Sistemas Distribuídos – Trabalho Prático

LEI – Universidade do Minho

2023/2024

Diogo Neto – 98197
Diogo Gonçalves – 101919
Guilherme Oliveira – 95021
Pedro Pinto – 87983

1 - Introdução

Este projeto traduziu-se no desenvolvimento de um serviço de *Cloud Computing FaaS*, ou *Function-as-a-Service*, que permite aos clientes executar código de resposta a eventos sem ter de lidar com a infraestrutura complexa tipicamente associada à construção e lançamento de aplicações de microsserviços.

Para alocar um software na internet, normalmente, é necessário prover e gerir um servidor virtual ou físico, assim como a administrar um sistema operativo e processos de hospedagem do servidor web. Com o FaaS, a gestão de *hardware* físico, do sistema operativo da máquina virtual e do *software* do servidor web são tratados automaticamente pelo provedor de serviços de nuvem. Isto permite aos desenvolvedores concentrarem-se exclusivamente nas funções individuais da sua aplicação.

O programa consiste num servidor local que é capaz de estabelecer comunicação com vários clientes e receber pedidos de cada um deles, concorrentemente.
Considerando que a única limitação do servidor é a memória, desde que não seja ultrapassado o limite da mesma, o servidor é capaz de manter uma fila de tarefas a executar.

2 - Cliente

Cada utilizador tem a si associada uma conta de cliente, ao utilizar o programa pela primeira vez é pedido ao utilizador para criar uma conta ou iniciar sessão, caso este já se tenha registado. Cliente é a entidade capaz de interagir com o ser servidor.

Quando um utilizador se regista como cliente os seus dados são enviados para o servidor para validação dos mesmos, isto é, o servidor verifica se já existem outros clientes com o mesmo *e-mail*. Após a validação, os dados do cliente são armazenados para que este possa fazer *login* futuramente.

Para que o utilizador se autentique como cliente, após introduzir as suas credenciais, o servidor verifica se o seu *e-mail* e *password* se encontram na base de dados e se estas informações correspondem às armazenadas. Em ambos os casos o servidor informa o cliente se o início de sessão/registo foi efetuado com sucesso.

Após uma autenticação bem-sucedida, o sistema recolhe tarefas dos clientes, isto é, o sistema pede ao utilizador uma tarefa para executar, nomeadamente o código correspondente à operação e a memória que esta necessita para ser executada, de imediato o servidor atribuiu um número ao pedido do utilizador. Enquanto o pedido do utilizador estiver a ser processado, este pode submeter novos pedidos e consultar o estado do servidor, nomeadamente a memória disponível e as tarefas na fila.

Para além de submeter pedidos manualmente o utilizador pode enviar ficheiros que contêm o código da tarefa e a quantidade de memória necessária para a sua execução. O programa irá enviar para o servidor cada um destes pedidos para que o cliente obtenha o processamento de cada um deles.

Cada resultado de um pedido é guardado num ficheiro como nome "*result_X*" na máquina do utilizador, onde *X* representa o *ID* da tarefa enviada.

3 – Conexão

A Conexão foi implementada essencialmente em `Connection.java` descreve a implementação de uma classe que representa uma conexão de socket TCP. A classe inclui campos para `DataInputStream` e `DataOutputStream`, responsáveis por manipular a entrada e a saída de dados na conexão. Dois blocos `ReentrantLock` (`rl` e `wl`) são utilizados para controlar acessos concorrentes durante operações de leitura e escrita, garantindo exclusão mútua mas também assegurando que uma thread que já possui o bloqueio possa adquiri-lo novamente sem bloqueio.
O construtor recebe um objeto `Socket` para inicializar os fluxos de entrada e saída, e os métodos `send` e `receive` são responsáveis por enviar e receber objetos `Frame` pela conexão, respeitando os bloqueios apropriados. O método `close` é implementado para encerrar os fluxos associados à conexão.
A estratégia de programação adotada utiliza *reentrant locks* para proporcionar flexibilidade e suporte a reentrada por parte da mesma thread. O controlo de leitura e escrita separado sugere uma abordagem cuidadosa para evitar conflitos entre operações concorrentes, evitando também *deadlocks* e contenção. No contexto do projeto, a classe `Connection` desempenha um papel fundamental facilitando a comunicação entre diferentes entidades do sistema distribuído, garantindo uma interface segura e concorrente para o envio e receção de dados.
Ainda, temos a classe `Demultiplexer` responsável por receber e despachar mensagens (`Frame`), facilitando a comunicação entre os diferentes componentes do sistema. A classe utiliza um objeto da classe `Connection` para receber as mensagens provenientes da conexão do socket.
Assim como na classe `Connection`, a classe `Demultiplexer` faz uso de um *`Reentrant Lock`*, denominado `l`, para garantir exclusão mútua em operações críticas. O mapa `map` mantém uma relação entre identificadores de tags e objetos `FrameValue`, que armazenam uma fila de mensagens associadas a uma determinada tag. Essa estrutura é crucial para lidar com as mensagens de forma assíncrona e organizada.
A classe interna `FrameValue` é utilizada para representar o valor associado a uma tag no mapa. Esta mantém uma contagem de threads em espera (`waiters`), uma fila de mensagens (queue), e uma condição (`c`) que é sinalizada sempre que uma nova mensagem é adicionada à fila.
O método `start` inicia uma thread para receber continuamente mensagens e armazená-las na fila correspondente ao identificador de tag associado. Os métodos `send` e `receive` são responsáveis por encaminhar mensagens para a conexão e recuperar mensagens da fila associada a uma tag específica, respectivamente.
No contexto do projeto, `Demultiplexer` atua como um componente intermediário entre a conexão do socket e as entidades do sistema distribuído, gerindo de maneira eficiente o envio de mensagens assíncronas entre os diferentes módulos do sistema.

