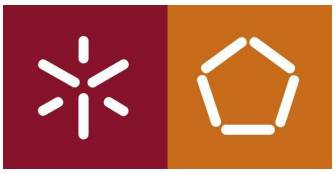
Universidade do Minho

Mestrado Integrado em Engenharia Informática

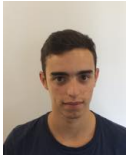


Alocação de Servidores na Nuvem

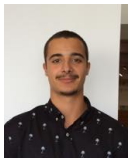
Sistemas Distribuídos

Grupo 54

Diogo Sobral (a82523)



Henrique Pereira (a80261)



Pedro Moreira (a82364)

Pedro Ferreira (a81135)

2018/2019

Conteúdo

1 Introdução

Este trabalho prático foi realizado no âmbito da Unidade Curricular de Sistemas Distribuídos e tinha como propósito o desenvolvimento de um serviço de alocação de servidores na nuvem. Além, disso, o custo incorrido pelos utilizadores deste serviço deveria ser calculado e contabilizado.

Ora, este sistema, segundo o enunciado, deveria permitir o aluguer (reserva) de servidores, quer por "compra direta"pelo preço fixo do *server*, quer por licitação, sendo estes atribuídos às melhores ofertas. Estando todos os servidores reservados, e haja uma proposta de compra direta, um servidor que tenha sido alugado por leilão é retirado ao atual dono e atribuído ao utilizador que fez a proposta de compra. O preço do servidor é horário, ou seja, os utilizadores pagam por um *server* consoante o tempo que ficam com ele, podendo libertá-los a qualquer momento. Outras funcionalidades existentes no sistema são o registo e autenticação do cliente, bem como a consulta das "dívidas"deste. O preço de cada servidor é atribuído consoante o tipo deste. Para este trabalho, decidimos criar os seguintes tipos de *server*, com os respetivos preços:

- *fast*, custando cada um 10 u.m./hora e existindo três servidores deste tipo: `fast.com`, `fast.net` e `fast.org`
- *medium*, custando cada um 7.5 u.m./hora e existindo três servidores deste tipo: `medium.com`, `medium.net` e `medium.org`
- *large*, custando cada um 5 u.m./hora e existindo três servidores deste tipo: `large.com`, `large.net` e `large.org`

O sistema teria que ser implementado em `JAVA`, recorrendo para tal a *threads* e *sockets* TCP.

Assim sendo, vamos explicar como definimos o sistema nas secções seguintes.

2 Descrição das classes implementadas

2.1 Relacionadas com o Cliente

- **Client -** Classe onde se efetua a ligação do cliente ao servidor e se dá início à execução de duas *threads*: uma associada a um objeto da classe Drawer e outra associada a um objeto da classe `Reader`.
- **Drawer -** Classe responsável pela implementação da interface apresentada ao utilizador, assim como pelo envio ao servidor das mensagens representativas dos *inputs* do utilizador. Essas mensagens são criadas nesta classe, concatenando aos *inputs* reconhecidos pelo servidor dados representativos das escolhas feitas pelo utilizador. Após o envio de uma mensagem, realizado através do método `server-request`, a *thread* bloqueia à espera da resposta, de forma a garantir que o estado que a interface deve apresentar é correto e que não se altera antes da resposta do servidor ter chegado. O método responsável pelo bloqueio encontra-se na classe `Log`,

que de seguida se descreve. Após o envio da resposta do servidor, o objeto da classe `Reader` imprime a resposta para o *stdout* e acorda a *thread* associada ao objeto do tipo `Drawer`, que de imediato atualiza o estado da interface.

- **Reader -** Classe responsável pela leitura das respostas enviadas pelo servidor, imprimindo-as para o *stdout*. Para cada mensagem recebida, verifica se a *thread* do objeto `Drawer` se encontra adormecida e se tal se verificar acorda-a.

- **Log -** Classe responsável por possibilitar a comunicação entre `Drawer` e `Reader`, permintindo saber se o *login* do cliente foi bem sucedido, se este se encontra à espera de resposta do servidor e se este efetuou *logout*. Em suma, é representativa das operações realizadas pelo utilizador no decorrer da interação com a aplicação.

2.2 Relacionadas com o Servidor

- **Server -** Classe responsável pela criação do servidor de *sockets* a usar. Armazena também os dados relativos aos utilizadores e aos servidores que podem ser reservados. O servidor criado é *multi-threaded*, impedindo que um cliente lento afete os restantes. Isto quer dizer que é uma criada uma nova *thread*, associada a um objeto da classe `ClienteHandler`, por cada conexão ao servidor. Os dados guardados nesta classe são partilhados pelas diferentes *threads*. A criação dos servidores disponíveis para reserva é efetuada nesta classe. Cada grupo de servidores do mesmo tipo é gerido por um objeto da classe `ServerTypeManager` (onde se encontra o controlo de concorrência para operações sob o grupo de servidores do mesmo tipo).

