Universidade do Minho
Departamento de Informática

Sistemas Distribuídos

Trabalho Prático

Realizado por:
José Rodrigues (a100692)
João Coelho (a100596)
Duarte Araújo (a100750)
Rafael Peixoto (a100754)

December 30, 2023
Ano Letivo 2023/24

Índice

1 Introdução

No contexto da computação distribuída e serviços em nuvem, conceitos introduzidos no âmbito da cadeira de Sistemas Distribuídos, o projeto propõe a implementação de um serviço de computação em nuvem com funcionalidade "Function-as-a-Service" (FaaS). O FaaS é uma abordagem inovadora que permite aos utilizadores enviar tarefas de computação para execução remota em servidores, sem a necessidade de administrar diretamente a infraestrutura subjacente. Este serviço será desenvolvido em Java, utilizando *threads* e *sockets TCP* para estabelecer comunicação eficiente entre os clientes e os servidores.

O objetivo principal deste projeto é criar um ambiente capaz de receber, executar e retornar resultados de tarefas de computação, sendo que o recurso limitante é a memória disponível. O serviço deve garantir uma boa utilização dos recursos, evitando que os pedidos em execução ultrapassem a quantidade máxima de memória.

Além das funcionalidade básicas, que incluem autenticação de utilizadores, execução de tarefas e consulta do estado do serviço, exploraremos funcionalidades avançadas, como a capacidade de submeter novos pedidos antes de receber respostas anteriores e garantir uma ordem de execução que evite bloqueios prolongados.

A proposta inclui ainda uma implementação distribuída, onde múltiplos servidores colaboram para gerir tarefas, cada um com uma configuração própria de memória. Esta abordagem visa melhorar a escalabilidade e eficiência do serviço.

Ao longo deste relatório, exploraremos a arquitetura e design do sistema, detalhes da implementação, estrutura do código-fonte, protocolo de comunicação, e os desafios enfrentados durante o desenvolvimento. Este projeto não só desafia a nossa compreensão dos conceitos fundamentais de computação em nuvem, mas também nos incentiva a explorar soluções inovadoras para problemas complexos de coordenação e distribuição de tarefas.

2 Arquitetura

A arquitetura do sistema é projetada para oferecer um serviço de computação em nuvem eficiente e escalável, com ênfase na execução de tarefas de computação distribuída. O sistema é composto por diferentes componentes, cada um desempenhando um papel específico na realização das funcionalidades propostas.

2.1 Clientes

Os clientes são máquinas locais que interagem com o serviço para enviar tarefas, autenticar-se e consultar o estado do sistema. Cada cliente utiliza a biblioteca fornecida para estabelecer uma conexão segura com o servidor e realizar operações no serviço.

2.2 Servidor Centralizado

Uma máquina centralizada responsável por receber tarefas, administrar a fila de espera e distribuir tarefas para os servidores de execução de programas(*Workers*). A autenticação de utilizadores ocorre neste servidor, garantindo acesso seguro ao serviço.

2.3 Servidores de Execução

Múltiplas máquinas dedicadas à execução de tarefas, sendo utilizadas para distribuir a carga de trabalho. Após a execução da tarefa ter sido concluída, o resultado é enviado novamente para o servidor centralizado que, por sua vez, irá transmitir o resultado para o devido cliente. Cada *Worker* possui uma configuração específica de memória disponível para otimizar a utilização dos recursos.

2.4 Comunicação Entre Componentes

A comunicação entre os componentes do sistema é realizada utilizando *sockets TCP*, proporcionando uma troca eficiente de dados. O servidor centralizado opera em duas portas distintas, 9090 para aceitar conexões de clientes e 9091 para conexões de *Workers*.

Cliente - Servidor Centralizado: Os clientes estabelecem conexões com o servidor centralizado na porta 9090, utilizando o conjunto de classes e métodos fornecidos para interagir com o serviço. As mensagens trocadas entre os clientes e o servidor centralizado são estruturadas para otimizar a transmissão de dados durante o envio de tarefas, autenticação e consultas de status do sistema.