4 – Frame

Para otimizar a troca de mensagens, utilizamos uma entidade de uma classe chamada *Frame*. Cada *Frame* incorpora três componentes principais: uma etiqueta numérica (*tag*), um identificador único para cada cliente, e o conteúdo da mensagem.

A *tag*, é um valor inteiro que facilita a categorização e o processamento da mensagem, já o identificador do cliente serve para que o servidor possa reconhecer o remetente da mensagem. Em situações onde o servidor é o remetente, este identificador pode ser deixado como uma string em branco, já que não tem relevância nesse contexto.

O conteúdo da mensagem é armazenado num array de bytes. Esta abordagem, que se baseia no uso de formatos binários, é especialmente vantajosa para a transmissão eficiente de grandes volumes de dados.

Na tabela abaixo podemos ver o tipo de pedido correspondente a cada tag.

Tag	Mensagem
0	Pedido de login por parte do cliente
1	Pedido de registo por parte do cliente
2	Pedido de execução de tarefa
3	Pedido de status de memória/ tarefas em espera no servidor

5 - Servidor

O servidor é responsável por atender aos pedidos de cada cliente.

Sempre que um cliente efetua um login ou registo, o servidor recorre à classe Accounts para consultar ou armazenar dados dos clientes, isto permite ao programa um nível de abstração maior.

Como existem várias operações de leitura e escrita em classes auxiliares do Servidor como Connection, Demultiplexer e Accounts, utilizamos *Locks* para evitar problemas no acesso a recursos partilhados, também conhecidos como problemas de exclusão mútua.

Como já referido, a principal função do servidor no nosso sistema de processamento de tarefas é atender aos pedidos realizados pelos clientes. Para isso, o servidor utiliza a classe BlockingQueue<Frame>, que armazena tarefas pendentes e controla a ordem de execução. Esta abordagem não só facilita a administração eficaz das tarefas pendentes, mas também assegura uma execução ordenada e justa através de um algoritmo de Round Robin.

Este método contribui significativamente para o balanceamento de carga e a otimização do uso de recursos, como a memória, que é monitorada para evitar sobrecarga.

Além disso, o servidor está equipado para lidar com várias situações excepcionais, incluindo erros e falhas, por meio de um sistema de tratamento de exceções.

A implementação de threads desempenha um papel vital na gestão de processos dos clientes. Cada cliente que se conecta ao servidor é imediatamente associado a uma thread dedicada, criada para lidar com todas as interações específicas desse cliente. Esta abordagem assegura que o servidor possa lidar com múltiplas solicitações de vários clientes.

Por outro lado, no processo de logout, a thread dedicada ao cliente trata de finalizar a sessão corretamente, isto inclui a atualização do estado do utilizador no sistema e a liberação de quaisquer recursos alocados durante a sessão.

As threads garantem que esses processos ocorram de maneira suave e ordenada, mesmo num ambiente onde múltiplos utilizadores podem estar entrar e sair do sistema simultaneamente. Esta abordagem multi-thread é crucial para manter o servidor responsivo e eficiente, permitindo que ele se adapte dinamicamente às variações na demanda de processamento. Além disso, a capacidade de lidar com várias sessões de utilizadores paralelamente ilustra a escalabilidade do servidor, tornando-o apto a operar efetivamente em ambientes de alto tráfego e com exigências intensivas de processamento de dados.

6 - Conclusão

Em suma, este projeto representa a materialização de um serviço inovador de Cloud Computing FaaS (Function-as-a-Service), visando proporcionar aos clientes a capacidade de executar código em resposta a eventos, sem a necessidade de gerir a infraestrutura complexa comumente associada à construção e implementação de microsserviços.
Embora tenhamos alcançado objetivos na implementação das funcionalidades básicas e avançadas de um serviço de Cloud Computing FaaS, apresentam-se também oportunidades

para uma expansão significativa. O sistema atualmente desenvolvido proporciona aos clientes a capacidade de autenticação, registo, execução de tarefas, consulta de estado do serviço e carregamento de tarefas provenientes de um ficheiro, operando de maneira eficiente na gestão de memória e fila de espera.
Contudo, reconhecemos que uma implementação distribuída, conforme sugerido no enunciado, poderia proporcionar melhorias substanciais. A arquitetura proposta de um serviço centralizado para gerir a fila de espera e atribuir tarefas a servidores dedicados para execução oferece escalabilidade e redundância. Esta abordagem poderia mitigar possíveis gargalos de desempenho, melhorando a eficiência geral do sistema.

Culmatando, o servidor, peça-chave do sistema, gere pedidos dos clientes e utiliza estratégias como Round Robin para a execução ordenada e justa das tarefas. O uso de locks em classes auxiliares, como Connection, Demultiplexer e Accounts, evita problemas de exclusão mútua em operações críticas, garantindo consistência e integridade no acesso a recursos compartilhados.

Em síntese, o projeto oferece uma solução robusta para a execução eficiente de tarefas num ambiente de Cloud Computing FaaS. As práticas avançadas de programação concorrente e a atenção meticulosa aos detalhes na comunicação entre entidades destacam-se como pontos positivos deste projeto na área da computação distribuída.