- **ServerTypeManager -** Classe que gere um grupo de servidores do mesmo tipo. Implementa as operações de compra de servidor, licitação e libertação de um servidor adquirido. Os servidores livres são mantidos numa lista ligada. As licitações são organizadas numa *queue* ordenada por ordem decrescente de valor de licitação.

- **Servidor -** Classe representativa dos servidores reservados pelos utilizadores. Guarda informação sobre o utilizador que num dado momento possui o servidor, registando a hora a que este foi adquirido, assim como o preço a que foi comprado e se foi comprado por leilão ou não. Desta forma é possível desafetar o servidor de um cliente que o tenha comprado em leilão, caso outro cliente o deseje comprar por preço nominal, assim como determinar o valor a pagar pelo período de utilização gasto, aquando da libertação do servidor.

- **ClientHandler -** Classe responsável por receber as mensagens enviadas pelo cliente ao servidor, processando-as, sendo que a cada conexão realizado é associado um objeto desta classe. Após o *login* ter sido efetuado com sucesso, é guardar informação sobre o utilizador ao qual o objeto

se encontra associado. As mensagens recebidas podem ter as seguintes assinaturas:

  - **LOGIN**: para efetuar o *login* de um utilizador;
  - **SIGN**: para registar um novo utilizador;
  - **LOGOUT**: para terminar a sessão de um utilizador;
  - **MONEY**: para consultar a dívida do utilizador;
  - **OSERVER**: para consultar os servidores na posse do utilizador;
  - **FREE**: para libertar um servidor;
  - **RENT**: para reservar um servidor de um certo tipo pelo seu preço nominal;
  - **BID**: para licitar sobre um determinado tipo de servidores;
  - **MSG**: para obter mensagens enviadas enquanto o utilizador se encontrava *offline*

- **Licitação -** Classe que guarda os dados relativos a uma licitação, identificando o utilizador que a realizou e o preço oferecido.

- **Utilizador -** Classe que guarda os dados relativos a um utilizador, nomeadamente: *username*, *password*, dívida corrente, lista de servidores em sua posse, *status* e caixa de entrada de mensagens. Possui também um *lock* explícito para garantir a consistência dos dados apresentados (ex: se, num dado momento, um utilizador detiver em sua posse um servidor, obtido através de leilão, e este for comprado a preço nominal por outro utilizador, ao consultar a dívida corrente poderia deparar-se com um valor que não correspondia à sua dívida real. Tal poderia acontecer pois a *thread* que se encontra associada ao cliente que decidiu comprar o servidor a preço nominal iria atualizar o valor da dívida do cliente que detinha o servidor anteriormente. Se a consulta de dívida acontecesse simultaneamente à compra, o valor visualizado poderia ser inferior ao real, se não existisse controlo de concorrência).

- **BidCheck -** Classe que serve para alertar o detentor de um servidor comprado a leilão, caso alguém o compre a preço nominal. Aquando da compra a leilão é criada e animada uma *thread* associada a um objeto desta classe, que adormece numa variável de condição (presente no Servidor), até que o utilizador liberte o servidor adquirido, ou então, até que alguém o compre pelo preço nominal, caso em que é enviada uma mensagem, ao utilizador que o comprou em leilão, a indicar o sucedido. Se o utilizador se encontrar *offline* então a mensagem é adicionada à sua caixa de mensagens.

3 Descrição das Operações

Como foi referido na secção anterior, definimos oito operações, relativas às mensagens recebidas pelo Servidor no `ClientHandler`, e que serão explicitadas de seguida.

3.1 LOGIN

Para efetuar *login* no servidor, o cliente deve enviar uma mensagem com o seguinte formato: `LOGIN;username;password`. Após decompor a mensagem através do método `commandLogin`, é invocado o método `logIn`. Este usa um bloco `synchronized` sobre o Map de clientes, que é partilho pelas diversas *threads* associadas a objetos da classe `ClientHandler`. Embora a operação de *login* apenas necessite de ler valores desse Map, é necessário obter o *intrinsic lock* deste na mesma, assim como na operação na registo de clientes. Uma vez que ambas as operações são feitas em memória, implementar um *reader-writer lock* poderia piorar o desempenho. Contudo, numa situação em que, por exemplo, fosse necessário aceder a uma base de dados para autenticar um cliente, e houvesse um número muito maior de *logins* do que de registos, seria de todo vantajoso a implementação do tipo de *lock* referido.

