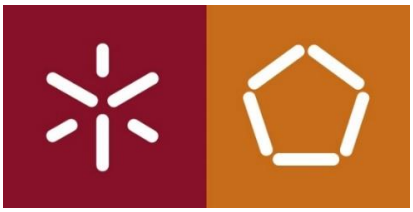
UNIVERSIDADE DO MINHO

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA



Alocação de servidores na nuvem

Sistemas Distribuídos
Grupo 24

A81712
Ana Pereira

A82474
Ana Ribeiro

A80564
Carla Cruz

A82061
Jéssica Lemos

Índice

1 INTRODUÇÃO 3

2 REALIZAÇÃO DO PROJETO 3

2.1 SERVIDORMAIN 3
2.2 CLIENTEMAIN 3
2.3 MENU 3
2.4 UTILIZADOR 4
2.5 SERVIDOR 4
2.6 RESERVA 4
2.7 LEILAO E LANCE 4
2.8 SERVERCLOUD 4

3 CONTROLO DE CONCORRÊNCIA 5

4 CONCLUSÃO 6

1 Introdução

É de conhecimento geral que existem plataformas de computação na nuvem, tais como o Google Cloud e Amazon EC2. Estas permitem reservar e usar servidores virtuais para processamento e armazenamento.

Assim, no âmbito da unidade curricular de Sistema Distribuídos, foi proposto o desenvolvimento de um serviço de alocação de servidores na nuvem e de contabilização do custo incorrido pelos utilizadores.

Neste trabalho pretende-se incluir todos os conhecimentos adquiridos ao longo das aulas, nomeadamente, criação de threads, controlo de concorrência e a conceção de clientes e servidores.

Neste relatório iremos explicar as decisões tomadas de modo a solucionar o problema apresentado.

2 Realização do Projeto

Para este projeto decidimos elaborar uma solução que tem por base as mensagens. Desta forma, a comunicação é feita através de Strings com significados específicos.

Nesta secção iremos abordar a implementação das classes fundamentais do nosso projeto que permitem o correto funcionamento da aplicação.

2.1 ServidorMain

Nesta classe tornou-se imperativo a criação de duas threads, a **ServidorReader** e a **ServidorWriter**, relativas ao Servidor. É através destas que o Servidor irá ler a informação do socket e dar respostas ao Cliente através do socket, utilizando a thread de leitura e de escrita respetivamente. Como tal, a **ServidorReader** recebe o input do cliente e interpreta o pedido deste, de modo a invocar os métodos que se encontram na classe principal, ou seja, na **ServerCloud**. Esta escreve o resultado desses métodos na **MensagemBuffer** do cliente, de modo a que **ServidorWriter** leia do buffer e envie para o socket do cliente.

2.2 ClienteMain

As classes **ClienteReader** e **ClienteWriter** dizem respeito às duas threads do Cliente, que foram criadas nesta main, e é através destas que o Cliente irá ler a informação do socket com a respetiva thread de leitura e enviará através da thread de escrita a sua resposta, utilizando o socket para que seja possível a informação chegar ao servidor. Desta forma, a **ClienteReader** recebe o output do socket do cliente e interpreta o seu conteúdo de modo a invocar os métodos necessários. Enquanto que a **ClienteWriter** escreve no socket as opções escolhidas pelo mesmo. No caso de escolha de opções inválidas o cliente será informado de tal através de mensagens de erro.

2.3 Menu

Esta classe tem como funcionalidade apresentar a interface ao utilizador. Esta permite a comunicação entre o Cliente e o Servidor. Nesta será feita a leitura da opção do menu

selecionada pelo utilizador, garantido que esta seja válida. Esta contém alguns métodos para a recolha de input do mesmo, pelo que eventos que impliquem a mudança de menu irão ter esta classe como auxílio.

2.4 Utilizador

Esta classe contém a informação de cada utilizador, nomeadamente, o email, a password bem como um buffer que permite guardar as mensagens enviadas para este. Para tal, houve a necessidade da criação da **MensagemBuffer** que suporta concorrência. Esta informação irá ser útil para a realização do login e no momento de registar novos utilizadores, sendo possível guardar a informação dada.

2.5 Servidor

Esta classe contém as informações relativas aos servidores que podem ser reservados. Para tal, guardamos o nome, o tipo, o preço bem como o estado que indica se este se encontra disponível ou reservado em leilão ou em pedido.