Servidor Centralizado - Workers: O servidor centralizado opera na porta 9091 para aceitar conexões de *Workers*. A comunicação bidirecional entre o servidor centralizado e os *Workers* é essencial para a distribuição eficiente de tarefas e a transmissão de resultados. As mensagens trocadas são projetadas para manter a integridade dos dados durante a transmissão, garantindo a sincronia operacional.

Assim, com toda a estrutura apresentada, torna-se possível visualizar a arquitetura do sistema na sua totalidade.

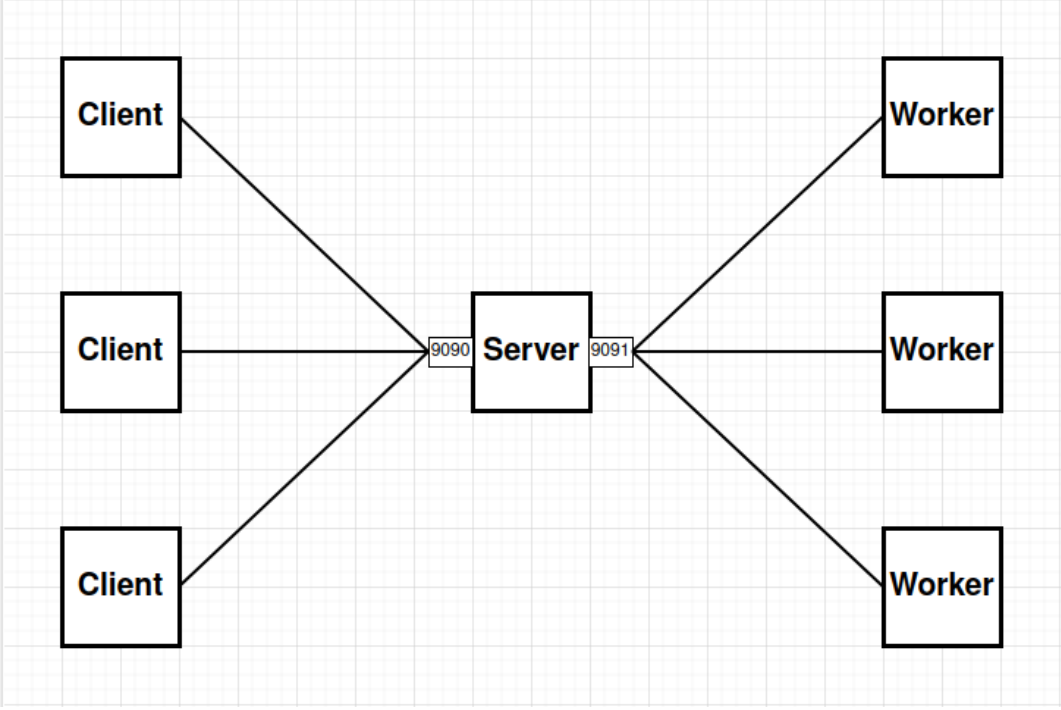

Fig. 1: Arquitetura Geral do Sistema

3 Algoritmos de decisão

No nosso trabalho foram desenvolvidos dois algoritmos de decisão que desempenham papéis fundamentais na ordenação da fila de espera de programas a executar e na escolha do melhor *worker* disponível, promovendo uma distribuição equitativa e eficaz das tarefas no ambiente distribuído.

Algoritmo de Ordenação da Fila de Espera de Programas: O primeiro algoritmo, responsável pela ordenação da fila de espera de programas, utiliza uma abordagem baseada na prioridade. A prioridade de um programa é calculada considerando fatores como consumo de memória e tamanho do arquivo do programa. A fórmula de cálculo da prioridade é:

$$\frac{1}{(0.9 \times \text{memory} + 0.1 \times \text{file.length})} \times 1000 \tag{1}$$

Este valor será posteriormente normalizado de forma a não se afastar demasiado de um certo intervalo de valores. A ordenação é realizada de forma decrescente com base na prioridade. Em caso de empate, são aplicados dois critérios adicionais para desempatar, sendo que o primeiro critério considera a quantidade de memória que o programa ocupa no *worker*, priorizando valores menores e, por sua vez, o segundo critério é a data de inserção do programa na fila de espera, conferindo prioridade aos programas mais antigos. É importante referir que a ordenação é realizada sempre que um elemento é adicionado à fila de espera.

