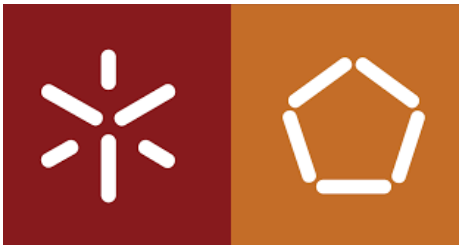

Licenciatura em Engenharia Informática

Universidade do Minho

Sistemas Distribuídos

Trabalho Prático

Cloud Computing

Dezembro 2023

Ema Maria Monteiro Martins a97678

Henrique Nuno Marinho Malheiro a97455

Lara Regina da Silva Pereira a100556

Martim de Oliveira e Melo Ferreira a100653

Introdução

O presente relatório foi desenvolvido no contexto do trabalho prático proposto na unidade curricular de Sistemas Distribuídos. Trata da descrição do processo de implementação de um serviço de *cloud computing* com funcionalidade *Function-as-a-Service* (FaaS). O sistema irá envolver um cliente numa máquina local que pretende enviar uma tarefa de computação para ser executada num servidor e, logo que haja disponibilidade, receber de volta o resultado.

Estrutura do trabalho

- **Módulo Client.java**
- **Módulo Message.java**
- **Módulo TaggedConnection.java**
- **Módulo Demultiplexer.java**
- **Módulo Server.java**
- **Módulo WaitingList.java**

Módulo Client.java

Quando um novo Client é instanciado, cria-se o socket, scanner, demultiplexer, lock e condition, tendo depois acesso ao menu1 que permite fazer registos, autenticações ou sair do programa..

Para se realizar uma autenticação, é pedido ao utilizador que introduza um nome de utilizador e uma palavra-passe. Em seguida, é enviado uma Message ao Servidor, com o type=0. O cliente aguarda pela resposta do Server, que indica se a autenticação foi concluída com sucesso ou insucesso. Em qualquer um dos casos, o cliente volta a um menu1.

Se o Client se quiser autenticar, é pedido ao utilizador que introduza um nome de utilizador e uma palavra-passe para posteriormente ser enviada uma Message com o type=1 para o Servidor. Se a autenticação não for realizada com sucesso o Client volta ao menu1. Caso seja realizada com sucesso, o Client tem acesso ao menu2 que lhe permite enviar tarefas, consultar a memória disponível, consultar o número de tarefas em fila de espera ou sair do programa.

Para enviar tarefas é pedido ao Client que introduza o caminho do ficheiro a executar, o caminho para o ficheiro resultado e o tamanho da tarefa, para que se possa enviar uma Message com type=2 para o servidor.

Enquanto são pedidas essas informações, a thread principal é adormecida para que não seja possível estarem duas threads a escrever em simultâneo no terminal. No caso de querer consultar a memória disponível, é enviado ao Server um Message com o type=3. Se se pretender consultar o número de tarefas em fila de espera, é mandado para o Server um Message com o type=4. Nestes dois últimos casos, a thread principal também é adormecida para que a escrita no terminal não tenha problemas.

Para enviar tarefas, consultar a memória disponível ou consultar o número de tarefas em fila de espera, são sempre criadas threads para enviar o respetivo Message para o Server e aguardar pela resposta do mesmo. Quando as respostas forem recebidas, são exibidas ao cliente.

Nas várias funções que efetuam pedidos ao servidor foram também implementados locks que permitem a atualização sequencial do número de pedido que está a ser efetuado, de forma a não causar um problema de corridas.

Quando o cliente pretender sair do programa, é fechado o scanner, as streams de escrita e de leitura, e o socket.

Módulo Message.java

A classe Message é o objeto padrão com que o cliente e o servidor irão comunicar. Cada Message é composta por um type (identifica a operação a ser realizada), um size (tamanho dessa operação), content (conteúdo da operação), um numMensagem (número da mensagem) e um counter (utilizada na waitingList do Server, dando uma noção do número de vezes que essa mensagem foi ultrapassada). No caso de uma Message ser uma tarefa que o cliente pretende enviar para o servidor executar, o size será fornecido pelo utilizador.

A classe Message contém as operações de serialize (escrever um Message no socket) e deserialize (ler um Message do socket, formando-o).

Módulo TaggedConnection.java

A Classe TaggedConnection é composta por um socket (estabelece a comunicação entre client e server), as DataStreams (permitem ler e escrever do socket), dois ReentrantReadWriteLock (um para escrita e outro para leitura no socket) e um Frame (estrutura que permite identificar uma determinada Message, recorrendo a uma tag).

Assim, temos a possibilidade de enviar e receber Frames concorrentemente através do socket (send e receive, respetivamente), onde estes métodos fazem o seu controlo de concorrência no acesso ao socket.

O TaggedConnection não garante que uma *thread* leia um Frame que não lhe pertença, podendo haver perdas de Frames, problema posteriormente resolvido com o Demultiplexer.

