Universidade do Minho

Mestrado Integrado em Engenharia Informática

Sistemas Distribuídos

Alocação de servidores na nuvem

Fábio :: A78508

Francisco :: A79821

Raphael :: A78848

Joel :: A79068

Grupo 12

Joel Rodrigues A79068

Raphael Oliveira A78848

Fábio Araújo A78508

Francisco Araújo A79821

6 de Janeiro de 2019

Conteúdo

1 Introdução

Neste trabalho prático, pretende-se desenvolver um serviço de alocação de servidores na nuvem, tal como acontece na *Google Cloud* ou na *Amazon EC2*, em que são permitidas reservas e utilização de servidores virtuais para processamento e armazenamento. Os vários tipos de servidores disponibilizados por estas plataformas são identificados por um nome, correspondendo a uma configuração de *hardware* e a uma taxa fixa (e.g., 1€/hora).

As reservas dos servidores podem ocorrer de duas maneiras: **reservas a pedido** ou **reservas a leilão**. De uma forma simplificada, na reserva a pedido, um servidor fica atribuído a um certo utilizador, até este mesmo o quiser libertar, sendo cobrada a taxa fixa, pelo tempo utilizado. Na reserva a leilão, um utilizador propõe um preço que está disposto a pagar pela reserva de um determinado tipo de servidor. Caso existam servidores disponíveis desse tipo, esta atribuição é imediata, caso contrário, só é atribuído o servidor quando a sua licitação é a maior entre as restantes para aquele tipo de servidor. Ainda, um servidor atribuído a um utilizador por este método de reserva pode ser realocado para outro utilizador, caso este novo utilizador tenha realizado uma reserva a pedido e não hajam mais servidores disponíveis deste mesmo tipo.

Para além destas principais funcionalidades, esta aplicação terá de permitir também o registo e autenticação de utilizadores, a libertação deliberada de um tipo de servidor, por parte de um certo utilizador, e, ainda, o consultar da conta corrente por parte dos utilizadores.

Por último, a construção desta aplicação é realizada à base da comunicação Cliente-Servidor, escritos em *Java*, através de *sockets TCP*, disponibilizando uma interface no Cliente que permita ao utilizador usufruir das funcionalidades descritas acima, enquanto que o Servidor mantém em memória a informação relevante para suportar essas mesmas funcionalidades, bem como fazer chegar aos utilizadores a informação pretendida.

2 Arquitetura da aplicação

2.1 Comunicação Cliente-Servidor

Nesta comunicação, existe um ***Servidor*** sempre à escuta na porta 12345, à espera de possíveis conexões com clientes. Conseguida uma conexão, inicia a *thread* ***TratarCliente***, a qual irá manipular os pedidos desse novo cliente, permitindo assim, ao servidor principal, múltiplas conexões, repetindo o mesmo processo de escuta a cada conexão obtida.

Esta *thread TratarCliente* é inicializada com o *socket* obtido, permitindo estabelecer conexão com o próprio cliente, lendo as suas mensagens, correspondentes a funcionalidades pretendidas, e escrevendo para o cliente uma determinada mensagem que irá determinar a ação do mesmo. Ainda, é inicializada com o ***ServidorSkeleton***, que por sua vez é inicializado com as estruturas utilizadas em todo o sistema: a classe ***Utilizadores*** e classe ***ServidoresCloud***. Desta forma, a thread *TratarCliente* irá apenas receber mensagens enviadas pelo cliente, chamando o método respetivo à funcionalidade

pretendida pelo mesmo, presente na classe *ServidorSkeleton*, o qual utilizará os métodos presentes nas classes *Utilizadores* e *ServidoresCloud* para efetuar o pretendido. Realizado o tal método, o *output* desse método é lido pela *thread TratarCliente*, a qual é responsável por transmitir ao cliente.

Na verdade, esta *thread* não comunica propriamente com o ***Cliente***, mas sim com um ***ServidorProxy*** inicializado nesta mesma classe Cliente. Esta classe *ServidorProxy* também é inicializada com o socket e com os *buffers* de leitura e escrita desse socket, por parte do cliente. Assim, a cada opção pretendida pelo cliente, utiliza os métodos presentes na classe *ServidorProxy*.