Algoritmo de Seleção do Melhor Worker Disponível: O segundo algoritmo desempenha um papel crucial na seleção do *worker* mais adequado para a execução de um programa. Este algoritmo considera o número de programas em execução nos *workers* e a quantidade de memória disponível. A escolha prioriza *workers* com menor carga de trabalho e maior disponibilidade de recursos, ou seja, mais memória disponível. É importante referir que a verificação da memória disponível é um critério de desempate.

4 Protocolo de Mensagens

O sistema desenvolvido utiliza um protocolo de mensagens para a comunicação eficiente entre os componentes, incluindo clientes, servidores e *workers*. O protocolo é implementado pela classe *Message*, que encapsula informações essenciais em mensagens bem definidas.

4.1 Estrutura da Mensagem

Cada mensagem consiste em dois campos principais:

1. Tipo (*Type*): Indica a natureza da mensagem, como solicitação de execução de programa, atualização de status ou resposta a uma operação específica.
2. Payload : Contém os dados associados à mensagem. É formatado como uma sequência de strings, sendo os argumentos separados por tabulações (‘ \t’).
3. Tamanho do Payload(*payload_ length* : Indica o comprimento do payload. Ao enviar uma mensagem, o tamanho do mesmo é adicionado automaticamente pela função de serialização. Na prática, a mensagem é composta por tipo, tamanho do payload e o próprio payload.

message_type	payload_length	payload

4.2 Serialização e Deserialização

Para transmitir mensagens através da rede, é necessário converter os objetos *Message* em fluxos de bytes (serialização) e reconstruí-los a partir desses fluxos (deserialização). As funções *serialize* e *deserialize* realizam essas operações, garantindo uma transmissão eficiente e confiável. Para além disso, é importante destacar que ambas as operações são implementadas em ciclos, especialmente úteis quando o tamanho do payload excede o tamanho do *buffer* disponível. Esse processo cíclico permite a leitura ou escrita contínua do payload, assegurando a integridade da comunicação em situações em que o tamanho do mesmo é substancialmente maior que o *buffer*.

5 Implementação

O Sistema de Execução Distribuída de Programas desenvolvido proporciona uma solução eficaz para a execução distribuída de programas em um ambiente distribuído. Desenvolvido em Java, este sistema é composto por diferentes componentes que colaboram harmoniosamente para atender aos pedidos dos utilizadores e garantir a execução eficiente de tarefas complexas. Assim sendo, encontra-se a seguir uma pequena apresentação de cada um dos componentes desenvolvidos.

5.1 Cliente

Começando pelo cliente, a sua implementação segue uma abordagem modular, adotando o padrão *Model-View-Controller*(MVC) para garantir uma separação clara das responsabilidades. Foi então desenvolvida uma classe *Client*, responsável por inicializar o controlador, a vista e um objeto da classe *Account*, onde os dados da conta do cliente serão armazenados. Para além disso, são chamadas duas funções, responsáveis por estabelecer a comunicação com o servidor e criar os canais de comunicação, e por criar o menu que o utilizador irá usar para interagir com o serviço. Assim, passando para o controlador, é nesta classe que estão definidos todos os métodos responsáveis pela comunicação com o servidor. Dependendo da escolha do utilizador no menu criado, será chamada a função do controlador responsável por tal operação e, dependendo da operação a ser realizada, os dados inseridos pelo utilizador serão lidos e, por sua vez, será enviada uma mensagem para o servidor que contêm as informações dadas. Caso a operação selecionada pelo utilizador tenha sido o registo, seria enviada uma mensagem com Message_Type = "REGISTER" e Payload = "NomeUtilizador\tPalavraPasse". Assim, depois da mensagem ter sido enviada para o servidor, caso a mesma seja uma mensagem de registo ou de *login*, a função ficará à espera de receber o resultado do servidor. É importante referir que se o *login* for bem sucedido será criado um outro menu somente disponível para utilizadores autenticados, onde os mesmos poderão enviar programas para executar ou questionar acerca do estado do servidor. Assim, nestas operações exclusivas a utilizadores autenticados, depois da mensagem ter sido enviada ao servidor, é criada uma *thread* que fica à espera da resposta do servidor. Dependendo do *Message_ Type* da resposta, o resultado será tratado por outras funções que estão encarregadas de escrever o resultado num ficheiro ou imprimir os dados para o utilizador poder observar. Esta *thread* é criada de modo a que o cliente seja capaz de receber diversos resultados concorrentemente.

