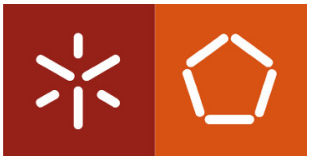
Universidade do Minho

Escola de Engenharia



Sistemas Distribuídos

Licenciatura em Engenharia Informática

Cloud Computing com funcionalidade Function-as-a-Service

Grupo 11

Gonçalo Gonçalves - [A90969]
Henrique Vaz - [A95533]
Pedro Oliveira - [A98712]
Vasco Rito - [A98728]

Dezembro, 2023

Conteúdo

1 Introdução 2

2 Estrutura da mensagem 2

3 Autenticação e registo 3

4 Pedido de execução 3

5 Consulta do estado atual 4

6 Submissão de novos pedidos 4

7 Ordem de execução de tarefas 4

8 Gestão da fila e Máquinas trabalhadoras 5

9 Testes 6

10 Diagrama de Sequência de um Fluxo Normal 6

11 Conclusão 7

1. Introdução

O presente relatório descreve o desenvolvimento de um serviço de cloud computing com a funcionalidade Function-as-a-Service (FaaS).

O objetivo deste projeto é implementar um sistema distribuído que permita a execução de tarefas de computação enviadas por clientes locais. O serviço, baseado em Cloud Computing, utiliza a abordagem FaaS, em que um cliente submete o código de uma tarefa para execução em servidores remotos. O desafio central é garantir uma eficiente utilização dos recursos disponíveis, com a memória sendo o fator limitante nos servidores.

Este relatório fornecerá uma visão geral das funcionalidades básicas e avançadas implementadas, bem como a abordagem distribuída adotada para gerir um conjunto de servidores. Será também discutida a implementação do protocolo de comunicação, a biblioteca do cliente e a interface do utilizador desenvolvidas.

Ao longo do relatório, serão destacadas as escolhas arquiteturais, desafios enfrentados e estratégias utilizadas para garantir a eficiência do sistema. O trabalho abrange desde a autenticação e execução de tarefas até a capacidade de lidar com múltiplos servidores de forma distribuída.

2. Estrutura da mensagem

A estrura geral de mensagem usada no nosso projeto é a da Figura 2, o campo `data` contém os dados em bytes que vão na mensagem (pode estar vazio, por exemplo numa mensagem de erro), o campo `length` representa o tamanho do *array de bytes data*, o campo `num` contém o valor de memória necessário (caso seja uma mensagem em que a memória seja relevante), o campo `tag` contém a *tag* atribuída à mensagem, e, por fim, o campo *msg* contém o tipo da mensagem, podendo ser um dos seguintes:

- 0. Desconexão
- 1. Autenticação
- 2. Registo
- 3. Enviar código
- 4. Autenticação errada
- 5. Autenticação efetuada
- 6. Username já utilizado
- 7. Username registado
- 8. Resultado FaaS
- 9. Erro na execução
- 10. Demasiada memória necessária
- 11. Enviar memória disponivel

- 12. Consulta estado atual.

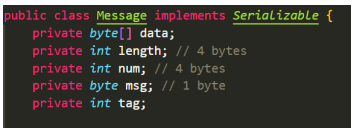


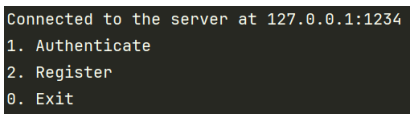
Figura 2.1: Estrutura de uma mensagem

3. Autenticação e registo

Para um cliente poder usar o nosso serviço então terá de efetuar o seu registo, caso já tenha o registo feito então terá de efetuar a sua autenticação.

Para a autenticação o programa cliente pede ao utilizador o seu *username* e *password* concatenando-os numa *string* separada por um espaço, criamos uma nova `Mensagem` com *id* 1 (simboliza uma tentativa de autenticação) e com a *string* anterior convertida para um *array* de bytes. No servidor principal ao receber a mensagem com este *id* verificamos se o *username* e *password* correspondem a alguma entrada no *Map* dos registos,caso exista envia uma mensagem de volta ao cliente com o *id* 5 (autenticação efetuada), caso não exista então a mensagem de volta vai com o *id* 4 (autenticação errada). Após o cliente receber a resposta altera o seu estado conforme a mesma, podendo ficar autenticado ou não.

