
Cloud Computing

TRABALHO REALIZADO POR:

BIANCA ARAÚJO DO VALE
JOÃO MACHADO GONÇALVES
MATILDE MARIA FERREIRA DE SOUSA FERNANDES
PEDRO DANTAS DA CUNHA PEREIRA



A95835



A97321



A95319



A97396

GRUPO 58
2023/2024
SISTEMAS DISTRIBUÍDOS
UNIVERSIDADE DO MINHO

Índice

1	Introdução	2
2	Arquitetura e Funcionalidades das Classes do Sistema	2
2.1	TaggedFrame	2
2.2	Client	2
2.3	Server	3
2.4	ClientHandler	3
2.5	WorkerHandler	3
2.6	Workers	4
3	Descrição das Operações	4
3.1	Register	4
3.2	Login	5
3.3	Execute	5
3.4	Execute file	6
3.5	Status	7
3.6	Exit	7
4	Conclusão	7

Índice de Imagens

1	<i>Register</i> de um Cliente	4
2	<i>Login</i> de um Cliente	5
3	<i>Execute</i> de um comando	5
4	<i>Execute</i> de um ficheiro	6
5	Ficheiro com os resultados de um <i>execute file</i>	6
6	<i>Status</i> após um <i>execute file</i>	7
7	<i>Status</i> no fim de um <i>execute file</i>	7
8	Saída de um Cliente	7

1 Introdução

O presente relatório refere-se ao trabalho prático que foi desenvolvido no âmbito da Unidade Curricular de Sistemas Distribuídos do terceiro ano da Licenciatura em Engenharia Informática.

De modo geral, foi-nos proposta a implementação de um serviço de *Cloud Computing* com funcionalidade *Function-as-a-Service*.

Este sistema funcionará sob a forma de um par cliente-servidor em Java recorrendo a *threads* e *sockets*, em que um cliente envia o código de uma tarefa de computação a ser executado num servidor, recebendo de volta o resultado, assim que haja disponibilidade.

O serviço deverá suportar diversas funcionalidades. Começando pelas mais básicas, o mesmo deverá suportar funções como a autenticação e registo de utilizadores, pedido de execução (através do envio do código da tarefa e a indicação da quantidade de memória necessária), e consulta do estado atual de ocupação do serviço e da sua fila de espera. Passando para as funcionalidades avançadas, este serviço deverá suportar funções em que o cliente permita submeter novos pedidos sem ter recebido as respostas dos anteriores e garantir uma ordem de execução das tarefas.

Ao longo do relatório iremos explicar as decisões que tomamos com o objetivo de solucionar o problema apresentado.

2 Arquitetura e Funcionalidades das Classes do Sistema

Nesta secção, falamos da estrutura e do papel de cada classe implementada no nosso projeto. Vamos detalhar como cada classe contribui para a arquitetura geral do sistema, destacando as suas responsabilidades específicas, mecanismos de interação e contribuições para o funcionamento eficiente do sistema. Cada subsecção irá focar-se numa classe específica, descrevendo as suas funcionalidades e interações com outras partes do sistema.

2.1 TaggedFrame

A classe *TaggedFrame* é responsável por encapsular e transmitir dados através de *sockets*, garantindo a integridade e a sincronização das comunicações com o servidor. A classe utiliza um *socket* para estabelecer uma conexão de rede e encapsula a *DataInputStream* e *DataOutputStream* para facilitar a leitura e a escrita de dados através dessa conexão. Para garantir operações seguras e evitar condições de corrida, usamos dois *ReentrantLocks* distintos, um para operações de envio (*sendLock*) e outro de receção (*receiveLock*), garantindo assim que cada operação seja atômica e segura. A subclasse interna *Frame* representa a estrutura de dados usada para a troca de informações com o servidor, através dos métodos *send* e *receive* podemos enviar e receber *frames*. Estes métodos encapsulam a complexidade de trabalhar diretamente com *streams* de dados e garantem que as informações sejam transmitidas de forma correta e segura.

2.2 Client

A classe responsável por estabelecer uma conexão com o *Server* e interagir com ele por meio de comandos específicos, podendo enviar tarefas para o *Server* executar apenas após se ter registado e feito login, o resultado destas é guardado num ficheiro especificado pelo utilizador.

A execução do cliente continua até que o utilizador escolha sair.

