

UNIVERSIDADE DO MINHO

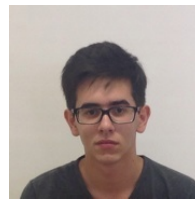
Relatório de Projeto SD

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA

SISTEMAS DISTRIBUIDOS

(1º SEMESTRE - 2018/2019)

a73974 Carlos Daniel Leitão da Silva Vieira



a73855 José Lopes Ramos



a73182 Pedro José Marques Terra Sousa



a74264 Rafael Machado da Silva



6 de Janeiro de 2019

Conteúdo

1	Introdução	2
2	Desenvolvimento	3
2.1	MakeFile	3
2.2	ServerCloud	3
2.3	Client	4
2.4	ClientHandler	4
2.5	Server	4
2.6	Reserve	4
2.7	User	4
2.8	Menu	4
2.9	ClientReceive	4
2.10	Tuple	4
3	Trabalho final	5
3.1	Funcionalidade 1	5
3.2	Funcionalidade 2	5
3.3	Funcionalidade 3	6
3.4	Funcionalidade 4	6
3.5	Funcionalidade 5	6
4	Conclusão	7

1 Introdução

Neste trabalho implementamos uma aplicação distribuída que permite fazer a gestão de um serviço de leilões.

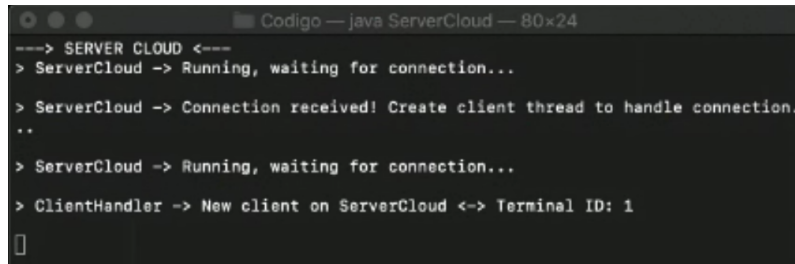
Esta aplicação é acedida por clientes que podem reservar e usar servidores virtuais para processamento e armazenamento de dados. Estas plataformas disponibilizam vários tipos de servidores, cada um com uma especificação e preço por utilização específica.

A reserva de um servidor pode ser feita a pedido ou em leilão. A pedido, um servidor fica atribuído até ser libertado pelo utilizador, sendo cobrado o seu preço nominal horário correspondente ao tempo utilizado. No caso da reserva em leilão, um utilizador propõe o preço que está disposto a pagar pela reserva do servidor de determinado tipo. Só lhe é atribuído um servidor quando o preço horário proposto for o maior entre os licitantes para esse tipo de servidor. Além disso, a reserva de um servidor em leilão pode ser cancelada pela nuvem de forma a satisfazer uma reserva a pedido quando não existam outros servidores disponíveis do tipo pretendido.

2 Desenvolvimento

Para iniciar a nossa aplicação devemos abrir o terminal e fazer *make*. De seguida tem que inserir a classe *java ServerCloud* e correr o programa.

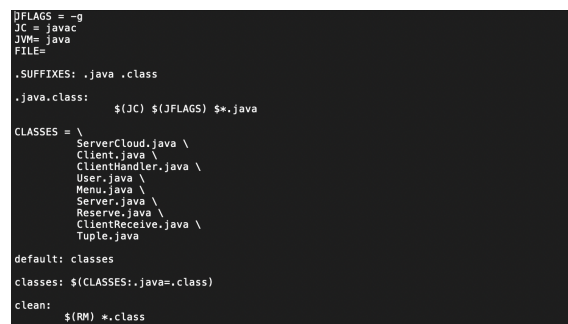
Para entrar no programa como Cliente devemos abrir outra janela de terminal escrevendo *java Client* e inserir os dados de cliente para autenticar e iniciar a aplicação aonde nos aparecerá o menu de utilizador.



```
====> SERVER CLOUD <====  
> ServerCloud -> Running, waiting for connection...  
  
> ServerCloud -> Connection received! Create client thread to handle connection..  
..  
  
> ServerCloud -> Running, waiting for connection...  
  
> ClientHandler -> New client on ServerCloud <-> Terminal ID: 1  
  
[]
```

Figura 1

2.1 MakeFile



```
JFLAGS = -g  
JC = javac  
JWM = java  
FILE =  
  
.SUFFIXES: .java .class  
.java.class: $(JC) $(JFLAGS) $*.java  
  
CLASSES = \  
    ServerCloud.java \  
    Client.java \  
    ClientHandler.java \  
    User.java \  
    Menu.java \  
    Server.java \  
    Reserve.java \  
    ClientReceive.java \  
    Tuple.java  
  
default: classes  
classes: $(CLASSES:.java=.class)  
clean: $(RM) *.class
```

Figura 2: código makefile

2.2 ServerCloud

Esta classe trata o servidor que vai ser sempre iniciado quando utilizamos a aplicação.

