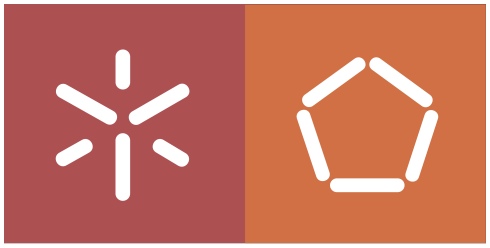
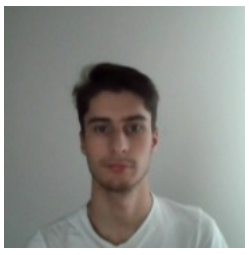


Universidade do Minho
Escola de Engenharia

Sistemas Distribuídos

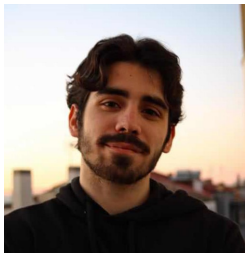
Relatório Trabalho Prático
Cloud Computing
Grupo 31

LEI - 3º Ano - 1º Semestre
Ano Letivo 2023/2024


Lucas Oliveira
A98695


Mike Pinto
A89292


Rafael Gomes
A96208


Tiago Carneiro
A93207

Braga,
28 de março de 2024

Conteúdo

Lista de Figuras

1 Introdução

Este trabalho foi realizado no âmbito da unidade curricular de Sistemas Distribuídos do curso de Engenharia Informatica, onde foi proposto a implementação de um serviço de *Cloud Computing* com funcionalidades de *Function-as-a-Service* (*FaaS*). Este trabalho foi realizado recorrendo a *Threads* e *Sockets* em *Java*.

Este serviço permite aos utilizadores enviar um código de uma tarefa de computação para ser executado num servidor remoto e obter o resultado do mesmo, onde o fator limitante para a execução de uma tarefa é a memória disponível nos servidores designados de "*Workers*". O código da tarefa de computação para ser executado é lido inicialmente de um ficheiro e enviado para um Servidor Central responsável pela gestão de todos os pedidos de tarefas recebidos e de solicitar a sua execução das tarefas em Servidores remotos denominados *"Workers"* cuja única função é executar a tarefa e enviar o resultado ao Servidor Central.

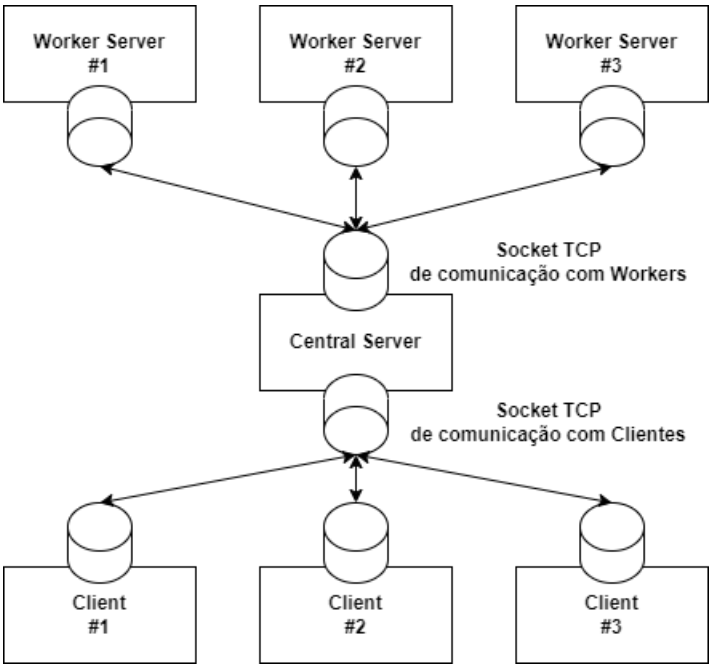


Figura 1: Esquema geral de funcionamento do sistema.

