread-safe*, tais como *Map*, *List* e *Bounded Buffer*, estruturas estas utilizadas extensivamente ao longo do projeto.

[1] Tantas *threads* de leitura e escrita quanto o número de clientes e *workers* conectados.

[2] A gestão (compilação e dependências) destes módulos é feita recorrendo ao Gradle.

Além disso, este módulo tem presente pacotes comuns entre as várias conexões (cliente ↔ servidor e servidor ↔ worker), bem como o protocolo de serialização explicado mais abaixo.

1.1. Uma Conexão

No módulo `commons` está presente uma classe `AbstractConnection<W, R>` que contém lógica de leitura (de pacotes do tipo R) e escrita (de pacotes do tipo W) para uma conexão (*socket*). Ao chamar o método `startReadWrite()`, uma *thread* de escrita e outra de leitura são iniciadas.

A *thread* de escrita está a ler de um *Bounded Buffer* (que espera até ser colocado lá algum pacote). Após ser colocado lá um pacote, através do método `enqueuePacket(W packet)`, essa *thread* consome-o, serializa-o para a conexão e faz *flush*. O protocolo TCP é encarregue de fazer com que o pacote chegue garantidamente e corretamente ao destinatário.

A *thread* de leitura fica à espera de receber um pacote no *socket*. Uma vez que o tamanho e tipo de dados recebidos são previsíveis, o processo de desserialização começa mal é recebido um dado e termina quando é lido o conteúdo esperado. Após a leitura completa do pacote, é chamado o método abstrato `handlePacket(R packet)` na mesma *thread*, que irá correr a lógica esperada para o *handling* daquele pacote. Cabe à implementação desse método fazer o mínimo possível ou passar o trabalho para outra *thread*.

Vale ressaltar que todas as conexões do programa (cliente → servidor, servidor → cliente, *worker* → servidor e servidor → *worker*) usam este padrão de duas *threads*.

2. Worker

Ao optar pela implementação distribuída, o programa do *worker* teve que ser desenvolvido de forma a executar *jobs* pedidos pelo servidor. Vários destes *workers* estarão em execução em simultâneo.

Inicialmente, um *worker* inicia ligando-se ao servidor na porta aberta para conexões de *workers*. Caso o servidor não esteja disponível, o *worker* fecha não fazendo tentativas de reconexão. O mesmo acontece quando o servidor fecha depois do *worker* ter feito essa conexão. Não achamos importante uma funcionalidade de reconexão, já que é esperado que o *worker* tenha sempre o mesmo ou menor tempo de vida que o servidor.

O *worker* inicia com parâmetros dois parâmetros, passados através da linha de comandos: **MAX CONCURRENT JOBS** e **MEMORY CAPACITY**. Estes parâmetros indicam, respetivamente, o **número máximo de *jobs* que podem *rodar* ao mesmo tempo**, que se traduz no número de *threads* disponíveis para a execução de *jobs*, e a capacidade de memória, como especificada no enunciado deste trabalho prático.

Ao ligar-se com o servidor, o *worker* informa-o da sua capacidade de memória e a partir daí começa a ler pedidos de *jobs*. Novamente, esta conexão segue o padrão especificado anteriormente.

Quando o *worker* recebe um pedido de *job* do servidor, insere-o num *Bounded Buffer*. Este, está a ser consumido por **MAX CONCURRENT JOBS** *threads*. Só **uma** *thread* pode receber **um** valor consumido.

Quando uma *thread* livre consome um pedido, verifica se a utilização de memória atual com a soma da memória necessária do *job* é menor do que a capacidade de memória do *worker*, caso contrário, espera até que seja maior, e então chama o *JobFunction* para os *bytes* recebidos. Se a memória necessária para executar o *job* for maior do que a capacidade total de memória do

worker, ele é simplesmente ignorado. Cabe ao servidor de não enviar *jobs* que o *worker* não consiga executar.

No fim da execução, em caso de sucesso ou erro, constrói a resposta adequada e coloca-a para ser escrita na conexão para o servidor[3].