De maneira a podermos fazer pedidos e receber respostas concorrentemente, temos duas *threads* a correr constantemente, cada uma dedicada ao envio/receção de mensagens. De maneira a conseguirmos identificar o pedido correspondente a cada resposta, é enviado

um parâmetro *tag* em cada pedido/resposta, este terá o formato "[X]", no qual X será o ID correspondente ao pedido feito. Esta *tag* será apresentada no ecrã tanto no envio como na receção dos pedidos (bem como no ficheiro de resultados).

2.3 Server

A classe *Server* implementa um servidor principal, que gere a distribuição de tarefas a ser executadas pelos diferentes trabalhadores (*workers*). É responsável por estabelecer e manter conexões de rede com clientes e *workers*, criando *threads* separadas para lidar com cada tipo de conexão, estando estas permanentemente à escuta por novas ligações. Esta abordagem permite ao servidor processar múltiplas solicitações de forma simultânea, assegurando a eficiência e estabilidade do sistema.

O *Server* integra uma subclasse *UserDatabase*, responsável pela gestão de autenticação dos utilizadores. Esta subclasse utiliza um mapa para armazenar as credenciais dos utilizadores e um mecanismo de *lock*, para garantir operações seguras e protegidas.

As tarefas recebidas dos clientes são guardadas numa fila (*queue*), para serem posteriormente distribuídas pelos *workers*. Esta *queue* segue um algoritmo de FCFS e cada tarefa é atribuída ao *worker* com mais memória disponível no momento da seleção. No caso da memória necessária para a tarefa ser superior à disponível em qualquer dos *workers*, o servidor espera até que haja algum que satisfaça este requisito. Isto é feito através de *Conditions* e *Signals* (enviados sempre que a memória de um *worker* é aumentada). Cada vez que uma tarefa é enviada a um *worker*, é também enviado um ID (que estará associado ao *TaggedFrame* do cliente que fez o pedido de execução da tarefa em questão), representado por "(X)" no ecrã. Isto é feito para que, sempre que os *workers* enviem as suas respostas, o servidor consiga associá-las aos devidos clientes. Isto não pode ser feito através da *tag* do próprio pedido, visto que vários clientes poderão (e terão) *tags* com o mesmo ID.

2.4 ClientHandler

Esta classe é instanciada para cada conexão de cliente que o servidor aceita, operando numa nova *thread* para lidar com as solicitações do cliente de maneira assíncrona e eficiente. Através do *socket* e associado a cada cliente e um *TaggedFrame* (associado a este socket), o *ClientHandler* estabelece canais de entrada e saída de dados para facilitar a comunicação.

As principais funcionalidades do *ClientHandler* incluem o processamento de comandos como registo e login de utilizadores, utilizando a subclasse *UserDatabase* para validar as credenciais. É também responsável por lidar com comandos de execução de tarefas, adicionando-as à *queue* do servidor para serem executadas pelos *workers*. No caso da memória necessária para a tarefa ser superior à do *worker* com mais memória total, esta não chega a entrar na *queue*, sendo enviada imediatamente uma resposta para informar o utilizador.

Além disso, a classe lida com o comando de status que permite aos clientes verificar o estado atual das tarefas e a memória disponível nos *workers*. Por fim, trata também do comando de saída que permite ao cliente desconectar-se de forma limpa, garantindo que todas as suas tarefas pendentes sejam removidas da *queue*.

2.5 WorkerHandler

A classe *WorkerHandler* atua como intermediária entre os *workers* e o servidor. Esta classe é instanciada para cada conexão estabelecida com um *worker* e opera continuamente para processar as mensagens enviadas pelos *workers*. Cada instância é executada

na sua própria *thread*, permitindo o processamento simultâneo de múltiplas conexões de *workers*.

A principal funcionalidade do *WorkerHandler* envolve a receção e processamento de mensagens dos *workers*. Quando um *worker* se regista no sistema, o *WorkerHandler* adiciona-o à lista de *workers* ativos no servidor e atualiza a memória máxima disponível se necessário. Este registo permite ao servidor acompanhar os recursos de cada *worker* e alocar tarefas de forma eficiente.

Além do registo, o *WorkerHandler* também gere a execução das tarefas. Quando um *worker* completa uma tarefa e envia os resultados de volta ao servidor, é o *WorkerHandler* que os redireciona para a *TaggedFrame* do cliente correspondente. Isto garante que os resultados das tarefas sejam entregues corretamente e de forma oportuna.