3.2 SIGN

Para efetuar o registo no servidor, o cliente deve enviar uma mensagem com o seguinte formato `SIGN;username;password`. O registo é efetuado através do método `registarCliente`, no qual após se obter o *lock* do Map dos clientes se verifica se o *username* inserido já não se encontra registado. Se este ainda não existir no Map, é então efetuada uma escrita sobre este, adicionando o novo utilizador.

3.3 LOGOUT

O envio deste comando resulta no término da conexão do cliente com o servidor. O servidor responde com a mensagem "END", perante a qual o `Reader` do cliente altera o valor da variável `logout` do objeto do tipo `Log`, o que faz com que o `Drawer` feche o `socket` e a conexão termine.

3.4 MONEY

O *output* deste comando corresponde ao resultado da execução do método `getDivida` no objeto representativo do cliente que se encontra a usar a aplicação. Primeiro é obtido o objeto referido, executando um bloco`synchronized` sobre o Map do clientes. A este objeto é então enviado o método, o qual começa por obter o *lock* explícito presente na classe `Utilizador` e, por fim, retorna o valor da dívida do cliente.

3.5 OSERVER

Esta operação é responsável por indicar ao utilizador quais os servidores que este possui no momento. Para tal, o cliente de enviar ao servidor a mensagem `OSERVER`. O servidor irá, após a leitura da mensagem, utilizar o método `getUser` que lhe permite, com um bloco *synchronized*, aceder ao Map dos clientes e obter o utilizador que enviou a mensagem. De seguida, executando o método `getOwnedServers` do utilizador em questão, indica os servidores que este possui.

A integridade da lista de servidores possuídos é garantida graças à utilização do *lock* do Utilizador, que deve ser adquirido no método `getOwnedServers`, antes de aceder à lista dos servidores em posse do cliente. Desta forma, garante-se que a lista não é alterada enquanto o servidor a estiver consultar.

3.6 FREE

Quando o servidor recebe a mensagem `FREE`, é acionado o método `commandFServer` com a mensagem passada como argumento na sua totalidade, para ser processada pelo método referido. Este, por sua vez, invoca `libertarServidor(id)`, sendo "id"o identificador do servidor, que tinha sido passado na primeira mensagem referida. Ora, é então obtido o `Utilizador` em questão com o `getUser` referido anteriormente e, de seguida, verifica-se se o servidor "id"pertence à lista de servidores do *user* em questão. Para controlo de concorrência, é feito um *lock* ao objeto do `Utilizador`, graças ao `Reentrant Lock` que faz parte desta classe, no momento em que se pretende aceder à lista de servidores possuídos deste. Caso não seja dono do servidor referido, é lançada uma mensagem `SERVER NOT OWNED`. Caso contrário, obtém-se o `ServerTypeManager` relativo ao tipo do servidor "id"(recorrendo a `getServerSMT`). Os passos seguintes são calcular o preço a pagar pela reserva do servidor (libertando-o), remover o servidor da lista de servidores possuídos pelo utilizador e atualizar a dívida do utilizador. No que toca ao primeiro passo indicado, a exclusividade de acesso ao objeto do `ServerTypeManager` é garantida pela ativação do *Reentrant Lock* declarado na classe, garantindo assim que apenas uma *thread* altera o estado dos servidores desse tipo. Esta ativação é feita no método `libertar(id)`, que permite obter o servidor em questão através do id. De seguida, no mesmo método, o *status* do servidor, cuja exclusividade é também garantida por um *lock* da classe `Servidor`, é obtido. Caso este seja um servidor reservado por licitação, o número dos servidores em leilão é diminuído numa unidade. A libertação do servidor em si é feita pelo método `freeServer`, que, recorrendo novamente à ativação do mesmo *lock* (para garantir que nenhuma outra *thread* o modifique ou aceda a esta a meio da operação de libertação), altera o seu estado para "0", removendo a indicação do dono atual e calculando o preço a pagar pela reserva efetuada. De seguida, ainda no bloco de exclusividade de `libertar`, é incrementado o número de servidores livres e adicionado o servidor libertado à lista de livres. Uma fase importante deste método é o sinal que é lançado para as restantes *threads* adormecidas que estejam à espera de uma reserva (caso existam), indicando que um servidor se encontra livre. As operações de remover o servidor e atualizar a dívida do utilizador utilizam também o *lock* da classe `Utilizador` referido anteriormente. Por fim, é lançada uma mensagem `SERVER PAYMENT` para o cliente, com o valor da reserva do servidor que acabou de libertar.

