Universidade do Minho

Conselho de Cursos de Ciências

Licenciatura em Ciências de Computação

Disciplina de Programação Concorrente

Ano Lectivo de 2015/2016

Gestor de Deslocações

João Vilaça (51159)
Elsa Amorim (70060)

Repositório: https://github.com/jpggvilaca/uber_erlang_java

Introdução

O objectivo do trabalho era implementar uma aplicação distribuída que permitisse a gestão de um serviço de taxi. Para isto teria de ser usado um servidor escrito em Erlang e um cliente escrito em Java. O serviço criado por estes dois componentes deveria ser responsável por tratar do registo e login do utilizador, sendo o utilizador um passageiro ou um condutor.
No caso do condutor, além do registo e login, deve ser possível anunciar disponibilidade para conduzir. No caso do passageiro deve também ser possível solicitar uma viagem indicando o ponto de partida e de chegada, devendo ser posterior confirmada a entrada no veículo caso este não cancele a viagem.

Obstáculos

Da parte do cliente, os obstáculos a serem ultrapassados eram o ter que garantir que:

1) A comunicação por sockets com um receptor e um transmissor através de threads.
2) No mesmo utilizador, o receptor não interfere com o transmissor.
3) Entre vários utilizadores as mensagens não são trocadas ou confundidas.
4) Como a transmissão/recepção pode não ser síncrona, garantir que estou a receber a resposta ao que transmiti e não a outra mensagem qualquer.

Da parte do servidor os obstáculos a serem ultrapassados eram o ter que garantir que consiga:

1) Gerir ligações ao cliente via socket
2) Validar registos e logins
3) Guardar listas de utilizadores, diferenciados (passageiro e condutor)
4) Estabelecer uma comunicação entre processos assim como entre eles e o socket
5) Guardar estados e gerir a lógica de alteração dos mesmos, consoante envio/recepção de mensagens inter-processo ou do socket

Organização

O código do projeto, em conjunto com os ficheiros de testes, api, Emakefile, sources de java, readme e relatório podem ser encontrados online no meu repositório do github https://github.com/jpggvilaca/uber_erlang_java . Os ficheiros da api, relatório e readme encontram-se na pasta raíz do projeto. Os ficheiros do cliente encontram-se na pasta “client” e os ficheiros do servidor na pasta “server”. A api contém as mensagens importantes passadas entre cliente e servidor como também mensagens passadas entre processos no servidor. O ficheiro ‘sources.txt’ contem todos os ficheiros java para serem compilados correndo apenas o comando ‘javac @sources.txt’. O ficheiro Emakefile contém todos os módulos a serem compilados pelo erlang sendo só preciso correr ‘erl -make’ no terminal.

Arquitetura e Implementação (Cliente)