5.2 Servidor

Passando agora para o lado do servidor, como já foi referido anteriormente, são disponibilizadas duas portas, a 9090 para conexões com clientes e a 9091 para conexões com *workers* e, a partir dessas

portas, são criados dois *serverSockets*, um para cada porta. De seguida, são criadas duas *threads*, cada uma responsável por aceitar conexões de um dos componentes(clientes e *workers*), que, por sua vez, sempre que aceitam uma conexão criam uma *thread handler* que será responsável pela comunicação com o elemento aceite (*ClientHandler* para clientes e *WorkerHandler* para *workers*). Assim sendo, começando pelo *handler* do cliente, este ficará à espera de mensagens do mesmo e, dependendo do *Message_ Type* da mensagem, irá realizar um certo conjunto de operações. Se a mensagem recebida for uma de registo, então os dados enviados serão recebidos e será feita uma verificação dos mesmos, de modo a verificar se já não existe uma conta registada no servidor com esses dados. Para isso é necessário fazer uma verificação do *map accounts* que, como o nome indica, guarda as informações das contas registadas no sistema, sendo que a *key* da estrutura é o nome de utilizador da conta, enquanto o valor é um objeto *Account*. Por outro lado, caso a mensagem recebida pelo servidor seja uma mensagem de *login*, após os dados serem recebidos, também será necessária uma verificação dos dados da conta, de modo a verificar se a mesma já se encontra registada no servidor e que ninguém a está a utilizar no momento. Para verificar esta segunda condição é utilizado outro *map* que é responsável por armazenar as contas que se encontram ativas, ou seja, que estão a ser utilizadas no momento da verificação. Esta estrutura guarda o canal de escrita para o cliente e tem como *key* o nome de utilizador. De seguida, caso o *login* seja concluído com sucesso, o servidor ficará agora à espera de receber pedidos do cliente que só lhe são permitidos caso esteja autenticado, ou seja, os pedidos de execução de programas e verificação do estado do servidor. Assim sendo, caso o servidor receba uma mensagem de execução de um programa, os dados serão interpretados e, por sua vez, será criado um objeto do tipo *ProgramRequest* que é responsável por guardar diversos dados importantes para a execução do programa e posterior envio do resultado para o cliente. De seguida, este objeto criado é adicionado a uma *PriorityQueue*, implementada pelo grupo, denominada *pendingPrograms* e é nesta fase que o algoritmo de decisão responsável por ordenar a fila de espera atua. Veja a secção 3 para mais detalhes sobre o algoritmo de decisão referido. Por fim, se a mensagem recebida pelo servidor for uma de verificação do estado do mesmo, será enviada uma resposta que informe o cliente acerca do máximo de memória disponível num único *worker*, ou seja, qual o limite de memória que um programa do cliente pode ocupar, e do número de *jobs* à espera de serem executados.

Por fim, agora no *WorkerHandler*, o servidor pode receber três tipos de mensagem. O primeiro tipo é uma mensagem *Memory_ Info* que o servidor utilizará para associar um limite de memória ao *worker* que a enviou. De seguida, pode ainda receber dois tipos de mensagens que estão relacionados, *Job_ Done* e *Job_ Failed*. Como é de se esperar, estes tipos de mensagem são enviados pelo *worker* quando a execução do programa termina, no entanto, caso a execução seja concluída sem problemas, será enviada uma mensagem do tipo *Job_ Done* com o resultado do programa, enquanto, caso haja problemas durante a execução e a mesma tenha de ser cancelada, será enviada uma mensagem do tipo *Job_ Failed* com uma mensagem de erro. Os dados destas mensagens são posteriormente enviados para o cliente, onde os valores serão tratados.