Para o registo o programa cliente pede ao utilizador o seu *username* e *password* concatenando-os numa *string* separada por espaço, criamos uma nova `Mensagem` com *id* 2 (simboliza uma tentativa de registo) e com a *string* anterior convertida para um *array de bytes*. No servidor principal ao receber a mensagem com este *id* o servidor verifica num *Map* de utilizadores se o utilizador já está registado, caso esteja envia uma mensagem de falha ($id = 6$ que simboliza *username* já em utilização), caso não esteja registado então o servidor adiciona uma nova entrada no *Map*, sendo a *key* o *username* e o *value* a *password* (encriptada) enviando de volta uma mensagem com *id* 7 (username registado). Após a receção da resposta do servidor o cliente tanto pode criar mais um registo ou autenticar-se.



4. Pedido de execução

Após estar o utilizador autenticado pode enviar tarefas para o servidor executar. Quando o utilizador seleciona no menu que pretende enviar trabalhos para o servidor (fig. 5) é lhe pedido o *path* do ficheiro que contém os dados e a memória que o trabalho vai precisar. É então criada uma nova *thread* para gerir esta interação. O conteúdo do ficheiro é convertido para `byte[]` e é criada uma nova `Mensagem` contendo os *bytes* do ficheiro, a memória necessária para o seu processamento e a `tag`, sendo que o *id* da `Mensagem` é 3 (enviar código). A `Mensagem` é então enviada para o servidor principal.

No servidor principal é recebida a `Mensagem` com o `id` 3 e cria com os seus dados um novo `Job` contendo a informação da `Mensagem` mais o `socket` do cliente para posteriormente ser enviada a resposta ao mesmo. A parte de execução do *Job* está contida nos pontos 7 e 8.

Após o `Job` executado, o servidor prepara uma nova `Mensagem` contendo os `bytes` já processados,

a mesma `tag` que a `Mensagem` original tinha, e com `id` 8 (resultado faas) no caso do `Job` ter sido executado com sucesso. No caso de haver falha então a `Mensagem` apenas irá conter a `tag` e o `id` 9 (erro na execução).

O programa cliente após receber a `Mensagem` de resposta do servidor cria um ficheiro com o conteúdo da `Mensagem` caso esta venha com a indicação de que foi obtido o resultado esperado, caso a `Mensagem` venha com a indicação que houve falha então é apresentada uma `Mensagem` de erro via *standard output*.

5. Consulta do estado atual

Quando o utilizador indica no programa cliente que pretende consultar o estado atual de ocupação (fig. 5) do serviço é criada uma nova *thread* para gerir a interação. O programa cliente envia então uma nova `Mensagem` com *id* 12 (consulta estado atual).

O servidor principal ao receber esta mensagem percorre a lista de servidores secundários contida na classe `SSQueue` somando cada um dos valores de memória disponível ate obtermos o total de memória disponível, como a gestão da memória dos servidores secundários é efetuada no servidor principal temos sempre acesso ao valor atualizado, sendo que quando um `Job` é enviado para executar retiramos à memória disponível e quando termina o valor é reposto. O servidor principal envia então uma nova `Mensagem` com *id* 12 e com o tamanho da sua `JobList` (fila de `Job` à espera de serem executados) de volta para o cliente.

O programa cliente ao receber a resposta imprime-a no *standard output*

```
Authentication successful. Welcome!
4. Enter File Name
5. Check Memory and Queue Size
0. Exit
```

Figura 5.1: Menu após utilizador autenticado

6. Submissão de novos pedidos

Como o programa cliente é *multi-threaded*, bastou criar-mos uma classe `Demultiplexer` que faz uso de uma *tag* para conseguirmos distinguir os diferentes pedidos que podem ser enviados em simultâneo, após o cliente receber as respostas aos pedidos escreve os resultados em ficheiros, sendo que a *tag* faz parte do nome do ficheiro de forma a ser mais fácil distinguir as mensagens.

7. Ordem de execução de tarefas

De forma a garantir uma ordem de execução das tarefas de forma a que os pedidos que exigem grandes capacidades de memória não fiquem em *starvation* enquanto os processos que exigem pouca memória executam resolvemos fazer uma adaptação do algoritmo *Round-Robin*.