O *worker* executa, portanto, os *jobs* recebidos pelo servidor por ordem de chegada, sem fazer reordenação e concorrentemente sempre que tiver memória disponível para tal[4]. Cabe ao servidor de fazer o escalonamento correto. O algoritmo de escalonamento usado pelo servidor será explicado num capítulo posterior.

3. Cliente

O programa do cliente foi feito baseado numa *CLI (command-line interface)* que, enquanto está a ser executado, lê comandos a partir do *input* do utilizador e executa-os.

```
[INFO] Type 'help' to see all available commands.
> help
[INFO] Available commands:
[INFO]  - benchmark <numberXmemory>...
[INFO]  - connect <host> <port>
[INFO]  - disconnect
[INFO]  - exit
[INFO]  - help
[INFO]  - job <file> <memory> (output path)
[INFO]  - jobs
[INFO]  - status
```

Como a interface terá mensagens a serem escritas enquanto o utilizador está a escrever no input (e.g. receber um resultado de um *job*), foi usada a biblioteca *JLine*[5] para resolver o problema do *standard output* escrever à frente do input do utilizador.

3.1. Comando

A interface do utilizador é baseada em comandos no terminal (dentro da aplicação), onde cada um executa uma determinada tarefa. Cada comando tem também um nome (a forma como são chamados) e um guia de utilização. Os comandos seguem o formato `comando argumentos...`, onde cada argumento pode ser obrigatório `<arg>` ou opcional `(arg)`.

3.2. Autenticação

O utilizador do programa pode-se conectar ao servidor através do comando `connect <host> <port>`. A maioria dos comandos não podem ser executados até que o cliente se autentique com o servidor.

```
> status
[ERROR] Not connected to server
```

[3]Entende-se com isto adicionar a resposta no *Bounded Buffer* da *thread* de escrita da conexão.

[4]Também existe o caso limite de existirem sempre *threads* livres, mas assume-se que a pessoa que configura o *worker* escolhe um número adequado de MAX CONCURRENT JOBS para o número de MEMORY CAPACITY também fornecido.

[5]https://github.com/jline/jline3

Após o programa abrir um *socket* para o servidor, pede ao utilizador o *username* e *password*, com o fim de se autenticar no servidor.

```
> connect localhost 8080
[INFO] Connected to server at 127.0.0.1:8080
Insert your username: meu_utilizador
Insert your password: ****
[INFO] Successfully authenticated as meu_utilizador (REGISTERED)
```

Após escrita dos campos, esses são serializados no pacote de autenticação e enviados para o servidor. O servidor responde então com o resultado da autenticação. Caso o resultado tenha sido positivo, as *threads* de escrita e leitura da conexão são iniciadas, novamente, com o padrão de conexão especificado anteriormente. A partir desse momento, o utilizador tem acesso aos comandos relativos a *jobs* e *status* do servidor. Em caso contrário, a conexão com o servidor é fechada.

3.3. Pedidos ao servidor

A partir do momento que o utilizador se autentica no servidor, já lhe pode enviar pacotes.

```
> job job.txt 50
[INFO] Sent job with id 0 with 75 bytes of data
```

```
> status
[INFO] Sent server status request. Response will be printed when received...
[INFO] Server status (127.0.0.1:8080):
[INFO]  - Connected workers: 3
[INFO]  - Total capacity: 250MB
[INFO]  - Max possible memory: 100MB
[INFO]  - Memory usage: 0%
[INFO]  - Jobs running: 0
```

O comando `job <ficheiro> <memória> (output)` envia um pedido de execução de *job* ao servidor. Quando executado, este lê os *bytes* do ficheiro de *input*, gera um *id* como forma de identificar a que *job* uma resposta futura se refere e coloca ambos os dados, juntamente com a memória necessária, num pacote. Assume-se que o ficheiro carregado não é grande demais para não caber numa região contígua de memória. Além disso, são também guardadas informações do `System.nanoTime()` atual para cálculo de *delay* de resposta e o nome do ficheiro de *output* para futura escrita em caso de sucesso (`ClientJob` e `ClientJobManager`).