Este tem um map de todos os servidores, um map de todos os users e um map das reservas existentes, guardando assim todos os dados necessários para a inicialização e funcionamento do programa.

Para além disso também define uma variável **porto** e um socket **Serversocket** que vão ser usado ao invocarmos o servidor. Este inicia o **socket** neste porto e fica á espera de interações da parte dos clientes. Quando um cliente aparece, o **ServerCloud** cria uma thread para este cliente usando a classe **ClienteHandler**.

2.3 Client

Quando um cliente inicia a aplicação através de "java Client" o **ServerCloud** vai criar uma thread à qual o Client se vai ligar criando uma socket que o liga ao **ServerCloud** através do **porto** definido (clientsocket).

Na classe Cliente, invocamos o método **startClient**. Com esse método conseguimos abrir o canal de escrita para o servidor e o de leitura da thread que o servidor criou para tratar da conexão e no fim fecha os canais de conexão ao servidor. No meio disso irá ser invocado o método **startMenu** dando início à interface gráfica com o cliente.

Consoante a escolha do Utilizador, esta classe envia um **PrintWriter out** para a **ClienteHandler** de modo a poder responder com aquilo que o Utilizador pretende.

2.4 ClientHandler

Quando iniciamos um thread em Severcloud estamos a criar um **ClientHandler** que servirá de gestor de comunicação entre o Client e o Servercloud nas atividades que envolvem troca e tratamento de informação que cada um envia ao outro.

Assim o ClientHandler vai criar um *in* para receber os dados do cliente de **Client** e um *out* para enviar para o cliente e depois ligamos a sua socket à socket do cliente.

Esta classe vai invocar o método de **handler** e consoante o *in* recebido do **Client** vai devolver um *out* diferente consoante os métodos invocados anteriormente para cada *in*.

Em algumas das opções, as mensagens recebidas são cortadas de modo a ter as palavras necessárias para usar os métodos necessários.

2.5 Server

Esta classe trata um server e guarda a informação e métodos para trabalhar no mesmo. Dentro dessa informação tem um **HashMap queue_users** que guarda para um email o valor dado por esse server.

2.6 Reserve

A classe **Reserve** compõe um dos maps do Servercloud e serve para guardar os dados de reservas que estejam ativas, nomeadamente o cliente, o valor pago, e qual o server que foi reservado.

2.7 User

A classe **User** é uma simples classe de modo a poder identificar o Utilizador com todos os *gets* e *sets* necessários para ela.

2.8 Menu

Para maximizar a compreensão do código mesmo para um leigo e para facilitar o mesmo me caso de mudança, criamos uma classe a que demos o nome de **Menu** em que criamos todos os Menus que vão ser precisos.

2.9 ClientReceive

Criamos esta classe para poder responder a uma parte da funcionalidade de modo a que o Utilizador que perdesse o leilão, recebesse uma mensagem.

2.10 Tuple

Criamos esta classe auxiliar para nos ajudar a contar o número de servidores de um certo tipo livres. Esta classe contém os **sets** e **gets** necessários.

3 Trabalho final

Nesta secção iremos falar de como conseguimos resolver todas as funcionalidades que nos eram propostas.

Antes de falar de como fizemos esta funcionalidade é de realçar que iniciamos o **ServerCloud** e que nesta iremos iniciar uma thread com a classe **ClientHandler** que possui um método *run* em que esta, através do método *handler*, está à espera de mensagens da classe **Client**, isto é, esta Thread irá fazer o tratamento da conexão com o Cliente.

3.1 Funcionalidade 1

Ao iniciar a classe **Client** esta invoca o método **startMenu**. Neste método e após invocar a interface precisa para este menu, este está à espera do que o Utilizador escrever.

Consoante a resposta do Utilizador, irá entrar em opções diferentes em que cada um das opções irá enviar uma mensagem **PrintWriter out** diferente. Após isso ele está à espera de receber algo, que de seguida irá retornar.

Na classe **ClientHandler**, no método *handler*, ao receber a mensagem do **Client** irá verificar qual é a opção e invocar o método.

Visto que a mensagem enviada pelo **Client** tem palavras desnecessárias é feito um *parser* dessa mensagem de modo a ficar com o que é preciso, email e/ou password.