5.3 Worker

Podemos agora passar para os *workers* que, como pode ser observado, funcionam em sintonia com o servidor. Quando uma mensagem de execução de um programa é tratada pelo servidor e, consequentemente, um *ProgramRequest* é adicionado à *PriorityQueue*, este terá então de ser enviado para um *worker* para ser executado. Assim, implementámos uma *thread*, que é criada no início da execução do servidor, e que, enquanto o mesmo estiver operacional, irá estar sempre à espera que *ProgramRequests* sejam adicionados à *PriorityQueue*, para que os possa enviar para os respetivos *workers*. Assim sendo, enquanto a *queue* se encontrar vazia esta *thread* ficará à espera, no entanto,

no instante em que um *ProgramRequest* é adicionado será feita uma verificação da memória dos *workers*, sendo que, caso haja *workers* disponíveis, os mesmos serão adicionados a uma lista denominada *availableWorkers*. Se, após a verificação, a lista se encontra vazia então a *thread* ficará à espera que algum *worker* acabe de executar programas que tenha pendentes para fazer novamente uma verificação global da memória disponível dos *workers*, esta espera é implementada utilizando *conditions*. Assim, se houver um ou mais *workers* disponíveis, será chamada uma função responsável por enviar o programa para o melhor *worker*. É nesta função que o algoritmo de decisão utilizado para selecionar o melhor *worker* disponível está definido, para mais detalhes sobre o algoritmo veja a secção 3. Depois do melhor *worker* ter sido encontrado, o programa é retirado da *queue pendingPrograms* e é chamada uma função que incrementa o valor da prioridade dos programas que se encontram na mesma. Este incremento é realizado apenas 50% das vezes que a função é chamada e equivale a 25% do valor da prioridade do programa que está a ser enviado para o *worker* executar. Esta função é importante para prevenir situações de *starvation*. Por fim, o programa é finalmente enviado para o melhor *worker* e tanto a sua memória disponível como o número de programas que o mesmo está a executar são atualizados. Cada *worker*, quando é inicializado, estabelece uma ligação com o servidor e cria os respetivos canais de comunicação com o mesmo. De seguida, envia uma mensagem para o servidor, denominada *Memory_ Info*, que é responsável por informar o mesmo acerca do seu limite de memória. Após a mensagem ter sido enviada, à semelhança do servidor, o *worker* fica à espera de receber mensagens só que, desta vez, as mensagens recebidas provém do servidor. O *worker* só recebe um tipo de mensagem, sendo ela uma mensagem de execução de um programa que lhe é enviada através do algoritmo de decisão apresentado anteriormente. Quando a mensagem é recebida e os dados são interpretados, o *worker* inicializa uma nova *thread* que será responsável pela execução do programa, sendo que o programa é executado utilizando as funções fornecidas pelos professores e o resultado é posteriormente enviado para o servidor. Esta *thread* é criada com o objetivo de permitir a execução de programas concorrentemente.

É importante referir que são usadas diversas primitivas de exclusão por todo o trabalho, de forma a garantir que não surgem condições de corrida ou outros tipos de problemas associados com o uso de *threads*. Algumas destas primitivas foram sendo abordadas ao longo da explicação da implementação, no entanto é impossível referir todas as primitivas utilizadas.

6 Conclusões Finais

Ao completar este projeto, obtivemos uma compreensão mais profunda dos sistemas distribuídos, enfrentando desafios e explorando soluções ao longo do caminho. O trabalho realizado reflete o nosso compromisso em desenvolver um Sistema de Execução Distribuída de Programas eficiente e modular.

Durante a implementação, utilizamos padrões de projeto, como o *Model-View-Controller*(MVC), para criar uma arquitetura coesa e facilitar o desenvolvimento e manutenção do código. A complexidade inerente aos sistemas distribuídos levou-nos a adotar abordagens criativas, fortalecendo o nosso conhecimento prático.

Estamos satisfeitos com os resultados alcançados, observando a eficácia do sistema na execução de tarefas complexas e na gestão de recursos distribuídos. No entanto, reconhecemos que há oportunidades para aprimoramento.

Como por exemplo a receção concorrente de mensagens por parte do cliente, que podia estar melhor implementada, bem como o próprio protocolo de mensagens utilizado que, apesar de funcionar, poderia, na nossa opinião, estar mais completo e eficiente.