O comando `status` envia um pedido de estado ao servidor. Nada é colocado junto ao pacote. O estado recebido tem as informações globais do servidor (o mesmo para todos os clientes). Ao receber a resposta deste pedido, como o servidor envia sempre o estado mais atualizado, não precisámos de saber a que pedido a resposta se refere, já que podemos assumir que o pacote mais recente[6] é o que tem informações mais atualizadas.

Ambos os comandos bloqueiam a *thread* de input **apenas** até o pacote ser colocado no *Bounded Buffer* de escrita da conexão. A resposta é impressa no terminal quando o cliente a recebe. Isto permite ao cliente poder submeter novos pedidos enquanto não recebe resposta dos anteriores.

[6] O TCP é responsável por assegurar a ordem correta de entrega dos pacotes.

Também existe um comando `benchmark` que permite enviar vários pedidos de *jobs* (sem conteúdo) facilmente. Este comando foi usado extensivamente como auxílio no teste e validação do algoritmo de escalonamento.

```
> benchmark 50x50 40x20 30x20
[INFO] Sending 50 jobs with 50MB memory
[INFO] Sending 40 jobs with 20MB memory
[INFO] Sending 30 jobs with 20MB memory
```

3.4. Resultado do *job* e escrita do resultado em ficheiro

O resultado de um *job* recebido por um cliente pode ter vários tipos:

- **Sucesso**, com os *bytes* de resultado
- **Falha**, com um código e uma mensagem de erro
- **Sem memória**, sem informação adicional

```
> job job.txt 50
[INFO] Sent job with id 0 with 5 bytes of data
[INFO] Job 0 completed successfully with 25 bytes.
[INFO] Saved job result for job 0 to file job-0.7z
> job job.txt 50
[INFO] Sent job with id 1 with 5 bytes of data
[ERROR] Job 1 failed with error code 138: Could not compute the job.
> job job.txt 150
[INFO] Sent job with id 2 with 5 bytes of data
[INFO] Job 2 failed due to not enough memory
```

Para qualquer tipo, é mostrada uma mensagem no terminal sobre informações do mesmo.

Em caso de sucesso, o resultado (juntamente com o nome do ficheiro) é *queued* para escrita em ficheiro no `JobResultFileWorker`. Esta classe é responsável por iniciar e manter uma *thread* a ler de um *Bounded Buffer*, que, ao ser colocado lá um resultado de sucesso de um *job*, escreve-o para o ficheiro pretendido.

3.5. Listagem de *jobs* pendentes e concluídos

Foi também desenvolvido um comando para listar todos os *jobs* pendentes e recebidos:

```
> jobs
[INFO]
[INFO] JOBS (1 scheduled, 123 finished, 124 total)
[INFO]
[INFO]  Scheduled jobs (1):
[INFO]   123  job-123.7z     50MB     2s ago
[INFO]
[INFO]  Received jobs (123):
[INFO]   105  job-105.7z     20MB     7s ago  SUCCESS
[INFO]    88   job-88.7z     20MB     7s ago  SUCCESS
[INFO]    59   job-59.7z     20MB     8s ago  FAILURE
[INFO]  116 more...
[INFO]
```

Este comando usa informações locais ao cliente e não faz nenhum pedido ao servidor.

3.6. API

Todas as funcionalidades da interface do cliente foram construídas usando o módulo `client-api`.

Uma demonstração da sua utilização pode ser vista aqui:

```
Client client = Client.createNewClient();
ServerNoAuthSession noAuthSession = client.connect("localhost", 8080);
AuthenticateResult authResult = noAuthSession.login("meu_username", "minha_pass");
if (!authResult.isSuccess()) return;

ServerSession session =
    noAuthSession.createLoggedSession(logger, client, my_listener);
session.startReadWrite();

session.scheduleJob(1, new byte[1024], 50); // Send job request
session.sendServerStatusRequest(); // Send server status request
```

3.7. Limitações e problemas do cliente

Durante o desenvolvimento do trabalho notámos algumas limitações e problemas no cliente que decidimos não resolver no trabalho prático, apresentando, portanto, justificações para tal.