2 Arquitetura da solução

2.1 Servidor

O Servidor é responsável por iniciar os *sockets TCP* para comunicação com os Clientes e Workers, a porta utilizada no *socket* do cliente por defeito é 9090(que pode ser passada como argumento ao iniciar o programa), e para os workers da porta 8080. A seguir o servidor lê um ficheiro de configuração que possui todas as contas registadas no serviço e cria uma Thread da classe Runnable *Job Manager* que fica a aguardar por pedidos pendentes.

A cada cliente que inicia uma conexão com o servidor é criada uma Thread da classe *Client-ConnectionHandle* que é responsável responder aos pedidos desse cliente. De forma semelhante a cada Worker que inicia uma conexão é criado uma Thread da classe *WorkerConnectionHandle* responsável pela comunicação entre o Servidor e o Worker.

2.1.1 Accounts

O Servidor utiliza uma instância da classe Accounts para registar e autenticar utilizadores no serviço. Esta classe utiliza possui um Map onde a chave é um atributo do tipo "String" relativa a um *username* e o valor mapeada é um atributo do tipo "String" relativa à *password* da conta do utilizador. Para garantir que o registo e autenticação das contas é *Thread-Safe* recorremos ao uso de um **ReentrantReadWriteLock** devido a, depois de uma fase inicial do programa, haver mais operações de leitura sobre as contas do que de escrita. Esta classe também é responsável por guardar/ler de um ficheiro as contas registadas no sistema.

2.1.2 JobManager

Para garantir uma boa gestão de tarefas pendentes, em execução ou concluídas, o Servidor recorre a uma instância da classe *JobManager* responsável por toda a lógica relativa à gestão de tarefas. Esta classe possui uma **PriorityQueue** para as tarefas pendentes de execução onde ao retirar uma tarefa da fila recorre a um comparador de duas tarefas pela memória e prioridade. Um job com prioridade 3 é sempre escolhido primeiro, nos outros casos é feito a seleção com base nas tarefas de menor memória(ao comparar por memoria a prioridade de outras tarefas é aumentada, para garantir uma ordem de execução de tarefas). Para as tarefas em execução é utilizado uma Map onde a chave é o identificador de um *Worker* e o valor uma Lista de tarefas em execução nesse *Worker*. Já para as tarefas concluidas é utilizado novamente um Map onde a chave é *username* do utilizador e o valor é um objeto *CompletedUserJobs* que possui uma queue de tarefas concluídas desse utilizador. Para garantir que esta classe é *Thread-Safe*, visto que é uma classe onde operam múltiplas *Threads*, é criado um **ReentrantLock** para cada estrutura. Para garantir paralelismo entre os métodos, e que não possuímos muitas Threads acordadas ao mesmo tempo, recorremos ao uso de **Conditions** para os métodos de inserção e remoção de tarefas das estruturas utilizadas.

2.2 Cliente

O serviço de Cliente é composto pelas classes **ClientSystem**, **ClientInterface** e **Client-Controler**. Estas classes são responsáveis por todo o funcionamento da aplicação de Cliente. É iniciado uma ligação com o servidor central e após essa conexão é criada uma pasta no diretório "home" do sistema operativo onde serão inseridos o código das tarefas computacionais e os seus resultados. Por fim é apresentado um menu inicial com as opções de "registo de utilizador", "autenticação" e "sair do programa". Após a autenticação do utilizador é apresentado um menu de cliente autenticado com todas as funcionalidades do programa, "Pedido de execução de tarefas", "Estado dos pedidos", "Estado do Serviço" e "Terminar sessão e Sair". Todas as opções disponíveis no menu de cliente autenticado são apenas envio de mensagens ao servidor central e após receber a resposta apresentá-las ao utilizador. Visando possuir paralelismo no envio e receção de mensagens recorremos ao uso da classe **Demultiplexer** onde é utilizado uma Thread para ficar à escuta de respostas provenientes do Servidor Central e é criada uma *Thread* responsável por aguardar resultado de jobs e apresentar o seu resultado ao utilizador.