Classe Transmitter: Esta classe é responsável por ler uma linha (string) do socket e usando uma BlockingQueue, adiciona essa linha à queue para ser depois consumida pelo cliente.

```
public class Transmitter extends Thread {
  (...)
  public void run() {
    try {
      String response = null;
      BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
      while ((response = reader.readLine()) != null) {
        // Add the response from the server to the queue
        queue.add(response);
      }

      socket.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

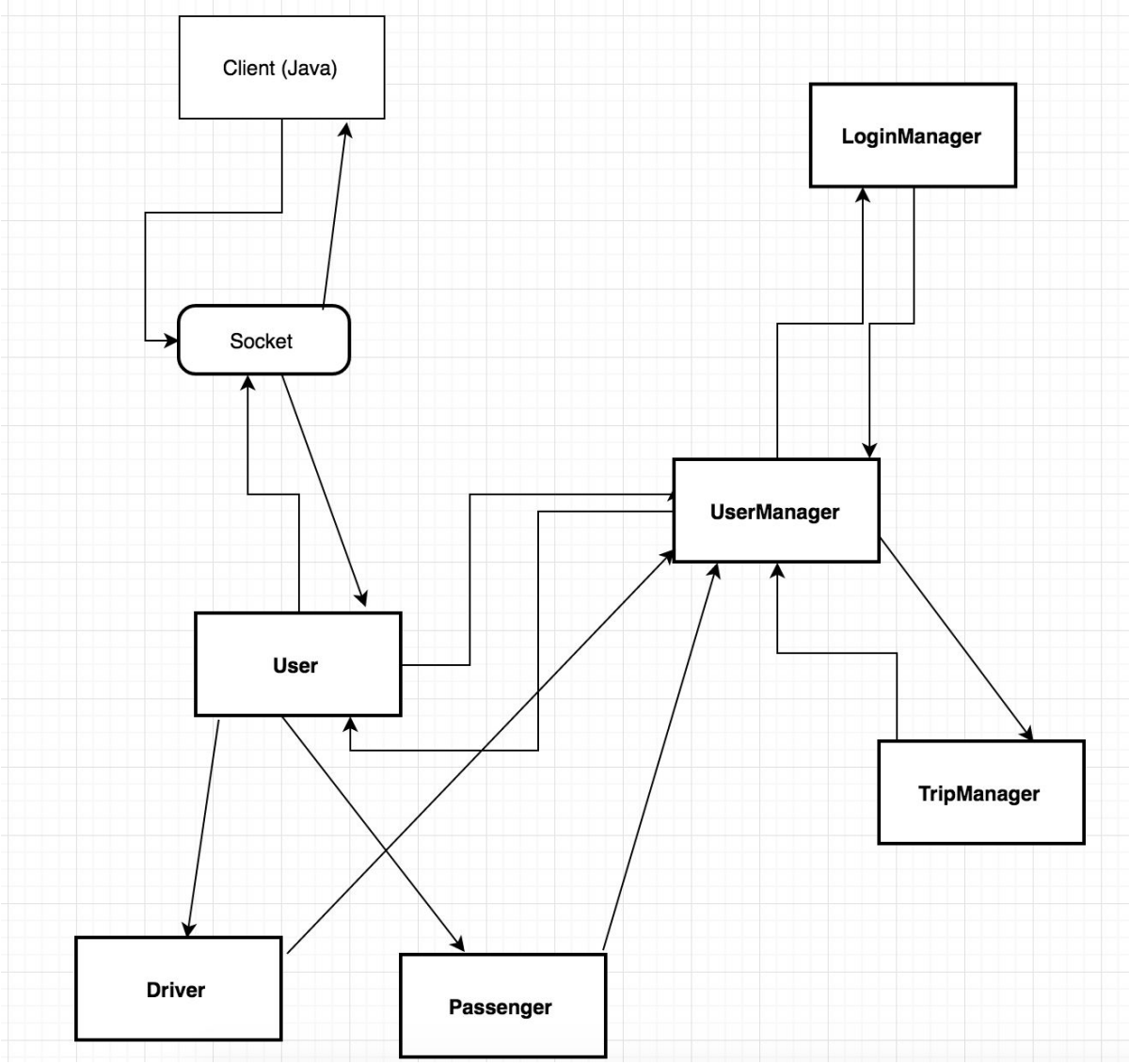
Classe Client: Esta classe é a classe principal que estabelece a ligação entre o System.in (cliente) e o socket.

```
(...)
BlockingQueue<String> messages = new LinkedBlockingQueue<>(); // Queue for socket messages
PrintWriter printer = new PrintWriter(socket.getOutputStream(), true); // Write to socket
BufferedReader reader = new BufferedReader(new InputStreamReader(System.in)); // Read from
System.in
Transmitter trans = new Transmitter(socket, messages); // Instance of transmitter
trans.start(); // Starts the thread that reads from the socket
(...)