Deste modo, criámos a variável `JobOrder` que possui uma variável denominada `memLeft` que é inicializada com o total de memória que um `Job` exige. De seguida definimos um *quantum*. Enquanto que no algoritmo original esse *quantum* reflete o tempo que um processo tem estipulado para executar de cada vez, no nosso caso não é possível dar segmentos de execução, pelo que sempre que um `JobOrder`

é chamado, esse quantum é removido à variável `memLeft` até esta ser zero.

Assim, teremos duas estruturas de dados. A primeira é uma `List`, de forma a manter a ordem de chegada para onde todos os `JobOrder` vão inicialmente. A segunda é um *Set* visto que a ordem já não é relevante, para onde vai o `JobOrder` sempre que atinge a marca zero na variável `memLeft`, funcionando como uma estrutura com prioridade.

Posto isto, teremos uma *Thread* responsável por distribuir os `JobOrder` que vão sendo adicionados. Enquanto a `List` estiver vazia esta *Thread* encontra-se adormecida em *await*, no entanto sempre que é adicionado um novo `JobOrder` esta acorda e com o auxílio do método `removeJob` vai escolher o primeiro `JobOrder` possível. Para isso, vai pegar no elemento que estiver na cabeça da `List` e remover o *quantum* à `memLeft` se for zero, é adicionado ao `Set` com prioridade e se a memória do `Job` estiver disponível então atualiza a memória disponível do servidor e devolve o `Job`, no caso de a memória estar indisponível a *Thread* fica em await até ser acordada através do método `freeCond`, que é um método estático que vai ser utilizado sempre que a execução de um `Job` é terminada. No caso de `memLeft` ser superior a zero, então o `JobOrder` é colocado na cauda da `List`.

Desta forma conseguimos garantir que todos os `Job` são executados evitando *starvation* dos processos que exigem mais memória e continuando a garantir que os processos que exigem menos memória são executados mais rapidamente sem ser uma fila *FIFO*. Para além disto, garantimos que todas as estruturas de dados e a sua manipulação são *Thread-safe* ao longo de toda esta execução recorrendo a locks para as aceder.

8. Gestão da fila e Máquinas trabalhadoras

De forma a garantir uma implementação distribuída resolvemos separar a nossa arquitetura em servidor Primário,responsável por gerir a fila de espera e atribuir tarefas e servidores Secundários responsáveis por executar tarefas.

Desta forma, o servidor Primário inicialmente, conhecendo os servidores Secundários, estabelece ligação com os mesmos, recebendo inicialmente de cada um uma mensagem indicando a quantidade de memória que cada um disponibiliza para a execução de tarefas. Assim que recebe esta informação, a partir da mesma e da instância `Connection` criada, cria uma instância de `SSData` que armazena na classe `SSQueue`. Esta última classe é composta por uma `PriorityQueue`, de forma a inserir as `SSData` pela ordem pretendida, que neste caso é a quantidade de memória disponível, traduzida no `Comparator`, e por toda a manipulação desta estrutura de dados de forma a ser *Thread-safe*.

Assim, sempre que a *Thread* responsável pela classe `JobManager` mencionada anteriormente é acordada, esta remove da `PriorityQueue` o servidor Secundário que possui maior memória disponível, atualiza a sua memória disponível subtraindo a quantidade de memória que o `Job` exige e volta a inserir na `PriorityQueue` de forma a que volte ser inserido ordenadamente.

Uma vez associado o processo ao servidor Secundário, é inicializada uma *Thread* responsável pela execução do mesmo, isto é, envia o `byte[]` correspondente ao ficheiro que lhe foi enviado para o tal servidor Secundário, de seguida recebe o output proveniente do mesmo, atualiza a memória disponível do servidor Secundário, e faz a respetiva tradução consoante tenha sido processado com sucesso ou não. Essa mensagem será reencaminhada pela mesma *Thread*(do servidor Principal) para o Cliente através da instância de `Connection` que está guardada em `Job`.

Quanto à execução no servidor Secundário, este faz uso do ficheiro `sd23.jar` enviando uma mensagem específica para o servidor Primário com o output no caso de ter sido realizado com sucesso, ou no caso de ter apanhado a `JobFunctionException` manda outro tipo de mensagem diferente para poderem

ser distinguidas.

9. Testes

Para testar o sistema é importante percebermos quais os argumentos que cada programa recebe.