2.6 Reserva

Com o intuito de conceder reservas, quer a pedido quer em leilão, tornou-se imperativo a criação de uma classe que armazenasse toda a informação relativa a esta, nomeadamente, o servidor reservado, o cliente que efetuou a reserva, o identificador da mesma, a data da realização bem como uma variável que indica se esta reserva ainda não foi cancelada e a data em que tal se sucedeu.

2.7 Leilao e Lance

Quando um cliente pretender fazer uma licitação de um servidor em leilão este fará um lance, o que nos conduziu à elaboração da classe **Lance** que contém o autor da proposta e o valor que está disposto a oferecer.

Existem leilões para os diferentes tipo de servidores, que neste caso são dois, os servidores pequenos e os servidores grandes. De modo, a conhecer o cliente que efetuou a licitação mais alta, armazenamos na classe **Leilao** o lance respetivo e ainda uma lista de licitadores. Deste modo, aquando do encerramento do leilão é possível informar o vencedor.

Na eventualidade de existirem servidores livres no momento da licitação um servidor será imediatamente atribuído ao cliente. Caso contrário, os clientes vão efetuando as suas propostas até ao encerramento do leilão, que ocorre quando um servidor fica livre.

2.8 ServerCloud

A **ServerCloud** é a classe fundamental do nosso projeto dado que armazena toda a informação referente aos utilizadores, os servidores, às reservas e aos leilões existentes.

Nesta encontram-se os métodos essenciais que permitem ao utilizador usufruir de todas as funcionalidades da nossa aplicação. Quando o **ServidorReader** invoca o método de registar sessão (registar) esta verifica se o utilizador já existe e caso tal se suceda informa o utilizador. Se não, o utilizador é registado, adicionando-o aos utilizadores armazenados. Note-se que, neste processo foi necessário dar *lock* aos utilizadores.

3 Controlo de concorrência

Como na elaboração deste projeto recorremos a threads tornou-se imperativo implementar determinadas técnicas de exclusão mútua que evitam *deadlocks* e mantém a coerência dos dados.

É realizado controlo de concorrência principalmente na classe **ServerCloud**. Nesta existem quatro *locks*, referentes ao utilizador, ao servidor, à reserva e ao leilão. Por exemplo, para o caso do *lock* do utilizador este é acionado aquando do registo bem como no iniciar sessão. Relativamente ao registo e iniciar sessão, tal revela-se essencial de modo a não registarmos dois utilizadores com o mesmo email ou permitir o login de um mesmo utilizador simultaneamente.

```
public Utilizador iniciarSessao(String email, String password, MensagemBuffer msg) throws PedidoInvalidoException {
    Utilizador u;
    utilizadorLock.lock();
    try {
        u = utilizadores.get(email);
        if (u == null || !u.verificaPassword(password)) throw new PedidoInvalidoException("Dados incorretos");
        else u.setNotificacoes(msg);
    } finally {
        utilizadorLock.unlock();
    }

    return u;
}

public void registar(String email, String password) throws UtilizadorExistenteException {
    utilizadorLock.lock();
    try {
        if (utilizadores.containsKey(email))
            throw new UtilizadorExistenteException("O email já existe");
        else utilizadores.put(email, new Utilizador(email, password));
    } finally {
        utilizadorLock.unlock();
    }
}
```

Figura 1- Registar e iniciar sessão na ServerCloud

No caso do **MensagemBuffer** visto que várias threads acedem a este simultaneamente foi necessário implementar um mecanismo de exclusão mútua implícita, o *synchronized*, que garante o controlo de concorrência. Outra das classes em que foi essencial implementar este mecanismo foi no **Leilao** para realizar uma proposta e para terminar um leilão.

4 Conclusão

Este projeto baseou-se na comunicação entre cliente – servidor e controlo de concorrência. Assim sendo, ao longo da realização do projeto tivemos em atenção a anulação de eventuais deadlocks. Este tornou-se um dos aspetos mais difíceis de implementar dado que é necessário garantir a exclusão mútua em regiões críticas. Tal é permitido através da utilização de variáveis de condição e locks. De modo a promover a concorrência recorremos threads.

Apesar de todas as dificuldades encontradas, consideramos que os objetivos propostos foram alcançados. Assim sendo, é possível utilizar o sistema implementado para reservar servidores a pedido ou em leilão.