Universidade do Minho

Sistemas Distribuídos - Trabalho Prático

Alocação de Servidores na Nuvem

Francisco Oliveira, a78416
Gonçalo Camaz, a76861
Raul Vilas Boas, a79617
Vitor Peixoto, a79175

6 de Janeiro de 2019

Conteúdo

1 Introdução

Com o aumento da quantidade de dados nos dispositivos tecnológicos e com a facilidade de acesso aos serviços que a rede nos fornece, o espaço que estes dispositivos nos fornecem torna-se pequeno e a necessidade de presença dos dados nos diversos dispositivos (PC, tablet, smartphone, etc.) leva a um crescimento exponencial de serviços na *nuvem*.

Assim sendo, foi-nos proposto construir um sistema de alocação de servidores na nuvem capaz de contabilizar o custo incorrido aos utilizadores. Este sistema deverá ter algumas funcionalidades como o aluguer de servidores por leilão ou por compra.

O sistema deverá também aplicar os conhecimentos adquiridos na unidade curricular de Sistemas Distribuídos, principalmente na utilização de *sockets* para a ligação entre servidor e clientes.

2 Desenvolvimento da Solução

2.1 Cliente

O sistema a ser desenvolvido deverá possuir um método de autenticação através do seu email e palavra-passe associada a cada cliente previamente registado no sistema.

No entanto, antes de se permitir a conexão para os clientes, é necessário estabelecer uma conexão ao servidor.

Com esse intuito, criou-se a classe `ClienteConnection` que permite ao utilizador estabelecer uma conexão ao servidor. Toda a interação Cliente-Servidor é facilitada por uma interface desenvolvida especialmente para este sistema, a ser explicada mais à frente.

```
// Connection managment /////
public String connect(String email, String password){
    try {
        this.socket = new Socket("127.0.0.1", 1234);
        // prepare all the streams
        this.out = new PrintWriter(socket.getOutputStream());
        this.in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        // autenticao
        String status = autenticacao(email, password);
        return status;
    }
    catch (IOException e) {
        System.out.println("[ClienteCon] Connect - RIP Socket !!!");
        System.out.println(e);
        return "FAIL";
    }
}
```

Figura 1: Método `connect()` da classe `ClienteConnection`.

No bloco de código acima, vemos o método `connect()` referente à ligação entre utilizador e servidor. Neste método é efetuada a criação do *socket*, bem como os respetivos *streams* para efetuar a comunicação. De seguida, o cliente efetua a ligação ao *socket* após autenticado. Caso a autenticação falhe, a ligação é fechada pelo servidor, caso contrário, o servidor devolve ao utilizador a informação relativa a si mesmo e todos os servidores existentes através dessa ligação, podendo agora o cliente efetuar pedidos ao servidor.

2.2 Servidor

O servidor cria um *socket* que escuta por tentativas de ligação de qualquer cliente. Após aceitar uma ligação, esse servidor cria uma *thread* denominada

`ServidorWorker`, que se responsabiliza por toda a interação com o cliente que estabeleceu ligação ao servidor.

Isto permite que o servidor fique livre dessa carga e volte imediatamente à escuta de mais tentativas de ligação.

Podemos ver na imagem abaixo o bloco de código responsável pela escuta de clientes e criação da *thread*:

```
public void run() {
    System.out.println("[Servidor] Iniciando o Servidor");
    while(this.running.get()){
        System.out.println("[Servidor] Servidor à escuta na porta " +
                           this.serverSocket.getLocalSocketAddress());
        try{
            this.clienteSocket = this.serverSocket.accept();
        }
        catch(IOException e){
            System.out.println("[Servidor] Erro a aceitar ligação do cliente");
            System.out.println(e);
        }
        try{
            // cria uma nova thread (worker) para a conexao
            new MainServidorWorker(clienteSocket,bd).start();
        }
        catch(IOException e){
            System.out.println("[Servidor] Erro a criar Thread para o cliente");
            System.out.println(e);
        }

    }
}
```

Figura 2: Método `run()`.

2.2.1 ServidorWorker

Cada *thread* `ServidorWorker` é responsável por toda a interação entre o servidor e um único cliente.

Primeiramente certifica-se que apenas utilizadores autenticados podem fazer pedidos. Para tal, após ser iniciado, o *Worker* espera a autenticação do utilizador. Qualquer tentativa incorreta resulta no fim da ligação, devendo o utilizador voltar a tentar autenticar-se.

Verificada a autenticação, o *Worker* agora espera pedidos do utilizador. Estes podem variar entre pedidos de servidores (normais ou leilão), libertar servidores seus, pedir atualização de informação ou terminar a ligação.

Para receber estes pedidos o *Worker* encontra-se num ciclo *while* à espera de algum pedido no *socket*, vindo do utilizador.