O servidor secundário recebe como argumento apenas o número da porta onde estará a "ouvir". A partir da pasta src podemos correr um servidor secundário na porta 12345 com:

→ java -cp . SecundaryServer.Main 12345

Após termos pelo menos um servidor secundário a correr teremos de correr o servidor principal, que recebe por argumento o endereço ip e a porta de cada um dos secundários a porta do servidor principal é fixa e é a 1234. A partir da pasta src podemos correr um servidor principal com:

→ java -cp . Server.Main 127.0.0.1 12345

Por fim para correr um cliente normal não precisamos de parametros, visto que a porta do servidor principal é fixa e conhecida. A partir da pasta src temos:

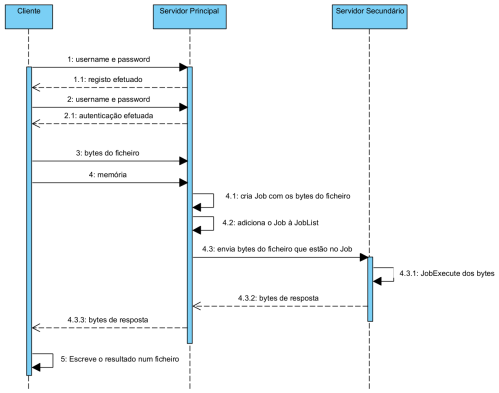
→ java -cp . cliente.Main

De forma a facilitar os testes foi também implementada uma versão do cliente para testes que salta a parte de autenticação e registo e apenas envia código para executar e espera resultados. Para executar esta versão basta ao executar o programa cliente passar como argumentos o *path* para o ficheiro e o valor da memória. A partir da pasta src:

→ java -cp . cliente.Main "C:\Users\foo\bar\test.txt"12

Posto isto, foram criados 3 scripts de forma a testar mais facilmente (tanto em formato *.bat* como em formato *.sh*), um script cria 3 servidores secundários(facilmente podemos alterar este valor), o segundo script cria o servidor principal e, por fim, temos um script que cria N clientes e envia as N mensagens em simultâneo para o servidor principal usando o cliente de testes. Desta forma conseguimos testar se os `Jobs` estão a ser bem distribuidos entre servidores secundários e verificar se há starvation de algum `Job`.

10. Diagrama de Sequência de um Fluxo Normal



11. Conclusão

O desenvolvimento do serviço de cloud computing com a funcionalidade Function-as-a-Service (FaaS) apresenta uma abordagem distribuída, permitindo que clientes enviem tarefas de computação para execução em servidores distribuídos. Ao longo do projeto, foram implementadas funcionalidades essenciais, discutidas escolhas arquiteturais e realizados testes para verificar a eficiência do sistema.

Destacamos algumas melhorias propostas para aprimorar ainda mais o sistema:

Client Handler Multithreaded: Esta melhoria visa aumentar a eficiência do sistema, uma vez que parte do processamento está a ser feito sequencialmente e poderia ser utilizada uma thread para isso, mas dado que a secção crítica não exige muitos recursos pode até servir como uma espécie de controlo de tráfego gerado pelo cliente, embora ela possa fazer pedidos em simultâneo da forma que está implementada de momento.

Resiliência a Falhas de Servidor Secundário: Reconhecemos a importância de tornar o sistema resiliente a falhas de servidores secundários sendo por isso sistemas autónomos. A introdução de mecanismos de resiliência, como redundância ou *failover*, permitirá que o sistema continue funcionando mesmo em caso de falha de um servidor secundário.

Gestão de Memória para Servidores com Capacidades Diferentes: Nós desenvolvemos a gestão de memória assumindo que os servidores têm todos memórias idênticas, no caso de as memórias serem diferentes teríamos de alterar o nosso algoritmo de distribuição de *jobs* apenas com a condição de o *job* exigir menos ou a mesma memória que a memória total do servidor a selecionar.

Essas melhorias visam aprimorar a eficiência, a resiliência e a capacidade de adaptação do sistema em diferentes cenários. Além disso, a implementação de monitorização contínuo e tratamento adequado de exceções contribuirá para a robustez do serviço de *cloud computing* baseado em *FaaS*.

Em conclusão, o sistema apresenta uma estrutura sólida, mas as melhorias propostas são essenciais para enfrentar desafios específicos, como aumento da carga, resiliência a falhas e variações nas capacidades dos servidores. A implementação dessas melhorias garantirá um serviço mais confiável e eficiente.