Um dos problemas é que o nosso `Logger` não tem qualquer tipo de controlo de concorrência para mensagens de várias linhas. Na execução de comandos como o `jobs` ou o `status` as chamadas consecutivas ao método `logger.info()` para apresentar o resultado do comando, podem-se intercalar, por exemplo, com mensagens recebidas de resultados de *jobs*.

```
[INFO]
[INFO] JOBS (1 scheduled, 123 finished, 124 total)
[INFO] Job 123 completed successfully with 25 bytes.   # <-------
[INFO]
[INFO] Saved job result for job 123 to file job-123.7z # <-------
[INFO]  Scheduled jobs (1):
[INFO]   123  job-123.7z     50MB        2s ago
[INFO]
```

Para manter a simplicidade de uso do *logger*, e por se tratar de um problema meramente visual, não foi implementado um *fix* para isto.

Outra limitação que notámos foi o facto de o cliente não ter qualquer forma de esperar por mensagens bloqueando a *thread* a meio da execução[7]. Isto poderá ser necessário numa futura expansão do projeto que necessite de bloquear o cliente até obter resposta do servidor. Pelo mesmo motivo, de manter a simplicidade da API de *handle* dos pacotes recebidos, e, também, por falta de necessidade, esta funcionalidade não foi desenvolvida.

[7] Apesar de isto acontecer, por exemplo, quando o cliente espera pelo resultado de autenticação, não acontece nunca mais após o `startReadWrite()` ser chamado.

4. Servidor

O servidor é responsável por fazer a ligação, indiretamente, entre o cliente e os *workers*. Ele inicia abrindo dois *ServerSockets*, um para receber conexões de clientes e outro para receber conexões de *workers*.

Assumimos que a única porta exposta para o exterior é a porta do *ServerSocket* dedicado a conexões de clientes, pelo que, não existe nenhuma validação de autenticidade dos *workers*.

Quando o servidor recebe alguma ligação em qualquer das duas portas, o processo de autenticação (clientes) ou *handshake* (*workers*) começa. Esse processo acontece numa *thread* separada para evitar possíveis clientes ou *workers* lentos.

4.1. Autenticação/*Handshake*

Quando um cliente se conecta, o servidor espera que receba os parâmetros de autenticação do utilizador. Quando recebe, verifica se já existe um utilizador com aquele nome. Se sim, verifica se a password recebida corresponde à que tem guardada, se não, regista o utilizador com essa password. Em caso de sucesso, iniciam-se as *threads* de escrita e leitura da conexão, novamente com o padrão de conexão especificado anteriormente. Em caso de insucesso, o *socket* para o cliente é fechado. Em qualquer um dos casos, o servidor envia o resultado de autenticação (registado, logado ou password errada) para o cliente.

Por este programa ser desenvolvido apenas para fins educacionais, as passwords estão guardadas em memória sem qualquer tipo de *hashing*. Sendo também enviadas através da rede sem qualquer tipo de encriptação.

Os *workers* seguem um processo similar, mas não têm qualquer autenticação. Quando um *worker se conecta*, o servidor espera (também noutra *thread*) por um *handshake*. Este pacote tem as informações de capacidade de memória máxima do *worker*. Após recebê-lo, o processo é o mesmo, começando as *threads* de leitura e escrita.

4.2. Redirecionamento de *jobs*

Como os identificadores (IDs) dos *jobs* dos clientes não são únicos (vários clientes podem enviar um *job* com o mesmo identificador), esse identificador é mapeado para um identificador interno único ao servidor, antes do *job* ser escalonado para os *workers*. Isto também traz o benefício de anonimato entre clientes e *workers*. Quando o servidor recebe a resposta de um *worker*, o identificador é mapeado de volta para o identificador original que o cliente enviou. Neste mapeamento também é guardado o cliente que enviou o *job*, para identificação posterior quando o servidor receber o resultado do *worker*.