Recebido o pedido, o *Worker* processa-o pelo `parse()`. Este método realiza as operações necessárias para efetuar o pedido e devolve uma resposta com o sucesso ou falha do pedido através do *socket* de volta ao utilizador como forma de *feedback*. Em caso de sucesso, um pedido de atualização de informação retorna também toda a informação pedida (informação do utilizador e servidores disponíveis) usando um *stream* de objetos pelo *socket*.

2.2.2 Base de Dados

Esta classe é onde a informação é guardada e onde se encontram os métodos de acesso. Contém um *Map* com todos os utilizadores (da classe `User`), caraterizados por variáveis de email, palavra-passe e servidores que tem alocados.

Tem também um *Map* com todos os servidores. A classe `Server`, tem toda a informação sobre um servidor, nomeadamente: tipo, preço e informação sobre reservas.

Esta classe contém também todos os métodos necessários para marcar o servidor como utilizado quando existe um pedido de reserva, licitação ou libertação por pedido de remoção do servidor.

```
public void lockAllUsers(){
    this.users.values().stream().forEach((u) -> {
        u.lock();
    });
}
public void unlockAllUsers(){
    this.users.values().stream().forEach((u) -> {
        u.unlock();
    });
}
```

Figura 3: Métodos que bloqueiam/desbloqueiam o acesso a todos os utilizadores.

```
// LOCK AND UNLOCK METHODS //
public void lock() {
    this.lockUser.lock();
}
public void unlock() {
    this.lockUser.unlock();
}
```

Figura 4: Métodos para bloquear/desbloquear o acesso a um utilizador.

Na figura acima encontrámos o nosso método `lock()`, que tem como função bloquear o acesso a uma instância de `User`. Isto permite evitar acessos concorrentes e possível corrupção ou sobreposição de escritas e leituras a memória. Este método permite-nos também, na classe `BaseDados`, bloquear todos os `Users`, quando tal é necessário.

Existe também um método `lock()` na classe `Server`, com o mesmo propósito.

2.3 Interface

Para facilitar o uso do sistema desenvolvido, as interações entre utilizador e sistema são efetuadas através de uma interface gráfica com auxílio da biblioteca para JAVA *Swing*, fornecida pelo IDE *Netbeans*.

Inicialmente é apresentada uma interface para o utilizador se autenticar, usando as suas credenciais (email e palavra-passe).

Em caso de falha, poderá tentar novamente. Em caso de sucesso, é então levado ao menu onde poderá fazer diversos pedidos, como reservar ou licitar um servidor que esteja disponível.

Pode também libertar qualquer servidor que tenha alocado para si, visíveis no separador *My Servers*.

Para além disto, também é informado dos custos relativos à aquisição dos servidores, custo por hora de todos os servidores alocados no momento e divida atual a pagar.

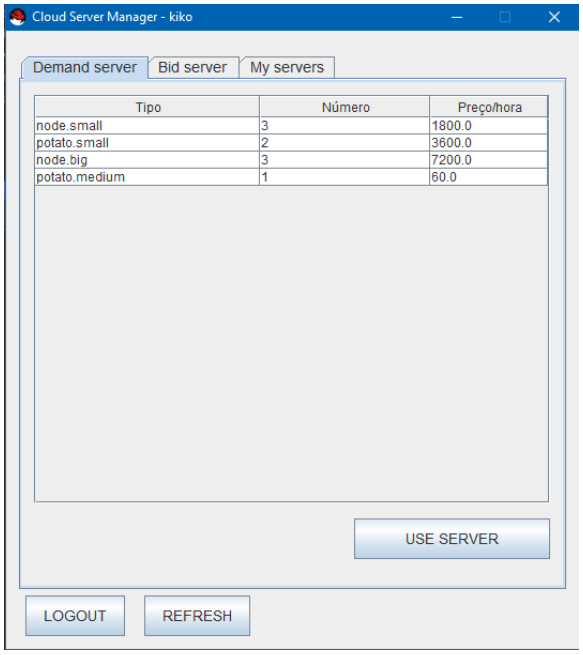

Figura 5: Separador para reserva de um servidor.

3 Conclusão

O grupo conseguiu implementar com sucesso o estabelecimento de comunicação entre cliente e servidor através de *sockets TCP*, garantindo também controlo de concorrência no que diz respeito ao acesso dos servidores que estavam disponíveis para aquisição, tanto através da compra como também através da licitação.

A nível de trabalho futuro poderíamos ter ainda no servidor uma lista de reservas, equivalente a uma fila de espera. Os clientes eram inseridos por ordem de chegada e assim que um servidor ficasse livre, o que chegou primeiro seria notificado de que haveria um servidor disponível, efetuando então a aquisição do mesmo caso ainda estivesse interessado. Finalmente, sempre que um cliente fosse perder o servidor (no caso de o ter adquirido através de um leilão), deveria ser notificado sendo que poderia-mos dar-lhe depois a opção de aumentar a licitação que efetuou para continuar a utilizar o servidor, ou então, perder o servidor para outro cliente.