Neste seguimento, é possível verificar dados bem encapsulados e as classes *Cliente* e *TratarCliente* não precisam de possuir conhecimentos para além do nível de comunicação por *sockets*, só precisando implementar interfaces disponibilizadas pelas classes *ServidorProxy* e *ServidorSkeleton*, para efetuar as funcionalidades pretendidas.

Assim, existem duas fases distintas nesta comunicação: **Fase Pré-Autenticação** e **Fase Pós-Autenticação**, as quais serão descritas sucintamente de seguida.

2.1.1 Fase Pré-Autenticação

Como dito anteriormente, sempre que um utilizador desejar interagir com o serviço deverá ser autenticado pelo servidor.

Desta forma, numa fase inicial, são apenas apresentadas as seguintes opções a cada cliente: Registar, *Login* e Sair. Para cada opção tomada, é utilizado um método presente na classe *ServidorProxy*, comunicando com a *thread TratarCliente*, sendo apresentadas as mensagens de seguida (método Registar):

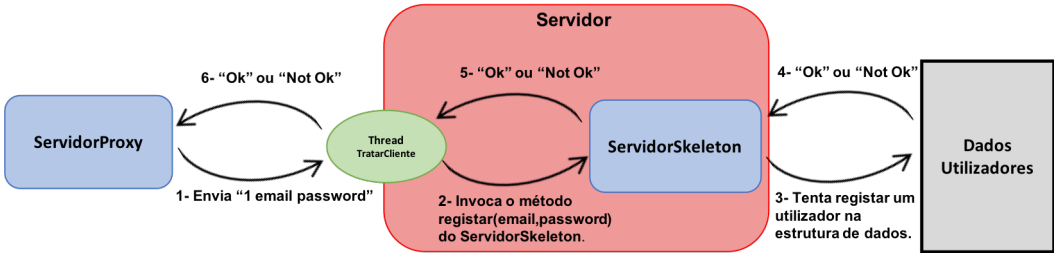

Figura 1: Mensagens intermédias no método *Registar*

Para as duas primeiras opções, caso o retorno lido pelo *ServidorProxy* seja "Not Ok", o cliente terá de voltar a repetir o processo de registo ou autenticação. Caso contrário, o cliente consegue seguir para a próxima fase, ou seja, a fase principal da aplicação.

2.1.2 Fase Pós-Autenticação

Passada a fase inicial, o cliente pode requisitar as principais funcionalidades do sistema (**algumas adicionadas pelo grupo**) :

- **Consultar saldo**: O cliente pode consultar o seu saldo disponível.

- **Consultar Servidores**: O cliente pode consultar o número de servidores disponíveis, separados por cada tipo. Ainda, consegue consultar o número de servidores atribuídos por leilão, o que pode influenciar o seu tipo de reserva, a sua taxa fixa e licitação mínima (caso queira reservar por leilão).

- **Consultar Propostas**: O cliente pode consultar as propostas para um determinado tipo de servidor, efetuadas por outros clientes, podendo assim avaliar as suas licitações, para determinar melhor a sua própria licitação.

- **Consultar minhas reservas**: O cliente pode consultar as suas reservas atuais, sendo visado todos os tipos de servidores bem como os seus respetivos *ids* de reserva. Aqui, também pode **terminar reservas**, inserindo o *id* de reserva de qual pretende terminar.

- **Depositar montante**: O cliente pode depositar um certo montante na sua conta corrente.

- **Efetuar Reserva**: O cliente pode efetuar dois tipos de reserva: leilão e pedido. Caso seja a pedido, só tem de inserir o tipo de servidor, caso seja o outro tipo de reserva também tem de inserir a sua licitação.

- **Consultar notificações**: O cliente pode consultar as suas notificações. Aqui são visadas mensagens quando lhe é atribuído um servidor para uma licitação que tinha sido registada anteriormente ou quando lhe é retirado um servidor, caso tenha sido reservado em leilão e houve uma reserva a pedido (com os servidores desse tipo todos indisponíveis), apresentando também o *timestamp* dessas ações.

- **Sair**: O cliente pode sair da aplicação. Caso tenha reservas ou propostas registadas, as mesmas continuam registadas.

Cada opção das demonstradas anteriormente irá despoletar uma troca de mensagens entre o cliente e o servidor, tais como os exemplos demonstrados na fase anterior, mas com uma maior complexidade.