Módulo Demultiplexer.java

O Demultiplexer faz uma “ponte” entre um Client e um Server, usando, para isso, o TaggedConnection. Para resolver o problema do TaggedConnection, o Demultiplexer tem uma única thread a ler do socket e a escrever nos diversos buffers (método start), havendo um buffer para cada thread do Client. Assim, uma determinada thread do Client só irá receber (através do método receive) Frames do seu buffer, resolvendo-se o problema.

Módulo Server.java

O servidor gere as várias solicitações de execução de tarefas, oriundas dos vários clientes com quem mantém conexão. O servidor é inicializado com um ServerSocket (que possibilitará as comunicações com os clientes via TCP), um Map<String,String> (que faz a correspondência entre um nome de utilizador e a respetiva palavra-passe), uma WaitingList (uma fila de espera para as várias tarefas que o servidor deverá executar). Por cada pedido de conexão vindo de um cliente, o servidor criará um fio de execução diferente que atenderá aos pedidos de único cliente.

Cada fio de execução, ou *thread*, do servidor terá acesso ao Map de utilizadores e à fila de espera. Terá ainda uma capacidade máxima total do servidor e a capacidade disponível para realização de novas tarefas, a TaggedConnection e ReentrantLocks.

Os vários fios de execução ficarão sempre à escuta de novas mensagens provenientes do cliente a que estão associados.

● Registo de um novo cliente

No caso da mensagem indicar a intenção do cliente efetuar o seu registo, o servidor irá verificar a existência prévia do nome de utilizador no Map<String,String>. O nome e a palavra-passe serão registados com sucesso caso o nome de utilizador ainda não exista. Caso contrário, não poderá ser realizado o registo. Será enviada uma mensagem de sucesso ou insucesso, respetivamente, ao cliente que efetuou o pedido de registo.

Esta operação foi identificada como uma zona crítica, que poderá dar origem a um problema de corridas. Caso dois clientes solicitem o registo do mesmo nome de utilizador em simultâneo, um dos utilizadores não conseguirá realizar o registo e originará um erro.

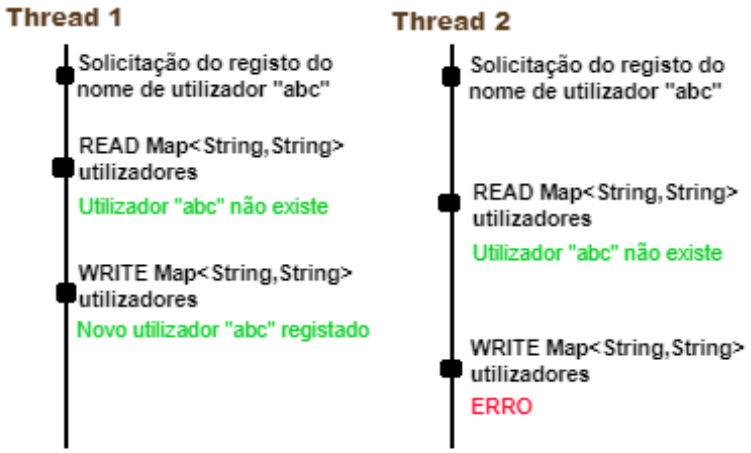

Figura 1 - Problema de corridas que surge no registo

Por este motivo, implementamos um ReentrantLock, que limita a permanência nesta zona de apenas uma *thread* de cada vez e resolve o problema anteriormente identificado.

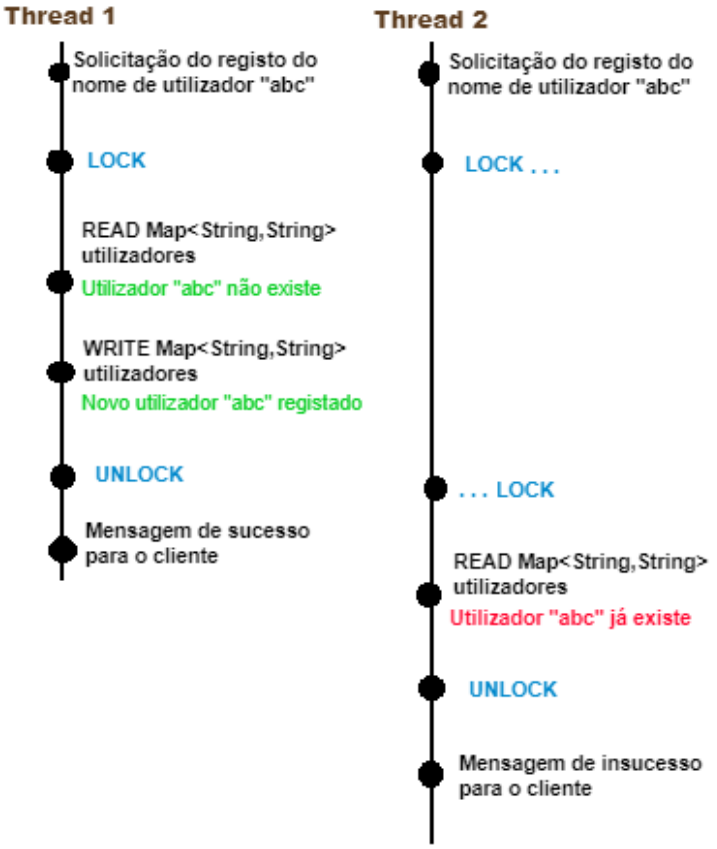

Figura 2 - Solução para o problema de corridas no registo (locks)