2.6 Workers

Cada *Worker* representa um dos *sub-servers* associados ao *Server* principal. Estes serão responsáveis por executar as tarefas que recebem do *Server* principal e enviar os resultados ao mesmo, tanto em caso de falha como em caso de sucesso.

Aquando da sua criação é necessário indicar a memória que este poderá disponibilizar e, após ser estabelecida a ligação com o *Server* principal, é guardada esta informação no mesmo.

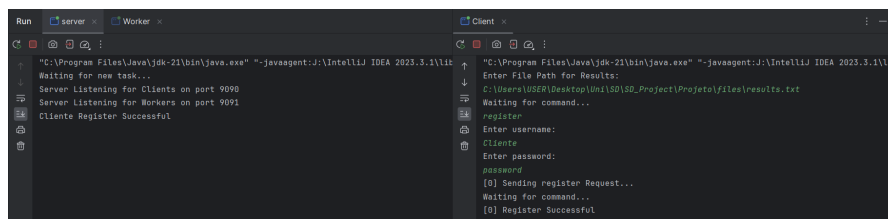
Cada *Worker* terá também uma *ThreadPool* de modo a poder executar tarefas e enviar as respetivas respostas concorrentemente, bem como fazer uma reutilização das *threads* já existentes ao invés de criá-las sempre que necessário.

Tal como qualquer outra ligação neste projeto, os *Workers* conectam-se e comunicam com *Server* principal através de sockets TCP.

3 Descrição das Operações

3.1 Register

Para realizar esta operação, o cliente deverá escrever "register" no terminal. Esta permite registar um utilizador através do nome de utilizador e da *password*.

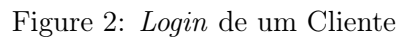


```
Run server x Worker x
C:\Program Files\Java\jdk-21\bin\java.exe" ~\javaagent-j:\IntelliJ IDEA 2023.3.1\lib
Waiting for new task...
Server Listening for Clients on port 9090
Server Listening for Workers on port 9091
Cliente Register Successful

Client x
C:\Program Files\Java\jdk-21\bin\java.exe" ~\javaagent-j:\IntelliJ IDEA 2023.3.1\lib
Enter File Path for Results:
C:\Users\USER\Desktop\IntelliJ\Project\Projeto\files\results.txt
Waiting for command...
register
Enter username:
Cliente
Enter password:
password
[0] Sending register Request...
Waiting for command...
[0] Register Successful
```

Figure 1: *Register* de um Cliente

De modo a efetuar o *login*, o utilizador deverá escrever "*login*" no terminal e em seguida deverá escrever o nome de utilizador e a respetiva *password*. É importante realçar que para efetuar as restantes operações, o cliente tem que efetuar primeiro a autenticação.



O comando "*execute*" poderá ser utilizado para executar tarefas, sendo de seguida necessário indicar o código da tarefa e a memória necessária para a mesma.



3.5 Status

De modo a consultar o estado atual da *queue* e a memória disponível em cada *worker* num dado momento, o utilizador poderá utilizar o comando "*status*".

```
status
[46] Sending Status request...
Waiting for command...
[46] Queue Size: 7
[46] Worker 0 Available Memory: 500
```

Figure 6: *Status* após um *execute file*

```
status
[47] Sending Status request...
Waiting for command...
[47] Queue Size: 0
[47] Worker 0 Available Memory: 1000
```

Figure 7: *Status* no fim de um *execute file*

3.6 Exit

Ao escrever "*exit*" no terminal, o cliente encerra todas as conexões, sendo os pedidos de execução tarefas com resposta pendente que se encontram na *queue* do servidor também removidas.

```
exit
Exiting & Aborting Tasks currently being Executed...

Process finished with exit code 0
```

Figure 8: Saída de um Cliente

4 Conclusão

Em jeito de conclusão, a realização do trabalho prático da Unidade Curricular de Sistemas Distribuídos, permitiu que o grupo conseguisse consolidar e "por em prática" os conhecimentos obtidos nas aulas teóricas e teórico-práticas.

Apesar de algumas dificuldades sentidas ao longo do projeto, consideramos que temos todas as funcionalidades que nos foram propostas implementadas. Existem também certos aspetos de menor escala em relação aos quais poderíamos ter tido mais atenção, como o uso de variáveis *volatile* ao invés de métodos com uso de *locks*, por exemplo.

Assim, finalizado o projeto, consideramos que, no geral, fomos de encontro ao resultado pretendido.