3.7 RENT

A mensagem `RENT` enviada para o servidor é acompanhada pelo preço pelo qual pretende obter o servidor e pelo tipo deste. Em primeiro lugar, é in-

vocado o método `commandRServer`, que utiliza o `adquirServidor`, passando como argumentos o preço e o tipo do servidor indicados. Fazendo novamente uso do `getUser`, é obtido o utilizador em questão. É também obtido da lista de `ServerTypeManager` aquele que diz respeito ao tipo dado. Assim, partindo destes dois objetos, é utilizado o método `adquirir(preco,user)`, que indicará o identificador do servidor alugado. No final do processo, é lançada uma mensagem `SERVER ACQUIRED`, com o resultado de `adquirir`. Entrando em maior detalhe, o método `adquirir` do `ServerTypeManager` faz uso do *lock* da classe. Utiliza-o para mais nenhuma *thread* aceder ao objeto e não criar problemas de concorrência, uma vez que serão alterados dados da classe, tais como o número de servidores livres e a lista destes. Ora, o método começa, após a abertura do bloco de exclusividade, por obter o número de servidores livres e o número de servidores em leilão. Caso ambos sejam nulos, a *thread* adormece, esperando que haja um servidor disponível para reservar. Caso haja um servidor livre, o número destes é reduzido e é obtido o servidor do `Map` de *servers* que corresponde àquele que está livre há mais tempo (pela `Queue` existente na classe). Tirando partido da função `buyServer` do servidor obtido, que tira partido do *lock* do objeto pelas mesmas razões de integridade, o seu estado muda para reservado por compra direta, atualizando também a informação relativa ao dono atual e à data de reserva. Caso haja apenas servidores em leilão, é feita uma procura para obter o primeiro servidor cujo status seja "2". Assim, é invocado o método `buyBiddenServer`, que, novamente utilizando o *lock* do servidor, atualiza o seu status para "1". Além disso, o servidor é removida da lista de servidores do utilizador que era dono do servidor. Tal é feito através de um bloco de exclusividade no utilizador para que mais nenhuma *thread* aceda à lista de servidores deste. De seguida, é calculada e atualizada a dívida do antigo dono. A data de aquisição é alterada para a data em que o servidor foi libertado (involuntariamente) pelo outro utilizador e o dono é também atualizado. Um passo importante é o facto de ser ativado o sinal `bid_steal`, que permite avisar o antigo dono do servidor em questão que este lhe foi "roubado".

3.8 BID

A receção de uma mensagem `BID` é acompanhada pelo tipo de servidor que o cliente pretende alugar e o valor da sua licitação. É no método `commandBServer` que o *parsing* da mensagem recebida é feito, obtendo com isso o utilizador (novamente recorrendo `getUser`) e o `ServerTypeManager`, e utilizando-os como argumento do método `licitar` deste último, que é bastante complexo. Isto porque inicia com o *lock*, para garantir a exclusividade de acesso ao objeto como foi referido anteriormente, e cria uma nova proposta (*Licitacao*), com o nome do utilizador que a fez e o preço que licitou, adicionando-o à lista de propostas existentes. Caso não haja servidores livres, ou caso a primeira proposta no topo da lista (ou seja, a mais alta) não seja do utilizador em questão, a *thread* é adormecida. Estando tudo nos conformes, o número de leilões é aumentado e o número de servidores livres é diminuído, removendo, então, a proposta da respetiva fila de espera. É então obtido o primeiro servidor da lista dos livres, no qual é invocado o método `bidServer`. Este, por sua vez, é semelhante ao

já referido e descrito `buyServer`, modificando apenas o status do server para "2"em vez de "1". Também aqui é utilizado um bloco *lock*, de forma a que o Servidor não seja alterado por mais nenhuma *thread*. O próximo passo é adicionar o servidor à lista de servidores do utilizador, utilizando também aqui uma zona de exclusividade mútua. É então ativado o sinal `livre`, indicando às *threads* adormecidas que estão à espera para comprar diretamente um servidor que há um servidor alugado em leilão. Depois disso, é corrida uma nova *thread* de *Bidcheck*, que verificará e alertará o utilizador se o server que adquiriu foi "roubado"por outro numa compra direta.

3.9 MSGS

Operação que permite obter mensagens relativas a eventuais desafetações de servidores comprados a leilão no período em que o utilizador se encontrava desconectado.

4 Conclusão

O sistema que desenvolvemos para este projeto foi, na opinião do grupo, bastante bem conseguido, uma vez que, além de cumprir os requisitos para o serviço de alocação de servidores na nuvem (descritos no enunciado e na introdução deste relatório), respeita as exigências estabelecidas no âmbito dos sistemas distribuídos. Isto porque, apesar de haver concorrência na sua execução, as regiões críticas são acedidas por apenas uma *thread*, não prejudicando portanto a integridade e consistência dos dados, devido ao facto de não haver sobreposição de alterações de um mesmo objeto por duas *threads* diferentes, por exemplo.

Em suma, o projeto como esperávamos, funcionando da maneira que era pedida e cumprindo o que era requerido do sistema, tendo sempre em conta os conceitos e procedimentos adquiridos nas aulas de Sistemas Distribuídos.