2.3 Worker

Ao iniciar um Worker é necessário passar como argumento do programa um valor inteiro que representa a memória total possuída. Esta classe é responsável por iniciar uma conexão com o servidor recorrendo a um *socket TCP* na porta 8080 e enviar imediatamente a informação da sua memória total. Após essa comunicação inicial, o *Worker* fica à espera de pedidos de execução de tarefas provenientes do servidor central. Ao receber um pedido de execução é criada uma Thread da classe *Runnable* **JobExecutor** que é somente responsável por executar a tarefa

recebida, recorrendo à biblioteca "sd23.jar" disponibilizada pela equipa docente, e enviar a resposta para o servidor central.

2.4 Comunicações

A comunicação entre, cliente e servidor, ou servidor cliente, como referido anteriormente é feita através do uso de *sockets TCP*. De modo a auxiliar a comunicação foram desenvolvidas as seguintes classes:

2.5 Message

Esta classe é a base entre todas as comunicações realizadas. Como variáveis de instância esta classe possui uma String **id** utilizada como identificador de uma mensagem, uma String **user** que identifica o utilizador que enviou a mensagem, um **array de bytes** com com a informação a ser transmitida(por normal Strings serializadas ou um objeto Job serializado) e um inteiro **type** que serve para identificar o tipo da mensagem, que podem ser:

- 1 - Criar Conta.
- 2 - Autenticação.
- 3 - Pedido de execução de uma tarefa.
- 4 - Pedido do estado do serviço.
- 5 - Resultado da execução de uma tarefa.
- 6 - Terminar Sessão.
- 7 - Informação sobre memória total(Utilizado apenas entre worker e servidor).
- 99 - Erro.

2.5.1 Job

Esta classe é encapsulada num objeto do tipo **Message**, cujo seu principal objetivo é o envio de informações sobre uma tarefa. Como variáveis de instância, esta classe possui um inteiro **id** que identifica qual o número da tarefa de um utilizador, uma string *user* que identifica qual o utilizador que solicitou a execução da tarefa, um inteiro **priority** que identifica a prioridade da tarefa na fila de tarefas pendentes do JobManager, um inteiro **memory** que identifica a memoria necessária para execução da tarefa, um **array de bytes** com o codigo da tarefa a ser executada ou com o resultado da sua execução e um inteiro **state** que identifica o estado do job, que podem ser:

- 0 - Execução Pendente.
- 1 - Em execução.
- 2 - Sucesso na execução.
- 3 - Erro na execução.

Esta classe implementa métodos de **serilização** e **deserilização** para um array de bytes, recorrendo ao uso de um **ByteArray[Input/Output]Stream** e um **Data[Input/Output]Stream**.

2.5.2 Connector

Esta classe implementa métodos de envio e receção de objetos do tipo **Message** recorrendo ao uso de **Data[Input/Output]Stream**. Para garantir o paralelismo no envio e receção de mensagens é utilizado dois **ReentrantLock's**, um para a escrita e outro para a leitura.

2.5.3 Demultiplexer

Esta classe encapsula um objeto do tipo **Connector**, utilizado para o envio e receção de mensagens como referido anteriormente. Esta classe também disponibiliza um método de receive onde é passado como argumento o **type** de uma mensagem. Uma thread que invoque este método fica a aguardar(utilizando um **Condition**) a receção de uma mensagem do tipo passado como argumento sendo acordada quando uma mensagem desse tipo é recebida.

Além disso, esta classe é **Runnable** possuindo um método run que fica à escuta de mensagens guardando as mensagens recebidas num Map. Esta classe é especialmente útil para o Cliente devido a este ser **multi-threaded** e de poder enviar pedido de execução de tarefas e receber resultado das mesmas paralelamente.