● Autenticação de um cliente

Se o cliente pretender autenticar-se, o servidor deve verificar se o seu nome de utilizador e palavra-passe correspondem a um par chave-valor do Map<String,String>. À semelhança do registo, caso o par exista, o cliente é autenticado com sucesso e recebe uma mensagem que o indica. Caso o par não exista, não é possível realizar a autenticação e o cliente recebe uma mensagem de insucesso.

Na autenticação do cliente, o acesso ao Map<String,String> também foi protegida através de locks uma vez que, apesar de ser apenas uma operação de leitura, poderá ser conflituosa com a operação de registo, caso se modifique o comportamento do cliente para efetuar um registo e uma autenticação em paralelo.

● Tarefa para execução

Quando o servidor recebe uma mensagem para execução de uma tarefa, adiciona a mensagem à fila de espera, implementada no módulo WaitingList.java.

Módulo WaitingList.java

A cada mensagem na fila de espera é-lhe adicionado um contador que serve para saber a prioridade de cada mensagem para sair da fila de espera, de forma a selecionar para execução as tarefas que estão à espera há mais tempo. Cada vez que é feita uma seleção que não inclui uma dada mensagem, o seu contador diminui.

Para o processo de seleção, a fila de espera é ordenada pelo número do contador de menor para maior, colocando no início da fila as mensagens que estão à espera há mais tempo. Após a ordenação, verifica-se o espaço livre no servidor e seleciona-se as tarefas que podem ser executadas, não ultrapassando esse valor. Após as tarefas serem selecionadas, são retiradas da fila de espera, colocadas numa lista auxiliar e enviadas para o servidor para serem executadas. Após a sua execução, o servidor atualiza o seu espaço livre para que mais tarefas possam ser escolhidas.

Zona crítica na execução de tarefas

Identificamos como zona crítica a adição de novas tarefas e a seleção das tarefas a serem executadas, uma vez que duas *threads* diferentes podem escrever na mesma posição da fila de espera e a que escrever por último irá eliminar o valor da thread que inseriu antes.

No entanto, o momento de seleção de tarefas para ser realizado também pode ser uma zona crítica. Como é uma estrutura que pode ser afetada por diferentes clientes ao mesmo tempo, devemos impossibilitar o acesso simultâneo. Isto pode levar a casos como um cliente que realiza um pedido de execução de uma tarefa, o

servidor atualiza a fila de espera e quer fazer a seleção de tarefas a ser executadas. Entretanto, outro cliente efetua outro pedido de execução de uma tarefa e insere uma nova tarefa na fila. A segunda tarefa externa terá a mesma prioridade que a primeira, uma vez que a seleção ainda não foi efetuada. Outra questão problemática está relacionada com a seleção das tarefas realizada pela thread do segundo cliente, que caso seja realizada em simultâneo com a seleção da thread do primeiro cliente, poderão ser selecionadas tarefas repetidas para serem executadas. Dito isto, decidimos implementar um lock à adição de uma mensagem à fila de espera e à consequente seleção de tarefas para execução.

● Consultar memória disponível

Caso o cliente queira consultar a memória disponível, o servidor responderá com uma mensagem com o valor correspondente. Identificamos a consulta de memória como uma zona crítica para evitar situações em que o servidor recebe uma thread com um pedido de execução, seguida de uma thread com um pedido de consulta de memória, e responde ao pedido de consulta de memória com um valor que não reflete a atualização feita no pedido de execução de uma nova tarefa. A implementação de locks nesta zona permite que a consulta de memória aguarde que a atualização da disponibilidade termine, e vice-versa.

● Consultar tarefas em fila de espera

Finalmente, o cliente pode ainda consultar as tarefas que se encontram em fila de espera. O servidor irá verificar o tamanho atual da WaitingList e responder com uma mensagem indicando este valor. À semelhança da situação anterior, esta é uma zona crítica. A implementação de locks nesta zona permite que a consulta do tamanho da fila de espera aguarde que a atualização da mesma termine (aquando da adição de um novo pedido de execução), e vice-versa.

Conclusão

Este trabalho permitiu-nos compreender a necessidade da utilização de locks e conditions, de forma a potenciar o desempenho de um programa, mantendo a sua correção. Deu-nos a oportunidade de interligar a maioria dos conceitos abordados nas aulas, interiorizando-os e permitindo uma reflexão mais profunda sobre os mesmos.

A parte do trabalho desenvolvido que realizamos foi de acordo com os objetivos pretendidos. No entanto, por motivos de gestão de tempo, não conseguimos proceder à realização da implementação distribuída do projeto.