Em caso de login este irá verificar se o email enviado pelo **Client** está no **HashMap users** e envia a resposta. Em caso de registo este irá acrescentar no *HashMap users* através do *put* enviando uma mensagem de confirmação.

3.2 Funcionalidade 2

Todos os servidores em que a o valor pago pelo Utilizador é igual ao valor do servidor entra automaticamente em leilão.

Querendo fazer uma reserva de um servidor, ao Utilizador irão ser apresentados vários servidores (devido a uma mensagem que é enviada para o **ClientHandler** e que retorna todos os servidores presentes no **HashMap servers** que esta última classe tem), no qual tem que escrever um valor posteriormente.

Com esses dados o **Client** manda-os para o **ClientHandler** e inicia-se o método **checkOffer**. Este método no início irá criar 2 arrays; 1 para os servidores livres e outros para os servidores em leilão, sendo que cada um dos arrays se refere ao Servidor que o Utilizador quer.

Quando o 1º array não é vazio, isto é, o servidor que o Utilizador quer encontra-se livre, este poderá fazer a compra caso insira um valor igual ao preço do Servidor.

Se todos os servidores tiverem em leilão e o Utilizador oferecer o mesmo valor que o preço desse mesmo Servidor, o servidor fica para o Utilizador, retornando 1, que indica que ficou a partir daquele momento ocupado.

Através do número retornado através da função **checkOffer**, irão ser enviadas mensagens diferentes para o **Client** que por sua vez vai retornar para o Utilizador. Antes disso, é atualizado o **HashMap reserves**.

3.3 Funcionalidade 3

Todos os servidores em que o valor pago pelo Utilizador é menor que o valor do servidor entra automaticamente em leilão. Foi por este ponto de vista que realizamos esta funcionalidade.

Ao escolher no menu a opção *CATALOGUE SERVERS* irão ser devolvidos todos os servidores. Através disso, o Utilizador já tem a informação de quanto é que cada servidor vale, portanto ao começar a fazer a compra, se inserir um valor inferior a esse entra automaticamente num leilão. Ao enviar a resposta para o **ClientHandler** este irá invocar o método **checkOffer**. Este método, tal como está dito na funcionalidade anterior, irá fazer a reserva de todos os servidores, quer seja em leilão ou não. Neste caso, caso o preço dado seja inferior ao preço que vale o Servidor, este irá entrar em leilão.

Aí, haverá um **Queue** de tamanho 1 com o Utilizador que deu o valor mais alto. Logo caso o preço seja maior ou igual ao preço dado pelo Utilizador que está na Queue, haverá uma troca de Utilizadores, retornando um inteiro.

Através desse inteiro, iremos ou não adicionar no **HashMap** reservas enviando uma mensagem para o **Client** para ele retornar.

3.4 Funcionalidade 4

Ao escolher no menu a opção *VIEW RESERVES* irão ser redirecionados para um sub menu (menu view reserves) onde ao escolher a primeira opção *FINISH* será pedido o id do servidor que deseja terminar. Após inserir e pressionar enter a aplicação termina o servidor e dá como (finish).

Enviando a informação para **Reserve** e chama o método **finishReserve()** que atua no objeto e atualiza a `date_final` com a data em que terminou, atualizando também o seu state e calcula o `priceFinal`.

3.5 Funcionalidade 5

Ao escolher no menu a opção *USER INFORMATION* irão ser redirecionados para um sub menu (menu User Information) onde ao escolher a terceira opção *CHECKDEBT* será retornado o valor que tem em dívida.

Caso não tenha nenhum valor em dívida irá retornar ao menu. Caso existe um valor a pagar, será dada a hipótese de pagar esse mesmo e quanto deseja pagar.

Dando a resposta "Y" é enviada a resposta para o **ClientHandler** este irá invocar o método **PAY DEBT**, onde este irá consultar o saldo do cliente, caso este não tenha suficiente é lançada a hipótese de adicionar o dinheiro necessário para pagar, se assim o fizer vai ser consultado o valor que está no **HashMap** das reservas e esse valor é atualizado.

4 Conclusão

O desenvolvimento deste trabalho pratico levou-nos a trabalhar na aplicação prática da teoria, conceitos e conhecimentos lecionados na Unidade Curricular sobre *Multithreading* realçando temas no âmbito do controlo de concorrência e também comunicação cliente/servidor.

Acreditamos ter conseguido utilizar os conhecimentos adquiridos nas aulas de modo a criar um programa eficaz capaz de lidar com o fluxo de informação e a utilização simultânea por parte de vários utilizadores, minimizando ao máximo (senão mesmo anulando) os problemas e erros que poderiam existir devido a concorrência, e capaz de lidar com os critérios requeridos no enunciado do trabalho.