3 Funcionalidades

Figura 2: Menu inicial e Menu de utilizador autenticado

3.1 Registo e autenticação de um utilizador

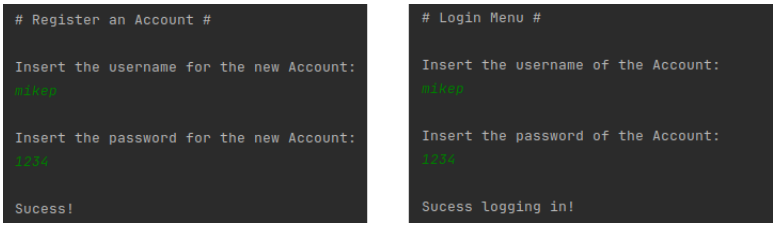


Figura 3: Caption

Ao iniciar o programa Cliente é apresentado ao utilizador o menu de registo/autenticação, neste menu o utilizador insere a opção pretendida a ser realizada, 1 - Autenticar, 2 - Registar conta e 0 - Sair do programa.

Na opção de autenticação é solicitado ao cliente o seu username e a password e enviado esta informação encapsulada num objeto **Message** com o **type** 2 ao Servidor. O servidor ao receber esta mensagem verifica se a conta se encontra registada no sistema e se as credenciais coincidem. A seguir o servidor envia uma mensagem ao Cliente com a indicação se a autenticação foi bem sucedida, encapsulando Strings num objeto **Message**, por exemplo, "Sucess", "Failed;Wrong Password", "Failed;Account not found". Se a autenticação for bem sucedida é apresentado ao utilizado o menu com todas as funcionalidades do programa.

Na opção de registo é solicitado ao cliente o username e a password a utilizar na conta. Estas informações são enviadas semelhantemente à opção de autenticação, mas desta vez com o campo **type 0**. O servidor verifica primeiramente se a conta já existe, se existir envia ao cliente essa informação, caso não exista então o servidor regista o utilizador e envia uma mensagem de sucesso ao cliente.

3.2 Pedido do estado do Serviço

```
# Service Status #

Available Memory: 600; Number of pending jobs: 0
```

Figura 4: Menu de estado de execução.

O utilizado ao selecionar a opção de estado do serviço envia uma mensagem ao Servidor com o **type 4**. Quando o servidor ao receber esta mensagem chama o método getServiceStatus() que devolve uma string com o número de tarefas pendentes e a memória disponível para ser utilizada enviando esta informação ao cliente numa Mensagem com o **type 4**.

3.3 Estado de execução de tarefas

```
# Job Status #

Job 1: Error executing job, to much memory nedded.
Job 2: Sucess executing job, received response with 139 bytes.
Job 3: Waiting execution Result.
```

Figura 5: Menu de estado de execução.

Esta funcionalidade não requer comunicação entre o cliente e o servidor, quando um cliente faz o pedido de execução de uma tarefa, é guardado numa Lista o Objeto Job enviado ao servidor. Depois do servidor enviar ao cliente o resultado do Job, a lista de tarefas do cliente é atualizada com a informação se a execução da tarefa foi bem sucedida ou não. No momento de escolha desta opção, o cliente então percorre a lista de Tarefas e cria uma Lista de "Strings" com o número de identificação da tarefa e qual o seu estado.

3.4 Pedido de execução de tarefa

```
# Job Execution Request #

Please insert job code file name:
file1

Please insert the total amout of memory required for this job:
500

Sucessfully send request for job execution, with code 2.
```

Figura 6: Pedido de execução de uma tarefa.