4.3. Algoritmo de escalonamento

A funcionalidade do escalonamento para os *workers* foi a parte que teve mais discussão e planeamento deste projeto. No planeamento partiu-se dos seguintes pressupostos:

- Assume-se que os workers não são lentos e têm o mesmo *hardware* (o tempo médio de execução de um *job* é igual em todos);
- Um *worker* executa os *jobs* por ordem de chegada, sem reordenamento e assim que tenha memória;
- O servidor consegue saber a memória disponível atual de um *worker* subtraindo a capacidade total de memória do *worker* com a soma de todas as memórias dos *jobs* pendentes.

Quando o servidor recebe um *job* para ser escalonado, invoca a função `scheduleJob`.

O funcionamento dessa função define o que o servidor deve fazer quando recebe um *job* para escalonar. O algoritmo segue a seguinte lógica:

1. Ao receber um *job* tenta encontrar um *worker* com memória disponível atual para o *job*;
2. Caso não encontre:
   1. Se o *job* não foi anteriormente colocado como pendente:
      1. Coloca-o numa lista de pendentes com o valor de ultrapassagens a 0;
      2. Termina.
   2. Se não e se o número de ultrapassagens que o *job* sofreu for maior que um valor estipulado:
      1. Encontra o *worker* com mais memória livre atualmente que consiga executar o *job*[8] futuramente;
3. Reencaminha o pedido para o *worker* escolhido (em 1. ou em 2.2.1.);
4. Adiciona 1 ao número de ultrapassagens de todos os *jobs* que foram escalonados antes deste.

Esta função também é executada sempre que o estado de memória de *workers* atualiza, ou seja:
- Quando um *worker* termina um *job*.
- Quando um *worker* se desconecta.

Isto permite ao servidor escalonar o máximo de *jobs* disponíveis, com ultrapassagens possíveis, e sem que pedidos fiquem para trás por um tempo indeterminado. O número máximo de ultrapassagens é um valor mágico, mas no nosso trabalho definimos como 50, isto é, caso nunca encontre um *worker* com memória disponível para tal, só podem ultrapassar 50 outros *jobs* para que este seja reencaminhado para um *worker* mesmo que esse não tenha memória disponível atualmente.

4.3.1. Algumas considerações adicionais

O passo 2.2.1 do algoritmo assume que o *worker* com mais memória livre será o *worker* que mais rapidamente terá memória disponível para o executar. Isto apesar de ser uma aproximação boa, não é sempre ideal, visto que outro *worker* mais cheio pode estar a completar um *job* com muita utilização de memória, e então, ao completá-la poderá ficar, mais rapidamente, esse livre. Uma escolha de *worker* que permita prever o futuro com mais precisão melhorará o algoritmo de escalonamento atual.

Também existe o problema de quando um *worker* se desconecta, alguns *jobs* podem ter mais memória necessária do que os *workers* restantes têm de capacidade, o que os torna impossíveis de serem executados. Nesse caso, o passo 2.2.1. do algoritmo não encontrará *workers* pelo que enviará "sem memória" como resultado para o *job* de volta para o cliente.

Para além disso, quando um *worker* se desconecta, todos os *jobs* previamente escalonados para ele que ainda estão por terminar, são novamente escalonados.

[8] Um *worker* consegue executar um *job* caso tenha capacidade de memória maior ou igual do que o *job* requer.

4.4. Resultado de um *job*

Quando um *worker* envia um resultado de um *job* de volta para o servidor, ou o algoritmo de escalonamento definiu que não existem *workers* com capacidade de memória para executá-lo, o mapeamento do identificador interno é trocado para o identificador enviado pelo cliente (como dito anteriormente) e a resposta (sendo ela sucesso, insucesso ou sem memória) é colocada para escrita na *thread* de escrita para o cliente certo[9]. Caso o cliente tenha se desconectado durante a execução do *job*, o resultado é simplesmente ignorado. Com isto, o ciclo de vida de um *job* acaba.