3 Funcionalidades do Servidor

Tal como dito anteriormente, para conseguir preencher todos os requisitos do sistema, a classe *ServidorSkeleton*, classe que tem acesso aos dados, possui armazenado em memória todos os dados relativos aos utilizadores bem como todos os dados relativos aos servidores da nuvem. Ainda, foram implementadas mais duas *threads* que possuem este acesso aos dados:

- ***AtribuirServidores***: Esta *thread* é iniciada logo após o primeiro registo de cada tipo de servidor da nuvem. Com isto, irá existir uma *thread* destas por cada tipo de servidor, cuja função é verificar se existem servidores desse tipo disponíveis, bem como registo de propostas (tentativas de reserva de leilão) para esse tipo de servidores. Quando os dois casos se verificarem, atribui um servidor a um certo utilizador que registou essa proposta.
- ***DescontaSaldo***: Esta implementação deve-se ao facto do grupo definir que o saldo do cliente iria ser descontado ao longo da reserva de um determinado servidor. Assim, definiu-se um tempo limite para simular um tempo real (por exemplo, 5 segundos corresponde a 1 hora), e, ao fim desse tempo, esta *thread* desconta, no saldo atual do utilizador, o valor da taxa fixa, no caso de ser reserva a pedido, ou a taxa licitada, no caso de ser reserva a leilão. Caso o valor do saldo chegue a zero, esta também tem a função de libertar o servidor e simultaneamente retirar a respetiva reserva.

Assim, pode-se verificar a arquitetura final do sistema, no seguinte esquema:

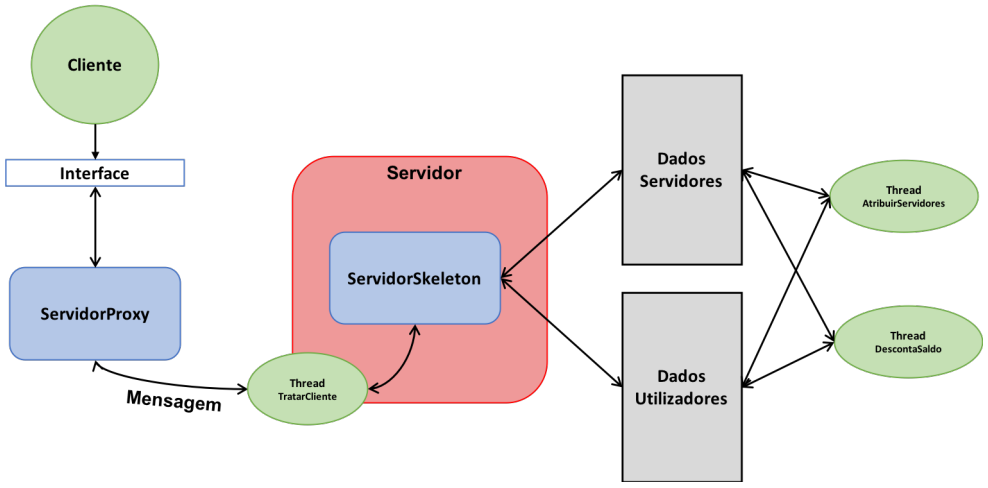

Figura 2: Arquitetura final do sistema

4 Controlo de concorrência

Através da utilização de *threads*, os mesmos dados podem ser acedidos e/ou alterados no mesmo instante, o que resulta em dados incoerentes. Para isso, foi necessário utilizar mecanismos de exclusão mútua, por exemplo, os *ReentrantLocks* ou o mecanismo *synchronized*.

Além dos mecanismos de exclusão mútua implementados nas funcionalidades relativas à classe *Utilizadores*, o grupo encontrou bastante mais dificuldade nos mecanismos implementados na classe *ServidoresCloud*.

Anteriormente, esta classe apresentava duas estruturas: um *HashMap* que guardava todos os servidores da nuvem, sendo a chave o tipo do servidor e o valor uma lista de servidores, e outro *HashMap* que guardava todas as propostas, sendo a chave um tipo de servidor e o valor uma lista de propostas.

Verificando que era muito complicado utilizar *ReentrantLock*, simultaneamente nessas duas estruturas, implementou-se uma nova classe ***Informacao***. Esta classe apresenta duas listas (uma de servidores e outra de propostas), sendo utilizada na classe *ServidoresCloud*, num *HashMap<String,Informacao>*.