(...)
while(true) {
  String readerInput = reader.readLine(); // Read from console
  printer.print(readerInput.trim()); // Sends it to the socket
  printer.flush();

  socketMessage = messages.take(); // Get the server response
  command = readerInput.substring(0, 3); // Get the parsed string from user input
  aux = readerInput.substring(readerInput.length() - 1, readerInput.length());

  (...)
```

O flow de dados então é o seguinte:

1) O cliente escreve a string no terminal (e.g. “1:reg:joao:vilaca:1:seat:ibiza”).
2) A instância do Transmitter (trans) envia a string para o socket.
3) A variável socketMessage retira da queue a mensagem recebida do socket (e.g. “register_ok”).
4) De acordo com a resposta do servidor, alteramos entre elementos do switch que, com a ajuda das variáveis command e aux, fazem parse do que o utilizador insere no terminal.
5) É mantido também um estado no cliente para saber se o utilizador é ou não condutor, se está ou não autenticado, e se já começou/acabou a viagem, para gerir a lógica de mensagens que é apresentada ao utilizador no terminal.

Arquitetura e Implementação (Servidor)



Acima encontra-se um pequeno esquema de como foi estruturado o servidor a nível de processos.
O tipo de mensagens que podem ser enviadas/recibas encontram-se no ficheiro ‘api.txt’ por isso a seguir passo a descrever o ‘flow’ de mensagens que pode acontecer para melhor explicar a imagem.

1) User recebe uma mensagem do socket e reencaminha-a para o Usermanager.
2) Consoante a mensagem, o UserManager delega a lógica sobre a mensagem para o LoginManager ou TripManager. Este guarda também uma lista de Users activos.
3) Se a mensagem for para LoginManager, a lógica de registo/login é feita e é enviada a mensagem correspondente de volta para o UserManager.
4) O UserManager ao receber essa mensagem reencaminha a mesma para o User.
5) O User ao receber essa mensagem envia a resposta para o socket através do comando gen_tcp:send(Socket, “mensagem”).
6) Se a mensagem for para TripManager, a lógica de viagens é feita e é enviada uma resposta para o Pid do utilizador.
7) O User ao receber essa mensagem, caso seja condutor, torna-se um processo “driver”, caso seja passageiro, torna-se um processo “passenger”, para facilitar o tratamento de mensagens recebidas por parte do TripManager

Flow do condutor:
1) Efectuar registo.
2) Efectuar login.
3) Apresentar disponibilidade para conduzir indicando as coordenadas do seu local.
4) Recebe mensagem com um pedido de viagem.
5) Recebe mensagem quando viagem chegar ao fim.

Flow do passageiro:
1) Efectuar registo.
2) Efectuar login.
3) Requisitar viagem indicando o ponto de partida e chegada.
4) Espera por condutores.
5) Quando existirem condutores disponíveis, é escolhido o mais próximo e é recebida a informação desse condutor.
6) Cancelar viagem antes do condutor chegar tendo que pagar metade do custo da viagem caso cancele passado 1 minuto do pedido de viagem.
7) Quando o condutor chegar, pode cancelar ou entrar no carro.
8) Recebe mensagem quando a viagem chegar ao fim.

Conclusão

Este trabalho permitiu-nos perceber mais sobre concorrência entre threads e comunicação via sockets TCP, tanto da parte de Java como de Erlang.
A nível de Erlang, permitiu-nos aprender mais sobre a linguagem e a arquitetura da mesma, como é que funciona a criação de processos, envio de mensagens inter-processos e entre processos e o socket.
Erlang é uma linguagem poderosa e extremamente rápida. Além de ser assíncrona, o envio de mensagens não assume que estas são enviadas com sucesso ou se o processo receptor as recebeu.
A comunicação via socket e gestão das mensagens é feita com relativa facilidade.
A nível do cliente, Java, aprendeu-se a lidar com o assincronismo por parte do erlang através de threads e locks, neste caso usando uma BlockingQueue para lidar com envio/recepção de mensagens via socket.