4.5. Estado do servidor

Um cliente pode pedir pelo estado atual do servidor. Quando o cliente o pede, o servidor reúne essas informações de todos os *workers*, faz os cálculos necessários para percentagens e afins, empacota-os no pacote de resposta, e coloca-o na *thread* de escrita do cliente. Por ser um processo leve, não é criada nenhuma outra *thread*, e então estes cálculos são feitos na *thread* de leitura referente ao cliente que enviou o pedido.

5. Protocolo (Serialização e Desserialização)

De forma a manter uma boa extensibilidade do projeto, uma biblioteca simples para serialização e desserialização de pacotes foi desenvolvida. Esta biblioteca, apelidada de **Frost**[10], contém as classes necessárias para a serialização e desserialização de pacotes, bem como interfaces para a implementação de pacotes serializáveis.

5.1. API

Um pacote `T` é serializável se houver uma interface `Serialize<T>` implementada e registada na instância de `Frost`.

```
public interface Serialize<T> {
    @NotNull T deserialize(SerializeInput input, Frost frost);
    void serialize(T object, SerializeOutput output, Frost frost);
}
```

Um exemplo de implementação desta classe pode ser vista aqui:

```
public record CSAuthPacket(String username, String password) {
    public static class Serialization implements Serialize<CSAuthPacket> {
        @Override
        public CSAuthPacket deserialize(SerializeInput input, ...) {
            String username = frost.readString(input);
            String password = frost.readString(input);
            return new CSAuthPacket(username, password);
        }
        @Override
        public void serialize(CSAuthPacket object, SerializeOutput output, ...) {
            frost.writeString(object.username(), output);
            frost.writeString(object.password(), output);
        }}}
```

[9] Como dito anteriormente, o mapeamento também guarda o cliente que enviou o *job*. Portanto, é possível identificá-lo por aí.

[10] Em referência à biblioteca Kryo, mas com um significado menos forte.

Dentro da classe `Frost`, podem ser encontrados mais métodos utilitários para escrita de primitivas e classes serializáveis.

Com esta classe implementada, utilizações da mesma podem ser feitas da seguinte forma:

```
Frost frost = new Frost();
frost.registerSerializer(CSAuthPacket.class, new CSAuthPacket.Serialization());

frost.writeSerializable(new CSAuthPacket(...), CSAuthPacket.class, output);
frost.flush(output);
CSAuthPacket read = frost.readSerializable(CSAuthPacket.class, input);
```

Mais exemplos podem ser encontrados em testes unitários ou no resto do programa.

5.2. Pacote Genérico

Como visto nos exemplos de API anteriores, o **Frost** já sabe à partida qual é o tipo de pacote para o ler corretamente da *stream* de *bytes* recebidos dos *sockets*. Contudo, não conseguimos saber à *priori*, por exemplo, qual é o tipo de pacote que um servidor recebe ao fazer leitura de pacotes de clientes[11]. Para resolver este problema, existe um pacote `GenericPacket<T>` (que também tem uma classe a implementar a interface `Serialize<T>`) que adiciona um campo de identificador de pacote (inteiro de 4 *bytes*) antes do conteúdo do pacote. Com este identificador, único para cada tipo de pacote, conseguimos identificar qual é e ler o conteúdo do pacote de acordo com o mesmo.

5.3. Descrição protocolar de cada pacote

As mensagens entre todas as entidades do programa são enviadas e lidas em formato binário *big-endian*. Faremos a descrição protolocar de cada pacote, indicando cada campo, o tipo de dados e a sua descrição. O tipo de dados será sempre um tipo de dados simples e usa o mesmo padrão de serialização/desserialização da classe disponível no Java DataInputStream/DataOuputStream.