4.1 Utilização dos *ReentrantLocks*

Tal como na classe Utilizadores, nesta classe a utilização de *ReentrantLocks* baseou-se em: obter o *lock* da classe em que me situo, obter um *lock* de uma classe mais aninhada (ex: ***lockInformacao***), libertar o *lock* obtido anteriormente e, por último, libertar o *lock* da classe mais aninhada. Para este caso concreto, por exemplo, uma reserva para um tipo de um servidor pode ocorrer ao mesmo tempo que outra reserva de outro tipo de servidor.

Neste seguimento, a utilização deste mecanismo permite muito maior flexibilidade do que o mecanismo *synchronized*.

4.2 Atribuição de Servidores a Propostas

Esta funcionalidade caraterizou-se por utilizar vários mecanismos de exclusão mútua: Como explicado anteriormente, esta funcionalidade está presente na *thread AtribuirServidores*, a qual está sempre à procura de servidores disponíveis ou propostas existentes para respetiva atribuição. Para esta funcionalidade não resultar numa espera ativa, o que iria consumir demasiado *CPU*, utilizou-se uma condição (***not_Servidores_or_Propostas***) no *ReentrantLock lockInformacao*, da classe *Informacao*.

Assim, neste método apresentado de seguida, para um dado tipo de servidor, verifica se existem servidores disponíveis e propostas. Caso não preencha um dos requisitos, este *lockInformacao* é libertado para outras tarefas, enquanto que esta *thread* fica "adormecida"naquela variável de condição. Quando se liberta um servidor (***informacao.servidores_disponiveis++***) ou quando se insere uma proposta, esta thread é "acordada", tentando outra vez adquirir o *lock* anterior.

Este processo é repetido até existirem servidores disponíveis e propostas para as respetivas atribuições.

```
atribuirServidoresPropostas(String nomeServidor){
    this.lock.lock();
    Informacao informacao = this.informacao.get(nomeServidor);
    informacao.lockInformacao.lock();
    this.lock.unlock();
    try{
        while((informacao.servidores_disponiveis==0) ||
                                    (informacao.propostas.size()==0)){
            informacao.not_Servidores_or_Propostas.await();
        }
        //atualizaInformacao(informacao);
    }finally{
        informacao.lockInformacao.unlock();
    }
}
```

5 Extras

Ao longo do desenvolvimento do sistema foram implementadas algumas funcionalidades para além daquelas requeridas no enunciado do trabalho e descritas na secção 1.

Por um lado, de forma a melhorar a interação dos utilizadores com a aplicação foi elaborada uma interface gráfica utilizando a API *JFrame* do Java (exemplos na figura 3), bem como algumas funcionalidades, como por exemplo, depositar um determinado montante, consultar as propostas existentes ou servidores disponíveis até ao momento e aceder às notificações do utilizador.

Por outro lado, para verificar o funcionamento do sistema, foram desenvolvidos ***Bots*** que simulam as interações de vários utilizadores com o sistema, permitindo assim testar o controlo de concorrência da aplicação e a possível ocorrência de erros.

Figura 3: Parte da interface implementada

6 Conclusão

Para concluir, todas as funcionalidades propostas inicialmente foram concluídas e bem implementadas. Esta boa implementação deveu-se sobretudo ao bom planeamento de todas as classes e possíveis mudanças das mesmas, bem como a arquitetura de todo o sistema.

Contudo, existiram alguns contratempos, tais como o controlo de concorrência, maioritariamente na classe *ServidoresCloud*, devido à necessidade de gestão da concorrência de várias *threads* às mesmas estruturas de dados, o que proporcionou um maior dispêndio de tempo nestas tarefas.

Mesmo assim, o grupo ainda conseguiu implementar uma interface gráfica, o que melhora a interação entre os utilizadores e aplicação, eliminando logo alguns erros à partida, que poderiam ter sido introduzidos pelo utilizador, podendo também conferir novas funcionalidades, que não tinham sido propostas inicialmente.

Em suma, revelou-se um trabalho proveitoso, tendo sido aplicados várias matérias em conjunto, lecionadas nas aulas desta Unidade Curricular e, por outro lado, possibilitou que puséssemos em prática estas matérias consideradas complexas no paradigma de programação concorrente, melhorando assim o conhecimento acerca das mesmas.