Um utilizador ao selecionar a opção para realizar pedido de execução de uma tarefa é-lhe solicitado o nome do ficheiro com o código da tarefa a ser executada e a memória necessária para a sua execução. O ficheiro é então lido através do método readFile() da classe **ClientController** e criado um Objeto do tipo **Job** com a informação recolhida. Este objeto é então serializado e enviado ao servido encapsulado numa Mensagem. O Servidor ao receber esta mensagem deserializa o Job, verifica se possui algum servidor **Worker** cuja memoria total seja igual ou superior à do Job, em caso afirmativo informa o Cliente que o Job irá ser executado quando

possível. Este Job é então adicionado à Queue de jobs pendentes da Classe **JobManager** e sinaliza a Thread do servidor da classe **JobManager** que existe um novo Job a executar. Esta Thread, se tiver à espera de tarefas, acorda, retira o próximo Job pendente da fila e verifica se algum dos **Workers** conectados possui memória disponível para a execução do Job, em caso afirmativo envia o Job encapsulado numa Mensagem ao **Servidor Worker** e coloca o Job no Map de Jobs em execução, em caso negativo fica a aguardar que algum worker envie o resultado de um **Job** e repete o processo. O Servidor Worker ao receber o Job cria uma Thread para executar o Job e no final envia o resultado ao Servidor Central. O Servidor Central, ao receber então a resposta do Job, sinaliza a Thread à espera de memória, retira o Job do Map de Jobs em execução e insere o Job no Map de Jobs concluídos. No fim sinaliza a Thread do Utilizador que espera respostas de Jobs para enviar ao utilizador. O Utilizador recebe então a resposta do Job numa mensagem com o **Type 5** e sinaliza a Thread que aguarda por resultado de tarefas, apresenta o resultado da tarefa ao utilizador e se a execução da tarefa tiver sido executada com sucesso escreve o resultado num ficheiro na pasta **JobResults** do utilizador.

No caso de um utilizador solicitar uma tarefa e fechar o programa sem receber o seu resultado, no momento em que o utilizador realiza uma nova autenticação é-lhe apresentado o resultado. Além disso, quando um **Servidor Worker** fecha inesperadamente, todos os jobs que estavam a ser executados por ele e que não obtiveram conclusão são novamente postos na fila de job's pendentes para serem enviados para outro **Worker's**.

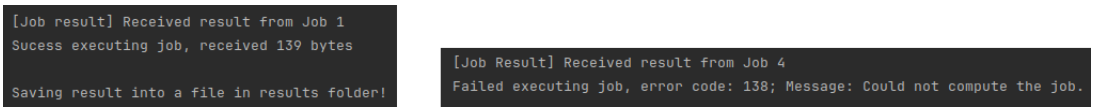


Figura 7: Notificações de conclusão de tarefas

4 Conclusão e trabalho futuro

A realização deste trabalho prático permitiu-nos aplicar os conhecimentos adquiridos pela unidade curricular de **Sistemas Distribuídos** sobre a implementação de serviços **cliente-servidor**, a manutenção correta de todo o ciclo de vida de uma **Thread** e como resolver **problemas de concorrência**. Foram desenvolvidas todas as funcionalidades necessárias para o bom funcionamento da aplicação de *Cloud Computing*. Foram implementados três sistemas distribuídos, Cliente, Servidor e Worker, todos dependentes entre si, bem como a criação de protocolos de comunicação entre estes sisteams. Para o cliente foi adicionada uma interface em texto para interação com o utilizador assim como leitura e escrita de ficheiros de tarefas. Para o Servidor para além das funcionalidades propostas foi adicionado a criação e leitura de um ficheiro de configuração com as contas registadas no sistema.

Acreditamos ter conseguido superar todos os objetivos propostos pelo trabalho, contudo, para trabalho futura fica, melhorar a parte de serilização de Objetos, em específico dos objetos da classe Job, devido a gastar recursos desnecessários na criação e fechamento de múltiplos **ByteArray[Input/Output]Stream** e **Data[Input/Output]Stream**. Além disso, para poupar recursos, realizar a implementação de uma **Thread Pool** evitando a criação e destruição de múltiplas Threads, por exemplo, nos Servidores Worker's no momento de execução de um Job.