Cliente → Servidor	**CSAuthPacket**	
Enviado quando um cliente quer se autenticar.		
Campo	Tipo de dados	Descrição
Username	String	Username do cliente a autenticar
Password	String	Password do cliente a autenticar

Servidor → Cliente	**SCAuthResult**	
Enviado pelo servidor ao cliente em resposta ao pedido de autenticação.		
Campo	Tipo de dados	Descrição
AuthenticateResult	int	0 = Logado com sucesso 1 = Password errada 2 = Registado com sucesso

[11] Pode ser um pacote de pedido de *job* ou um pacote de pedido de estado global do servidor.

Worker → Servidor	**WSHandshakePacket**	
Enviado pelo *worker* ao servidor para o servidor conhecer as suas capacidades.		
Campo	Tipo de dados	Descrição
Max Memory Capacity	int	Capacidade de memória suportada pelo *worker*

Como estes pacotes são enviados no começo da conexão, e com isto, estes pacotes são esperados, não é necessário prefixar os pacotes com identificadores como os próximos serão.

Cliente → Servidor Servidor → Worker	**JobRequest**	
Enviado pelo cliente ao servidor ou pelo servidor aos *workers* para fazer um pedido de *job*.		
Campo	Tipo de dados	Descrição
Packet Id	int	Sempre igual a 1
Job Id	int	Identificador do pacote (para ser usado na resposta)
Data	byte[][12]	Bytes do *job* a ser executado
Memory Needed	int	Memória necessária para o *job*

Servidor → Cliente Worker → Servidor	**JobResult (Em caso de sucesso)**	
Enviado pelo servidor ao cliente ou *worker* ao servidor para informar do resultado de um *job*.		
Campo	Tipo de dados	Descrição
Packet Id	int	Sempre igual a 2
Result Type	int	Sempre igual a 0
Job Id	int	Identificador do pacote (usado no pedido)
Data	byte[][12]	Bytes resultantes do *job*

[12] *Array* de *bytes* são serializadas com o seu tamanho em `int` como prefixo e o seu conteúdo de seguida. Na desserialização, esse inteiro é lido e é alocada uma *array* desse tamanho, colocando nela o conteúdo sem serem necessárias realocações.

Servidor → Cliente Worker → Servidor	**JobResult (Em caso de erro)**	
Enviado pelo servidor ao cliente ou *worker* ao servidor para informar do resultado de um *job*.		
Campo	Tipo de dados	Descrição
Packet Id	int	Sempre igual a 2
Result Type	int	Sempre igual a 1
Job Id	int	Identificador do pacote (usado no pedido)
Error Code	int	Código de erro da execução do *job*
Error Message	String	Mensagem de erro da execução do *job*

Servidor → Cliente	**JobResult (Em caso de sem memória)**	
Enviado pelo servidor ao cliente para informar que não há *workers* com a memória necessária.		
Campo	Tipo de dados	Descrição
Packet Id	int	Sempre igual a 2
Result Type	int	Sempre igual a 2

Cliente → Servidor	**CSServerStatusRequestPacket**	
Enviado pelo cliente ao servidor no pedido de estado global do servidor.		
Campo	Tipo de dados	Descrição
Packet Id	int	Sempre igual a 3

Servidor → Cliente	**SCServerStatusResponsePacket**	
Enviado pelo servidor ao cliente como resposta ao pedido de estado global do servidor.		
Campo	Tipo de dados	Descrição
Packet Id	int	Sempre igual a 4
Connected Workers	int	Número de *workers* conectados ao servidor
Total Capacity	int	Soma total de memória disponível em cada um dos *workers*
Max Possible Mem.	int	Memória máxima de todos os *workers*
Memory Usage %	int	Percentagem de uso de memória nos *workers*
Jobs Currently Running	int	Número de *jobs* a serem executados neste momento

Conclusão

Para concluir, o trabalho prático foi desenvolvido com sucesso, cumprindo todos os requisitos do enunciado. Consideramos que a realização do trabalho foi uma grande oportunidade para aprender e explorar mais sobre programação concorrente e sistemas distribuídos, aplicando conceitos aprendidos dentro e fora das aulas.